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,
Nanjing 210023, China

Abstract

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 [**?** **?** **?** **?** **?** **?** **?**] 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 makes GNNs achieve good accuracy in not only graph analytical tasks [**?** **?** **?**] (like node classification and link prediction) but also computer vision tasks (like human-object interaction [**?**], human parsing [**?**], and video object segmentation [**?**]).

To train GNNs easily, a series of GNN libraries/systems [**?** **?** **?** **?** **?**] are proposed. PyTorch Geometric (PyG) [**?**], NeuGraph [**?**], PGL [**?**] and Deep Graph Library (DGL) [**?**] 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 [**?**] [**?**]) and other implementation techniques (like 2D graph partitioning [**?**]) 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. [**?**] and Zhang et al. [**?**] 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

*Corresponding authors with equal contribution
Email address: {wangzhaokang, wangyp}@smail.nju.edu.cn, {cfyuan, gurong, yhuang}@nju.edu.cn (Zhaokang Wang, Yunpan Wang, Chunfeng Yuan, Rong Gu* , Yihua Huang*)

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 $\times$ $\{$vertex, edge$\}$ calculation). We choose GCN [**?**], GGNN [**?**], GAT [**?**] and GaAN [**?**] 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.

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 *undirected* 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. We use v_x $(0 \leq x < |\mathcal{V}|)$ to denote a vertex and $e_{x,y} = (v_x, v_y)$ to denote the edge pointing from v_x to v_y. The adjacency set of v_x is $\mathcal{N}(v_x) = \{v|(v_x, v) \in \mathcal{E}\}$. We denote a *vector* with a bold lower case letter like $\boldsymbol{h}$ and a *matri*x with a bold upper case letter like $\boldsymbol{W}$. Table 1 summarizes the common symbols used throughout this work. We use blue characters $\boldsymbol{w}$/$\boldsymbol{W}$ to denote weight vectors/matrices that are model parameters to train in a GNN.

Table 1: Notations

Category	Symbol	Meaning
Graph Structure	$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	The simple undirected input graph with the vertex set $\mathcal{V}$ and the edge set $\mathcal{E}$.
	v_x	The x-th vertex of the input graph.
	$e_{x,y}$	The edge pointing from v_x to v_y of the input graph.
	$\mathcal{N}(v_x)$	The adjacency set of v_x in the input graph.
	$\bar{d}$	The average degree of the input graph.
GNN Definition	L	The number of GNN layers.
	K	The number of heads in a GNN layer.
	ϕ^l	The messaging function of the GNN layer l.
	Σ^l	The aggregation function of the GNN layer l.
	γ^l	The vertex updating function of the GNN layer l.
	$\phi^{l,i}$ / $\Sigma^{l,i}$ / $\gamma^{l,i}$	The messaging/aggregation/updating function of the i-th sub-layer of the GNN layer l.
	$\boldsymbol{W}^l, \boldsymbol{W}^{(k)}/\boldsymbol{b}, \boldsymbol{a}$	The matrices/vectors represented by the blue characters are the weight matrices/vectors that need to be learned in the GNN.
Vector	$\boldsymbol{v}_x$	The feature vector of the vertex v_x.
	$\boldsymbol{e}_{x,y}$	The feature vector of the edge $e_{x,y}$.
	$\boldsymbol{h}_x^l$	The input hidden vector of the graph neuron corresponding to v_x in the GNN layer l.
	$\boldsymbol{h}_x^{l+1}$	The output hidden vector of the graph neuron corresponding to v_x in the GNN layer l.
	$\boldsymbol{m}_{x,y}^l$	The message vector of the edge $e_{x,y}$ outputted by ϕ^l of the GNN layer l.
	$\boldsymbol{s}_x^l$	The aggregated vector of the vertex v_x outputted by Σ^l of the GNN layer l.
	$\boldsymbol{h}_x^{l,i}$ / $\boldsymbol{m}_{x,y}^{l,i}$ / $\boldsymbol{s}_x^{l,i}$	The hidden/message/aggregated vector of the vertex v_x outputted by $\gamma^{l,i}$/$\phi^{l,i}$/$\Sigma^{l,i}$ of the i-th sub-layer of the GNN layer l.
	d_{in}^l, d_{out}^l	The dimension of the input/output hidden vectors of the GNN layer l.
	$dim(\boldsymbol{x})$	The dimension of a vector $\boldsymbol{x}$.

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.

In the inpug layer, A GNN receives a graph $\mathcal{G}$ as the input. Every vertex v_x in $\mathcal{G}$ is attached with a feature vector $\boldsymbol{v}_x$ to describe the properties of the vertex. Every edge $e_{x,y}$ of $\mathcal{G}$ may also be attached with a feature vector $\boldsymbol{e}_{x,y}$. The input layer of a GNN receives feature vectors from all vertices and passes the feature vectors to the first GNN layer (i.e. GNN layer 0).

A GNN usually consists of more than one GNN layers. Each GNN layer consists of $|\mathcal{V}|$ graph neurons, where $|\mathcal{V}|$ is the number of vertices in $\mathcal{G}$. Each graph neuron corresponds to a vertex in $\mathcal{G}$. Different GNNs mainly differ in the graph neurons that they use. We elaborate on details of graph neurons later. GNN layers are *sparsely* connected with the input layer and other GNN layers.

- In the first GNN layer (layer 0), a graph neuron collects feature vectors from the input layer. For the graph neuron corresponding to the vertex v_x, it collects the feature vector $\boldsymbol{v}_x$ and the feature vectors $\boldsymbol{v}_y$ of the vertices v_y that are adjacent to v_x (i.e. $v_y \in \mathcal{N}(v_x)$). The graph neuron aggregates input feature vectors, apply non-linear transformation, and outputs a hidden vector $\boldsymbol{h}_x^1$ for v_x. Take the demo GNN in Figure 1(a) as an 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

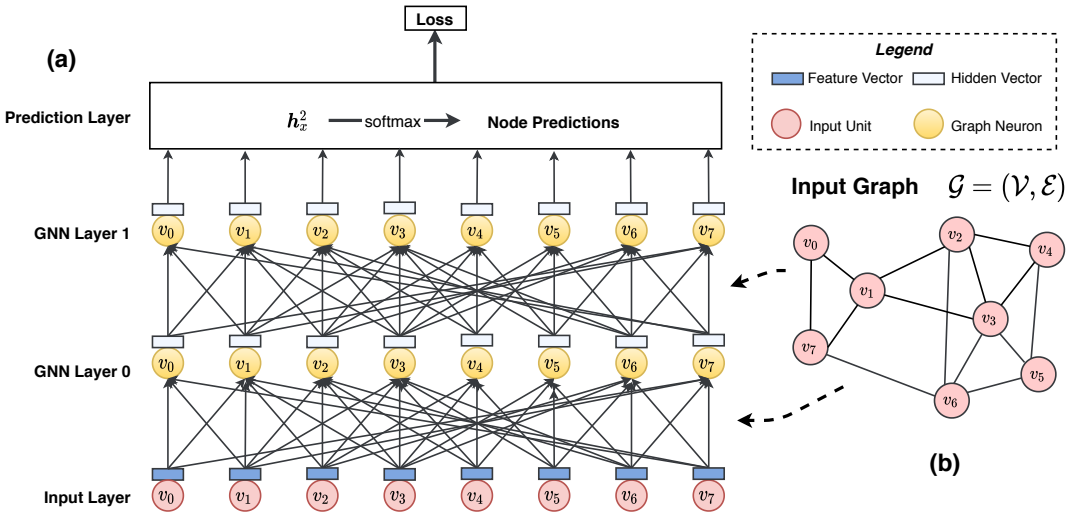


Figure 1: Structure of a typical graph neural network. (a) Demo GNN, (b) Demo graph. The target application is the node classification. The demo GNN has two GNN layers.

layer 0 collects the input feature vectors $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3, \boldsymbol{v}_4, \boldsymbol{v}_5, \boldsymbol{v}_6\}$ from the input layer and outputs $\boldsymbol{h}_3^1$. The connections between the first GNN layer and the input layer are determined by the topology of $\mathcal{G}$. The graph neuron of v_x in the first GNN layer is connected with the input unit of v_y in the input layer only if there is an edge $e_{x,y}$ between v_x and v_y in $\mathcal{G}$. Since most real-world graphs are very *sparse* (i.e. $|\mathcal{E}| \ll |\mathcal{V}|^2$), the connections between the first GNN layer and the input layer are also sparse, different from the traditional neural networks.

- In the next GNN layer (layer 1), the graph neuron corresponding to v_x collects the hidden vector of itself $\boldsymbol{h}_x^1$ and its adjacent vertices ($\boldsymbol{h}_y^1$ with $v_y \in \mathcal{N}(v_x)$) from the *previous* GNN layer. Thus, the connections between the first and the second GNN layers are also determined by the topology of $\mathcal{G}$. Based on the collected hidden vectors, the graph neuron in the layer 1 outputs a new hidden vector $\boldsymbol{h}_x^2$ for v_x.

- A GNN allows stacking more GNN layers to support deeper graph analysis. Assume there are L GNN layers in total. The last GNN layer (layer $L-1$) outputs a hidden vector $\boldsymbol{h}_x^L$ for every vertex v_x. $\boldsymbol{h}_x^L$ is an embedding vector that encodes the knowledge learned from the input layer and all the previous GNN layers. Since $\boldsymbol{h}_x^L$ is affected by v_x and the vertices in the L-hop neighborhood of v_x, analyzing a graph with more GNN layers means analyzing each vertex with a *wider* scope. The hidden vectors $\boldsymbol{h}_x^L$ of the last GNN layer are fed to the prediction layer to generate the output for 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 $\boldsymbol{h}_x^L$ as the input and a vector of probabilities as the output. If the prediction task is the edge prediction, the hidden 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 GNNs, compared with traditional graph-based machine learning methods. The traditional methods need to construct input feature vectors for vertices and edges manually or use embedding methods like DeepWalk [**?**] and node2vec [**?**]. The feature vector generation is independent from the model training. Therefore, generated feature vectors may not be suitable for downstream prediction tasks. In GNNs, gradients are propagated from the prediction layer back to 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, a GNN can automatically extract 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 is a small neural network. For the graph neuron corresponding to v_x at the layer l, it receives the hidden vector of itself $\boldsymbol{h}_x^l$ and the hidden vectors of its adjacent vertices $\boldsymbol{h}_y^l$ with $v_y \in \mathcal{N}(v_x)$ from the previous GNN layer[1]. The graph neuron of v_x aggregates the received hidden vectors, applies non-linear transformations, and outputs a new hidden vector $\boldsymbol{h}_x^{l+1}$.

We follow the message-passing model [**?**] to formally define a graph neuron. The message-passing model is widely used in the cutting-edge GNN libraries like PyG [**?**] and DGL [**?**]. Figure 2 shows the internal structure of a graph neuron in the message-passing model. A graph neuron at the layer l are made of three *differentiable* functions: the messaging function ϕ^l, the aggregation function Σ^l, and the updating function γ^l.

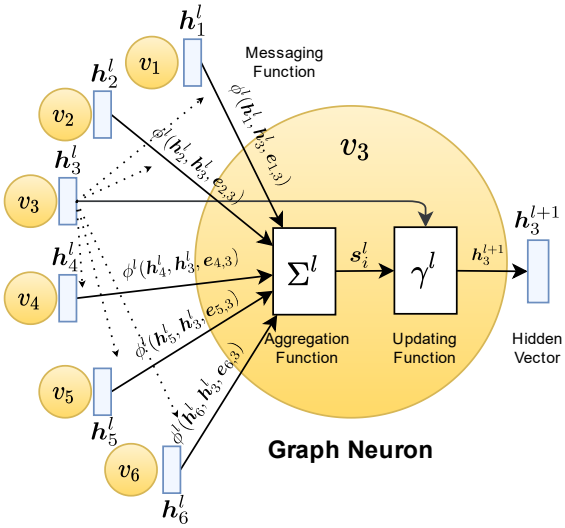


Figure 2: Graph neuron of v_3 in the GNN layer l with the demo graph $\mathcal{G}$ in Figure 1b. $\phi^l/\Sigma^l/\gamma^l$ are the messaging/aggregation/updating functions in the message-passing model, respecitvely.

The graph neuron of v_x calculates the output hidden vector $\boldsymbol{h}_x^{l+1}$ with the three functions in three steps:

1. For every adjacent edge (v_y, v_x) of v_x ($v_y \in \mathcal{N}(v_x)$), the messaging function ϕ^l receives the output hidden vectors $\boldsymbol{h}_x^l$ and $\boldsymbol{h}_y^l$ from the previous GNN layer and the edge feature vector $\boldsymbol{e}_{y,x}$ (and other feature vectors associated with v_x/v_y if necessary) as the input. ϕ^l emits a *message vector* $\boldsymbol{m}_{y,x}^l$ for every edge (v_y, v_x) at the layer l, i.e. $\boldsymbol{m}_{y,x}^l = \phi^l(\boldsymbol{h}_y^l, \boldsymbol{h}_x^l, \boldsymbol{e}_{y,x}, \dots)$;
2. The aggregation function Σ^l then aggregates the message vectors $\boldsymbol{m}_{y,x}^l$ of the adjacent edges ($v_y \in \mathcal{N}(v_x)$) to produce an *aggregated vector* $\boldsymbol{s}_x^l$, i.e. $\boldsymbol{s}_x^l = \sum_{v_y \in \mathcal{N}(v_x)}^l \boldsymbol{m}_{y,x}^l$;
3. The updating function γ^l calculates the hidden vector of this layer $\boldsymbol{h}_x^{l+1}$ based on the hidden vector from the previous layer $\boldsymbol{h}_x^l$ and the aggregated vector $\boldsymbol{s}_x^l$ (and other feature vectors associated with v_x if necessary), i.e. $\boldsymbol{h}_x^{l+1} = \gamma^l(\boldsymbol{h}_x^l, \boldsymbol{s}_x^l, \dots)$.

Briefly, the behaviour of a graph neuron under the message-passing model can be defined as

$$\boldsymbol{h}_x^{l+1} = \gamma^l(\boldsymbol{h}_x^l, \Sigma_{v_y \in \boldsymbol{N}(v_x)}^l \phi^l(\boldsymbol{h}_y^l, \boldsymbol{h}_x^l, \boldsymbol{e}_{y,x}, \dots), \dots). \tag{1}$$

The end-to-end training requires ϕ^l and γ^l (like multi-layer perceptrons and GRUs) 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 *edge calculation* functions, since they are conducted over every edge in $\mathcal{G}$. We regard γ as the *vertex calculation* function, as

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

it is conducted over every vertex in $\mathcal{G}$. Table 2 and Table 3 list the edge calculation functions and the vertex calculation functions of typical GNNs, respectively. Some complex GNNs like GAT [**?**] and GaAN [**?**] use more than one message passing phase in each GNN layer. We regard every message passing phase in a GNN layer as a *sub-layer*. We will give out more details on sub-layers when we introduce GAT.

GNN	Type	Σ^l	$\phi^l(\boldsymbol{h}_y^l, \boldsymbol{h}_x^l, \boldsymbol{e}_{y,x}, \dots)$	Complexity
ChebNet [**?**]	Spectral	sum	$\boldsymbol{m}_{y,x}^l = \boldsymbol{e}_{y,x}\boldsymbol{h}_y^l$	$O(d_{in})$
GCN [**?**]	Spectral	sum	$\boldsymbol{m}_{y,x}^l = \boldsymbol{e}_{y,x}\boldsymbol{h}_y^l$	$O(d_{in})$
AGCN [**?**]	Spectral	sum	$\boldsymbol{m}_{y,x}^l = \tilde{e}_{y,x}^l\boldsymbol{h}_y^l$	$O(d_{in})$
GraphSAGE [**?**]	Non-spectral	mean/LSTM	$\boldsymbol{m}_{y,x}^l = \boldsymbol{h}_y^l$	$O(1)$
GraphSAGE-pool [**?**]	Non-spectral	max	$\boldsymbol{m}_{y,x}^l = \delta(\boldsymbol{W}_{pool}^l\boldsymbol{h}_y^l + b^l)$	$O(d_{in} * d_{out})$
Neural FPs [**?**]	Non-spectral	sum	$\boldsymbol{m}_{y,x}^l = \boldsymbol{h}_y^l$	$O(1)$
SSE [**?**]	Recurrent	sum	$\boldsymbol{m}_{y,x}^l = \boldsymbol{v}_y \parallel \boldsymbol{h}_y^l$	$O(f + d_{in})$
GGNN [**?**]	Gated	sum	$\boldsymbol{m}_{y,x}^l = \hat{\boldsymbol{h}}_y^l$	$O(d_{in} * d_{out})$
GAT [**?**]	Attention	sum	Sub-layer 0:	concat: $O(d_{out})$
			$\boldsymbol{m}_{y,x}^{l,0} = \|_{k=1}^K \exp(LeakyReLU(\boldsymbol{a}^T[\hat{\boldsymbol{h}}_y[k] \parallel \hat{\boldsymbol{h}}_x[k]]))$	average: $O(K * d_{out})$
			Sub-layer 1 (multi-head concatenation):	Two Sub-layers
			$\boldsymbol{m}_{y,x}^{l,1} = \|_{k=1}^K \frac{\exp(LeakyReLU(\boldsymbol{a}^T[\hat{\boldsymbol{h}}_y[k] \parallel \hat{\boldsymbol{h}}_x[k]]))}{\boldsymbol{h}_x^{l,0}[k]}\hat{\boldsymbol{h}}_y[k]$	
			Sub-layer 1 (multi-head average) :	
			$\boldsymbol{m}_{j,i}^{l,1} = \frac{1}{K}\sum_{k=1}^K \frac{\exp(LeakyReLU(\boldsymbol{a}^T[\hat{\boldsymbol{h}}_y[k] \parallel \hat{\boldsymbol{h}}_x[k]]))}{\boldsymbol{h}_x^{l,0}[k]}\hat{\boldsymbol{h}}_y[k]$	
GaAN [**?**]	Attention	Sub-layer 0: sum Sub-layer 1: sum Sub-layer 2: max Sub-layer 3: mean	Sub-layer 0:	$O(\max(d_a, d_v, d_m) * K * d_{in})$
			$\boldsymbol{m}_{y,x}^{l,0} = \|_{k=1}^K \exp((\boldsymbol{W}_{(k),1}^l\boldsymbol{h}_y^l + \boldsymbol{b}_{(k),1}^l)^T(\boldsymbol{W}_{(k),2}^l\boldsymbol{h}_x^l + \boldsymbol{b}_{(k),2}^l))$	Four Sub-layers $\boldsymbol{W}_{(k),1}^l \in \mathbb{R}^{d_a \times d_{in}}$ $\boldsymbol{W}_{(k),v}^l \in \mathbb{R}^{d_v \times d_{in}}$ $\boldsymbol{W}_m^l \in \mathbb{R}^{d_m \times d_{in}}$
			Sub-layer 1:	
			$\alpha_{(k)} = \frac{\exp((\boldsymbol{W}_{(k),1}^l\boldsymbol{h}_y^l + \boldsymbol{b}_{(k),1}^l)^T(\boldsymbol{W}_{(k),2}^l\boldsymbol{h}_x^l + \boldsymbol{b}_{(k),2}^l))}{\boldsymbol{h}_x^{l,0}[k]}$	
			$\boldsymbol{m}_{y,x}^{l,1} = \|_{k=1}^K \alpha_{(k)} LeakyReLU(\boldsymbol{W}_{(k),v}^l\boldsymbol{h}_y^l + \boldsymbol{b}_{(k),v}^l)$	
			Sub-layer 2:	
			$\boldsymbol{m}_{y,x}^{l,2} = \boldsymbol{W}_m^l\boldsymbol{h}_y^l + \boldsymbol{b}_m^l$	
			Sub-layer 3:	
			$\boldsymbol{m}_{y,x}^{l,3} = \boldsymbol{h}_y^l$	

Table 2: Typical graph neural networks and their edge calculation functions. d_{in} and d_{out} are the dimensions of the input and output hidden vectors, respectively. Blue variables are model parameters to learn. δ is the activation function.

GNN	$\gamma^l(\boldsymbol{h}_x^l, \boldsymbol{s}_x^l, \dots)$	Complexity
ChebNet [**?**]	$\boldsymbol{h}_x^{l+1} = 2\boldsymbol{s}_x^l - \boldsymbol{h}_x^{l-1}$	$O(d_{out})$
GCN [**?**]	$\boldsymbol{h}_x^{l+1} = \boldsymbol{W}^l \boldsymbol{s}_x^l$	$O(d_{in} * d_{out})$
AGCN [**?**]	$\boldsymbol{h}_x^{l+1} = \boldsymbol{W}^l \boldsymbol{s}_x^l$	$O(d_{in} * d_{out})$
GraphSAGE [**?**]	$\boldsymbol{h}_x^{l+1} = \delta(\boldsymbol{W}^l[\boldsymbol{s}_x^l \parallel \boldsymbol{h}_x^l])$	$O(d_{in} * d_{out})$
GraphSAGE-pool [**?**]	$\boldsymbol{h}_x^{l+1} = \boldsymbol{s}_x^l$	$O(1)$
Neural FPs [**?**]	$\boldsymbol{h}_x^{l+1} = \delta(\boldsymbol{W}^{l,\lvert\mathcal{N}(v_i)\rvert}(\boldsymbol{h}_x^l + \boldsymbol{s}_x^l))$	$O(d_{in} * d_{out})$
SSE [**?**]	$\boldsymbol{h}_x^{l+1} = (1-\alpha)\boldsymbol{h}_x^l + \alpha\delta(\boldsymbol{W}_1^l\delta(\boldsymbol{W}_2^l[\boldsymbol{v}_x \parallel \boldsymbol{s}_x^l]))$	$O((f + d_{in}) * d_{out})$
GGNN [**?**]	Preprocessing: $\hat{\boldsymbol{h}}_x^l = \boldsymbol{W}^l \boldsymbol{h}_x^l$;	$O(d_{in} * d_{out})$
	$\boldsymbol{z}_x^l = \delta(\boldsymbol{W}^z \boldsymbol{s}_x^l + \boldsymbol{b}^{sz} + \boldsymbol{U}^z \boldsymbol{h}_x^l + \boldsymbol{b}^{hz})$	
	$\boldsymbol{r}_x^l = \delta(\boldsymbol{W}^r \boldsymbol{s}_x^l + \boldsymbol{b}^{sr} + \boldsymbol{U}^r \boldsymbol{h}_x^l + \boldsymbol{b}^{hr})$	
	$\bar{\boldsymbol{h}}_x^{l+1} = \tanh(\boldsymbol{W}\boldsymbol{s}_x^l + \boldsymbol{b}^s + \boldsymbol{U}(\boldsymbol{r}_x^l \odot \boldsymbol{h}_x^l) + \boldsymbol{b}^h)$	
	$\boldsymbol{h}_x^{l+1} = (1 - \boldsymbol{z}_x^l) \odot \boldsymbol{h}_x^l + \boldsymbol{z}_x^l \odot \bar{\boldsymbol{h}}_x^{l+1}$	
GAT [**?**]	Preprocessing: $\hat{\boldsymbol{h}}_x^l = \parallel_{k=1}^K \boldsymbol{W}_{(k)}^l \boldsymbol{h}_x^l$;	concat:$O(d_{in} * d_{out})$
	Sub-layer 0:	average:$O(K * d_{in} * d_{out})$
	$\boldsymbol{h}_x^{l,0} = \boldsymbol{s}_x^{l,0}$	Two Sub-layers
	Sub-layer 1:	
	$\boldsymbol{h}_x^{l+1} = \boldsymbol{h}_x^{l,1} = \delta(\boldsymbol{s}_x^{l,1})$	
GaAN [**?**]	Sub-layer 0/1/2:	$O(\max(K * d_v + d_{in}, 2 * d_{in} + d_m) * d_{out})$ Four Sub-layers
	$\boldsymbol{h}_x^{l,*} = \boldsymbol{s}_x^{l,*}$	
	Sub-layer 3:	
	$\boldsymbol{g}_x^l = \boldsymbol{W}_g^l[\boldsymbol{h}_x^l \parallel \boldsymbol{h}_x^{l,2} \parallel \boldsymbol{s}_x^{l,3}] + \boldsymbol{b}_g^l$	
	$\boldsymbol{h}_x^{l+1} = \boldsymbol{h}_x^{l,3} = \boldsymbol{W}_o^l[\boldsymbol{h}_x^l \parallel (\boldsymbol{g}_x^l \odot \boldsymbol{h}_x^{l,1})] + \boldsymbol{b}_o^l$	

Table 3: Typical graph neural networks and their vertex calculation functions. d_{in} and d_{out} are the dimensions of the input and output hidden 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 the layer l. δ is the activation function.

2.3. Representative GNNs

We use O_ϕ/O_Σ/O_γ 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 O_ϕ + O_Σ and the vertex calculation complexity O_γ. In Table 2 and Table 3, we list the edge and vertex calculation complexity of each GNN, 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).

Since we focus on analyzing the performance bottleneck in training GNNs, 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.

2.3.1. GCN (Low Vertex & Low Edge Complexity)

Graph convolution network (GCN [**?**]) introduces the first-order approximation of the spectral-based graph convolutions into graph neural networks. It has only one parameter to learn at each layer, i.e. the weight matrix $\boldsymbol{W}^l$ in the updating function γ^l. A GCN graph neuron can be expressed as $\boldsymbol{h}_x^{l+1} = \boldsymbol{W}^l \sum_{v_y \in \mathcal{N}(v_x)} w_{y,x} \boldsymbol{h}_y^l$, where $w_{y,x}$ is the normalized weight of the edge $e_{y,x}$. According to the associative law of the matrix multiplication, $\boldsymbol{h}_x^{l+1} = \sum_{v_y \in \mathcal{N}(v_x)} w_{y,x} \boldsymbol{W}^l \boldsymbol{h}_y^l$. Since the dimension of $\boldsymbol{h}_x^{l+1}$ is usually smaller than $\boldsymbol{h}_x^l$ in practical GCNs, the implementation of GCN in PyG chooses to first conduct the vertex calculation $\hat{\boldsymbol{h}}_y^l = \boldsymbol{W}^l \boldsymbol{h}_y^l$ for each vertex v_y and then conduct the edge calculation $\boldsymbol{h}_x^{l+1} = \sum_{v_y \in \mathcal{N}(v_x)} w_{y,x} \hat{\boldsymbol{h}}_y^l$. As $\hat{\boldsymbol{h}}_y^l$ has the same dimension as $\boldsymbol{h}_x^{l+1}$, the implementation significantly reduces the computation cost of the edge calculation.

2.3.2. GGNN (High Vertex & Low Edge Complexity)

GGNN [**?**] introduces the gated recurrent unit (GRU) into graph neural networks. The updating function ϕ^l 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

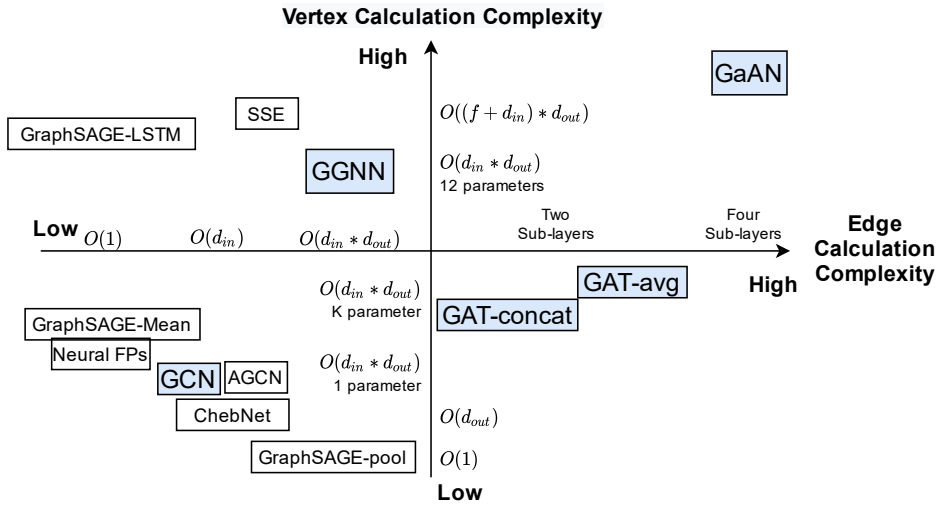


Figure 3: Complexity quadrants of typical GNNs. We compare the complexity according to the number of sub-layers, the Big-O notation, and the number of parameters to train.

the dimension of $\boldsymbol{h}^{l+1}$ is equal to the dimension of $\boldsymbol{h}^{l}$. Since the messaging function ϕ^l only uses the hidden vector $\boldsymbol{h}_y^l$ of the source vertex v_y for an edge $e_{y,x}$, in the implementation of PyG, GGNN conducts the pre-processing vertex calculation $\hat{\boldsymbol{h}}_x^l = \boldsymbol{W}^l \boldsymbol{h}_x^l$ for every vertex v_x before the message passing. The messaging function ϕ^l directly uses $\hat{\boldsymbol{h}}_y^l$ as the message vector for every edge $e_{y,x}$. 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.

2.3.3. GAT (Low Vertex & High Edge Complexity)

GAT [**?**] introduces the multi-head attention mechanism into graph neural networks. Each GAT layer has K heads that generate K independent views for an edge, where K is a hyper-parameter. The views of K heads can be merged by concatenating or by averaging. For concatenating, the dimension of the hidden vector of each head d_{head} is d_{out}/K. For averaging, d_{head} is d_{out}.

Each GAT layer consists of a vertex pre-processing phase and two sub-layers (i.e., message-passing phases).

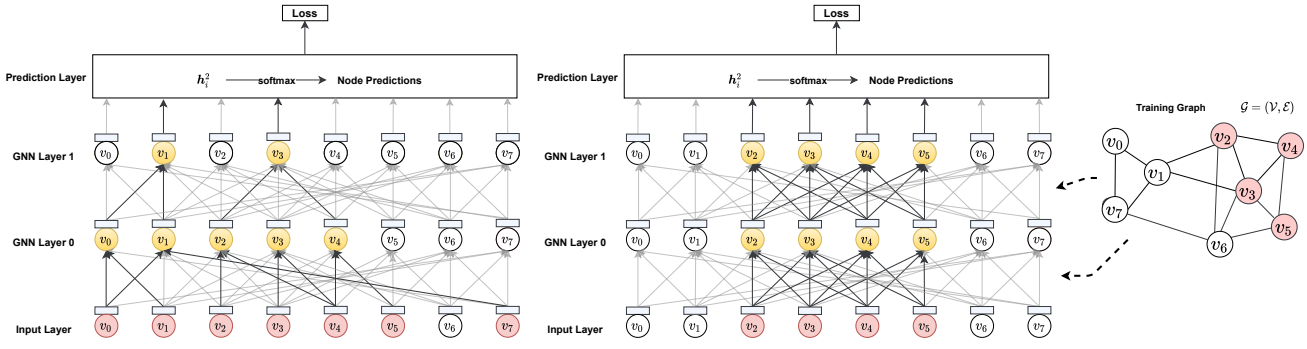
The vertex pre-processing phase calculates the attention vector $\hat{\boldsymbol{h}}_x^l$ for every vertex v_x by $\hat{\boldsymbol{h}}_x = \|_{k=1}^K \boldsymbol{W}_{(k)}^l \boldsymbol{h}_x^l$. We denote the attention sub-vector generated by the k-th head as $\hat{\boldsymbol{h}}_x[k] = \boldsymbol{W}_{(k)}^l \boldsymbol{h}_x^l$.

The first sub-layer of GAT (defined in Equation 2) uses the attention vectors to emit the attention weight vector $\boldsymbol{m}_{y,x}^{l,0}$ for every edge $e_{y,x}$ and aggregates the attention weight vectors for every vertex v_x to get the weight sum vector $\boldsymbol{h}_x^{l,0}$.

$$
\begin{aligned}
\boldsymbol{m}_{y,x}^{l,0} &= \phi^{l,0}(\boldsymbol{h}_y^l, \boldsymbol{h}_x^l, \boldsymbol{e}_{y,x}, \hat{\boldsymbol{h}}_y, \hat{\boldsymbol{h}}_x) = \|_{k=1}^K \exp(LeakyReLU(\boldsymbol{a}^T[\hat{\boldsymbol{h}}_y[k] \parallel \hat{\boldsymbol{h}}_x[k]])), \\
\boldsymbol{s}_x^{l,0} &= \textstyle\sum_{v_y \in \mathcal{N}(v_x)} \boldsymbol{m}_{y,x}^{l,0}, \\
\boldsymbol{h}_x^{l,0} &= \gamma^{l,0}(\boldsymbol{h}_x^l, \boldsymbol{s}_x^{l,0}) = \boldsymbol{s}_x^{l,0}.
\end{aligned}
\tag{2}
$$

The second sub-layer of GAT (defined in Equation 3) uses the weight sum vectors to normalize the attention weights for every edge and aggregates the attention vectors $\hat{\boldsymbol{h}}_y^l$ with the normalized weights. The aggregated attention vectors $\boldsymbol{s}_x^{l,1}$ are transformed by an activation function δ and are outputted as the hidden vectors of the current layer $\boldsymbol{h}_x^{l+1}$.

$$
\begin{aligned}
\boldsymbol{m}_{y,x}^{l,1} &= \phi^{l,1}(\boldsymbol{h}_y^{l,0}, \boldsymbol{h}_x^{l,0}, \boldsymbol{e}_{y,x}, \hat{\boldsymbol{h}}_y, \hat{\boldsymbol{h}}_x) = \|_{k=1}^K \frac{\exp(LeakyReLU(\boldsymbol{a}^T[\hat{\boldsymbol{h}}_y[k] \parallel \hat{\boldsymbol{h}}_x[k]]))}{\boldsymbol{h}_x^{l,0}[k]} \hat{\boldsymbol{h}}_y[k], \\
\boldsymbol{s}_x^{l,1} &= \textstyle\sum_{v_y \in \mathcal{N}(v_x)} \boldsymbol{m}_{y,x}^{l,1}, \\
\boldsymbol{h}_x^{l+1} = \boldsymbol{h}_x^{l,1} &= \gamma^{l,1}(\boldsymbol{h}^{l,0}, \boldsymbol{s}_x^{l,1}) = \delta(\boldsymbol{s}_x^{l,1}).
\end{aligned}
\tag{3}
$$



(a) Neighbor sampling (b) Graph sampling

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

2.3.4. GaAN (High Vertex & High Edge Complexity)

Based on the multi-head mechanism, GaAN [**?**] introduces a convolutional subnetwork to control the weight of each head. Each GaAN layer consists of four sub-layers (as defined in Table 2 and Table 3). 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 sum of attention weights of every vertex in all heads $\boldsymbol{h}_x^{l,0}$. The second sub-layer aggregates the hidden vectors from the previous GaAN layer with the normalized attention weights for all heads $\alpha_{(k)}$. The third and fourth sub-layers aggregate the hidden vectors from the GaAN previous layer with the element-wise max and mean operators separately. The vertex updating funciton $\gamma^{l,4}$ of the fourth sub-layer uses the hidden vectors of the last two sub-layers $\boldsymbol{h}_x^{l,1}, \boldsymbol{h}_x^{l,2}$ to generate the output hidden vector $\boldsymbol{h}_x^{l+1}$.

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 [**?**]. 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 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 [**? ? ? ? ?**] sample nodes or edges layer by layer. The sampled subgraphs of different GNN layers may be *different*, as shown in Figure 4a. GraphSAGE [**?**] 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_x in the GNN layer l, GraphSAGE samples at most S^l neighbors of v_x from 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 [**? ? ?**] 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 [**?**] is the representative technique. Given a training graph $\mathcal{G}$, it partitions $\mathcal{G}$ into dense clusters. For each mini-batch, it randomly pick N clusters to form the sampled subgraph, where N 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 our experimental setting in Section 3.1 and then give out our experimental scheme in Sec-

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

Table 4: Dataset overview. $\bar{d}$ represents the average vertex degree. $dim(\boldsymbol{v})$ is the dimension of the input feature vector.

tion 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 server with the Linux 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 card 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 GNNs with PyG 1.5.0[2].

Datasets. We used six real-world graph datasets as listed in Table 4 that were popular in the GNN accuracy evaluation [**? ? ?**]. For directed graphs, PyG converts them into undirected ones during data loading. Thus, the average degree of a directed graph $\bar{d} = \frac{2|\mathcal{E}|}{|\mathcal{V}|}$. For an undirected graph, the average degree is defined as $\bar{d} = \frac{|\mathcal{E}|}{|\mathcal{V}|}$. For the cam dataset, we generated random dense feature vectors. We also used random graphs generated by the R-MAT graph generator [**?**] in some experiments, to explore the effects of graph topological characteristics (like average degrees) on performance bottlenecks. Input feature vectors of random graphs were random dense vectors with the 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 in the semi-supervised learning setting. All vertices and their input feature vectors were used, but only a part 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, and GaAN. To compare performance characteristics of the four GNNs side-by-side, we used a unified GNN structure for them: Input Layer $\rightarrow$ GNN Layer 0 $\rightarrow$ GNN Layer 1 $\rightarrow$ Softmax Layer (to prediction). The structure was popular in the experimental evaluation of GCN [**?**], GAT [**?**], and GaAN [**?**]. Since a GGNN layer requires the input and output hidden vectors have the same dimension, we added two multi-layer perceptron (MLP) layers to transform the dimensions of the input/output feature vectors: Input Layer $\rightarrow$ MLP $\rightarrow$ GGNN Layer 0 $\rightarrow$ GGNN Layer 1 $\rightarrow$ MLP $\rightarrow$ Softmax Layer. Unless otherwise specified, we stored the dataset and the model parameters on the GPU side. The GNN training was conducted also on the GPU side.

Hyper-parameters. We use $dim(\boldsymbol{x})$ to denote the dimension of a vector $\boldsymbol{x}$. We picked the hyper-parameters of GNNs according to their popularity in their references [**? ? ? ?**]. Unless otherwise mentioned, we used the same set of hyper-parameters for all the datasets. Some hyper-parameters (like dimensions of hidden vectors) were common in the four GNNs and we set them to the same values. For GCN/GAT/GaAN, we set $\boldsymbol{h}_x^0 = \boldsymbol{v}_x$, $dim(\boldsymbol{h}_x^1) = 64$, and $dim(\boldsymbol{h}_x^2) = \#Class$ (the number of classes in the dataset). For GAT, the first GAT layer contained 8 heads, and the attention vector of each head had a dimension of 8. The first GAT layer merged attention vectors of 8 heads by concatenating. The second GAT layer used a single head with a dimension of $\#Class$. For GGNN, we set $dim(\boldsymbol{h}_x^0) = dim(\boldsymbol{h}_x^1) = dim(\boldsymbol{h}_x^2) = 64$. We used 8 heads in the both GaAN layers with $d_a = d_v = 8$, and $d_m = 64$.

[2] https://pytorch-geometric.readthedocs.io/en/1.5.0/index.html

Sampling Techniques. We picked the neighbor sampler from GraphSAGE [**?**] and the cluster sampler from ClusterGCN [**?**] as the typical sampling techniques. For the neighbor sampler, we set the neighborhood sample sizes of the GNN layer 0 and the GNN layer 1 to 10 and 25, respectively. We set the default batch size to 512, according to [**?**]. For the cluster sampler, we partitioned the input graph into 1500 partitions and used 20 partitions per batch, according to [**?**].

3.2. Experimental Scheme

To find out performance bottlenecks in GNN training, we conducted the experimental analysis with four questions. The answers to those questions will give us a more comprehensive view of 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, such as 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 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 verified the validity of the time complexity analysis in Table 2 and Table 3. The complexity analysis allowed us to analyze performance bottlenecks theoretically.

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

We decomposed the GNN training time on different levels: the layer level, the 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 was the performance bottleneck. Optimizing its implementation will significantly reduce the training time.

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

The limited memory capacity of a GPU prevents us from training GNNs on big graphs. We measured the peak memory usage during GNN training under different graph scales, input feature dimensions, and average degrees. Based on the results, we analyzed which was 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 sampling techniques remove performance bottlenecks in GNN training?* (Section 4.4)

Theoretically, 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 effectiveness of sampling techniques, we measured the training time and peak memory usage under different batch sizes. If sampling techniques were effective, they will be the keys to conduct GNN training on very big graphs. If they were not effective, we want to find out which impairs its efficiency.

4. Evaluation Results and Analysis

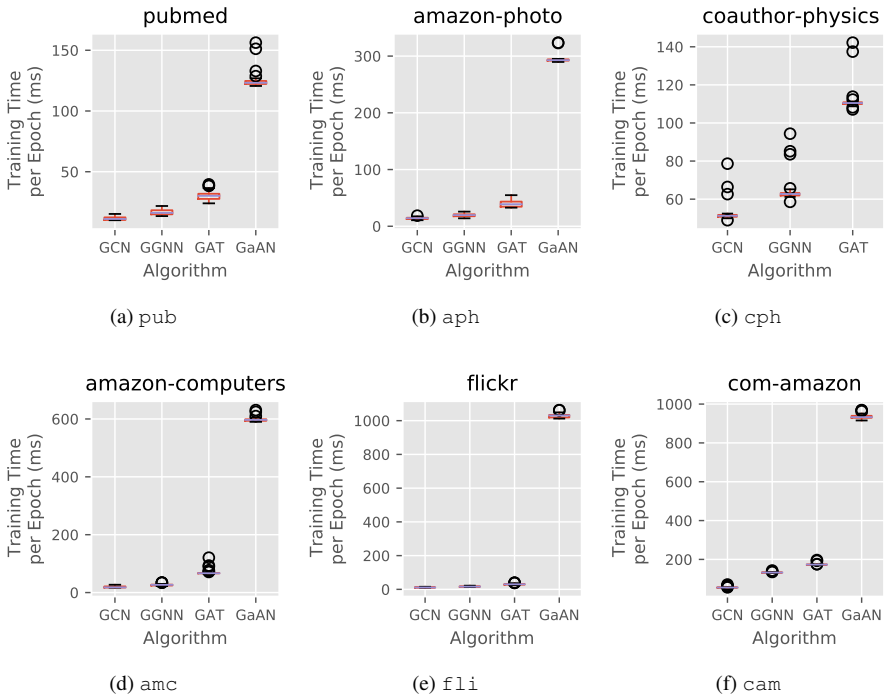
We answer the four questions in Section 3.2 one by one with experiments. Without otherwise mentioned, the reported training time per epoch was the average wall-clock training time of 50 epochs, excluding abnormal epochs [3].

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

4.1. *Effects of Hyper-parameters on Performance*

According to Table 2 and Table 3, the time complexity of the messaging function ϕ^l and the updating function γ^l 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 2 and Table 3, we first compared the training time of the four GNNs. The ranking of the training time was GaAN $\gg$ GAT $>$ GGNN $>$ GCN in all cases. Since the real-world graphs had more edges than vertices ($|\mathcal{E}| > |\mathcal{V}|$), the time complexity of the edge calculation stage affected more than the vertex calculation stage. The ranking was consistent with the time complexity analysis.

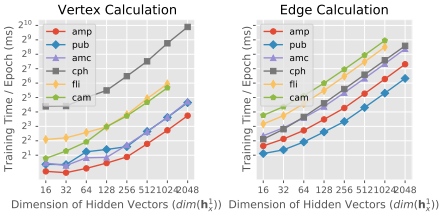


(a) `pub` (b) `aph` (c) `cph`

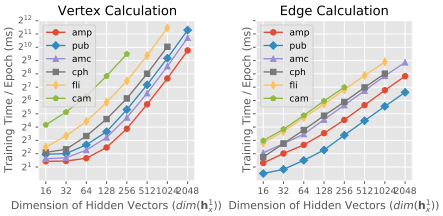
(d) `amc` (e) `fli` (f) `cam`

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.

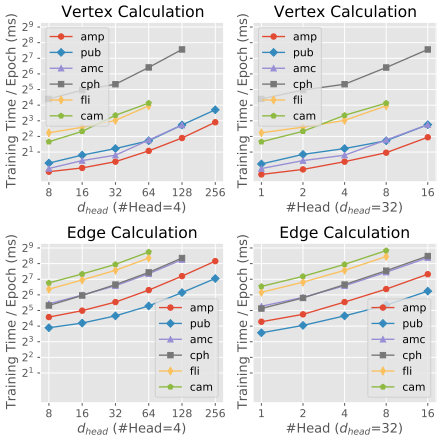
To further evaluate effects of hyper-parameters on performance, we measured the training time of each GNN with varying hyper-parameters in Figure 6.



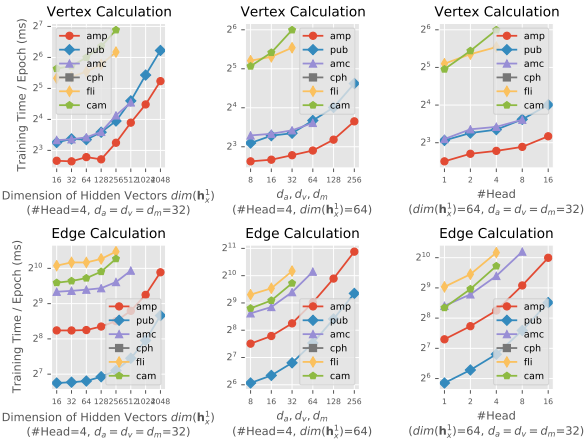
(a) GCN



(b) GGNN



(c) GAT



(d) GaAN

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

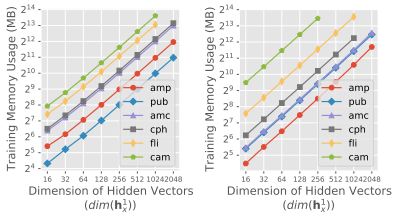
For GCN and GGNN, $dim(\boldsymbol{h}_x^0)$ and $dim(\boldsymbol{h}_x^1)$ were solely determined by the dataset with $d_{in}^0 = dim(\boldsymbol{h}_x^0) = dim(\boldsymbol{v_x})$ and $d_{out}^1 = dim(\boldsymbol{h}_x^2) = \#Class$. Therefore, the only modifiable hyper-parameter was the dimension

of $\boldsymbol{h}_x^1$ that affected the dimension of output hidden vectors of the layer 0 d_{out}^0 and the dimension of input hidden vectors of the layer 1 d_{in}^1 simultaneously, i.e. $dim(\boldsymbol{h}_x^1) = d_{out}^0 = d_{in}^1$. According to the time complexity analysis, if we fixed other hyper-parameters but only increased $dim(\boldsymbol{h}_x^1)$, the computational costs of the GNN layer 0 and the GNN layer 1 should both increase linearly with $dim(\boldsymbol{h}_x^1)$, 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 $dim(\boldsymbol{h}_x^1)$ when $dim(\boldsymbol{h}_x^1)$ was big, consistent with the theoretical analysis.

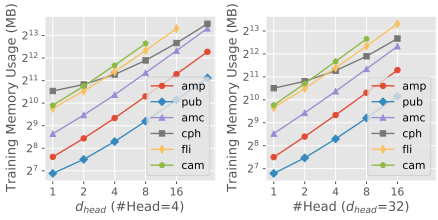
For GAT, we modified the number of heads K and the dimension of each head d_{head} in the GAT layer 0. The dimension of $\boldsymbol{h}_x^1$ is determined as $dim(\boldsymbol{h}_x^1) = Kd_{head}$. Thus, the computational costs of the GAT layer 0 and the GAT layer 1 should increase linearly with K and d_{head} separately. Figure 6c confirms the theoretical analysis.

For GaAN, it is also based on the multi-head mechanism. Its time complexity shoud be affected by $dim(\boldsymbol{h}_x^1)$ ($d_{out}^0 = d_{in}^1 = dim(\boldsymbol{h}_x^1)$), d_a, d_v, d_m, and the number of heads K. Figure 6d demonstrates that the training time increased linearly with the hyper-parameters, except for $dim(\boldsymbol{h}_x^1)$. As $dim(\boldsymbol{h}_x^1)$ increased, the training time increased first slightly and then linearly. We observed similar phenomena in GCN, GGNN, and GAT: When the values of hyper-parameters were too low, GNN training could not make full use of the computing power of the GPU. When the values of hyper-parameters became high enough, training time increased linearly, supporting the time complexity analysis.

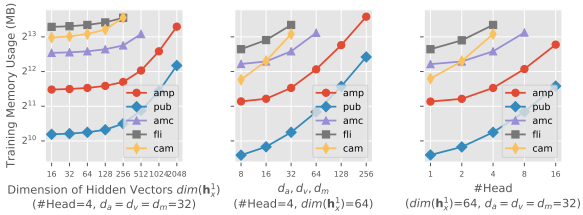
We further measured the effects of the hyper-parameters on the peak GPU memory usage in Figure 7. The memory usage also increased linearly as the hyper-parameters increased for all GNNs, except for GaAN on $dim(\boldsymbol{h}_x^1)$. As the hidden vectors $\boldsymbol{h}_x^1$ consumed a small proportion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



(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.

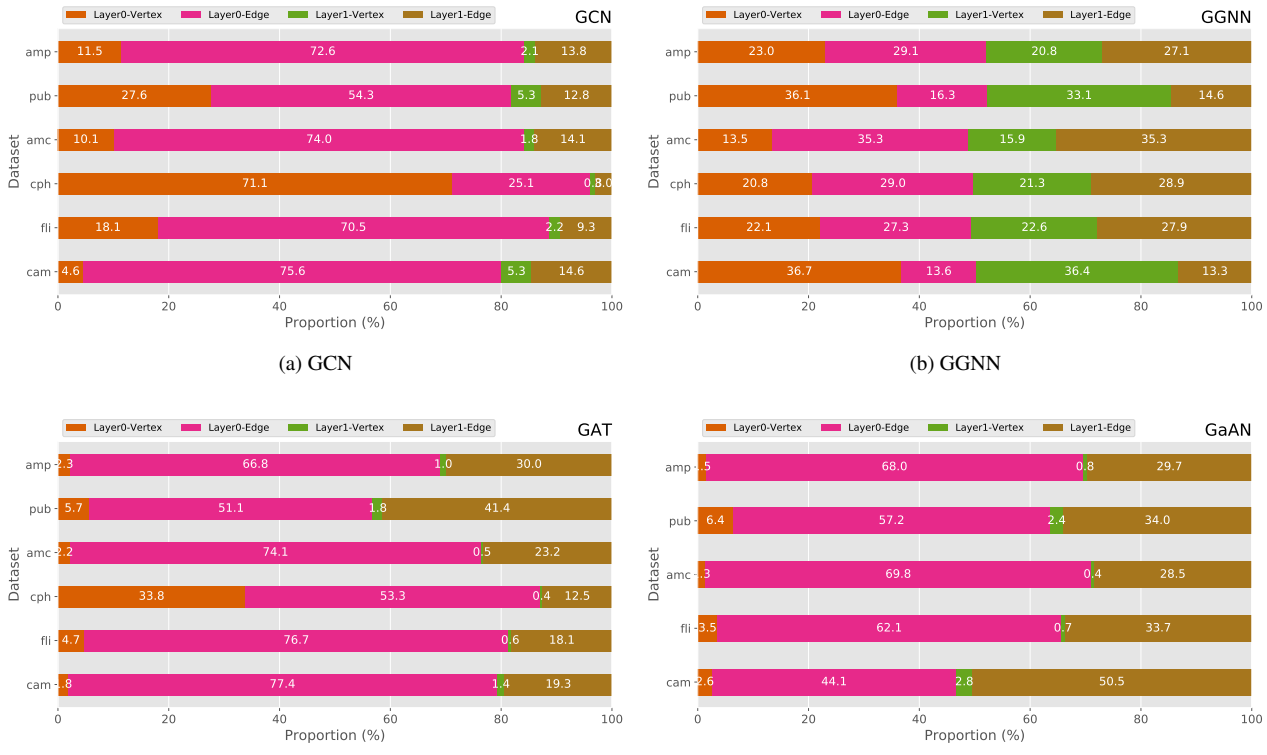
Summary. The complexity analysis in Table 2 and Table 3 is valid. Fixing other hyper-parameters, each hyper-parameter itself affects the training time and the memory usage of a GNN Layer *in a linear way*. Algorithm engineers can adjust hyper-parameters according to the time complexity to avoid explosive growth in the training time and memory usage.

4.2. Training Time Breakdown

To find out which stage/step dominated the training time, we decomposed the training time and analyzed performance bottlenecks 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 was the summation of the time in the forward, backward, and evaluation phases. In GCN, GAT, and GaAN, the time spent on the layer 0 was much larger than the layer 1. In those GNNs, the dimensions of the input/output hidden vectors in the layer 0 were much larger than the dimensions in the layer 1: $d^0_{in} = dim(\boldsymbol{v}_x)$, $d^0_{out} = d^1_{in} = 64$, $d^1_{out} = \#Class$, and $dim(\boldsymbol{v}_x) \gg \#Class$. For GaAN, since it required the dimensions of the input/output hidden vectors must be the same, the hyper-parameters were set to $d^0_{in} = d^0_{out} = d^1_{in} = d^1_{out} = 64$ and the training time of both layers was close.



(a) GCN (b) GGNN

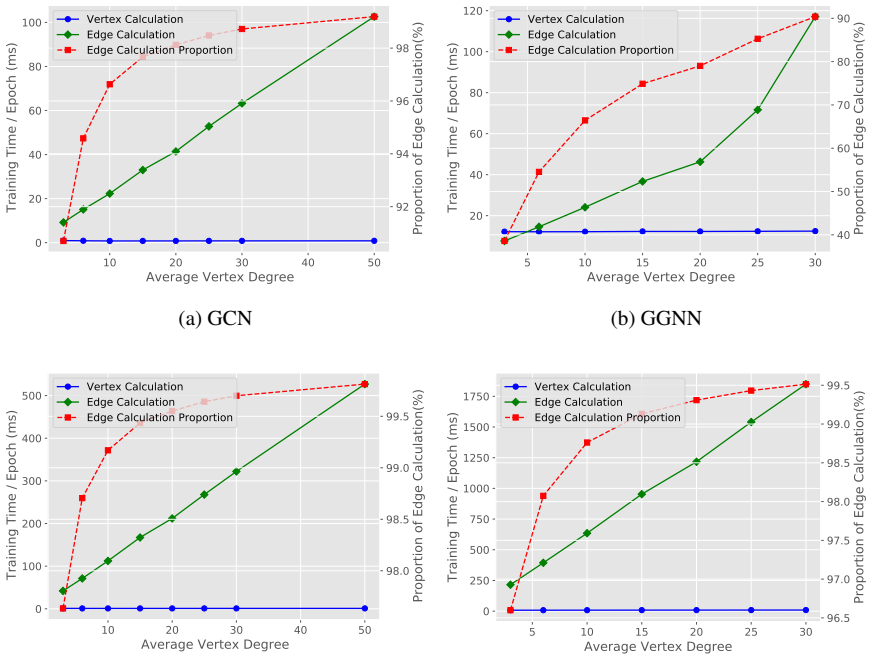
(c) GAT (d) GaAN

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

Each GNN layer was further divided into the vertex and the edge calculation stages. In Figure 8, GCN spent most of the training time on the edge calculation stage on most datasets. A special case was the `cph` dataset. The dimension of the input feature vectors was very high in `cph`, making the vertex calculation stage of the GCN Layer 0 spend considerable time. GGNN also spent the majority of its training time on the edge calculation stage, but the high time complexity of its vertex updating function γ^l made the proportion of the vertex calculation in the total training time much higher than the other GNNs. For GAT and GaAN, due to their high edge calculation complexity, the edge calculation stage was the dominant stage.

The experimental results also indicated that the average degree of the dataset affected the proportion of the edge/vertex calculation time in the total training time. For GaAN, the time spent on the vertex calculation stage exceeded the edge calculation stage on the `pub` and `cam` datasets, because the average degrees of the two datasets were low, making $|\mathcal{E}|$ and $|\mathcal{V}|$ much closer. To evaluate the effects of the average degree, we generated random graphs with 50,000 vertices and 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 increased, the training time of the edge calculation stage grew *linearly*. For GCN, GAT, and GaAN, the edge calculation stage dominated the entire training time even when the average degrees were small. Only for GGNN that had high vertex and low edge calculation complexity, the training time of the vertex calculation stage exceeded the edge calculation stage under low average degrees (< 5).

In summary, *the edge calculation stage was the most time-consuming stage in GNN training*. Improving its efficiency is the key to reduce the GNN training time.



(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.

4.2.2. Step Level in Edge Calculation

We further investigated the most time-consuming step of the edge calculation stage. In the implementation of PyG, the edge calculation stage consists of four steps: collection, messaging, aggregation, and vector updating, as shown in Figure 10. The edge index is a matrix with $|\mathcal{E}|$ rows and two columns. It holds the edge set of the graph. The two columns of the edge index store the source and the target vertex IDs of each edge. The collection step copies the hidden vectors from the previous GNN layer $\boldsymbol{h}_x^l$ and $\boldsymbol{h}_y^l$ to the both endpoints of each edge $e_{x,y}$ in the edge index, forming the parameter tensor $[\boldsymbol{h}_x^l, \boldsymbol{h}_y^l, \boldsymbol{e}_{x,y}]$ of the messaging function ϕ^l. This step only involves data movement. The messaging step calls the messaging function ϕ^l on all edges to get message vectors $\boldsymbol{m}_{x,y}^l$. The aggregation step aggregates the message vectors with the same target vertex into an aggregated vector $\boldsymbol{s}_x^l$. The vector updating step is optional. It performs an additional transformation on the aggregated vectors (for example, adding the bias in GCN).

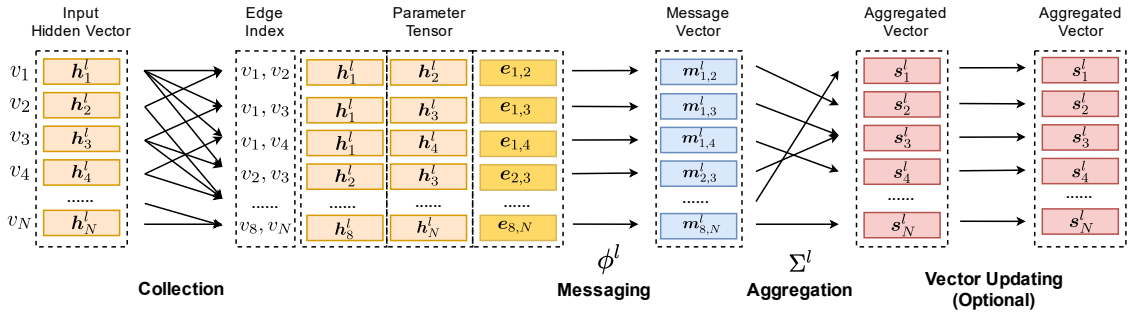


Figure 10: Step decomposition of the edge calculation stage of the GNN layer l.

We decomposed the execution time of the edge calculation stage in Figure 11. In each GNN, the proportions of the four steps were rather stable, rarely affected by datasets. For GAT and GaAN with the high edge calculation complexity, the messaging step consumed most of the training time. For GCN and GGNN with the low edge complexity, the proportions of the steps were close. Since the messaging function ϕ^l of GGNN used the pre-computed $\hat{\boldsymbol{h}}_y^l$ as the message vector directly, the time spent on the messaging step of GGNN was negligible. Although the collecting step did not conduct any computation and only involved data movement, it occupied noticeable execution time in the four GNNs.

The results indicate that *the performance bottlenecks of the edge calculation stage depend on the complexity of the messaging function* ϕ^l. When the time complexity of ϕ^l is high, the messaging step is the performance bottleneck. Optimizing the implementation of ϕ can significantly reduce training time. Otherwise, the collection and the aggregation steps are performance bottlenecks. Improving the efficiency of the two steps can benefit all GNNs.

(a) GCN

(b) GGNN

(c) GAT

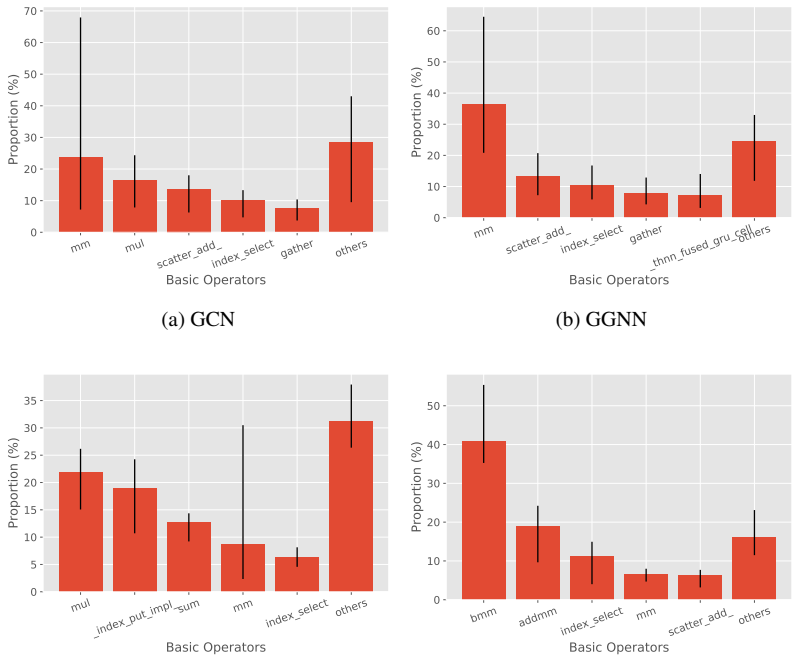
(d) GaAN

Figure 11: Training time breakdown of the edge calculation stage.

4.2.3. *Operator Level*

The functions ϕ, Σ and γ in the edge and vertex calculation stages are made up of a series of basic operators implemented on the GPU side, such as the matrix multiplication `mm`, the elementwise multiplication `mul` and

the index-based selection `index_select`. Figure 12 shows the top-5 time-consuming basic operators in each GNN, averaged over all datasets.



(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 was averaged over all datasets with the error bar indicating the maximum and the minimum.

GCN. The most time-consuming basic operator was the matrix multiplication `mm` used in the vertex updating function γ. The elementwise multiplication `mul` used in the messaging function ϕ was also time-consuming. The other three operators were 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 consumed the majority of the training time.

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

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

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

The most time-consuming operators in the four GNNs were the matrix multiplication `mm` and the elementwise multiplication `mul`, *making GNN training suitable for GPUs*. Although the aggregation step in the edge calculation stage was relatively simple (like sum and mean), the related operators–`scatter_add` and `gather`–still consumed a certain amount of the time. The two operators had to synchronize between hardware threads to avoid updating the same aggregated vector at the same time. They also conducted non-regular memory access with the access pattern determined by the edge set dynamically. For GPUs, they were less efficient than `mm`. The index-based selection operator `index_select` used in the collection step consumed about 10% of the training time in all GNNs. Improving the efficiency of `scatter_add`/`gather`/`index_select` can benefit all kinds of GNNs.

Summary of Training Time Breakdown. The GNN training was suitable for GPUs. *The edge calculation stage was 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 depended on the time complexity of the messaging function ϕ. If the time complexity of ϕ was high, ϕ dominated the training time of the edge calculation stage. Optimizations should focus on improving its efficiency. Otherwise, the collection step and the aggregation step dominated the training time. The collection step suffered from lots of data movement. The aggregation step suffered from data synchronization and non-regular data access.

4.3. Memory Usage Analysis

During the GNN training, we stored all data (including datasets and intermediate results) in the on-chip memory of the GPU. Compared with the main memory on the host side (90 GB), the capacity of the GPU memory (16 GB) was very limited. *The GPU memory capacity limited the scales of the graphs that it could handle*. For example, GaAN was unable to train on the `cph` dataset due to the out of memory exception.

Figure 13 shows the peak memory usage of each phase during the GNN training on the `amp` dataset. The trends on the other datasets were similar. *The GNN training achieved its peak memory usage in the forward and the backward phases*. The forward phase generated lots of intermediate results. Some key intermediate results were cached for the gradient calculation in the backward phase, increasing memory usage. For example, Figure 14 shows the computation graph of the vertex updating function γ^l of GGNN. Each operator in the computation graph generated an intermediate tensor. Some key intermediate tensors are cached. The cached tensors were the main source of memory usage in the loss phase. By the end of the backward phase, the cached tensors were released. Since the evaluation phase did not have to calculate the gradients, it did not cache intermediate tensors. Its memory usage declined sharply.

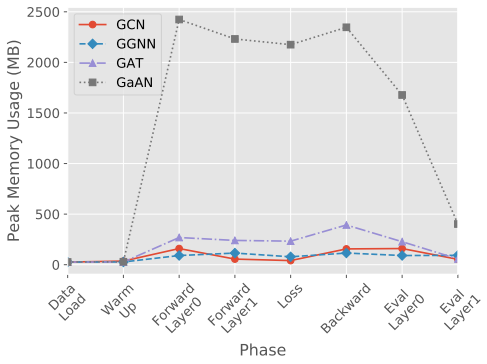


Figure 13: Memory usage of each phase during the GNN training. Dataset: `amp`.

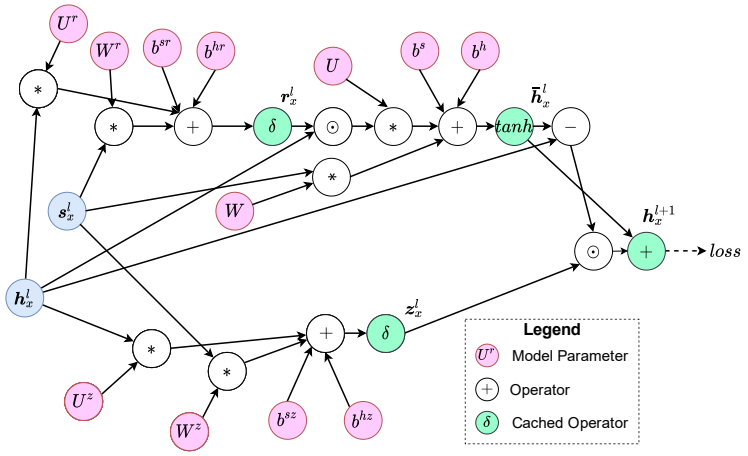


Figure 14: Computation graph of the vertex updating function γ of GGNN.

The peak memory usage during the GNN training far exceeded the size of the dataset itself. We defined 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 compares MER of different GNNs. GCN had the lowest MER (up to 15) while GaAN had the highest MER (up to 104). *The high MERs limited the data scalability of GNNs*, making GPUs unable to handle big graphs. Figure 15 also indicates that the same GNN had different MERs for different datasets. Two characteristics of a dataset affected the MER: the dimension of the input feature vectors and the average degree of the graph.

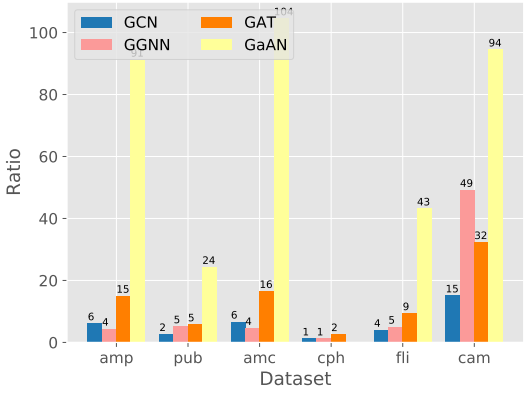


Figure 15: Memory expansion ratios of typical GNNs.

To find out how the dimension of input feature vectors affected the MER, we generated random input feature vectors with different dimensions for the `cam` dataset and measured the MER in Figure 16. Under the same hyper-parameters, *the MER decreased as the dimension of input feature vectors increased*. When the dimension of the input feature vectors was high, the size of the dataset itself was large. The size became comparable to the size of intermediate results, making MERs low.

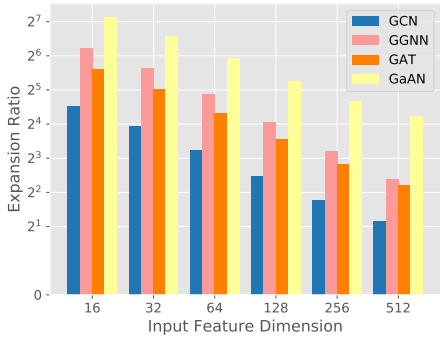
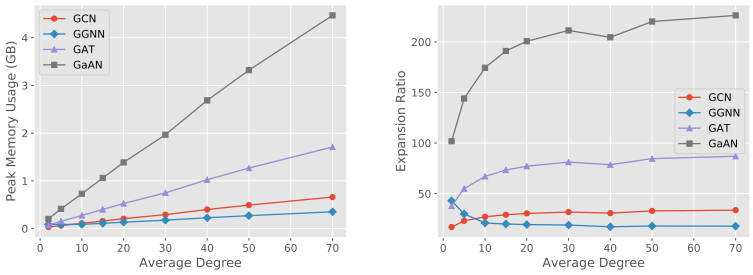


Figure 16: Memory expansion ratio under different dimensions of input feature vectors. Dataset: `cam`.

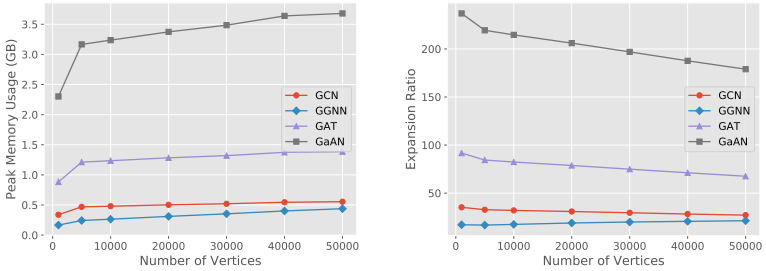
Average degrees also affected MERs by influencing the relative sizes of intermediate results from the edge and the vertex calculation stages. Fixing the number of vertices $|\mathcal{V}|$, we generated random graphs with different average degrees. Figure 17 shows how the memory usage changed according to the average degree. As the average degree $\bar{d}$ increased, the peak memory usage increased *linearly* with $\bar{d}$. The edge calculation stage gradually dominated the memory usage and *the MER converged to a stable value*. The stable value was determined by the complexity of the edge calculation stage. Except for GGNN, the MERs of the other GNNs increased as $\bar{d}$ increased. As GGNN had high vertex calculation complexity, the MERs related to the vertex calculation stage were much higher than the edge calculation stage. When the edge calculation stage dominated the memory usage, its MERs became smaller.



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

Figure 17: Memory usage under different average degrees. The random graphs were generated by fixing the number of vertices at 10K and the dimension of input feature vectors at 32.

We also fixed the number of edges $|\mathcal{E}|$ and generated random graphs with different $|\mathcal{V}|$. Figure 18 shows how the memory usage changed according to $|\mathcal{V}|$. MERs of all GNNs were insensitive to $|\mathcal{V}|$, compared to $|\mathcal{E}|$. Except for GGNN, the MERs of the other GNNs declined as $|\mathcal{V}|$ increased because the sizes of the datasets increased more quickly than the sizes of the intermediate results. As GGNN had high vertex calculation complexity, the sizes of the intermediate results were very sensitive to $|\mathcal{V}|$. It indicated that *the intermediate results of the edge calculation stage dominated the memory usage during the GNN training*.



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

Figure 18: Memory usage under different numbers of vertices. The random graphs were generated by fixing the number of edges at 500K and the dimension of input feature vectors at 32.

Summary of Memory Usage. The *high* memory expansion ratio severely restricted the data scalability of GNN training. The memory usage mainly came from the intermediate results of the *edge calculation stage*. Fixing the number of vertices, the memory usage increased *linearly* along with the number of edges. Fixing the GNN structure and the hyper-parameters, increasing the dimension of input feature vectors could reduce the memory expansion ratio. To reduce the memory usage of GNN training, optimizations should focus on reducing memory footprints of the edge calculation stage.

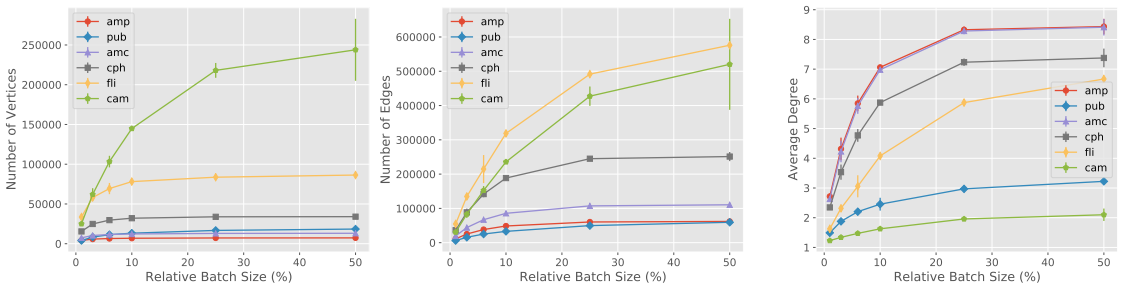
4.4. Effects of Sampling Techniques on Performance

With the sampling techniques, GNNs were trained in a mini-batch manner. Each mini-batch updated 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 both declined significantly.

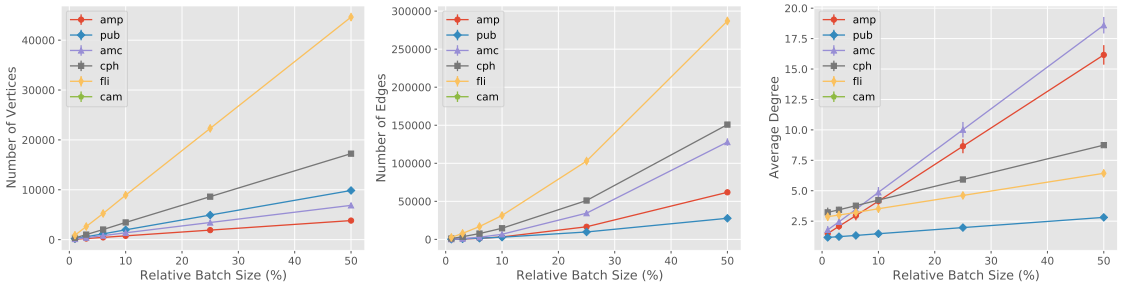
In the implementation in PyG, the GNN model and the dataset resided on the GPU side. To process each epoch, PyG sampled the original dataset in the main memory and generated several batches. Each batch was a small subgraph of the dataset. To train on each batch, PyG sent the sampled subgraph to the GPU, calculated the gradients on the subgraph, and updated the model parameters directly on the GPU. With the sampling techniques, the model parameters were updated by a stochastic gradient descent optimizer. PyG conducted the

evaluation phase every several epochs or batches (either on the CPU side or the GPU side) to determine whether to stop the training. In this section, the experiments focused on the training phase of each batch.

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



(a) Neighbor sampler



(b) Cluster sampler

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

The average degree of the sampled subgraph was *much lower* than the average degree of the original 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 was 31.1, but the average degree of the sampled subgraph was only 5.8. For the cluster sampler, the average degree was 3.0. Figure 20 compares the degree distribution of the sampled subgraphs with the original graph. The slopes of the curves were similar, indicating that the sampled subgraphs still followed the power-law degree distribution. However, the numbers of high-degree vertices were much less than the original graph, lowering the average degrees. According to the experimental results in Section 4.2, if the average degree became lower, the proportion of the training time spent on the vertex calculation stage would become higher, especially for GGNN.

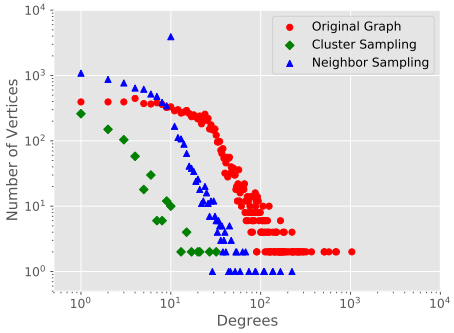
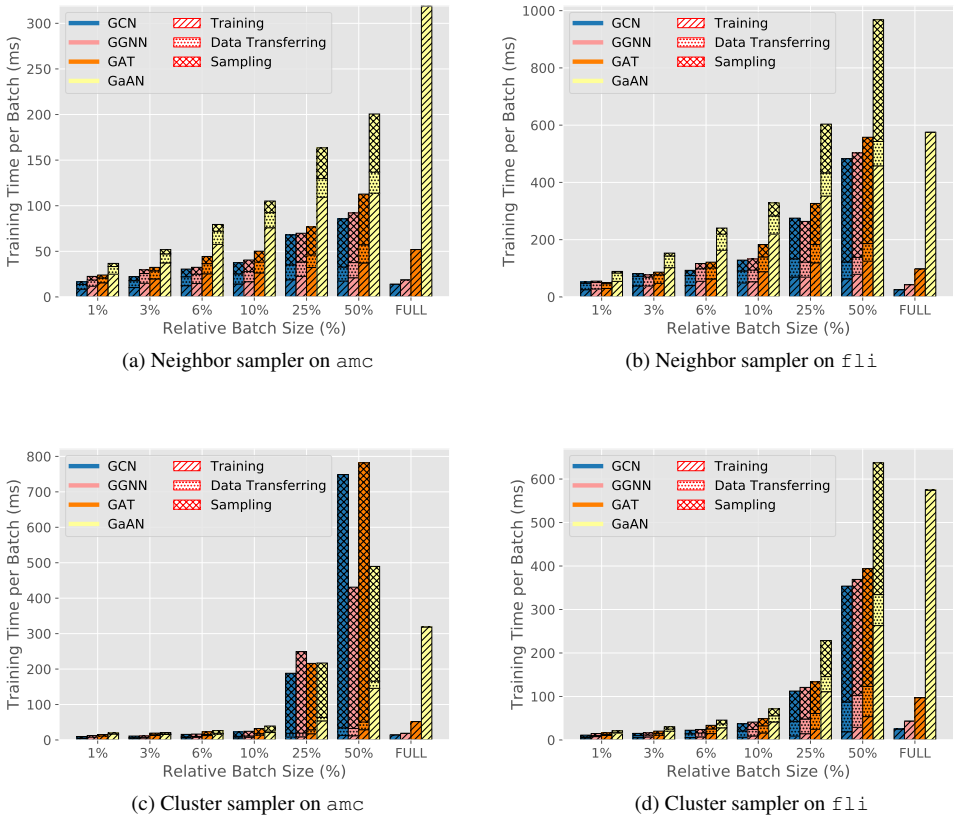


Figure 20: Vertex degree distribution of the sampled subgraph (relative batch size: 6%) and the original graph. Dataset:`amp`.

To find out performance bottlenecks of the sampling techniques, we decomposed the training time per batch into three phases: *sampling* on the CPU side, *transferring* sampled subgraphs from the CPU side to the GPU side, and *training* with the sampled subgraphs on the GPU side. Figure 21 shows the time breakdown of the four GNNs under different relative batch sizes. For the neighbor sampler, the sampling technique reduced the training time per batch only when the batch size was very small. When the batch became bigger, the sampling and the data transferring phases introduced noticeable overheads, making the training time exceed the full-batch training. For the clustering sampler, the sampled subgraph was smaller than the neighbor sampler under the same relative batch size. The reduction in the training time was more obvious than the neighbor sampler. However, the overheads increased quickly as the relative batch size increased. The training time under the 25% relative batch size already exceeded the time of full-batch training. The experimental results indicated that the current implementation of the sampling techniques in PyG was inefficient. When the batch size was large, more than 50% of the time had been spent on sampling and data transferring. *The sampling techniques were only efficient under small batch sizes.*



(a) Neighbor sampler on `amc`

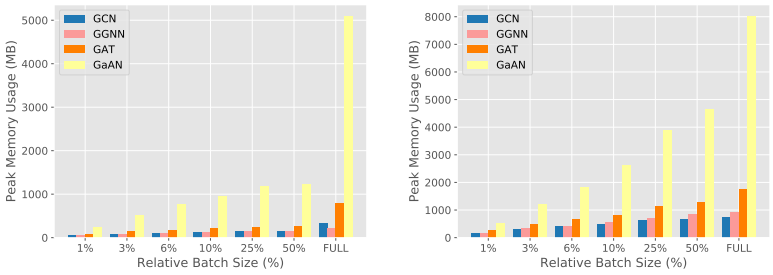
(b) Neighbor sampler on `fli`

(c) Cluster sampler on `amc`

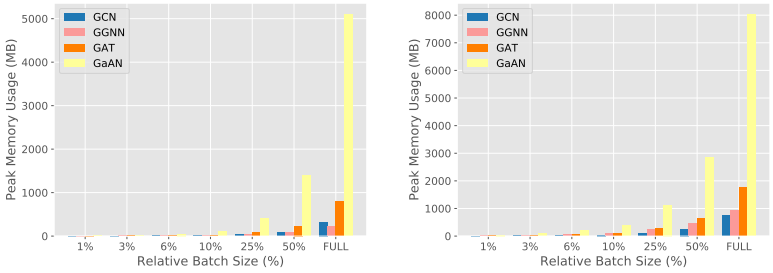
(d) Cluster sampler on `fli`

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

The main advantage of the sampling techniques was *reducing the peak memory usage* during training. Figure 22 shows the memory usage under different batch sizes. The peak memory usage declined significantly even under big batch sizes. The sampling techniques made training GNNs on big graphs *possible* for GPUs.



(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 that the full graph participated in the training.

The disadvantage of the sampling technique was wasting GPU resources. As the sampling techniques were only effective under small batch sizes, the sampled subgraphs were very small in those cases. They could not 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 batch in Figure 23. As the number of vertices increased, the training time was almost unchanged except for GaAN. The training time of GaAN increased only with $|\mathcal{V}| \geq 4000$.

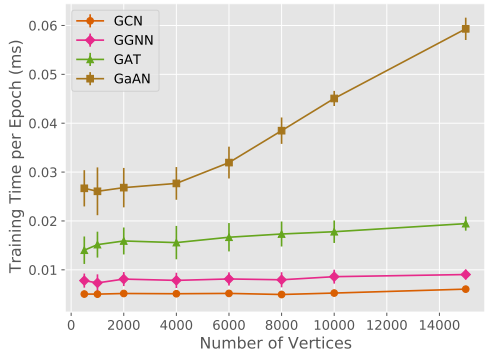


Figure 23: Training time per epoch on small random graphs. For each number of vertices, we generated 50 random graphs with the average degree of 4.0 and reported the average training time per batch (without the evaluation phase). The error bar indicates the standard deviation.

Summary of Sampling Techniques. The sampled subgraphs had lower average degrees than the original graph. With small batch sizes, the sampling techniques could significantly reduce the training time per batch and the peak memory usage. However, small batch sizes could not make full use of the computing power of a GPU. With big batch sizes, the current implementation of the sampling techniques in PyG was inefficient. The time spent on the sampling phase and the data transferring phase even exceeded 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 2 and Table 3 points out performance bottlenecks 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 the attention mechanism with specially optimized basic operators on the GPU side 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 [**?** **?** **?**] 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) [**?** **?**]. 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 [**?**]. Another possible solution [**?**] 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.[**?**], Zhang et al.[**?**] and Wu et al.[**?**] 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. [**?**] evaluate the accuracy of popular GNNs on the node classification task. Dwived et al. [**?**] further compare the accuracy of popular GNNs fairly in a controlled environment. Hu et al. [**?**] 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. [**?**] 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. [**?**] analyze the architectural characteristics of the GNN inference on GPUs under SAGA-NN [**?**] 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[**?** **?**] form a complementary study on the efficiency issue of GNNs.

Libraries/Systems of GNNs. PyG [**?**] and DGL [**?**] both adopt the message-passing model as the underlying programming model for GNNs and support training big datasets with the sampling techniques. PyG [**?**] is built upon PyTorch and it uses optimized CUDA kernels for GNNs to achieve high performance. DGL [**?**] 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 [**?**] 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 [**?**] 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 [**?**] 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 subgraphs 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.