pgl

Release 1.0.1

PaddlePaddle

Jun 28, 2020

INTRODUCTION

Paddle Graph Learning (PGL) is an efficient and flexible graph learning framework based on PaddlePaddle.

The newly released PGL supports heterogeneous graph learning on both walk based paradigm and message-passing based paradigm by providing MetaPath sampling and Message Passing mechanism on heterogeneous graph. Furthermor, The newly released PGL also support distributed graph storage and some distributed training algorithms, such as distributed deep walk and distributed graphsage. Combined with the PaddlePaddle deep learning framework, we are able to support both graph representation learning models and graph neural networks, and thus our framework has a wide range of graph-based applications.

CHAPTER

ONE

HIGHLIGHT: EFFICIENCY - SUPPORT SCATTER-GATHER AND LODTENSOR MESSAGE PASSING

One of the most important benefits of graph neural networks compared to other models is the ability to use node-to-node connectivity information, but coding the communication between nodes is very cumbersome. At PGL we adopt **Message Passing Paradigm** similar to DGL to help to build a customize graph neural network easily. Users only need to write `send` and `recv` functions to easily implement a simple GCN. As shown in the following figure, for the first step the send function is defined on the edges of the graph, and the user can customize the send function ϕ^e to send the message from the source to the target node. For the second step, the recv function ϕ^v is responsible for aggregating $\oplus$ messages together from different sources.

As shown in the left of the following figure, to adapt general user-defined message aggregate functions, DGL uses the degree bucketing method to combine nodes with the same degree into a batch and then apply an aggregate function $\oplus$ on each batch serially. For our PGL UDF aggregate function, we organize the message as a LodTensor in PaddlePaddle taking the message as variable length sequences. And we **utilize the features of LodTensor in Paddle to obtain fast parallel aggregation**.

Users only need to call the `sequence_ops` functions provided by Paddle to easily implement efficient message aggregation. For examples, using `sequence_pool` to sum the neighbor message.

```
import paddle.fluid as fluid
def recv(msg):
    return fluid.layers.sequence_pool(msg, "sum")
```

Although DGL does some kernel fusion optimization for general sum, max and other aggregate functions with scatter-gather. For **complex user-defined functions** with degree bucketing algorithm, the serial execution for each degree bucket cannot take full advantage of the performance improvement provided by GPU. However, operations on the PGL LodTensor-based message is performed in parallel, which can fully utilize GPU parallel optimization. In our experiments, PGL can reach up to 13 times the speed of DGL with complex user-defined functions. Even without scatter-gather optimization, PGL still has excellent performance. Of course, we still provide build-in scatter-optimized message aggregation functions.

1.1 Performance

We test all the following GNN algorithms with Tesla V100-SXM2-16G running for 200 epochs to get average speeds. And we report the accuracy on test dataset without early stoppping.

Dataset	Model	PGL Accuracy	PGL speed (epoch time)	DGL 0.3.0 speed (epoch time)
Cora	GCN	81.75%	0.0047s	**0.0045s**
Cora	GAT	83.5%	**0.0119s**	0.0141s
Pubmed	GCN	79.2%	**0.0049s**	0.0051s
Pubmed	GAT	77%	0.0193s	**0.0144s**
Citeseer	GCN	70.2%	**0.0045**	0.0046s
Citeseer	GAT	68.8%	**0.0124s**	0.0139s

If we use complex user-defined aggregation like GraphSAGE-LSTM that aggregates neighbor features with LSTM ignoring the order of recieved messages, the optimized message-passing in DGL will be forced to degenerate into degree bucketing scheme. The speed performance will be much slower than the one implemented in PGL. Performances may be various with different scale of the graph, in our experiments, PGL can reach up to 13 times the speed of DGL.

Dataset	PGL speed (epoch time)	DGL 0.3.0 speed (epoch time)	Speed up
Cora	**0.0186s**	0.1638s	8.80x
Pubmed	**0.0388s**	0.5275s	13.59x
Citeseer	**0.0150s**	0.1278s	8.52x

CHAPTER

TWO

HIGHLIGHT: FLEXIBILITY - NATIVELY SUPPORT HETEROGENEOUS GRAPH LEARNING

Graph can conveniently represent the relation between things in the real world, but the categories of things and the relation between things are various. Therefore, in the heterogeneous graph, we need to distinguish the node types and edge types in the graph network. PGL models heterogeneous graphs that contain multiple node types and multiple edge types, and can describe complex connections between different types.

2.1 Support meta path walk sampling on heterogeneous graph

The left side of the figure above describes a shopping social network. The nodes above have two categories of users and goods, and the relations between users and users, users and goods, and goods and goods. The right of the above figure is a simple sampling process of MetaPath. When you input any MetaPath as UPU (user-product-user), you will find the following results

Then on this basis, and introducing word2vec and other methods to support learning metapath2vec and other algorithms of heterogeneous graph representation.

2.2 Support Message Passing mechanism on heterogeneous graph

Because of the different node types on the heterogeneous graph, the message delivery is also different. As shown on the left, it has five neighbors, belonging to two different node types. As shown on the right of the figure above, nodes belonging to different types need to be aggregated separately during message delivery, and then merged into the final message to update the target node. On this basis, PGL supports heterogeneous graph algorithms based on message passing, such as GATNE and other algorithms.

CHAPTER

THREE

LARGE-SCALE: SUPPORT DISTRIBUTED GRAPH STORAGE AND DISTRIBUTED TRAINING ALGORITHMS

In most cases of large-scale graph learning, we need distributed graph storage and distributed training support. As shown in the following figure, PGL provided a general solution of large-scale training, we adopted PaddleFleet as our distributed parameter servers, which supports large scale distributed embeddings and a lightweighted distributed storage engine so tcan easily set up a large scale distributed training algorithm with MPI clusters.

CHAPTER

FOUR

MODEL ZOO

The following are 13 graph learning models that have been implemented in the framework.

Model	feature
GCN	Graph Convolutional Neural Networks
GAT	Graph Attention Network
GraphSage	Large-scale graph convolution network based on neighborhood sampling
unSup-GraphSage	Unsupervised GraphSAGE
LINE	Representation learning based on first-order and second-order neighbors
DeepWalk	Representation learning by DFS random walk
MetaPath2Vec	Representation learning based on metapath
Node2Vec	The representation learning Combined with DFS and BFS
Struct2Vec	Representation learning based on structural similarity
SGC	Simplified graph convolution neural network
GES	The graph represents learning method with node features
DGI	Unsupervised representation learning based on graph convolution network
GATNE	Representation Learning of Heterogeneous Graph based on MessagePassing

The above models consists of three parts, namely, graph representation learning, graph neural network and heterogeneous graph learning, which are also divided into graph representation learning and graph neural network.

CHAPTER

FIVE

SYSTEM REQUIREMENTS

PGL requires:

- paddle >= 1.6
- cython

PGL supports both Python 2 & 3

CHAPTER

SIX

INSTALLATION

You can simply install it via pip.

```
pip install pgl
```

CHAPTER

SEVEN

THE TEAM

PGL is developed and maintained by NLP and Paddle Teams at Baidu

CHAPTER

EIGHT

LICENSE

PGL uses Apache License 2.0.

CHAPTER

NINE

PADDLE GRAPH LEARNING (PGL)

Paddle Graph Learning (PGL) is an efficient and flexible graph learning framework based on PaddlePaddle.

The newly released PGL supports heterogeneous graph learning on both walk based paradigm and message-passing based paradigm by providing MetaPath sampling and Message Passing mechanism on heterogeneous graph. Furthermor, The newly released PGL also support distributed graph storage and some distributed training algorithms, such as distributed deep walk and distributed graphsage. Combined with the PaddlePaddle deep learning framework, we are able to support both graph representation learning models and graph neural networks, and thus our framework has a wide range of graph-based applications.

9.1 Highlight: Efficiency - Support Scatter-Gather and LodTensor Message Passing

One of the most important benefits of graph neural networks compared to other models is the ability to use node-to-node connectivity information, but coding the communication between nodes is very cumbersome. At PGL we adopt **Message Passing Paradigm** similar to DGL to help to build a customize graph neural network easily. Users only need to write `send` and `recv` functions to easily implement a simple GCN. As shown in the following figure, for the first step the send function is defined on the edges of the graph, and the user can customize the send function ϕ^e to send the message from the source to the target node. For the second step, the recv function ϕ^v is responsible for aggregating $\oplus$ messages together from different sources.

As shown in the left of the following figure, to adapt general user-defined message aggregate functions, DGL uses the degree bucketing method to combine nodes with the same degree into a batch and then apply an aggregate function $\oplus$ on each batch serially. For our PGL UDF aggregate function, we organize the message as a LodTensor in PaddlePaddle taking the message as variable length sequences. And we **utilize the features of LodTensor in Paddle to obtain fast parallel aggregation**.

Users only need to call the `sequence_ops` functions provided by Paddle to easily implement efficient message aggregation. For examples, using `sequence_pool` to sum the neighbor message.

```
import paddle.fluid as fluid
def recv(msg):
    return fluid.layers.sequence_pool(msg, "sum")
```

Although DGL does some kernel fusion optimization for general sum, max and other aggregate functions with scatter-gather. For **complex user-defined functions** with degree bucketing algorithm, the serial execution for each degree bucket cannot take full advantage of the performance improvement provided by GPU. However, operations on the PGL LodTensor-based message is performed in parallel, which can fully utilize GPU parallel optimization. In our experiments, PGL can reach up to 13 times the speed of DGL with complex user-defined functions. Even without scatter-gather optimization, PGL still has excellent performance. Of course, we still provide build-in scatter-optimized message aggregation functions.

9.1.1 Performance

We test all the following GNN algorithms with Tesla V100-SXM2-16G running for 200 epochs to get average speeds. And we report the accuracy on test dataset without early stoppping.

Dataset	Model	PGL Accuracy	PGL speed (epoch time)	DGL 0.3.0 speed (epoch time)
Cora	GCN	81.75%	0.0047s	**0.0045s**
Cora	GAT	83.5%	**0.0119s**	0.0141s
Pubmed	GCN	79.2%	**0.0049s**	0.0051s
Pubmed	GAT	77%	0.0193s	**0.0144s**
Citeseer	GCN	70.2%	**0.0045**	0.0046s
Citeseer	GAT	68.8%	**0.0124s**	0.0139s

If we use complex user-defined aggregation like GraphSAGE-LSTM that aggregates neighbor features with LSTM ignoring the order of recieved messages, the optimized message-passing in DGL will be forced to degenerate into degree bucketing scheme. The speed performance will be much slower than the one implemented in PGL. Performances may be various with different scale of the graph, in our experiments, PGL can reach up to 13 times the speed of DGL.

Dataset	PGL speed (epoch time)	DGL 0.3.0 speed (epoch time)	Speed up
Cora	**0.0186s**	0.1638s	8.80x
Pubmed	**0.0388s**	0.5275s	13.59x
Citeseer	**0.0150s**	0.1278s	8.52x

9.2 Highlight: Flexibility - Natively Support Heterogeneous Graph Learning

Graph can conveniently represent the relation between things in the real world, but the categories of things and the relation between things are various. Therefore, in the heterogeneous graph, we need to distinguish the node types and edge types in the graph network. PGL models heterogeneous graphs that contain multiple node types and multiple edge types, and can describe complex connections between different types.

9.2.1 Support meta path walk sampling on heterogeneous graph

The left side of the figure above describes a shopping social network. The nodes above have two categories of users and goods, and the relations between users and users, users and goods, and goods and goods. The right of the above figure is a simple sampling process of MetaPath. When you input any MetaPath as UPU (user-product-user), you will find the following results

Then on this basis, and introducing word2vec and other methods to support learning metapath2vec and other algorithms of heterogeneous graph representation.

9.2.2 Support Message Passing mechanism on heterogeneous graph

Because of the different node types on the heterogeneous graph, the message delivery is also different. As shown on the left, it has five neighbors, belonging to two different node types. As shown on the right of the figure above, nodes belonging to different types need to be aggregated separately during message delivery, and then merged into the final message to update the target node. On this basis, PGL supports heterogeneous graph algorithms based on message passing, such as GATNE and other algorithms.

9.3 Large-Scale: Support distributed graph storage and distributed training algorithms

In most cases of large-scale graph learning, we need distributed graph storage and distributed training support. As shown in the following figure, PGL provided a general solution of large-scale training, we adopted PaddleFleet as our distributed parameter servers, which supports large scale distributed embeddings and a lightweighted distributed storage engine so tcan easily set up a large scale distributed training algorithm with MPI clusters.

9.4 Model Zoo

The following are 13 graph learning models that have been implemented in the framework.

Model	feature
GCN	Graph Convolutional Neural Networks
GAT	Graph Attention Network
GraphSage	Large-scale graph convolution network based on neighborhood sampling
unSup-GraphSage	Unsupervised GraphSAGE
LINE	Representation learning based on first-order and second-order neighbors
DeepWalk	Representation learning by DFS random walk
MetaPath2Vec	Representation learning based on metapath
Node2Vec	The representation learning Combined with DFS and BFS
Struct2Vec	Representation learning based on structural similarity
SGC	Simplified graph convolution neural network
GES	The graph represents learning method with node features
DGI	Unsupervised representation learning based on graph convolution network
GATNE	Representation Learning of Heterogeneous Graph based on MessagePassing

The above models consists of three parts, namely, graph representation learning, graph neural network and heterogeneous graph learning, which are also divided into graph representation learning and graph neural network.

9.5 System requirements

PGL requires:

- paddle >= 1.6
- cython

PGL supports both Python 2 & 3

9.6 Installation

You can simply install it via pip.

```
pip install pgl
```

9.7 The Team

PGL is developed and maintained by NLP and Paddle Teams at Baidu

9.8 License

PGL uses Apache License 2.0.

CHAPTER

TEN

QUICK START

10.1 Quick Start Instructions

10.1.1 Install PGL

To install Paddle Graph Learning, we need the following packages.

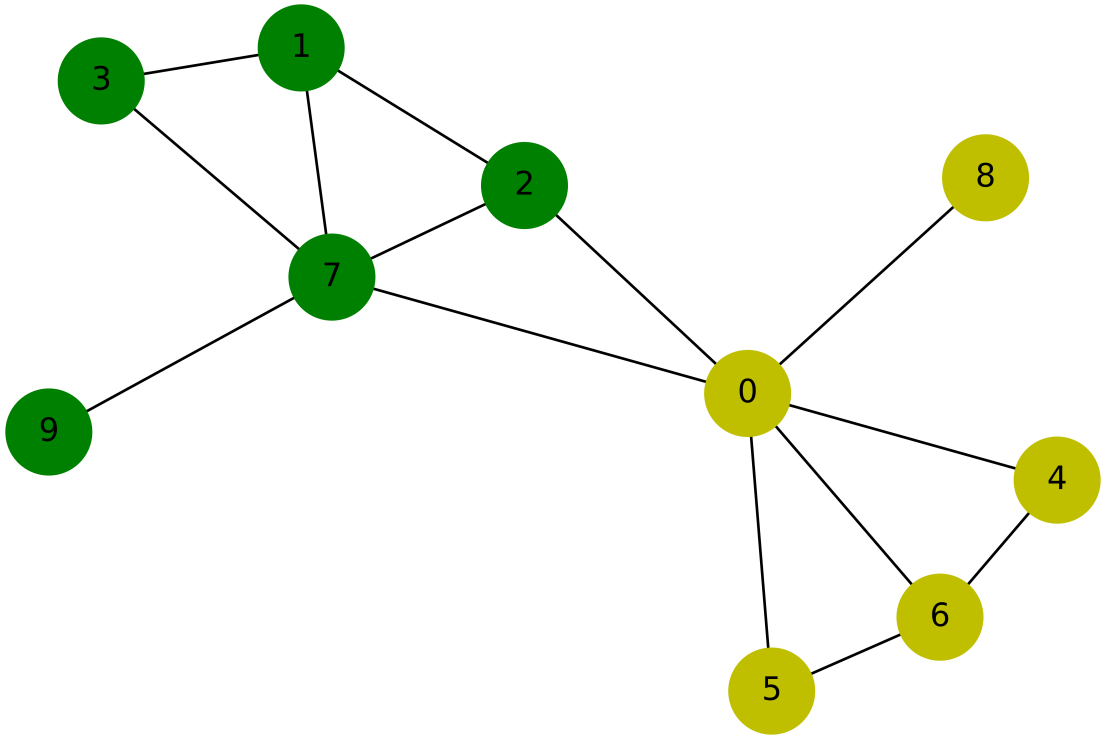
```
paddlepaddle >= 1.6
cython
```

We can simply install pgl by pip.

```
pip install pgl
```

10.1.2 Step 1: using PGL to create a graph

Suppose we have a graph with 10 nodes and 14 edges as shown in the following figure:



Our purpose is to train a graph neural network to classify yellow and green nodes. So we can create this graph in such way:

```
import pgl
from pgl import graph  # import pgl module
import numpy as np

def build_graph():
    # define the number of nodes; we can use number to represent every node
    num_node = 10
    # add edges, we represent all edges as a list of tuple (src, dst)
    edge_list = [(2, 0), (2, 1), (3, 1),(4, 0), (5, 0),
                 (6, 0), (6, 4), (6, 5), (7, 0), (7, 1),
                 (7, 2), (7, 3), (8, 0), (9, 7)]

    # Each node can be represented by a d-dimensional feature vector, here for simple,
↪ the feature vectors are randomly generated.
    d = 16
    feature = np.random.randn(num_node, d).astype("float32")
    # each edge has it own weight
    edge_feature = np.random.randn(len(edge_list), 1).astype("float32")

    # create a graph
    g = graph.Graph(num_nodes = num_node,
                    edges = edge_list,
                    node_feat = {'feature':feature},
                    edge_feat ={'edge_feature': edge_feature})

    return g

# create a graph object for saving graph data
```

(continues on next page)

(continued from previous page)

```
g = build_graph()
```

After creating a graph in PGL, we can print out some information in the graph.

```
print('There are %d nodes in the graph.'%g.num_nodes)
print('There are %d edges in the graph.'%g.num_edges)

# Out:
# There are 10 nodes in the graph.
# There are 14 edges in the graph.
```

Currently our PGL is developed based on static computational mode of paddle (we'll support dynamic computational model later). We need to build model upon a virtual data holder. GraphWrapper provide a virtual graph structure that users can build deep learning models based on this virtual graph. And then feed real graph data to run the models.

```
import paddle.fluid as fluid

use_cuda = False
place = fluid.CUDAPlace(0) if use_cuda else fluid.CPUPlace()

# use GraphWrapper as a container for graph data to construct a graph neural network
gw = pgl.graph_wrapper.GraphWrapper(name='graph',
                        node_feat=g.node_feat_info())
```

10.1.3 Step 2: create a simple Graph Convolutional Network(GCN)

In this tutorial, we use a simple Graph Convolutional Network(GCN) developed by Kipf and Welling to perform node classification. Here we use the simplest GCN structure. If readers want to know more about GCN, you can refer to the original paper.

- In layer $l$each node u_i^l has a feature vector h_i^l;
- In every layer, the idea of GCN is that the feature vector h_i^{l+1} of each node u_i^{l+1} in the next layer are obtained by weighting the feature vectors of all the neighboring nodes and then go through a non-linear transformation.

In PGL, we can easily implement a GCN layer as follows:

```
# define GCN layer function
def gcn_layer(gw, nfeat, efeat, hidden_size, name, activation):
    # gw is a GraphWrapperfeature is the feature vectors of nodes

    # define message function
    def send_func(src_feat, dst_feat, edge_feat):
        # In this tutorial, we return the feature vector of the source node as message
        return src_feat['h'] * edge_feat['e']

    # define reduce function
    def recv_func(feat):
        # we sum the feature vector of the source node
        return fluid.layers.sequence_pool(feat, pool_type='sum')

    # trigger message to passing
    msg = gw.send(send_func, nfeat_list=[('h', nfeat)], efeat_list=[('e', efeat)])
    # recv funciton receives message and trigger reduce funcition to handle message
    output = gw.recv(msg, recv_func)
```

(continues on next page)

(continued from previous page)

```
    output = fluid.layers.fc(output,
                     size=hidden_size,
                     bias_attr=False,
                     act=activation,
                     name=name)
    return output
```

After defining the GCN layer, we can construct a deeper GCN model with two GCN layers.

```
output = gcn_layer(gw, gw.node_feat['feature'], gw.edge_feat['edge_feature'],
                   hidden_size=8, name='gcn_layer_1', activation='relu')
output = gcn_layer(gw, output, gw.edge_feat['edge_feature'],
                   hidden_size=1, name='gcn_layer_2', activation=None)
```

10.1.4 Step 3: data preprocessing

Since we implement a node binary classifier, we can use 0 and 1 to represent two classes respectively.

```
y = [0,1,1,1,0,0,0,1,0,1]
label = np.array(y, dtype="float32")
label = np.expand_dims(label, -1)
```

10.1.5 Step 4: training program

The training process of GCN is the same as that of other paddle-based models.

- First we create a loss function.
- Then we create a optimizer.
- Finally, we create a executor and train the model.

```
# create a label layer as a container
node_label = fluid.layers.data("node_label", shape=[None, 1],
            dtype="float32", append_batch_size=False)

# using cross-entropy with sigmoid layer as the loss function
loss = fluid.layers.sigmoid_cross_entropy_with_logits(x=output, label=node_label)

# calculate the mean loss
loss = fluid.layers.mean(loss)

# choose the Adam optimizer and set the learning rate to be 0.01
adam = fluid.optimizer.Adam(learning_rate=0.01)
adam.minimize(loss)

# create the executor
exe = fluid.Executor(place)
exe.run(fluid.default_startup_program())
feed_dict = gw.to_feed(g) # gets graph data

for epoch in range(30):
    feed_dict['node_label'] = label
```

(continues on next page)

(continued from previous page)

```
    train_loss = exe.run(fluid.default_main_program(),
        feed=feed_dict,
        fetch_list=[loss],
        return_numpy=True)
    print('Epoch %d | Loss: %f'%(epoch, train_loss[0]))
```

10.2 Quick Start with Heterogenous Graph

10.2.1 Install PGL

To install Paddle Graph Learning, we need the following packages.

```
paddlepaddle >= 1.6
cython
```

We can simply install pgl by pip.

```
pip install pgl
```

10.2.2 Introduction

In real world, there exists many graphs contain multiple types of nodes and edges, which we call them Heterogeneous Graphs. Obviously, heterogenous graphs are more complex than homogeneous graphs.

To deal with such heterogeneous graphs, PGL develops a graph framework to support graph neural network computations and meta-path-based sampling on heterogenous graph.

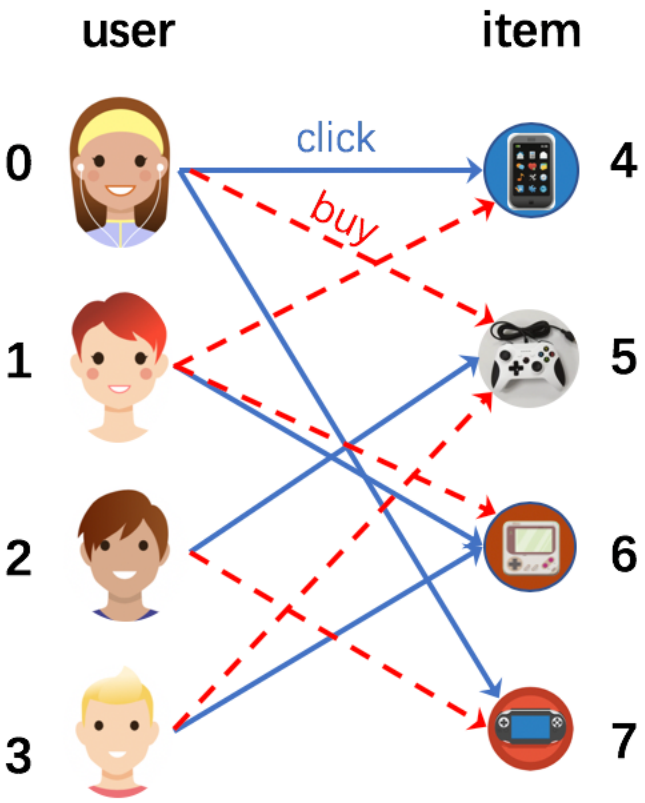
The goal of this tutorial:

- example of heterogenous graph data;
- Understand how PGL supports computations in heterogenous graph;
- Using PGL to implement a simple heterogenous graph neural network model to classfiy a particular type of node in a heterogenous graph network.

10.2.3 Example of heterogenous graph

There are a lot of graph data that consists of edges and nodes of multiple types. For example, **e-commerce network** is very common heterogenous graph in real world. It contains at least two types of nodes (user and item) and two types of edges (buy and click).

The following figure depicts several users click or buy some items. This graph has two types of nodes corresponding to “user” and “item”. It also contain two types of edge “buy” and “click”.



10.2.4 Creating a heterogenous graph with PGL

In heterogenous graph, there exists multiple edges, so we should distinguish them. In PGL, the edges are built in below format:

```
edges = {
    'click': [(0, 4), (0, 7), (1, 6), (2, 5), (3, 6)],
    'buy': [(0, 5), (1, 4), (1, 6), (2, 7), (3, 5)],
        }
```

In heterogenous graph, nodes are also of different types. Therefore, you need to mark the type of each node, the format of the node type is as follows:

```
node_types = [(0, 'user'), (1, 'user'), (2, 'user'), (3, 'user'), (4, 'item'),
              (5, 'item'),(6, 'item'), (7, 'item')]
```

Because of the different types of edges, edge features also need to be separated by different types.

```
import numpy as np

num_nodes = len(node_types)

node_features = {'features': np.random.randn(num_nodes, 8).astype("float32")}

edge_num_list = []
for edge_type in edges:
```

(continues on next page)

(continued from previous page)

```
    edge_num_list.append(len(edges[edge_type]))

edge_features = {
    'click': {'h': np.random.randn(edge_num_list[0], 4)},
    'buy': {'h':np.random.randn(edge_num_list[1], 4)},
}
```

Now, we can build a heterogenous graph by using PGL.

```
import paddle.fluid as fluid
import paddle.fluid.layers as fl
import pgl
from pgl import heter_graph
from pgl import heter_graph_wrapper

g = heter_graph.HeterGraph(num_nodes=num_nodes,
                           edges=edges,
                           node_types=node_types,
                           node_feat=node_features,
                           edge_feat=edge_features)
```

In PGL, we need to use graph_wrapper as a container for graph data, so here we need to create a graph_wrapper for each type of edge graph.

```
place = fluid.CPUPlace()

# create a GraphWrapper as a container for graph data
gw = heter_graph_wrapper.HeterGraphWrapper(name='heter_graph',
                                    edge_types = g.edge_types_info(),
                                    node_feat=g.node_feat_info(),
                                    edge_feat=g.edge_feat_info())
```

10.2.5 MessagePassing

After building the heterogeneous graph, we can easily carry out the message passing mode. In this case, we have two different types of edges, so we can write a function in such way:

```
def message_passing(gw, edge_types, features, name=''):
    def __message(src_feat, dst_feat, edge_feat):
        return src_feat['h']
    def __reduce(feat):
        return fluid.layers.sequence_pool(feat, pool_type='sum')

    assert len(edge_types) == len(features)
    output = []
    for i in range(len(edge_types)):
        msg = gw[edge_types[i]].send(__message, nfeat_list=[('h', features[i])])
        out = gw[edge_types[i]].recv(msg, __reduce)
        output.append(out)
    # list of matrix
    return output
```

```
edge_types = ['click', 'buy']
features = []
```

(continues on next page)

(continued from previous page)

```
for edge_type in edge_types:
    features.append(gw[edge_type].node_feat['features'])
output = message_passing(gw, edge_types, features)

output = fl.concat(input=output, axis=1)

output = fluid.layers.fc(output, size=4, bias_attr=False, act='relu', name='fc1')
logits = fluid.layers.fc(output, size=1, bias_attr=False, act=None, name='fc2')
```

10.2.6 data preprocessing

In this case, we implement a simple node classifier, we can use 0,1 to represent two classes.

```
y = [0,1,0,1,0,1,1,0]
label = np.array(y, dtype="float32").reshape(-1,1)
```

10.2.7 Setting up the training program

The training process of the heterogeneous graph node classification model is the same as the training of other paddlepaddle-based models.

- First we build the loss function;
- Second, creating a optimizer;
- Finally, creating a executor and execute the training program.

```
node_label = fluid.layers.data("node_label", shape=[None, 1], dtype="float32", append_
↪batch_size=False)


loss = fluid.layers.sigmoid_cross_entropy_with_logits(x=logits, label=node_label)

loss = fluid.layers.mean(loss)


adam = fluid.optimizer.Adam(learning_rate=0.01)
adam.minimize(loss)


exe = fluid.Executor(place)
exe.run(fluid.default_startup_program())
feed_dict = gw.to_feed(g)

for epoch in range(30):
    feed_dict['node_label'] = label

    train_loss = exe.run(fluid.default_main_program(), feed=feed_dict, fetch_
↪list=[loss], return_numpy=True)
    print('Epoch %d | Loss: %f'%(epoch, train_loss[0]))
```

10.3 GCN: Graph Convolutional Networks

Graph Convolutional Network (GCN) is a powerful neural network designed for machine learning on graphs. Based on PGL, we reproduce GCN algorithms and reach the same level of indicators as the paper in citation network benchmarks.

10.3.1 Simple example to build GCN

To build a gcn layer, one can use our pre-defined `pgl.layers.gcn` or just write a gcn layer with message passing interface.

```
import paddle.fluid as fluid
def gcn_layer(graph_wrapper, node_feature, hidden_size, act):
    def send_func(src_feat, dst_feat, edge_feat):
        return src_feat["h"]

    def recv_func(msg):
        return fluid.layers.sequence_pool(msg, "sum")

    message = graph_wrapper.send(send_func, nfeat_list=[("h", node_feature)])
    output = graph_wrapper.recv(recv_func, message)
    output = fluid.layers.fc(output, size=hidden_size, act=act)
    return output
```

10.3.2 Datasets

The datasets contain three citation networks: CORA, PUBMED, CITESEER. The details for these three datasets can be found in the paper.

10.3.3 Dependencies

- paddlepaddle>=1.6
- pgl

10.3.4 Performance

We train our models for 200 epochs and report the accuracy on the test dataset.

Dataset	Accuracy
Cora	~81%
Pubmed	~79%
Citeseer	~71%

10.3.5 How to run

For examples, use gpu to train gcn on cora dataset.

```
python train.py --dataset cora --use_cuda
```

Hyperparameters

- dataset: The citation dataset "cora", "citeseer", "pubmed".
- use_cuda: Use gpu if assign use_cuda.

10.4 GAT: Graph Attention Networks

Graph Attention Networks (GAT) is a novel architectures that operate on graph-structured data, which leverages masked self-attentional layers to address the shortcomings of prior methods based on graph convolutions or their approximations. Based on PGL, we reproduce GAT algorithms and reach the same level of indicators as the paper in citation network benchmarks.

10.4.1 Simple example to build single head GAT

To build a gat layer, one can use our pre-defined `pgl.layers.gat` or just write a gat layer with message passing interface.

```
import paddle.fluid as fluid
def gat_layer(graph_wrapper, node_feature, hidden_size):
    def send_func(src_feat, dst_feat, edge_feat):
        logits = src_feat["a1"] + dst_feat["a2"]
        logits = fluid.layers.leaky_relu(logits, alpha=0.2)
        return {"logits": logits, "h": src_feat }

    def recv_func(msg):
        norm = fluid.layers.sequence_softmax(msg["logits"])
        output = msg["h"] * norm
        return output

    h = fluid.layers.fc(node_feature, hidden_size, bias_attr=False, name="hidden")
    a1 = fluid.layers.fc(node_feature, 1, name="a1_weight")
    a2 = fluid.layers.fc(node_feature, 1, name="a2_weight")
    message = graph_wrapper.send(send_func,
            nfeat_list=[("h", h), ("a1", a1), ("a2", a2)])
    output = graph_wrapper.recv(recv_func, message)
    return output
```

10.4.2 Datasets

The datasets contain three citation networks: CORA, PUBMED, CITESEER. The details for these three datasets can be found in the paper.

10.4.3 Dependencies

- paddlepaddle>=1.6
- pgl

10.4.4 Performance

We train our models for 200 epochs and report the accuracy on the test dataset.

Dataset	Accuracy
Cora	~83%
Pubmed	~78%
Citeseer	~70%

10.4.5 How to run

For examples, use gpu to train gat on cora dataset.

```
python train.py --dataset cora --use_cuda
```

Hyperparameters

- dataset: The citation dataset "cora", "citeseer", "pubmed".
- use_cuda: Use gpu if assign use_cuda.

10.5 Using StaticGraphWrapper for Speed Optimization

10.5.1 PGL Examples for GCN with StaticGraphWrapper

Graph Convolutional Network (GCN) is a powerful neural network designed for machine learning on graphs. Based on PGL, we reproduce GCN algorithms and reach the same level of indicators as the paper in citation network benchmarks.

However, different from the reproduction in **examples/gcn**, we use `pgl.graph_wrapper.StaticGraphWrapper` to preload the graph data into gpu or cpu memories which achieves better performance on speed.

Datasets

The datasets contain three citation networks: CORA, PUBMED, CITESEER. The details for these three datasets can be found in the paper.

Dependencies

- paddlepaddle>=1.6
- pgl

Performance

We train our models for 200 epochs and report the accuracy on the test dataset.

Dataset	Accuracy	epoch time	examples/gcn	Improvement
Cora	~81%	0.0047s	0.0104s	2.21x
Pubmed	~79%	0.0049s	0.0154s	3.14x
Citeseer	~71%	0.0045s	0.0177s	3.93x

How to run

For examples, use gpu to train gcn on cora dataset.

```
python train.py --dataset cora --use_cuda
```

Hyperparameters

- dataset: The citation dataset "cora", "citeseer", "pubmed".
- use_cuda: Use gpu if assign use_cuda.

10.5.2 PGL Examples for GAT with StaticGraphWrapper

Graph Attention Networks (GAT) is a novel architectures that operate on graph-structured data, which leverages masked self-attentional layers to address the shortcomings of prior methods based on graph convolutions or their approximations. Based on PGL, we reproduce GAT algorithms and reach the same level of indicators as the paper in citation network benchmarks.

However, different from the reproduction in **examples/gat**, we use `pgl.graph_wrapper.StaticGraphWrapper` to preload the graph data into gpu or cpu memories which achieves better performance on speed.

Datasets

The datasets contain three citation networks: CORA, PUBMED, CITESEER. The details for these three datasets can be found in the paper.

Dependencies

- paddlepaddle>=1.6
- pgl

Performance

We train our models for 200 epochs and report the accuracy on the test dataset.

Dataset	Accuracy	epoch time	examples/gat	Improvement
Cora	~83%	0.0119s	0.0175s	1.47x
Pubmed	~78%	0.0193s	0.0295s	1.53x
Citeseer	~70%	0.0124s	0.0253s	2.04x

How to run

For examples, use gpu to train gat on cora dataset.

```
python train.py --dataset cora --use_cuda
```

Hyperparameters

- dataset: The citation dataset "cora", "citeseer", "pubmed".
- use_cuda: Use gpu if assign use_cuda.

10.6 node2vec: Scalable Feature Learning for Networks

Node2vec is an algorithmic framework for representational learning on graphs. Given any graph, it can learn continuous feature representations for the nodes, which can then be used for various downstream machine learning tasks. Based on PGL, we reproduce node2vec algorithms and reach the same level of indicators as the paper.

10.6.1 Datasets

The datasets contain two networks: BlogCatalog and Arxiv.

10.6.2 Dependencies

- paddlepaddle>=1.4
- pgl

10.6.3 How to run

For examples, use gpu to train gcn on cora dataset.

```
# multiclass task example
python node2vec.py --use_cuda --dataset BlogCatalog --save_path ./tmp/node2vec_
↪BlogCatalog/ --offline_learning --epoch 400

python multi_class.py --use_cuda --ckpt_path ./tmp/node2vec_BlogCatalog/paddle_model -
↪-epoch 1000

# link prediction task example
python node2vec.py --use_cuda --dataset ArXiv --save_path
./tmp/node2vec_ArXiv --offline_learning --epoch 10
```

(continues on next page)

(continued from previous page)

```
python link_predict.py --use_cuda --ckpt_path ./tmp/node2vec_ArXiv/paddle_model --
↪epoch 400
```

10.6.4 Hyperparameters

- dataset: The citation dataset "BlogCatalog" and "ArXiv".
- use_cuda: Use gpu if assign use_cuda.

Experiment results

Dataset	model	Task	Metric	PGL Result	Reported Result
BlogCatalog	deepwalk	multi-label classification	MacroF1	0.250	0.211
BlogCatalog	node2vec	multi-label classification	MacroF1	0.262	0.258
ArXiv	deepwalk	link prediction	AUC	0.9538	0.9340
ArXiv	node2vec	link prediction	AUC	0.9541	0.9366

10.7 GraphSAGE: Inductive Representation Learning on Large Graphs

GraphSAGE is a general inductive framework that leverages node feature information (e.g., text attributes) to efficiently generate node embeddings for previously unseen data. Instead of training individual embeddings for each node, GraphSAGE learns a function that generates embeddings by sampling and aggregating features from a node's local neighborhood. Based on PGL, we reproduce GraphSAGE algorithm and reach the same level of indicators as the paper in Reddit Dataset. Besides, this is an example of subgraph sampling and training in PGL.

10.7.1 Datasets

The reddit dataset should be downloaded from the following links and placed in directory `./data`. The details for Reddit Dataset can be found here.

- reddit.npz https://drive.google.com/open?id=19SphVl_Oe8SJ1r87Hr5a6znx3nJu1F2J
- reddit_adj.npz: https://drive.google.com/open?id=174vb0Ws7Vxk_QTUtxqTgDHSQ4El4qDHt

10.7.2 Dependencies

- paddlepaddle>=1.6
- pgl

10.7.3 How to run

To train a GraphSAGE model on Reddit Dataset, you can just run

```
python train.py --use_cuda --epoch 10 --graphsage_type graphsage_mean --normalize --
↪symmetry
```

If you want to train a GraphSAGE model with multiple GPUs, you can just run

```
CUDA_VISIBLE_DEVICES=0,1 python train_multi.py --use_cuda --epoch 10 --graphsage_type␣
↪graphsage_mean --normalize --symmetry  --num_trainer 2
```

Hyperparameters

- epoch: Number of epochs default (10)
- use_cuda: Use gpu if assign use_cuda.
- graphsage_type: We support 4 aggregator types including "graphsage_mean", "graphsage_maxpool", "graphsage_meanpool" and "graphsage_lstm".
- normalize: Normalize the input feature if assign normalize.
- sample_workers: The number of workers for multiprocessing subgraph sample.
- lr: Learning rate.
- symmetry: Make the edges symmetric if assign symmetry.
- batch_size: Batch size.
- samples_1: The max neighbors for the first hop neighbor sampling. (default: 25)
- samples_2: The max neighbors for the second hop neighbor sampling. (default: 10)
- hidden_size: The hidden size of the GraphSAGE models.

10.7.4 Performance

We train our models for 200 epochs and report the accuracy on the test dataset.

Aggregator	Accuracy	Reported in paper
Mean	95.70%	95.0%
Meanpool	95.60%	94.8%
Maxpool	94.95%	94.8%
LSTM	95.13%	95.4%

10.8 DGI: Deep Graph Infomax

Deep Graph Infomax (DGI) is a general approach for learning node representations within graph-structured data in an unsupervised manner. DGI relies on maximizing mutual information between patch representations and corresponding high-level summaries of graphs—both derived using established graph convolutional network architectures.

10.8.1 Datasets

The datasets contain three citation networks: CORA, PUBMED, CITESEER. The details for these three datasets can be found in the paper.

10.8.2 Dependencies

- paddlepaddle>=1.6
- pgl

10.8.3 Performance

We use DGI to pretrain embeddings for each nodes. Then we fix the embedding to train a node classifier.

Dataset	Accuracy
Cora	~81%
Pubmed	~77.6%
Citeseer	~71.3%

10.8.4 How to run

For examples, use gpu to train gcn on cora dataset.

```
python dgi.py --dataset cora --use_cuda
python train.py --dataset cora --use_cuda
```

Hyperparameters

- dataset: The citation dataset "cora", "citeseer", "pubmed".
- use_cuda: Use gpu if assign use_cuda.

10.9 Distributed Deepwalk in PGL

Deepwalk is an algorithmic framework for representational learning on graphs. Given any graph, it can learn continuous feature representations for the nodes, which can then be used for various downstream machine learning tasks. Based on PGL, we reproduce distributed deepwalk algorithms and reach the same level of indicators as the paper.

10.9.1 Datasets

The datasets contain two networks: BlogCatalog.

10.9.2 Dependencies

- paddlepaddle>=1.6
- pgl>=1.0

10.9.3 How to run

We adopt PaddlePaddle Fleet as our distributed training frameworks `pgl_deepwalk.cfg` is config file for deepwalk hyperparameter and `local_config` is a config file for parameter servers. By default, we have 2 pservers and 2 trainers. We can use `cloud_run.sh` to help you startup the parameter servers and model trainers.

For examples, train deepwalk in distributed mode on BlogCataLog dataset.

```
# train deepwalk in distributed mode.
sh cloud_run.sh

# multiclass task example
python3 multi_class.py --use_cuda --ckpt_path ./model_path/4029 --epoch 1000
```

10.9.4 Hyperparameters

- dataset: The citation dataset "BlogCatalog".
- hidden_size: Hidden size of the embedding.
- lr: Learning rate.
- neg_num: Number of negative samples.
- epoch: Number of training epoch.

Experiment results

Dataset	model	Task	Metric	PGL Result	Reported Result
BlogCatalog	distributed deepwalk	multi-label classification	MacroF1	0.233	0.211

10.10 Distribute GraphSAGE in PGL

GraphSAGE is a general inductive framework that leverages node feature information (e.g., text attributes) to efficiently generate node embeddings for previously unseen data. Instead of training individual embeddings for each node, GraphSAGE learns a function that generates embeddings by sampling and aggregating features from a node's local neighborhood. Based on PGL, we reproduce GraphSAGE algorithm and reach the same level of indicators as the paper in Reddit Dataset. Besides, this is an example of subgraph sampling and training in PGL.

For purpose of high scalability, we use redis as distribute graph storage solution and training graphsage against redis server.

10.10.1 Datasets(Quickstart)

The reddit dataset should be downloaded from reddit_adj.npz and reddit.npz. The details for Reddit Dataset can be found here.

- reddit.npz: https://drive.google.com/open?id=19SphVl_Oe8SJ1r87Hr5a6znx3nJu1F2J
- reddit_adj.npz: https://drive.google.com/open?id=174vb0Ws7Vxk_QTUtxqTgDHSQ4El4qDHt

Download `reddit.npz` and `reddit_adj.npz` into `data` directory for further preprocessing.

10.10.2 Dependencies

```
pip install -r requirements.txt
```

10.10.3 How to run

1. Preprocessing and start reddit data service

```
pushd ./redis_setup
    /bin/bash ./before_hook.sh
popd
```

2. training GraphSAGE model

```
sh ./cloud_run.sh
```

10.11 GES: Graph Embedding with Side Information

Graph Embedding with Side Information is an algorithmic framework for representational learning on graphs. Given any graph, it can learn continuous feature representations for the nodes, which can then be used for various downstream machine learning tasks. Based on PGL, we reproduce ges algorithms.

10.11.1 Datasets

The datasets contain two networks: BlogCatalog.

10.11.2 Dependencies

- paddlepaddle>=1.6
- pgl>=1.0.0

10.11.3 How to run

For examples, train ges on cora dataset.

```
# train deepwalk in distributed mode.
sh gpu_run.sh
```

10.11.4 Hyperparameters

- dataset: The citation dataset "BlogCatalog".
- hidden_size: Hidden size of the embedding.
- lr: Learning rate.

- neg_num: Number of negative samples.
- epoch: Number of training epoch.

10.12 LINE: Large-scale Information Network Embedding

LINE is an algorithmic framework for embedding very large-scale information networks. It is suitable to a variety of networks including directed, undirected, binary or weighted edges. Based on PGL, we reproduce LINE algorithms and reach the same level of indicators as the paper.

10.12.1 Datasets

Flickr network is a social network, which contains 1715256 nodes and 22613981 edges.

You can dowload data from here.

Flickr network contains four files:

- flickr-groupmemberships.txt.gz
- flickr-groups.txt.gz
- flickr-links.txt.gz
- flickr-users.txt.gz

After downloading the datauncompress them, let's say, in **./data/flickr/** . Note that the current directory is the root directory of LINE model.

Then you can run the below command to preprocess the data.

```
python data_process.py
```

Then it will produce three files in **./data/flickr/** directory:

- nodes.txt
- edges.txt
- nodes_label.txt

10.12.2 Dependencies

- paddlepaddle>=1.6
- pgl

10.12.3 How to run

For examples, use gpu to train LINE on Flickr dataset.

```
# multiclass task example
python line.py --use_cuda --order first_order --data_path ./data/flickr/ --save_dir ./
↪checkpoints/model/

python multi_class.py --ckpt_path ./checkpoints/model/model_epoch_20 --percent 0.5
```

10.12.4 Hyperparameters

- -use_cuda: Use gpu if assign use_cuda.
- -order: LINE with First_order Proximity or Second_order Proximity
- -percent: The percentage of data as training data

Experiment results

Dataset	model	Task	Metric	PGL Result	Reported Result
Flickr	LINE with first_order	multi-label classification	MacroF1	0.626	0.627
Flickr	LINE with first_order	multi-label classification	MicroF1	0.637	0.639
Flickr	LINE with second_order	multi-label classification	MacroF1	0.615	0.621
Flickr	LINE with second_order	multi-label classification	MicroF1	0.630	0.635

10.13 SGC: Simplifying Graph Convolutional Networks

Simplifying Graph Convolutional Networks (SGC) is a powerful neural network designed for machine learning on graphs. Based on PGL, we reproduce SGC algorithms and reach the same level of indicators as the paper in citation network benchmarks.

10.13.1 Datasets

The datasets contain three citation networks: CORA, PUBMED, CITESEER. The details for these three datasets can be found in the paper.

10.13.2 Dependencies

- paddlepaddle 1.5
- pgl

10.13.3 Performance

We train our models for 200 epochs and report the accuracy on the test dataset.

Dataset	Accuracy	Speed with paddle 1.5 (epoch time)
Cora	0.818 (paper: 0.810)	0.0015s
Pubmed	0.788 (paper: 0.789)	0.0015s
Citeseer	0.719 (paper: 0.719)	0.0015s

10.13.4 How to run

For examples, use gpu to train SGC on cora dataset.

```
python sgc.py --dataset cora --use_cuda
```

Hyperparameters

- dataset: The citation dataset "cora", "citeseer", "pubmed".
- use_cuda: Use gpu if assign use_cuda.

10.14 struc2vec: Learning Node Representations from Structural Identity

Struc2vec is is a concept of symmetry in which network nodes are identified according to the network structure and their relationship to other nodes. A novel and flexible framework for learning latent representations is proposed in the paper of struc2vec. We reproduce Struc2vec algorithm in the PGL.

10.14.1 DataSet

The paper of use air-traffic network to valid algorithm of Struc2vec. The each edge in the dataset indicate that having one flight between the airports. Using the the connection between the airports to predict the level of activity. The following dataset will be used to valid the algorithm accuracy.Data collected from the Bureau of Transportation Statistics2 from January to October, 2016. The network has 1,190 nodes, 13,599 edges (diameter is 8). Link

- usa-airports.edgelist
- labels-usa-airports.txt

10.14.2 Dependencies

If use want to use the struc2vec model in pgl, please install the gensim, pathos, fastdtw additional.

- paddlepaddle>=1.6
- pgl
- gensim
- pathos
- fastdtw

10.14.3 How to use

For examples, we want to train and valid the Struc2vec model on American airpot dataset

```
python struc2vec.py –edge_file data/usa-airports.edgelist –label_file data/labels-usa-airports.txt –train True –valid True –opt2 True
```

10.14.4 Hyperparameters

Args	Meaning
edge_file	input file name for edges
label_file	input file name for node label
emb_file	input file name for node label
walk_depth	The step3 for random walk
opt1	The flag to open optimization 1 to reduce time cost
opt2	The flag to open optimization 2 to reduce time cost
w2v_emb_size	The dims of output the word2vec embedding
w2v_window_size	The context length of word2vec
w2v_epoch	The num of epoch to train the model.
train	The flag to run the struc2vec algorithm to get the w2v embedding
valid	The flag to use the w2v embedding to valid the classification result
num_class	The num of class in classification model to be trained

10.14.5 Experiment results

Dataset	Model	Metric	PGL Result	Paper repo Result
American airport dataset	Struc2vec without time cost optimization	ACC	0.6483	0.6340
American airport dataset	Struc2vec with optimization 1	ACC	0.6466	0.6242
American airport dataset	Struc2vec with optimization 2	ACC	0.6252	0.6241
American airport dataset	Struc2vec with optimization1&2	ACC	0.6226	0.6083

10.15 GATNE: General Attributed Multiplex HeTerogeneous Network Embedding

GATNE is a algorithms framework for embedding large-scale Attributed Multiplex Heterogeneous Networks(AMHN). Given a heterogeneous graph, which consists of nodes and edges of multiple types, it can learn continuous feature representations for every node. Based on PGL, we reproduce GATNE algorithm.

10.15.1 Datasets

YouTube dataset contains 2000 nodes, 1310617 edges and 5 edge types. And we use YouTube dataset for example.

You can dowload YouTube datasets from here

After downloading the data, put them, let's say, in ./data/ . Note that the current directory is the root directory of GATNE model. Then in ./data/youtube/ directory, there are three files:

- train.txt
- valid.txt
- test.txt

Then you can run the below command to preprocess the data.

```
python data_process.py --input_file ./data/youtube/train.txt --output_file ./data/
↪youtube/nodes.txt
```

10.15.2 Dependencies

- paddlepaddle>=1.6
- pgl>=1.0.0

10.15.3 Hyperparameters

All the hyper parameters are saved in config.yaml file. So before training GATNE model, you can open the config.yaml to modify the hyper parameters as you like.

for example, you can change the "use_cuda" to "True " in order to use GPU for training or modify "data_path" to use different dataset.

Some important hyper parameters in config.yaml:

- use_cuda: use GPU to train model
- data_path: the directory of dataset
- lr: learning rate
- neg_num: number of negatie samples.
- num_walks: number of walks started from each node
- walk_length: walk length

10.15.4 How to run

Then run the below command:

```
python main.py -c config.yaml
```

Experiment results

	PGL result	Reported result
AUC	84.83	84.61
PR	82.77	81.93
F1	76.98	76.83

10.16 metapath2vec: Scalable Representation Learning for Heterogeneous Networks

metapath2vec is a algorithm framework for representation learning in heterogeneous networks which contains multiple types of nodes and links. Given a heterogeneous graph, metapath2vec algorithm first generates meta-path-based

random walks and then use skipgram model to train a language model. Based on PGL, we reproduce metapath2vec algorithm.

10.16.1 Datasets

You can dowload datasets from here

We use the "aminer" data for example. After downloading the aminer data, put them, let's say, in ./data/net_aminer/ . We also need to put "label/" directory in ./data/.

10.16.2 Dependencies

- paddlepaddle>=1.6
- pgl>=1.0.0

10.16.3 Hyperparameters

All the hyper parameters are saved in config.yaml file. So before training, you can open the config.yaml to modify the hyper parameters as you like.

for example, you can change the "use_cuda" to "True " in order to use GPU for training or modify "data_path" to specify the data you want.

Some important hyper parameters in config.yaml:

- **use_cuda**: use GPU to train model
- **data_path**: the directory of dataset that you want to load
- **lr**: learning rate
- **neg_num**: number of negative samples.
- **num_walks**: number of walks started from each node
- **walk_length**: walk length
- **metapath**: meta path scheme

10.16.4 Metapath randomwalk sampling

Before training, we should generate some metapath random walks to train skipgram model. we can run the below command to produce metapath randomwalk data.

```
python sample.py -c config.yaml
```

10.16.5 Training and Testing

After finishing metapath randomwalk sampling, you can run the below command to train and test the model.

```
python main.py -c config.yaml

python multi_class.py --dataset ./data/out_aminer_CPAPC/author_label.txt --word2id ./
↪checkpoints/train.metapath2vec/word2id.pkl  --ckpt_path ./checkpoints/train.
↪metapath2vec/model_epoch5/
```

(continues on next page)

(continued from previous page)

10.16.6 Experiment results

train_percent	Metric	PGL Result	Reported Result
50%	macro-F1	0.9249	0.9314
50%	micro-F1	0.9283	0.9365

10.17 Unsupervised GraphSAGE in PGL

GraphSAGE is a general inductive framework that leverages node feature information (e.g., text attributes) to efficiently generate node embeddings for previously unseen data. Instead of training individual embeddings for each node, GraphSAGE learns a function that generates embeddings by sampling and aggregating features from a node's local neighborhood. Based on PGL, we reproduce GraphSAGE algorithm and reach the same level of indicators as the paper in Reddit Dataset. Besides, this is an example of subgraph sampling and training in PGL. For purpose of unsupervised learning, we use graph edges as positive samples for graphsage training.

10.17.1 Datasets(Quickstart)

The dataset `./sample.txt` is handcrafted bigraph for quick demo purpose, which format is `src \t dst`.

10.17.2 Dependencies

```
- paddlepaddle>=1.6
- pgl
```

10.17.3 How to run

1. Training

```
python train.py --data_path ./sample.txt --num_nodes 2000 --phase train
```

2. Predicting

```
python train.py --data_path ./sample.txt --num_nodes 2000 --phase predict
```

The resulted node embedding is stored in `emb.npy` file, which latter can be loaded using `np.load`.

Hyperparameters

- epoch: Number of epochs default (1)
- use_cuda: Use gpu if assign use_cuda.

- layer_type: We support 4 aggregator types including "graphsage_mean", "graphsage_maxpool", "graphsage_meanpool" and "graphsage_lstm".
- sample_workers: The number of workers for multiprocessing subgraph sample.
- lr: Learning rate.
- batch_size: Batch size.
- samples: The max neighbors sampling rate for each hop. (default: [10, 10])
- num_layers: The number of layer for graph sampling. (default: 2)
- hidden_size: The hidden size of the GraphSAGE models.
- checkpoint. Path for model checkpoint at each epoch. (default: 'model_ckpt')

10.18 API Reference

10.18.1 pgl.graph module: Graph Storage

This package implement Graph structure for handling graph data.

class pgl.graph.**Graph**(*num_nodes*, *edges=None*, *node_feat=None*, *edge_feat=None*)

Bases: object

Implementation of graph structure in pgl.

This is a simple implementation of graph structure in pgl.

Parameters

- **num_nodes** – number of nodes in a graph
- **edges** – list of (u, v) tuples
- **node_feat** (*optional*) – a dict of numpy array as node features
- **edge_feat** (*optional*) – a dict of numpy array as edge features (should have consistent order with edges)

Examples

```
import numpy as np
num_nodes = 5
edges = [ (0, 1), (1, 2), (3, 4)]
feature = np.random.randn(5, 100)
edge_feature = np.random.randn(3, 100)
graph = Graph(num_nodes=num_nodes,
            edges=edges,
            node_feat={
                "feature": feature
            },
            edge_feat={
                "edge_feature": edge_feature
            })
```

property adj_dst_index

Return an EdgeIndex object for dst.

property adj_src_index
Return an EdgeIndex object for src.

dump(*path*)

property edge_feat
Return a dictionary of edge features.

edge_feat_info()
Return the information of edge feature for GraphWrapper.

This function return the information of edge features. And this function is used to help constructing GraphWrapper

Returns A list of tuple (name, shape, dtype) for all given edge feature.

Examples

```
import numpy as np
num_nodes = 5
edges = [ (0, 1), (1, 2), (3, 4)]
feature = np.random.randn(3, 100)
graph = Graph(num_nodes=num_nodes,
        edges=edges,
        edge_feat={
            "feature": feature
        })
print(graph.edge_feat_info())
```

The output will be:

```
[("feature", [None, 100], "float32")]
```

property edges
Return all edges in numpy.ndarray with shape (num_edges, 2).

property graph_lod
Return Graph Lod Index for Paddle Computation

has_edges_between(*u*, *v*)
Check whether some edges is in graph.

Parameters

- **u** – a numpy.array of src nodes ID.
- **v** – a numpy.array of dst nodes ID.

Returns

A numpy.array of bool, with the same shape with *u* and *v*, exists[i] is True if (u[i], v[i]) is a edge in graph, Flase otherwise.

Return type exists

indegree(*nodes=None*)
Return the indegree of the given nodes

This function will return indegree of given nodes.

Parameters **nodes** – Return the indegree of given nodes, if nodes is None, return indegree for all nodes

Returns A numpy.ndarray as the given nodes' indegree.

node2vec_random_walk(*nodes*, *max_depth*, *p=1.0*, *q=1.0*)
Implement of node2vec stype random walk.

Reference paper: https://cs.stanford.edu/~jure/pubs/node2vec-kdd16.pdf.

Parameters

- **nodes** – Walk starting from nodes
- **max_depth** – Max walking depth
- **p** – Return parameter
- **q** – In-out parameter

Returns A list of walks.

node_batch_iter(*batch_size*, *shuffle=True*)
Node batch iterator

Iterate all node by batch.

Parameters

- **batch_size** – The batch size of each batch of nodes.
- **shuffle** – Whether shuffle the nodes.

Returns Batch iterator

property node_feat
Return a dictionary of node features.

node_feat_info()
Return the information of node feature for GraphWrapper.

This function return the information of node features. And this function is used to help constructing GraphWrapper

Returns A list of tuple (name, shape, dtype) for all given node feature.

Examples

```
import numpy as np
num_nodes = 5
edges = [ (0, 1), (1, 2), (3, 4)]
feature = np.random.randn(5, 100)
graph = Graph(num_nodes=num_nodes,
        edges=edges,
        node_feat={
            "feature": feature
        })
print(graph.node_feat_info())
```

The output will be:

```
[("feature", [None, 100], "float32")]
```

property nodes
Return all nodes id from 0 to `num_nodes - 1`

property num_edges
Return the number of edges.

property num_graph
Return Number of Graphs

property num_nodes
Return the number of nodes.

outdegree(*nodes=None*)
Return the outdegree of the given nodes.

This function will return outdegree of given nodes.

Parameters **nodes** – Return the outdegree of given nodes, if nodes is None, return outdegree for all nodes

Returns A numpy.array as the given nodes' outdegree.

predecessor(*nodes=None*, *return_eids=False*)
Find predecessor of given nodes.

This function will return the predecessor of given nodes.

Parameters

- **nodes** – Return the predecessor of given nodes, if nodes is None, return predecessor for all nodes.
- **return_eids** – If True return nodes together with corresponding eid

Returns Return a list of numpy.ndarray and each numpy.ndarray represent a list of predecessor ids for given nodes. If `return_eids=True`, there will be an additional list of numpy.ndarray and each numpy.ndarray represent a list of eids that connected nodes to their predecessors.

Example

```
import numpy as np
num_nodes = 5
edges = [ (0, 1), (1, 2), (3, 4)]
graph = Graph(num_nodes=num_nodes,
        edges=edges)
pred, pred_eid = graph.predecessor(return_eids=True)
```

This will give output.

```
pred:
      [[],
       [0],
       [1],
       [],
       [3]]

pred_eid:
      [[],
       [0],
       [1],
       [],
       [2]]
```

random_walk(*nodes*, *max_depth*)
Implement of random walk.

This function get random walks path for given nodes and depth.

Parameters

- **nodes** – Walk starting from nodes
- **max_depth** – Max walking depth

Returns A list of walks.

sample_edges(*sample_num*, *replace=False*)
Sample edges from the graph

This function helps to sample edges from all edges.

Parameters

- **sample_num** – The number of samples
- **replace** – boolean, Whether the sample is with or without replacement.

Returns (u, v), eid each is a numy.array with the same shape.

sample_nodes(*sample_num*)
Sample nodes from the graph

This function helps to sample nodes from all nodes. Nodes might be duplicated.

Parameters **sample_num** – The number of samples

Returns A list of nodes

sample_predecessor(*nodes*, *max_degree*, *return_eids=False*, *shuffle=False*)
Sample predecessor of given nodes.

Parameters

- **nodes** – Given nodes whose predecessor will be sampled.
- **max_degree** – The max sampled predecessor for each nodes.
- **return_eids** – Whether to return the corresponding eids.

Returns Return a list of numpy.ndarray and each numpy.ndarray represent a list of sampled predecessor ids for given nodes. If `return_eids=True`, there will be an additional list of numpy.ndarray and each numpy.ndarray represent a list of eids that connected nodes to their predecessors.

sample_successor(*nodes*, *max_degree*, *return_eids=False*, *shuffle=False*)
Sample successors of given nodes.

Parameters

- **nodes** – Given nodes whose successors will be sampled.
- **max_degree** – The max sampled successors for each nodes.
- **return_eids** – Whether to return the corresponding eids.

Returns Return a list of numpy.ndarray and each numpy.ndarray represent a list of sampled successor ids for given nodes. If `return_eids=True`, there will be an additional list of numpy.ndarray and each numpy.ndarray represent a list of eids that connected nodes to their successors.

sorted_edges(*sort_by='src'*)
Return sorted edges with different strategies.

This function will return sorted edges with different strategy. If `sort_by="src"`, then edges will be sorted by `src` nodes and otherwise `dst`.

Parameters **sort_by** – The type for sorted edges. ("src" or "dst")

Returns A tuple of (sorted_src, sorted_dst, sorted_eid).

subgraph(*nodes*, *eid=None*, *edges=None*, *edge_feats=None*, *with_node_feat=True*, *with_edge_feat=True*)
Generate subgraph with nodes and edge ids.

This function will generate a `pgl.graph.Subgraph` object and copy all corresponding node and edge features. Nodes and edges will be reindex from 0. Eid and edges can't both be None.

WARNING: ALL NODES IN EID MUST BE INCLUDED BY NODES

Parameters

- **nodes** – Node ids which will be included in the subgraph.
- **eid** (*optional*) – Edge ids which will be included in the subgraph.
- **edges** (*optional*) – Edge(src, dst) list which will be included in the subgraph.
- **with_node_feat** – Whether to inherit node features from parent graph.
- **with_edge_feat** – Whether to inherit edge features from parent graph.

Returns A `pgl.graph.Subgraph` object.

successor(*nodes=None*, *return_eids=False*)
Find successor of given nodes.

This function will return the successor of given nodes.

Parameters

- **nodes** – Return the successor of given nodes, if nodes is None, return successor for all nodes.
- **return_eids** – If True return nodes together with corresponding eid

Returns Return a list of numpy.ndarray and each numpy.ndarray represent a list of successor ids for given nodes. If `return_eids=True`, there will be an additional list of numpy.ndarray and each numpy.ndarray represent a list of eids that connected nodes to their successors.

Example

```
import numpy as np
num_nodes = 5
edges = [ (0, 1), (1, 2), (3, 4)]
graph = Graph(num_nodes=num_nodes,
        edges=edges)
succ, succ_eid = graph.successor(return_eids=True)
```

This will give output.

```
succ:
      [[1],
       [2],
       [],
       [4],
       []]

succ_eid:
      [[0],
       [1],
       [],
       [2],
       []]
```

class pgl.graph.**SubGraph**(*num_nodes*, *edges=None*, *node_feat=None*, *edge_feat=None*, *reindex=None*)

Bases: *pgl.graph.Graph*

Implementation of SubGraph in pgl.

Subgraph is inherit from `Graph`. The best way to construct subgraph is to use `Graph.subgraph` methods to generate Subgraph object.

Parameters

- **num_nodes** – number of nodes in a graph
- **edges** – list of (u, v) tuples
- **node_feat** (*optional*) – a dict of numpy array as node features
- **edge_feat** (*optional*) – a dict of numpy array as edge features (should have consistent order with edges)
- **reindex** – A dictionary that maps parent graph node id to subgraph node id.

reindex_from_parrent_nodes(*nodes*)

Map the given parent graph node id to subgraph id.

Parameters **nodes** – A list of nodes from parent graph.

Returns A list of subgraph ids.

reindex_to_parrent_nodes(*nodes*)

Map the given subgraph node id to parent graph id.

Parameters **nodes** – A list of nodes in this subgraph.

Returns A list of node ids in parent graph.

class pgl.graph.**MultiGraph**(*graph_list*)

Bases: *pgl.graph.Graph*

Implementation of multiple disjoint graph structure in pgl.

This is a simple implementation of graph structure in pgl.

Parameters **graph_list** – A list of Graph Instances

Examples

```
batch_graph = MultiGraph([graph1, graph2, graph3])
```

10.18.2 pgl.graph_wrapper module: Graph data holders for Paddle GNN.

This package provides interface to help building static computational graph for PaddlePaddle.

class pgl.graph_wrapper.**BaseGraphWrapper**
Bases: object

This module implement base class for graph wrapper.

Currently our PGL is developed based on static computational mode of paddle (we'll support dynamic computational model later). We need to build model upon a virtual data holder. BaseGraphWrapper provide a virtual graph structure that users can build deep learning models based on this virtual graph. And then feed real graph data to run the models. Moreover, we provide convenient message-passing interface (send & recv) for building graph neural networks.

NOTICE: Don't use this BaseGraphWrapper directly. Use `GraphWrapper` and `StaticGraphWrapper` to create graph wrapper instead.

property edge_feat
Return a dictionary of tensor representing edge features.

Returns A dictionary whose keys are the feature names and the values are feature tensor.

property edges
Return a tuple of edge Tensor (src, dst).

Returns A tuple of Tensor (src, dst). Src and dst are both tensor with shape (num_edges,) and dtype int64.

property graph_lod
Return graph index for graphs

Returns A variable with shape [None] as the Lod information of multiple-graph.

indegree()
Return the indegree tensor for all nodes.

Returns A tensor of shape (num_nodes,) in int64.

property node_feat
Return a dictionary of tensor representing node features.

Returns A dictionary whose keys are the feature names and the values are feature tensor.

property num_graph
Return a variable of number of graphs

Returns A variable with shape (1,) as the number of Graphs in int64.

property num_nodes
Return a variable of number of nodes

Returns A variable with shape (1,) as the number of nodes in int64.

recv(*msg*, *reduce_function*)
Recv message and aggregate the message by reduce_fucntion

The UDF reduce_function function should has the following format.

```
def reduce_func(msg):
    '''
        Args:
            msg: A LodTensor or a dictionary of LodTensor whose batch_size
                 is equals to the number of unique dst nodes.
```

(continues on next page)

(continued from previous page)

```
        Return:
            It should return a tensor with shape (batch_size, out_dims). The
            batch size should be the same as msg.
    '''
    pass
```

Parameters

- **msg** – A tensor or a dictionary of tensor created by send function..
- **reduce_function** – UDF reduce function or strings "sum" as built-in function. The built-in "sum" will use scatter_add to optimized the speed.

Returns A tensor with shape (num_nodes, out_dims). The output for nodes with no message will be zeros.

send(*message_func*, *nfeat_list=None*, *efeat_list=None*)

Send message from all src nodes to dst nodes.

The UDF message function should has the following format.

```
def message_func(src_feat, dst_feat, edge_feat):
    '''
        Args:
            src_feat: the node feat dict attached to the src nodes.
            dst_feat: the node feat dict attached to the dst nodes.
            edge_feat: the edge feat dict attached to the
                       corresponding (src, dst) edges.

        Return:
            It should return a tensor or a dictionary of tensor. And each␣
↪tensor
            should have a shape of (num_edges, dims).
    '''
    pass
```

Parameters

- **message_func** – UDF function.
- **nfeat_list** – a list of names or tuple (name, tensor)
- **efeat_list** – a list of names or tuple (name, tensor)

Returns A dictionary of tensor representing the message. Each of the values in the dictionary has a shape (num_edges, dim) which should be collected by `recv` function.

class pgl.graph_wrapper.**GraphWrapper**(*name*, *node_feat=[]*, *edge_feat=[]*, ***kwargs*)

Bases: *pgl.graph_wrapper.BaseGraphWrapper*

Implement a graph wrapper that creates a graph data holders that attributes and features in the graph are `fluid.layers.data`. And we provide interface `to_feed` to help converting `Graph` data into `feed_dict`.

Parameters

- **name** – The graph data prefix

- **node_feat** – A list of tuples that decribe the details of node feature tenosr. Each tuple mush be (name, shape, dtype) and the first dimension of the shape must be set unknown (-1 or None) or we can easily use `Graph.node_feat_info()` to get the node_feat settings.
- **edge_feat** – A list of tuples that decribe the details of edge feature tenosr. Each tuple mush be (name, shape, dtype) and the first dimension of the shape must be set unknown (-1 or None) or we can easily use `Graph.edge_feat_info()` to get the edge_feat settings.

Examples

```
import numpy as np
import paddle.fluid as fluid
from pgl.graph import Graph
from pgl.graph_wrapper import GraphWrapper

place = fluid.CPUPlace()
exe = fluid.Excecutor(place)

num_nodes = 5
edges = [ (0, 1), (1, 2), (3, 4)]
feature = np.random.randn(5, 100)
edge_feature = np.random.randn(3, 100)
graph = Graph(num_nodes=num_nodes,
            edges=edges,
            node_feat={
                "feature": feature
            },
            edge_feat={
                "edge_feature": edge_feature
            })

graph_wrapper = GraphWrapper(name="graph",
            node_feat=graph.node_feat_info(),
            edge_feat=graph.edge_feat_info())

# build your deep graph model
...

# Initialize parameters for deep graph model
exe.run(fluid.default_startup_program())

for i in range(10):
    feed_dict = graph_wrapper.to_feed(graph)
    ret = exe.run(fetch_list=[...], feed=feed_dict )
```

property holder_list
Return the holder list.

to_feed(*graph*)
Convert the graph into feed_dict.

This function helps to convert graph data into feed dict for `fluid.Excecutor` to run the model.

Parameters **graph** – the `Graph` data object

Returns A dictionary contains data holder names and its corresponding data.

class pgl.graph_wrapper.**StaticGraphWrapper**(*name*, *graph*, *place*)
Bases: *pgl.graph_wrapper.BaseGraphWrapper*

Implement a graph wrapper that the data of the graph won't be changed and it can be fit into the GPU or CPU memory. This can reduce the time of swapping large data from GPU and CPU.

Parameters

- **name** – The graph data prefix
- **graph** – The static graph that should be put into memory
- **place** – fluid.CPUPlace or fluid.CUDAPlace(n) indicating the device to hold the graph data.

Examples

If we have a immutable graph and it can be fit into the GPU or CPU. we can just use a `StaticGraphWrapper` to pre-place the graph data into devices.

```
import numpy as np
import paddle.fluid as fluid
from pgl.graph import Graph
from pgl.graph_wrapper import StaticGraphWrapper

place = fluid.CPUPlace()
exe = fluid.Excecutor(place)

num_nodes = 5
edges = [ (0, 1), (1, 2), (3, 4)]
feature = np.random.randn(5, 100)
edge_feature = np.random.randn(3, 100)
graph = Graph(num_nodes=num_nodes,
            edges=edges,
            node_feat={
                "feature": feature
            },
            edge_feat={
                "edge_feature": edge_feature
            })

graph_wrapper = StaticGraphWrapper(name="graph",
            graph=graph,
            place=place)

# build your deep graph model

# Initialize parameters for deep graph model
exe.run(fluid.default_startup_program())

# Initialize graph data
graph_wrapper.initialize(place)
```

initialize(*place*)

Placing the graph data into the devices.

Parameters **place** – fluid.CPUPlace or fluid.CUDAPlace(n) indicating the device to hold the graph data.

10.18.3 pgl.layers: Predefined graph neural networks layers.

Generate layers api

pgl.layers.**gcn**(*gw*, *feature*, *hidden_size*, *activation*, *name*, *norm=None*)
Implementation of graph convolutional neural networks (GCN)

This is an implementation of the paper SEMI-SUPERVISED CLASSIFICATION WITH GRAPH CONVOLUTIONAL NETWORKS (https://arxiv.org/pdf/1609.02907.pdf).

Parameters

- **gw** – Graph wrapper object (`StaticGraphWrapper` or `GraphWrapper`)
- **feature** – A tensor with shape (num_nodes, feature_size).
- **hidden_size** – The hidden size for gcn.
- **activation** – The activation for the output.
- **name** – Gcn layer names.
- **norm** – If `norm` is not None, then the feature will be normalized. Norm must be tensor with shape (num_nodes,) and dtype float32.

Returns A tensor with shape (num_nodes, hidden_size)

pgl.layers.**gat**(*gw*, *feature*, *hidden_size*, *activation*, *name*, *num_heads=8*, *feat_drop=0.6*, *attn_drop=0.6*, *is_test=False*)
Implementation of graph attention networks (GAT)

This is an implementation of the paper GRAPH ATTENTION NETWORKS (https://arxiv.org/abs/1710.10903).

Parameters

- **gw** – Graph wrapper object (`StaticGraphWrapper` or `GraphWrapper`)
- **feature** – A tensor with shape (num_nodes, feature_size).
- **hidden_size** – The hidden size for gat.
- **activation** – The activation for the output.
- **name** – Gat layer names.
- **num_heads** – The head number in gat.
- **feat_drop** – Dropout rate for feature.
- **attn_drop** – Dropout rate for attention.
- **is_test** – Whether in test phrase.

Returns A tensor with shape (num_nodes, hidden_size * num_heads)

pgl.layers.**gin**(*gw*, *feature*, *hidden_size*, *activation*, *name*, *init_eps=0.0*, *train_eps=False*)
Implementation of Graph Isomorphism Network (GIN) layer.

This is an implementation of the paper How Powerful are Graph Neural Networks? (https://arxiv.org/pdf/1810.00826.pdf).

In their implementation, all MLPs have 2 layers. Batch normalization is applied on every hidden layer.

Parameters

- **gw** – Graph wrapper object (`StaticGraphWrapper` or `GraphWrapper`)
- **feature** – A tensor with shape (num_nodes, feature_size).

- **name** – GIN layer names.
- **hidden_size** – The hidden size for gin.
- **activation** – The activation for the output.
- **init_eps** – float, optional Initial ϵ value, default is 0.
- **train_eps** – bool, optional if True, ϵ will be a learnable parameter.

Returns A tensor with shape (num_nodes, hidden_size).

class pgl.layers.**Set2Set**(*input_dim*, *n_iters*, *n_layers*)
Bases: object

Implementation of set2set pooling operator.

This is an implementation of the paper ORDER MATTERS: SEQUENCE TO SEQUENCE FOR SETS (https://arxiv.org/pdf/1511.06391.pdf).

forward(*feat*)

Parameters **feat** – input feature with shape [batch, n_edges, dim].

Returns output feature of set2set pooling with shape [batch, 2*dim].

Return type output_feat

pgl.layers.**graph_pooling**(*gw*, *node_feat*, *pool_type*)
Implementation of graph pooling

This is an implementation of graph pooling

Parameters

- **gw** – Graph wrapper object (StaticGraphWrapper or GraphWrapper)
- **node_feat** – A tensor with shape (num_nodes, feature_size).
- **pool_type** – The type of pooling (“sum”, “average” , “min”)

Returns A tensor with shape (num_graph, hidden_size)

pgl.layers.**graph_norm**(*gw*, *feature*)
Implementation of graph normalization

Reference Paper: BENCHMARKING GRAPH NEURAL NETWORKS

Each node features is divied by sqrt(num_nodes) per graphs.

Parameters

- **gw** – Graph wrapper object (StaticGraphWrapper or GraphWrapper)
- **feature** – A tensor with shape (num_nodes, hidden_size)

Returns A tensor with shape (num_nodes, hidden_size)

10.18.4 pgl.data_loader module: Some benchmark datasets.

This package implements some benchmark dataset for graph network and node representation learning.

class pgl.data_loader.**CitationDataset**(*name*, *symmetry_edges=True*, *self_loop=True*)
Bases: object

Citation dataset helps to create data for citation dataset (Pubmed and Citeseer)

Parameters

- **name** – The name for the dataset (“pubmed” or “citeseer”)
- **symmetry_edges** – Whether to create symmetry edges.
- **self_loop** – Whether to contain self loop edges.

graph
The Graph data object

y
Labels for each nodes

num_classes
Number of classes.

train_index
The index for nodes in training set.

val_index
The index for nodes in validation set.

test_index
The index for nodes in test set.

class pgl.data_loader.**CoraDataset**(*symmetry_edges=True*, *self_loop=True*)
Bases: object

Cora dataset implementation

Parameters

- **symmetry_edges** – Whether to create symmetry edges.
- **self_loop** – Whether to contain self loop edges.

graph
The Graph data object

y
Labels for each nodes

num_classes
Number of classes.

train_index
The index for nodes in training set.

val_index
The index for nodes in validation set.

test_index
The index for nodes in test set.

class pgl.data_loader.**ArXivDataset**(*np_random_seed=123*)
Bases: object

ArXiv dataset implementation

Parameters **np_random_seed** – The random seed for numpy.

graph
The Graph data object.

class pgl.data_loader.**BlogCatalogDataset**(*symmetry_edges=True*, *self_loop=False*)
Bases: object

BlogCatalog dataset implementation

Parameters

- **symmetry_edges** – Whether to create symmetry edges.
- **self_loop** – Whether to contain self loop edges.

graph
The `Graph` data object.

num_groups
Number of classes.

train_index
The index for nodes in training set.

test_index
The index for nodes in validation set.

10.18.5 pgl.utils.paddle_helper module: Some helper function for Paddle.

paddle_helper package contain some simple function to help building paddle models.

`pgl.utils.paddle_helper.`**constant**(*name*, *value*, *dtype*, *hide_batch_size=True*)
Create constant variable with given data.

This function helps to create constants variable with given numpy.ndarray data.

Parameters

- **name** – variable name
- **value** – numpy.ndarray the value of constant
- **dtype** – the type of constant
- **hide_batch_size** – If set the first dimenstion as unknown, the explicit batch size may cause some error in paddle. For example, when the value has a shape of (batch_size, dim1, dim2), it will return a variable with shape (-1, dim1, dim2).

Returns A tuple contain the constant variable and the constant variable initialize function.

Examples

```
import paddle.fluid as fluid
place = fluid.CPUPlace()
exe = fluid.Executor(place)
constant_var, constant_var_init = constant(name="constant",
                  value=np.array([5.0],
                  dtype="float32"))
exe.run(fluid.default_startup_program())
# Run After default startup
constant_var_init(place)
```

`pgl.utils.paddle_helper.`**gather**(*input*, *index*)
Gather input from given index.

Slicing input data with given index. This function rewrite paddle.fluid.layers.gather to fix issue: https://github.com/PaddlePaddle/Paddle/issues/17509 when paddlepaddle's version is less than 1.5.

Parameters

- **input** – Input tensor to be sliced
- **index** – Slice index

Returns A tensor that are sliced from given input data.

pgl.utils.paddle_helper.**lod_constant**(*name*, *value*, *lod*, *dtype*)
Create constant lod variable with given data,

This function helps to create constants lod variable with given numpy.ndarray data and lod information.

Parameters

- **name** – variable name
- **value** – numpy.ndarray the value of constant
- **dtype** – the type of constant
- **lod** – lod infos of given value.

Returns A tuple contain the constant variable and the constant variable initialize function.

Examples

```
import paddle.fluid as fluid
place = fluid.CPUPlace()
exe = fluid.Executor(place)
constant_var, constant_var_init = lod_constant(name="constant",
                  value=np.array([[5.0], [1.0], [2.0]],
                  lod=[2, 1],
                  dtype="float32"))
exe.run(fluid.default_startup_program())
# Run After default startup
constant_var_init(place)
```

pgl.utils.paddle_helper.**scatter_add**(*input*, *index*, *updates*)
Scatter add updates to input by given index.

Adds sparse updates to input variables.

Parameters

- **input** – Input tensor to be updated
- **index** – Slice index
- **updates** – Must have same type as input.

Returns Same type and shape as input.

pgl.utils.paddle_helper.**scatter_max**(*input*, *index*, *updates*)
Scatter max updates to input by given index.

Adds sparse updates to input variables.

Parameters

- **input** – Input tensor to be updated
- **index** – Slice index
- **updates** – Must have same type as input.

Returns Same type and shape as input.

pgl.utils.paddle_helper.**sequence_softmax**(*x*)
Compute sequence softmax over paddle LodTensor

This function compute softmax normalization along with the length of sequence. This function is an extention of `fluid.layers.sequence_softmax` which can only deal with LodTensor whose last dimension is 1.

Parameters **x** – The input variable which is a LodTensor.

Returns Output of sequence_softmax

10.18.6 pgl.utils.mp_reader module: MultiProcessing reader helper function for Paddle.

Optimized Multiprocessing Reader for PaddlePaddle

pgl.utils.mp_reader.**deserialize_data**(*data*)

pgl.utils.mp_reader.**log = <Logger pgl.utils.mp_reader (DEBUG)>**

pgl.utils.mp_reader.**multiprocess_reader**(*readers*, *use_pipe=True*, *queue_size=1000*, *pipe_size=10*)
multiprocess_reader use python multi process to read data from readers and then use multiprocess.Queue or multiprocess.Pipe to merge all data. The process number is equal to the number of input readers, each process call one reader. Multiprocess.Queue require the rw access right to /dev/shm, some platform does not support. you need to create multiple readers first, these readers should be independent to each other so that each process can work independently. An example: .. code-block:: python

reader0 = reader(["file01", "file02"]) reader1 = reader(["file11", "file12"]) reader1 = reader(["file21", "file22"]) reader = multiprocess_reader([reader0, reader1, reader2],

queue_size=100, use_pipe=False)

pgl.utils.mp_reader.**numpy_deserialize_data**(*data*)
deserialize_data

pgl.utils.mp_reader.**numpy_serialize_data**(*data*)
serialize_data

pgl.utils.mp_reader.**serialize_data**(*data*)

10.18.7 pgl.heter_graph module: Heterogenous Graph Storage

This package implement Heterogeneous Graph structure for handling Heterogeneous graph data.

class pgl.heter_graph.**HeterGraph**(*num_nodes*, *edges*, *node_types=None*, *node_feat=None*, *edge_feat=None*)
Bases: `object`

Implementation of heterogeneous graph structure in pgl

This is a simple implementation of heterogeneous graph structure in pgl.

Parameters

- **num_nodes** – number of nodes in a heterogeneous graph
- **edges** – dict, every element in dict is a list of (u, v) tuples.
- **node_types** (*optional*) – list of (u, node_type) tuples to specify the node type of every node

- **node_feat** (*optional*) – a dict of numpy array as node features
- **edge_feat** (*optional*) – a dict of dict as edge features for every edge type

Examples

```
import numpy as np
num_nodes = 4
node_types = [(0, 'user'), (1, 'item'), (2, 'item'), (3, 'user')]
edges = {
    'edges_type1': [(0,1), (3,2)],
    'edges_type2': [(1,2), (3,1)],
}
node_feat = {'feature': np.random.randn(4, 16)}
edges_feat = {
    'edges_type1': {'h': np.random.randn(2, 16)},
    'edges_type2': {'h': np.random.randn(2, 16)},
}

g = heter_graph.HeterGraph(
                num_nodes=num_nodes,
                edges=edges,
                node_types=node_types,
                node_feat=node_feat,
                edge_feat=edges_feat)
```

property edge_feat
Return edge features of all edge types.

edge_feat_info()
Return the information of edge feature for HeterGraphWrapper.

This function return the information of edge features of all edge types. And this function is used to help constructing HeterGraphWrapper

Returns A dict of list of tuple (name, shape, dtype) for all given edge feature.

property edge_types
Return a list of edge types.

edge_types_info()
Return the information of all edge types.

Returns A list of all edge types.

indegree(*nodes=None*, *edge_type=None*)
Return the indegree of the given nodes with the specified edge_type.

Parameters

- **nodes** – Return the indegree of given nodes. if nodes is None, return indegree for all nodes.
- **edge_types** – Return the indegree with specified edge_type. if edge_type is None, return the total indegree of the given nodes.

Returns A numpy.ndarray as the given nodes' indegree.

node_batch_iter(*batch_size*, *shuffle=True*, *n_type=None*)
Node batch iterator

Iterate all nodes by batch with the specified node type.

Parameters

- **batch_size** – The batch size of each batch of nodes.
- **shuffle** – Whether shuffle the nodes.
- **n_type** – Iterate the nodes with the specified node type. If n_type is None, iterate all nodes by batch.

Returns Batch iterator

property node_feat
Return a dictionary of node features.

node_feat_info()
Return the information of node feature for HeterGraphWrapper.

This function return the information of node features of all node types. And this function is used to help constructing HeterGraphWrapper

Returns A list of tuple (name, shape, dtype) for all given node feature.

property node_types
Return the node types.

property nodes
Return all nodes id from 0 to `num_nodes - 1`

property num_edges
Return edges number of all edge types.

property num_nodes
Return the number of nodes.

num_nodes_by_type(*n_type=None*)
Return the number of nodes with the specified node type.

outdegree(*nodes=None*, *edge_type=None*)
Return the outdegree of the given nodes with the specified edge_type.

Parameters

- **nodes** – Return the outdegree of given nodes, if nodes is None, return outdegree for all nodes
- **edge_types** – Return the outdegree with specified edge_type. if edge_type is None, return the total outdegree of the given nodes.

Returns A numpy.array as the given nodes' outdegree.

predecessor(*edge_type*, *nodes=None*, *return_eids=False*)
Find predecessor of given nodes with the specified edge_type.

Parameters

- **nodes** – Return the predecessor of given nodes, if nodes is None, return predecessor for all nodes
- **edge_types** – Return the predecessor with specified edge_type.
- **return_eids** – If True return nodes together with corresponding eid

sample_nodes(*sample_num*, *n_type=None*)
Sample nodes with the specified n_type from the graph

This function helps to sample nodes with the specified n_type from the graph. If n_type is None, this function will sample nodes from all nodes. Nodes might be duplicated.

Parameters

- **sample_num** – The number of samples
- **n_type** – The nodes of type to be sampled

Returns A list of nodes

sample_predecessor(*edge_type*, *nodes*, *max_degree*, *return_eids=False*, *shuffle=False*)
Sample predecessors of given nodes with the specified edge_type.

Parameters

- **edge_type** – The specified edge_type.
- **nodes** – Given nodes whose predecessors will be sampled.
- **max_degree** – The max sampled predecessors for each nodes.
- **return_eids** – Whether to return the corresponding eids.

Returns Return a list of numpy.ndarray and each numpy.ndarray represent a list of sampled predecessor ids for given nodes with specified edge type. If `return_eids=True`, there will be an additional list of numpy.ndarray and each numpy.ndarray represent a list of eids that connected nodes to their predecessors.

sample_successor(*edge_type*, *nodes*, *max_degree*, *return_eids=False*, *shuffle=False*)
Sample successors of given nodes with the specified edge_type.

Parameters

- **edge_type** – The specified edge_type.
- **nodes** – Given nodes whose successors will be sampled.
- **max_degree** – The max sampled successors for each nodes.
- **return_eids** – Whether to return the corresponding eids.

Returns Return a list of numpy.ndarray and each numpy.ndarray represent a list of sampled successor ids for given nodes with specified edge type. If `return_eids=True`, there will be an additional list of numpy.ndarray and each numpy.ndarray represent a list of eids that connected nodes to their successors.

successor(*edge_type*, *nodes=None*, *return_eids=False*)
Find successor of given nodes with the specified edge_type.

Parameters

- **nodes** – Return the successor of given nodes, if nodes is None, return successor for all nodes
- **edge_types** – Return the successor with specified edge_type. if edge_type is None, return the total successor of the given nodes and eids are invalid in this way.
- **return_eids** – If True return nodes together with corresponding eid

class pgl.heter_graph.**SubHeterGraph**(*num_nodes*, *edges*, *node_types=None*, *node_feat=None*, *edge_feat=None*, *reindex=None*)

Bases: *pgl.heter_graph.HeterGraph*

Implementation of SubHeterGraph in pgl.

SubHeterGraph is inherit from `HeterGraph`.

Parameters

- **num_nodes** – number of nodes in a heterogeneous graph
- **edges** – dict, every element in dict is a list of (u, v) tuples.
- **node_types** (*optional*) – list of (u, node_type) tuples to specify the node type of every node
- **node_feat** (*optional*) – a dict of numpy array as node features
- **edge_feat** (*optional*) – a dict of dict as edge features for every edge type
- **reindex** – A dictionary that maps parent hetergraph node id to subhetergraph node id.

reindex_from_parrent_nodes(*nodes*)
Map the given parent graph node id to subgraph id.

Parameters **nodes** – A list of nodes from parent graph.

Returns A list of subgraph ids.

reindex_to_parrent_nodes(*nodes*)
Map the given subgraph node id to parent graph id.

Parameters **nodes** – A list of nodes in this subgraph.

Returns A list of node ids in parent graph.

10.18.8 pgl.heter_graph_wrapper module: Heterogenous Graph data holders for Paddle GNN.

This package provides interface to help building static computational graph for PaddlePaddle.

class pgl.heter_graph_wrapper.**HeterGraphWrapper**(*name*, *edge_types*, *node_feat={}*, *edge_feat={}*, ***kwargs*)

Bases: object

Implement a heterogeneous graph wrapper that creates a graph data holders that attributes and features in the heterogeneous graph. And we provide interface to_feed to help converting Graph data into feed_dict.

Parameters

- **name** – The heterogeneous graph data prefix
- **node_feat** – A dict of list of tuples that decribe the details of node feature tenosr. Each tuple mush be (name, shape, dtype) and the first dimension of the shape must be set unknown (-1 or None) or we can easily use HeterGraph.node_feat_info() to get the node_feat settings.
- **edge_feat** – A dict of list of tuples that decribe the details of edge feature tenosr. Each tuple mush be (name, shape, dtype) and the first dimension of the shape must be set unknown (-1 or None) or we can easily use HeterGraph.edge_feat_info() to get the edge_feat settings.

Examples

```
import paddle.fluid as fluid
import numpy as np
from pgl import heter_graph
from pgl import heter_graph_wrapper
```

(continues on next page)

(continued from previous page)

```
num_nodes = 4
node_types = [(0, 'user'), (1, 'item'), (2, 'item'), (3, 'user')]
edges = {
    'edges_type1': [(0,1), (3,2)],
    'edges_type2': [(1,2), (3,1)],
}
node_feat = {'feature': np.random.randn(4, 16)}
edges_feat = {
    'edges_type1': {'h': np.random.randn(2, 16)},
    'edges_type2': {'h': np.random.randn(2, 16)},
}

g = heter_graph.HeterGraph(
                num_nodes=num_nodes,
                edges=edges,
                node_types=node_types,
                node_feat=node_feat,
                edge_feat=edges_feat)

gw = heter_graph_wrapper.HeterGraphWrapper(
                    name='heter_graph',
                    edge_types = g.edge_types_info(),
                    node_feat=g.node_feat_info(),
                    edge_feat=g.edge_feat_info())
```

to_feed(*heterGraph*, *edge_types_list='__ALL__'*)

Convert the graph into feed_dict.

This function helps to convert graph data into feed dict for `fluid.Excecutor` to run the model.

Parameters

- **heterGraph** – the `HeterGraph` data object
- **edge_types_list** – the edge types list to be fed

Returns A dictinary contains data holder names and its coresponding data.

CHAPTER

ELEVEN

THE TEAM

11.1 The Team

PGL is developed and maintained by NLP and Paddle Teams at Baidu

PGL is developed and maintained by NLP and Paddle Teams at Baidu

CHAPTER

TWELVE

LICENSE

PGL uses Apache License 2.0.

PYTHON MODULE INDEX

p

INDEX

N

O

P

R

S

T

V

Y