Chapter 11
Platforms and Practice of Heterogeneous Graph Representation Learning

Abstract It is challenging to build a Heterogeneous Graph (HG) representation learning model because HG is heterogeneous, irregular, and sparse. An easy-to-use and friendly framework is important for a beginner to make an understanding and get deep into this field. In this chapter, we are going to introduce OpenHGNN, a toolkit that can help to build HG models in a predesigned pipeline. And we will present the procedures with three well-known HG models that are HAN, the model first introduces attention mechanism to heterogeneous graph neural networks, RGCN, a model used to model multi-relational graphs with GCN, and HERec, a heterogeneous graph embedding method for recommendation.

11.1 Introduction

Graph is a kind of irregular structure data. Compared to regular structure data like images, graph cannot be directly computed by traditional deep learning platforms (e.g. TensorFlow [1], PyTorch [6], etc.). And graph is usually very large scale and highly sparse, which also increases the difficulty to apply these frameworks. To tackle these problems, some libraries (e.g. DGL [9], PyG [3], etc.) have been developed to support tensor computation over graphs with the principle of message passing followed by most Graph Neural Network (GNN). These frameworks greatly facilitate the implementation of GNN for engineers and researchers.

However, because a Heterogeneous Graph (HG) can be more complex than a homogeneous graph and Heterogeneous Graph Neural Networks (HGNN) are designed with different paradigms, most of the existing graph learning libraries are not friendly enough to implement a HGNN or even don't support HG. Here, we are going to introduce OpenHGNN[1], an open-source toolkit for HGNN based on DGL and PyTorch, developed by BUPT GAMMA Lab[2]. OpenHGNN is specifically

[1] https://github.com/BUPT-GAMMA/OpenHGNN

[2] https://github.com/BUPT-GAMMA

designed for heterogeneous graph representation learning, integrating many popular models and datasets, and it is easy-to-use, extensible, and efficient.

In this chapter, we will show how to build a HGNN in practice based on OpenHGNN. We will first give a brief introduction to the mainstream deep learning platforms and graph leaning libraries. Then we introduce the pipeline of developing on OpenHGNN, and the following part is three heterogeneous graph neural network model instances (HAN [10], HERec [8], RGCN [7]) that developed on OpenHGNN.

11.2 Foundation Platforms

The platforms to be introduced below have three types: deep learning platforms, platforms of graph machine learning, and platforms of heterogeneous graph representation learning. These three types of platforms usually have a hierarchical relation, i.e., platforms of graph machine learning are built upon deep learning platforms and the platforms of heterogeneous graph representation learning are built upon platforms of graph machine learning. From the first to the third type of platforms, their application scenarios are more and more focused, and they are more and more friendly and convenient for developing models of heterogeneous graphs.

11.2.1 Deep Learning Platforms

Deep learning platforms are collections of software which abstracts the underlying hardware and software stacks to expose simple APIs to deep learning developers. They usually support GPU devices and have efficiency optimizations on common computation operations like matrix multiplication, which significantly accelerate the computation speed of machine learning programs. Platforms of graph machine learning are also usually built based on these deep learning platforms.

11.2.1.1 TensorFlow

TensorFlow [1] is an end-to-end open-source machine learning platform. It was originally developed by the Google Brain team in Google's Machine Intelligence Research organization to conduct machine learning and deep neural networks research. It is now general enough to be applicable in a wide variety of other domains. It is one of the earliest platforms and is still thriving nowadays.

TensorFlow has a comprehensive, flexible ecosystem of tools, libraries, and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML-powered applications. And it provides stable Python and C++ APIs, as well as non-guaranteed backward compatible API for other languages

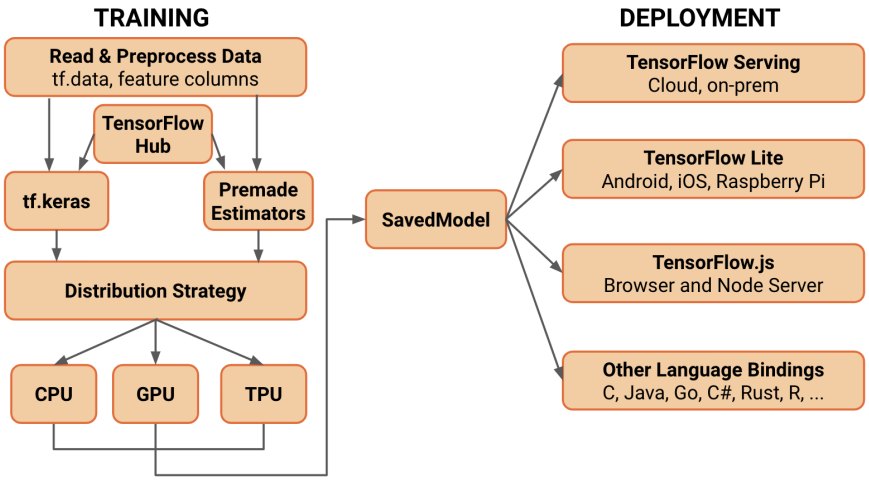

Fig. 11.1 TensorFlow Architecture

like Java and JavaScript. The modules of TensorFlow can be classified into modules for training and modules for deployment, which can be seen in Fig. 11.1.

Beginners of deep learning platforms may be confused by the concept of the graph mode in TensorFlow. Actually, many other deep learning platforms also have similar concepts and hence it is worth elaborating. The graphs, also known as computational graphs, are kinds of data structure and they can be saved, executed, and restored without the original Python code. Graph execution enables portability outside Python and tends to offer better performance, while you can also run a TensorFlow program *eagerly*, i.e., operation by operation like a normal python program. The eager mode has high flexibility and is easy to learn while graph mode is more efficient.

TensorFlow has the following features:

- **Easy Model Building**. TensorFlow offers multiple levels of abstraction so you can choose the right one according to your needs. Building and training models by using the high-level Keras API can help beginners of TensorFlow and machine learning. For large ML training tasks, TensorFlow also provides the Distribution Strategy API for distributed training on different hardware configurations without changing the model definition.
- **Robust ML Production**. TensorFlow also provides a direct and robust path to production. Whether on servers, edge devices, or the web, training and deploying models using TensorFlow can be easy, no matter what language or platform you use. If you require a full production ML pipeline, TensorFlow Extended (TFX) is needed. For running inference on mobile and edge devices, use TensorFlow Lite. For training and deploying models in JavaScript environments, use TensorFlow.js.
- **Powerful Experimentation for Research**. As a frequently-used platform in the research field, building and training state-of-the-art models without sacrificing

speed or performance is a matter of course. TensorFlow provides the flexibility and control with features like the Keras Functional API and Model Subclassing API for creation of complex topologies. TensorFlow also supports an ecosystem of powerful add-on libraries and models to experiment with, including Ragged Tensors, TensorFlow Probability, Tensor2Tensor, and BERT.

11.2.1.2 PyTorch

PyTorch [6] is an open-source machine learning framework based on the Torch library, primarily developed by Facebook's AI Research lab (FAIR). While the initial release of PyTorch is about one year later than TensorFlow, there is a rapidly rising trend of its users in recent years.

PyTorch designs tensor computation (like NumPy) with strong GPU acceleration. Hence it can be utilized as a replacement for NumPy to use the power of GPUs. Besides, it uses a technique called reverse-mode auto-differentiation, which allows users to change the way network behaves arbitrarily with zero lag or overhead. Based on these characteristics, it has grown as a deep learning research platform that provides both flexibility and speed.

PyTorch and TensorFlow are usually considered the two most famous deep learning platforms. They may be very different at the beginning while they have similar features later. For example, in TensorFlow1.x, there was no eager mode which made debugging on TensorFlow codes very difficult. Nowadays, both PyTorch and TensorFlow have graph mode and eager mode.

PyTorch has the following features:

- **Production Ready**. With TorchScript, PyTorch provides ease-of-use and flexibility in eager mode, while seamlessly transitioning to graph mode for speed, optimization, and functionality in C++ runtime environments. For deploying PyTorch models at scale, TorchServe is an easy-to-use tool. It is cloud and environment agnostic and supports features such as multi-model serving, logging, metrics and the creation of RESTful endpoints for application integration. PyTorch also supports asynchronous execution of collective operations and peer-to-peer communication accessible from both Python and C++, which are helpful for distributed training.
- **Robust Ecosystem**. An active community of researchers and developers have built a rich ecosystem of tools and libraries, which can extend PyTorch and support development in nearly all machine learning fields from computer vision to reinforcement learning. PyTorch also supports exporting models in the standard ONNX (Open Neural Network Exchange) format for direct access to ONNX-compatible platforms, runtimes, visualizers, etc. Furthermore, PyTorch is well supported on mainstream cloud platforms, providing frictionless development and easy scaling through prebuilt images, large-scale training on GPUs, ability to run models in a production scale environment, etc.
- **C++ Frontend**. Except Python, PyTorch also provides a C++ frontend. It is a pure C++ interface that follows the design and architecture of the Python frontend. It

is intended to enable research in high performance, low latency, and bare metal C++ applications.

11.2.1.3 MXNet

Apache MXNet [2] is an open-source deep learning software framework, developed by Apache software foundation. Unlike the rest three deep learning platforms, MXNet does not have a background of a single company or institution.

MXNet is designed for both efficiency and flexibility, which is similar to the design philosophy of PyTorch. It allows users to mix symbolic and imperative programming to maximize both efficiency and productivity. At its core, MXNet contains a dynamic dependency scheduler that automatically parallelizes both symbolic and imperative operations on the fly. A graph optimization layer on top of that makes symbolic execution fast and memory efficient.

MXNet has the following features:

- **Hybrid Frontend** MXNet provides hybridize functionality to switch to symbolic mode from imperative mode simply.
- **Distributed Training**. MXNet supports multi-gpu or multi-host training with near-linear scaling efficiency. MXNet recently introduced support for Horovod, the distributed learning framework developed by Uber.
- **8 Language Bindings**. MXNet supports 8 languages. It is integrated into Python deeply and supports Scala, Julia, Clojure, Java, C++, R, and Perl. Combined with the hybridization feature, this allows a very smooth transition from Python training to deployment in the language of your choice to shorten the time to production.

11.2.1.4 PaddlePaddle

PaddlePaddle[3] is an open-source deep learning platform which is derived from industry practice. It is developed by Baidu based on its research and industrial application experience of deep learning technologies.

A prominent characteristic of PaddlePaddle is that it has a close relation with the industry field. For example, there are abundant open-source algorithms, especially pre-trained models integrated into the platform, which can accelerate the application to industrial scenarios.

PaddlePaddle has the following features:

- **Convenient Development**. PaddlePaddle also supports both declarative and imperative programming. The network structures can be designed automatically to some degree, and even in some scenarios, the auto-designed model can outperform human experts.

[3] https://paddlepaddle.org.cn

- **Large-Scale Training**. PaddlePaddle supports hundreds of billions of features and trillions of parameters, which is necessary for industrial development.
- **Deployment on Multiple Terminals**. PaddlePaddle is compatible with models trained by multiple open-source frameworks and can be deployed to multiple kinds of terminals with high inference speed.

11.2.2 Platforms of Graph Machine Learning

In the early years, without platforms of graph machine learning, developers and researchers had to design graph machine learning models on deep learning platforms mentioned above. However, there are significant semantic gaps between the tensor-centric perspective of the platforms above and that of a graph. The performance gaps between the computation and memory-access patterns, which are induced by the sparse nature of graphs and the underlying parallel hardware optimized for dense tensor operations, also exist. Hence platforms of graph machine learning thrives in recent years to abstract and integrate these to provide a succinct programming interface.

11.2.2.1 DGL

DGL [9], Deep Graph Library, is an easy-to-use, high-performance, and scalable Python package for deep learning on graphs. It is built for easy implementation of graph neural network model family, on the top of existing deep learning frameworks. The main sponsors of DGL include AWS, NSF, NVIDIA, and Intel.

DGL is framework agnostic, meaning if a deep graph model is a component of an end-to-end application, the rest of the logics can be implemented in any major frameworks, such as PyTorch, Apache MXNet or TensorFlow. DGL offers a versatile control of message passing, speed optimization via auto-batching and highly tuned sparse matrix kernels, and multi-GPU/CPU training to scale to graphs of hundreds of millions of nodes and edges.

DGL has the following features:

- **Framework Agnostic**. The backend platform of DGL can be anyone of PyTorch, TensorFlow, and MXNet. Users can choose their familiar deep learning platforms and get used to DGL in a short time.
- **Supporting GPU Well** DGL provides a powerful graph class that can reside on either CPU or GPU. It bundles structural data as well as features for a better control. DGL provides a variety of functions for computing with graph objects including efficient and customizable message passing primitives for graph neural networks.
- **Models, Modules, and Benchmarks** DGL collects a rich set of example implementations of popular GNN models of a wide range of topics. Researchers can search for related models to innovate new ideas from or use them as baselines

for experiments. Moreover, DGL provides many state-of-the-art GNN layers and modules for users to build new model architectures based on them.
- **Scalable and Efficient** DGL distills the computational patterns of GNNs into a few generalized sparse tensor operations suitable for extensive parallelization. It also extensively optimizes the whole stack to reduce the overhead in communication, memory consumption, and synchronization. As a result, it is convenient to train models using DGL on large-scale graphs across multiple GPUs or multiple machines and relatively easy to scale to billion-sized graphs.

11.2.2.2 PyG

PyG [3], PyTorch Geometric, is a library for deep learning on irregularly structured input data such as graphs, point clouds, and manifolds built upon PyTorch.

PyG contains various methods for deep learning on graphs and other irregular structures, also known as geometric deep learning, from a variety of published papers. It also contains easy-to-use mini-batch loaders for operating on many small and single giant graphs, multi-GPU support, a large number of common benchmark datasets, the GraphGym experiment manager, and helpful transforms for learning on arbitrary graphs, 3D meshes, or point clouds. Unlike DGL, PyG can only use PyTorch as its backend.

PyG has the following features:

- **Friendly to PyTorch Users** PyG is especially friendly to PyTorch users. It utilizes a tensor-centric API and keeps design principles close to vanilla PyTorch.
- **Comprehensive and Well-maintained GNN Models** PyG also contains Comprehensive and well-maintained GNN models. Most of the state-of-the-art Graph Neural Network architectures have been implemented by library developers or authors of research papers.
- **High Flexibility** Existing PyG models can easily be extended for conducting your own research with GNNs. Making modifications to existing models or creating new architectures is simple, thanks to its easy-to-use message passing API, and a variety of operators and utility functions.
- **GraphGym Integration** GraphGym lets users easily reproduce GNN experiments, and is able to launch and analyze thousands of different GNN configurations. It is also customizable by registering new modules to a GNN learning pipeline.

11.2.3 Platforms of Heterogeneous Graph Representation Learning

It may seem that platforms of graph machine learning are strong enough after having a tour on them, while heterogeneous graph representation learning has its unique characteristics. To handle heterogeneous node types and relation types and mine the

abundant semantic information behind them, heterogeneous graph models are usually more complicated than usual graph models, while many mainstream heterogeneous models are not integrated into the platforms of graph machine learning mentioned above. Furthermore, the training procedures of heterogeneous models are also usually more complicated, which may be an obstacle for beginners to comprehend.

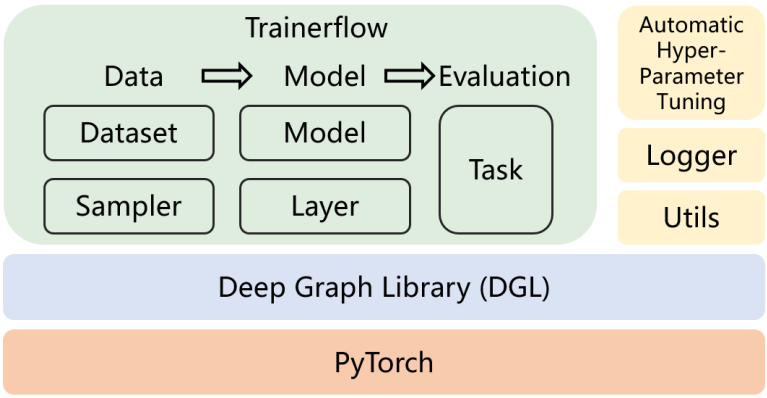

Fig. 11.2 OpenHGNN Architecture

OpenHGNN[4] is an open-source toolkit for heterogeneous graph neural networks based on DGL and PyTorch, developed by BUPT GAMMA Lab.

The main modules of OpenHGNN are dataset, model, sampler, layer and task, which are shown in Fig. 11.2. The functions and relations of these modules will be elaborated on in the next section.

OpenHGNN is helpful to its users in every aspect in the field of heterogeneous graph models. Reading the well-organized model codes provided by the library makes users more clear to details of models, comprehending the trainerflow architectures makes users more familiar with heterogeneous graph tasks and modifying on these models makes users build their new models conveniently and flexibly. OpenHGNN is also convenient for researchers to test model performances as baselines in one command. Here is an example:

```
python main.py -m GTN -d imdb4GTN -t node_classification -g 0
```

OpenHGNN has the following features:

- **Various Models** OpenHGNN provides many open-source mainstream heterogeneous models, such as HAN, HetGNN [12] and GTN [11]. It is more convenient to learn heterogeneous models by virtue of well-organized and well-commented model codes. Modification on these existing models are also easy.

[4] https://github.com/BUPT-GAMMA/OpenHGNN

- **High Extensibility** OpenHGNN is highly extensible. It unifies all these models into a single framework composed of tasks and models, which will be elaborated on later. Datasets, trainerflows and models are decoupled well, and therefore users can customize one of them without affecting others. For example, if a user wants to implement his own model, he does not have to implement the trainer flow and only needs to implement the bare model class.
- **High Efficiency** The efficiency of OpenHGNN is guaranteed by the efficient APIs of the backend DGL.

11.3 Practice of Heterogeneous Graph Representation Learning

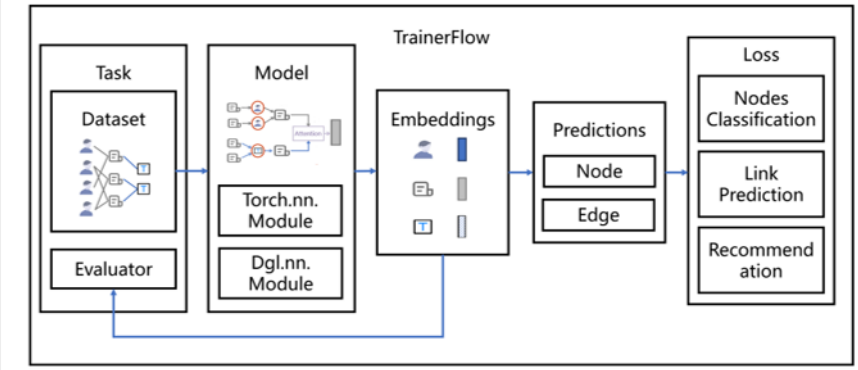

Fig. 11.3 Pipline of OpenHGNN

In this part, we will show how to build a heterogeneous graph learning model based on OpenHGNN in practice. OpenHGNN mainly contains three parts, **trainerflow**, **model** and **task**. The relations between them are shown in Fig 11.3. The trainerflow, containing the model and the task, is an abstraction of a predesigned workflow that trains and evaluates a model on a given dataset for a specific use case. The model plays the role of an encoder and it will output the node embedding for a given heterogeneous graph. The task makes a connection between the embeddings and downstream tasks.

In the next part of the section, we will show how to build a new model from scratch with source codes. The rest of this section is organized in the following order:

- **Construct a dataset**. This part will introduce how to construct a new dataset OpenHGNN not integrated. We will first introduce how to create a `dgl.heterograph` object and then how to integrate it to OpenHGNN.

- **Build a trainerflow**. Not all the models follow a universal training procedure, especially for HGNN. In this part, we will show how to implement a model and how to build a new trainerflow for it if needed.
- **Examples in practice**. The last part is the implementation of three models (i.e. HAN [10], RGCN [7], HERec [8]).

11.3.1 Build a New Dataset

In order to better introduce the pipeline and practice of OpenHGNN, we first introduce the data structure and data processing in OpenHGNN. We have processed a number of heterogeneous graph dataset in advance and integrated into the OpenHGNN. Several types of datasets including academic networks (e.g., ACM, DBLP, Aminer), information networks (e.g., IMDB, LastFM), and recommendation graphs (e.g., Amazon, MovieLens, Yelp) are the most wildly-used datasets which could be specified directly when running experiments with the command line.

Before implementing a model or testing an existing model performance, we need to check if the dataset is supported by OpenHGNN. If not, what we should do first is to construct a new dataset. In this section, we use the ACM dataset [10] for node classification as an example to show the procedures.

First we will show an example about how to build a node classification dataset. In DGL, a HG is specified with a series of relation subgraphs as below. Each relation name is a string `(source node type, edge type, destination node type)` associated with a relation subgraph. The following code snippet is an example of creating a heterogeneous graph in DGL.

```
>>> import dgl
>>> import torch as th

>>> # Create a heterograph with 3 node types and 3 edge types.
>>> graph_data = {
...     ('drug','interacts','drug'): (th.tensor([0,1]), th.tensor([1,2])),
...     ('drug','interacts','gene'): (th.tensor([0,1]), th.tensor([2,3])),
...     ('drug','treats','disease'): (th.tensor([1]), th.tensor([2]))
... }
>>> g = dgl.heterograph(graph_data)
>>> g.ntypes
['disease', 'drug', 'gene']
>>> g.etypes
['interacts', 'interacts', 'treats']
>>> g.canonical_etypes
[('drug', 'interacts', 'drug'),
 ('drug', 'interacts', 'gene'),
 ('drug', 'treats', 'disease')]
```

We recommend to set the feature name as 'h'.

```
>>> g.nodes['drug'].data['h'] = th.ones(3, 1)
```

DGL provides `dgl.save_graphs` and `dgl.load_graphs` respectively for saving and loading heterogeneous graphs in binary format. So we can use `dgl.save_graphs` to store graphs into the disk.

```
>>> dgl.save_graphs("demo_graph.bin", g)
```

Now a binary file containing the dataset has been created, and we should move it to the directory *openhgnn/dataset/*. In *NodeClassificationDataset.py*, the dataset will be loaded and some extra information will also be needed. For example, the `category`, `num_classes` and `multi_label` (if necessary) should be set with "drug", 3, True, representing the node type to predict, the number of classes, and whether the task is multi-label classification respectively.

```
>>> if name_dataset == 'demo_graph':
...     data_path = './openhgnn/dataset/demo_graph.bin'
...     g, _ = load_graphs(data_path)
...     g = g[0].long()
...     self.category = 'drug'
...     self.num_classes = 3
...     self.multi_label = False
```

We have described how to construct a heterogeneous graph in DGL. In the following part, we will introduce how to build the ACM dataset. We will first provide the description of the dataset and then we will show how to process the original file to DGL heterogeneous graph data structure.

The ACM dataset is a citation network that is commonly used on HGNN. There are four node types and eight link types in the original ACM dataset. Node types contain **paper**, **author**, **subject** and **term**. Each node has a unique ID and a type ID used to identify the node type. Taking **paper** as an example, it contains a unique ID, the type ID of the node(e.g. 0 for paper) and the feature of this node. In the original txt file, each row represents a node and its format is as follows:

```
0    'Influence and correlation...' 0   "1,1,1..."
1    'Efficient semi-streaming...'  0   "0,1,0..."
...
3025 'IMohammad Mahdian'            1   "1,1,1..."
3026 'Ravi Kumar'                   1   "1,1,1..."
...
```

The first column is node ID, the second column is node name, the third column is node type ID and the last column is node feature and the feature dimensions of nodes with the same type are consistent. The meta information of node type are as follows:

```
{
    "node type":
    {
        "0": "paper",
        "1": "author",
        "2": "subject",
        "3": "term"
    }
}
```


Note that node IDs are continuous. Taking **paper-cite-paper** as an edge example, it contains the source node ID, the target node ID, the edge type ID (e.g. 0 for paper-cite-paper and 7 for term-paper) and the weight of each edge (In ACM dataset, all weights are 1.0). In the original file, each row represents an edge and its format is as follows:

```
0 179  0 1.0
0 2697 0 1.0
1 2523 0 1.0
1 2589 0 1.0
...
```

For example, the first column means an edge from node 0 to node 179, the edge type ID is 0 and the edge weight is 1.0. The meta information of edge type are as follows:


```
{
    "link type": {
        "0": {
            "start": 0,
            "end": 0,
            "meaning": "paper-cite-paper"
        },
        ...
        "7": {
            "start": 3,
            "end": 0,
            "meaning": "term-paper"
        }
    }
}
```


Finally, there is a file about node label. Each row represents the target node ID, the node type and its category, and the format is as follows:

```
622  0 2
1923 0 0
951  0 0
1957 0 1
```

Note that only the target node has category (paper is the target node type here), so each element in the second column has the same value 0. The meta information of node category are as follows:

```
{
    "node type": {
        "0": {
            "0": "database",
            "1": "wireless communication",
            "2": "data mining"
        }
    }
}
```

For example, the paper node 622 belongs to the data mining field and the paper node 1923 is published in the database field.

In the next part, we first show how to generate a dictionary which can be transformed into DGL heterogeneous graph data structure. This can be done by the following code:

```
>>> meta_graphs = {}
>>> for i in range(8):
...     edge = edges[edges[2] == i]
...     source_node = edge.iloc[:,0].values -
...                   np.min(edge.iloc[:,0].values)
...     target_node = edge.iloc[:,1].values -
...                   np.min(edge.iloc[:,1].values)
...     meta_graphs[(node_info[str(link_info[str(i)]['start'])],
...                  link_info[str(i)]['meaning'],
...                  node_info[str(link_info[str(i)]['end'])])]
...         = (torch.tensor(source_node), torch.tensor(target_node))
```

Then a heterogeneous graph object can be constructed with function `dgl.heterograph`:

```
>>> g = dgl.heterograph(meta_graphs)
```

The following code can store node features directly on the graph:

```
>>> g.nodes['paper'].data['h'] = torch.FloatTensor(paper_feature)
>>> g.nodes['author'].data['h'] = torch.FloatTensor(author_feature)
>>> g.nodes['subject'].data['h'] = torch.FloatTensor(subject_feature)
>>> dgl.save_graphs("acm.bin", g)
```

In the end, we can output the specific details of the processed graph through the following code:

```
>>> import dgl
>>> g = dgl.load_graphs('acm.bin')
>>> print(g)
Graph(num_nodes={'author':5959, 'paper':3025, 'subject':56, 'term':1902},
      num_edges={('author', 'author-paper', 'paper'): 9949,
                 ('paper', 'paper-author', 'author'): 9949,
                 ('paper', 'paper-cite-paper', 'paper'): 5343,
                 ('paper', 'paper-ref-paper', 'paper'): 5343,
                 ('paper', 'paper-subject', 'subject'): 3025,
                 ('paper', 'paper-term', 'term'): 255619,
                 ('subject', 'subject-paper', 'paper'): 3025,
                 ('term', 'term-paper', 'paper'): 255619},
      metagraph=[('author', 'paper', 'author-paper'),
                 ('paper', 'author', 'paper-author'),
                 ('paper', 'paper', 'paper-cite-paper'),
                 ('paper', 'paper', 'paper-ref-paper'),
                 ('paper', 'subject', 'paper-subject'),
                 ('paper', 'term', 'paper-term'),
                 ('subject', 'paper', 'subject-paper'),
                 ('term', 'paper', 'term-paper')])
```

By now, the binary graph file you stored can be used directly by OpenHGNN to train your model.

11.3.2 Build a New Model

11.3.2.1 Model

The model acts as a graph encoder. Given a heterogeneous graph and node features, the model will return a dictionary of node embeddings. It also allows to output the embedding of target nodes which are participated in loss calculation.

The Model mainly consists of two parts, a model builder and a forward propagator. Each model inherits the BaseModel class, and must implement the classmethod `build_model_from_args`. With that, two parameters named **args** and **hg** can be used to build up a custom model with model-specific hyper-parameters. So it is necessary to implement the function `build_model_from_args` in the model. Here is an example:

```
class RGAT(BaseModel):
    @classmethod def build_model_from_args(cls, args, hg):
    return cls(in_dim=args.hidden_dim,
                out_dim=args.hidden_dim,
                h_dim=args.out_dim,
```

```
                etypes=hg.etypes,
                num_heads=args.num_heads,
                dropout=args.dropout)
```

The function `forward` is the core of the model. It is noted that in OpenHGNN, we prepossess the feature of dataset outside of model. Specifically, we use a linear layer with a bias for each node type to map all node features to a shared feature space. So the parameter **h_dict** of `forward` in model is not original, and the model need not feature prepossessing.

11.3.2.2 Trainerflow

The trainerflow, containing the model and the task, is an abstraction of a predesigned-workflow that trains and evaluates a model on a given dataset for specific utilization.

As shown in Fig. 11.3, the trainerflow contains the whole training process. A Trainerflow object includes **task**, **model**, **optimizer** and **dataloader** in order to execute the training process. The main training process is done by function `train` and the single training step is done by `_full_train_step` and its mini-batch version `_mini_train_step`. After one or several training steps, the function `_test_step` can be applied to test the validation performance of model to select the best model parameters.

There are three main trainers named **node_classification_flow** which is designed for a semi-supervised node classification model, **link_prediction** which is designed link-prediction task and **recommendation** for recommendation scenario.

Some models (e.g. HetGNN [12], HeGAN [4] etc.) demanding special training processes has their own trainerflow. Generally, if the paper has a fancy loss function or an accelerated sampling method, it needs a customized trainerflow.

The following demo may give a sight of how a trainerflow organized and worked in OpenHGNN.

```
class Trainerflow(BaseFlow):
    def train():
        # _full_train_step()
        # _test_step()
    def _full_train_step():
        # output = model(input)
        # loss = loss_fn(output, label)
        # loss.backward()
        # optimizer.step()
    def _test_step():
        # task.evaluate()
```

11.3.2.3 Task

The Task module encapsulates several objects related to the task, including **Dataset**, **Evaluator**, **Labels** and **dataset_split**. The dataset contains the heterogeneous graph as DGLGraph, as well as node/edge features and additional dataset-specific information. The evaluator takes the predictions of the model and the labels to give the evaluation results and this can be done by the function `evaluate`. The function `get_graph` will return the DGLGraph, and the function `get_labels` will return the ground truth labels. There are three kinds of supported tasks in the OpenHGNN, including **Node classification**, **Link prediction** and **Recommendation**.

11.3.2.4 Register in OpenHGNN

Before training the model in OpenHGNN, what we should do first is to register the model with `@register_model(New_model)`. Here is an example:

```
from openhgnn.models import BaseModel, register_model
@register_model('New_model')
class New_model(BaseModel):
    # Implementation details
```

The training process of models differs from each other, which needs a model trainer that can be applied. Similar to the implementation of a model, we need to register the model trainer with `@register_trainer("New_trainer")`. Here is an example:

```
from openhgnn.trainerflow import BaseFlow, register_flow
@register_flow('New_trainer')
class New_trainer(BaseFlow):
    # Implementation details
```

For more details about the developing process based on OpenHGNN, please refer to the official documents[5]. In the following of the section, we will introduce several examples on how to implement a model based on OpenHGNN.

11.3.3 Practice of HAN

HAN [10] firstly proposes a heterogeneous graph neural network based on the hierarchical attention, including node-level and semantic-level attentions. It enables the graph neural network to be directly applied to the heterogeneous graph, and further facilitates the heterogeneous graph based applications.

[5] https://openhgnn.readthedocs.io/en/latest/

The node-level attention aims to learn the importance between a node and its meta-path based neighbors. The mathematical formulas are as follows:

$$\alpha_{ij}^{\Phi} = \text{softmax}_j\left(e_{ij}^{\Phi}\right) = \frac{\exp\left(\sigma\left(a_{\Phi}^{\mathrm{T}} \cdot \left[h_i' \| h_j'\right]\right)\right)}{\sum_{k \in \mathcal{N}_i^{\Phi}} \exp\left(\sigma\left(a_{\Phi}^{\mathrm{T}} \cdot \left[h_i' \| h_k'\right]\right)\right)},$$

$$z_i^{\Phi} = \prod_{k=1}^{K} \sigma\left(\sum_{j \in \mathcal{N}_i^{\Phi}} \alpha_{ij}^{\Phi} \cdot h_j'\right).$$

The semantic-level attention is able to learn the importance of different meta-paths. The mathematical formula is as follows:

$$Z = \mathcal{F}_{att}(Z^{\Phi_1}, Z^{\Phi_2}, ..., Z^{\Phi_P}).$$

With the importance score learned from both node-level and semantic-level, the proposed model can generate node embedding by aggregating features from neighbors based on meta-path in a hierarchical manner. In the following part, we will show how to use OpenHGNN to implement it.

11.3.3.1 HAN basemodel

To create a model `HAN`, we could imitate the case of 11.3.2. In this section, we will show each functional part of the HAN source code and explain the message passing process over HG and its mathematical definition. We implement the class method `build_model_from_args`, other functions like `__init__`. Here is our code.

```
@register_model('HAN')
class HAN(BaseModel):
    @classmethod
    def build_model_from_args(cls, args, hg):
        if args.meta_paths is None:
            meta_paths = extract_metapaths(args.category,
                                           hg.canonical_etypes)
        else:
            meta_paths = args.meta_paths

        return cls(meta_paths=meta_paths,
                   category=args.category,
                   in_size=args.hidden_dim,
                   hidden_size=args.hidden_dim,
                   out_size=args.out_dim,
                   num_heads=args.num_heads,
                   dropout=args.dropout)
```

```
    def __init__(self, meta_paths, category, in_size, hidden_size,
                 out_size, num_heads, dropout):
        super(HAN, self).__init__()
        self.category = category
        self.layers = nn.ModuleList()
        self.layers.append(HANLayer(meta_paths, in_size, hidden_size,
        num_heads[0], dropout))
        for l in range(1, len(num_heads)):
            self.layers.append(
                HANLayer(meta_paths, hidden_size * num_heads[l-1],
                         hidden_size, num_heads[l], dropout)
            )
        self.linear = nn.Linear(hidden_size * num_heads[-1], out_size)

    def forward(self, g, h_dict):
        h = h_dict[self.category]
        for gnn in self.layers:
            h = gnn(g, h)

        return {self.category: self.linear(h)}
```

This piece of code is the overall structure about HAN model. Function `__init__` will initialize the model and the function `forward` will encoder original features into new features. In general, the model should output all the node embeddings in a form of dictionary. It is allowed to output only the embedding of the target node participating in the loss calculation.

11.3.3.2 HANlayer

HAN is mainly composed of two parts, **Node-level Attention** and **Semantic-level Attention**. The model which we implement in OpenHGNN incorporates these two parts in `HANLayer`.

```
class HANLayer(nn.Module):
    def __init__(self, meta_paths, in_size, out_size,
                 layer_num_heads, dropout):
        super(HANLayer, self).__init__()
        self.meta_paths = meta_paths
        # One GAT layer for each meta path based adjacency matrix
        self.gat_layers = nn.ModuleList()
        semantic_attention = SemanticAttention(in_size=out_size
                                               * layer_num_heads)
        self.model = MetapathConv(
            meta_paths,
```

```
            [GATConv(in_size, out_size, layer_num_heads, dropout,
                     activation=F.elu, allow_zero_in_degree=True)
            for _ in meta_paths],
            semantic_attention
        )
        self._cached_graph = None
        self._cached_coalesced_graph = {}
    def forward(self, g, h):
        if self._cached_graph is None or self._cached_graph is not g:
            self._cached_graph = g
            self._cached_coalesced_graph.clear()
            for meta_path in self.meta_paths:
                self._cached_coalesced_graph[meta_path] =
                dgl.metapath_reachable_graph(g, meta_path)
        h = self.model(self._cached_coalesced_graph, h)

        return h
```

Metapath Subgraph Extraction

We use `dgl.metapath_reachable_graph(g,meta_path)` API encapsulated by DGL to extract homogeneous graph from heterogeneous graph according to meta-path. For example:

```
g = dgl.heterograph({
    ('A', 'AB', 'B'): ([0, 1, 2], [1, 2, 3]),
    ('B', 'BA', 'A'): ([1, 2, 3], [0, 1, 2])})
new_g = dgl.metapath_reachable_graph(g, ['AB', 'BA'])
new_g.edges(order='eid')
# (tensor([0, 1, 2]), tensor([0, 1, 2]))
```

This part of code is the core component of HAN. `HANLayer` is inherited from `nn.Module` and acts as a complete aggregation of HAN. We use `GATConv`, which is implemented by DGL, to implement the node-level attention mechanism. And semantic-level attention is implemented by `MetapathConv`, which we will introduce later.

After the aggregation of node-level attention mechanism, we will get node embeddings based on different meta-paths, as much as the predetermined meta-paths. At the same time, we extend node-level attention to multi-head attention and concatenate the learned embeddings to enhance the representation power of the model. Then based on different meta-paths we perform semantic-level attention fusion to learn different importance weights of each meta-path. The code of `MetapathConv` is as follows:

```
class MetapathConv(nn.Module):
    def __init__(self, meta_paths, mods, macro_func, **kargs):
        super(MetapathConv, self).__init__()
        # One GAT layer for each meta path based adjacency matrix
        self.mods = nn.ModuleList(mods)
        self.meta_paths = meta_paths
        self.SemanticConv = macro_func

    def forward(self, g_list, h):
        for i, meta_path in enumerate(self.meta_paths):
            new_g = g_list[meta_path]
            semantic_embeddings.append(self.mods[i](new_g, h).flatten(1))

        return self.SemanticConv(semantic_embeddings)
```

`self.SemanticConv` is an aggregation function fusing meta-paths and we could get the final embeddings by it.

The code of `SemanticConv` is as follows:

```
class SemanticAttention(nn.Module):
    def __init__(self, in_size, hidden_size=128):
        super(SemanticAttention, self).__init__()

        self.project = nn.Sequential(
            nn.Linear(in_size, hidden_size),
            nn.Tanh(),
            nn.Linear(hidden_size, 1, bias=False)
        )

    def forward(self, z, nty=None):
        if len(z) == 0:
            return None
        z = torch.stack(z, dim=1)
        w = self.project(z).mean(0)                    # (M, 1)
        beta = torch.softmax(w, dim=0)                 # (M, 1)
        beta = beta.expand((z.shape[0],) + beta.shape) # (N, M, 1)

        return (beta * z).sum(1)                       # (N, D * K)
```

After that, we will get the final embedding which aggregates all meta-paths information by Attention mechanism.

11.3.4 Practice of RGCN

RGCN [7] is the first to show that the GCN [5] framework can be applied to modeling relational data, which contributes to the application of neural network in heterogeneous graph. In RGCN, we execute graph convolution on subgraph of every relation to produce subrelation representations. Then we can weight the sum of these subrelation representations include a self-loop representation. The weight is the normalized in-degree of every relation. This is the key of RGCN. We summarize the above as the following formula.

$$h_i^{(l+1)} = \sigma\left(\sum_{r\in\mathcal{R}}\sum_{j\in\mathcal{N}_i^r}\frac{1}{c_{i,r}}W_r^{(l)}h_j^{(l)} + W_0^{(l)}h_i^{(l)}\right).$$

OpenHGNN also uses DGL method which is based on message passing mechanism to implement RGCN layer. In this part, we focus on describing the interface of RGCN and the code of convolution.

Here is the interface of RGCN model in OpenHGNN:

```
def __init__(self, in_dim, h_dim, out_dim, etypes, num_bases,
             num_hidden_layers=1, dropout=0, use_self_loop=False)
```

Next is the code of RelGraphConvLayer:

```
self.conv = dglnn.HeteroGraphConv({
                rel: dglnn.GraphConv(in_feat, out_feat, norm='right',
                weight=False, bias=False)for rel in rel_names
            })
```

The function `HeteroGraphConv` receives a dictionary containing all the subgraph convolution models as a parameter. DGL gives a pseudo-code for implementation of `HeteroGraphConv`:

```
outputs = {nty : [] for nty in g.dsttypes}
# Apply submodules on their associating relation graphs in parallel
for relation in g.canonical_etypes:
    stype, etype, dtype = relation
    dstdata = relation_submodule(g[relation], ...)
    outputs[dtype].append(dstdata)

# Aggregate the results for each destination node type
rsts = {}
for ntype, ntype_outputs in outputs.items():
    if len(ntype_outputs) != 0:
        rsts[ntype] = aggregate(ntype_outputs)

return rsts
```

The code of `self.conv` used in `forward` function:

```
def forward(self, g, inputs):
    g = g.local_var()
    if self.use_weight:
        weight = self.basis() if self.use_basis else self.weight
        wdict = {self.rel_names[i]: {'weight': w.squeeze(0)}
                 for i, w in enumerate(th.split(weight, 1, dim=0))}
    else:
        wdict = {}

    if g.is_block:
        inputs_src = inputs
        inputs_dst = {k: v[:g.number_of_dst_nodes(k)]
                      for k, v in inputs.items()}
    else:
        inputs_src = inputs_dst = inputs

    hs = self.conv(g, inputs_src, mod_kwargs=wdict)
```

The function `forward` receives a parameter `inputs` which contains the input of function `HeteroGraphConv`. The dict `inputs_src` contains node features for each node type and the dict `wdict` is the weight dict of every subrelation which is used in a linear or non-linear transformation.

11.3.5 Practice of HERec

HERec [8] is the first attempt which adopts the network embedding approach to extract useful information from heterogeneous information network and leverage such information for rating prediction. HERec mainly contains two key components: metapath-based random walk and Skip-Gram model. As a practice of HERec, we will focus on the former component metapath-based random walk. As the earliest model, HERec utilizes a similar random walk with meatapth2vec model. And we will introduce how to implement the two ways of random walk.

11.3.5.1 HERec random walk

As mentioned in 11.3.3.2, we use the `dgl.metapath_reachable_graph(g, meta_path)` API to extract metapth subgraph which is a homogeneous graph.

For a homogeneous graph, we can use the random walk applied in the homogeneous graph. `dgl.sampling.random_walk` provides an API to generate random walk traces.

Example of normal random walk, `g1` will generate four traces from the source nodes [0, 1, 2, 0] and each trace has five nodes.

```
>>> g1 = dgl.graph(([0, 1, 1, 2, 3], [1, 2, 3, 0, 0]))
>>> dgl.sampling.random_walk(g1, [0, 1, 2, 0], length=4)
(tensor([[0, 1, 2, 0, 1],
         [1, 3, 0, 1, 3],
         [2, 0, 1, 3, 0],
         [0, 1, 2, 0, 1]]), tensor([0, 0, 0, 0, 0]))
```

Taking ACM as an example, we extract two metapath subgraphs with metapath PAP and PSP. PAP-subgraph can generate traces by random walk.

```
>>> PAP = ['paper-author', 'author-paper']
>>> PSP = ['paper-subject', 'subject-paper']
>>> pap_subgraph = dgl.metapath_reachable_graph(hg, PAP)
Graph(num_nodes=3025, num_edges=29767,
      ndata_schemes={'test_mask': Scheme(shape=(), dtype=torch.uint8),
      'train_mask': Scheme(shape=(), dtype=torch.uint8),
      'label': Scheme(shape=(), dtype=torch.float32),
      'h': Scheme(shape=(1902,), dtype=torch.float32)}
      edata_schemes={})
>>> psp_subgraph = dgl.metapath_reachable_graph(hg, PSP)
Graph(num_nodes=3025, num_edges=2217089,
      ndata_schemes={'test_mask': Scheme(shape=(), dtype=torch.uint8),
      'train_mask': Scheme(shape=(), dtype=torch.uint8),
      'label': Scheme(shape=(), dtype=torch.float32),
      'h': Scheme(shape=(1902,), dtype=torch.float32)}
      edata_schemes={})
>>> pap_traces = dgl.sampling.random_walk(pap_subgraph,
...     torch.arange(pap_subgraph.num_nodes()), length=4)
(tensor([[   0,    0,   20, 1807,  734],
        [   1,  773,    5,    1, 2576],
        [   2, 2519, 2701,  616,  616],
        ...,
        [3022, 1678, 3022,  275,  275],
        [3023, 3023, 3023, 3023, 3023],
        [3024, 3024, 3024, 3024, 3024]]), tensor([0, 0, 0, 0, 0]))
>>> psp_traces = dgl.sampling.random_walk(psp_subgraph,
...     torch.arange(psp_subgraph.num_nodes()), length=4)
(tensor([[   0,   75,   75,  586,  716],
        [   1, 1764, 2512, 1468, 1641],
        [   2,  189, 2786,  737, 2743],
        ...,
        [3022, 2641,  347,  684, 2923],
```

```
        [3023,  722, 2707, 2394, 1380],
        [3024, 1450,  235,  803, 1884]]), tensor([0, 0, 0, 0, 0]))
```

And traces could generate embedding through skipgram model. Finally, we can get the final embedding by fusing the two metapath-types embedding of paper nodes.

11.3.5.2 Metapath random walk

We also generate random walk traces from an array of starting nodes directly based on the given metapath.

```
>>> g2 = dgl.heterograph({
...     ('user','follow','user'): ([0, 1, 1, 2, 3], [1, 2, 3, 0, 0]),
...     ('user','view','item'): ([0, 0, 1, 2, 3, 3], [0, 1, 1, 2, 2, 1]),
...     ('item','viewed-by','user'): ([0, 1, 1, 2, 2, 1],
...                                   [0, 0, 1, 2, 3, 3])
... })
>>> dgl.sampling.random_walk(g2, [0, 1, 2, 0],
...     metapath=['follow', 'view', 'viewed-by'] * 2)
(tensor([[0, 1, 1, 1, 2, 2, 3],
         [1, 3, 1, 1, 2, 2, 2],
         [2, 0, 1, 1, 3, 1, 1],
         [0, 1, 1, 0, 1, 1, 3]]), tensor([0, 0, 1, 0, 0, 1, 0]))
```

Taking ACM as an example too, we will generate traces through metapath PAPAP.

```
>>> dgl.sampling.random_walk(
...     hg, torch.arange(hg.num_nodes('paper')),
...     metapath=['paper-author', 'author-paper']*2)
(tensor([[   0,    1,  731,    1,   20],
        [   1,    3,    1,    5,    1],
        [   2,    7,  229,   12,    4],
        ...,
        [3022, 3774, 1678, 3775, 1678],
        [3023, 5915, 2998, 5915, 2998],
        [3024, 5956, 3024, 5955, 3024]]), tensor([1, 0, 1, 0, 1]))
```

11.4 Conclusion

The implementation of HGNN is a major obstacle for a beginner getting started. Although the source codes of some models are released, it is not easy to understand

and apply to new tasks. With the help of OpenHGNN, we can easily implement a new model and test the performance among various datasets. This chapter introduces the HGNN toolkit OpenHGNN and the pipeline of implementing a new model on it. We also provide three classical model implementation instances. The first model is HAN, which is the first model to introduce attention mechanism to HGNN. The second is RGCN, the first model to use graph convolution network to model multi-relational graph. The last one is HERec, which applies heterogeneous graph embedding to recommendation scenarios. These models are the most representative for learning on heterogeneous graphs. For the built-in models and datasets, we can run the model in just one command. For non-built-in models or datasets, implementation and testing are also very simple. OpenHGNN is under developing and more features and more models (including graph embedding methods) will be incorporated in the future version. For more details about OpenHGNN, please visit the website *https://github.com/BUPT-GAMMA/OpenHGNN*.

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015). URL https://www.tensorflow.org/. Software available from tensorflow.org
2. Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., Zhang, Z.: Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274 (2015)
3. Fey, M., Lenssen, J.E.: Fast graph representation learning with PyTorch Geometric. In: ICLR Workshop on Representation Learning on Graphs and Manifolds (2019)
4. Hu, B., Fang, Y., Shi, C.: Adversarial learning on heterogeneous information networks. In: KDD, pp. 120–129 (2019)
5. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: ICLR (2017)
6. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library. In: H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, R. Garnett (eds.) Advances in Neural Information Processing Systems, vol. 32. Curran Associates, Inc. (2019). URL https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf
7. Schlichtkrull, M., Kipf, T.N., Bloem, P., Van Den Berg, R., Titov, I., Welling, M.: Modeling relational data with graph convolutional networks. In: European Semantic Web Conference, pp. 593–607. Springer (2018)
8. Shi, C., Hu, B., Zhao, X., Yu, P.: Heterogeneous information network embedding for recommendation. IEEE Transactions on Knowledge and Data Engineering (2018)
9. Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., Zhang, Z.: Deep graph library: A graph-centric, highly-performant package for graph neural networks. arXiv preprint arXiv:1909.01315 (2019)

10. Wang, X., Ji, H., Shi, C., Wang, B., Ye, Y., Cui, P., Yu, P.S.: Heterogeneous graph attention network. In: The World Wide Web Conference, pp. 2022–2032 (2019)
11. Yun, S., Jeong, M., Kim, R., Kang, J., Kim, H.J.: Graph transformer networks. In: NIPS, pp. 11,960–11,970 (2019)
12. Zhang, C., Song, D., Huang, C., Swami, A., Chawla, N.V.: Heterogeneous graph neural network. In: KDD, pp. 793–803 (2019)