AGL: A Scalable System for Industrial-purpose Graph Machine Learning

Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, Yuan Qi

Ant Financial Services Group, Hangzhou, China

{dalong.zdl, huangxi.hx, ziqiliu, jun.zhoujun, zhiyang.hzhy, xianzheng.sxz, zhibang.zg, fred.wl,,lingyao.zzq, yuan.qi}@antfin.com

ABSTRACT

Machine learning over graphs has been emerging as powerful learning tools for graph data. However, it is challenging for industrial communities to leverage the techniques, such as graph neural networks (GNNs), and solve real-world problems at scale because of inherent data dependency in the graphs. As such, we cannot simply train a GNN with classic learning systems, for instance, parameter server that assumes data parallelism. Existing systems store the graph data in-memory for fast accesses either in a single machine or graph stores from remote. The major drawbacks are three-fold. First, they cannot scale because of the limitations on the volume of the memories, or the bandwidth between graph stores and workers. Second, they require extra development of graph stores without well exploiting mature infrastructures such as MapReduce that guarantee good system properties. Third, they focus on training but ignore optimizing the performance of inference over graphs, thus makes them an unintegrated system.

In this paper, we design AGL, a scalable and integrated system, with fully-functional training and inference for GNNs. Our system design follows the message passing scheme underlying the computations of GNNs. We design to generate the K-hop neighborhood, an information-complete subgraph for each node, as well as do the inference simply by merging values from in-edge neighbors and propagating values to out-edge neighbors via MapReduce. In addition, the K-hop neighborhood contains information-complete subgraphs for each node, thus we simply do the training on parameter servers due to data independence. Our system AGL, implemented on mature infrastructures, can finish the training of a 2-layer GNN on a graph with billions of nodes and hundred billions of edges in 14 hours, and complete the inference in 1.2 hours.

PVLDB Reference Format:
Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, Yuan Qi.

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415539

AGL: a Scalable System for Industrial-purpose Graph Machine Learning. *PVLDB*, 13(12): 3125-3137, 2020.
DOI: https://doi.org/10.14778/3415478.3415539

1. INTRODUCTION

In recent years, both the industrial and academic communities have paid much more attention to machine learning over graph structure data. The ***G**raph **M**achine **L**earning* (abbreviated as GML) not only claims successes in traditional graph mining tasks (e.g., node classifications [11, 19, 8, 13], link property predictions [26] and graph property predictions [2, 23]), but also brings great improvement to the tasks of other domains (e.g., knowledge graph [7, 21], NLP [28], Computer Vision [6, 16], etc.). Besides, more and more Internet companies have applied the GML technique in solving various industrial problems and achieved great successes (e.g., recommendation [27, 25], marketing [15], fraud detection [14, 9], loan default prediction [22], etc.).

To use graph machine learning techniques to solve real-world problems by leveraging industrial-scale graphs, we are required to build a learning system with *scalability*, *fault tolerance*, and *integrality* of the fully-functional training/inference workload. However, the computation graph of graph machine learning tasks are fundamentally different from traditional learning tasks because of *data dependency*. That is, the computation graph of each sample is independent of other samples in existing classic parameter server frameworks [29] assuming *data parallelism*, while the computation graph of each node in graph learning tasks is dependent on the K-hop neighbors of that node. The data dependency in graph learning tasks makes that we can no longer store the samples in disks and access them through pipelines [29]. Instead, we have to store the graph data in-memory for fast data accesses. This makes us fail to simply build a learning and inference system for graph learning tasks based on existing parameter server architectures that simply maintain the model consistency in parameter servers and do the workload in each worker parallelly.

However, the real industrial graph data could be huge. The social graph in Facebook[1] includes over two billion nodes and over a trillion edges [12, 3]. The heterogenous financial graph in Ant Financial[2] contains billions of nodes

[1] `https://en.wikipedia.org/wiki/Facebook,_Inc.`
[2] `https://en.wikipedia.org/wiki/Alipay`

and hundreds of billion edges with rich attribute information, as well as the e-commerce graph in Alibaba[3]. The graph data at this scale may result into 100 TB of data by counting features associated with those nodes and edges. Those data are infeasible to be stored in a single machine like DGL. Furthermore, the communications between the graph storage engine storing the graphs and features associated with nodes and edges, and workers could be huge. This requires a well-structured network with high enough bandwidth.

To summarize, firstly, existing industrial designs of learning systems require the in-memory storage of graph data either in a single monster machine that could not handle real industrial-scale graph data, or in a customized graph store that could lead to a huge amount of communications between graph stores and workers. This makes them *not scale* to larger graph data. Second, they do *not well exploit the classic infrastructures*, such as MapReduce or parameter servers, for fault tolerance purposes. In real production environment, there could be thousands of graph learning tasks running everyday, and the fault tolerence and failure recovery of graph services are critical. Third, most of existing frameworks pay more attentions to the training of graph learning models, but *ignore the system integrality*, for example, optimizing the performance of inference tasks when deploying graph machine learning models.

Take all those concerns into considerations, we build AGL (**A**nt **G**raph machine **L**earning system), an integrated system for industrial-purpose graph learning. The key insight of our system design is based on the *message passing (merging and propagation)* scheme underlying the computation graph of graph neural networks.

In the phase of training graph neural networks, we propose to construct *K-hop neighborhood* that provides information-complete subgraphs for computing each node's K-hop embeddings based on message passing by *merging* neighbors from in-edges and *propagating* merged information to neighbors along out-edges. The benefit of decomposing the original graph into tiny pieces of subgraphs, i.e. K-hop neighborhood, is that the computation graph of each node is *independent* of other nodes *again*. That means we can still enjoy the properties of fault tolerance, flexible model consistency from classic parameter server frameworks without extra efforts on maintaining the graph stores [24].

In the inference phase of graph neural networks, we propose to split a well trained k-layer graph neural networks into k slices plus one slice related to the prediction model. We do message passing by first merging the k-th layer embedding from each node's in-edge neighbors, then propagating embeddings to their out-edge neighbors, with k starts from 1-st slice to k-th slice.

We abstract all the message passing schemes in training and inference, and implement them simply using MapReduce [4]. Since both MapReduce and parameter servers have been developed as infrastructures commonly in industrial companies, our system for graph machine learning tasks can still benefit the properties like fault tolerance and scalibility even with commodity machines which are cheap and widely used. Moreover, compared with the inference based on architectures like DGL and AliGraph, the implementation of our inference maximally utilizes each nodes' embeddings, so as to significantly boost inference jobs. Besides, we propose several techniques to accelerate the floating point calculations in training procedures from model level to operator level. As a result, we successfully accelerate the training of GNNs in a single machine compared with DGL/PyG, and achieve a near-linear speedup with a CPU cluster in real product scenarios.

It's worth noting that, when working on a graph with 6.23×10^9 nodes and 3.38×10^{11} edges, AGL can finish the training of a 2-layer GAT model with 1.2×10^8 target nodes in 14 hours (7 epochs until convergence, 100 workers), and completes the inference on the whole graph in only 1.2 hours. To our best knowledge, this is the largest-ever application of graph embeddings and proves the high scalability and efficiency of our system in real industrial scenarios.

2. RELATED WORKS

In this section, we discuss related works that aim to design graph learning systems.

Early efforts have been made to make full use of computation resources (CPU, GPU, Memory, and so on) on a single machine to efficiently train a GNN model. Based on message passing, Deep Graph Library (DGL) [20] and PyTorch Geometric (PyG) [5] are designed to utilize both CPUs and GPUs. However, they can hardly scale to industrial-scale graphs, since those graphs are usually attributed with rich features and can not fit in a single machine. Inspired by GraphSage[8], PinSage[25] perform localized convolutions by sampling the neighborhood around a node, and design a MapReduce pipeline to efficiently run inference tasks. However, it has the same scalability limitations with DGL and PyG, since PinSage is also deployed on a single machine.

Recently, there's a trend to design GML systems in the distributed manner. Facebook presents PyTorch-BigGraph (PBG) [12], a large-scale *network embedding* system, which aims to produce unsupervised node embedding from multi-relation data. However, PBG is not suitable for plenty of real-world scenarios, in which graphs have rich attributes over nodes and edges (called *attributed graph*). AliGraph[24] implements distributed in-memory graph storage engine, and in training phase, workers will query subgraphs related to a batch of nodes, and do the training workloads. However, the network bandwidth could be a bottleneck when huge among of subgraphs are requested by lots of workers in parallel. Moreover, in industrial scenarios, there could have many graph learning tasks running everyday. It could be expensive to store so many graph data in memory. As a result, it is still challenging to build an an efficient and scalable GML system for industrial GML purposes.

3. PRELIMINARIES

In this section, we introduce some notations, and highlight the fundamental computation paradigm, i.e. message passing, in graph neural networks (GNN). Finally, we introduce the concept of K-hop neighborhood to help realize the data independency in graph learning tasks. Both of the abstraction of message passing scheme and K-hop neighborhood play an important role in the design of our system.

3.1 Notations

A *directed and weighted attributed graph* can be defined as $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, \mathbf{A}, \mathbf{X}, \mathbf{E}\}$, where $\mathcal{V}$ and $\mathcal{E} \in \mathcal{V} \times \mathcal{V}$ are the node set

[3] https://en.wikipedia.org/wiki/Alibaba_Group

and edge set of $\mathcal{G}$, respectively. $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$ is the sparse weighted adjacent matrix such that its element $\mathbf{A}_{v,u} > 0$ represents the weight of a directed edge from node u to node v (i.e., $(v,u) \in \mathcal{E}$), and $\mathbf{A}_{v,u} = 0$ represents there is no edge (i.e., $(v,u) \notin \mathcal{E}$). $\mathbf{X} \in \mathbb{R}^{|\mathcal{V}|\times f^n}$ is a matrix consisting of all nodes' f^n-dimensional feature vectors, and $\mathbf{E} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|\times f^e}$ is a sparse tensor consisting of all edges' f^e-dimensional feature vectors. Specifically, $\mathbf{x}_v$ denotes the feature vector of v, $\mathbf{e}_{v,u}$ denotes the feature vector of edge (v,u) if $(v,u) \in \mathcal{E}$, otherwise $\mathbf{e}_{v,u} = \mathbf{0}$. In our setting, an *undirected graph* is treated as a special directed graph, in which each undirected edge (v,u) is decomposed as two directed edges with the same edge feature, i.e., (v,u) and (u,v). Moreover, we use $\mathcal{N}_v^+$ to denote the set of nodes directly pointing at v, i.e., $\mathcal{N}_v^+ = \{u : \mathbf{A}_{v,u} > 0\}$, $\mathcal{N}_v^-$ to denote the set of nodes directly pointed by v, i.e., $\mathcal{N}_v^- = \{u : \mathbf{A}_{u,v} > 0\}$, and $\mathcal{N}_v = \mathcal{N}_v^+ \cup \mathcal{N}_v^-$. In other words, $\mathcal{N}_v^+$ denotes the set of in-edge neighbors of v, while $\mathcal{N}_v^-$ denotes the set of out-edge neighbors of v. We call the edges pointing at a certain node as its *in-edges*, while the edges pointed by this node as its *out-edges*.

3.2 Graph Neural Networks

Most GML models aim to encode a graph structure (e.g., node, edge, subgraph or the entire graph) as a low dimensional embedding, which is used as the input of the downstream machine learning tasks, in an end-to-end or decoupled manner. The proposed AGL mainly focuses on GNNs, which is a category of GML models widely-used. Each layer of GNNs generates the intermediate embedding by aggregating the information of target node's in-edge neighbors. After stacking several GNN layers, we obtain the final embedding, which integrate the entire receptive field of the targeted node. Specifically, we abstract the computation paradigm of the k^{th} GNN layer as follows:

$$\mathbf{h}_v^{(k+1)} = \phi^{(k)}(\{\mathbf{h}_i^{(k)}\}_{i\in\{v\}\cup\mathcal{N}_v^+}, \{\mathbf{e}_{v,u}\}_{\mathbf{A}_{v,u}>0}; \mathbf{W}_\phi^{(k)}), \quad (1)$$

where $\mathbf{h}_v^{(k)}$ denotes node v's intermediate embedding in the k^{th} layer and $\mathbf{h}_v^{(0)} = \mathbf{x}_v$. The function $\phi^{(k)}$ parameterized by $\mathbf{W}_\phi^{(k)}$, takes the embeddings of v and its in-edge neighbors $\mathcal{N}_v^+$, as well as the edge features associated with v's in-edges as inputs, and outputs the embedding for the next GNN layer.

The above computations of GNNs can be formulated in the message passing paradigm. That is, we collect keys (i.e., node ids) and their values (i.e., embeddings). We first merge all the values from each node's in-edge neighbors to have the new values for the nodes. After that, we propagate the new values to destination nodes via out-edges. After K times of such merging and propagation, we complete the computation of GNNs. We will discuss in the following sections that such a paradigm will be generalized to the training and inference of GNNs.

3.3 K-hop Neighborhood

Definition 1. ***K-hop neighborhood***. The K-hop neighborhood w.r.t. a targeted node v, denoted as $\mathcal{G}_v^K$, is defined as the *induced attributed subgraph* of $\mathcal{G}$ whose node set is $\mathcal{V}_v^K = \{v\} \cup \{u : d(v,u) \le K\}$, where $d(v,u)$ denotes the length of the shortest path from u to v. Its edge set consists of the edges in $\mathcal{E}$ that have both endpoints in its node set, i.e. $\mathcal{E}_v^K = \{(u,u') : (u,u') \in \mathcal{E} \wedge u \in \mathcal{V}_v^K \wedge u' \in \mathcal{V}_v^K\}$. Moreover, it contains the feature vectors of the nodes and edges in the K-hop neighborhood, $\mathbf{X}_v^K$ and $\mathbf{E}_v^K$. Without loss of generality, we define the 0-hop neighborhood w.r.t. v as the node v itself.

The following theorem shows the connection between the computation of GNNs and the K-hop neighborhood.

Theorem 1. Let $\mathcal{G}_v^K$ be the K-hop neighborhood w.r.t. the target node v, then $\mathcal{G}_v^K$ contains the **sufficient and necessary information** for a K layers GNN model, which follows the paradigm of Equation 1, to generate the embedding of node v.

First, the 0^{th} layer embedding is directly assigned by the raw feature, i.e., $\mathbf{h}_v^{(0)} = \mathbf{x}_v$, which is also the 0-hop neighborhood. And then, from Equation 1, it's easy to find that the output embedding of v in each subsequent layer is generated only based on the embedding of the 1-hop in-edge neighbors w.r.t. v from the previous layer. Therefore, by applying mathematical induction, it's easy to prove Theorem 1. Moreover, we can extend the theorem to a batch of nodes. That is, the intersection of the K-hop neighborhoods w.r.t. a batch of nodes provides the sufficient and necessary information for a K layers GNN model to generate all the node embeddings in the batch. This simple theorem implies that in a K layers GNN model the target node's embedding at the K^{th} layer only depends on its K-hop neighborhood, rather than the entire graph.

4. SYSTEM

In this section, we first give an overview of our AGL system. Then, we elaborate on three core modules, i.e., GraphFlat, GraphTrainer, and GraphInfer. At last, we give a demo example on how to implement a simple GCN[11] model with the proposed AGL system.

4.1 System Overview

Our major motivation for building AGL is that the industrial communities desiderate an *integrated* system of fully-functional training/inference over graph data, with *scalability*, and in the meanwhile has the properties of *fault tolerance* based on *mature industrial infrastructures* like MapReduce, parameter servers, etc. That is, instead of requiring a single monster machine or customized graph stores with huge memory and high bandwidth networks, which could be expensive for Internet companies to upgrade their infrastructures, we sought to give a solution based on mature and classic infrastructures, which is ease-to-deploy while enjoying various properties like fault tolerance and so on. Second, we need the solution based on mature infrastructures scale to industrial-scale graph data. Third, besides the optimization of training, we aim to boost the inference tasks over graphs because labeled data are very limited (say ten million) in practice compared with unlabeled data, typically billions of nodes, to be inferred.

The principle of designing AGL is based on the message passing scheme underlying the computations of GNNs. That is, we first merge all the information from each node's in-edge neighbors, and then propagate the merged information to the destination nodes via out-edges. We repeatedly apply such a principle to the training and inference processes, and develop *GraphFlat* and *GraphInfer*. Basically, GraphFlat is

to generate independent K-hop neighborhoods in the training process, while GraphInfer is to infer nodes' embeddings given a well trained GNN model.

Based on the motivation and design principle, the proposed AGL leverages several powerful parallel architectures, such as MapReduce and Parameter Server, to build each of its components with exquisitely-designed distributed implementations. As a result, even being deployed on the clusters with machines that have relatively low computing capacity and limited memory, AGL gains comparable effectiveness and higher efficiency against several state-of-the-art systems. Moreover, it has the ability to perform fully-functional graph machine learning over the industrial-scale graph with billions of nodes and hundred billions of edges.

Figure 1 depicts the system architecture of AGL, which consists of three modules:

(1) **GraphFlat**. GraphFlat is an efficient and distributed generator, based on message passing, for generating K-hop neighborhoods that contain information complete subgraphs of each targeted nodes. Those tiny K-hop neighborhoods are flattened to a protobuf strings[4] and stored on a distributed file system. Since the K-hop neighborhood contains sufficient and necessary information for each targeted node, we can load one or a batch of them rather than the entire graph into memory, and do the training similar to any other traditional learning methods. Besides, we propose a re-indexing technique together with a sampling framework to handle "hub" nodes in real-world applications. Our design is based on the observation that the amount of labeled nodes is limited, and we can store those K-hop neighborhoods associated with the labeled nodes in disk without too much cost.

(2) **GraphTrainer**. Based on the data independency guaranteed by GraphFlat, GraphTrainer leverages many techniques, such as pipeline, pruning, and edge-partition, to eliminate the overhead on I/O and optimize the floating point calculations during the training of GNN models. As a result, GraphTrainer gains a high near-linear speedup in real industrial scenarios even on a generic CPU cluster with commodity machines.

(3) **GraphInfer**. We develop GraphInfer, a distributed inference module that splits K layer GNN models into K slices, and applies the message passing K times based on MapReduce. GraphInfer maximally utilizes the embedding of each node because all the intermediate embedding at the k-th layer will be propagated to next round of message passing. This significantly boosts the inference tasks.

Details about our system will be presented in the following sections.

4.2 GraphFlat: Distributed Generator of k-hop Neighborhood

The major issue of training graph neural networks is the inherent data dependency among graph data. To do the feedforward computation of each node, we have to read its associated neighbors and neighbors' neighbors, and so on so forth. This makes us fail to deploy such network architecture simply based on existing parameter server learning frameworks. Moreover, developing extra graph stores for the query of each node's subgraphs is expensive for most of industrial companies. That is, such a design would not benefit us with existing commonly deployed infrastructures that are mature and guarantee various properties like fault tolerance.

[4] https://en.wikipedia.org/wiki/Protocol_Buffers

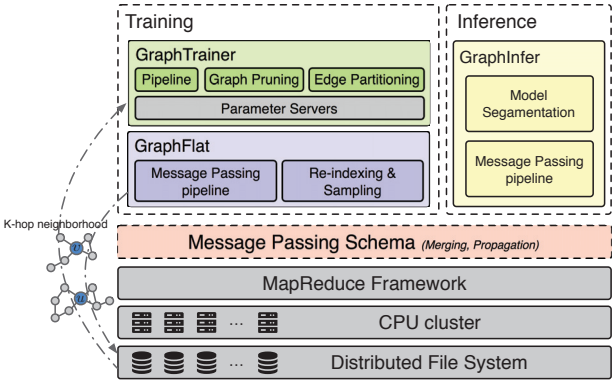

Figure 1: **System architecture of AGL.**

Fortunately, according to Theorem 1, the K-hop neighborhood w.r.t. a target node provides sufficient and necessary information to generate the K^{th}-layer node embedding. Therefore, we can divide an industrial-scale graph into massive of tiny K-hop neighborhoods *w.r.t.* their target nodes in advance, and load one or a batch of them rather than the entire graph into memory in the training phase. Following this idea, we develop GraphFlat, an efficient distributed generator for the K-hop neighborhood. Moreover, we further introduce a re-indexing strategy and design a sampling framework to handle "hub" nodes and ensures the load balance of GraphFlat. The details are presented as follows.

4.2.1 Distributed pipeline to generate K-hop neighborhood

In this section, we design a distributed pipeline to generate K-hop neighborhoods in the spirit of message passing, and implement it with MapReduce infrastructure.

Figure 2 illustrates the workflow of the proposed pipeline. The key insight behind is that, for a certain node v, we first *receive and merge* the information from the in-edge neighbors $\mathcal{N}_v^+$ pointing at v, then *propagate* the merged results to the out-edge neighbors $\mathcal{N}_v^-$ pointed by v. By repeating this procedure k times, we finally get the K-hop neighborhoods.

Assume that we take a *node table* and an *edge table* as input. Specifically, the *node table* consists of node ids and node features, while the *edge table* consists of source node ids, destination node ids, and the edge features. The overall pipeline to generate the K-hop neighborhood can be summarized as follows:

(1) **Map**. The *Map* phase runs only once at the beginning of the pipeline. For a certain node, the *Map* phase generates three kinds of information, i.e., the self information (i.e., node features), the in-edge information (i.e., features of the in-edge, and the neighbor node) and the out-edge information (i.e., features of the out-edge). Note that we set the node id as the *shuffle key* and the various information as the *value* for the following *Reduce* phase.

(2) **Reduce**. The *Reduce* phase runs K times to generate the K-hop neighborhood. In the k^{th} round, a reducer

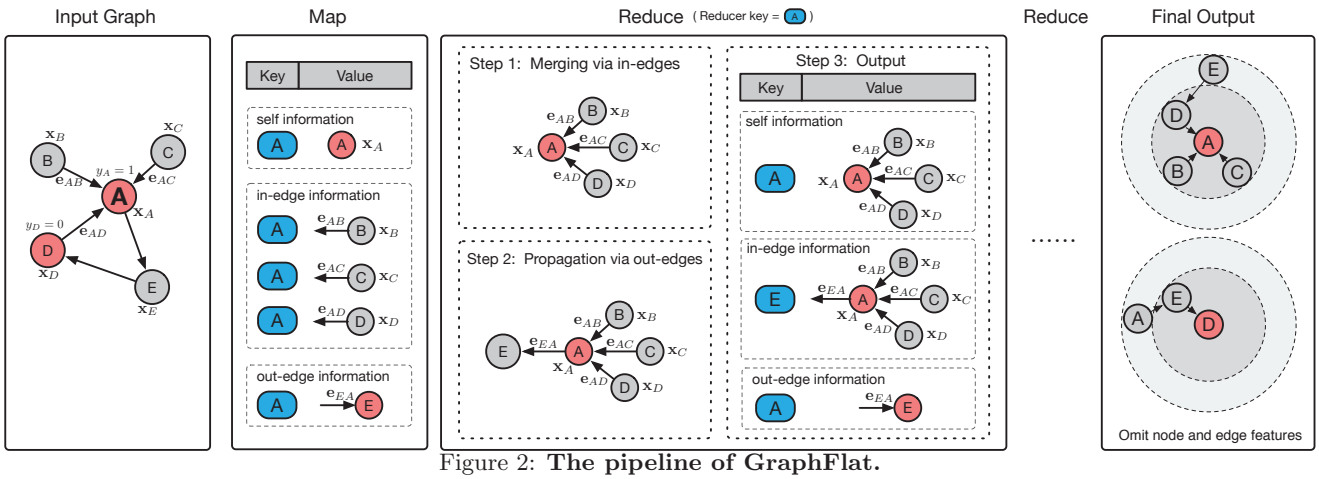

Figure 2: **The pipeline of GraphFlat.**

first collects all *values* (i.e., three kinds of information) with the same *shuffle key* (i.e., the same node ids), then *merges* the self information and the in-edge information as its new self information. Note that the new self information become the node's K-hop neighborhood. Next, the new self information is *propagated* to other destination nodes pointed along the out-edges, and is used to construct the new in-edge information w.r.t. the destination nodes. All of the out-edge information remain unchanged for the next *reduce* phase. At last, the reducer outputs the new data records, with the node ids and the updated information as the new *shuffle key* and *value* respectively, to the disk.

(3) **Storing**. After K *Reduce* phases, the final self information becomes the K-hop neighborhood. We transform the self information of all targeted nodes into the protobuf strings and store them into the distributed filesystem.

Throughout the MapReduce pipeline, the key operations are *merging* and *propagation*. In each round, given a node v, we merge its self information and in-edge information from last round, and the merged results serve as the self information of v. We then propagate the new self information via out-edges to the destination nodes. At the end of this pipeline, the K-hop neighborhood w.r.t. a certain targeted node is flattened to a protobuf string. That's why we call this pipeline **GraphFlat**. Note that, since the K-hop neighborhood *w.r.t.* to a node helps discriminate the node from others, we also call it *GraphFeature*.

4.2.2 *Sampling & Indexing*

The distributed pipeline described in the previous subsection works well in most cases. However, the degree distribution of the graphs can be skewed due to the existence of "hub" nodes, especially in the industrial scenario. This makes some of the K-hop neighborhoods may cover almost the entire graph. On the one hand, in the *Reduce* phase of GraphFlat, reducers that process such "hub" nodes could be much slower than others thus damage the load balances of GraphFlat. On the other hand, the huge K-hop neighborhoods w.r.t. those "hub" nodes may cause the Out Of Memory (OOM) problem in both GraphFlat and the downstream model training. Moreover, the skewed data may also lead to poor accuracy of the trained GNN model. Hence, we employ the re-indexing strategy and design a sampling framework for reducer in GraphFlat.

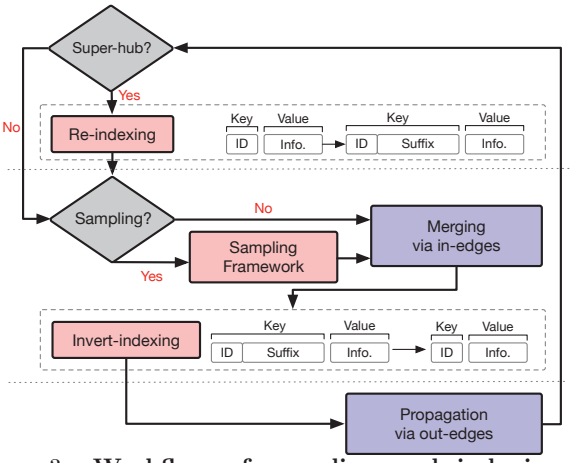

Figure 3: **Workflow of sampling and indexing in GraphFlat.**

Figure 3 illustrates the reducer with re-indexing and sampling strategies in GraphFlat. Three key components of performing re-indexing and sampling are introduced as follows:

- **Re-indexing**. When the in-degree of a certain *shuffle key* (i.e., node id) exceeds a pre-defined threshold (like 10k), we will update *shuffle keys* by appending random suffixes, which is used to randomly partition the data records with the original *shuffle key* into smaller pieces.

- **Sampling framework**. We build a distributed sampling framework and implement a set of sampling strategies (e.g., uniform sampling, weighted sampling), to reduce the scale of the K-hop neighborhoods, especially for those "hub" nodes.

- **Inverted indexing**. This component is responsible for replacing the reindexed *shuffle key* with the original *shuffle key*. After that, the data records are outputted to the disk waiting for the downstream task.

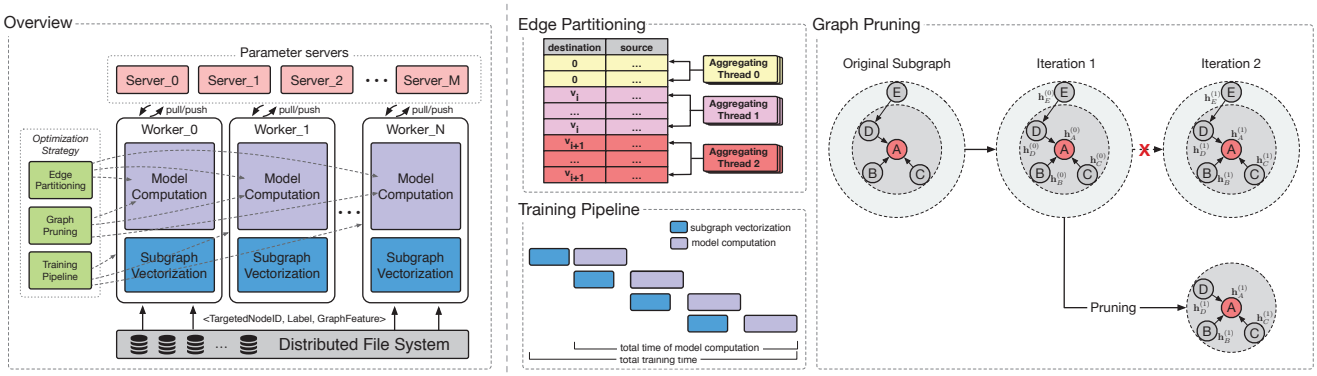

Figure 4: **Training workflow and optimization strategies.**

Before sampling, the *re-indexing* component is to uniformly map data records associated with the same "hub" node to a set of reducers. It helps alleviate the load balance problem that could be caused by those "hub" nodes. Then the sampling framework samples a portion of the data records *w.r.t.* a *shuffle key*. After that, the merging and propagation operations are performed as the original *Reducer* does. Next, the *inverted indexing* component will recover the reindexed *shuffle key* as the original *shuffle key* (i.e., node id) for the downstream task.

With re-indexing we make the process of "hub" nodes being partitioned over a set of reducers, thus well maintain the load balances. With sampling, the scale of K-hop neighborhoods is decreased to an acceptable size.

4.2.3 More Discussions on GrpahFlat

As we have discussed, we propose GraphFlat to store sufficient and nesseary information for the computation of each node in a K layers GNN model. In this part, we discuss the variance of the sampling and space cost in GraphFlat. Without loss of generality, typically we have the following k-th GNN layer:

$$\mathbf{h}_v^{k+1} = \phi^{(k)}\Big(\sum_{i \in \mathcal{N}_v^+} a_{vi}\mathbf{h}_i^{(k)}\Big). \tag{2}$$

where we simply assume the commonly used sum aggregator [11, 8], and a_{ij} as the weight. We can recast the evaluation of Eq.(2) in its expectation form, then approximate the evaluation using Monte Carlo estimates:

$$\begin{aligned}\mathbf{h}_v^{k+1} &= \phi^{(k)}(N(v)\mathbb{E}_{p_v}[\mathbf{h}_i^{(k)}]) \\ &\approx \phi^{(k)}(N(v)\frac{1}{n}\sum_{i_s}\mathbf{h}_{i_s}^{(k)}),\ \ i_s \sim p_v,\ p_{vj} \propto a_{vj}\end{aligned} \tag{3}$$

where we let $N(v) = \sum_{i \in \mathcal{N}_v^+} a_{vi}$, and n as the sample size. Let we denote N_v^+ as the in-degree of vertex v, and typically we have $a_{vi} = \frac{1}{N_v^+}$ for all $i \in \mathcal{N}_v^+$, i.e. the mean aggregator in GraphSAGE [8]. Thus we have $N(v) = 1$ for all vertex $v \in \mathcal{V}$. Using Monte Carlo estimates, we have the following variance of the estimate as: $\sigma = \frac{1}{n}\mathbb{E}[\|\frac{1}{n}\sum_{i_s} h_{i_s}^{(k)} - \frac{1}{N_v^+}\sum_{i \in \backslash_v^+}\mathbf{h}_i^{(k)}\|^2]$. Since the embeddings will be normalized in unit balls after the non-linear transformations at each layer [8], then we simply have $\sigma \leq \frac{1}{n}(4 + \frac{1}{n} + \frac{1}{N_v^+})$. Using Chebyshev's inequalit, we have: $p(|\hat{\mu} - \sum_{i \in \mathcal{N}_v^+} a_{vi}\mathbf{h}_i^{(k)}| > t) \leq \frac{\sigma}{t^2}$ for all $t > 0$, where we let $\hat{\mu}$ denote our estimate $\frac{1}{n}\sum_{i_s}\mathbf{h}_{i_s}^{(k)}$. Note that the variance σ decreases linearly as sample size n increases, and the deviation of $\hat{\mu}$ from the ground truth decreases with rate $O(\frac{1}{t^2})$. Empirically, we found that with a moderate number of sample size, e.g. $n = 20$, the sample can approximate the ground truth sum aggregator well enough. As such, we simply maintain a sampled K-hop neighborhood with a moderate sample size for each node.

Assuming a K-hop neighborhood, and sparse node features with the number of non-zero values as F. If we assume the in-degree of the graph is upper bounded by the sample size, i.e. n as discussed above. Then we have the worst space cost for storing each node in disk as $O(n^k \times F)$. Due to the feature sparsity in practice, the value F is commonly a small constant. The major bottleneck of our space cost comes from the term $O(n^k)$. In practice we set $k \leq 3$ since Deeper GNNs cannot generalize much better [11], which is reasonable because the diameter of real-world connected graphs is small, typically upper bounded by the log of number of nodes [1]. As such, the term $O(n^k)$ will be well bounded with a moderate sample size.

4.3 GraphTrainer: Distributed Graph Training Framework

We implement GraphTrainer, an efficient distributed graph training framework that is shown in Figure 4. The overall architecture of GraphTrainer follows the designs of parameter server, which consists of two sets of components: the workers that perform the bulk of computation during model training, and the servers that maintain the current version of the graph model parameters. Since the K-hop neighborhood contains sufficient and necessary information to train the GNN model, the training workers of GraphTrainer become independent of each other. They just have to process their own partitions of training data, and do not need extra communications with other workers. Therefore, the training of a GNN model becomes similar to the training of a conventional machine learning model, in which the training data on each worker is self-contained. Moreover, since most K-hop neighborhoods are tiny subgraphs taking little memory footprints, training workers in GraphTrainer only require to be deployed on the commodity machines with limited computation resources (i.e., CPU, memory, network bandwidth).

Considering the property of the K-hop neighborhood as well as the characteristics of GNN training computation, we propose several optimization strategies, including training

pipeline, graph pruning, and edge partitioning, to improve the training efficiency. The rest of this subsection first introduce the overall training workflow, and then elaborate several graph-specific optimization strategies.

4.3.1 Training workflow

As shown in Figure 4, the training workflow mainly includes two phases, i.e., subgraph vectorization and model computation. We take the node classification task as an example to illustrate the two phases. In the node classification task, a batch of training examples can be formulated as a set of triples $\mathcal{B} = \{< TargetedNodeId, Label, GraphFeature >\}$. Different from the training process of the conventional machine learning models, which directly performs model computation, the training process of GNNs has to merge the subgraphs described by *GraphFeatures* together, and then vectorize the merged subgraph as the following three matrices.

- **Adjacency matrix**: $\mathbf{A}_{\mathcal{B}}$. A sparse matrix with nodes and edges of the merged subgraph. Edges in the sparse matrix are sorted by their destination nodes.
- **Node feature matrix**: $\mathbf{X}_{\mathcal{B}}$. A matrix to record the features of all nodes in the merged subgraph.
- **Edge feature matrix**: $\mathbf{E}_{\mathcal{B}}$. A matrix to record the features of all edges in the merged subgraph.

Note that these three matrices contain all information of the K-hop neighborhood w.r.t. all targeted nodes in $\mathcal{B}$. They will be fed to the model computation phase, together with the node ids and labels. Based on the three matrices as well as the ids and labels of targeted nodes, the model computation phase is responsible for performing the forward and backward calculations.

4.3.2 Optimization strategies

In this subsection, we will elaborate three graph-specific optimization strategies in different level, to boost the training efficiency. That is training pipeline (batch-level), graph pruning (graph-level) and edge partitioning (edge-level).

Training pipeline. During GNN model training, each worker first read a batch of its training data from the disks, then it performs *subgraph vectorization* and *model computation*. Performing these steps sequentially is time-consuming. To address this problem, we build a pipeline that consists of two stages: preprocessing stage including data reading and subgraph vectorization, and model computation stage. The two stages operate in a parallel manner. Since the time consumed by the preprocessing stage is relatively shorter than that of the model computation stage, after several rounds, the total training time is nearly equal to that of performing model computation only.

Graph pruning. Given the three matrices $\mathbf{A}_{\mathcal{B}}$, $\mathbf{X}_{\mathcal{B}}$, and $\mathbf{E}_{\mathcal{B}}$ w.r.t. batch $\mathcal{B}$, we revise Equation 1 w.r.t. $\mathcal{B}$ as follows:

$$\mathbf{H}_{\mathcal{B}}^{(k+1)} = \Phi^{(k)}(\mathbf{H}_{\mathcal{B}}^{(k)}, \mathbf{A}_{\mathcal{B}}, \mathbf{E}_{\mathcal{B}}; \mathbf{W}_{\Phi}^{(k)}), \tag{4}$$

where $\mathbf{H}_{\mathcal{B}}^{(k)}$ denotes the k^{th}-layer intermediate embeddings of all nodes that appear in the K-hop neighborhood w.r.t. all targeted nodes in $\mathcal{B}$, and $\Phi^{(k)}$ denotes the aggregating function of the k^{th} layer. We assume that the final embedding is the K^{th}-layer embedding, i.e., $\mathbf{H}_{\mathcal{B}}^{(K)}$.

However, Equation 4 contains many unnecessary computations. On one hand, only the targeted nodes of $\mathcal{B}$ are labeled. Their embedding will be fed to the following part of the model. That means other embeddings in $\mathbf{H}_{\mathcal{B}}^{(K)}$ are unnecessary to the following part of the model. On the other hand, the three matrices $\mathbf{A}_{\mathcal{B}}$, $\mathbf{X}_{\mathcal{B}}$ and $\mathbf{E}_{\mathcal{B}}$ can provide sufficient and necessary information only for the targeted nodes. Thus other embeddings in $\mathbf{H}_{\mathcal{B}}^{(K)}$ could be generated incorrectly due to the lack of sufficient information.

Tackling this problem, we propose a *graph pruning* strategy to reduce the unnecessary computations mentioned above. Given a targeted node v, for any node u, we use $d(v, u)$ to denote the number of edges in the shortest path from u to v. Given a batch of targeted nodes $\mathcal{V}_{\mathcal{B}}$, for any node u, we define the distance between u and $\mathcal{V}_{\mathcal{B}}$ as $d(\mathcal{V}_{\mathcal{B}}, u) = \min(\{d(v, u)\}_{v \in \mathcal{V}_{\mathcal{B}}})$. After going deep into the computation paradigm of GNN models, we have the following observation. *Given the k^{th}-layer embedding, the receptive field of the next $(k+1)^{th}$-layer embedding become the 1-hop neighborhood*. This observation motivates us to prune unnecessary nodes and edges from $\mathbf{A}_{\mathcal{B}}$. Specifically, in the k^{th} layer, we prune every node u with $d(\mathcal{V}_{\mathcal{B}}, u) > K - k + 1$, as well as its associated edges, from $\mathbf{A}_{\mathcal{B}}$ to generate a pruned adjacent matrix $\mathbf{A}_{\mathcal{B}}^{(k)}$. Therefore, Equation 4 is revised as follows:

$$\mathbf{H}_{\mathcal{B}}^{(k+1)} = \Phi^{(k)}(\mathbf{H}_{\mathcal{B}}^{(k)}, \mathbf{A}_{\mathcal{B}}^{(k)}, \mathbf{E}_{\mathcal{B}}; \mathbf{W}_{\Phi}^{(k)}). \tag{5}$$

Note that if we treat the adjacency matrix as a sparse tensor, only non-zero values are involved in model computation. Essentially, the graph pruning strategy is to reduce the non-zero values in the adjacency matrix of each layer. Therefore, it truly helps reduce unnecessary computations for most GNN algorithms. Moreover, each $\mathbf{A}_{\mathcal{B}}^{(k)}$ can be precomputed in the subgraph vectorization phase. With the help of the training pipeline strategy, it takes nearly no extra time to perform graph pruning. The right part of Figure 4 gives a toy example to illustrate the graph pruning strategy w.r.t. one targeted node (i.e., node A).

Edge partitioning. As shown in Equation 5, the aggregator $\Phi^{(k)}$ is responsible for aggregating information for each node along its edges in the sparse adjacent matrix $\mathbf{A}_{\mathcal{B}}^{(k)}$. Several aggregation operators, such as sparse matrix multiplication, will be applied very frequently during the model computation phase, which makes the optimization of aggregation become very essential for the GML system. However, the conventional deep learning frameworks (e.g., TensorFlow, PyTorch) seldom address this issue since they are not specially designed for the GML technique.

Tackling this problem, we propose an edge partitioning strategy to perform graph aggregation in parallel. The key insight is that a node only aggregates information along the edges pointing at it. If all edges with the same destination node can be handle with the same thread, the multi-thread aggregation could be very efficient since there will be no conflicts between any two threads. To achieve this goal, we partition the sparse adjacent matrix into t parts and ensure that the edges with the same destination node (i.e., the entries in the same row) fall in the same partition. The edge partitioning strategy is illustrated at the top of the middle part of Figure 4.

After edge partitioning, each partition will be handle with a thread to perform aggregation independently. On one hand, the number of nodes in a batch of training examples is usually much larger than the number of threads. On the

other hand, the number of neighbors for each node (i.e., the number of non-zero entries in each row) will not be too large after applying sampling in GraphFlat. Therefore, the multi-thread aggregation can achieve load balancing thus gains a significant speedup when training GNN models.

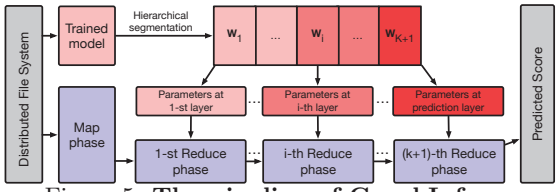

Figure 5: **The pipeline of GraphInfer.**

4.4 GraphInfer: distributed framework for GNN model inference

Performing the GNN model inference over industrial-scale graphs could be an intractable problem. On one hand, the data scale and use frequency of inference tasks could be quite higher than that of training tasks in industrial scenarios, which require a well-designed inference framework to boost the efficiency of inference tasks. On the other hand, since different K-hop neighborhoods described by *GraphFeatures* could overlap with each other, directly performing inference on *GraphFeatures* could lead to redundant computations that are time-consuming.

Hence, we develop GraphInfer, a distributed framework for GNN model inference over huge graphs by following the message passing scheme. We first perform hierarchical model segmentation to split a well-trained K-layer GNN model into $K+1$ slices in terms of the model hierarchy. Then, based on the message passing scheme, we develop a MapReduce pipeline to infer with different slices in the order from lower layers to higher layers. Specifically, the k^{th} *Reduce* phase loads the k^{th} model slice, *merges* the embeddings of the last layer from in-edge neighbors to generate intermediate embeddings of the k^{th} layer, and *propagate* those intermediate embeddings via the out-edges to the destination nodes for the next *Reduce* phase. Figure 5 describes the overall architecture of GraphInfer, which can be summarized as follows:

1. **Hierarchical model segmentation**. A K layers GNN model is split into $K+1$ slices in terms of the model hierarchy. Specifically, the k^{th} slice ($k \leq K$) consists of all parameters of the k^{th} GNN layer, while the $K+1^{\text{th}}$ slice consists of all parameters of the final prediction model.

2. **Map**. Similar to GraphFlat, the *Map* phase here also runs only once at the beginning of the pipeline. For a certain node, the *Map* phase also generates three kinds of information, i.e., the self information, the in-edge information and the out-edge information, respectively. Then, the node id is set as the "shuffle key" and the various information as the "value" for the following *Reduce* phase.

3. **Reduce**. The *Reduce* phase runs $K+1$ times in which the former K rounds are to generate the K^{th}-layer node embedding while the last round to perform the final prediction. For the former K rounds, a reducer acts similar to that in GraphFlat. In the merging stage, instead of generating k-hop neighborhood, the reducer here loads its model slice to infer the node embedding based on the self information and in-edge information, and set the result as the new self information. The last *Reduce* phase is responsible to infer the final predicted score and output it as the inference results.

There is no redundant computation in the above pipeline, which reduces the time cost to a great extent.

4.5 Demonstration

Figure 6 demonstrates how to use AGL to perform data generation with GraphFlat, model training with GraphTrainer, and inference with GraphInfer. In addition, we also give an example on how to implement a simple GCN model.

For each module stated in subsection 4.3, we provide a well-encapsulated interface respectively. *GraphFlat* is to transform raw inputs into K-hop neighborhoods. Users only need to choose a sampling strategy and prepare a node table together with an edge table, to generates K-hop neighborhoods w.r.t. their target nodes. Those K-hop neighborhoods are the inputs of GraphTrainer and are formulated as $\mathcal{B} = \{< TargetedNodeId, Label, GraphFeature >\}$ as stated in subsection 4.3. Then, by feeding *GraphTrainer* a set of configurations like the model name, input, distributed training settings (the number of workers and parameter servers) and so on, a GNN model will be trained distributedly on the cluster. For a certain model, *GraphTrainer* will perform the two-stage pipelines (i.e., *subgraph vectorization* and *model computation*) in the "producer-consumer" manner. After that, *GraphInfer* will load the well-trained model together with the inference data to perform the inference procedure. In this way, developers only need to focus on the implementation of the GNN model.

Here, we take GCN as an example to show how to develop GNN models in AGL. In "GCNLayer", by calling the *aggregator* function, the information will be aggregated to target nodes from their direct neighbors according to Equation 1. Then, by stacking k "GCNLayer"s, we build a GCN model. The *GraphTrainer* will call this model and feed it vectorized subgraphs to train the model. It's quite simple and just like coding for a standalone application.

5. EXPERIMENT

In this section, we conduct extensive experiments to evaluate the proposed AGL system.

5.1 Experimental settings

5.1.1 Datasets.

We employ four datasets in our experiments, including three public datasets (Cora[18], PPI[30], Ogbn product[10]) and an industrial-scale social graph provided by Alipay[5] (called UUG, **U**ser-**U**ser **G**raph).

- **Cora.** Cora is a citation network with 2708 nodes and 5429 edges. Each node is associated with 1433 dimensional features and belongs to one of seven classes.

- **PPI.** PPI is a protein-protein interaction dataset, which consists of 24 independent graphs with 56944

[5]https://www.alipay.com/

```
###########GraphFlat###########
GraphFlat -n node_table -e edge_table -h hops -s sampling_strategy;

###########GraphTrainer###########
GraphTrainer -m model_name -i input -c training_configs;

###########GraphInfer###########
GraphInfer -m model -i input -c infer_configs;

###########Model File###########
class GCNModel:
  def __init__(...)
  # configuration and init weights
  ...
  def call(adj_list, node_feature, ...):
    # initial node_embedding with raw node_feature,  like:
    # node_embedding = node_feature
    ....
    # multi-layers
    for k in range(multi_layers):
      node_embedding = GCNlayer(adj_list[k], node_embedding)
      # other process like dropout
      ...
    target_node_embedding = look_up(node_embedding, targetID)
    return target_node_embedding

...

class GCNLayer:
  def __init__(...)
  # configuration and init weights
  ...
  def call(self, adj, node_embedding):

    # some preprocess
    ...
    # aggregator with edge_partition
    node_embedding = aggregator(adj, node_embedding)
  return node_embedding
```

Figure 6: **A demo example of using AGL.**

nodes and 818716 edges in total. Each node contains 50-dimensional features and belongs to several of 121 classes (multi-label problem).

- **Ogbn-product.** The dataset ogbn-products represens an Amazon product co-purchasing network. It consists of 2,449,029 nodes and 61,859,140 edges. Nodes represent products sold in Amazon, and edges between two products indicate that the products are purchased together. Note that, Each node in this dataset is attributed with 100-dimensional features and belongs to one of 47 classes.

- **UUG.** UUG[6] consists of massive social relations collected from various scenarios of Alipay, in which nodes represent users and edges represent various kinds of interactions between users. It contains as many as 6.23×10^9 nodes and 3.38×10^{11} edges. Nodes are described with 656-dimensional features and alternatively belong to two classes. To our best knowledge, it is the largest attributed graph for GML tasks in the literature.

Following the experimental settings in [11, 19, 8, 10], *Cora*, *PPI*, and *Ogbn-product* are divided into three parts as the training, validation, and test set, respectively. For the *UUG* dataset, 1.2×10^8 nodes out of 1.5×10^8 labeled nodes are set as the training set, while 5×10^6 and 1.5×10^7 are set as the validation and test set, respectively. Details about those three datasets are summarized in Table 1.

[6]The data set does not contain any Personal Identifiable Information (PII). The data set is desensitized and encrypted. Adequate data protection was carried out during the experiment to prevent the risk of data copy leakage, and the data set was destroyed after the experiment. The data set is only used for academic research, it does not represent any real business situation.

Table 1: **Summary of datasets and model configurations**

Indices	Cora	PPI	Ogbn	UUG
#Nodes	2,708	56,944	2,449,029	6.23×10^9
#Edges	5,429	818,716	61,859,140	3.38×10^{11}
#feature	1,433	50	100	656
#Classes	7	121	47	2
#Train	140	44,906	196,615	1.2×10^8
#Valid	500	6,514	39,323	5×10^6
#Test	1,000	5,524	2,213,091	1.5×10^7
#Layers	2	3	3	2
Embedding	16	64	256	8
#Epochs	200	200	20	10

5.1.2 *Evaluation*

We design a set of experiments to verify the effectiveness, efficiency, and scalability of our system. Several famous open-source GML systems are used for comparison:

- **DGL** [20]. Deep Graph Library (DGL) is a package that interfaces between existing tensor-oriented frameworks (e.g., PyTorch and MXNet) and the graph structured data.

- **PyG** [5]. PyTorch Geometric (PyG) is a library for deep learning on irregularly structured input data such as graphs, point clouds and manifolds, built upon PyTorch[17].

- **AliGraph** [24]. AliGraph (also named Graph-Learn now) is a framework designed to simplify the application of graph neural networks(GNNs). It not only can operate on a single machine, but also can be deployed in distributed mode.

Configuration. We first evaluate three widely-used GNNs (i.e., GCN[11], GAT[19], and GraphSAGE[8]) on two public datasets (i.e., Cora and PPI) for AGL and those three GML systems in standalone mode, respectively. Configurations, such as the number of layers, embedding size, and training epochs, are illustrated in Table 1. We record average results after 10 runs for each experiment to mitigate variance. For a fair comparison, we tune hyperparameters (e.g., learning rate, dropout ratio, etc.) for those GNNs by comprehensively referring to the details reported in [20, 24, 5] together with official examples and guidelines of DGL, PyG, and AliGraph. A container (Intel Xeon E5-2682 v4@2.50GHz) is exclusively used for each system to maintain the same running environment.

Then, we conduct a set of experiments on Ogbn-product to test the distributed mode for AGL and AliGraph. Meanwhile, we use an industrial datatset, UUG, to verify the scalability of our system. Configurations for models on those two datasets also can be found in Table 1. We deploy AGL and AliGraph on a CPU cluster consisting of more than one thousand machines (each machine is powered by a 32-core CPU with 64G memory and 200G HDD). In training phase, we use 10 workers to train models on Ogbn-product, and 100 workers for UUG. Note that, the cluster used here is not exclusively used, and different tasks may be running on

Table 2: **Effectiveness of different GNNs trained with different systems**

Datasets	Methods	Base	PyG	DGL	AliGraph	AGL
Cora (Accuracy)	GCN	0.813	0.818	0.811	0.802	0.811
	GAT	0.830	0.831	0.828	0.823	0.830
PPI (micro-F1)	GraphSage	0.598	0.632	0.636	–	0.635
	GAT	0.973	0.983	0.976	–	0.977
Ogbn (Accuracy)	GCN	0.757	–	–	0.723	0.744
	GraphSage	0.780	–	–	0.745	0.775
UUG (AUC)	GCN	–	–	–	–	0.681
	GraphSage	–	–	–	–	0.708
	GAT	–	–	–	–	0.867

this cluster at the same time, which is common in industrial environment.

Metrics. We conduct experimental evaluations from several different aspects.

We demonstrate the *effectiveness* of AGL by reporting *accuracy* and *micro-F_1* score on Cora and PPI in standalone mode, while illustrate the *accuracy* on Ogbn-product in distributed mode, compared with other GML systems.

Meanwhile, we report the average *time cost* per epoch in the training phase on PPI to show the *training efficiency* of those systems. Specially, we present the *convergence curves* on Ogbn-product to analyze the *training efficiency* in distributed mode.

We use UUG to verify the *scalability* of our system. We train a node classification model with the UUG dataset, and perform the inference task over the whole UUG dataset. By reporting the *time cost* of both the training and inference phases, we demonstrate the superior *efficiency* of our proposal in the industrial scenario. Moreover, we illustrate the *convergence curves* and the *speedup ratio* analyze the *scalability* of our system.

5.2 Results and Analysis

In this section, we present experimental results together with some analysis following the evaluation protocol in subsection 5.1.

5.2.1 Evaluation on public datasets.

We present and analyze results on Cora, PPI, Ogbn product compared with DGL, PyG, and AliGraph, to demonstrate the *effectiveness* and the *efficiency* of AGL.

Effectiveness. Table 2 illustrates the performance comparisons between AGL and other GML systems.

We report results of GCN, GraphSage, GAT in different GML systems (i.e., AGL, DGL, PyG, AliGraph) on Cora and PPI to analyze the performance of AGL in standalone mode. In most case, the bias of performance is less than 0.01 for all those three algorithms in different systems. This demonstrates that all those systems achieve comparable results on those two datasets, which are the basic benchmarks in this field. Note that, since the multi-label task is not well supported by AliGraph, results on PPI for AliGraph (v0.1) are not include in this table.

Furthermore, we also present results on Ogbn-product to analyze the performance of AGL in distributed mode. It's obvious that AGL achieves better results compared with AliGraph. Note that, the result of GraphSage in AGL is at the same level with the best result[7] reported in [10], which proves the effectiveness of AGL.

Efficiency. We present time cost on PPI in standalone mode, and show results of AGL with different optimization strategies stated in subsection 4.3 (i.e., graph pruning and edge partitioning) to analyze the effect of those strategies. Furthermore, we draw convergence curves and report training speeds on Ogbn-product to verify the *efficiency* of AGL in distributed mode.

Table 3 reports the average time cost per epoch in training phase and demonstrates a gifted speed of AGL in standalone mode. Generally, in the training phase, our system ($\text{AGL}_{+\text{pruning\&partition}}$) achieves a $\mathbf{5}\times \sim \mathbf{13}\times$ speedup compared with PyG, and a $\mathbf{1.2}\times \sim \mathbf{3.5}\times$ speedup compared with DGL. For all three GNN models at different depths, the performance of our system is superior to the other two systems to varying degrees. Specially, compared with PyG, our system achieves the biggest improvement, i.e., a $\mathbf{7}\times \sim \mathbf{13}\times$ speedup, when training GraphSAGE models. Compared with DGL, when training 1-layer GNN models, our system also gains significant improvement, i.e., a $\mathbf{2.5}\times \sim \mathbf{3.5}\times$ speedup.

Moreover, we verify the superiority of the proposed optimization strategies, i.e., graph pruning and edge partitioning, in Table 3. AGL_{Base} means training only with the pipeline strategy, while $\text{AGL}_{+\text{pruning}}$, $\text{AGL}_{+\text{partition}}$, and $\text{AGL}_{+\text{pruning\&partition}}$ represent training with graph pruning strategy, edge partition strategy, and both of them, respectively. Results can be summarized as follows: First, either the graph pruning strategy or the edge partitioning strategy works consistently well on different GNN models, which is proved by comparing the result of $\text{AGL}_{+\text{pruning}}$ or $\text{AGL}_{+\text{patition}}$ with that of $\text{AGL}_{+\text{base}}$. Furthermore, when comparing the result of $\text{AGL}_{+\text{pruning}}$ or $\text{AGL}_{+\text{patition}}$ with that of $\text{AGL}_{+\text{pruning\&partition}}$, we observe that a significant improvement is achieved by combining these two optimization strategies together. Second, these two strategies individually lead to different results in different situations. The edge partitioning strategy achieves better speedup when applied in GCN and GraphSAGE than in GAT, while the graph pruning strategy doesn't work in training 1-layer GNN model but demonstrates its power when training deeper GNN models.

These observations are benefit with two strategies. The graph pruning strategy aims to mitigate unnecessary computations by reducing edges that won't be used to propagate information to target nodes. Meanwhile, the edge partitioning strategy is to speed up the aggregation step in parallel. Since those two strategies optimize some key steps of training GNN models, their advantages benefit the training of GNN models in general, but may fail in some special cases. For example, if we train a 1-layer GNN model, it's reasonable that the pruning strategy won't work, as every edge plays a role in propagating information to target nodes and there is no unnecessary computation. Moreover, if a model consists of more dense computations (like computing attentions) than aggregating information via edges, those strategies will be weakened, since the dense computation takes the most of the total time cost.

Furthermore, we also illustrate convergence curves and training speed to demonstrate the *efficiency* in distributed

[7]trained in full batch mode (each batch contains the full training set) rather than in mini-batch mode

Table 3: **Time cost(s) per epoch on PPI in Standalone mode**

	GCN			GraphSAGE			GAT		
	1-layer	2-layer	3-layer	1-layer	2-layer	3-layer	1-layer	2-layer	3-layer
PyG	3.49	6.43	9.62	4.47	6.98	10.15	44.29	65.32	85.21
DGL	1.09	1.35	1.62	1.14	1.39	1.64	16.14	21.47	26.03
AGL_{base}	0.48	2.75	4.10	0.46	2.47	3.94	4.75	25.72	36.86
$\text{AGL}_{\text{+pruning}}$	0.48	1.93	3.23	0.46	1.67	2.99	4.75	13.88	20.01
$\text{AGL}_{\text{+partition}}$	0.42	1.22	1.60	0.34	0.97	1.39	4.63	22.65	33.45
$\text{AGL}_{\text{+pruning\&partition}}$	**0.42**	**1.13**	**1.52**	**0.34**	**0.88**	**1.35**	**4.63**	**13.73**	**18.63**

Table 4: **Inference efficiency on User-User Graph.**

Methods	Phase	Time cost (s)	CPU cost (core*min)	Memory cost (GB*min)
Original	GraphFlat	13,454	436,016	654,024
	Forward propagation	5,760	93,240	1,053,150
	Total	18,214	529,256	1,707,174
GraphInfer	Total	4,423	267,764	401,646

mode. Taking GraphSage on Ogbn-product as an example, the left part in Figure 7 shows convergence curves compared with AliGraph, while the right part illustrates the training speed of them. We train the model with 10 workers and 10 parameters servers and for each GraphSage layer, we sample 20 neighbors for a certain node. With the sample configuration, it's clear that AGL achieves a better convergence with fewer training epochs. With GraphFlat, the surrounding environments of nodes (K-hops neighborhood) are fixed in AGL, no matter in training, validation, or testing phase. However, in AliGraph, the surrounding environment for a certain node may vary due to runtime sampling. We think that's why AGL converges better with training epochs compared with AliGraph.

Meanwhile, AGL also demonstrate a superior training speed in Figure 7. Since AGL supports both dense and sparse modes, we test them in the comparisons with AliGraph (only dense feature supported). Also, taking GraphSage on Ogbn-product as an example, with 100-dimensional dense feature, it's obvious that AGL takes less time cost (per batch) compared with AliGraph. When training the Ogbn-product with 100-dimensional dense feature in sparse mode, AGL is only slightly slower than AliGraph. Furthermore, when we expand 100-dimensional dense feature to 1000-dimensional sparse feature by appending zeros, AGL shows significantly superiority in training speed. Figure 7 further proves the *efficiency* of AGL.

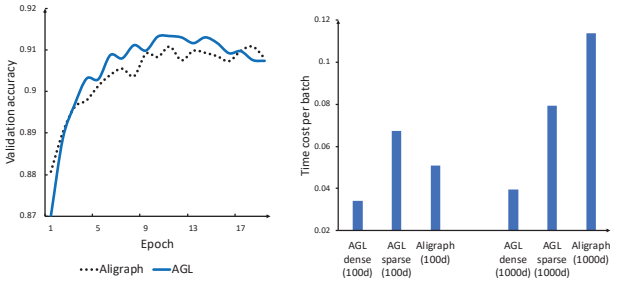

Figure 7: **Convergence curves and time cost (per batch) on Ogbn-product dataset in distributed mode.**

5.2.2 *Evaluation on Industrial Dataset*

We implement the proposed system using MapReduce and parameter server framework, and deploy it on a CPU cluster consisting of more than one thousand machines (each machine is powered by a 32-core CPU with 64G memory and 200G HDD). Then, we conduct experiments on the industrial dataset, i.e., UUG dataset, to demonstrate the scalability and efficiency of the proposed system in industrial scenarios.

Performance. Performance always matters most. Table 2 describes results of three different GNN models (i.e., GCN, GraphSage, GAT) on UUG. We get comparable results for GCN and GraphSAGE, but witness a significant improvement for GAT. We think it is reasonable, since the GAT model learns different weights for neighbors, which may play different roles (i.e., friend, colleague, and so on) w.r.t. their targeted node and have different influences on it. To our best effort, we failed to deploy other systems on UUG (OOM or other problems). Therefore, their results are not included.

Industrial training. Scalability is one of the most important criterion for industrial GML systems. In this subsection, we focus on evaluating the training scalability of AGL on two aspects, i.e., *convergence* and *speedup*. To do that, we train a GAT model on the industrial UUG dataset with different number of workers and report the results of convergence and speedup in Figure 8 and Figure 9, respectively. Moreover, we also illustrate time cost per batch by varying batch size and the number of neighbors in Figure 10.

Convergence. Figure 8 demonstrates the training scalability of our system in terms of convergence. Its y-axis denotes the AUC of GNN model, while the x-axis denotes the number of training epochs. In general, our system eventually converges to the same level of AUC regardless of the num-

ber of training workers. As shown in Figure 8, though more training epochs are required in the distributed mode, the convergence curves finally reach the same level of AUC as that trained with a single worker. Hence, the model effectiveness is guaranteed under distributed training, which verifies that our system can scale to industrial graphs without considering convergence.

Speedup. We also demonstrate the training scalability in terms of speedup ratio. As shown in Figure 9, our system achieves a near-linear speedup with slope ratio of 0.8, which means that if you double the number of training workers, you will get **1.8**× faster. In the experiment, we scale the number of training workers form 1 to 100 with 10 intervals. As a result, our system achieves a constantly high speedup and the slope ratio hardly decreases. For example, when the number of training workers reaches 100, we have **78**× faster, which is only slightly lower than the expected value 80. Note that, all these experiments are conducted on a cluster in the real production environment. There may exist different tasks operating on the same physical machine. The overhead in network communication may slightly increase as the number of training workers increases, causing perturbations in the slope ratio of the speedup curve. That again proves the robustness of our system in the industrial scenario.

We also analyze the training speed by varying batch sizes and the number of neighbors in a 2-hop GAT model. Figure 10 demonstrates a linear increasing of time cost when enlarging a training batch, while shows an exponential growth as the number of neighbors increases. Theoretically, the scale of K-hops neighbors is expanded exponentially with the average number of neighbors as the base while k as the exponent. Therefore, the total computations and time cost follows an exponential growth along with the number of neighbors. However, when enlarging the batch size in training time, different K-hops neighborhoods will be merged, which reduces some computations for overlapping parts. As a result, though we witness a near-linear increasing of time cost (for a batch), the ratio is smaller than that of batch size, which eventually helps speed up the full training phase.

It's worth noting that, it only takes about 14 hours to train a 2-layer GAT model on UUG until it converges to a stable state. Specifically, in our experiment, the GraphFlat takes about 3.7 hours with 1000 workers to generate GraphFeature, while the GraphTrainer takes about 10 hours with only 100 workers on the CPU clusters to train a GAT model. The total pipeline can be finished in 14 hours, which is remarkable for industrial applications. Furthermore, during the training phase, the training task only needs 5.5 GB memory for each worker (550 GB in total), which is far less than the memory cost for storing the entire graph (35.5 TB).

In summary, thanks to its ingenious architectural design, the proposed AGL meets the industrial scalability requirements for training GNN models over industrial graphs.

Industrial inference. We evaluate the efficiency of GraphInfer over the entire User-User Graph, which consists of 6.23×10^9 nodes and 3.38×10^{11} edges. In Table 4, we report the time and resource consumed by such an inference task. Since no GML system can handle such a large scale graph, we compare GraphInfer with the original inference module based on *GraphFeature*. Note that, all these experiments are operated with the same concurrency, i.e., 1000 workers.

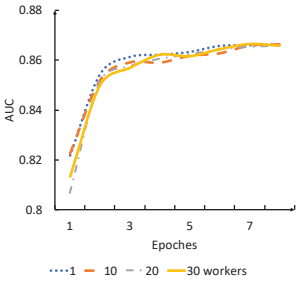

Figure 8: **Convergence**

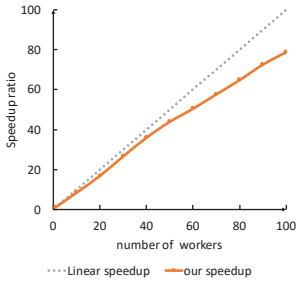

Figure 9: **Speedup**

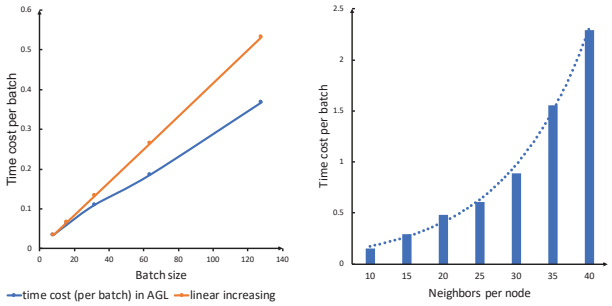

Figure 10: **Time cost per batch by varying batch size and neighbors**

From Table 4, we can observe that GraphInfer consistently outperforms the original inference module in both time cost and resource cost. GraphInfer takes about 1.2 hours to infer the predicted scores of 6.23 billion nodes with a 2-layer GAT model generating 8-dimensional embedding, which is just about $\frac{1}{4}$ of the time spent by the original inference module. Moreover, GraphInfer also saves 50% CPU cost and 76% memory cost, respectively. Compared with the original inference module based on *GraphFeature*, GraphInfer avoids repeated computing by employing the message passing scheme, which is the reason why it outperforms the original inference module.

6. CONCLUSION

In this paper, we present AGL, an integrated system designed for industrial-scale graph learning tasks. Our system design follows the message passing scheme underlying the computation of GNNs, where we simply merge values from in-edge neighbors and propagate merged values to out-edge neighbors. With this programming principle, we design to implement the construction of K-hop neighborhood, an information-complete subgraph for each node, and the inference in MapReduce. In addition, the K-hop neighborhood ensures the independency among nodes in the graph, thus makes us simply train the model with parameter servers. AGL maximally utilizes the calculation of each embedding at inference, while optimizes the training from model level to operator level. As a result, AGL successfully achieves a nearly linear speedup in training with 100 workers. AGL can finish the training of a 2-layer graph attention network on a graph with billions of nodes and hundreds of billions of edges in 14 hours, and complete the inference in only 1.2 hour. We have all these achievements based only on mature infrastructures such as parameter server and MapReduce.

7. REFERENCES

[1] B. Bollobás and O. Riordan. The diameter of a scale-free random graph. *Combinatorica*, 24(1):5–34, 2004.

[2] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

[3] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.

[4] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[5] M. Fey and J. E. Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[6] V. Garcia and J. Bruna. Few-shot learning with graph neural networks. *arXiv preprint arXiv:1711.04043*, 2017.

[7] T. Hamaguchi, H. Oiwa, M. Shimbo, and Y. Matsumoto. Knowledge transfer for out-of-knowledge-base entities: A graph neural network approach. *arXiv preprint arXiv:1706.05674*, 2017.

[8] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034, 2017.

[9] B. Hu, Z. Zhang, C. Shi, J. Zhou, X. Li, and Y. Qi. Cash-out user detection based on attributed heterogeneous information network with a hierarchical attention mechanism. In *Proceedings of the AAAI*, volume 33, pages 946–953, 2019.

[10] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.

[11] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[12] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich. Pytorch-biggraph: A large-scale graph embedding system. *arXiv preprint arXiv:1903.12287*, 2019.

[13] Z. Liu, C. Chen, L. Li, J. Zhou, X. Li, L. Song, and Y. Qi. Geniepath: Graph neural networks with adaptive receptive paths. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4424–4431, 2019.

[14] Z. Liu, C. Chen, X. Yang, J. Zhou, X. Li, and L. Song. Heterogeneous graph neural networks for malicious account detection. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 2077–2085. ACM, 2018.

[15] Z. Liu, D. Wang, Q. Yu, Z. Zhang, Y. Shen, J. Ma, W. Zhong, J. Gu, J. Zhou, S. Yang, et al. Graph representation learning for merchant incentive optimization in mobile payment marketing. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 2577–2584, 2019.

[16] K. Marino, R. Salakhutdinov, and A. Gupta. The more you know: Using knowledge graphs for image classification. In *2017 CVPR*, pages 20–28. IEEE, 2017.

[17] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[18] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

[19] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[20] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, et al. Deep graph library: Towards efficient and scalable deep learning on graphs. *arXiv preprint arXiv:1909.01315*, 2019.

[21] Z. Wang, Q. Lv, X. Lan, and Y. Zhang. Cross-lingual knowledge graph alignment via graph convolutional networks. In *EMNLP*, pages 349–357, 2018.

[22] J. C. Westland, T. Q. Phan, and T. Tan. Private information, credit risk and graph structure in p2p lending networks. *arXiv preprint arXiv:1802.10000*, 2018.

[23] S. Yan, Y. Xiong, and D. Lin. Spatial temporal graph convolutional networks for skeleton-based action recognition. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[24] H. Yang. Aligraph: A comprehensive graph neural network platform. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3165–3166. ACM, 2019.

[25] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD*, pages 974–983. ACM, 2018.

[26] D. Zhang, X. Song, Z. Liu, Z. Zhang, X. Huang, L. Wang, and J. Zhou. Dsslp: A distributed framework for semi-supervised link prediction. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 1557–1566. IEEE, 2019.

[27] M. Zhang and Y. Chen. Inductive graph pattern learning for recommender systems based on a graph neural network. *arXiv preprint arXiv:1904.12058*, 2019.

[28] Y. Zhang, Q. Liu, and L. Song. Sentence-state lstm for text representation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 317–327, 2018.

[29] J. Zhou, X. Li, P. Zhao, C. Chen, L. Li, X. Yang, Q. Cui, J. Yu, X. Chen, Y. Ding, et al. Kunpeng: Parameter server based distributed learning systems and its applications in alibaba and ant financial. In *Proceedings of the 23rd ACM SIGKDD*, pages 1693–1702, 2017.

[30] M. Zitnik and J. Leskovec. Predicting multicellular function through multi-layer tissue networks. *Bioinformatics*, 33(14):i190–i198, 2017.