Empirical Exploration of the Performance Bottleneck in Graph Neural Network Training

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

Abstract: The graph neural network (GNN) has become a research hotspot for its state-of-the-art performance in many graph analysis tasks. 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). 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 could focus on it. The sampling techniques are essential for training big graphs on GPU, but their current implementations still have room for improvement. Improving its efficiency is another potential optimization direction.

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

# Introduction

In recent years, the graph neural network (GNN) becomes a hot research topic in the field of artificial intelligence. Many GNNs [[1](#_bookmark46), [2](#_bookmark47), [3](#_bookmark48), [4](#_bookmark49), [5](#_bookmark50), [6](#_bookmark51), [7](#_bookmark52)] 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](#_bookmark53), [9](#_bookmark54), [10](#_bookmark55)], 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 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 GPU. 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 $\times$ {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 GPU. The experimental results verify the correctness of our time complexity analysis. 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 GPU. 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:

1. To reduce training time, optimizations should focus on improving the efficiency of the edge calculation. One may consider developing optimized operators for the attention mechanism 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.

2. 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.

3. 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.

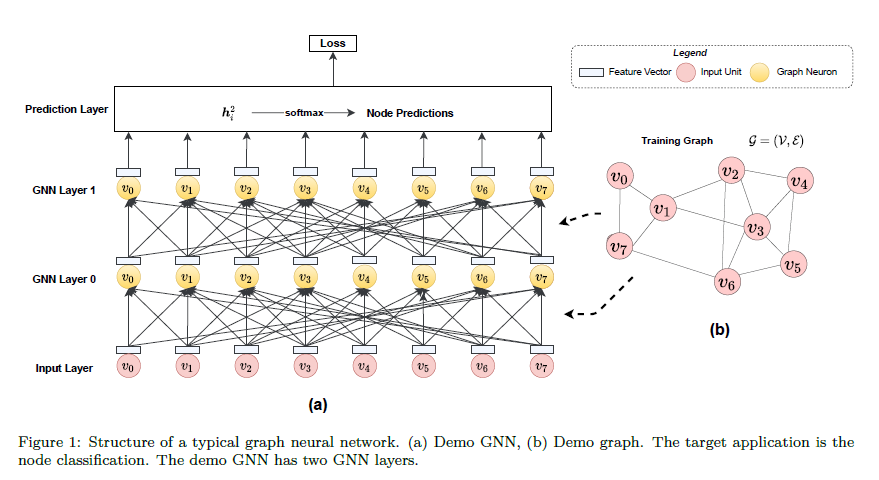
## Outline

We briefly survey the typical GNNs in Section [2](#sec:review_of_gnns). We introduce our experimental setting and targets in Section [3](#sec:experimental_design). The experimental results are presented and analyzed in Section [4](#sec:experiment_results). We summarize the key findings and give out potential optimization directions in Section [5](#sec:insights). We introduce the related work in Section [6](#sec:related_work) and finally conclude our work in Section [7](#sec:conclusion).

# 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 as , where and are the vertex set and the edge set of , respectively. Let and as the number of vertices/edges. We use to denote a vertex and to denote the edge pointing from to . The adjacency set of is . We denote a *vector* with a bold lower case letter like and a *matri*x with a bold upper case letter like .

## 2.1 Structure of Graph Neural Networks

As illustrated in Figure [1](#fig:general_structure_of_gnn" \t "_blank), a typical GNN can be decomposed into three parts: an input layer + several GNN layers + a prediction layer.

A GNN receives a graph as the input. Every vertex in is attached with a feature vector to describe the properties of the vertex. The edges of may also be attached with feature vectors The input layer of a GNN receives feature vectors from all vertices and passes them to GNN layers.

A GNN layer consists of graph neurons, where is the number of vertices in . Each graph neuron corresponds to a vertex in . In the first GNN layer (Layer 0), the graph neuron of the vertex collects input feature vectors of itself and the vertices that are adjacent to in (i.e., ) from the input layer. After aggregating input feature vectors and applying the non-linear transformation, the graph neuron outputs a hidden feature vector for . Take the demo GNN in Figure [1](#fig:general_structure_of_gnn)(a) as the example. Since , the graph neuron of at layer 0 collects the feature vectors {, , , , , } from the input layer and outputs . Different GNNs mainly differ in the graph neurons that they use. We elaborate on their details later.

The connection between the input layer and the first GNN layer is determined by the topology of . In the traditional neural networks, neurons of neighboring layers are fully connected. In GNNs, two graph neurons are connected only if their corresponding vertices have an edge between them in . Most real-world graphs are very *sparse*, i.e. .

In the next GNN layer (Layer 1), the graph neuron of collects the hidden feature vectors of itself and its neighbors ( with ) from the *previous* GNN layer. Based on the collected hidden vectors, the graph neuron in Layer 1 outputs a new hidden feature vector for . Though there are only two GNN layers in Figure [1](#fig:general_structure_of_gnn), a GNN allows us to stack more GNN layers to support deeper graph analysis.

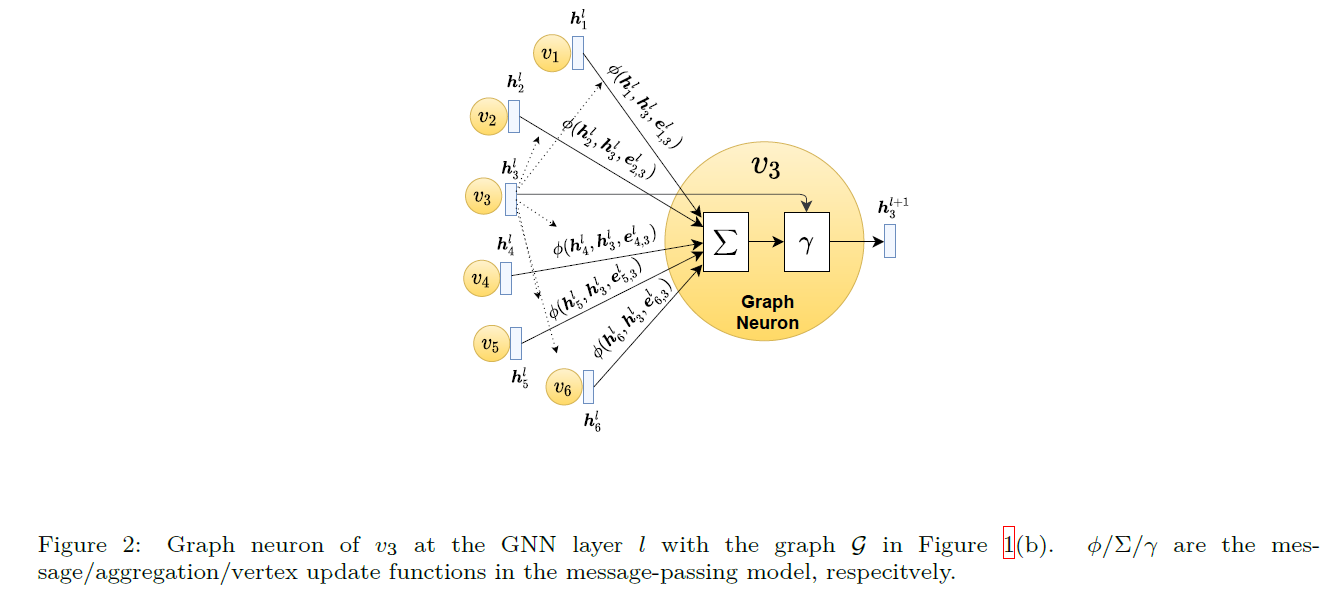
Assume there are GNN layers. The last GNN layer (Layer ) outputs a hidden feature vector for every vertex . As an embedding vector, encodes the knowledge learned from the input layer and all the previous GNN layers. Since is affected by and the vertices in the -hop neighborhood of , analyzing a graph with a *deeper* GNN means analyzing each vertex with a *wider* scope.

The hidden feature vectors 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 as the example, as shown in Figure [1](#fig:general_structure_of_gnn). The node classification predicts a label for every vertex in . In this case, the prediction layer can be a simple softmax layer with 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 other graph-based machine learning methods. For example, traditional graph embedding methods DeepWalk and node2vec mostly like intuition-based algorithms . We can calculate the gradients of the loss function on the model parameters from the prediction layer directly. With the help of the backpropagation technique, the gradient is propagated from the prediction layer back to the previous GNN layers layer by layer. The model parameters are updated with a gradient descent optimizer like Adam. Except for the input feature vector, there is no need to conduct handworked feature extraction. In a fully parameterized way, the GNN automatically extracts an embedding vector for each vertex from its -hop neighborhood. The parameters are tuned according to the specific prediction task, leading to high prediction accuracy.

## 2.2 Graph Neuron and Message-passing Model

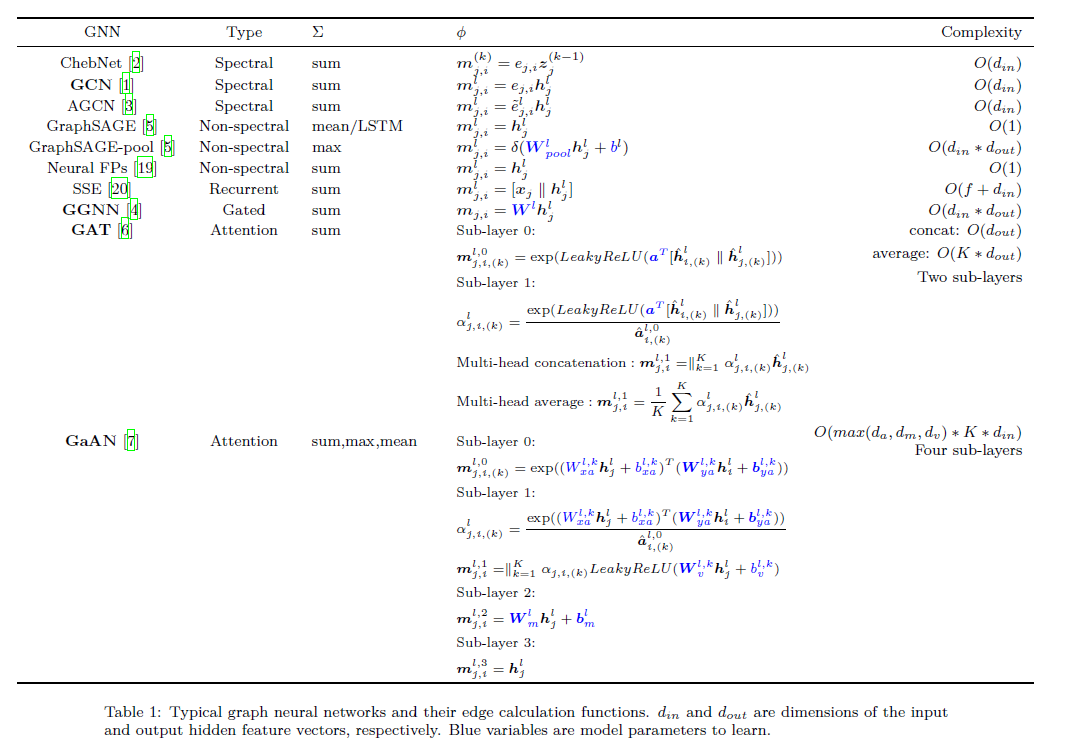
Graph neurons are building blocks of a GNN. A GNN layer consists of graph neurons. Each vertex corresponds to a graph neuron. A graph neuron as shown in Figure [2](#fig:graph_neuron_structure) is a small neural network. The graph neuron of at layer receives hidden feature vectors from the graph neurons of and its neighbors () at the previous GNN layer [[1]](#footnote-0). The graph neuron aggregates the received hidden feature vectors, applies non-linear transformations, and outputs a new hidden feature vector .



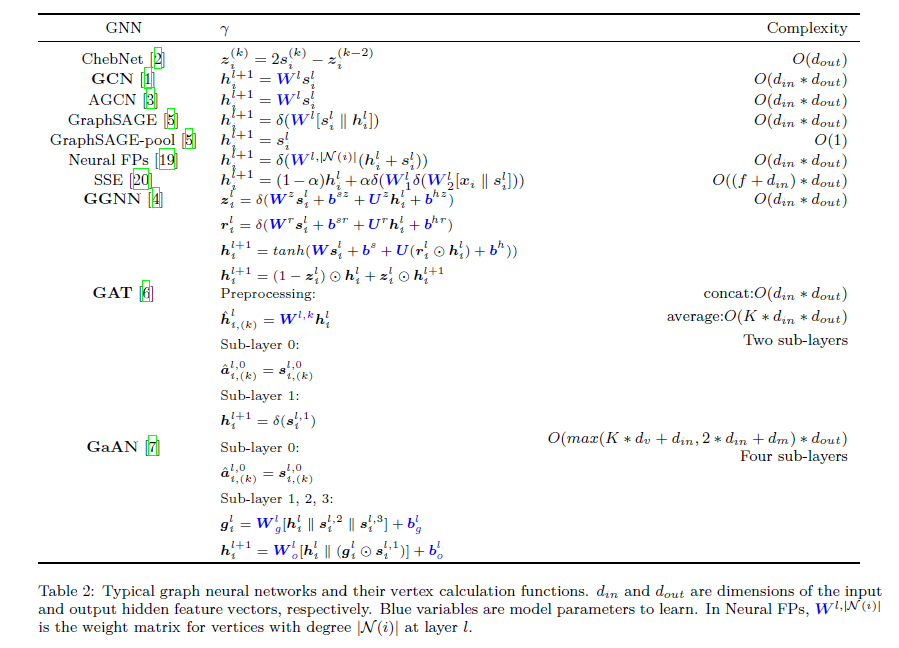
We follow the message-passing model [18] to formally define a graph neuron. The message-passing model is widely used in the cutting-edge GNN training systems like PyTorch Geometric (PyG) [11] and Deep Graph Library (DGL) [12]. Figure [2](#fig:graph_neuron_structure) shows the structure of a graph neuron in the message-passing model. A graph neuron collects messages from its neighbor graph neurons and outputs a hidden feature vector according to the aggregated messages. Graph neurons at layer are made of three *differentiable* functions: , and . The graph neuron calculates the output hidden vector by



is the *message* function. For every incident edge of , receives the output hidden feature vectors and of the previous GNN layer and the edge feature vector as the input. emits a message vector for every edge at layer , i.e., . For , the message vectors of its adjacent edges () are aggregated by the *aggregation* function to produce an aggregated vector , i.e., . ’s aggregated vector and its hidden vector from the previous GNN layer are fed into the *vertex update* function to calculate the output hidden vector of the current layer , i.e., The end-to-end training requires and (like multi-layer perceptrons and GRU) and (like mean, sum, element-wise min/max) are *differentiable* to make the whole GNN differentiable.



Different GNNs adopt different kinds of graph neurons and have different definitions of the three functions. and are the *edge calculation* functions. They are conducted over every edge in . is the *vertex calculation* function. It is conducted over every vertex in . Table [1](#tab:gnn_overview_edge) and Table [2](#tab:gnn_overview_vertex) 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 GNN sub-layers and a summation layer:  with the GNN sub-layers , , and , where is the matrix of output hidden feature vectors of the layer . 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 , and the other three layers calculate //.

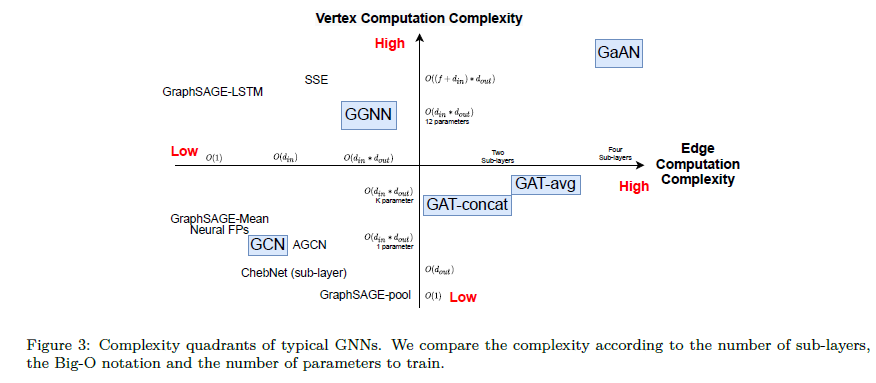


## 2.3 Classification of GNNs

Since we focus on analyzing the performance bottleneck in training GNNs, we classify the typical GNNs from the view of time complexity. We use // 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 + and the vertex calculation complexity .

In Table [1](#tab:gnn_overview_edge) and Table [2](#tab:gnn_overview_vertex), 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 and and the dimensions of the model parameters (like the number of heads in GAT and the dimensions of the view vectors / in GaAN).

We classify the typical GNNs into four quadrants based on their edge/vertex complexity as shown in Figure [3](#fig:gnn_complexity_quadrant). 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) is 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 in . A GCN graph neuron can be expressed as , where is the normalized weight of the edge . According to the associative law of the matrix multiplication, . Since the dimension of is usually smaller than in practical GCNs, the implementation of GCN in PyTorch Geometric chooses to first conduct the vertex calculation for each vertex and then conduct the edge calculation . As has the same dimension as , the implementation significantly reduces the computation cost of the edge calculation.

**GGNN** (high vertex & low edge calculation complexity): GGNN introduces the gated recurrent unit (GRU) into the graph neural networks. The vertex update function 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 is equal to the dimension of . Since the message function only uses the hidden feature vector of the source vertex of an edge , in the implementation, GGNN conducts the pre-processing vertex calculation for every vertex before the message-passing. The message function directly uses as the message vector for the edge . In this way, GGNN further reduces the time complexity of the edge calculation to without increasing the time complexity of the vertex calculation.

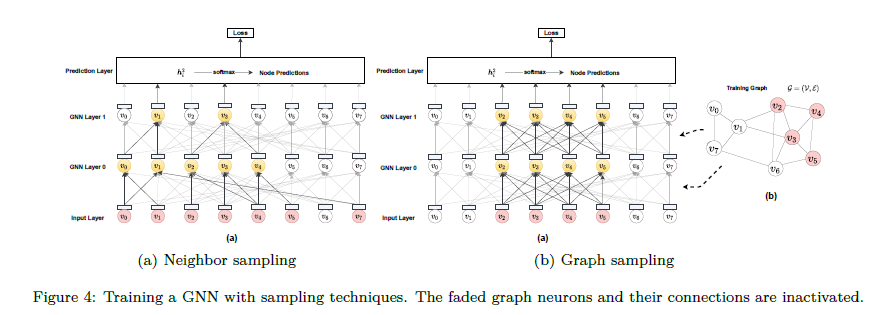
**GAT** (low vertex & high edge calculation complexity): GAT introduces the attention and multi-head mechanism into the graph neural networks. The heads generate independent views for an edge, where is a hyper-parameter. The views of heads can be merged by concatenating or by averaging. For concatenating, the dimension of the hidden feature vector of each head is . For averaging, is , multiplying the complexity by . Each GAT layer consists of a preprocessing step and two sub-layers. In the preprocessing step, GAT calculates the attention vectors for every vertex in every head . The first sub-layer uses the attention vectors to calculate the attention weights of every edge in every head . After the message-passing, the first-layer gets the summation of attention weights of every vertex in every head . The second sub-layer gets the normalized attention weights 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 of the second sub-layer as the output hidden vector directly.

**GaAN** (high vertex & edge calculation complexity): Based on the multi-head mechanism, GaAN introduces a convolutional subnetwork to control the weight of each head. Different from to multi-head mechanism in GAT, GaAN project the center vertex feature to the query vector (dimension is ), and the neighboring vertex features are projected to get key and value vectors. The dimension of key and values vectors are the same, we set to . is the number of attention heads. The weight of each head is calculated by a convolutional network, which takes the feature of each vertex and its neighbors as input to get the gated values with a dimension of . The convolutional network is built by combining average pooling and max pooling. Moreover, the max-pooling needs to project the vertex feature to a transformed vector with a dimension of , costing an additional computational overhead. Each GaAN layer consists of four sub-layers. The first sub-layer aims to get the summation of attention weights of every vertex in each head. The second sub-layer uses to aggregate neighboring vertices’ query vector with the normalized attention weights for every head. And the third sub-layer is used in max-pooling and it gets the element-wise max vector of neighboring vertices’ transformed vectors by message-passing. The last sub-layer gets element-wise mean vector of neighboring vertices’ vectors by message-passing. The gate values is obtained by using a fully-connected layer with vector concatenating , and . At last, GaAN uses a fully-connected layer with concatenating and the result of element-wise multiplying and to get the output hidden vector.

## 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 [21]. 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, several sampling techniques are proposed. In each mini-batch, they sample a small subgraph from the whole graph and uses the subgraph to update the model parameters. In other words, 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](#fig:gnn_sampling). Inactive 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. Based on whether different GNN layers sample different subgraphs, the existing sampling techniques can be divided into two groups [22]: the neighbor sampling and the graph sampling.



The neighbor sampling techniques [5,23,24,25,26] sample the subgraphs layer by layer. The first sample several vertices from in the last GNN layer. Then they repeatedly sample the neighbors of those vertices in the previous layer until the input layer. The sampled subgraphs of different GNN layers may be *different*, as shown in Figure [4(a)](#fig:gnn_sampling_neighbor_sampling). GraphSAGE [5] is the representative technique. For every sampled vertex in the GNN layer , GraphSAGE samples at most neighbors of in the previous GNN layer. is the hyperparameter that is usually much smaller than . In this way, GraphSAGE limits the neighborhood sizes of the vertices in the sampled subgraph, especially high-degree vertices.

The graph sampling techniques[21,22,27] sample a subgraph for each mini-batch and use the same sampled graph for all GNN layers, as shown in Figure [4(b)](#fig:gnn_sampling_graph_sampling). They differ in the methods to sample subgraphs. The cluster sampler technique [21] is the representative technique. Given a training graph , it partitions into closely connected clusters. For each mini-batch, it randomly picks clusters, where is the hyperparameter. The technique keeps the inter-cluster edges and intra-cluster edges in the subgraph.

# 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](#sec:experimental_env) and then give out our experimental scheme in Section [3.2](#sec:experimental_scheme). The evaluation results are presented and analyzed later in Section [4](#sec:experiment_results).

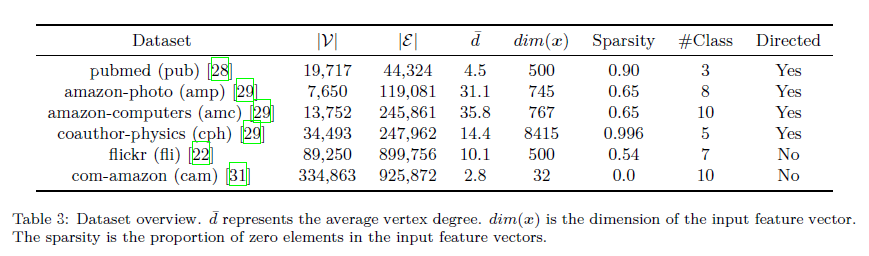
## 3.1 Experimental Setting

### 3.1.1 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 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 the GNNs with PyTorch Geometric 1.5.0.

### 3.1.2 Dataset

We used six real-world graph datasets as listed in Table [3](#tab:dataset_overview) that were popular in the GNN accuracy evaluation[22,28,29] . For directed graphs, PyG converts them into undirected ones during the data loading. Thus, the average degree of a directed graph . For an undirected graph, already contains two-direction edges and . For the cam dataset, we generated random dense feature vectors. Since we mainly focus on the training efficiency of GNNs instead of accuracies, we also used random graphs in the experiments. To evaluate the effects of graph topological characteristics (like the average degree) on the performance bottleneck uniformly, we used the R-MAT graph generator [30]. Input feature vectors of the random graphs were random dense vectors with a dimension of 32. Vertices were divided into 10 classes randomly.



### 3.1.3 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 check the accuracy of the current parameters.

### 3.1.4 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 perceptrons(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 in the memory of GPU. All the training was conducted on GPU.

### 3.1.5 Hyper-parameters

We use to denote the dimension of a vector . We picked the hyper-parameters of GNNs according to their original papers. As GNNs used different hyper-parameters for different datasets, we picked the most popular hyper-parameters and used the same set of hyper-parameters for all the datasets in our experiments. Some hyper-parameters like the dimensions of hidden feature vectors were common. We set them to the same values in the four GNNs. For GCN/GAT/GaAN, we set , , and . For GAT, we set the hyper-parameters according to [6]. The first GAT layer had 8 heads with dimension and merged the heads by concatenating. The second GAT layer used a single head with dimension . For GGNN, since it uses extra MLP layers to transform the dimensions of the input/output feature vectors, we set . We used 8 heads in the both GaAN layers with and .

### 3.1.6 Sampling Techniques

We picked the neighbor sampler from GraphSAGE [5] and the cluster sampler from ClusterGCN [21] as the typical neighbor sampling and graph sampling techniques, respectively. For the neighbor sampler, we used the neighborhood sample sizes (25 for GNN Layer 1, and 10 for GNN Layer 0) and the default batch size (512) from [5]. For the cluster sampler, we partitioned every input graph into 1500 partitions and used 20 partitions per batch according to [21].

## 3.2 Experimental Scheme

To find out the performance bottleneck in GNN training, we conduct the bottleneck analysis with four questions. The answers to those questions will give us a more comprehensive view of the performance characteristics of GNN training. We design extensive experiments to find the answers empirically.

* *How do the hyper-parameters affect the training time and the memory usage of a GNN?* (Section [4.1](#X4696a7fab003a5a9bc4e724e896e90d9c382ec1))
* Every GNN has a group of hyper-parameters like the number of GNN layers and the dimension 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 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](#tab:gnn_overview_edge) and Table [2](#tab:gnn_overview_vertex). If the complexity analysis is valid, we can analyze the bottleneck theoretically.
* *Which stage is the most time-consuming stage in GNN training?* (Section [4.2](#sec:training_time_breakdown))
* We can decompose the training time on different levels: layer level, edge/vertex computation level, and the basic operator level. On each level, we breakdown 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.
* *Which consumes most of memory in GNN training?* (Section [4.3](#sec:memory_usage_analysis))
* The limited memory capacity of a GPU card is the main factor preventing us from training GNNs on big graphs. We measured the peak memory usage in the GNN training under different graph scales, input feature dimensions, and average vertex degrees. Based on the results, we analyze which is the most memory-consuming component. Reducing its memory usage will enable us to train bigger GNNs under the same memory capacity.
* *Can the sampling techniques remove the performance bottleneck in GNN training?* (Section [4.4](#Xdb0ebba72c419e0ce21d0399ff0205a76eed932))
* In theory, the sampling techniques can significantly reduce the 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.

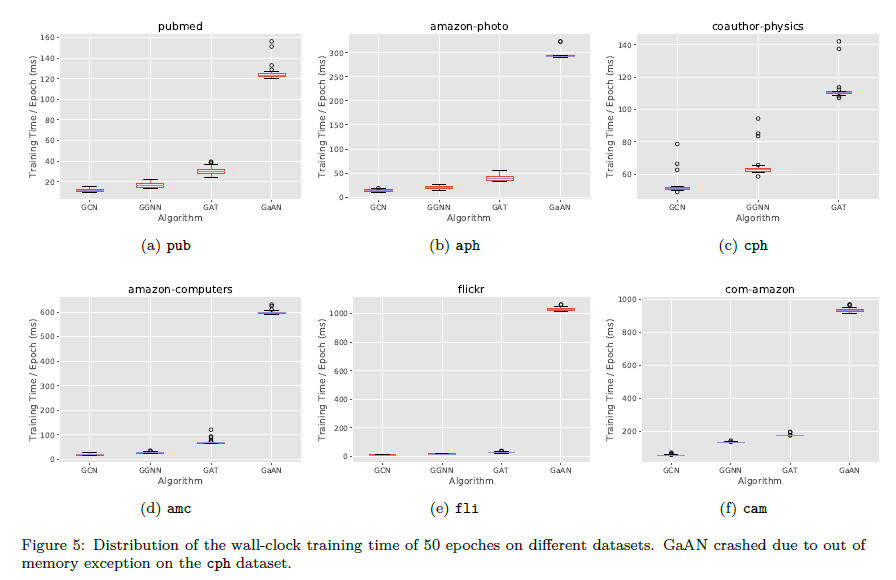
# Evaluation Results and Analysis

We answer the four questions in Section [3.2](#sec:experimental_scheme) 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 [[2]](#footnote-1).

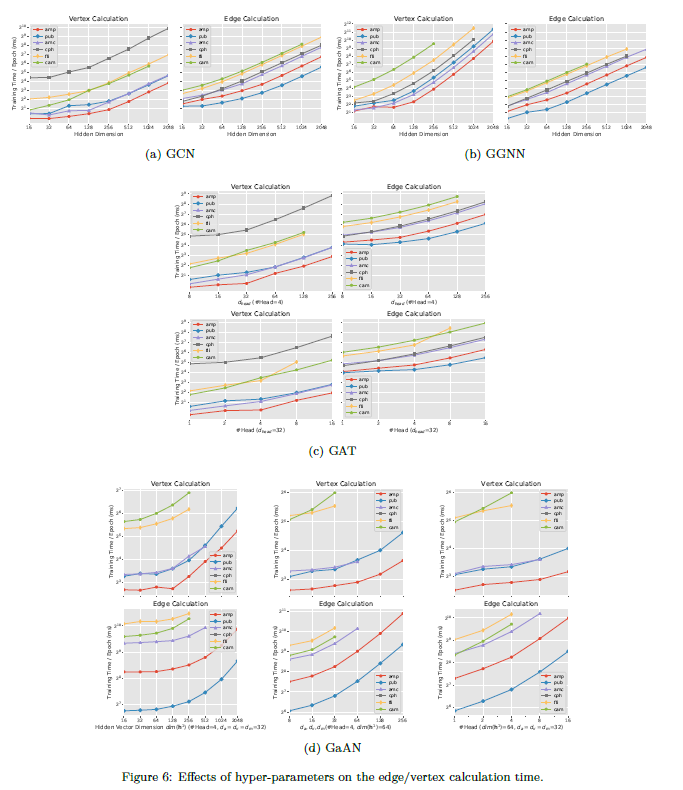
## 4.1 Effects of Hyper-parameters on Performance

Theoretically, the hyper-parameters should affect the training time in a *linear* way. According to Table [1](#tab:gnn_overview_edge) and Table [2](#tab:gnn_overview_vertex), the time complexities of and 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](#tab:gnn_overview_edge) and Table [2](#tab:gnn_overview_vertex), we first compare the training time of the four GNNs on the same dataset. Figure [5](#fig:exp_absolute_training_time) shows the wall-clock training time per epoch of the GNNs on the real-world datasets. The ranking of the training time was GaAN GAT GGNN GCN in all cases. Since the real-world graphs had more edges than edges (), the time complexity of the edge computation affected more than the vertex computation. The ranking was consistent with the time complexity analysis.



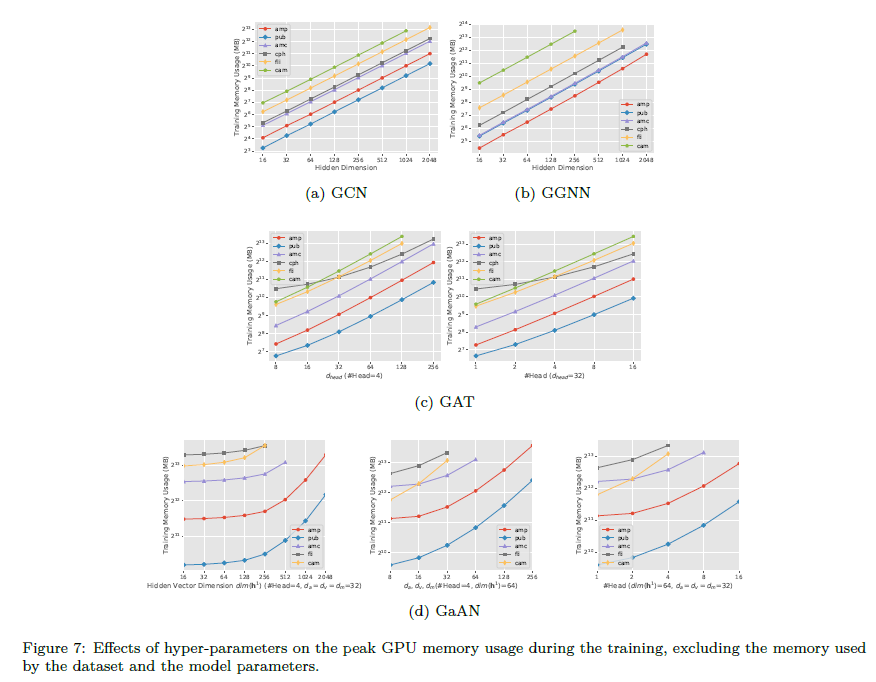
To further evaluate the effects of hyper-parameters, we measured the training time of each GNN with varying hyper-parameters in Figure [6](#X911015107f1ce458c9c61873a94aa2890cea07d).



For GCN and GGNN, the only modifiable hyper-parameter is the hidden dimension with . and is determined by the dataset with and #Classes. Figure [6(a)](#_4.1_Effects_of) and Figure [6(b)](#Xd2ed27b4f59a267ad44923915689f39b5cfc83b) show that the training time of GCN and GGNN increased linearly under big , consistent with the time complexity analysis.

For GAT, we modified the number of heads and the dimension of each head in the GAT layer 0. The dimension of the hidden feature vector was determined correspondingly as . Figure [6(c)](#X3608bded97b6978c6cbe9e1e8fb7d5480b1dbfc) shows that the GAT training time increases linearly under big and .

For GaAN, it is also based on the multi-head mechanism. Its time complexity is affected by , , and the number of heads . Figure 6(d) demonstrates that the training time increased linearly with the hyper-parameters, except for . As increased, the training time increased first slightly and then linearly. We could observe similar phenomena in GCN, GGNN, and GAT with low hyper-parameters. When the hyper-parameter was too low, the GNN training could not make use of the full computing power of the GPU. When it became high enough, the training time increased linearly.



The experimental results support 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 . As the hidden feature vectors consumed a small proportion of memory in GaAN, the growth in the memory usage was not noticeable until was large enough.

### Summary

The complexity analysis in Table [1](#tab:gnn_overview_edge) and Table [2](#tab:gnn_overview_vertex) is valid. The hyper-parameters affect the training time and the memory usage of the GNN training **in a linear way**. Algorithm engineers can use larger hyper-parameters to increase the expressive power of a GNN model without worrying about the 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 breakdown the training time and analyze the performance bottleneck level by level.

### 4.2.1 Layer Level

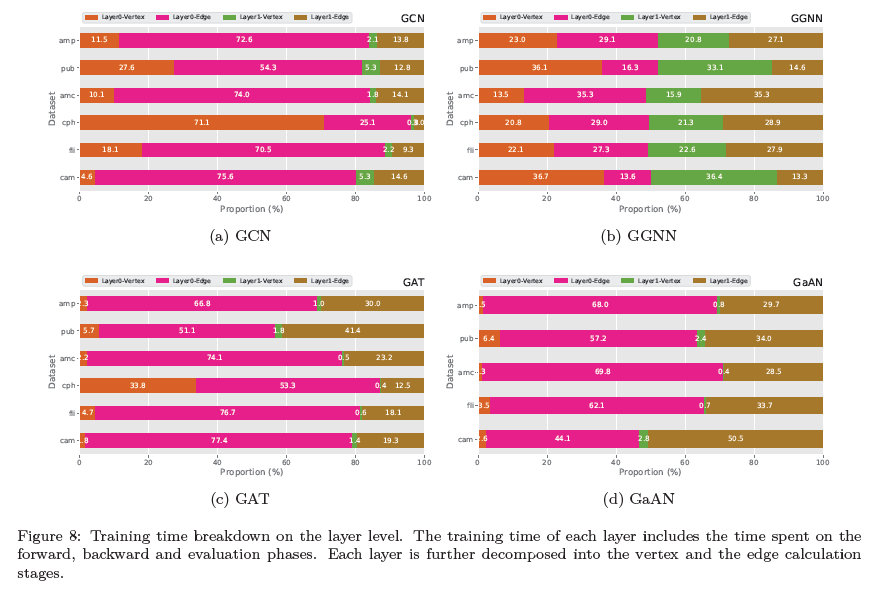
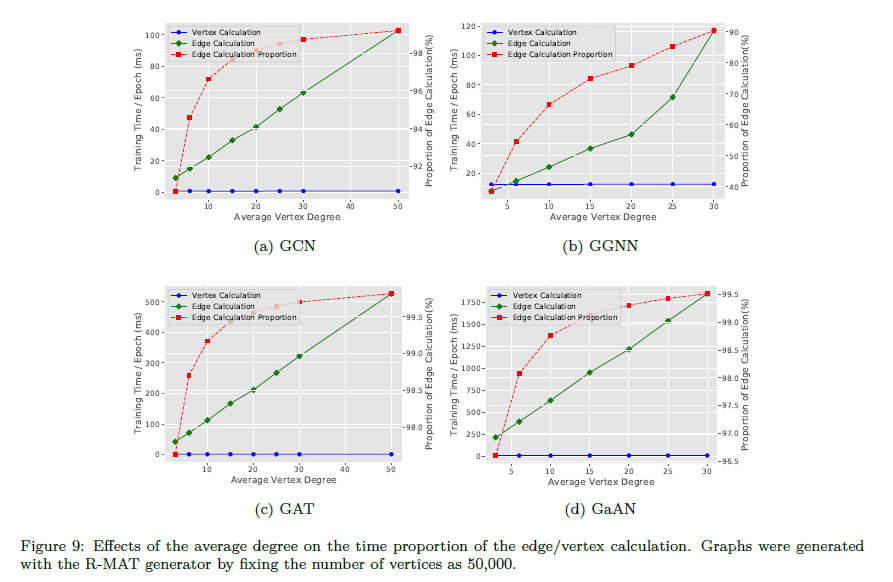


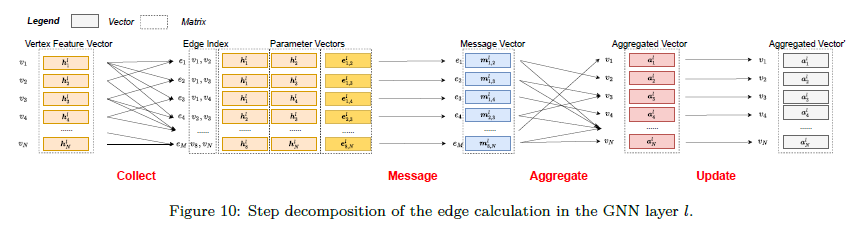
Figure 8 decomposes the training time of a GNN on the layer level. The training time of each layer is the summation of the time in the forward, backward, and evaluation phases. In GCN, GAT, and GaAN, the time spent on layer 0 was much larger than layer 1. In those GNNs, the dimensions of the input/output feature vectors in the layer 0 were much larger than the dimensions in the layer 1. , and and . For GaAN, since it required the dimensions of the input/output feature vectors must be the same, and the training time of both layers were close.

Each GNN layer can be further divided into the vertex and the edge calculation stages. In Figure 8, GCN spent most of the training time on the edge calculation in most datasets. A special case is 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. But the high time complexity of its vertex update function made the ratio of the vertex calculation in the total training time much higher than other GNNs. For GAT and GaAN, due to their high edge calculation complexity, the edge calculation is the absolutely dominant stage. In summary, *the edge calculation is the most time-consuming stage in the GNN training*.

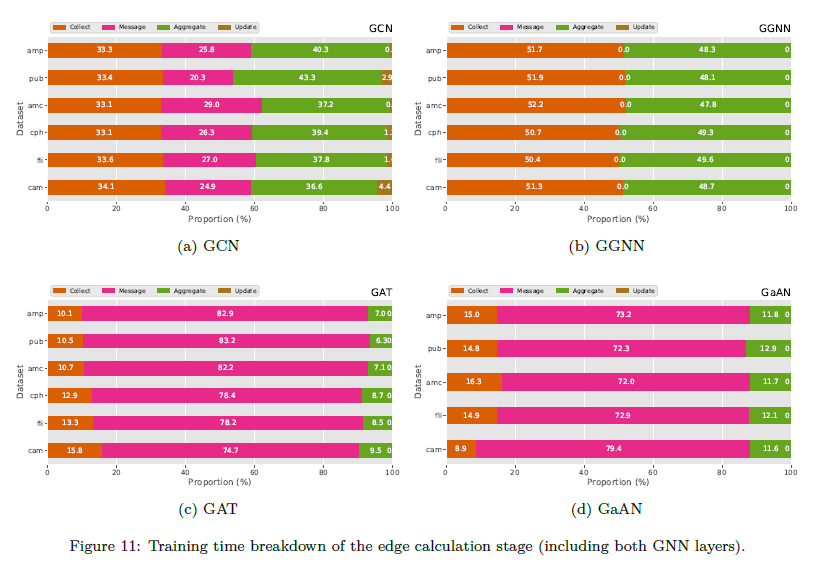


The experimental results also indicate that the average degree of the dataset affects the time-consuming proportion of the edge/vertex calculation. For GaAN, the time spent on the vertex calculation exceeded the edge calculation on the pub and cam datasets, because the average degrees of the two datasets were low, making and much closer. To evaluate the effects of the average degree, we used the R-MAT model 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 increased, the training time of the edge calculation grew *linearly*. For GCN, GAT, and GaAN, the edge calculation dominated the entire training time even under small average degrees. Only for GGNN that had high vertex and low edge calculation complexities, the training time of the vertex calculation could exceed the edge calculation under low average degrees (). Therefore, *improving the efficiency of the edge calculation is the key to reduce the GNN training time*.

### 4.2.2 Step Level in Edge Calculation



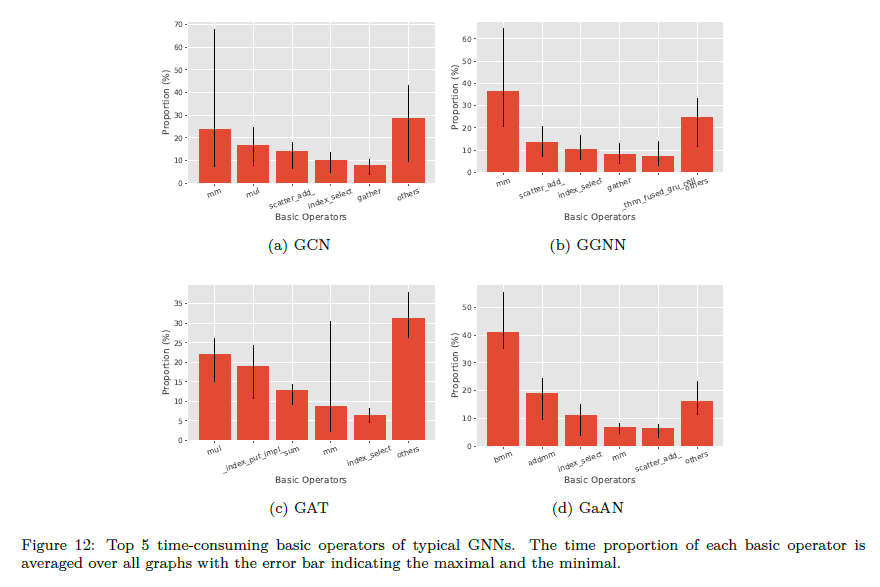
In the implementation of PyG, the edge calculation stage can be decomposed into four steps: collect, message, aggregate, and update, as shown in Figure 10. The edge index is a matrix with rows and 2 columns that holds the edge set of the graph, where . The two columns of the matrix store the source vertex and the target vertex of each edge, respectively. the collection step copies the vertex feature vectors from the previous layer to the ends of each edge in the edge index, to form the parameters of the message function . This step only involves the data movement. the messaging step calls the message function to get message vectors of all edges . The aggregate step aggregates the message vectors with the same target vertex into an aggregated vector with the aggregation operation . the updating step is optional. It performs an additional transformation on the aggregated vectors (for example, adding bias in GCN). The aggregated vectors (after the updating step) will be fed into the vertex update function as one of the input parameters.



We decomposed the execution time of the edge calculation step in Figure 11. In each GNN, the proportion 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 execution time. For GCN and GGNN with the low edge complexity, the proportions of the steps were close. Since the message function of GGNN used the pre-computed as the message vector directly in the implementation, the time spent on the messaging step of GGNN was negligible. Although the collection step did not conduct any computation and only involved data movement, it occupied noticeable execution time in all the GNNs. The experiments show that *the performance bottleneck on the step level depends on the complexity of the edge calculation*. For GNNs with the high edge calculation complexity, the message function is the performance bottleneck. Optimizing its implementation can significantly reduce the training time. For the other GNNs, optimization should focus on reducing the costs of the collect and the aggregation steps. Additionally, improving the efficiency of the collection step can benefit all GNNs.

### 4.2.3 Operator Level

The functions , and in the vertex and edge calculation are made up of a series of basic operators implemented on 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 was the matrix multiplication mm used in the vertex update function . The elementwise multiplication mul used in the message function was also time-consuming. The other three operators were used in the edge calculation: 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 consumed the majority of the training time.

#### GGNN

The top basic operator was mm used in the vertex calculation. Due to the high time complexity in , the proportion of the mm operator was much higher than the other operators. The thnn\_fused\_gru\_cell operator that was used in the backward phase of was also noticeable. The other three operators were used in the edge calculation.

#### GAT

All the top basic operators except for mm were related to the edge calculation. The mm operator was used in the vertex update function .

#### GaAN

The top basic operator was bmm used in the message function . The addmm operator and the mm operator were used in both the vertex and the edge calculation, where the edge calculation is dominant.

In general, the most time-consuming operator in GNN calculation is still the matrix multiplication mm and the elementwise multiplication mul, *making GNN training suitable for GPU*. Although the aggregate step in the edge calculation is relatively simple (like sum and mean), the related operators scatter\_add and gather still consumed a certain amount of the time. They have to synchronize between hardware threads to avoid updating the same aggregated vector at the same time and they conducted non-regular memory access with the access pattern determined by the edge set dynamically. For GPU, they were less efficient than mm. The index-based selection index\_select operator from the collection step consumed 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 (like overlapping them with ) can benefit the training of all kinds of GNNs.

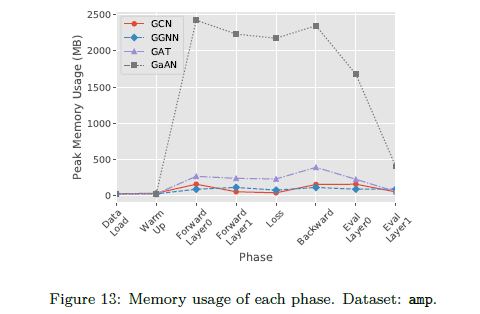
#### Summary

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

* If the time complexity of is **high**, the **efficiency of**  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 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 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 scale of graphs that a GPU can train GNNs on*. For example, GaAN was unable to complete the training due to memory overflow exception on cph dataset.



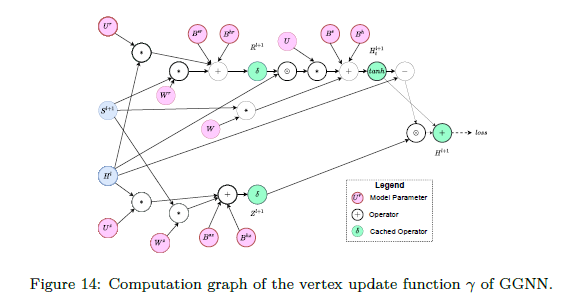


Figure 13 shows the peak memory usage of each phase in the GNN training on the amp dataset. The trend was similar to other datasets. *The GNN training achieved its peak memory usage in the forward and the backward phases*. The forward phase generated a large number of intermediate results. Some key intermediate results were cached, increasing memory usage. The cached results were used in the gradient calculation in the backward phase. Figure 14 shows the computation graph of the vertex update function of GGNN. The computation graph had a large number of operators, each operator generating an intermediate matrix. Some key intermediate matrices were cached. The cached matrices were the main source of memory usage in the loss phase. By the end of the backward phase, the cached results were released. Since the evaluation phase did not need to calculate the gradients, it did not cache intermediate results. Its memory usage declined 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 compares MER of different GNNs. GCN had the lowest MER (up to 14) while GaAN had the highest MER (up to 101). *The high MER limits the data scalability of GNNs*, making GPU unable to handle big graphs.

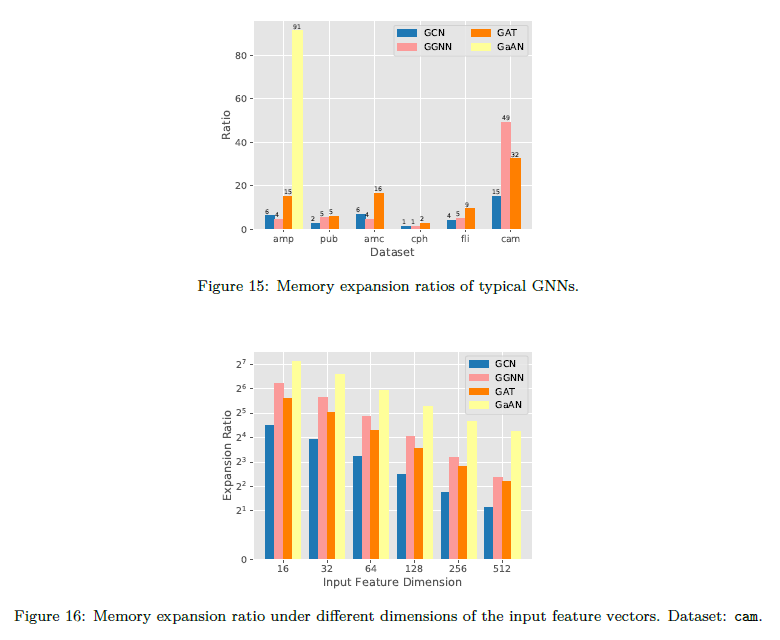
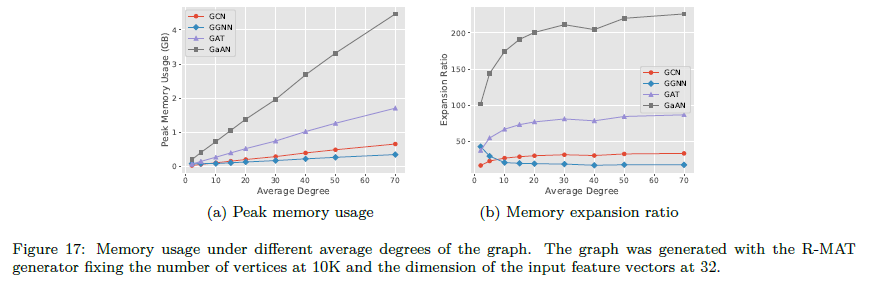
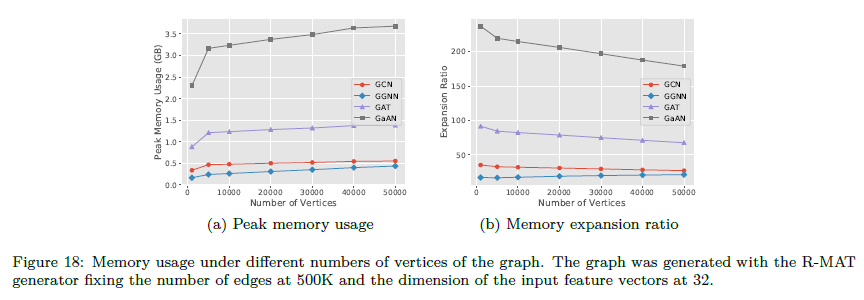


Figure 15 also indicates that the same GNN had different MERs for different datasets. Two characteristics of the dataset affect the MER: the dimension of the input feature vector and the average degree.

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 becomes 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 decreased as the dimension of the input feature vectors increased*.

The average degree of the graph also affects MER by influencing the relative scale of the intermediate results from the edge/vertex calculation. Fixing the number of vertices , 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 increased, the number of edges increased and the peak memory usage increased *linearly* with . The edge calculation 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. Except for GGNN, the MER of the other GNNs increased as increased. As GGNN had high vertex calculation complexity, the MER related to the vertex calculation was much larger than the edge calculation. When the edge calculation dominated the memory usage, its MER became smaller.





We also fixed the number of edges in the graph and generated random graphs with different . Figure 18 shows how the memory usage changed according to . All GNNs were much more insensitive to the changes in compared to . Except for GGNN, the MER of the other GNNs declined as increased because the scale of the dataset increased more quickly than the scales of the intermediate results. As GGNN had high vertex calculation complexity, the scales of the intermediate results were much more sensitive to . It indicates that *the intermediate results of the edge calculation dominated the memory usage during the GNN training*.

### Summary

The *high* memory expansion ratio severely restricted the data scalability of the GNN training. The memory usage mainly came from the intermediate results of the *edge calculation*. Fixing the number of vertices, the memory usage increased *linearly* along with the number of edges. Optimizing the memory usage of the edge calculation could significantly reduce the memory expansion ratio. Fixing the GNN structure and the hyper-parameters, increasing the dimension of the input feature vectors could also reduce the memory expansion ratio.

## 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 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 GPU, calculated the gradients in the subgraph, and updated the model parameters directly on GPU. With the sampling techniques, the model parameters were updated by a stochastic gradient descent optimizer. It conducted the evaluation phase every several epochs (either on the CPU side or the GPU side) to determine whether to stop the training. The experiments focused on the training phase of each batch.

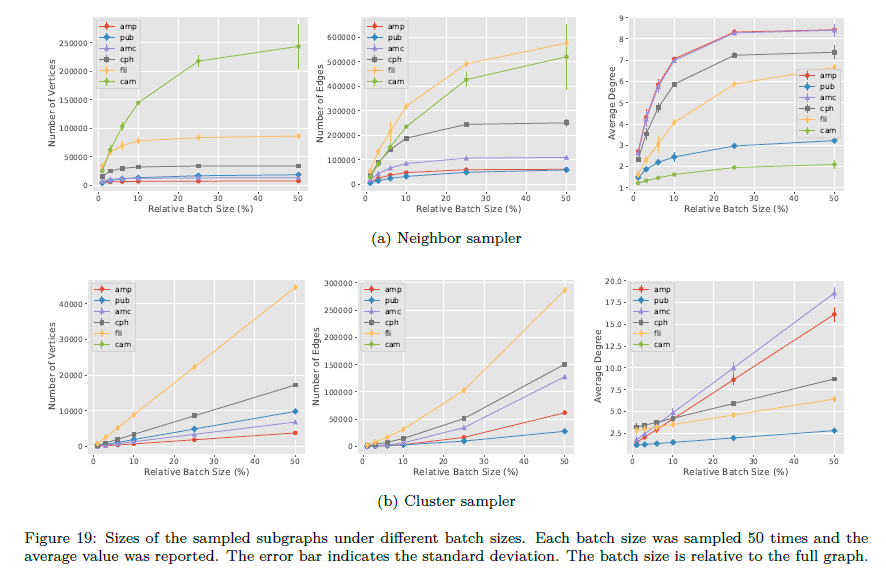
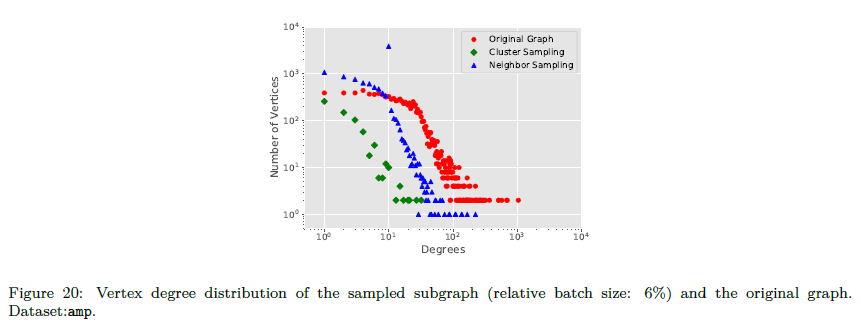
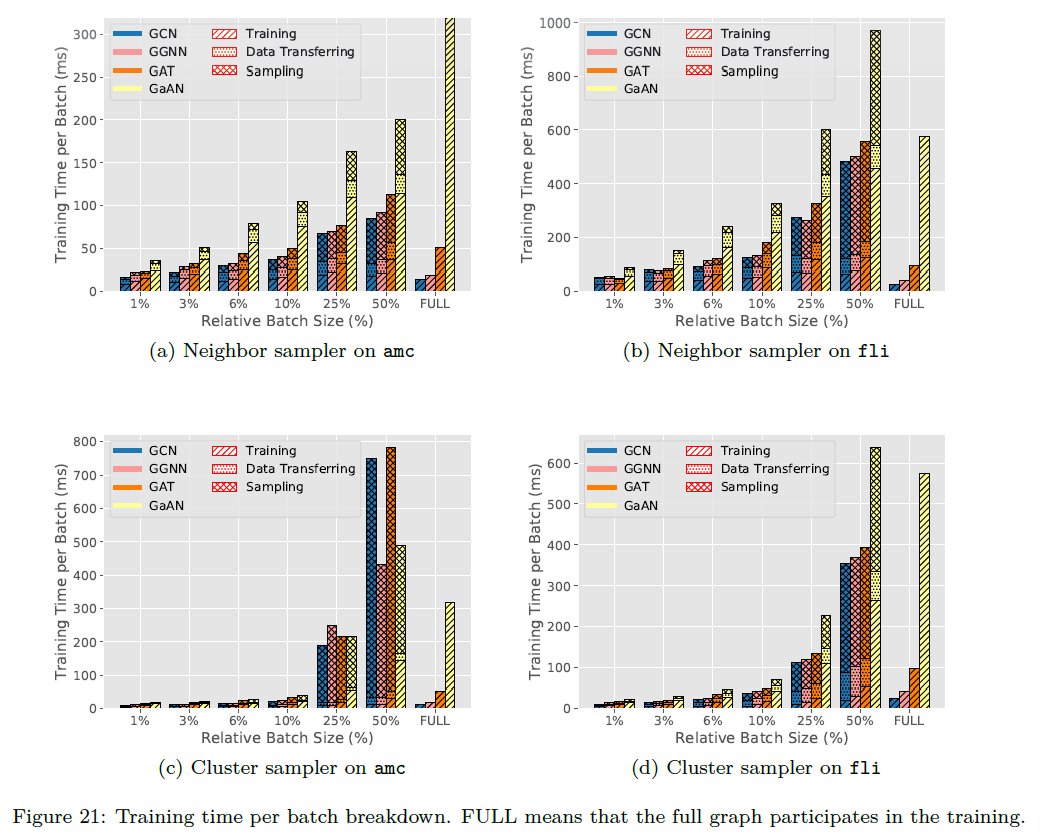


Figure 19 shows how the size of the sampled subgraph changed with the batch size. For the neighbor sampler, the relative batch size is the proportion of the sampled vertices of the last GNN layer in all vertices of the graph. For the cluster sampler, the relative batch size is the proportion of the sampled partitions to 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 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.

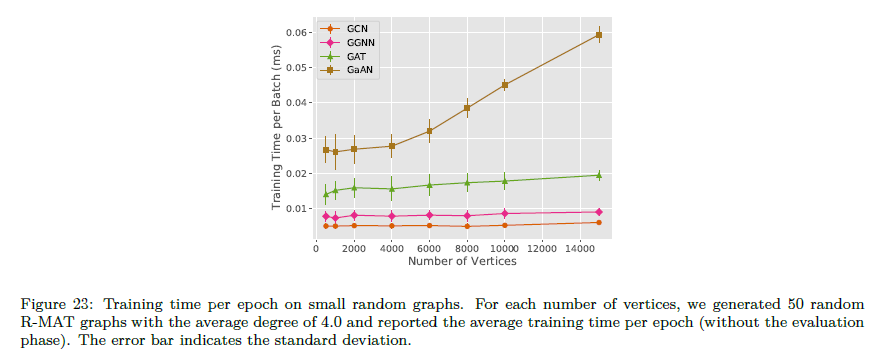


It is worth noting that the average degree of the sampled subgraph was *much lower* than the average degree of the whole graph, especially when the relative batch size is low. Taking the neighbor sampler with the relative batch size of 6% as an example, the average degree of the amp dataset was 31.1, but the average degree of the sampled subgraph was only 5.8. For the cluster sampler, it was even lower to 3.0. Figure 20 compares the degree distribution of the sampled subgraph with the original graph. The slopes of the curves were similar. It indicates that the sampled subgraphs still followed the power-law degree distribution. However, the numbers of vertices in the sampled subgraphs were much smaller than the original graph, significantly lowering the average degrees. According to the experimental results in Section [4.2](#sec:training_time_breakdown), if the average degree becomes lower, the proportion of the training time spent on vertex calculation will become higher, especially for GGNN.

To find out the performance bottleneck with the sampling techniques, we decomposed the training time per batch into three phases: *sampling* on CPU, *transferring* sampled subgraphs from CPU to GPU and *training* with the subgraphs on GPU. 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, even 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 indicate that the implementation of the sampling techniques in PyG is inefficient. When the batch size was slightly big, more than 50% of the time had 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 declined significantly even under big batch sizes. The sampling techniques make training GNNs on big graphs possible for GPU.

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 cases of training GNNs with small sampled subgraphs, we generated random graphs with few vertices and measured the training time per epoch 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 .

### Summary

With small batch size, the sampling techniques could significantly reduce the training time per batch and the peak memory usage during the training. However, small batch sizes could not make full use of the computing power of a GPU. The sampled subgraphs had lower average degrees than the whole graph. However, the current implementation of the sampling techniques was every inefficient under big batch sizes. The time spent on the sampling phase and the data transferring phase could even exceed the training phase.

# Insights

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

1. *The time complexity in Table*[1](#tab:gnn_overview_edge) *and Table*[2](#tab:gnn_overview_vertex) *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 and .
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 both increase *linearly* with the dimensions of the hidden feature vectors. 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 explosive growth in the training time and memory usage.
3. *Performance optimizations should focus on improving the efficiency of the edge calculation.* The edge calculation is the most time-consuming phase in most GNNs.
   * If the complexity of the message function is high, the implementation of is critical to performance. Improving its efficiency can significantly reduce 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 GPU is a potential optimization.
   * If the complexity of is low, the efficiency of the collection step and the aggregation step becomes critical. The existing GNN libraries [11,12,13] already introduce the *fused* operator to improve their efficiency. When the message function is an assignment or a scalar multiplication of the hidden feature vector of the source vertex, the libraries replace the collect, message, and aggregate 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 calculation. 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 with more complex operations like matrix multiplication. A potential optimization is proposing an implementation of the edge calculation that generates the aggregated vectors directly from the input hidden feature 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 limits the data scalability of the GNN training.* The memory expansion ratios of the typical GNNs are very high, making GPU unable to handle big graphs. To train GNNs with big datasets, 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 [32] comes from deep neural network training. It only checkpoints key intermediate results during the forward propagation and recalculates the missing results on demand during the backpropagation. Implementing the checkpoint mechanism in 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 subgraphs sampled with small batch sizes are 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.

# Related Work

## 6.1 Survey of GNNs

[8, 9, 10] survey the existing graph neural network models and classify them from an algorithmic view. They summarize the parallels and differences between the architecture of different GNN models. The typical applications of GNNs are also briefly introduced. Those surveys focus on comparing the existing GNNs theoretically not empirically.

## 6.2 Evaluation of GNNs

Shchur et al. [29] evaluate the accuracy of popular GNN models on the node classification task. Dwived et al. [33] further compare the accuracy of popular GNNs fairly in a controlled environment. Hu et al. [34] propose the open graph benchmark that provides a group of 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 (like PageRank), and MLP-based neural network on GPU. They provide optimization guidelines for the software and the hardware sides. Zhang et al. [17] compare the architectural characteristics of the GNN inference on GPU under the unified SAGA-NN [13] programming model. They find that 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 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 [16, 17] form a complementary study on the efficiency issue of GNNs.

## 6.3 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.

# Conclusion

In this work, we systematically study 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. The training time and the memory usage increase linearly with the hyper-parameters of a GNN.

To find out the performance bottleneck in time, we decompose the training time per epoch on different levels. The training time breakdown analysis indicates that the edge calculation and its related basic operators are the performance bottleneck for most GNNs. Moreover, the intermediate results produced during the edge calculation cause high memory usage, limiting the data scalability. Adopting sampling techniques can reduce training time and memory usage significantly. However, the current implementation brings considerable sampling overheads. The small sampled graphs cannot make full use of the computing power of a GPU either. Our analysis indicates that the edge calculation should be the main target of optimizations. Reducing its memory usage and improving its efficiency can significantly improve the performance of GNN training. Based on the analysis, we propose several potential optimizations for the GNN learning frameworks. We believe that our analysis can help developers to have a better understanding of the characteristics of GNN computation.

# Reference

[1] T. N. Kipf, M. Welling, 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.

URL https://openreview.net/forum?id=SJU4ayYgl

[2] M. Defferrard, X. Bresson, P. Vandergheynst, Convolutional neural networks on graphs with fast localized spectral filtering, in: D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon,

R. Garnett (Eds.), Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain, 2016, pp. 3837–3845.

URL http://papers.nips.cc/paper/6081-convolutional-neural-networks-on-graphs-with-fast-localized

[3] R. Li, S. Wang, F. Zhu, J. Huang, 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, pp. 3546–3553.

URL https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16642

[4] Y. Li, D. Tarlow, M. Brockschmidt, R. S. Zemel, 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.

URL http://arxiv.org/abs/1511.05493

[5] W. L. Hamilton, Z. Ying, J. Leskovec, 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, pp. 1024–1034.

URL http://papers.nips.cc/paper/6703-inductive-representation-learning-on-large-graphs

[6] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, Y. Bengio, Graph attention net- works, in: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings, 2018.

URL https://openreview.net/forum?id=rJXMpikCZ

[7] J. Zhang, X. Shi, J. Xie, H. Ma, I. King, D. Yeung, 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, pp. 339– 349.

URL http://auai.org/uai2018/proceedings/papers/139.pdf

[8] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, M. Sun, Graph neural networks: A review of methods and applications (2018). arXiv:1812.08434.

URL https://arxiv.org/abs/1812.08434

[9] Z. Zhang, P. Cui, W. Zhu, Deep learning on graphs: A survey, CoRR abs/1812.04202. arXiv: 1812.04202.

URL http://arxiv.org/abs/1812.04202

[10] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, P. S. Yu, A comprehensive survey on graph neural networks, IEEE Transactions on Neural Networks and Learning Systems (Early Access) (2020) 1–21. doi:10.1109/TNNLS.2020.2978386.

[11] M. Fey, J. E. Lenssen, Fast graph representation learning with PyTorch Geometric (2019). arXiv: 1903.02428.

URL https://pytorch-geometric.readthedocs.io/

[12] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. J. Smola, Z. Zhang, Deep graph library: Towards efficient and scalable deep learning on graphs (2019). arXiv:1909.01315.

URL https://www.dgl.ai/

[13] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, Y. Dai, 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, pp. 443–458.

URL https://www.usenix.org/conference/atc19/presentation/ma

[14] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, J. Zhou, AliGraph: A comprehensive graph neural network platform, Proc. VLDB Endow. 12 (12) (2019) 2094–2105. doi:10.14778/ 3352063.3352127.

URL http://www.vldb.org/pvldb/vol12/p2094-zhu.pdf

[15] Baidu, Paddle graph learning.

URL https://github.com/PaddlePaddle/PGL

[16] M. Yan, Z. Chen, L. Deng, X. Ye, Z. Zhang, D. Fan, Y. Xie, Characterizing and understanding gcns on gpu, IEEE Computer Architecture Letters 19 (1) (2020) 2225.

[17] Z. Zhang, J. Leng, L. Ma, Y. Miao, C. Li, M. Guo, Architectural implications of graph neural networks, IEEE Computer Architecture Letters 19 (1) (2020) 59–62.

[18]J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, G. E. Dahl, Neural message passing for quantum chemistry, in: D. Precup, Y. W. Teh (Eds.), Proceedings of the 34th International Conference on Machine Learning, Vol. 70 of Proceedings of Machine Learning Research, PMLR, International Convention Centre, Sydney, Australia, 2017, pp. 1263–1272.

URL http://proceedings.mlr.press/v70/gilmer17a.html

[19] D. Duvenaud, D. Maclaurin, J. Aguilera-Iparraguirre, R. Gómez-Bombarelli, T. Hirzel, A. Aspuru-Guzik, R. P. Adams, 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, pp. 2224–2232.

URL http://papers.nips.cc/paper/5954-convolutional-networks-on-graphs-for-learning-molecular-fin

[20] H. Dai, Z. Kozareva, B. Dai, A. J. Smola, L. Song, 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, pp. 1114–1122. URL http://proceedings.mlr.press/v80/dai18a.html

[21] W. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, C. Hsieh, Cluster-GCN: An efficient algorithm for training deep and large graph convolutional networks, CoRR abs/1905.07953. arXiv:1905.07953.

URL http://arxiv.org/abs/1905.07953

[22] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, V. K. Prasanna, 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.

URL https://openreview.net/forum?id=BJe8pkHFwS

[23] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, J. Leskovec, 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, pp. 974–983. doi:10.1145/3219819.3219890.

URL https://doi.org/10.1145/3219819.3219890

[24] J. Chen, T. Ma, C. Xiao, 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.

URL https://openreview.net/forum?id=rytstxWAW

[25] J. Chen, J. Zhu, L. Song, 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, pp. 941–949.

URL http://proceedings.mlr.press/v80/chen18p.html

[26] W. Huang, T. Zhang, Y. Rong, J. Huang, 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, pp. 4563–4572.

URL http://papers.nips.cc/paper/7707-adaptive-sampling-towards-fast-graph-representation-learnin

[27] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, V. K. Prasanna, 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, pp. 462–471. doi:10.1109/IPDPS.2019.00056.

URL https://doi.org/10.1109/IPDPS.2019.00056

[28] Z. Yang, W. W. Cohen, R. Salakhutdinov, Revisiting semi-supervised learning with graph embed- dings, in: M. Balcan, K. Q. Weinberger (Eds.), Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016, Vol. 48 of JMLR Workshop and Conference Proceedings, JMLR.org, 2016, pp. 40–48.

URL http://proceedings.mlr.press/v48/yanga16.html

[29] O. Shchur, M. Mumme, A. Bojchevski, S. Günnemann, Pitfalls of graph neural network evaluation, CoRR abs/1811.05868. arXiv:1811.05868.

URL http://arxiv.org/abs/1811.05868

[30] D. Chakrabarti, Y. Zhan, C. Faloutsos, R-mat: A recursive model for graph mining, in: Proceedings of the 2004 SIAM International Conference on Data Mining, pp. 442–446. doi: 10.1137/1.9781611972740.43.

[31] J. Yang, J. Leskovec, Defining and evaluating network communities based on ground-truth, in:

M. J. Zaki, A. Siebes, J. X. Yu, B. Goethals, G. I. Webb, X. Wu(Eds.), 12th IEEE International Conference on Data Mining, ICDM 2012, Brussels, Belgium, December 10-13, 2012, IEEE Computer Society, 2012, pp. 745–754. doi:10.1109/ICDM.2012.138.

[32] T. Chen, B. Xu, C. Zhang, C. Guestrin, Training deep nets with sublinear memory cost, CoRR abs/1604.06174.

URL http://arxiv.org/abs/1604.06174

[33] V. P. Dwivedi, C. K. Joshi, T. Laurent, Y. Bengio, X. Bresson, Benchmarking graph neural networks, CoRR abs/2003.00982. arXiv:2003.00982.

URL https://arxiv.org/abs/2003.00982

[34] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, J. Leskovec, Open graph benchmark: Datasets for machine learning on graphs, CoRR abs/2005.00687. arXiv:2005.00687. URL https://arxiv.org/abs/2005.00687

1. For the GNN layer 0, graph neurons receive input feature vectors, i.e., [↑](#footnote-ref-0)
2. 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. 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 as abnormal epochs. [↑](#footnote-ref-1)