GammaGL: A Multi-Backend Library for Graph Neural Networks

Yaoqi Liu
Beijing University of Posts and Telecommunications
Beijing, China
yaoqiliu@bupt.edu.cn

Cheng Yang
Beijing University of Posts and Telecommunications
Beijing, China
Peng Cheng Laboratory
Shenzhen, China
yangcheng@bupt.edu.cn

Tianyu Zhao
Beijing University of Posts and Telecommunications
Beijing, China
tyzhao@bupt.edu.cn

Hui Han
Beijing University of Posts and Telecommunications
Beijing, China
hanhui@bupt.edu.cn

Siyuan Zhang
Beijing University of Posts and Telecommunications
Beijing, China
1580124318@qq.com

Jing Wu
Beijing University of Posts and Telecommunications
Beijing, China
buptfhxy@163.com

Guangyu Zhou
Beijing University of Posts and Telecommunications
Beijing, China
1029175863@qq.com

Hai Huang
Beijing University of Posts and Telecommunications
Beijing, China
hhuang@bupt.edu.cn

Hui Wang
Peng Cheng Laboratory
Shenzhen, China
wangh06@pcl.ac.cn

Chuan Shi[*]
Beijing University of Posts and Telecommunications
Beijing, China
Peng Cheng Laboratory
Shenzhen, China
shichuan@bupt.edu.cn

ABSTRACT

Graph Neural Networks (GNNs) have shown their superiority in modeling graph-structured data, and gained much attention over the last five years. Though traditional deep learning frameworks such as TensorFlow and PyTorch provide convenient tools for implementing neural network algorithms, they do not support the key operations of GNNs well, e.g., the message passing computation based on sparse matrices. To address this issue, GNN libraries such as PyG are proposed by introducing rich Application Programming Interfaces (APIs) specialized for GNNs. However, most current GNN libraries only support a specific deep learning framework as the backend, e.g., PyG is tied up with PyTorch. In practice, users usually need to combine GNNs with other neural network components, which may come from their co-workers or open-source codes with different deep-learning backends. Consequently, users have to be familiar with various GNN libraries, and rewrite their GNNs with corresponding APIs. To provide a more convenient user experience, we present Gamma Graph Library (GammaGL), a GNN library that supports multiple deep learning frameworks as backends. GammaGL uses a framework-agnostic design that allows users to easily switch between deep learning backends on top of existing components with a single line of code change. Following the tensor-centric design idea, GammaGL splits the graph data into several key tensors, and abstracts GNN computational processes (such as message passing and graph mini-batch operations) into a few key functions. We develop many efficient operators in GammaGL for acceleration. So far, GammaGL has provided more than 40 GNN examples that can be applied to a variety of downstream tasks. GammaGL also provides tools for heterogeneous graph neural networks and recommendations to facilitate research in related fields. We present the performance of models implemented by GammaGL and the time consumption of our optimized operators to show the efficiency. Our library is available at https://github.com/BUPT-GAMMA/ GammaGL.

[*]The corresponding author.

SIGIR '23, July 23–27, 2023, Taipei, Taiwan.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9408-6/23/07…$15.00
https://doi.org/10.1145/3539618.3591891

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

Graph Neural Networks, Deep Learning, Frameworks

ACM Reference Format:
Yaoqi Liu, Cheng Yang, Tianyu Zhao, Hui Han, Siyuan Zhang, Jing Wu, Guangyu Zhou, Hai Huang, Hui Wang, and Chuan Shi. 2023. GammaGL: A Multi-Backend Library for Graph Neural Networks. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '23), July 23–27, 2023, Taipei, Taiwan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3539618.3591891

1 INTRODUCTION

Graph is a relational data structure with a set of nodes and edges, and can be used to model many real-world scenarios, e.g., social relation network, citation network, and e-commerce network. To model graph-structured data, Graph Neural Networks (GNNs) can learn effective representations of nodes, edges, or graphs based on deep learning, and have achieved state-of-the-art (SOTA) performance in many graph-based tasks in the last five years [22, 31, 36, 59].

Traditional deep learning frameworks like TensorFlow [1] and PyTorch [34] make it easier and faster to build deep learning algorithms without knowing underlying details. However, implementing GNNs with these traditional frameworks will cause low-efficiency issues due to the following challenges:

- Most deep learning frameworks provide data management for regular data like images or texts. But graphs are non-Euclidean, irregular, and more complex. For example, a heterogeneous graph contains graph topology, type information, and features. Typical deep learning frameworks do not have a proper way to store and query graphs.
- Real-world graphs are usually sparse. Using the calculation methods for dense data will lead to a waste of computing and memory resources. Typical deep learning frameworks usually lack enough support for calculating sparse data.
- Real-world graphs have a large number of nodes that can reach millions or even billions. Graph mini-batch training is a solution to extend the algorithms under memory limits. However, graph mini-batch training involves graph slicing, which needs the consideration of node-dependent properties and makes it non-trivial to get the data for the current batch like images or texts.

To address the above challenges, several GNN libraries like PyTorch-Geometric (PyG) [13] and Deep Graph Library (DGL) [44] were proposed and can provide tools like graph data management, sparse data operation, and large-scale graph training. Some toolkits like CogDL [7], TorchDrug [58], and Recbole-GNN [54, 55] can also help users to implement GNN algorithms for different applications. We list some mainstream GNN libraries and toolkits in Table 1.

However, most current GNN libraries only support a specific deep learning framework as the backend, e.g., PyG is tied up with PyTorch. In practice, it would be much more convenient for users if the library can switch between different deep-learning backends. For example, users usually need to combine GNNs with other neural network components, which may come from their co-workers or open-sourced codes with different deep learning backends. In this case, users have to be familiar with various GNN libraries, and rewrite their GNNs with corresponding APIs. Moreover, some users may have hardware limitations, e.g., they only have Huawei-Ascend instead of Nvidia-GPU. Note that the supported hardware devices of GNN libraries are determined by the backends. Most existing GNN libraries are based on TensorFlow or PyTorch, and only support Nvidia-GPU hardware. Supporting multiple backends will also increase the hardware usability of a GNN library.

In this work, we introduce Gamma Graph Library (GammaGL), a multi-backend library for GNNs based on TensorLayerX (TLX)[1] [30], a deep-learning framework encapsulating four backends. We summarize the features of GammaGL as follows:

- **Multi-backend support**: GammaGL supports multiple deep learning backends, including TensorFlow [1], PyTorch [34], PaddlePaddle [33], MindSpore[2]. GammaGL allows users to switch between different backends with a single line of code change. Thus, a GNN will be implemented with the same script across different backends. After specifying a backend in GammaGL, users can also use other APIs in the backend simultaneously.
- **Tensor-centric**: Most GNN libraries and toolkits adopt tensor-centric design principles such as PyG. To provide users with familiar interfaces and help them learn our library quickly, GammaGL also utilizes tensor-centric APIs and is friendly to users familiar with existing GNN libraries.
- **Heterogeneous GNN models**: Compared with typical homogeneous graphs, heterogeneous graphs contain richer information on node and edge types, and are widely used in modeling recommendation systems, e.g., the interactions among users, items, and sellers can be represented as a heterogeneous graph. GammaGL also implements popular heterogeneous graph neural network models, tools, and pipelines to assist relevant research and applications.
- **Recommendation pipeline**: Graph-based recommendation is one of the most popular recommendation scenarios. GammaGL provides several graph-based recommendation models and a training pipeline for users. Users may utilize the pipeline and tools to design and evaluate their graph-based recommendation models.

We will introduce GammaGL in detail in the following sections. The core design of GammaGL will be described in Section 2. In Section 3, we will make a comparison with other GNN libraries and show our features. Section 4 will conduct experiments on popular graph tasks and present the performance of our efficient operators.

2 SYSTEM DESIGN

In this section, we will present the system design of GammaGL. As shown in Figure 1, GammaGL is built on TensorLayerX [30], a multi-backend AI framework, which covers most deep learning features. However, as TensorLayerX lacks functions for GNN, we design three key components: data management, message passing paradigm, and sampling operators for building GNN models. We also implement and encapsulate computation operations across backends. We will introduce each key module in the following parts.

[1]https://github.com/tensorlayer/TensorLayerX
[2]https://github.com/mindspore-ai/mindspore

Table 1: Open-source GNN libraries and toolkits. (framework-neutral means supporting multi-backend but requires more code modifications when switching backends, while framework-agnostic means supporting multi-backend with a single line of code change when switching backends.)

	Multi-Backend	Programming Style	Heterogeneous GNN	Efficient Message Passing Operator	Efficient Sampling Operator	Recommendation Pipeline
PyG [13]	-	tensor-centric	✓	✓	✓	-
DGL [44]	framework-neutral	graph-centric	✓	✓	✓	-
PGL [33]	-	graph-centric	✓	✓	✓	-
TFG [24]	-	tensor-centric	-	-	✓	-
Graph-learn [56]	-	tensor-centric	✓	✓	✓	-
CogDL [7]	-	tensor-centric	✓	✓	-	-
TorchDrug [58]	-	tensor-centric	-	-	-	-
StellarGraph [11]	-	tensor-centric	✓	-	✓	-
Spektral [17]	-	tensor-centric	-	✓	-	-
RecBole-GNN [54, 55]	-	tensor-centric	✓	-	✓	✓
GammaGL	framework-agnostic	tensor-centric	✓	✓	✓	✓

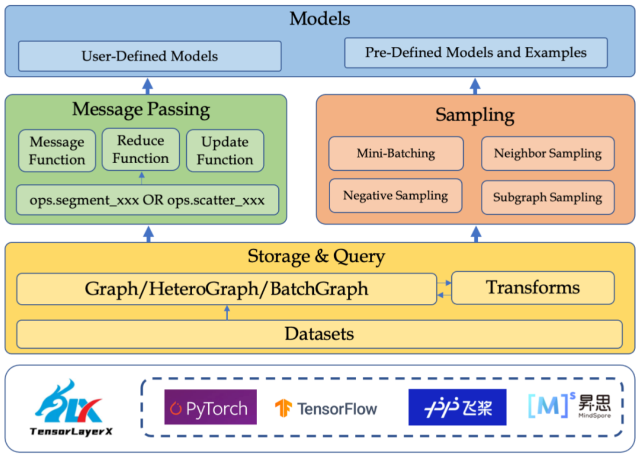


Figure 1: The architecture of GammaGL.

2.1 Multi-backend Support

To satisfy the diversity of users' commands, the most exciting design principle of GammaGL is supporting multi-backend. Each module in GammaGL is compatible with four mainstream backends with a unified API, which allows users to switch backends by setting an environment variable. Next, we will introduce the design of our multi-backend support.

Like most domain packages, GammaGL is built on top of the deep learning frameworks, enjoying most deep learning features (e.g., automatic differentiation, rich neural networks, and operation, data management). To support different backends, GammaGL is built on TensorLayerX, which is a multi-backend deep learning framework that encapsulates TensorFlow [1], PyTorch [34], PaddlePaddle [33], MindSpore, and so on. Besides, it allows users to run the code on different hardware like Nvidia-GPU, Huawei-Ascend, Cambricon, and more.

Although TensorLayerX provides rich deep learning APIs to implement neural networks across backends, it still lacks the necessary functions for GNN implementation, which is a common problem for traditional frameworks like Tensorflow, and PyTorch. Therefore, GammaGL builds three key modules to support GNN implementation based on native operators of TensorLayerX, which could also support different backends. Besides, for efficiency, GammaGL also implements and encapsulates computation operations across backends, which will be introduced later. As for modules outside of GNN, GammaGL could directly use the operations in existing frameworks without reinventing the wheel.

2.2 Data Management

Although most deep learning frameworks provide data management features, they are only suitable for regular data like images and texts and unfriendly for irregular data like graphs. GammaGL builds the data management module to abstract graph data and process datasets, which offer data storage and query for other modules.

2.2.1 Graph Data Abstraction. The basic graph data structures in GammaGL are *Graph*, *HeteroGraph*, and *BatchGraph*, which are used for homogeneous, heterogeneous, and batch graphs, respectively. In this part, we will describe the details of the data storage.

Graph. In graph theory, researchers study graphs with a set of nodes and edges, i.e, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ means node sets, $\mathcal{E}$ means edges representing nodes connectivity. GammaGL introduces $\mathcal{G} = (A, X, Y)$, where $A \in R^{|\mathcal{V}|\times|\mathcal{V}|}$ is the adjacency matrix representing the connectivity of a graph, as shown in Eq. (1). Although A contains rich structural information, it is far from enough to store feature (or attribute) information in deep learning. So $X \in R^{|\mathcal{V}|\times d}$ is defined to represent the node feature matrix used in core computation named Message Passing in GNN, where d is the feature dimension. To optimize the model, the label information is introduced, e.g., Y is the node label matrix.

Graph used to represent real-world scenarios is usually sparse, which means most elements of the adjacency matrix A are zero. Therefore, GammaGL denotes graph structure with a sparse matrix in a coordinate (COO) format. Specifically, a triplet (i, j, k) can represent a nonzero element in A, where (i, j) is the edge index, and k is the edge weight, which corresponds to the k in EQ.(1). If not considering node type or edge type, GammaGL models *graph* as a homogeneous graph with (*edge_index, edge_weight, x, y*), where *edge_index* with *edge_weight*, *x*, and *y* correspond to the matrix A, X, and Y, respectively.

$$A_{ij} = \begin{cases} k & (i,j) \in \mathcal{E}, k \neq 0 \\ 0 & (i,j) \notin \mathcal{E} \end{cases} . \tag{1}$$

Processing the adjacency matrix with the COO sparse matrix has the following advantages:

- Compared with dense matrices, COO sparse matrices can save more memory consumption in most graphs.
- The core computation of GNN is message passing, which means the computation complexity is proportional to the number of edges. The COO sparse matrix in GNN will reduce the computation compared with the dense matrix.

HeteroGraph. Heterogeneous graphs are graphs with more than one node type or edge type, which contain richer semantic information and a more complex structure. As a powerful method, Heterogeneous Graph Neural Network (HGNN) models are applied in many scenarios with more complex model structures such as graph-based recommendations. Providing enough HGNN-related APIs will facilitate the development of HGNN and other applications. Heterogeneous graphs can be treated as a set of relational subgraphs, which is a set of nodes and edges with the same edge type. Therefore, in GammaGL, heterogeneous graphs are represented as *HeteroGraph*, which contains multi-relational sub-graphs [53]. As for differences in types and dimensionality, the feature of nodes will be processed into a dictionary with node type as the key, and features as the value. To summarize, A heterogeneous graph can be modeled as (*edge_index_dict, edge_weight_dict, x_dict, y_dict*), where the elements are nested with a dictionary. The other design principle is similar to that of the *Graph* object.

BatchGraph. The graph-level task aims to research a batch of graphs. However, due to the different number of nodes in graphs, calculating on a batch of graphs simultaneously with traditional deep learning frameworks is not easy. In GammaGL, we put the adjacency matrix of each graph into the diagonal of a chunking matrix, and we concatenate the node features of graphs in the node dimension so that we can carry out training on multiple graphs in parallel. The object that contains multiple graphs in GammaGL is *BatchGraph*. A *BatchGraph* can be modeled as (*edge_index, edge_weight, x, y, offset*), where *edge_index*, *edge_weight*, *x*, and *y* are inheriting from *Graph* or *HeteroGraph*. *offset* is an integer vector that is used to determine which nodes each graph contains. For example, The *i*-th graph contains nodes with IDs from *offset*[*i*] to *offset*[*i+1*].

2.2.2 Dataset Management. Although traditional deep learning frameworks like PyTorch have provided data management modules, they still lack the core functions for graph data pre-processing and transformation. For example, users wish to add metapath in heterogeneous graphs to support some GNN models like HAN [46], which is difficult using modules in deep learning frameworks. Therefore, the graph dataset management module is provided in GammaGL. With this module, users can do any complex pre-processing or transformation operations.

The dataset module is used for downloading, processing, saving, and loading data from external resources. GammaGL supports a lot of built-in datasets covering various applications like node classification, link prediction, graph classification, and APIs for customized datasets.

As some common operations like downloading, and file management have already been done in our base class, users can create their datasets by inheriting the base class and making their adjustments. For example, users can specify the downloading URLs, and perform the pre-processing operations they expect.

2.3 Message Passing Module

GNN message passing is considered the most popular computation paradigm in GNN at present. Most of the GNN algorithms can be expressed as three steps: message creation, neighbor aggregation, and feature update. Here are the detailed expressions:

$$\begin{aligned} m^{(l+1)} &= \phi\left(x_i^{(l)}, x_j^{(l)}, w_e^{(l)}\right), (i,e,j) \in \mathcal{E} \\ h^{(l+1)} &= \rho\left(\left\{m^{(l+1)} : (i,e,j) \in \mathcal{E}\right\}\right) \\ x_j^{(l+1)} &= \psi\left(h^{(l+1)}, x_j^{(l)}\right), j \in \mathcal{V} \end{aligned} \tag{2}$$

where ϕ is the message creation function, ρ is the neighbor aggregation function, ψ is the feature update function, i and j is the source and target node, e is the edge, $x_i^{(l)}$ is the feature of node i in layer l, and $w_e^{(l)}$ is the edge weight of edge e in layer l.

The message passing module, as the computational core component of GNNs, determines the computational efficiency of GNN libraries. Traditional deep learning frameworks lack the support for GNN computation as the GNN computation is essentially sparse, while traditional deep learning frameworks and TensorLayerX focus on dense tensor computation, and provide less support for sparse computation. Therefore, we have developed a unified message-passing module across four backends, which can be considered an extension of TensorLayerX. We have also optimized the inefficient operators by using underlying optimization to be compatible with hardware after comparing the efficiency of operators under different backends. Moreover, following the design of PyG and DGL, we have developed efficient fused operators [49]. As shown in Eq. (2), in the message creation step, common methods are gathering features from nodes and packing them into message tensors, which consumes more memory. By fusing the message creation and neighbor aggregation, we can avoid edge data generation and improve the efficiency of the computation.

MessagePassing is the base class for message passing in GammaGL which contains functions *message*, *aggregate*, and *update* that correspond to the three steps in Eq. (2). All graph neural layers designed based on message passing need to inherit from this class and override some of the functions as needed. The detailed description is listed here.

- *message*: The message function is mainly used in the process of message creation and sending to the edge. For example, the message function in GCN [29] returns the features of the source node itself, and the message function in GAT [41]

calculates the attention coefficients of the source and target nodes through the attention mechanism and uses the coefficients as the weights of the sent messages.
- *aggregate*: The aggregate function is responsible for the process of aggregating messages from neighbor edges, which generally takes the form of *sum*, *mean*, *max*, *min*, *LSTM* [23], etc. GCN, and GAT all use the *sum* method.
- *update*: The update function usually returns the input value itself. In some cases, such as models that require adding residual connections, the update function needs to be modified.

For fused operations, we also provide method *message_aggregate*, which integrates the process of message creation and neighbor aggregation. Using this method can improve the efficiency of some concise aggregation models, such as GCN [29], SGC [48], etc.

2.4 Sampling Operator

Real-world graphs are usually large and may contain millions or billions of nodes. Directly applying GNN on a such large graph will result in out-of-memory (OOM). In computer vision, researchers can directly split images into multi-batches and train models with a mini-batch of images so that a common computer can train the model. Traditional deep learning frameworks and TensorLayerX provide mini-batch APIs for these independent data. However, graph is data-dependent, which means the random splitting causing information lack could not be simply applied to graphs. To get the embedding of target nodes, we need to sample the neighbors of these nodes while traditional deep learning frameworks lack such domain-specific operations. Therefore, it is necessary to develop a sampling module to deal with this challenge.

We have designed *NeighborSampler* to sample neighbors for a batch of target nodes. The sampling procedure will not involve the gradient descent algorithm, which means this module is independent of the deep learning framework. To sample neighbors efficiently, GammaGL develops operations with Cython and achieves parallel sampling on GPU. Next, we construct batches of bipartite graphs with the sampled nodes and extract corresponding node features. Last, we feed these bipartite graphs with features into the model and train the model as usual.

2.5 Implemented GNN Models

Based on the key components of GammaGL, we have implemented 40+ GNN models covering various methods like supervised, unsupervised, random walk, and heterogeneous graph learning. The models are listed in Table 2. The performance in Section 4 can demonstrate the effectiveness of our implementation.

3 CONNECTION AND DIFFERENCE WITH OTHER GNN LIBRARIES

Several GNN libraries have emerged in recent years. For example, PyTorch-Geometric (PyG) [13] and Deep Graph Library (DGL) [44] are the most widely used libraries. tf_geometric [24], Spekral [17], and StellarGraph [11] are libraries based on TensorFlow. Paddle Graph Library (PGL) [33] is an efficient and easy-to-use GNN library based on PaddlePaddle. Graph-learn [56] is a distributed framework designed for the development and application of large-scale GNNs.

Table 2: Implemented GNNs in GammaGL.

Supervised Learning	GCN [29]	GAT [41]
	GraphSAGE [19]	GCNII [8]
	ChebNet [12]	JKNet [51]
	DGCNN [47]	APPNP [16]
	AGNN [39]	SIGN [14]
	GNN-FiLM [5]	DropEdge [36]
	HardGAT [15]	MixHop [2]
	HCHA [3]	GATv2 [6]
	GEN [45]	FAGCN [4]
	SGC [48]	GAE [28]
	VGAE [28]	SEAL [52]
	GIN [50]	PNA [10]
Unsupervised Learning	GPRGNN [9]	GRACE [57]
	GraphGAN [43]	MERIT [27]
	MVGRL [21]	DGI [42]
	InfoGraph [38]	
Random Walk Model	Node2Vec [18]	DeepWalk [35]
HGNN	RGCN [37]	HAN [46]
	HGT [25]	SimpleHGN [32]
	HPN [26]	LightGCN [22]
	CompGCN [40]	

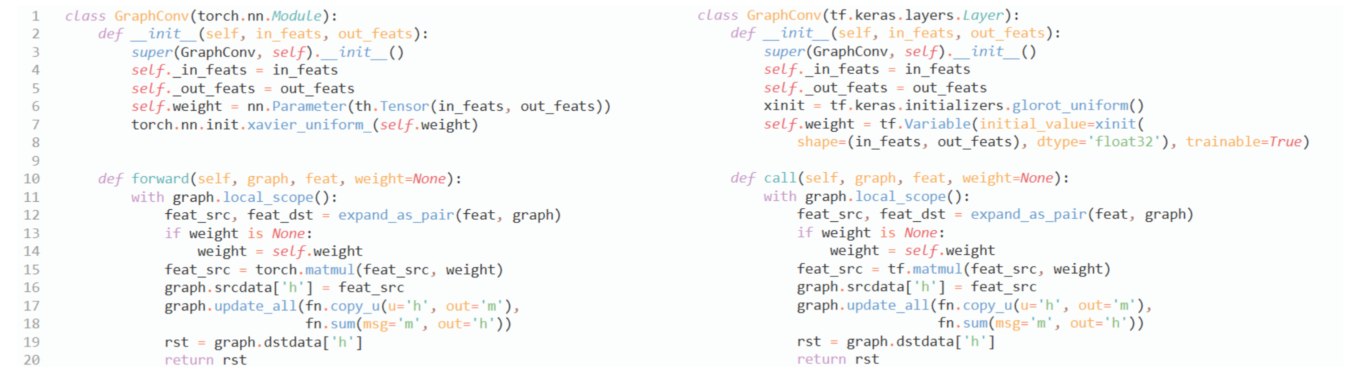
Here, we will give a detailed discussion about the connection and difference between GammaGL and these libraries.

3.1 Framework-agnostic

Most libraries or toolkits shown in Table 1 only support a single backend. In practical applications, users prefer a certain deep learning framework for its unique features or API style and some modules outside of the GNN library may be implemented with other deep learning frameworks. Users who want to use these GNN libraries or toolkits have to use the backend they support or convert the implemented modules to the corresponding code implementation. However, in GammaGL, users do not have to worry about this problem. Users can choose their preferable deep learning framework among the four backends.

Although DGL can support multi-backend, it is a framework-neutral design, which means only graph-related operations are general to multi-backend, while other modules are associated with backends. As a result, for a complete GNN training procedure, users need to modify part of the codes when switching different backends. For example, we can see from Figure 2, which gives an example of implementing the GCN convolution layer using DGL with PyTorch and Tensorflow, respectively. Users who want to switch the backends in DGL need to modify part of the code. We summarize the changes as follows:

- The parent class to be inherited, e.g., modify *torch.nn.Module* to *tensorflow.keras.layers.Layer* in line 1.
- The parameters and the initialization in lines 6, 7, and 8.
- The tensor operations related to the deep learning frameworks in line 15.



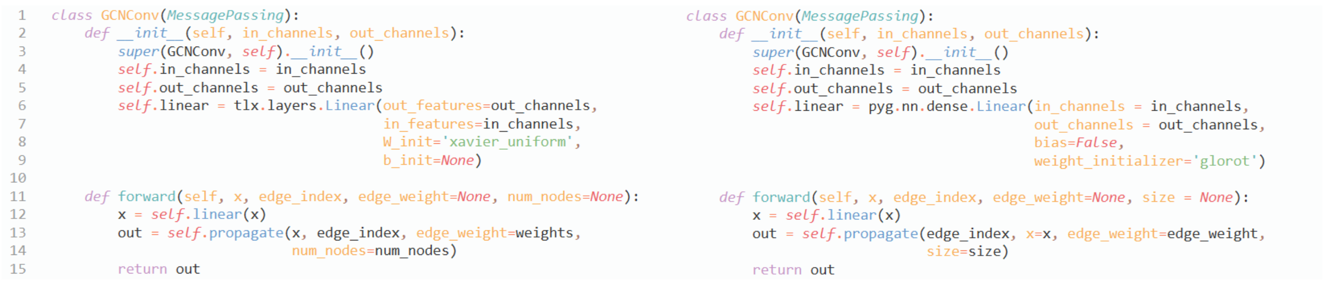
```python
class GraphConv(torch.nn.Module):
    def __init__(self, in_feats, out_feats):
        super(GraphConv, self).__init__()
        self._in_feats = in_feats
        self._out_feats = out_feats
        self.weight = nn.Parameter(th.Tensor(in_feats, out_feats))
        torch.nn.init.xavier_uniform_(self.weight)


    def forward(self, graph, feat, weight=None):
        with graph.local_scope():
            feat_src, feat_dst = expand_as_pair(feat, graph)
            if weight is None:
                weight = self.weight
            feat_src = torch.matmul(feat_src, weight)
            graph.srcdata['h'] = feat_src
            graph.update_all(fn.copy_u(u='h', out='m'),
                             fn.sum(msg='m', out='h'))
            rst = graph.dstdata['h']
            return rst
```

```python
class GraphConv(tf.keras.layers.Layer):
    def __init__(self, in_feats, out_feats):
        super(GraphConv, self).__init__()
        self._in_feats = in_feats
        self._out_feats = out_feats
        xinit = tf.keras.initializers.glorot_uniform()
        self.weight = tf.Variable(initial_value=xinit(
            shape=(in_feats, out_feats), dtype='float32'), trainable=True)

    def call(self, graph, feat, weight=None):
        with graph.local_scope():
            feat_src, feat_dst = expand_as_pair(feat, graph)
            if weight is None:
                weight = self.weight
            feat_src = tf.matmul(feat_src, weight)
            graph.srcdata['h'] = feat_src
            graph.update_all(fn.copy_u(u='h', out='m'),
                             fn.sum(msg='m', out='h'))
            rst = graph.dstdata['h']
            return rst
```

Figure 2: The implementation of GCN convolution with PyTorch and Tensorflow in DGL. (The implementation on the left is using PyTorch and the right is Tensorflow, which is the full name of tf.)



```python
class GCNConv(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super(GCNConv, self).__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.linear = tlx.layers.Linear(out_features=out_channels,
                                        in_features=in_channels,
                                        W_init='xavier_uniform',
                                        b_init=None)

    def forward(self, x, edge_index, edge_weight=None, num_nodes=None):
        x = self.linear(x)
        out = self.propagate(x, edge_index, edge_weight=weights,
                             num_nodes=num_nodes)
        return out
```

```python
class GCNConv(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super(GCNConv, self).__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.linear = pyg.nn.dense.Linear(in_channels = in_channels,
                                          out_channels = out_channels,
                                          bias=False,
                                          weight_initializer='glorot')

    def forward(self, x, edge_index, edge_weight=None, size = None):
        x = self.linear(x)
        out = self.propagate(edge_index, x=x, edge_weight=edge_weight,
                             size=size)
        return out
```

Figure 3: The implementation of GCN convolution with GammaGL and PyG. (The implementation on the left is using GammaGL and the right is PyG.)

However, the design of GammaGL is truly framework-agnostic. The code implemented in GammaGL not only supports multi-backend but also needs a single line of code change when switching different backends. We can refer to the left code in Figure 3. Our implementation has already supported the four backends. Users only need to modify a single line to specify their backend without changing other codes. Here are two ways to specify backends in GammaGL.

```
# Command line
TL_BACKEND=tensorflow python train.py
```

```python
# Python Script
# Support tensorflow, torch, paddle, mindspore
import os
os.environ['TL_BACKEND'] = 'tensorflow'
```

3.2 Tensor-centric

Most of the GNN libraries and toolkits utilize a tensor-centric design. For example, PyG the most widely used GNN library provides tensor-centric API. GammaGL also adopts the tensor-centric design. As many users are familiar with these APIs, our APIs are designed to be as similar as possible to the design of other libraries and toolkits. The code snippet presented in Figure 3 gives an example of implementing a toy graph convolution layer with GammaGL and PyG. We can find that the code in GammaGL is similar to PyG. Moreover, if users want to import the modules from other libraries that are based on the backend we support, they can directly use them without modifying too many lines of code in GammaGL.

Some other libraries like DGL use graph-centric design, which takes graphs as the central object. Graph-centric design can improve software consistency but can be more difficult for beginners to understand. We can compare Figure 2 and Figure 3, the graph-centric and the tensor-centric code example. The algorithms are built around the graph in DGL, while the algorithms are built around the tensor in GammaGL.

3.3 Heterogeneous GNN

Heterogeneous graphs are powerful data structures to model real-world scenarios such as recommendations. They often contain complex structures and rich semantic information. HGNNs can be used to learn the representations of heterogeneous graphs and are getting popular for their excellent performance. However, processing heterogeneous graphs is difficult. On the one hand, the source data can be stored in different formats. On the other hand, the complexity of HGNNs leads to a diversity of heterogeneous graph structures [20].

Some GNN libraries like TFG, TorchDrug, and Spektral do not have any modules for HGNNs. Other libraries like DGL and PyG take heterogeneous graphs as a Python dictionary of relational sub-graphs. They also provide useful tools to pre-process heterogeneous graphs and implement HGNN algorithms.

In GammaGL, we use a similar design to PyG and DGL. We store heterogeneous graphs as a dictionary. We have implemented some utilities for heterogeneous graphs pre-processing. We also provide more HGNN examples for users to choose from compared with PyG and DGL.

Table 3: Description of the datasets.

	#Nodes	#Edges	#Ntypes	#Etypes	#Graphs
Cora	2,708	10,556	1	1	1
PubMed	19,717	88,648	1	1	1
MUTAG	avg 17.9	avg 39.6	1	1	188
IMDB-BINARY	avg 19.8	avg 193.1	1	1	1,000
IMDB	11,616	17,106	3	2	1
ogbn-arxiv	169,343	1,166,243	1	1	1
Reddit	232,965	114,615,892	1	1	1

3.4 Recommendation

The graph-based recommendation is a downstream task of GNN. Many GNN models for recommendations are emerging. However, many GNN libraries and toolkits do not provide a pipeline for recommendations except Recbole-GNN, which is a toolkit specially designed for graph-based recommendations.

In GammaGL, we provide a graph-based recommendation pipeline, which contains data processing like data downloading and negative sampling, several models for the graph-based recommendation, and the definition of loss functions.

4 EVALUATION

To verify the consistency of performance across different backends, we conduct several experiments on different tasks, like node classification, link prediction, and graph classification, which are mainstream GNN research scenarios. Besides, we also give the time consumption of our efficient operators compared with PyG and DGL operators.

All of the datasets we used are listed in Table 3. The datasets *Cora* and *PubMed* are used for node classification and link prediction. The datasets *MUTAG* and *IMDB-BINARY* are used for graph classification, which contain multiple graphs so the number of nodes and edges are averaged. The datasest *Cora*, *PubMed*, *ogbn-arxiv*, and *Reddit* are used to test the efficiency of our operators.

4.1 Node Classification

Under a semi-supervised setting, the node classification task is to classify the nodes utilizing a small part of node labels and graph structure information. To reduce variance, we repeat 5 times, and report average and standard deviation results.

We first conduct node classification experiments on homogeneous GNN models like GCN [29], GAT [41], GraphSAGE [19], AGNN [39], SGC [48] with dataset *Cora* and *PubMed*. As shown in Table 4, the performance of the models implemented in GammaGL shows a high reproducibility of the results reported in the respective papers and are consistent under different backends. The performance ranking order of models is almost consistent across different backends.

We also conduct node classification tasks on heterogeneous GNN models like RGCN [37], HGT [25], HAN, HPN [26] with dataset *IMDB* under the same setting. Table 7 lists the performance of these models. Some results like the RGCN results may not be satisfactory as the implementation of HGNN is complex and it may not be suitable for this dataset. The other results show the effectiveness of the HGNN models in GammaGL.

4.2 Link Prediction

Link prediction is an important task in GNNs, which is applied in many scenarios like recommendations. The task usually splits existing edges as train/valid/test edges. We evaluate models GAE [28] and VGAE [28] on dataset *Cora* and *PubMed* under four backends. We repeat 5 times, and report the average and standard deviation of the area under the ROC curve (AUC) in Table 5. Due to the random split, the variance resulting from the backend is larger than the node classification task. However, the errors across different backends are almost less than 1%.

4.3 Graph Classification

Graph-level tasks could be applied in many promising scenarios, like drug discovery. We choose models infograph [38] and GIN [50], which is designed for graph classification, to run on dataset *MUTAG* and *IMDB-BINARY* with four backends. Similar to node classification, we also run five times to get the average accuracy and list them in Table 6. Although the variance of GIN is much larger than infograph, different backends represent the variance of the same size, which verifies the consistency of implementation in different backends of GammaGL.

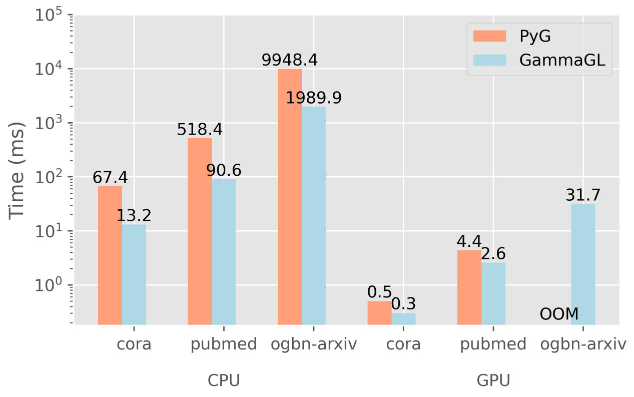


Figure 4: The time consumption of *PyG.scatter_max* and *GammaGL.segment_max*. (The left outcome is running on CPU and the right is running on GPU, OOM means out of memory.)

.

4.4 Efficiency Experiment

Traditional deep learning frameworks and TensorLayerX do not offer high-level API for message passing and sampling operations, so we optimize and develop some low-level operators and encapsulate them into APIs for high-level applications.

Table 4: The accuracy of node classification. (%)

	Cora				PubMed			
Model \ Backend	TensorFlow	PyTorch	PaddlePaddle	Mindspore	TensorFlow	PyTorch	PaddlePaddle	Mindspore
GCN	81.92±0.83	81.86±0.55	81.83±0.22	81.50±0.64	79.50±0.45	79.08±0.27	78.62±0.30	79.28±0.17
GAT	83.26±0.96	82.44±0.43	83.54±0.75	82.90±0.53	78.64±0.41	78.50±0.75	78.82±0.71	78.62±0.52
GraphSAGE	83.32±0.84	81.13±1.08	82.94±0.72	82.24±0.92	78.61±0.58	78.81±0.64	78.43±0.62	77.80±0.83
AGNN	83.28±0.64	83.00±0.65	83.48±0.35	83.16±0.43	79.02±0.82	79.10±0.20	78.94±0.43	79.80±0.40
SGC	81.45±0.37	81.69±0.18	81.65±0.20	81.62±0.32	79.10±0.00	79.16±0.05	79.17±0.05	78.83±0.12

Table 5: The AUC of link prediction. (%)

	Cora				PubMed			
Model \ Backend	TensorFlow	PyTorch	PaddlePaddle	Mindspore	TensorFlow	PyTorch	PaddlePaddle	Mindspore
GAE	91.30±0.85	92.02±0.44	91.16±0.73	91.05±0.58	81.92±0.83	81.86±0.55	81.83±0.22	81.50±0.64
VGAE	92.91±0.62	90.80±0.32	91.42±0.23	91.23±0.47	83.26±0.96	82.44±0.43	83.54±0.75	82.90±0.53

Table 6: The accuracy of graph classification. (%)

	MUTAG				IMDB-BINARY			
Model \ Backend	TensorFlow	PyTorch	PaddlePaddle	Mindspore	TensorFlow	PyTorch	PaddlePaddle	Mindspore
infograph	89.20±0.25	90.65±0.20	89.20±0.53	89.56±0.36	72.00±0.15	72.15±0.23	72,13±0.12	72.11±0.14
GIN	89.76±5.59	89.39±4.22	90.10±5.72	89.49±5.56	83.80±5.71	82.40±5.32	81.80±5.64	81.40±5.36

Table 7: The accuracy of HGNN on *IMDB*. (%)

Model \ Backend	TensorFlow	PyTorch	PaddlePaddle	Mindspore
RGCN	48.54±0.62	48.30±1.20	48.44±1.10	48.26±0.94
HGT	55.98±2.09	54.93±1.34	54.51±1.99	54.71±1.59
HAN	57.78±0.51	55.66±1.05	56.58±0.51	56.74±0.68
HPN	58.05±0.38	57.23±0.47	57.75±0.34	57.51±0.36

4.4.1 Message Passing Operators. For message-passing operators, we have developed the *segment_max* and *segment_sum* in the PyTorch and PaddlePaddle backends, respectively. In addition, we refer to the operator-fusion strategy used in PyG and DGL and design the fused operator *spmm_sum* to fuse the message creation and aggregation, which can greatly reduce memory and time consumption of message passing computation. We conduct our experiments on CPU and GPU, respectively, with datasets Cora, PubMed, and ogbn-arxiv by randomly assigning a 256-dimension feature to each node. For *segment_sum*, we use the PaddlePaddle backend to test our original implementation while PyG uses the PyTorch backend as it does not support PaddlePaddle. For *spmm_sum*, we use PyTorch as the backend to ensure the fairness of the test. Our experiment environment is also listed below.

Machine Environment. Our experiments are conducted on the servers with RTX 2080Ti (12GB version) and Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz. The software environment includes torch 1.10.1+cu111, tensorflow 2.8.0, paddlepaddle 2.3.2.post111, mindspore 1.8.1, dgl 0.9.0, and torch-scatter 2.0.9.

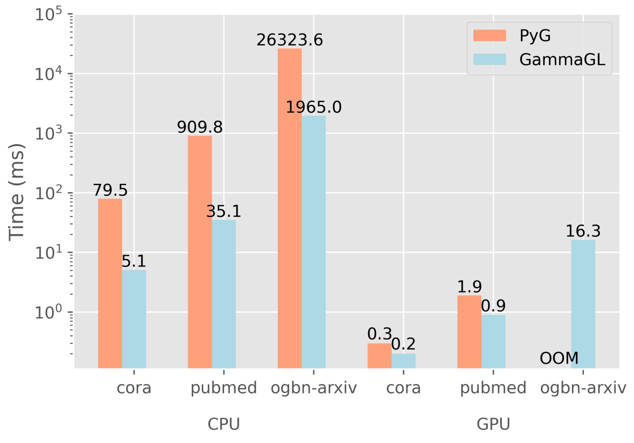


Figure 5: The time consumption of *PyG.scatter_sum* and *GammaGL.segment_sum*. (The left outcome is running on CPU and the right is running on GPU, OOM means out of memory.)

Figure 4 and Figure 5 show the time consumption of PyG operators, and the optimized implementation of *segment_max* and *segment_sum*, respectively. The operators in PyG are *scatter_max* and *scatter_sum*, but we do not get similar operators in DGL due to its graph-centric design. For the *segment_max* operator, our optimized implementation has achieved about 5× on CPU and 1.6× in GPU compared with *scatter_max* in PyG. Our optimized operator

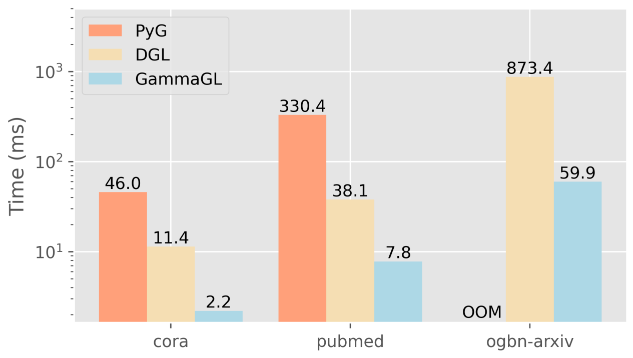


Figure 6: The time consumption of *PyG.gather+scatter_sum*, *DGL.copy_u+sum*, and *GammaGL.spmm_sum*. (Only tested in GPU, OOM means out of memory.)

Table 8: Time consumption (s) of sampling in different GNN libraries.

	PyG	DGL	GammaGL
Full Sample (CPU)	11.00±0.50	9.80±0.20	11.65±0.20
Sub-graph Sample (CPU)	10.50±0.30	9.50±0.20	11.20±0.20
GPU Sample	1.60±0.05	0.53±0.04	2.55±0.01

can still perform efficient operations, while operators in PyG would run out of memory. For the *segment_sum* operator, it is about 15× on CPU and 1.8× on GPU faster compared with the *scatter_sum* in PyG.

From Figure 6, we can see the *spmm_sum* on GPU gets the best performance compared with PyG and DGL. The *copy_u+sum* in DGL can also generate a fused operator, which performs similar functions. We get 30× faster compared with *gather+scatter_sum* in PyG, and 8× compared with *copy_u+sum* in DGL.

4.4.2 Sampling operators. The sampling operators in GammaGL are developed using Cython without any deep learning framework. We also implement sampling on GPU to speed up the sampling process.

We perform experiments on the Reddit-self-loop dataset, with the number of neighbors of the two layers 25 and 10, respectively. The outcome is listed in Table 8. Although the time consumption in PyG and DGL is less than that of our sampling operator, which is because the operators implemented in PyG and DGL are optimized with efficient C++ kernels, they run at the same order of magnitude. Our GPU sampling operator is 4× faster than the CPU sampling.

5 CONCLUSION

We present Gamma Graph Library (GammaGL), a GNN library that supports four deep-learning backends. GammaGL takes framework-agnostic, and tensor-centric as the core design and develops several efficient operators to optimize the GNN computation. GammaGL has supported many models covering various applications. Now, we are actively developing GammaGL and will continue to improve each module in the future.

ACKNOWLEDGMENTS

This work is supported in part by the National Natural Science Foundation of China (No. U20B2045, U1936220, 62192784, 62172052, 62002029, 61772082).

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: a system for large-scale machine learning.. In *Osdi*, Vol. 16. Savannah, GA, USA, 265–283.
[2] Sami Abu-El-Haija, Bryan Perozzi, Amol Kapoor, Nazanin Alipourfard, Kristina Lerman, Hrayr Harutyunyan, Greg Ver Steeg, and Aram Galstyan. 2019. Mixhop: Higher-order graph convolutional architectures via sparsified neighborhood mixing. In *international conference on machine learning*. PMLR, 21–29.
[3] Song Bai, Feihu Zhang, and Philip HS Torr. 2021. Hypergraph convolution and hypergraph attention. *Pattern Recognition* 110 (2021), 107637.
[4] Deyu Bo, Xiao Wang, Chuan Shi, and Huawei Shen. 2021. Beyond low-frequency information in graph convolutional networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 3950–3957.
[5] Marc Brockschmidt. 2020. Gnn-film: Graph neural networks with feature-wise linear modulation. In *International Conference on Machine Learning*. PMLR, 1144–1152.
[6] Shaked Brody, Uri Alon, and Eran Yahav. 2022. How Attentive are Graph Attention Networks?. In *International Conference on Learning Representations*.
[7] Yukuo Cen, Zhenyu Hou, Yan Wang, Qibin Chen, Yizhen Luo, Zhongming Yu, Hengrui Zhang, Xingcheng Yao, Aohan Zeng, Shiguang Guo, Yuxiao Dong, Yang Yang, Peng Zhang, Guohao Dai, Yu Wang, Chang Zhou, Hongxia Yang, and Jie Tang. 2021. CogDL: A Toolkit for Deep Learning on Graphs. *arXiv preprint arXiv:2103.00959* (2021).
[8] Ming Chen, Zhewei Wei, Zengfeng Huang, Bolin Ding, and Yaliang Li. 2020. Simple and deep graph convolutional networks. In *International conference on machine learning*. PMLR, 1725–1735.
[9] Eli Chien, Jianhao Peng, Pan Li, and Olgica Milenkovic. 2021. Adaptive Universal Generalized PageRank Graph Neural Network. In *International Conference on Learning Representations*.
[10] Gabriele Corso, Luca Cavalleri, Dominique Beaini, Pietro Liò, and Petar Veličković. 2020. Principal neighbourhood aggregation for graph nets. *Advances in Neural Information Processing Systems* 33 (2020), 13260–13271.
[11] CSIRO's Data61. 2018. StellarGraph Machine Learning Library. https://github.com/stellargraph/stellargraph.
[12] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in neural information processing systems* 29 (2016).
[13] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).
[14] Fabrizio Frasca, Emanuele Rossi, Davide Eynard, Ben Chamberlain, Michael Bronstein, and Federico Monti. 2020. Sign: Scalable inception graph neural networks. *arXiv preprint arXiv:2004.11198* (2020).
[15] Hongyang Gao and Shuiwang Ji. 2019. Graph representation learning via hard and channel-wise attention networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 741–749.
[16] Johannes Gasteiger, Aleksandar Bojchevski, and Stephan Günnemann. 2019. Combining Neural Networks with Personalized PageRank for Classification on Graphs. In *International Conference on Learning Representations*.
[17] Daniele Grattarola and Cesare Alippi. 2021. Graph neural networks in tensorflow and keras with spektral [application notes]. *IEEE Computational Intelligence Magazine* 16, 1 (2021), 99–106.
[18] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 855–864.
[19] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).
[20] Hui Han, Tianyu Zhao, Cheng Yang, Hongyi Zhang, Yaoqi Liu, Xiao Wang, and Chuan Shi. 2022. OpenHGNN: An Open Source Toolkit for Heterogeneous Graph Neural Network. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. 3993–3997.
[21] Kaveh Hassani and Amir Hosein Khasahmadi. 2020. Contrastive multi-view representation learning on graphs. In *International conference on machine learning*. PMLR, 4116–4126.
[22] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, and Meng Wang. 2020. Lightgcn: Simplifying and powering graph convolution network for recommendation. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*. 639–648.

[23] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[24] Jun Hu, Shengsheng Qian, Quan Fang, Youze Wang, Quan Zhao, Huaiwen Zhang, and Changsheng Xu. 2021. Efficient graph deep learning in tensorflow with tf_geometric. In *Proceedings of the 29th ACM International Conference on Multimedia*. 3775–3778.

[25] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. 2020. Heterogeneous graph transformer. In *Proceedings of the web conference 2020*. 2704–2710.

[26] Houye Ji, Xiao Wang, Chuan Shi, Bai Wang, and S Yu Philip. 2021. Heterogeneous graph propagation network. *IEEE Transactions on Knowledge and Data Engineering* 35, 1 (2021), 521–532.

[27] Ming Jin, Yizhen Zheng, Yuan-Fang Li, Chen Gong, Chuan Zhou, and Shirui Pan. 2021. Multi-Scale Contrastive Siamese Networks for Self-Supervised Graph Representation Learning. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, Zhi-Hua Zhou (Ed.). ijcai.org, 1477–1483.

[28] Thomas N Kipf and Max Welling. 2016. Variational Graph Auto-Encoders. *NIPS Workshop on Bayesian Deep Learning* (2016).

[29] Thomas N. Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. (2017).

[30] Cheng Lai, Jiarong Han, and Hao Dong. 2021. TensorLayer 3.0: A Deep Learning Library Compatible With Multiple Backends. In *2021 IEEE International Conference on Multimedia & Expo Workshops (ICMEW)*. IEEE, 1–3.

[31] Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. 2018. Learning deep generative models of graphs. *arXiv preprint arXiv:1803.03324* (2018).

[32] Qingsong Lv, Ming Ding, Qiang Liu, Yuxiang Chen, Wenzheng Feng, Siming He, Chang Zhou, Jianguo Jiang, Yuxiao Dong, and Jie Tang. 2021. Are we really making much progress? revisiting, benchmarking and refining heterogeneous graph neural networks. In *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining*. 1150–1160.

[33] Yanjun Ma, Dianhai Yu, Tian Wu, and Haifeng Wang. 2019. PaddlePaddle: An open-source deep learning platform from industrial practice. *Frontiers of Data and Domputing* 1, 1 (2019), 105–115.

[34] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[35] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 701–710.

[36] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. 2020. DropEdge: Towards Deep Graph Convolutional Networks on Node Classification. In *ICLR*. https://openreview.net/forum?id=Hkx1qkrKPr

[37] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *The Semantic Web: 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, Proceedings 15*. Springer, 593–607.

[38] Fan-Yun Sun, Jordan Hoffmann, Vikas Verma, and Jian Tang. 2020. Infograph: Unsupervised and semi-supervised graph-level representation learning via mutual information maximization. (2020).

[39] Kiran K Thekumparampil, Chong Wang, Sewoong Oh, and Li-Jia Li. 2018. Attention-based graph neural network for semi-supervised learning. *arXiv preprint arXiv:1803.03735* (2018).

[40] Shikhar Vashishth, Soumya Sanyal, Vikram Nitin, and Partha Talukdar. 2020. Composition-based Multi-Relational Graph Convolutional Networks. In *International Conference on Learning Representations*.

[41] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2018. Graph attention networks. (2018).

[42] Petar Velickovic, William Fedus, William L Hamilton, Pietro Liò, Yoshua Bengio, and R Devon Hjelm. 2019. Deep graph infomax. *ICLR (Poster)* 2, 3 (2019), 4.

[43] Hongwei Wang, Jia Wang, Jialin Wang, Miao Zhao, Weinan Zhang, Fuzheng Zhang, Xing Xie, and Minyi Guo. 2018. Graphgan: Graph representation learning with generative adversarial nets. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 32.

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

[45] Ruijia Wang, Shuai Mou, Xiao Wang, Wanpeng Xiao, Qi Ju, Chuan Shi, and Xing Xie. 2021. Graph structure estimation neural networks. In *Proceedings of the Web Conference 2021*. 342–353.

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

[47] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E Sarma, Michael M Bronstein, and Justin M Solomon. 2019. Dynamic graph cnn for learning on point clouds. *Acm Transactions On Graphics (tog)* 38, 5 (2019), 1–12.

[48] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. 2019. Simplifying graph convolutional networks. In *International conference on machine learning*. PMLR, 6861–6871.

[49] Zhiqiang Xie, Minjie Wang, Zihao Ye, Zheng Zhang, and Rui Fan. 2022. Graphiler: Optimizing Graph Neural Networks with Message Passing Data Flow Graph. *Proceedings of Machine Learning and Systems* 4 (2022), 515–528.

[50] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How powerful are graph neural networks? (2019).

[51] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. 2018. Representation learning on graphs with jumping knowledge networks. In *International conference on machine learning*. PMLR, 5453–5462.

[52] Muhan Zhang and Yixin Chen. 2018. Link prediction based on graph neural networks. *Advances in neural information processing systems* 31 (2018).

[53] Tianyu Zhao, Cheng Yang, Yibo Li, Quan Gan, Zhenyi Wang, Fengqi Liang, Huan Zhao, Yingxia Shao, Xiao Wang, and Chuan Shi. 2022. Space4hgnn: a novel, modularized and reproducible platform to evaluate heterogeneous graph neural network. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2776–2789.

[54] Wayne Xin Zhao, Yupeng Hou, Xingyu Pan, Chen Yang, Zeyu Zhang, Zihan Lin, Jingsen Zhang, Shuqing Bian, Jiakai Tang, Wenqi Sun, Yushuo Chen, Lanling Xu, Gaowei Zhang, Zhen Tian, Changxin Tian, Shanlei Mu, Xinyan Fan, Xu Chen, and Ji-Rong Wen. 2022. RecBole 2.0: Towards a More Up-to-Date Recommendation Library. In *CIKM*.

[55] Wayne Xin Zhao, Shanlei Mu, Yupeng Hou, Zihan Lin, Yushuo Chen, Xingyu Pan, Kaiyuan Li, Yujie Lu, Hui Wang, Changxin Tian, Yingqian Min, Zhichao Feng, Xinyan Fan, Xu Chen, Pengfei Wang, Wendi Ji, Yaliang Li, Xiaoling Wang, and Ji-Rong Wen. 2021. RecBole: Towards a Unified, Comprehensive and Efficient Framework for Recommendation Algorithms. In *CIKM*. ACM, 4653–4664.

[56] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: a comprehensive graph neural network platform. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2094–2105.

[57] Yanqiao Zhu, Yichen Xu, Feng Yu, Qiang Liu, Shu Wu, and Liang Wang. 2020. Deep graph contrastive representation learning. *arXiv preprint arXiv:2006.04131* (2020).

[58] Zhaocheng Zhu, Chence Shi, Zuobai Zhang, Shengchao Liu, Minghao Xu, Xinyu Yuan, Yangtian Zhang, Junkun Chen, Huiyu Cai, Jiarui Lu, et al. 2022. Torchdrug: A powerful and flexible machine learning platform for drug discovery. *arXiv preprint arXiv:2202.08320* (2022).

[59] Yuanxin Zhuang, Lingjuan Lyu, Chuan Shi, Carl Yang, and Lichao Sun. 2022. Data-Free Adversarial Knowledge Distillation for Graph Neural Networks. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, Lud De Raedt (Ed.). International Joint Conferences on Artificial Intelligence Organization, 2441–2447. https://doi.org/10.24963/ijcai.2022/339 Main Track.