CogDL: A Comprehensive Library for Graph Deep Learning

Yukuo Cen
Zhenyu Hou
Yan Wang
cyk20@mails.tsinghua.edu.cn
houzy21@mails.tsinghua.edu.cn
wang-y17@mails.tsinghua.edu.cn
Tsinghua University, China

Qibin Chen
Yizhen Luo
Zhongming Yu
cqb19@mails.tsinghua.edu.cn
luoyz18@mails.tsinghua.edu.cn
yzm18@mails.tsinghua.edu.cn
Tsinghua University, China

Hengrui Zhang
Xingcheng Yao
Aohan Zeng
ehengrui-18@mails.tsinghua.edu.cn
yx18@mails.tsinghua.edu.cn
zhah22@mails.tsinghua.edu.cn
Tsinghua University, China

Shiguang Guo
Tsinghua University, China
gsg18@mails.tsinghua.edu.cn

Yuxiao Dong
Tsinghua University, China
yuxiaod@tsinghua.edu.cn

Yang Yang
Zhejiang University, China
yangya@zju.edu.cn

Peng Zhang
Zhipu AI, China
zpjamper@gmail.com

Guohao Dai
Shanghai Jiao Tong University, China
daiguohao@situ.edu.cn

Yu Wang
Tsinghua University, China
yu-wang@tsinghua.edu.cn

Chang Zhou
Alibaba Group, China
ericzhou.zc@alibaba-inc.com

Hongxia Yang
Alibaba Group, China
yang.yhx@alibaba-inc.com

Jie Tang
Tsinghua University, China
jietang@tsinghua.edu.cn

ABSTRACT
Graph neural networks (GNNs) have attracted tremendous attention from the graph learning community in recent years. It has been widely adopted in various real-world applications from diverse domains, such as social networks and biological graphs. The research and applications of graph deep learning present new challenges, including the sparse nature of graph data, complicated training of GNNs, and non-standard evaluation of graph tasks. To tackle the issues, we present CogDL\textsuperscript{1}, a comprehensive library for graph deep learning that allows researchers and practitioners to conduct experiments, compare methods, and build applications with ease and efficiency. In CogDL, we propose a unified design for the training and evaluation of GNN models for various graph tasks, making it unique among existing graph learning libraries. By utilizing this unified trainer, CogDL can optimize the GNN training loop with several training techniques, such as mixed precision training. Moreover, we develop efficient sparse operators for CogDL, enabling it to become the most competitive graph library for efficiency. Another important CogDL feature is its focus on ease of use with the aim of facilitating open and reproducible research of graph learning. We leverage CogDL to report and maintain benchmark results on fundamental graph tasks, which can be reproduced and directly used by the community.

\textsuperscript{1}Jie Tang is the corresponding author.

\textsuperscript{2}The code is available at: \url{https://github.com/THUDM/cogdl}

CCS CONCEPTS
• Software and its engineering → Software libraries and repositories; • Mathematics of computing → Graph algorithms; • Computing methodologies → Learning latent representations.

KEYWORDS
graph deep learning, graph neural networks, graph library.

ACM Reference Format:

1 INTRODUCTION
Graph-structured data have been widely utilized in many real-world scenarios. Inspired by recent trends of representation learning on computer vision (CV) and natural language processing (NLP), graph neural networks (GNNs) [34, 71] are proposed to apply neural architectures to perform message passing over graph structures. For example, graph convolutional networks (GCNs) [48] simplify the spectral graph convolutions with first-order approximation and utilize a layer-wise propagation rule of graph convolution. GraphSAGE [37] adopts a sampling-based method to generate node embeddings for large graphs, which can be applied to the inductive setting. Recent effort in GNNs has been focused on theoretical understanding [98], generalization [26], self-supervised learning [43, 86], and capacity to handle web-scale applications [104]. To date, GNNs have achieved impressive performance on various graph machine learning tasks in diverse domains [42].
With the prosperity of GNNs, the evaluation and fair comparison of GNN models can still be a concerning issue. Although there are enormous research works on graph machine learning, we observe that different papers may use their evaluation settings for the same graph dataset, making results indistinguishable. For example, as a widely used dataset, Cora [73], some papers use the “standard” semi-supervised splits following Planetoid [102], while others adopt random splits or fully-supervised splits. Reported results of a GNN model on the same dataset may differ in various papers, making it challenging to compare the performance of different GNN methods [22, 24, 42, 74].

Moreover, the graph-structured data is fundamentally different from natural language or images, which can be easily formatted as Tensor and thus straightforwardly processed by GPUs. The non-Euclidean and sparse nature of graph data requires a better storage format for efficient computation, especially on GPUs. However, current deep learning frameworks do not well support the sparse computation of graph-structure data. For example, the PyTorch API of sparse tensors is limited and inefficient for graph operations. To bridge the gap, several open-source efforts have been made to develop dedicated libraries for the efficient development of graph representation learning research. For example, PyTorch Geometric (PyG) [27] is a graph learning library built upon PyTorch [62] to easily write and train GNNs for various applications. Deep Graph Library (DGL) [91] is an efficient and scalable package for deep learning on graphs, which provides several APIs allowing arbitrary message-passing computation over large-scale graphs. However, these popular graph libraries mainly focus on implementing basic GNN operators and do not take the whole training and evaluation of GNNs into consideration. The end-to-end performance of training GNNs could still be improved.

Present work: CogDL. In this paper, we introduce CogDL, a comprehensive library for graph deep learning that allows researchers, practitioners, and developers to train and evaluate GNN models under different graph tasks with ease and efficiency. The advantage of CogDL lies in its design of a unified GNN trainer, standard evaluation, and modular wrappers. Specifically, CogDL unifies the training and evaluation of graph neural network (GNN) models, making it different from other graph learning libraries. In addition, it is also equipped with decoupled modules that can be flexibly plugged for training GNNs, including ModelWrapper and DataWrapper. CogDL provides built-in GNN-specific wrappers, which could not be offered by other libraries.

The goal of CogDL is to accelerate research and applications of graph deep learning. Based on the design of the unified trainer and decoupled modules, CogDL users can instantiate a GNN trainer by trainer = Trainer(...), and then call trainer.run(...) to conduct experiments easily, which is not supported by existing graph libraries. We also provide a high-level API (i.e., experiment) on top of Trainer for one-line usage. Using the unified trainer not only saves users’ time in writing additional code but also gives users the opportunity to enable many built-in training techniques, such as mixed precision training. For example, users only need to set Trainer(fp16=True) to enable the feature of mixed precision training without any modification. Furthermore, CogDL provides many tasks with standard evaluation as possible for users (as shown in Table 1), and maintains experimental results on these fundamental graph tasks with recorded hyperparameter configurations. We summarize the characteristics of CogDL as follows:

- **Ease of use.** We provide simple APIs in CogDL such that users only need to write one line of code to train and evaluate any graph representation learning methods. CogDL develops a unified trainer with decoupled modules for users to train GNNs easily. Based on this unique design, CogDL can provide extensive features such as hyperparameter optimization, efficient training techniques, and experiment management.
- **End-to-end performance.** Efficiency is one of the most significant characteristics of CogDL. CogDL develops well-optimized sparse kernel operators to speed up the training of GNN models. For example, these efficient sparse operators enable CogDL to achieve about 2× speedups on the 2-layer GCN [48] and GAT [87] models compared with PyG and DGL.
- **Reproducibility.** Users can quickly conduct the experiments to reproduce the results based on CogDL, as illustrated in Listing 1. CogDL implements 70+ models and collects 70+ datasets for 10 fundamental graph tasks, as shown in Table 1, facilitating open, robust, and reproducible deep learning research on graphs.

Relevance to Web. GNNs have been widely used in web research and applications, such as social influence prediction [64, 68], social recommendation [25], and web advertising [103]. However, there still remain several challenges to applying powerful GNN models to real web applications. Practitioners need to not only choose an appropriate GNN but also train the GNNs on large-scale graphs. To mitigate the issues, CogDL provides unique advantages for web research and applications. First, the well-optimized sparse operators can speed up the process of new research and applications. Besides, the unified trainer with built-in wrappers can be directly utilized to conduct experiments with ease and efficiency. In addition, CogDL provides the benchmarking results of GNN methods for all kinds of graph tasks, which could help web-related users to choose appropriate methods. Finally, we have successfully applied CogDL to a real-world academic system—AMiner. More details about the application can be found in Appendix B.

### 2 THE COGDL FRAMEWORK

In this section, we present the unified design of the CogDL library, which is built on PyTorch [62], a well-known deep learning library. Based on PyTorch, CogDL provides efficient implementations of many models and our designed sparse operators. To date, it supports 10 important graph tasks, such as node classification. Table 1 lists all supported tasks, datasets, and models in the CogDL. All the models and datasets are compatible with the unified trainer for conducting experiments on different graph tasks.

**Listing 1: One-line code to conduct experiments.**

```
from cogdl import experiment
def experiment(dataset=["cora","citeseer"], model=["gcn","gat","gnnii"],
```
Table 1: All supported tasks, datasets, and models in CogDL. To date, CogDL integrates 10 important graph tasks with the standard evaluation of 70+ datasets. Besides, CogDL implements 70+ graph representation methods, which can be directly used by users. Note that some datasets and methods can be used in several tasks.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Datasets</th>
<th>Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Embedding</td>
<td>PPI, Wikipedia, Blogcatalog, DBLP, Flickr, YouTube</td>
<td>DeepWalk [63], LINE [80], node2vec [36], ProNE [110], NetMF [67], NetSMF [66], SDNE [90], Spectral [83], DNGR [12], GraRep [11], HOPE [61], AutoProNE [40]</td>
</tr>
<tr>
<td>Node Classification</td>
<td>Cora, Citeseer, Pubmed, Chameleon, Cornell, Film, Squreel, Texas, Wisconsin, PPI, Flickr, Reddit, Amazon, Yelp, GitHub, Elliptic, Yelpchi, Alpha, Clothing, Electronics, Wiki, Weibo, BGP, SSN, ogbn-arxiv, ogbn-products, ogbn-proteins, ogbn-papers100M</td>
<td>Chebyshev [20], GCN [48], GAT [87], GraphSAGE [37], APPNP [49], DGI [86], GCNII [17], MVGRL [39], GRAND [26], DropEdge [69], Graph-Unet [31], GDC [50], PPRGo [8], GraphSAINT [109], GraphMix [88], SIGN [28], MixHop [3], GRACE [117], DeeperGCN [53], DseneGCN [56], SGC [96], M3S [77], C&amp;S [46], SAGN [75], RevGNN [52], Cluster-GCN [18], InfoMotif [70], GraphMAE [41]</td>
</tr>
<tr>
<td>Graph Classification</td>
<td>MUTAG, PTC, NC1, PROTEINS, IMDB-B, IMDB-M, COLLAB, REDDIT-B, REDDIT-M5k, REDDIT-M12k, NC109, ENZYMES, ogbg-molhiv, ogbg-molpcba, ogbg-ppa, ogbg-code</td>
<td>DGK [100], graph2vec [58], GIN [98], PATCHY_SAN [60], DiffPool [105], InfoGraph [76], DGCNN [94], SortPool [111]</td>
</tr>
<tr>
<td>Heterogeneous Node Classification</td>
<td>DBLP, ACM, IMDB</td>
<td>Simple-HGN [55], GTN [107], HAN [92], PTE [79], Metapath2vec [21], Hin2vec [29]</td>
</tr>
<tr>
<td>Link Prediction</td>
<td>Cora, Citeseer, Pubmed, ogbl-ppa, ogbl-ddi, ogbl-collab, ogbl-citation2</td>
<td>GAE [47], VGAE [47], network embedding methods</td>
</tr>
<tr>
<td>Knowledge Graph Completion</td>
<td>FB13, FB15k, FB15k-237, WN18, WN18RR</td>
<td>TransE [9], DistMult [101], ComplEx [84], RotatE [78], RGCN [72], CompGCN [85]</td>
</tr>
<tr>
<td>Graph Clustering</td>
<td>Cora, Citeseer, Pubmed</td>
<td>GAE [47], VGAE [47], AGC [112], DAEGC [89]</td>
</tr>
<tr>
<td>Graph Attacks and Defenses</td>
<td>grb-cora, grb-citeseer, grb-flickr, grb-reddit, grb-aminer</td>
<td>GNN-SVD [23], GNNGuard [113], RoubustGCN [115], RND [119], TDGLA [118], FGSN [33], PGLG [57], SPEIT [114], DICE [95], FGA [15], FLIP [7], NEA [7], STACK [97]</td>
</tr>
<tr>
<td>Multiplex Link Prediction</td>
<td>Amazon, YouTube, Twitter</td>
<td>GATNE [13], network embedding methods</td>
</tr>
<tr>
<td>Top-k Similarity Search</td>
<td>KDD-ICDM, SIGIR-CIKM, SIGMOD-ICDE</td>
<td>GCC [65]</td>
</tr>
</tbody>
</table>

2.1 Unified Trainer

CogDL provides a unified trainer for GNN models, which takes over the entire loop of the training process. The unified trainer, which contains much engineering code, is implemented flexibly to cover arbitrary GNN training settings. We design four decoupled modules for the GNN training, including Model, ModelWrapper, Dataset, DataWrapper. The ModelWrapper is for the training and testing steps, while the DataWrapper is designed to construct data loaders used by ModelWrapper. The main contributions of most GNN papers mainly lie in three modules except Dataset. For example, the GCN paper trains the GCN model under the (semi-)supervised and full-graph setting, while the DGI paper trains the GCN model by maximizing local-global mutual information. The training method of the DGI is considered as a model wrapper named dgi_mw, which could be used for other scenarios. Our unified design also supports the precomputation-based GNN models, such as SIGN [28] and SAGN [75].

Based on the design of the unified trainer and decoupled modules, we could do arbitrary combinations of models, model wrappers, and data wrappers. For example, if we want to apply the DGI, a representative graph self-supervised learning method, to large-scale datasets, all we need is to substitute the full-graph data wrapper with the neighbor-sampling or clustering data wrappers without additional modifications. If we design a new GNN architecture for a specific task, we only need to write essential PyTorch-style code for the model. The rest could be automatically handled by CogDL by specifying the model wrapper and the data wrapper. We could quickly conduct experiments for the model using the trainer via...
trainer = Trainer(epochs,...) and trainer.run(...). Moreover, based on the unified trainer, CogDL provides native support for many useful features, including hyperparameter optimization, efficient training techniques, and experiment management without any modification to the model implementation.

We introduce efficient techniques of GNN training, which credit to the unified trainer of CogDL. These features could be enabled through fp16=True or actn=True in Trainer API.

**Mixed Precision Training.** CogDL supports mixed precision training, which is a popular technique to relieve the GPU memory and speed up the training process. PyTorch provides a convenient method for mixed precision in torch.cuda.amp, which integrates NVIDIA apex. For example, we conduct experiments to test the performance of mixed precision training.

**Activation Compression Training.** With the emergence of deep GNNs, the activation footprints occupy more and more GPU memory. We need to store the activation output of each layer to compute the gradient in the backward pass, which costs much GPU memory in the training step. We extend the activation compression training technique [16, 54] to GNNs. The main advantage of activation compression training is that the GPU memory of training could dramatically decrease.

Based on the provided trainer, CogDL also supports the following features for conducting experiments with ease.

**Hyperparameter Optimization (HPO).** Hyperparameter optimization is an important feature for GNN libraries since current GNN models utilize more hyperparameters than before. We integrate a popular library, Optuna [4], into CogDL to enable HPO. Optuna is a fast AutoML library based on Bayesian optimization. CogDL implements hyperparameter search and even model search for graph learning. The key of the HPO is to define a search space. The search space will be automatically utilized by CogDL to start searching and output the best results. The usage of HPO can be found in Appendix A.2.

**Experiment Management.** Experiment management is crucial for training deep learning models, which could be utilized by researchers and developers for debugging and designing new blocks. CogDL integrates TensorBoard and WandB [6] for experiment management. TensorBoard is a common visualization toolkit for tracking metrics like loss and accuracy. WandB is a central dashboard to keep track of the experimental status. The experiment management could be enabled easily by assigning the logger argument.

### 2.2 Efficient Sparse Operators

We introduce the well-optimized sparse operators of CogDL, including SpMM-like operators, multi-head SpMM, and edge-wise softmax, which are developed and implemented in CogDL for GNN models. We first introduce the abstraction of sparse operators in CogDL and its corresponding optimization. The General Sparse Matrix-Matrix multiplication (GSpMM) operator is widely used in most GNNs. The reason is that many GNNs apply a general aggregation operation for each node from its neighbors:

\[
h_u^{(l+1)} = \phi \left( \{ \psi \left( h_v^{(l)}, m_e \right) \mid (u, e, v) \in E \} \right)
\]

where \((u, e, v)\) denotes an edge \(e\) from node \(u\) to node \(v\) with a message \(m_e\), and \(h_v^{(l)}\) is the hidden representation of node \(v\) of layer \(l\). When the reduce operation \(\phi\) is a summation, and the compute operation \(\psi\) is a multiplication, such an operation can be described as an SpMM operator \(H^{(l+1)} = AH^{(l)}\), where the sparse matrix \(A\) represents the adjacency matrix of the input graph \(G\), and each row of \(H^{(l)}\) represents a representation vector of nodes (e.g., \(h_u^{(l)}\) and \(h_v^{(l)}\)).

We observe that in existing graph neural network libraries such as PyG [27], or DGL [91], their backend support is not tailored to GNN. In PyG, the message-passing flow is built by torch-scatter, which is borrowed from the design principles of traditional graph processing. In DGL, its GSpMM function leverage APIs from cuSPARSE [59], which is a vendor library that mainly targets for scientific computing of sparse matrices. Another observation is that many previous GPU-based SpMM works fail to be aware of the adaptability of workload balance in graphs. For example, GE-SpMM [44] and Sputnik [30] are row-split SpMM methods that could not efficiently support graphs that follow a power-law distribution. CogDL is fully aware of the workload-load balance problem compared to PyG/DGL. We observe that the algorithm of neighbor group [45, 93] could better slice the large nodes in the graph and thus has better efficiency when facing power-law distribution graphs. However, some graphs may naturally conform to node load balancing, such as the k-sampled graph [37], so it is better to use the row-split backend directly.

To tackle the following problems, we propose a new balance abstraction that not only exploits the data of the graph itself but also effectively accelerates our backend. As shown in Listing 2, our workload balance handler would be generated by a data-aware balancer, which could indicate whether to use a row-split or neighbor group algorithm. Empirically, when the degrees of the majority (90%) of the nodes are close to the average degree, then we would choose row-split as our backend since it is suitable for graphs that are naturally node balanced. Note that our workload balancer is a preprocessing method since it would generate the auxiliary arrays that are needed by the GSpMM backend. The Listing 2 demonstrates that our GSpMM’s inputs include a work balance handler, graph data, and input features. Users could choose the reduce and compute operator, as illustrated in Equation 1.

**Listing 2: workload balance abstraction in CogDL backend**

```python
workload_balance = balancer(graph) # preprocessing
out_feat = GSpMM(workload_balance, graph, in_feat, reduce_op, compute_op)
```

In our GSpMM design, we implement the warp-based thread mapping algorithm, which would bind different neighbor groups or split rows to different warps. This allows us to assign multiple adjacent rows or neighbor groups into a single CUDA block. Since the same processor shares an L1 cache, adjacent rows or neighbor groups can reduce the L1 loaded data to improve locality. Multi-head SpMM is used to compute the aggregation procedure of GNN models with multi-head attention mechanisms (such as GAT), which means each edge has a different edge weight for each attention head; therefore, multiple SpMM operators are needed. PyG and DGL use multiple SpMM with broadcasting to implement it. In
Table 2: Performance of mixed precision training.

<table>
<thead>
<tr>
<th>Memory</th>
<th>Accuracy</th>
<th>Training Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2080 Ti</td>
</tr>
<tr>
<td>w/o fp16</td>
<td></td>
<td>2.20 it/s</td>
</tr>
<tr>
<td>with fp16</td>
<td></td>
<td>3.17 it/s</td>
</tr>
</tbody>
</table>

Table 3: Accuracy (%) and activation memory (MB) of 2-layer GCNs on three datasets.

<table>
<thead>
<tr>
<th></th>
<th>origin (32bit)</th>
<th>act (4bit)</th>
<th>act (3bit)</th>
<th>act (2bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flickr</td>
<td>51.17 (288)</td>
<td>51.08 (37)</td>
<td>51.14 (26)</td>
<td>51.20 (18)</td>
</tr>
<tr>
<td>Reddit</td>
<td>95.33 (1532)</td>
<td>95.32 (194)</td>
<td>95.31 (158)</td>
<td>95.34 (112)</td>
</tr>
<tr>
<td>Yelp</td>
<td>39.86 (4963)</td>
<td>40.06 (773)</td>
<td>40.21 (665)</td>
<td>39.89 (551)</td>
</tr>
</tbody>
</table>

our multi-head SpMM kernel, we apply the memory coalescing technique by allocating consecutive warps along the dimension of the head. As different heads share the same location of non-zero elements, this data will be cached by the streaming multiprocessors and reused among different heads, which also saves bandwidth.

When considering a GAT model, we also need a sampled dense-dense matrix multiplication (SDDMM) operator $T = A \odot (PQ^T)$, where $T$ and $A$ are the sparse matrices, $P$ and $Q$ are two dense matrices, $\odot$ is element-wise matrix multiplication. The SDDMM operator is used for back-propagating the gradients to the sparse adjacency matrix since the adjacency matrix of the GAT model is computed by the attention mechanism. CogDL further speeds up multi-head graph attention by optimizing Edge-wise Softmax, which is defined by $a_{uv}^* = \exp(a_{uv})/(\sum_{v \in N(u)} \exp(a_{uv}))$, where $a_{uv}$ is the attention score between nodes $u$ and $v$.

In our design of the Edge-wise Softmax kernel, we use the method of warp level intrinsic provided by NVIDIA, a typical method that could accelerate the scan and reduce operation. To prevent the spillover of the exponent, we first apply the scan to find the max value of weight and then subtract this maximum from each value. After that, we compute each value by applying the exponent function and reduce all the values within the warp to acquire the sum of all the exponent values. As far as we know, CogDL is the first framework that provides a GPU-based fused Edge-wise Softmax kernel. Therefore, we do not provide the results of the experiment with PyG and DGL.

3 EVALUATION

In this section, we conduct extensive experiments to evaluate the framework design and characteristics of CogDL.

3.1 Training Techniques

For mixed precision training, we run a 2-layer GCN model with 2048 hidden units on the Reddit dataset. From Table 2, the mixed precision training brings 27% memory savings and $1.44x \sim 2.02x$ speedup on NVIDIA 2080 Ti and 3090 GPUs with almost no performance drop. For activation compression training, we conduct experiments of the 2-layer GCN model with 256 hidden units on three datasets for the activation compression training in the CogDL, as shown in Table 3. The performance demonstrates that training GNNs with actnn brings $6.4x \sim 16x$ memory savings compared with 32-bit training. The accuracy with actnn (2.3,4,bit) is almost the same with 32-bit training.

3.2 Sparse Operators

We conduct experiments of GSpMM and multi-head SpMM on Reddit and Yelp datasets compared to DGL, as shown in Figure 1. For GSpMM, CogDL achieves $1.17x \sim 3.47x$ speedup with different reduce functions on Reddit and $1.27x \sim 4.01x$ speedup with different reduce functions on Yelp. For multi-head SpMM, CogDL achieves $1.22x \sim 2.03x$ and $1.31x \sim 2.05x$ speedup on Reddit and Yelp datasets, respectively.

3.3 End-to-End Performance

We compare the end-to-end inference time of GCN and GAT models on several datasets with other popular GNN frameworks: CogDL v0.5.2, PyTorch-Geometric (PyG) v2.0.2, and Deep Graph Library (DGL) v0.7.2 with PyTorch backend. The statistics of datasets could be found in Table 9. We conduct experiments using Python 3.7.10 and PyTorch v1.8.0 on servers with Nvidia GeForce RTX 2080 Ti (11GB GPU Memory) and 3090 (24GB GPU Memory). From Table 4, CogDL achieves at most $2x$ speedup on the 2-layer GCN model compared with PyG and DGL. For the 2-layer GAT model, CogDL achieves $1.32x \sim 2.36x$ speedup compared with PyG and DGL. Furthermore, OOM occurs when running the PyG’s GAT model on Reddit and Yelp datasets, even using 3090 (24G). The GAT model implemented by DGL also cannot run the Yelp dataset using 2080Ti (11G). The results demonstrate significant advantages of CogDL in inference time and memory savings, compared to state-of-the-art GNN frameworks.
4 BENCHMARKING

In this section, with CogDL, we introduce several downstream tasks to evaluate implemented methods. The statistics of the datasets are shown in Table 9 and Table 10 in the Appendix. Based on the experiment API, we build a reliable leaderboard for each task, which maintain state-of-the-art results.

4.1 Network Embedding

Setup. Unsupervised node classification task aims to learn a mapping function $f : V \rightarrow \mathbb{R}^d$ that projects each node to a $d$-dimensional space ($d \ll |V|$) in an unsupervised manner. Structural properties of the network should be captured by the mapping function. We collect the most popular datasets used in the unsupervised node classification task, including BlogCatalog [108], Wikipedia [67], PPI [10], DBLP [81], Flickr [82]. We implement and compare the following methods for the unsupervised node classification task, including SpectralClustering [83], DeepWalk [63], LINE [80], node2vec [36], GraRep [11], HOPE [61], NetMF [67], ProNE [110], NetSMF [66]. Skip-gram network embedding considers the vertex paths traversed by random walks over the network as the sentences and leverages Skip-gram for learning latent vertex representation. For matrix factorization based methods, they first compute a proximity matrix and perform matrix factorization to obtain the embedding. Actually, NetMF [67] has shown the aforementioned Skip-gram models with negative sampling can be unified into the matrix factorization framework with closed forms. We run all network embedding methods on five real-world datasets and report the Micro-F1 results (%) using logistic regression with L2 normalization.

Results and Analysis. Table 5 shows the results and we find some interesting observations. DeepWalk and node2vec consider vertex paths traversed by random walk to reach high-order neighbors. NetMF and NetSMF factorize diffusion matrix $\sum_{i=0}^{K} a_i \hat{A}^i$ rather than adjacency matrix $A$. ProNE and LINE are essentially 1-order methods, but ProNE further propagates the embeddings to enlarge the receptive field. Incorporating global information can improve performance but may hurt efficiency. The propagation in ProNE, which is similar to graph convolution, shows that incorporating global information as a post-operation is effective. The results demonstrate that matrix factorization (MF) methods like NetMF and ProNE are very powerful and full of vitality as they outperform Skip-gram based methods in almost all datasets. ProNE and NetSMF are also of high efficiency and scalability and able to embed super-large graphs in feasible time in one single machine.

4.2 Node Classification

Setup. This task is for node classification with GNNs in semi-supervised and self-supervised settings. Different from the previous part, nodes in these graphs, like Cora and Reddit, have node features and are fed into GNNs with prediction or representation as output. Cross-entropy loss and contrastive loss are set for semi-supervised and self-supervised settings, respectively. The datasets consist of two parts, including both semi-supervised and fully-supervised settings. Semi-supervised datasets include three citation networks, Citeseer, Cora, and Pubmed [73]. Fully-supervised datasets include social networks (Reddit, Yelp, and Flickr), bioinformatics (PPI) from GraphSAINT [109], and ogbn-arxiv [42]. We implement all the following models, including ChebyNet [20], GCN [48], GAT [87], GraphSAGE [37], APPNP [49], DGI [86], GCNI [17], MVGRL [39], GRAND [26], DropEdge [69], Graph-Unet [31], GDC [50], PPRGo [8], GraphSAINT [109]. For evaluation, we use prediction accuracy for multi-class and micro-F1 for multi-label datasets.

Results and Analysis. Table 6 and Table 7 summarize the evaluation results of all compared models in semi-supervised and fully-supervised datasets, respectively, under the setting of node classification. We observe that incorporating high-order information plays an important role in improving the performance of models, especially in citation datasets (Cora, CiteSeer, and PubMed) that are of relatively small scale. Most high-order models, such as GRAND, APPNP, and GDC, aim to use graph diffusion matrix $\hat{A} = \sum_{i=0}^{K} a_i \hat{A}^i$ to collect information of distant neighbors. GCNI uses the residual connection with identity mapping to resolve over-smoothing in GNNs and achieves remarkable results. GRAND [26] leverages unlabeled data in semi-supervised settings and obtains state-of-the-art results. As for self-supervised methods, DGI and MVGRL maximize local and global mutual information. MVGRL performs better by replacing the adjacency matrix with graph diffusion matrix but is less scalable. On the contrary, GRACE, which optimizes InfoNCE, does not perform well on citation datasets with the public split.
Table 5: Micro-F1 score (%) of network embedding methods reproduced by CogDL. 50% of nodes are labeled for training in PPI, Blogcatalog, and Wikipedia, 5% in DBLP and Flickr. These datasets correspond to different downstream scenarios.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Method</th>
<th>PPI (50%)</th>
<th>Wikipedia (50%)</th>
<th>Blogcatalog (50%)</th>
<th>DBLP (5%)</th>
<th>Flickr (5%)</th>
<th>Reproducible</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NetMF [67]</td>
<td>23.73 ± 0.22</td>
<td>57.42 ± 0.56</td>
<td>42.47 ± 0.35</td>
<td>56.72 ± 0.14</td>
<td>36.27 ± 0.17</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>ProNE [110]</td>
<td>24.60 ± 0.39</td>
<td>56.06 ± 0.48</td>
<td>41.16 ± 0.26</td>
<td>56.85 ± 0.28</td>
<td>36.36 ± 0.11</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>NetSMF [66]</td>
<td>23.88 ± 0.35</td>
<td>53.81 ± 0.58</td>
<td>40.62 ± 0.35</td>
<td>59.76 ± 0.41</td>
<td>35.49 ± 0.07</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>Node2vec [36]</td>
<td>20.67 ± 0.54</td>
<td>54.59 ± 0.51</td>
<td>40.16 ± 0.29</td>
<td>57.36 ± 0.39</td>
<td>36.13 ± 0.13</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>LINE [80]</td>
<td>21.82 ± 0.56</td>
<td>52.46 ± 0.26</td>
<td>38.06 ± 0.39</td>
<td>49.78 ± 0.37</td>
<td>31.61 ± 0.09</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>DeepWalk [63]</td>
<td>20.74 ± 0.40</td>
<td>49.53 ± 0.54</td>
<td>40.48 ± 0.47</td>
<td>57.54 ± 0.32</td>
<td>36.09 ± 0.10</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>SpectralClustering [83]</td>
<td>22.48 ± 0.30</td>
<td>49.35 ± 0.34</td>
<td>41.41 ± 0.34</td>
<td>43.68 ± 0.58</td>
<td>33.09 ± 0.07</td>
<td>Yes</td>
</tr>
<tr>
<td>8</td>
<td>Hope [61]</td>
<td>21.43 ± 0.32</td>
<td>54.04 ± 0.47</td>
<td>33.99 ± 0.35</td>
<td>56.15 ± 0.22</td>
<td>28.97 ± 0.19</td>
<td>Yes</td>
</tr>
<tr>
<td>9</td>
<td>GraRep [11]</td>
<td>20.60 ± 0.34</td>
<td>54.37 ± 0.40</td>
<td>33.48 ± 0.30</td>
<td>52.76 ± 0.42</td>
<td>31.83 ± 0.12</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 6: Accuracy (%) reproduced by CogDL for semi-supervised and self-supervised node classification on Citation datasets. ↓ and ↑ mean our results are lower or higher than the result in original papers.

<table>
<thead>
<tr>
<th>Method</th>
<th>Cora</th>
<th>Citeseer</th>
<th>Pubmed</th>
<th>Reproducible</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRAND [26]</td>
<td>84.8</td>
<td>75.1</td>
<td>82.4</td>
<td>Yes</td>
</tr>
<tr>
<td>GCNII [17]</td>
<td>85.1</td>
<td>71.3</td>
<td>80.2</td>
<td>Yes</td>
</tr>
<tr>
<td>MGRL [39]</td>
<td>83.6 ↓</td>
<td>73.0</td>
<td>80.1</td>
<td>Partial</td>
</tr>
<tr>
<td>APPNP [49]</td>
<td>84.3 ↑</td>
<td>72.0</td>
<td>80.0</td>
<td>Yes</td>
</tr>
<tr>
<td>Graph-Uncet [31]</td>
<td>83.3 ↓</td>
<td>71.2 ↓</td>
<td>79.0</td>
<td>Partial</td>
</tr>
<tr>
<td>GDC [50]</td>
<td>82.5</td>
<td>72.1</td>
<td>79.8</td>
<td>Yes</td>
</tr>
<tr>
<td>GAT [87]</td>
<td>82.9</td>
<td>71.0</td>
<td>78.9</td>
<td>Yes</td>
</tr>
<tr>
<td>DropEdge [69]</td>
<td>82.1</td>
<td>72.1</td>
<td>79.7</td>
<td>Yes</td>
</tr>
<tr>
<td>GCN [48]</td>
<td>81.5</td>
<td>71.4 ↑</td>
<td>79.5</td>
<td>Yes</td>
</tr>
<tr>
<td>DGI [86]</td>
<td>82.0</td>
<td>71.2</td>
<td>76.5</td>
<td>Yes</td>
</tr>
<tr>
<td>JK-net [99]</td>
<td>81.8</td>
<td>69.5</td>
<td>77.7</td>
<td>Yes</td>
</tr>
<tr>
<td>Chebyshev [20]</td>
<td>79.0</td>
<td>69.8</td>
<td>68.6</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 7: Results of node classification on fully-supervised datasets, including full-batch and sampling-based methods. Flickr, Reddit, and ogbn-arxiv use accuracy metric, whereas PPI uses Micro-F1 metric.

<table>
<thead>
<tr>
<th>Method</th>
<th>Flickr</th>
<th>PPI</th>
<th>Reddit</th>
<th>arxiv</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCNII</td>
<td>52.6 ± 0.1</td>
<td>96.5 ± 0.2</td>
<td>96.4 ± 0.0</td>
<td>72.5 ± 0.1</td>
</tr>
<tr>
<td>GraphSAINT</td>
<td>52.0 ± 0.1</td>
<td>99.3 ± 0.1</td>
<td>96.1 ± 0.0</td>
<td>71.5 ± 0.2</td>
</tr>
<tr>
<td>GAT</td>
<td>51.9 ± 0.3</td>
<td>97.3 ± 0.2</td>
<td>95.9 ± 0.1</td>
<td>72.3 ± 0.1</td>
</tr>
<tr>
<td>Cluster-SAGE</td>
<td>50.9 ± 0.1</td>
<td>97.9 ± 0.1</td>
<td>95.8 ± 0.1</td>
<td>69.8 ± 0.1</td>
</tr>
<tr>
<td>GCN</td>
<td>54.2 ± 0.1</td>
<td>75.7 ± 0.1</td>
<td>95.1 ± 0.0</td>
<td>71.7 ± 0.3</td>
</tr>
<tr>
<td>APPNP</td>
<td>52.3 ± 0.1</td>
<td>62.3 ± 0.1</td>
<td>96.2 ± 0.1</td>
<td>70.7 ± 0.1</td>
</tr>
<tr>
<td>PPRGo</td>
<td>51.1 ± 0.2</td>
<td>48.6 ± 0.1</td>
<td>94.5 ± 0.1</td>
<td>69.2 ± 0.1</td>
</tr>
<tr>
<td>SGC</td>
<td>50.2 ± 0.1</td>
<td>47.2 ± 0.1</td>
<td>94.5 ± 0.1</td>
<td>67.1 ± 0.0</td>
</tr>
</tbody>
</table>

4.3 Graph Classification

Setup. Graph classification assigns a label to each graph and aims to map graphs into vector spaces. Graph kernels are historically dominant and employ a kernel function to measure the similarity between pairs of graphs to map graphs into vector spaces with deterministic mapping functions. But they suffer from computational bottlenecks. Recently, graph neural networks attract much attention and indeed show promising results in this task. In the context of graph classification, GNNs often employ the readout operation to obtain a compact representation of the graph and apply classification based on the readout representation. We collect eight popular benchmarks used in graph classification tasks, including Bioinformatics datasets (PROTEINS, MUTAG, PTC, and NCI1), and Social networks (IMDB-B, IMDB-M, COLLAB, REDDIT-B). We implement the following graph classification models and compare their results: GIN [98], DiffPool [105], SAGPool [51], SortPool [111], DGCNN [94] and Infograph [76] are based on GNNs. PATCHY_SAN [60] is inspired by convolutional neural networks. Deep graph kernels (DGK) [100] and graph2vec [58] use graph kernels. As for evaluation, for supervised methods we adopt 10-fold cross-validation with 90%/10% split and repeat 10 times; for unsupervised methods, we perform the 10-fold cross-validation with LIB-SVM [14]. Then we report the accuracy for classification performance of all methods on these datasets.

Results and Analysis. Table 8 reports the results of the aforementioned models on the task, including both unsupervised and supervised graph classification. The development of GNNs for graph classification is mainly in two aspects. One line (like GIN) aims to design more powerful convolution operations to improve the expressiveness. Another line is to develop effective pooling methods to generate the graph representation. Neural network based methods show promising results in bioinformatics datasets (MUTAG, PTC, PROTEINS, and NCI1), where nodes are with given features. But in social networks (IMDB-B, IMDB-M, COLLAB, REDDIT-B) lacking node features, methods based on graph kernels achieve good performance and even surpass neural networks. Graph kernels are more capable of capturing structural information to discriminate non-isomorphic graphs, while GNNs are better encoders with features. Global pooling, which is used in GIN and SortPool, collects node features and applies a readout function. Hierarchical pooling, such as DiffPool and SAGPool, is proposed to capture structural information in different graph levels, including nodes and subgraphs. The experimental results indicate that, though hierarchical pooling seems more complex and intuitively would capture more information, it does not show significant advantages over global pooling.
Table 8: Accuracy Results (%) of both unsupervised and supervised graph classification. ‡ and † mean our results are lower or higher than the results in original papers. ‡ means the experiment is not finished in 24 hours for one seed.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>MUTAG</th>
<th>PTC</th>
<th>NCI1</th>
<th>PROTEINS</th>
<th>IMDB-B</th>
<th>IMDB-M</th>
<th>COLLAB</th>
<th>REDDIT-B</th>
<th>Reproducible</th>
</tr>
</thead>
<tbody>
<tr>
<td>GIN [98]</td>
<td>92.06</td>
<td>67.82</td>
<td>81.66</td>
<td>75.19</td>
<td>76.10</td>
<td>51.80</td>
<td>79.52</td>
<td>83.10</td>
<td>Yes</td>
</tr>
<tr>
<td>InfoGraph [76]</td>
<td>88.95</td>
<td>60.74</td>
<td>76.64</td>
<td>73.93</td>
<td>74.50</td>
<td>51.33</td>
<td>79.40</td>
<td>76.55</td>
<td>Yes</td>
</tr>
<tr>
<td>graph2vec [38]</td>
<td>83.68</td>
<td>54.76</td>
<td>↓</td>
<td>71.85</td>
<td>73.30</td>
<td>73.90</td>
<td>52.27</td>
<td>85.58</td>
<td>†</td>
</tr>
<tr>
<td>SortPool [111]</td>
<td>87.25</td>
<td>62.04</td>
<td>73.99</td>
<td>↑</td>
<td>74.48</td>
<td>75.40</td>
<td>50.47</td>
<td>80.07</td>
<td>†</td>
</tr>
<tr>
<td>DiffPool [105]</td>
<td>85.18</td>
<td>58.00</td>
<td>69.09</td>
<td>75.30</td>
<td>72.50</td>
<td>50.50</td>
<td>79.27</td>
<td>81.20</td>
<td>Yes</td>
</tr>
<tr>
<td>PATCHY_SAN [60]</td>
<td>86.12</td>
<td>61.60</td>
<td>69.82</td>
<td>75.38</td>
<td>76.00</td>
<td>46.40</td>
<td>74.34</td>
<td>60.61</td>
<td>Yes</td>
</tr>
<tr>
<td>DGCNN [94]</td>
<td>83.33</td>
<td>56.72</td>
<td>65.96</td>
<td>66.75</td>
<td>71.60</td>
<td>49.20</td>
<td>77.45</td>
<td>86.20</td>
<td>Yes</td>
</tr>
<tr>
<td>SAGPool [51]</td>
<td>71.73</td>
<td>↓</td>
<td>59.92</td>
<td>72.87</td>
<td>74.03</td>
<td>74.80</td>
<td>51.33</td>
<td>___</td>
<td>†</td>
</tr>
<tr>
<td>DGK [100]</td>
<td>85.58</td>
<td>57.28</td>
<td>___</td>
<td>72.59</td>
<td>55.00</td>
<td>40.40</td>
<td>___</td>
<td>___</td>
<td>Partial</td>
</tr>
</tbody>
</table>

4.4 Other Graph Tasks

As shown in Table 1, CogDL also provides other important tasks in graph learning. In this section, we briefly introduce the setting and evaluation of some tasks, including heterogeneous node classification, link prediction, multiplex link prediction, etc.

Heterogeneous Node Classification. This task is built for heterogeneous GNNs conducted on heterogeneous graphs such as academic graphs. For heterogeneous node classification, we use macro-F1 to evaluate the performance of all heterogeneous models under the setting of Graph Transformer Networks (GTN) [107]. The heterogeneous datasets include DBLP, ACM, and IMDB. We implement these models, including GTN [107], HAN [92], PTE [79], metapath2vec [21], hin2vec [29].

Link Prediction. Many real applications, such as citation link prediction, can be viewed as a link prediction task [47]. We formulate the link prediction task as: given a part of edges in the graph, after learning representations for each vertex v ∈ V, the method needs to accurately predict the edges that belong to the rest of the edges rather than random edges. In practice, we remove 15 percent of original edges for each dataset and adopt ROC AUC [38] as the evaluation metric.

Multiplex Link Prediction. GATNE [13] formalizes the representation learning problem of multiplex networks and proposes a unified framework to solve the problem in transductive and inductive settings. We follow the setting of GATNE [13] to build the multiplex heterogeneous link prediction task. In CogDL, for those methods that only deal with homogeneous networks, we separate the graphs based on the edge types and train the method on each separate graph. We also adopt ROC AUC as the evaluation metric.

Knowledge Graph Completion. This task aims to predict missing links in a knowledge graph through knowledge graph embedding. In CogDL, we implement two families of knowledge graph embedding algorithms: triplet-based knowledge graph embedding and knowledge-based GNNs. The former includes TransE [9], DistMult [101], ComplEx [84], and RotatE [78]; the latter includes RGCN [72] and CompGCN [85]. We evaluate all implemented algorithms with standard benchmarks including FB15k-237, WN18 and WN18RR with Mean Reciprocal Rank (MRR) metric.

5 DISCUSSIONS

With the success of graph neural networks, there are dozens of graph libraries developed in the past few years, including PGL [1], Graph Nets [5], Jraph [32], Graph-Learn [116], Spektral [35], StellarGraph [19], Deep Graph Library (DGL) [91], and PyTorch Geometric (PyG) [27]. Among them, PyG [27] and DGL [91] are two of the most well-known libraries with efficient operators and easy-to-use APIs. PyG provides CUDA kernels for sparse operators with high data throughput. PyG 2.0 further utilizes GraphGym [106] to achieve new features like reproducible configuration and hyperparameter optimization. DGL not only provides flexible and efficient message-passing APIs but also allows users to easily port and leverage the existing components across multiple deep learning frameworks (e.g., PyTorch [62], TensorFlow [2]). However, CogDL not only supports flexible implementations of GNN models but also provides a unified trainer for the end-to-end training of GNN models with efficient techniques.

6 CONCLUSIONS

This paper introduces CogDL, a comprehensive library for graph deep learning that focuses on the research and applications of graph representation learning. CogDL provides simple APIs such that users can train and evaluate graph learning methods with ease and efficiency. Besides, CogDL implements many state-of-the-art models with wrappers for various graph tasks. Finally, CogDL collects and maintains leaderboards with reproducible configurations on widely-used public datasets. In the future, we will consistently improve the efficiency of CogDL and support more features to satisfy the latest need of both scientific and industrial applications. For example, the dynamic scenario of graph learning, where nodes and edges will continuously be added and removed, needs to be considered. Besides, graph self-supervised learning and pre-training have become one of the most exciting research directions in recent years, which presents a new challenge for graph libraries.

ACKNOWLEDGMENTS

This research was supported by the Natural Science Foundation of China (NSFC) 61825602 and 62276148 and the Tsinghua-Siemens Joint Research Center for Industrial Intelligence and Internet of Things (JCIIOT).


[81] Reza Zafarani and Huan Liu. 2009. Social computing data repository at ASU.


A COGDL PACKAGE

In this section, we introduce the details of the COGDL library.

A.1 Basic Components

The basic elements in COGDL include Graph, Dataset, and Model.

Graph. The Graph is the basic data structure of COGDL to store graph data with abundant graph operations. Graph supports both convenient graph modification, such as removing/adding edges, and efficient computing, such as SpMM. We also provide common graph manipulations, including adding self-loops, graph normalization, sampling neighbors, obtaining induced subgraphs, etc.

Dataset. The dataset component reads in data and processes it to produce tensors of appropriate types. Each dataset specifies its loss function and evaluator through function get_loss_fn and get_evaluator for training and evaluation. Our package provides many real-world datasets and easy-to-use interfaces for users to define customized datasets. COGDL provides two ways to construct a customized dataset. One way is to convert raw data files into the required format of COGDL and specify the argument data_path. The other way is to define a customized dataset class with the necessary functions to load and process the data.

Listing 3: Definition of datasets

class CoraDataset(Dataset):
    def process(self):
        """Preprocess data to cogdl.Graph"""
        def get_evaluator(self):
            """Return evaluator defined in COGDL"""
            return accuracy
        def get_loss_fn(self):
            """Return loss function defined in COGDL"""
            return cross_entropy_loss

Model. COGDL implements various graph neural networks as research baselines and for applications. A GNN layer such as GCNLayer has already been implemented based on Graph and efficient sparse operators. A model in the package consists of the model builder (i.e., __init__) and forward propagation (i.e., forward) functions in the PyTorch style. Listing 4 shows the APIs implemented in each model to provide a unified paradigm for usage, where add_args is used to define model-specific hyperparameters for experiments.

Listing 4: Definition of GNN models

```python
from cogdl.layers import GCNLayer

class GCN(BaseModel):
    def add_args(parser):
        parser.add_argument("--hidden", type=int)

    def __init__(self, nfeats, nclasses, hidden):
        self.layer1 = GCNLayer(nfeats, hidden)
        self.layer2 = GCNLayer(hidden, nclasses)

    def forward(self, graph):
        h = graph.x
        h = self.layer1(graph, h)
        h = self.layer2(graph, h)
        return h
```

A.2 Basic Usage

Figure 2 illustrates the benefits of COGDL’s unified trainer and modular wrappers over PyG and DGL. Users can directly use the Trainer for running the experiments and do not need to write tedious code to build the training loop for GNN training and evaluation. Besides, we introduce another easy-to-use usage for conducting experiments through experiment API. We can feed a dataset, a model, and hyper-parameters to the experiment API, which calls the Trainer API. Producing the results with one-line code provides a convenient way for experiments. Furthermore, we integrate a popular library, Optuna [4], into COGDL to enable hyper-parameter search. By feeding the search_space function that defines the search space of hyper-parameters, COGDL will start the searching process for the validation metric and output the best results.

Listing 5: Basic usage of COGDL

```python
from cogdl import experiment

def func(trial):
    """Return loss function defined in COGDL"""
    return accuracy

# basic usage
experiment(dataset="cora", model="gcn", hidden_size=32, max_epoch=200) # basic usage
```

B APPLICATIONS

Our library has been successfully applied to a large-scale academic mining and search system—AMiner (https://www.aminer.cn/)—for the analysis of papers and scholars, as illustrated in Figure 3. There are more than 320 million papers with more than 1 billion citation links in the AMiner database. Most applications (e.g., paper tagging) in AMiner are related to academic papers and scholars. Therefore, it is important to obtain accurate embeddings for papers and scholars for downstream tasks.

Take paper tagging as an example. Each publication in AMiner has several tags extracted from the raw text (e.g., title and abstract) of each publication. However, publications with citation links may have similar tags, and we can utilize the citation network to improve the quality of tags. The paper tagging problem can be considered as a multi-label node classification task, where each label represents a tag. We can utilize paper embeddings to improve the quality of paper tags since we find many papers missing tags in the AMiner database. With the help of the AMiner team, we train a downstream classifier for the paper tagging task on 4.8 million papers in computer science. The experimental results demonstrate that those embeddings could help improve the recall by 12.8% compared with the previous method. In addition to the paper tagging task, the embeddings generated by COGDL can also help the paper recommendation of AMiner. Specifically, the AMiner homepage gives paper recommendations for users based on users’ historical behaviors. The quality of the paper embeddings plays an important role in the recommendation. We hope that our library can help more users to build better services, especially those web-related applications, benefiting the community of graph learning.
Figure 2: Illustration of training the GCN model on the Cora dataset using CogDL, PyG, and DGL. The trainer of CogDL takes over the entire loop of the training and evaluation process. The ModelWrapper consists of training and testing steps, while the DataWrapper handles the data preparation and transformation.

Table 9: Dataset statistics for node classification (“m” stands for multi-label classification)

<table>
<thead>
<tr>
<th>Setting</th>
<th>Dataset</th>
<th>#Nodes</th>
<th>#Edges</th>
<th>#Features</th>
<th>#Classes</th>
<th># Train / Val / Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semi-supervised Setting</td>
<td>Cora</td>
<td>2,708</td>
<td>5,429</td>
<td>1,433</td>
<td>7</td>
<td>140 / 500 / 1,000</td>
</tr>
<tr>
<td></td>
<td>Citeseer</td>
<td>3,327</td>
<td>4,732</td>
<td>3,703</td>
<td>6</td>
<td>120 / 500 / 1,000</td>
</tr>
<tr>
<td></td>
<td>Pubmed</td>
<td>19,717</td>
<td>44,338</td>
<td>500</td>
<td>3</td>
<td>60 / 500 / 1,000</td>
</tr>
<tr>
<td>Fully-Supervised Setting</td>
<td>PPI</td>
<td>56,944</td>
<td>818,736</td>
<td>50</td>
<td>121 (m)</td>
<td>0.79 / 0.11 / 0.10</td>
</tr>
<tr>
<td></td>
<td>Flickr</td>
<td>89,350</td>
<td>899,756</td>
<td>500</td>
<td>7</td>
<td>0.50 / 0.25 / 0.25</td>
</tr>
<tr>
<td></td>
<td>Reddit</td>
<td>232,965</td>
<td>11,606,919</td>
<td>602</td>
<td>41</td>
<td>0.66 / 0.10 / 0.24</td>
</tr>
<tr>
<td></td>
<td>Yelp</td>
<td>716,847</td>
<td>6,977,410</td>
<td>300</td>
<td>100 (m)</td>
<td>0.75 / 0.10 / 0.15</td>
</tr>
<tr>
<td></td>
<td>ogbn-arxiv</td>
<td>169,343</td>
<td>1,166,243</td>
<td>128</td>
<td>40</td>
<td>0.54 / 0.18 / 0.28</td>
</tr>
<tr>
<td>Network Embedding Setting</td>
<td>PPI</td>
<td>3,890</td>
<td>76,584</td>
<td>-</td>
<td>50 (m)</td>
<td>0.50 / - / 0.50</td>
</tr>
<tr>
<td></td>
<td>Wikipedia</td>
<td>4,777</td>
<td>184,812</td>
<td>-</td>
<td>40 (m)</td>
<td>0.50 / - / 0.50</td>
</tr>
<tr>
<td></td>
<td>Blogcatalog</td>
<td>10,312</td>
<td>333,983</td>
<td>-</td>
<td>39 (m)</td>
<td>0.50 / - / 0.50</td>
</tr>
<tr>
<td></td>
<td>DBLP</td>
<td>51,264</td>
<td>127,968</td>
<td>-</td>
<td>60 (m)</td>
<td>0.05 / - / 0.95</td>
</tr>
<tr>
<td></td>
<td>Flickr</td>
<td>80,513</td>
<td>5,899,882</td>
<td>-</td>
<td>195 (m)</td>
<td>0.05 / - / 0.95</td>
</tr>
</tbody>
</table>

Table 10: Dataset statistics for graph classification

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Graphs</th>
<th>#Classes</th>
<th>#Feats</th>
<th>#Nodes</th>
<th>#Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUTAG</td>
<td>188</td>
<td>2</td>
<td>7</td>
<td>17.9</td>
<td>19.8</td>
</tr>
<tr>
<td>PTC</td>
<td>344</td>
<td>2</td>
<td>18</td>
<td>14.3</td>
<td>14.7</td>
</tr>
<tr>
<td>PROTEINS</td>
<td>1,113</td>
<td>2</td>
<td>3</td>
<td>39.1</td>
<td>72.8</td>
</tr>
<tr>
<td>NCI</td>
<td>4,110</td>
<td>2</td>
<td>37</td>
<td>29.8</td>
<td>32.3</td>
</tr>
<tr>
<td>IMDB-B</td>
<td>1,000</td>
<td>2</td>
<td>-</td>
<td>19.8</td>
<td>96.5</td>
</tr>
<tr>
<td>IMDB-M</td>
<td>1,500</td>
<td>3</td>
<td>-</td>
<td>13.0</td>
<td>65.9</td>
</tr>
<tr>
<td>REDDIT-B</td>
<td>2,000</td>
<td>2</td>
<td>-</td>
<td>429.6</td>
<td>497.8</td>
</tr>
<tr>
<td>COLLAB</td>
<td>5,000</td>
<td>3</td>
<td>-</td>
<td>74.5</td>
<td>2457.8</td>
</tr>
</tbody>
</table>

Figure 3: Illustration of how CogDL helps real-world applications in AMiner. Most applications of AMiner rely on a large-scale academic graph. CogDL can help AMiner to generate accurate embeddings for papers and scholars from the billion-scale academic graph for downstream tasks.