DGI: An Easy and Efficient Framework for GNN Model Evaluation

Peiqi Yin
AWS Shanghai AI Lab
Shanghai, China
Southern University of Science and Technology
Shenzhen, China
The Chinese University of Hong Kong
Shatin, Hong Kong SAR
pqyin22@cse.cuhk.edu.hk

Xiao Yan*
Southern University of Science and Technology
Shenzhen, China
yanx@sustech.edu.cn

Jinjing Zhou
TensorChord
Shanghai, China
zhoujinjing09@gmail.com

Qiang Fu
George Washington University
Washington, D.C, USA
charlesfoo@gwu.edu

Zhenkun Cai
AWS Shanghai AI Lab
Shanghai, China
zkcai@amazon.com

James Cheng
The Chinese University of Hong Kong
Shatin, Hong Kong SAR
jcheng@cse.cuhk.edu.hk

Bo Tang
Southern University of Science and Technology
Shenzhen, China
tangb3@sustech.edu.cn

Minjie Wang
AWS Shanghai AI Lab
Shanghai, China
minjiw@amazon.com

ABSTRACT

While many systems have been developed to train graph neural networks (GNNs), efficient *model evaluation*, which computes node embedding according to a given model, remains to be addressed. For instance, using the widely adopted *node-wise* approach, model evaluation can account for over 90% of the time in the end-to-end training process due to *neighbor explosion*, which means that a node accesses its multi-hop neighbors. The *layer-wise* approach avoids neighbor explosion by conducting computation layer by layer in GNN models. However, layer-wise model evaluation takes considerable implementation efforts because users need to manually decompose the GNN model into layers, and different implementations are required for GNN models with different structures.

In this paper, we present DGI—a framework for easy and efficient GNN model evaluation, which automatically translates the training code of a GNN model for layer-wise evaluation to minimize user effort. DGI is general for different GNN models and evaluation requests (e.g., computing embedding for all or some of the nodes), and supports out-of-core execution on large graphs that cannot fit in CPU memory. Under the hood, DGI traces the computation graph of GNN model, partitions the computation graph into layers that are suitable for layer-wise evaluation according to tailored rules, and executes each layer efficiently by reordering the computation tasks and managing device memory consumption. Experiment results show that DGI matches hand-written implementations of layer-wise evaluation in efficiency and consistently outperforms node-wise evaluation across different datasets and hardware settings, and the speedup can be over 1,000x.

KDD '23, August 6–10, 2023, Long Beach, CA, USA.

ACM ISBN 979-8-4007-0103-0/23/08…$15.00
https://doi.org/10.1145/3580305.3599805

CCS CONCEPTS

• **Computing methodologies** → **Artificial intelligence**; • **Information systems** → *Computing platforms.*

KEYWORDS

Graph neural networks, System for machine learning

ACM Reference Format:
Peiqi Yin, Xiao Yan*, Jinjing Zhou, Qiang Fu, Zhenkun Cai, James Cheng, Bo Tang, and Minjie Wang. 2023. DGI: An Easy and Efficient Framework for GNN Model Evaluation. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '23), August 6–10, 2023, Long Beach, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3580305.3599805

1 INTRODUCTION

Graph data are ubiquitous in domains such as social networks [37, 47], knowledge representations [18, 19, 34], and bio-informatics [10, 17]. In recent years, graph neural networks (GNNs) have shown outstanding performance for many tasks on graph including node classification [16, 22], clustering [39], and link prediction [29, 60]. A plethora of GNN models with different structures have been proposed, e.g., GCN [27], GAT [43], JKNet [51] and APPNP [28], and developing new modes is still an active research area. These models generally stack L graph aggregation (also called convolution) layers

*Xiao Yan is corresponding author.

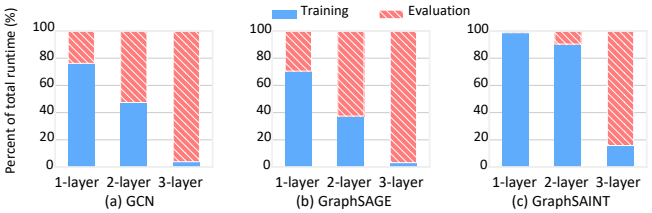


Figure 1: Composition of model training time and node-wise evaluation time in a model training pipeline for the *OGBN-Products* graph on a 32GB-V100 GPU. We conduct evaluation every 10 training epochs following the settings of models in the OGB Leaderboard [22].

to compute an output embedding h_v^L for each node v in a graph $G=(V,E)$, and each layer can be expressed as

$$h_v^l = \text{AGGREGATE}^l(h_u^{l-1},\ \forall u \in \mathcal{N}(v) \cup v;\ w^l), \tag{1}$$

where set $\mathcal{N}(v)$ contains the in-neighbors of node v, h_v^l is the embedding of node v in the l^{th} layer (h_v^0 is the input node embedding given in data), and w^l is the parameter of function AGGREGATE^l.

GNN model evaluation computes output embedding for nodes in the graph with a given model and can take different forms for different use cases. (i) *Full evaluation*, which computes output embedding for all nodes in the graph and is required when the node embedding is used for downstream applications (e.g., recommendation, fraud detection). (ii) *Partial evaluation*, which computes output embedding for a set of nodes (e.g., those with ground-truth labels) and can be used to monitor model training (e.g., for termination or hyper-parameter selection). While GNN training usually samples the neighbors of the nodes for efficiency [7, 8, 11, 20, 57, 64], model evaluation typically uses the full neighborhood for high accuracy. Compare with full-neighbor evaluation, sampling-based evaluation normally degrades accuracy, which is unfavorable for accuracy-critical applications such as recommendation. However, DGI also supports (iii) sampling-based evaluation when the user is willing to trade accuracy for efficiency.

A natural solution to GNN model evaluation is to directly utilize the code for training, which computes output embedding for each target node v individually by first collecting its L-hop neighbors and then applying Eq. (1). We call this approach *node-wise execution* and report the time taken by model training and evaluation for some popular GNNs in Figure 1. The results show that evaluation time increases quickly with the number of layers and can account for up to 94% of the end-to-end time. This is because node-wise execution suffers from *neighbor explosion*: (i) The L-hop neighbors of a target node and the input embeddings of these neighbors are enormous, which makes data preparation expensive. (ii) To fit the memory of GPU, evaluation is conducted in small batch. Target nodes in different batches compute intermediate embeddings (i.e., in layers with $l < L$) for their common neighbors multiple times, resulting in repetitive computation. Neighbor explosion is prominent for model evaluation because it usually uses the full neighborhood as opposed to the neighbor sampling in training [7, 11, 20].

We present an efficient *layer-wise evaluation* approach, which conducts computation layer by layer and handles the tasks of all target nodes in the same layer batch by batch. Layer-wise evaluation is equivalent to node-wise evaluation in that they produce the same results. The advantages of layer-wise evaluation include: (i) it avoids the neighbor explosion problem as a node only accesses its 1-hop neighbors in each layer. The small input set also enables efficient out-of-core execution for large graphs by storing data in external memory and loading only the input set of each batch. (ii) Common computation tasks in a layer from different target nodes are merged to eliminate repetitive computation. However, substantial implementation efforts are required to enjoy the efficiency of layer-wise evaluation. In particular, users need to manually decompose a GNN model into layers, merge the computation tasks in each layer, and manage the intermediate embeddings and node batching. Figures 2(b-c) show that GNN models can go beyond the simple linear structure in Figure 2(a) and new GNN models are becoming even more complex [49], and thus can be difficult to decompose GNN models into layers for evaluation. As such, DGL, a state-of-the-art graph learning library, only provides handwritten layer-wise evaluation codes for several popular GNN models.

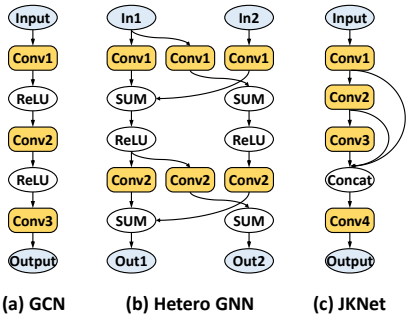


Figure 2: The structure of different GNN models. Conv **is a graph aggregation layer, and we omit the activation functions in (c).**

To mitigate the programming difficulty of layer-wise evaluation, we build DGI—a framework that makes GNN model evaluation easy and efficient. DGI generalizes across different GNN models (e.g., homogeneous and heterogeneous ones) and automatically translates the training code for layer-wise execution, thus removing the burden of writing separate code for model evaluation. In particular, DGI uses a tracer to resolve the computation graph of a GNN model, designs tailored rules to partition the computation graph into layers that yield low evaluation costs, and manages the dependencies among these layers. DGI also dynamically adjusts the batch size based on runtime statistics to avoid OOM and fully utilize device memory, and reorders the graph nodes to form batches for better input sharing in each batch. DGI can handle different model evaluation requests, i.e., full evaluation, partial evaluation and sampling-based evaluation. DGI also supports out-of-core execution for large graphs. Users can enjoy the efficiency of layer-wise evaluation and all these functionalities via simple configurations.

We evaluate DGI on 6 real graphs, 5 GNN models with different structures, and various hardware configurations. The results show that DGI matches or even outperforms handwritten layer-wise evaluation codes. Compared with node-wise evaluation, DGI runs faster across all settings, and the speedup can be up to 1,000x and is usually 10x-100x. Experiments also show that DGI handles different kinds of model evaluation requests efficiently and its running time scales linearly with the number of layers. Micro experiments show that dynamic batch size control and node reordering optimizations in DGI are effective in reducing model evaluation time.

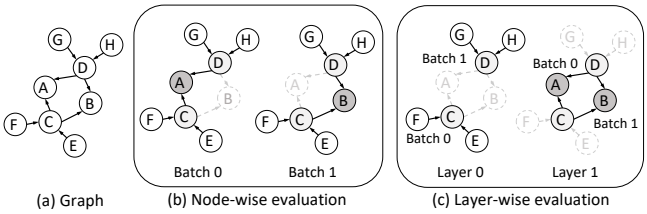


Figure 3: Node-wise and layer-wise evaluation for a 2-layer GNN model on a toy graph. Both examples illustrate the process to compute output embeddings for node A and B, in two batches with a batch size of 1.

2 LAYER-WISE EVALUATION APPROACH

Given a graph $G = (V, E)$ and the input (i.e., layer-0) embedding of all nodes $\mathcal{H}^0$, GNN evaluation computes output embedding using an L-layer GNN model. Algorithm 1 is the pseudo-code of node-wise evaluation, and Figure 3(b) provides a running example. In Algorithm 1, set $\mathcal{N}_v^L$ contains all L-hop neighbors of node v, and $\mathcal{H}^0(\mathcal{N}_v^L)$ indicates the input embedding of these neighbors. We assume that input data is stored on an external device (e.g., CPU) and fetched to the computing device (e.g., GPU) for execution in batches. In a batch, node-wise evaluation takes three steps, i.e., data preparation (Lines 4-6 in Algorithm 1), data transfer (Lines 7 and 9), and computation (Line 8).

For data preparation, a target node v accesses all its L-hop neighbors, which results in a large input set (i.e., neighbor explosion) as the number of neighbors increases quickly with hops in graph. Regarding computation, each batch of target nodes is handled independently, and thus different batches of target nodes conduct repetitive computation for their common neighbors. For the example in Figure 3(b), computing the layer-2 embeddings of both nodes A and B requires the layer-1 embeddings of nodes C and D, and thus the layer-1 embeddings of nodes C and D are computed twice. Empirically, the running time of node-wise evaluation increases exponentially with the number of layers and constitutes a major part of the training pipeline for multi-layer models.

Algorithm 2 is the pseudo-code for layer-wise evaluation, and Figure 3(c) shows a running example. In Algorithm 2, $\mathcal{H}^l$ contains the layer-l embedding for all nodes, and set $\mathcal{N}_v$ is the 1-hop neighbors of node v. Algorithm 2 shows that layer-wise evaluation conducts computation layer by layer, which differs from the node-by-node scheme of node-wise evaluation. One advantage of layer-wise evaluation is the small input set as each node only requires its 1-hop neighbors in data preparation (Line 6 of Algorithm 2), which makes layer-wise evaluation less likely to run OOM. Also, the small input set allows a large batch size in the inner loop (Line 3 of Algorithm 2) and thus enables good input sharing. For the example in Figure 3(c), if the layer-2 embedding of nodes A and B are computed in a batch, then the layer-1 embedding of nodes C and D are loaded to GPU only once and shared by nodes A and B. Another advantage of layer-wise evaluation is that it completely eliminates repetitive computation. For the example in Figure 3(c), both nodes A and B require the layer-1 embedding of nodes C and D, which are computed only once in layer-1.

Apart from full evaluation we present in Algorithm 2, Layer-wise evaluation can also handle other kinds of evaluation requests. In particular, partial evaluation computes output embedding for only some of the graph nodes and can be used to monitor model training (where usually only some nodes have labels). We can conduct BFS of depth L from the target nodes to mark a node set V^l, which contains the nodes for which layer-l embeddings need to be computed. Algorithm 2 only needs to change the node set from V to V^l in Line 3. Moreover, Sampling-based evaluation selects some neighbors for aggregation, which is similar to training and can be used to trade accuracy for efficiency. In this case, we mark the node set V^l according to the sampled l-hop neighbors.

A crucial drawback of layer-wise evaluation is its programming difficulty. First, node-wise evaluation can directly utilize the training code, while layer-wise evaluation requires writing separate code for evaluation. Second, it can be difficult to decompose a GNN model into layers for evaluation when it goes beyond a linear stack of layers. For the examples in Figures 2(b) and (c), heterogeneous GNN models have complex connections among the normal operators and graph convolutions, while JKNet has jumping links that connect multiple graph convolution layers. In these cases, a programmer needs to determine a layer partition that is efficient for evaluation and manages the dependencies of the intermediate embeddings. We show an example in the Appendix for JKNet, whose forward code has only 8 LOCs, but layer-wise evaluation code needs more than 30 LOCs even after removing some API calls. Additional programming difficulties include: (i) collecting the evaluation tasks in each layer for partial and sampling evaluation, (ii) managing the batch size to fit the memory of the computing device, and (iii) handling out-of-core execution for large graphs stored on disk.

Discussion. The layer-by-layer execution pattern of layer-wise evaluation is also used by some early systems for GNN training, e.g., NeuGraph [36] and ROC [36]. These systems handle simple

Algorithm 1 Node-wise Evaluation for an L-layer GNN

1: $\mathcal{H}^L \leftarrow$ empty tensor for holding output embedding
2: **for** each batch V_B of nodes in V **do**
3: $\quad\mathcal{N} \leftarrow$ null neighbor set, $\mathcal{H} \leftarrow$ null input embedding set
4: $\quad$**for** each node v in V_B **do**
5: $\quad\quad\mathcal{N} = \mathcal{N} \cup \mathcal{N}_v^L$, $\mathcal{H} = \mathcal{H} \cup \mathcal{H}^0(\mathcal{N}_v^L)$
6: $\quad$**end for**
7: $\quad$Copy $\mathcal{N}$ and $\mathcal{H}$ to the computing device
8: $\quad$Run Eq. (1) for each node v in V_B to obtain $\mathcal{H}^L(V_B)$
9: $\quad$Store $\mathcal{H}^L(V_B)$ to back to $\mathcal{H}^L$
10: **end for**

Algorithm 2 Layer-wise Evaluation for an L-layer GNN

1: **for** layer $l \in \{1, 2, \cdots, L\}$ **do**
2: $\quad\mathcal{H}^l \leftarrow$ empty tensor for layer-l embedding
3: $\quad$**for** each batch V_B of nodes in V **do**
4: $\quad\quad\mathcal{N} \leftarrow$ null neighbor set, $\mathcal{H} \leftarrow$ null input embedding set
5: $\quad\quad$**for** each node $v \in V_B$ **do**
6: $\quad\quad\quad\mathcal{N} = \mathcal{N} \cup \mathcal{N}_v$, $\mathcal{H} = \mathcal{H} \cup \mathcal{H}^{l-1}(\mathcal{N}_v)$
7: $\quad\quad$**end for**
8: $\quad\quad$Copy $\mathcal{N}$ and $\mathcal{H}$ to the computing device
9: $\quad\quad$Run layer-l of Eq. (1) and get the output $\mathcal{H}^l(V_B)$
10: $\quad\quad$Store $\mathcal{H}^l(V_B)$ (layer-l embedding of batch V_B) into $\mathcal{H}^l$
11: $\quad$**end for**
12: $\quad$Release storage for $\mathcal{H}^{l-1}$ if it is no longer needed
13: **end for**

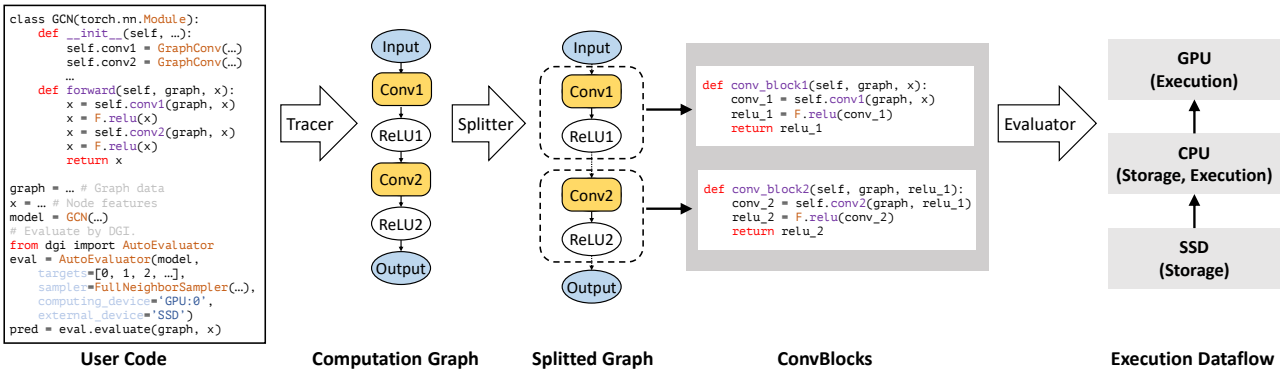


Figure 4: The workflow of DGI.

GNNs with a linear structure and conduct full-batch training (uses all graph nodes in each iteration). As the GNN models become more complex, most GNN training systems use the node-wise approach, which does not require conducting layer decomposition and works naturally with min-batch training (uses a batch of nodes in each iteration). Thus, there is usually a gap between the training code and layer-wise evaluation.

There do exist manual implementations that utilize the idea of layer-wise for GNN models with a simple linear structure (e.g., GCN and GraphSAGE). Compared with them, this section makes 3 contributions: (1) we formally describe the general procedure of layer-wise evaluation; (2) we extend layer-wise evaluation to GNN models that go beyond a linear structure; (3) we show that layer-wise evaluation can handle different model evaluation requests.

3 THE DGI DESIGN

API and Overview. DGI is developed to make the use of layer-wise evaluation easy by automatically translating the training code for execution. We show an example of using DGI in the left part of Figure 4. Programmers first import the *AutoEvaluator* class from the *DGI* library, and then use the (training code of the) GNN model to initialize an *AutoEvaluator* object. Parameters to specify include the target nodes that need to compute output embedding (for partial evaluation, set as all nodes in the graph by default), the neighbor sampling strategy (for sampling evaluation, use all neighbors by default), the computing device (e.g., a GPU), and where to store the output tensors (e.g., CPU memory and SSD). Then the path of the graph data and input node embedding (i.e., graph and x) are fed to the *AutoEvaluator* to conduct model evaluation. The output embeddings are stored in *pred*. By default, DGI uses a single GPU for evaluation, and stores data in CPU memory.

DGI consists of three key modules, i.e., *tracer*, *splitter*, and *evaluator*, as shown in Figure 4. The tracer traces the Python code of the GNN model and converts it into a computation graph, which consists of operators such as graph aggregation (denoted as Conv) and activation function. The splitter uses customized rules to partition the computation graph into *ConvBlocks*, with each ConvBlock corresponding to one layer of the GNN model. The tracer generates codes for executing each of those ConvBlocks. The splitter also generates a schema describing the dependencies of the ConvBlocks (i.e., a ConvBlock takes input from which ConvBlocks). The evaluator executes the ConvBlocks sequentially (i.e., layer by layer) and runs each ConvBlock in batches. For each batch, the target nodes and their 1-hop neighbors are collected for data preparation, then the batch is transferred to the GPU for computation, and finally the output embeddings of the target nodes are dumped to the CPU. DGI can be configured to fetch data from and store output embedding in SSD when CPU memory is not sufficient.

3.1 Tracer

The tracer converts the training code of a GNN model into a computation graph. We implement DGI tracer using Torch.fx as it generates computation graph in Python format. In particular, Torch.fx uses placeholders called *proxies* to replace the actual tensors and records the operators encountered by the proxies in the forward pass of a model. The resultant computation graph is a DAG, where nodes are operators and edges are the dependencies between operators. We classify the operators into two categories, i.e., *normal operator* and *graph convolution operator* (Conv for short). Normal operators work on the tensor of each individual graph node/edge, e.g., Add, ReLU, and MatMul. A Conv aggregates the 1-hop neighbors of a graph node to update its embedding. During tracing, DGI tracer recognizes the Conv operators by matching computation patterns. To allow the splitter to determine the execution order of the Conv operators, we associate an integer l to each Conv operator to indicate the layer it belongs to. In particular, we traverse the computation graph from the input, and mark the Conv operators that have no preceding Conv operators as layer 1. For a Conv operator with preceding Conv operators (called precursors), its layer is marked as one plus the maximum layer of its precursors.

3.2 Splitter

The DGI splitter partitions the computation graph of a GNN model into sub-graphs, and each sub-graph is treated as a ConvBlock (i.e., layer). We adopt the following three rules for computation graph partitioning.

Layer indexing. It means that Conv operators are organized into the same ConvBlock if and only if they have the same layer counter l. For example in Figure 5(a), Conv2 and Conv3 are in the same ConvBlock because their layer counters are both 2. As the Conv

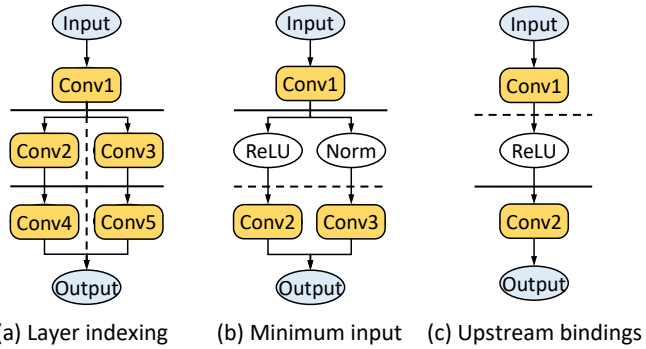


Figure 5: Illustrations of the rules for computation graph partitioning. The solid line denotes the partitioning adopted by DGI while the dotted line is an inferior alternative.

operators in a ConvBlock have the same layer counter, the layer indexing rule ensures that a target node only needs its 1-hop neighbor nodes to execute each ConvBlock. An alternative is to treat each Conv operator as a ConvBlock (which also satisfies layer-wise evaluation), for example, Conv2 and Conv3 could be two ConvBlocks. However, this partitioning misses possible opportunities for *input sharing* among ConvBlocks. For the example in Figure 5(a), both Conv2 and Conv3 take the output embedding of Conv1 as input, and thus can share input embeddings when they are loaded to the GPU. When treated as two separate ConvBlocks, Conv2 and Conv3 need to load the embedding generated by Conv1 individually.

Minimum input. It means that when there are multiple ways to partition two consecutive ConvBlocks, choose the way that yields the minimum number of input tensors. Note that when an edge in the computation graph is selected as a cut, the output node embedding of the upstream ConvBlock is saved on the CPU and loaded to the GPU for the downstream ConvBlock. For the example in Figure 5(b), both the dotted line and the solid line satisfy the layer indexing rule. We choose the solid line as it only requires each neighbor node to transfer one embedding vector to the GPU for the second ConvBlock (by applying the ReLU and Norm operators to a node embedding after it is loaded to the GPU). In contrast, if the dotted line is used, each neighbor node needs to transfer two embedding vectors to the GPU (as ReLU and Norm are applied to the output of the upstream ConvBlock). Thus, the minimum input rule minimizes the neighbor node data required by each ConvBlock for small data transfer cost.

Upstream binding. It means that a normal operator should be grouped with the upstream ConvBlock when there are multiple partitioning schemes to minimize data transfer. For example, in Figure 5(c), we choose the solid line as the cut as it puts the ReLU operator in the first ConvBlock. If we group normal operators with the downstream ConvBlock (i.e., using the dotted line), there will be repetitive computation. The reason is that a node usually serves as a neighbor node for multiple target nodes (e.g., node C is used by nodes A and B in Figure 3), and if the dotted line is used, each target node needs to apply the normal operator on its neighbor nodes individually. In contrast, the solid line applies the normal operator only once for each neighbor node when generating output for the upstream ConvBlock. Thus, the upstream binding rule avoids repetitive computation.

Based on these rules, we adopt a 3-step procedure to partition the computation graph into ConvBlocks. First, we determine the ConvBlock of each Conv operator according to the layer counters of the Conv operators (i.e., layer indexing). Second, starting from the Conv operators of the same ConvBlock, we traverse their preceding operators in the computation graph to find the partition points with the minimum number of inputs (i.e., minimum input). If there are multiple minimum input points, we choose the one that groups the largest number of normal operators into the upstream ConvBlocks (i.e., upstream binding). Note that minimum input and upstream binding conflict sometimes (e.g., Figure 5(b)). We prioritize minimum input as repetitive data loading is more expensive than the repetitive execution of normal operators.

Given the partitioning results of the splitter, the tracer registers each ConvBlock to a Python function for execution. As a ConvBlock may take input from multiple ConvBlocks (e.g., JKNet), we use a schema to record the dependencies among the ConvBlocks, and each ConvBlock fetches input embedding according to the schema. We store the output embedding of each ConvBlock in a separate map, and the output of a ConvBlock is deleted when there are no ConvBlocks depending on it.

3.3 Evaluator

The evaluator executes the ConvBlocks sequentially and runs each ConvBlock batch by batch. To execute a ConvBlock, the evaluator first allocates memory for output node embeddings. For each batch, execution takes four steps: (i) *data preparation*, a set of target nodes V_B are selected for the batch, and for each node $v \in V_B$, collect its 1-hop in-neighbors into the input set along with their embeddings (we modify DGL dataloader for this purpose); (ii) *data transfer*, input set of batch V_B is transferred to the GPU; (iii) *computation*, the output embeddings for nodes in V_B are computed on the GPU; and (iv) *embedding dumping*, the output embeddings are dumped to CPU. To achieve a short evaluation time, the evaluator should execute each ConvBlock in a small batch size while avoiding OOM, and we discuss node batching in Section 4.

For full evaluation, the target nodes of all ConvBlocks are the complete graph node set. For partial evaluation, the evaluator annotates the target nodes for each ConvBlock before execution. For an L-layer GNN model, the L^{th} layer target node set V^L is specified by the user. The $(L-1)^{\text{th}}$ layer target node set V^{L-1} contains the 1-hop in-neighbors of nodes in V^L. The annotation process goes on recursively until obtaining V^1, which is the target node set for layer-1. For a ConvBlock on layer-l ($l \in [1, L]$), the evaluator only computes output embedding for nodes in V^l. For sampling evaluation, we annotate the sampled neighbors for nodes in V^{l+1} as V^l. We observe that the annotation process for V^l is expensive and the result target node sets contain most of the graph nodes if V^{l+1} ($l < L$) is already very large. As an optimization, we skip the annotation and directly assign all nodes to V^l when $|V^{l+1}| \times d_{avg} \geq n$, where d_{avg} is the average degree and n is the number of nodes.

Table 1: Statistics of the graphs used in the experiments.

Name	Nodes	Edges	Dim.	Avg Deg.	Tot. Size
Livejournal1	4.84 M	86.2 M	100	17.8	2.45 G
OGBN-Products	2.45 M	126 M	100	51.5	2.11 G
OGBN-Papers100M	111 M	1.72 B	100	15.5	64.9 G
Friendster	65.6 M	3.61 B	128	55.0	58.2 G
OGBN-MAG	1.93 M	21.1 M	128	10.9	0.94 G
OGBN-MAG240M	244 M	3.46 B	768	14.2	201 G

4 HANDLE LARGE GRAPHS

Real-world graphs can be large and cause two problems for evaluation. First, target nodes and input embedding may not fit in GPU memory, which requires organizing the target nodes into batches for computation and controlling the batch size. Second, graph data and intermediate embedding may not fit in CPU memory, which requires disk storage. We discuss related optimizations in this part.

4.1 Dynamic Batch Size Control

A properly chosen batch size should avoid exceeding GPU memory and be as large as possible. This is because a large batch size allows better input sharing (i.e., multiple target nodes share the same input node embedding) and thus reduces data transfer. However, determining a good batch size is challenging. First, it is difficult to estimate the memory consumption of GNN models due to reasons such as temporary tensors [15], and thus even experienced programmers can only choose a safe but small batch size before execution. Second, a static batch size may not be suitable for all batches. For example, a small batch size is needed if a batch mainly contains nodes with large in-degrees, while a large batch size should be used if a batch mainly contains nodes with small in-degrees.

Considering the challenges, we use a dynamic strategy to configure the batch size according to run-time statistics. In particular, we observe that the number of target nodes (i.e., batch size, denoted as n_t) and the total in-degrees of the target nodes (denoted as n_i) are closely related to the GPU memory consumption of a batch, and thus keep adding target nodes to a batch until either the node count threshold or edge count threshold is exceeded. The two thresholds are initialized as safe values to avoid OOM and adjusted in the evaluation process. Specifically, for the k^{th} batch, we measure its peak memory consumption as M_k, and adjust the thresholds for the next batch as

$$r = M/M_k, \quad n_t^{k+1} = r \times n_t^k, \quad n_i^{k+1} = r \times n_i^k, \tag{2}$$

where n_t^k (resp. n_i^k) is the node count (resp. edge count) threshold for the k^{th} batch. M is the expected memory consumption and set as 90% of the available GPU memory. The rationale is to increase batch size if memory is underutilized. If a batch runs OOM, we reduce the thresholds to half of their previous values, and DGI re-conducts the batch. For quick node batching, we precompute prefix sum for the in-degrees of the nodes such that binary search can be used to find a batch of target nodes meeting the thresholds.

4.2 Node Reorder

A naive solution to node batching is to randomly group the target nodes. This is sub-optimal as some target nodes can share input embedding to reduce data transfer. Take the graph in Figure 3 for example and consider a batch size of 2. If nodes A and C form a batch, 4 neighbor nodes are required while only 2 neighbor nodes are required if nodes A and B form a batch. Therefore, we should organize target nodes sharing common neighbors into the same batch. For this purpose, natural solutions partition the graph into strongly connected components and treat nodes in each component as a batch. However, graph partitioning tools such as METIS [26] and KaFFPaE [40] have long running time. Node reordering algorithms like Rabbit [3] and Gorder [48] run fast but are designed for improving cache hit and perform poorly for our purpose (see Section 5.3). Thus, we use a lightweight strategy to renumber the nodes and take nodes with consecutive IDs as a batch. In particular, RCMK [6], a BFS-based algorithm, is used to enumerate the nodes, and IDs are assigned to the nodes in their enumeration order. The rationale is that nodes adjacent in the BFS order are likely to share common neighbors.

4.3 SSD Support

When the graph data is large or a cheap machine is utilized, the graph data or output embedding may not fit in CPU memory. DGI allows to store graph data and embeddings in SSD and automatically loads them to conduct model evaluation. In particular, we use numpy [21] *memmap* to map SSD as part of CPU memory. A new graph type called *SSDGraph* is utilized to load sub-graphs to CPU from the graph data on SSD. *SSDNeighborSampler* is developed to extract the adjacency lists of the target nodes from *SSDGraph* to CPU memory for each batch. We store the graph data on SSD in CSC format (indptr, indices), where the in-neighbors of node i are kept continuously in $indices[indptr[i] : indptr[i+1]]$. For model evaluation, node batches do not need to be shuffled as in training, and thus we use continuous dst nodes in CSC indptr (e.g., node i to node j) for each batch and extract the src nodes as $indices[indptr[i] : indptr[j]]$. This read pattern avoids the slow random access of SSD by reading data in blocks. The required input embeddings are also loaded to CPU memory using *memmap*. After the GPU finishes a batch, DGI writes the output embeddings back to SSD. DGI's SSD support totally removes the CPU memory bottleneck in model evaluation and makes it possible to evaluate models on large graphs by very cheap machines.

5 EXPERIMENTAL EVALUATION

We introduce the experiment settings in Section 5.1, compare DGI, manually written layer-wise evaluation, and node-wise evaluation in Section 5.2, and evaluate the optimizations of DGI in Section 5.3.

5.1 Experiment Settings

We use 3 popular homogeneous GNN models, i.e., GCN [27], GAT [43], and JKNet [51], and 2 heterogeneous GNN models, i.e., RGCN [41] and HGT [23]. GCN uses mean pooling to aggregate neighbor embeddings while GAT adopts multi-head attention (we use 2 attention heads). The final layer of JKNet aggregates neighbor embeddings from all preceding layers. Both RGCN and HGT work on heterogeneous graphs and aggregate neighbor embeddings based on certain *src-edge-dst* type triplets. For all models, we mainly use 2 or 3 layers following the most widely used configurations [55]. The dimension of intermediate embedding is set as 128. We conduct experiments on 6 popular and public graphs, i.e., *OGBN-Products*, *OGBN-MAG*,

Table 2: Execution time of node-wise evaluation, manual-written layer-wise evaluation and DGI (in seconds). For small datasets (*OGBN-Products*, *Livejournal1* and *OGBN-MAG*), we use 3-layers GNN models. For large datasets (*OGBN-Papers100M*, *Friendster* and *OGBN-MAG240M*), node-wise evaluation may encounter a super large input node set for 3-hop neighbors, which results in OOM. So here we use 2-layer GNN models for those large datasets. When node-wise evaluation runs for more than 10,000s, we estimate its execution time by $t \times \frac{1}{\alpha}$, where t is the elapsed execution time and α is the percentage of processed target nodes.

	Products			Livejournal1			MAG		Papers100M			Friendster			MAG240M	
GNN Models	GCN	GAT	JKNet	GCN	GAT	JKNet	RGCN	HGT	GCN	GAT	JKNet	GCN	GAT	JKNet	RGCN	HGT
Node-wise (16GB)	7110	9740	7150	4330	5220	4890	208	263	3620	3980	3470	57300	81900	63700	29400	143000
DGL-manual (16GB)	9.08	13.8	11.6	15.2	27.2	19.9	24.6	29	343	631	378	743	926	750	2166	2915
DGI (16GB)	6.08	8.96	6.70	11.0	16.9	13.5	19.7	22.0	267	476	272	425	741	435	999	1580
Node-wise (32GB)	1690	1970	1760	1630	2110	1990	143	204	1950	3070	1940	48700	64900	33600	18200	70200
DGL-manual (32GB)	7.12	11.7	8.48	13.1	21.9	15.9	17.8	21.7	310	530	297	470	711	471	2019	2789
DGI (32GB)	4.51	6.09	4.20	6.32	10.5	7.10	12.8	14.9	177	397	175	238	465	231	643	1010

OGBN-Papers100M, *OGBN-MAG240M* from the Open Graph Benchmark [22], and *Friendster*, *Livejournal1* from the Stanford Network Analysis Platform [53]. Table 1 reports the statistics of the graphs. Among them, *OGBN-MAG* and *OGBN-MAG240M* are heterogeneous graphs used to experiment heterogeneous GNNs.

We mainly compare our DGI with node-wise evaluation, and both of them are implemented on top of DGL. We also include the manually written layer-wise evaluation implementations in DGL. As DGL only provides code for GCN, GAT and RGCN, we write layer-wise evaluation code for JKNET and HGT by ourselves. We denote these manually written implementations as *DGL-manual*.

We use two different machines for the experiments. One machine has an Intel(R) Xeon(R) E5-2686 CPU with 96 cores, 488 GB main memory, and one NVIDIA V100 GPU with 16GB HBM (denoted as V100-16GB). The other machine has an Intel(R) Xeon(R) Gold 6126 CPU with 24 cores, 1.5TB main memory, and one NVIDIA V100 with 32GB GPU HBM (denoted as V100-32GB). The two machines have sufficient main memory to store data and intermediate embedding for all graphs, and we use them to explore the influence of GPU memory. We use the execution time of evaluation as the main performance metric and keep 3 effective numbers in the results. For partial evaluation, we randomly sample some nodes from the graph as the target nodes. For sampling evaluation, we use a fan-out of 10 for all layers following the GraphSage [20] implementation in DGL [44], which means that each target node samples 10 neighbors. To run node-wise and DGL-manual evaluation, we use a grid search to find the batch size that minimizes evaluation time.

5.2 Main Results

Full evaluation. We report the execution time of node-wise evaluation, DGL-manual, and DGI for full evaluation in Table 2, which lead to several observations. First, DGI runs faster than DGL-manual in all cases and the speedup can be up to 2.76x. This is because DGL-manual does not have the node reorder and dynamic batch size control optimizations in DGI. Second, DGI consistently outperforms node-wise evaluation, and the speedup ranges from 10.6x to over 1000x. Considering the homogeneous GNNs (i.e., GCN, GAT and JKNet), DGI yields larger speedup for 3-layer models and dense graphs (i.e., having large average degrees). In particular, 3-layer models all observe over 2 orders of magnitude speedup while 2-layer models may have 10x speedups. For 2-layer models, the speedup of DGI is 10x for *OGBN-Papers100M* (with an average degree of 15.5) and 100x for *Friendster* (with an average degree of 55). This is because more layers and denser graphs make node-wise evaluation access more neighbor nodes, which aggravates the neighbor explosion problem. Third, all solutions run faster on V100-32GB than V100-16GB because larger GPU memory allows larger batch size, resulting in more chances for input sharing. Finally, DGI generally observe smaller speedups over node-wise for the heterogeneous GNNs (i.e., RGCN and HGT) than the homogeneous GNNs. This is because each Conv only aggregates neighbors of a specific type in heterogeneous GNNs, and thus the neighbor explosion problem of node-wise evaluation is alleviated.

In Table 3, we compare node-wise evaluation, DGL-manual and DGI by changing the number of layers in the GNN models. The results show that the running time of node-wise evaluation increases quickly with the number of layers. This is because node-wise evaluation accesses L-hop neighbors for L-layer models, and the number of neighbors increases quickly with L. In contrast, the execution time of DGL-manual and DGI increases almost linearly with the number of layers as layer-wise evaluation only accesses one-hop neighbors and is free from the neighbor explosion problem. For 1-layer models, node-wise evaluation and the layer-wise evaluation of DGI are equivalent, and thus they yield comparable running time. For 4-layer models, DGI usually outperforms node-wise evaluation by more than 1000x, and node-wise evaluation easily goes OOM.

To understand the speedup of DGI over node-wise evaluation, we decompose their running time in Figure 7. The first 4 time consumptions are introduced in Section 3.3, and *initialization* means initializing dataloader and creating empty tensors for output embeddings. Notice that the y-axis uses different scales. The results show that node-wise evaluation uses significantly longer time than DGI for data preparation, data transfer and GNN forward, which makes the time for embedding dumping and initialization negligible. In particular, each of those three parts takes thousands of seconds for node-wise evaluation but only several seconds for DGI. This is because node-wise evaluation accesses L-hops neighbors for data preparation, and the large input set results in a small batch size, which makes input sharing difficult. In contrast, data preparation is quick for DGI as it only accesses 1-hop neighbors, and small input set enables large batch size for good input sharing.

Partial evaluation. We compare node-wise evaluation and DGI for partial evaluation in Figure 6. We omit DGL-manual in this experiment as it does not support partial evaluation. The results show

Table 3: Execution time of node-wise evaluation, manual-written layer-wise evaluation and DGI (in seconds) for full evaluation the changing the number of layers. The graph is *OGBN-Products*, "⋆" means that JKNet requires at least 2 layers, and "OOM" indicates that an out-of-memory exception was raised.

	GCN				GAT				JKNet			
# of Layers	1	2	3	4	1	2	3	4	1	2	3	4
Node-wise (V100-16GB)	3.16	153	7110	OOM	3.83	189	8740	OOM	⋆	152	7150	OOM
DGL-manual (V100-16GB)	3.24	6.46	9.08	11.5	3.81	9.03	13.8	20.3	⋆	6.68	11.6	14.8
DGI (V100-16GB)	2.60	4.37	6.10	9.79	2.62	5.68	8.96	14.5	⋆	4.31	6.70	11.9
Node-wise (V100-32GB)	2.56	26.8	1690	69400	2.83	29.4	1970	122000	⋆	70.2	1760	90900
DGL-manual (V100-32GB)	2.61	4.87	7.12	9.54	2.96	7.25	11.7	16.2	⋆	4.86	8.48	12.2
DGI (V100-32GB)	1.75	3.32	4.51	5.40	1.95	4.76	6.09	8.10	⋆	3.29	4.20	5.71

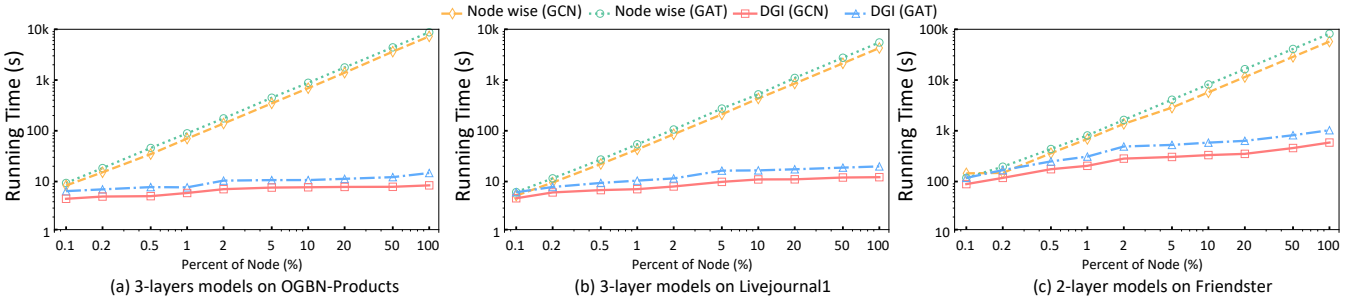


Figure 6: Running time of DGI and node-wise for partial evaluation (on V100-16GB GPU). Note that both axes use log scale.

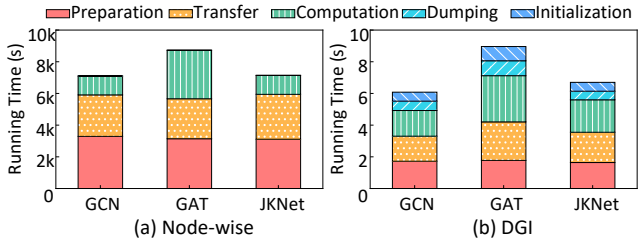


Figure 7: Decomposing the running time of node-wise evaluation and DGI for full evaluation. The results are obtained for the *OGBN-Products* graph on V100-16GB with 3-layer models.

that when increasing the number of target nodes, the evaluation time increases linearly for node-wise evaluation but sub-linearly for DGI. This is because DGI effectively shares common computation and input nodes for different target nodes while sharing is difficult for node-wise evaluation due to the small batch size caused by neighbor explosion. While DGI and node-wise evaluation perform comparably when the target node set is small (e.g., 0.1% of all graph nodes), the speedup of DGI over node-wise evaluation increases with the number of target nodes. This is because more target nodes enable DGI to better share input and computation.

Comparison with GNNAutoScale. GNNAutoScale [14] uses historical embedding to reduce computation in both training and evaluation. For model evaluation, it directly uses the historical embeddings of the last layer, which are computed and saved during training, to compute the prediction for the target nodes. While this strategy is efficient in theory, GNNAutoScale has two drawbacks. First, it only supports GNNs with a linear structure (e.g., GCN, GAT). Second, as pointed out by ReFresh [24], it suffers from a huge accuracy drop either when the graph is large or the model is complex. Table 4 compares the test accuracy and evaluation time of DGI and GNNAutoScale. Although GNNAutoScale runs faster than DGI because it only computes the last layer, it comes with a significant accuracy drop, especially for GAT. This may be unacceptable for critical applications such as recommendation and fraud detection because accuracy directly affects revenue and security.

Table 4: Compares the test accuracy and evaluation time of DGI and GNNAutoScale on the OGBN-Products graph.

Test ACC / Time	GCN-2 layers	GCN-3 layers	GAT-2 layers	GAT-3 layers
DGI	0.760 / 4.37 s	0.765 / 6.10 s	0.787 / 5.68 s	0.813 / 8.96 s
GNNAutoScale	0.746 / 1.90 s	0.751 / 1.90 s	0.605 / 3.23 s	0.267 / 3.22 s

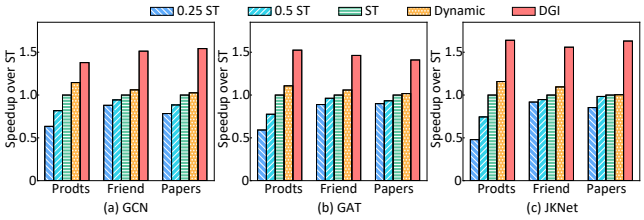
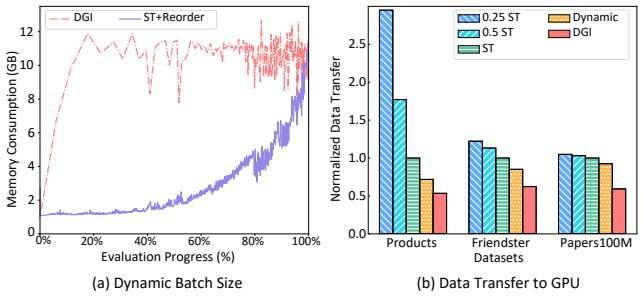


Figure 8: Effect of dynamic batch size and node ordering for 3-layer models on V100-16GB GPU. evaluation time is reported as speedup over *ST*.

5.3 Micro Experiments

DGI comes with two key optimizations, i.e., node reorder and dynamic batch size control. To evaluate their effects, we design 4 baselines: *0.25ST*, *0.5ST*, *ST*, and *Dynamic*, which all adopt the layer-wise evaluation scheme. *ST* uses the best static batch size (found by grid search) that minimizes evaluation time. We also include *0.5ST* and *0.25ST*, which use 0.5 and 0.25 of the *ST* batch size, respectively, and reflect practical scenarios where users choose a safe batch size. *Dynamic* uses dynamic batch size control but disables node reorder.



(a) Dynamic Batch Size (b) Data Transfer to GPU

Figure 9: The effects of node reordering and dynamic batch size control. We use 3-layer GCN and the *Friendster* graph.

Figure 8 reports the evaluation time of DGI and the baselines. The results show that *Dynamic* consistently outperforms *ST*, which indicates that dynamic batch size control is effective. To understand its benefit, we plot the GPU memory consumption for DGI and DGI without dynamic batch size control (denoted as *ST+Reorder*) in Figure 9(a). The results show that DGI fully utilizes GPU memory except for the initial bootstrap period. In contrast, static batch size (i.e., *ST+Reorder*) is limited by the memory consumption of the high-degree nodes (in the last few batches) and thus has severe memory under-utilization. Figure 8 also shows that node reordering gives another performance boost to DGI (by comparing *Dynamic* with DGI). This is because node reordering organizes nodes with similar neighbors into the same batch to reduce the number of input neighbor embeddings. Figure 9(b) reports the data transfer for the baselines. The results show that node reordering enables DGI to save up to 39.7% of the data traffic compared with Dynamic.

We include the comparison of different reorder algorithms and hardware choices in Appendix.

6 RELATED WORKS

GNN models. Early models such as GCN [27], GAT [43] and APPNP [28] are usually a simple linear stack of the graph convolution layers and differ in their aggregation function. Then, some GNN models are designed to facilitate efficient training. In particular, GraphSage [20] randomly down-samples the neighbors of each node when conducting graph convolution. FastGCN [7] and LADIES [64] limit the total number of sampled nodes in each layer. GraphSAINT [57], Cluster-GCN [11] and ShadowGNN [56] limit the computation of each training iteration to a subgraph. Many GNN models adopt more complex structures to improve model capacity. For instance, JKNet [51], ResGCN [31] and DenseGCN [31] allow the layers connect with each other flexibly. PNA [12] and HAG-Net [30] use multiple neighbor aggregation operations in each layer. MixHop [2] concatenates the features from different layers based on the power of the adjacency matrix. Heterogeneous GNNs [23, 41, 45] involve multiple types of graph convolutions and connect them according to data (e.g., node and edge label). A comprehensive survey of the design space for GNN modes is provided in [49, 55], and it is envisioned that GNN models will become even more complex. As a result, manually decomposing these GNN models for efficient layer-wise evaluation is challenging but our DGI provides a general and efficient solution.

GNN systems. DGL [44] and PyTorch Geometric (PyG) [13] are the two most widely used frameworks for GNN training. PyTorch-Direct [38] and TorchQuiver [1] train GNN on large graphs that exceed GPU memory by storing data on CPU memory and accessing them via CUDA unified virtual addressing (UVA). DGCL [4], DistDGL [62], Neugraph [36], Roc [25] and AliGraph [52] scale GNN training to multiple machines. There are many other systems that improve GNN training from different perspectives, e.g., GNNAdvisor [46], G^3 [32], Dorylus [42], ByteGNN [61], BGL [33], FuseGNN [9], SeaStar [50] and DSP [5]. Compared to GNN training, much fewer works consider GNN model evaluation. In particular, GCNP [63] computes node embedding efficiently by reducing the dimension of the intermediate embeddings. GNNAutoScale [14] uses the historical embeddings of the neighbors to conduct graph aggregation. These works design algorithms to trade accuracy for efficiency while our DGI is a system solution and runs exact evaluation. PCGraph [59] reduces data transfer overhead by caching node embeddings in GPU. Partition-based sparse matrix multiplication (SPMM) is used to run each layer of GNN models efficiently on FPGA [58]. These optimizations (i.e., embedding caching and layer execution) can also be integrated into DGI.

7 CONCLUSIONS AND FUTURE DIRECTION

We build DGI, a framework that makes model evaluation easy and efficient for GNN models. We observe that the popular node-wise evaluation approach is inefficient and present a layer-wise evaluation approach, which is efficient but difficult to program. To mitigate such programming difficulty, DGI automatically trace the computation graph for GNN models, partitions the computation graph for execution, and manages node batching. DGI is general for different GNN models and evaluation requests, and supports out-of-core execution for large graphs.

Currently, we assume that DGI works for a static graph. This is observed in many applications that periodically update the graph structure, GNN model and node embedding [35, 54]. A challenging scenario is to update the node embedding for dynamic graphs in real-time. In particular, the graph is receiving updates (e.g., edge insertion/deletion, node feature changes), node embedding needs to be updated accordingly for applications with high requirements on accuracy (e.g., recommendation). For an L-layer GNN model, a node/edge update can affect the embedding of its L-hop neighbors, which results in an enormous update set. We are considering theory/algorithm to limit the update set by allowing some approximations, and DGI can take the pruned update set as target nodes.

8 ACKNOWLEDGMENTS

We thank the reviewers for their valuable comments. This work was partially supported by CUHK direct grant 4055146, and the Guangdong Basic and Applied Basic Research Foundation (Grant No. 2021A1515110067). Bo Tang was supported by Shenzhen Fundamental Research Program (Grant No. 20220815112848002) and a research gift from Huawei. Bo Tang is also affiliated with the Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen, China.

REFERENCES

[1] 2021. Quiver. https://github.com/quiver-team/torch-quiver.

[2] Sami Abu-El-Haija, Bryan Perozzi, Amol Kapoor, Nazanin Alipourfard, Kristina Lerman, Hrayr Harutyunyan, Greg Ver Steeg, and Aram Galstyan. 2019. Mixhop: Higher-order graph convolutional architectures via sparsified neighborhood mixing. In *international conference on machine learning*. PMLR, 21–29.

[3] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. 2016. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 22–31.

[4] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: an efficient communication library for distributed GNN training. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 130–144.

[5] Zhenkun Cai, Qihui Zhou, Xiao Yan, Da Zheng, Xiang Song, Chenguang Zheng, James Cheng, and George Karypis. 2023. DSP: Efficient GNN training with multiple GPUs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 392–404.

[6] Wing-Man Chan and Alan George. 1980. A linear time implementation of the reverse Cuthill-McKee algorithm. *BIT Numerical Mathematics* 20, 1 (1980), 8–14.

[7] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *International Conference on Learning Representations*.

[8] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *International Conference on Machine Learning*. PMLR, 942–950.

[9] Zhaodong Chen, Mingyu Yan, Maohua Zhu, Lei Deng, Guoqi Li, Shuangchen Li, and Yuan Xie. 2020. fuseGNN: accelerating graph convolutional neural network training on GPGPU. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[10] Mark Cheung and José MF Moura. 2020. Graph Neural Networks for COVID-19 Drug Discovery. In *IEEE International Conference on Big Data (Big Data)*. IEEE, 5646–5648.

[11] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 257–266.

[12] Gabriele Corso, Luca Cavalleri, Dominique Beaini, Pietro Liò, and Petar Velickovic. 2020. Principal Neighbourhood Aggregation for Graph Nets. *CoRR* abs/2004.05718 (2020). arXiv:2004.05718 https://arxiv.org/abs/2004.05718

[13] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[14] Matthias Fey, Jan E Lenssen, Frank Weichert, and Jure Leskovec. 2021. Gnnautoscale: Scalable and expressive graph neural networks via historical embeddings. In *International Conference on Machine Learning*. PMLR, 3294–3304.

[15] Yanjie Gao, Yu Liu, Hongyu Zhang, Zhengxian Li, Yonghao Zhu, Haoxiang Lin, and Mao Yang. 2020. *Estimating GPU Memory Consumption of Deep Learning Models*. Association for Computing Machinery, New York, NY, USA, 1342–1352. https://doi.org/10.1145/3368089.3417050

[16] Alberto Garcia Duran and Mathias Niepert. 2017. Learning graph representations with embedding propagation. *Advances in neural information processing systems* 30 (2017).

[17] Thomas Gaudelet, Ben Day, Arian R Jamasb, Jyothish Soman, Cristian Regep, Gertrude Liu, Jeremy BR Hayter, Richard Vickers, Charles Roberts, Jian Tang, et al. 2021. Utilizing graph machine learning within drug discovery and development. *Briefings in bioinformatics* 22, 6 (2021), bbab159.

[18] Congcong Ge, Xiaoze Liu, Lu Chen, Baihua Zheng, and Yunjun Gao. 2022. LargeEA: Aligning Entities for Large-scale Knowledge Graphs. *PVLDB* 15, 2 (2022), 237–245.

[19] Takuo Hamaguchi, Hidekazu Oiwa, Masashi Shimbo, and Yuji Matsumoto. 2017. Knowledge transfer for out-of-knowledge-base entities: a graph neural network approach. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. 1802–1808.

[20] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 1025–1035.

[21] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362.

[22] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems* 33 (2020), 22118–22133.

[23] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. 2020. Heterogeneous graph transformer. In *Proceedings of The Web Conference*. 2704–2710.

[24] Kezhao Huang, Haitian Jiang, Minjie Wang, Guangxuan Xiao, David Wipf, Xiang Song, Quan Gan, Zengfeng Huang, Jidong Zhai, and Zheng Zhang. 2023. ReFresh: Reducing Memory Access from Exploiting Stable Historical Embeddings for Graph Neural Network Training. *arXiv preprint arXiv:2301.07482* (2023).

[25] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems* 2 (2020), 187–198.

[26] George Karypis and Vipin Kumar. 1997. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. (1997).

[27] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*.

[28] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. 2018. Predict then Propagate: Graph Neural Networks meet Personalized PageRank. In *International Conference on Learning Representations*.

[29] Jérôme Kunegis and Andreas Lommatzsch. 2009. Learning spectral graph transformations for link prediction. In *Proceedings of the 26th Annual International Conference on Machine Learning*. 561–568.

[30] Dawei Leng, Jinjiang Guo, Lurong Pan, Jie Li, and Xinyu Wang. 2021. Enhance Information Propagation for Graph Neural Network by Heterogeneous Aggregations. *arXiv preprint arXiv:2102.04064* (2021).

[31] Guohao Li, Matthias Müller, Ali K. Thabet, and Bernard Ghanem. 2019. Can GCNs Go as Deep as CNNs? *CoRR* abs/1904.03751 (2019). arXiv:1904.03751 http://arxiv.org/abs/1904.03751

[32] Husong Liu, Shengliang Lu, Xinyu Chen, and Bingsheng He. 2020. G3: when graph neural networks meet parallel graph processing systems on GPUs. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2813–2816.

[33] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. 2021. Bgl: Gpu-efficient gnn training by optimizing graph data i/o and preprocessing. *arXiv preprint arXiv:2112.08541* (2021).

[34] Xiaoze Liu, Junyang Wu, Tianyi Li, Lu Chen, and Yunjun Gao. 2023. Unsupervised Entity Alignment for Temporal Knowledge Graphs. In *WWW*.

[35] Ziqi Liu, Chaochao Chen, Xinxing Yang, Jun Zhou, Xiaolong Li, and Le Song. 2018. Heterogeneous graph neural networks for malicious account detection. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. 2077–2085.

[36] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. {NeuGraph}: Parallel Deep Neural Network Computation on Large Graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 443–458.

[37] Supriyo Mandal and Abyayananda Maiti. 2021. Graph Neural Networks for Heterogeneous Trust based Social Recommendation. In *International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[38] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. 2021. Pytorch-direct: Enabling gpu centric data access for very large graph neural network training with irregular accesses. *arXiv preprint arXiv:2101.07956* (2021).

[39] Kaspar Riesen and Horst Bunke. 2010. *Graph classification and clustering based on vector space embedding*. Vol. 77. World Scientific.

[40] Peter Sanders and Christian Schulz. 2013. Think locally, act globally: Highly balanced graph partitioning. In *International Symposium on Experimental Algorithms*. Springer, 164–175.

[41] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *European semantic web conference*. Springer, 593–607.

[42] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, et al. 2021. Dorylus: Affordable, Scalable, and Accurate {GNN} Training with Distributed {CPU} Servers and Serverless Threads. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 495–514.

[43] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rJXMpikCZ

[44] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315* (2019).

[45] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S Yu. 2019. Heterogeneous graph attention network. In *The world wide web conference*. 2022–2032.

[46] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2020. GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. *arXiv preprint arXiv:2006.06608* (2020).

[47] Zhouxia Wang, Tianshui Chen, Jimmy SJ Ren, Weihao Yu, Hui Cheng, and Liang Lin. 2018. Deep Reasoning with Knowledge Graph for Social Relationship Understanding. In *Proceedings of the 26th International Joint Conference on Artificial*

Intelligence.
[48] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*. 1813–1828.
[49] Lanning Wei, Huan Zhao, and Zhiqiang He. 2022. Designing the Topology of Graph Neural Networks: A Novel Feature Fusion Perspective. In *Proceedings of the ACM Web Conference 2022*. 1381–1391.
[50] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chenguang Zheng, James Cheng, and Fan Yu. 2021. Seastar: vertex-centric programming for graph neural networks. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 359–375.
[51] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. 2018. Representation learning on graphs with jumping knowledge networks. In *International Conference on Machine Learning*. PMLR, 5453–5462.
[52] Hongxia Yang. 2019. Aligraph: A comprehensive graph neural network platform. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 3165–3166.
[53] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213.
[54] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 974–983.
[55] Jiaxuan You, Rex Ying, and Jure Leskovec. 2020. Design Space for Graph Neural Networks. *CoRR* abs/2011.08843 (2020). arXiv:2011.08843 https://arxiv.org/abs/2011.08843
[56] Hanqing Zeng, Muhan Zhang, Yinglong Xia, Ajitesh Srivastava, Andrey Malevich, Rajgopal Kannan, Viktor Prasanna, Long Jin, and Ren Chen. 2021. Decoupling the Depth and Scope of Graph Neural Networks. In *Advances in Neural Information Processing Systems*, A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan (Eds.). https://openreview.net/forum?id=d0MtHWY0NZ
[57] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2019. Accurate, Efficient and Scalable Graph Embedding. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
[58] Bingyi Zhang, Hanqing Zeng, and Viktor Prasanna. 2020. Hardware acceleration of large scale gcn inference. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 61–68.
[59] Lizhi Zhang, Zhiquan Lai, Yu Tang, Dongsheng Li, Feng Liu, and Xiaochun Luo. 2021. PCGraph: Accelerating GNN Inference on Large Graphs via Partition Caching. In *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. IEEE, 279–287.
[60] Muhan Zhang and Yixin Chen. 2018. Link prediction based on graph neural networks. *Advances in neural information processing systems* 31 (2018).
[61] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. 2022. ByteGNN: efficient graph neural network training at large scale. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1228–1242.
[62] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. Distdgl: distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 36–44.
[63] Hongkuan Zhou, Ajitesh Srivastava, Hanqing Zeng, Rajgopal Kannan, and Viktor Prasanna. 2021. Accelerating large scale real-time GNN inference using channel pruning. *arXiv preprint arXiv:2105.04528* (2021).
[64] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. 2019. Layer-dependent importance sampling for training deep and large graph convolutional networks. *Advances in neural information processing systems* 32 (2019).

A AN EXAMPLE OF HAND-WRITTEN LAYER-WISE EVALUATION FOR JKNET

```
# We adopt the forward function of JKNet from the DGL
# library, and implement layer-wise evaluation for JKNet
# manually.
class JKNet(nn.Module):
    def __init__(...):
        ... # init for JKNet.

    def forward(self, g, x):
        hs = []
        h = x
        for layer in self.layers:
            h = layer(g, h)
            h = self.dropout(h)
            hs.append(h)
        h = torch.cat(hs, dim=-1)
        return self.conv(h)

    def evaluate(self, g, batch_sizes, x):
        feat_lst = []
        for l, layer in enumerate(self.layers):
            ret = torch.zeros(
                (g.num_nodes(), self.hidden))
            feat_lst.append(ret)
            dl = NodeDataLoader(
                batch_size=batch_sizes[0], ...)
            for in_nodes, out_nodes, blocks in dataloader:
                block = blocks[0].to("GPU")
                h = x[in_nodes].to("GPU")
                h = layer(block, h)
                h = self.dropout(h)
                feat_lst[-1][out_nodes] =  h.cpu()
            x = feat_lst[-1]

        ret = torch.zeros(
            (g.num_nodes(), self.n_classes))
        dl = NodeDataLoader(
                batch_size=batch_sizes[1], ...)
        for in_nodes, out_nodes, blocks in dataloader:
            block = blocks[0].to("GPU")
            h_lst = []
            for feat in feat_lst:
                h_lst.append(feat[in_nodes].to("GPU"))
                h = torch.cat(h_lst, dim=-1)
                out_val = self.conv(h)
            ret[out_nodes] = out_val.cpu()
        return ret
```

Listing 1: Simplified handwritten code for the layer-wise evaluation of JKNet.

```
graph = ... # Graph data
x = ... # Node features
nids = torch.arange(graph.num_nodes())
model = JKNet(...)

# Evaluation by \sys.
from dgl.dataloading import FullNeighborSampler
from dgi import AutoEvaluator
eval = AutoEvaluator(model,
            targets=nids,
            sampler=FullNeighborSampler(1),
            computing_device="GPU:0",
            external_device="CPU")
# Here takes the same arguments as
# forward function in JKNet.
pred = eval.evaluate(graph, x)
```

Listing 2: Code for using DGI for layer-wise inference of JKNet.

To demonstrate the challenges of coding layer-wise evaluation and the usability of DGI, Listing 1 shows how to use handwritten code to conduct layer-wise evaluation for JKNet. For the forward function, JKNet first computes each layer's embeddings and records them in an array. Finally, it concretes all embeddings and uses an additional graph convolution to perform the output. It consists of only 8 LOCs (i.e., Line 8-16 in Listing 1) and node-wise influence is easy to implement by directly using the forward function. However, layer-wise evaluation needs approximately 30 LOCs (from Line 18 in Listing 1) even the code is already simplified by removing some of the complex and troublesome API calls. Line 20 first enumerate convolution layers, and perform layer-wise evaluation. For the additional graph convolution layer in JKNet, we need to write another layer-wise code for it (line 38 - 45). Note that the final layer of JKNet (i.e., the jumping knowledge layer) aggregates the output embedding of all preceding layers. The handwritten code needs to explicitly consider whether UVA memory is used in the CPU and manually configure the batch size for the dataloader. If data are stored on SSD, the handwritten code will be even more complex. Besides, the evaluation speed of layer-wise execution is sensitive to

Table 6: Execution time of DGI (in seconds) on different machines. The models have 3 layers, and cases that store data on SSD are marked in bold. The first number is the capacity of main memory.

Machine	OGBN-Papers100M			Friendster		
	GCN	GAT	JKNet	GCN	GAT	JKNet
128 GB + 16 GB T4	**1828**	**2840**	**2030**	**3070**	**4510**	**3220**
192 GB + 16 GB T4	**1590**	**2360**	**1730**	**2660**	**3930**	**3030**
488 GB + 16 GB V100	425	732	498	582	1290	801
1.5 TB + 32 GB V100	277	644	325	356	822	459

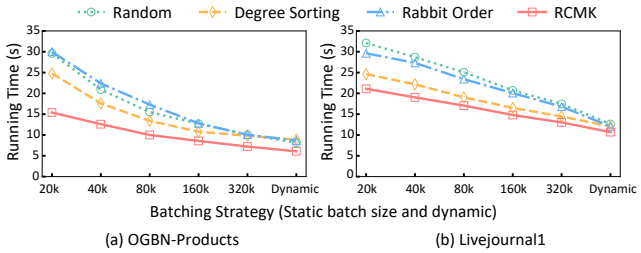


Figure 11: DGI using different node reordering methods. The experiments are conducted using 3-layer GCN on V100-16GB GPU.

Table 5: Comparation of DGI and sampling-based node-wise evaluation for 3-layer GCN on *OGBN-Products*.

	Fanout	Num. Votes	Run. Time (s)	Test Acc. (%)
DGI-full	-	-	6.10	76.49
DGI-sampling	10	-	4.57	75.31
Node-wise	10	10	107.75	75.77
Node-wise	10	5	53.87	75.66
Node-wise	10	1	10.77	75.23
Node-wise	5	10	59.99	74.83
Node-wise	5	5	30.04	74.42
Node-wise	5	1	5.99	73.15

multiple factors such as the size of each node batch and the order to process nodes.

Listing 2 shows how to perform layer-wise evaluation using DGI. DGI provide interface that users only need to input the model to initialize an *AutoInferencer*, which will automatically trace the split the forward function and pack to the evaluate function. Then, users only need to call the evaluation function provide by *AutoEvaluator*, which takes the arguments as the same as the model forward function. DGI completely frees the user from having to write all this tedious code.

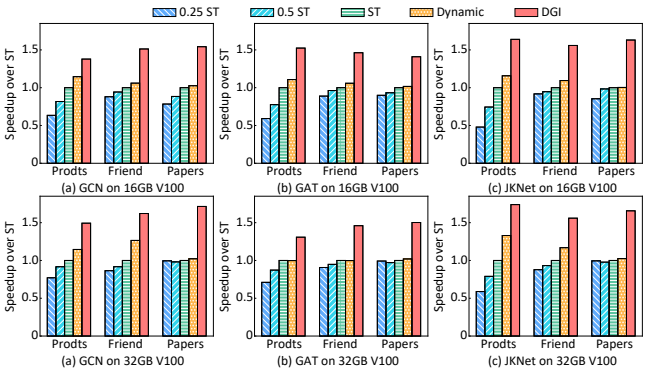


Figure 10: Speedup of dynamic batch size and node ordering in DGI over using the best static batch size found by profiling.

B SAMPLED-BASED EVALUATION

Instead of exact evaluation with the complete neighbor set, some users may use neighbor sampling for evaluation to trade accuracy for efficiency. We compare the running time and test accuracy of DGI-full, DGI-sampling and node-wise sampling evaluation in Table 5. For sampling-based evaluation, we experiment with different numbers of neighbors (fanout, e.g., 5 and 10) to sample, and conduct evaluation multiple times and merge the results for each node by voting to improve test accuracy. The results show that full-neighbor evaluation achieves considerably higher accuracy (up to 3.34%) than sampling evaluation but runs only slightly longer than sampling evaluation on DGI. Even with node sampling, node-wise evaluation runs slower than DGI-full in most cases.

C MORE EXPERIMENTS COMPARING THE DGI VARIANTS

We compare the DGI variants (introduced in Figure 8 of the main paper) on the V100 GPU with 32GB memory in Figure 10. To observe the influence of GPU memory, we also include the results on the V100 GPU with 16GB memory alongside. The results show that dynamic batch size and node ordering are effective for both GPU configurations—*Dynamic* performs close to or better than *ST*, and DGI significantly outperforms *Dynamic* with node ordering. The performance gap among different batch sizes (i.e., *0.25ST*, *0.5ST* and *ST*) is smaller for 32GB GPU memory than 16GB GPU memory because 32GB memory allows larger batch sizes for all baselines and the gains of input sharing diminish with batch size.

D HARDWARE CHOICES

Cloud providers (e.g., AWS, Azure and Alibaba) offer machine instances with different memory capacities and GPUs, and DGI can adapt to different instances by storing data in either SSD or main memory. We do not conduct the experiment DGL-manual since DGL does not have the ability to perform computation on SSD. An interesting question is how to select a proper machine instance to conduct GNN evaluation. We report such results in Table 6, which show that evaluation time significantly reduces when all data can be stored in main memory. Even if data does not fit in main memory, larger memory capacity still reduces evaluation time.

E REORDER ALGORITHMS

We compare RCMK with 3 alternatives: (i) *Random*, which randomly shuffles the nodes; (ii) *Degree Sorting*, which sorts the nodes by their in-degrees; (iii) Rabbit Order [3], which uses hierarchical community detection to determine node order. Figure 11 shows that RCMK consistently yields shorter evaluation time than the other methods, which justifies using RCMK in DGI. This is because RCMK uses BFS to assign node IDs, and nodes with adjacent IDs are likely to share input nodes. We conjecture that Rabbit Order has poor performance because it focuses on improving graph locality by storing neighbors in CPU caches for graph algorithms. Thus, it generates small neighborhood and is not suitable for GNN evaluation.