pgl

Release 2.0.0a

PaddlePaddle

Jan 28, 2021

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.

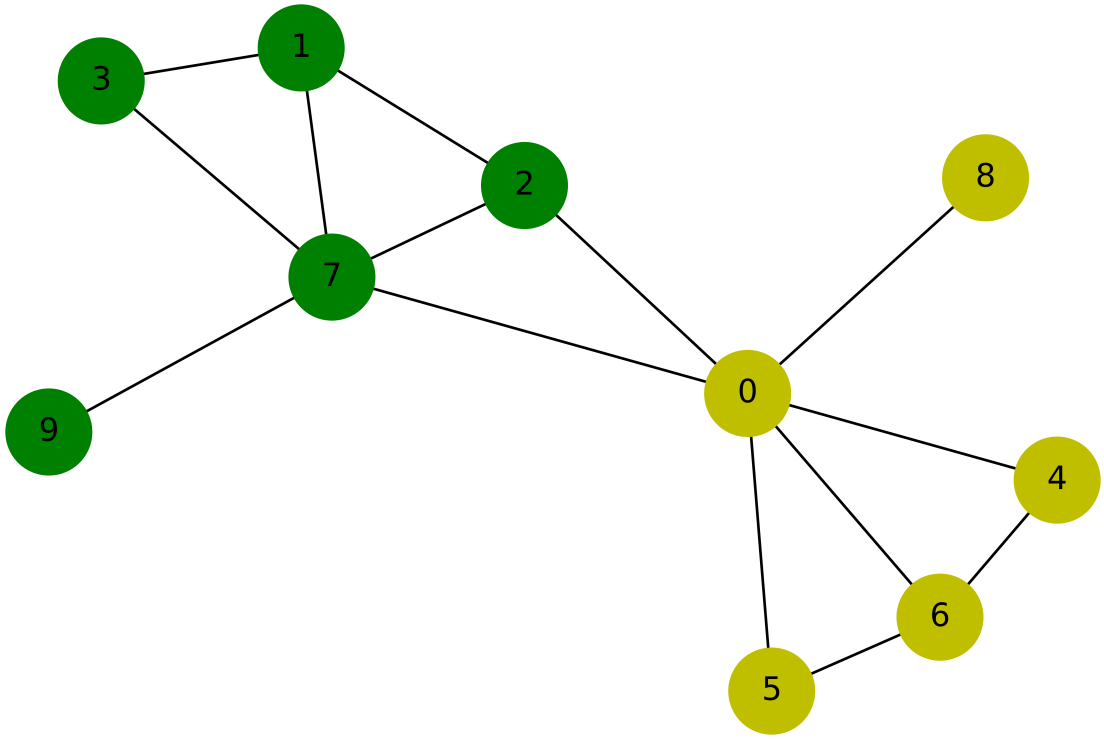
```
paddlepaddle >= 2.0.0rc
cython
```

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(),
                        edge_feat=g.edge_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
```

(continues on next page)

(continued from previous page)

```
    output = gw.recv(msg, recv_func)
    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 Graph Isomorphism Network (GIN)

Graph Isomorphism Network (GIN) is a simple graph neural network that expects to achieve the ability as the Weisfeiler-Lehman graph isomorphism test. Based on PGL, we reproduce the GIN model.

10.2.1 Datasets

The dataset can be downloaded from here. After downloading the datauncompress them, then a directory named `./dataset/` can be found in current directory. Note that the current directory is the root directory of GIN model.

10.2.2 Dependencies

- paddlepaddle >= 2.0.0
- pgl >= 2.0

10.2.3 How to run

For examples, use GPU to train GIN model on MUTAG dataset.

```
export CUDA_VISIBLE_DEVICES=0
python main.py --use_cuda --dataset_name MUTAG  --data_path ./dataset
```

10.2.4 Hyperparameters

- data_path: the root path of your dataset
- dataset_name: the name of the dataset
- fold_idx: The $fold\_idx^{th}$ fold of dataset splited. Here we use 10 fold cross-validation
- train_eps: whether the ϵ parameter is learnable.

10.2.5 Experiment results Accuracy

	MUTAG	COLLAB	IMDBBINARY	IMDBMULTI
PGL result	90.8	78.6	76.8	50.8
paper reuslt	90.0	80.0	75.1	52.3

10.3 Easy Paper Reproduction for Citation Network (Cora / Pubmed / Citeseer)

This page tries to reproduce all the **Graph Neural Network** paper for Citation Network (Cora/Pubmed/Citeseer), which is the **Hello world** dataset (**small** and **fast**) for graph neural networks. But it's very hard to achieve very high performance.

All datasets are runned with public split of **semi-supervised** settings. And we report the averarge accuracy by running 10 times.

10.3.1 Experiment Results

Model	Cora	Pubmed	Citeseer	Remarks
Vanilla GCN (Kipf 2017)	0.807(0.010)	0.794(0.003)	0.710(0.007)	•
GAT (Veličković 2017)	0.834(0.004)	0.772(0.004)	0.700(0.006)	•
SGC(Wu 2019)	0.818(0.000)	0.782(0.000)	0.708(0.000)	•
APPNP (Johannes 2018)	0.846(0.003)	0.803(0.002)	0.719(0.003)	Almost the same with the results reported in Appendix E.
GCNII (64 Layers, 1500 Epochs, Chen 2020)	0.846(0.003)	0.798(0.003)	0.724(0.006)	•

10.3.2 How to run the experiments?

```
# Device choose
export CUDA_VISIBLE_DEVICES=0
# GCN
python train.py --conf config/gcn.yaml --use_cuda --dataset cora
python train.py --conf config/gcn.yaml --use_cuda --dataset pubmed
python train.py --conf config/gcn.yaml --use_cuda --dataset citeseer


# GAT
python train.py --conf config/gat.yaml --use_cuda --dataset cora
python train.py --conf config/gat.yaml --use_cuda --dataset pubmed
python train.py --conf config/gat.yaml --use_cuda --dataset citeseer


# SGC (Slow version)
python train.py --conf config/sgc.yaml --use_cuda --dataset cora
python train.py --conf config/sgc.yaml --use_cuda --dataset pubmed
python train.py --conf config/sgc.yaml --use_cuda --dataset citeseer

# APPNP
python train.py --conf config/appnp.yaml --use_cuda --dataset cora
```

(continues on next page)

(continued from previous page)

```
python train.py --conf config/appnp.yaml --use_cuda --dataset pubmed
python train.py --conf config/appnp.yaml --use_cuda --dataset citeseer

# GCNII (The original code use 1500 epochs.)
python train.py --conf config/gcnii.yaml --use_cuda --dataset cora --epoch 1500
python train.py --conf config/gcnii.yaml --use_cuda --dataset pubmed --epoch 1500
python train.py --conf config/gcnii.yaml --use_cuda --dataset citeseer --epoch 1500
```

10.4 API Reference

10.4.1 pgl.graph: Graph Storage

This package implement Graph structure for handling graph data.

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

Bases: object

Implementation of graph interface in pgl.

This is a simple implementation of graph structure in pgl. *pgl.Graph* is alias on *pgl.graph.Graph*

Parameters

- **edges** – list of (u, v) tuples, 2D numpy.ndarry or 2D paddle.Tensor
- **(optional** (*num_nodes*) – int, numpy or paddle.Tensor): Number of nodes in a graph. If not provided, the number of nodes will be infered from edges.
- **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 1:

- Create a graph with numpy.
- Convert it into paddle.Tensor .
- Do send recv for graph neural network.

```
import numpy as np
import pgl

num_nodes = 5
edges = [ (0, 1), (1, 2), (3, 4)]
feature = np.random.randn(5, 100).astype(np.float32)
edge_feature = np.random.randn(3, 100).astype(np.float32)
graph = pgl.Graph(num_nodes=num_nodes,
            edges=edges,
            node_feat={
                "feature": feature
            },
            edge_feat={
                "edge_feature": edge_feature
            })
graph.tensor()
```

(continues on next page)

(continued from previous page)

```
model = pgl.nn.GCNConv(100, 100)
out = model(graph, graph.node_feat["feature"])
```

Examples 2:

- Create a graph with paddle.Tensor.
- Do send recv for graph neural network.

```
import paddle
import pgl

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

model = pgl.nn.GCNConv(100, 100)
out = model(graph, graph.node_feat["feature"])
```

property adj_dst_index
Return an EdgeIndex object for dst.

property adj_src_index
Return an EdgeIndex object for src.

static batch(*graph_list*)
This is alias on *pgl.Graph.disjoint* with merged_graph_index=False

classmethod disjoint(*graph_list*, *merged_graph_index=False*)
This method disjoint list of graph into a big graph.

Parameters

- **graph_list** (*Graph List*) – A list of Graphs.
- **merged_graph_index** – whether to keeped the graph_id that the nodes belongs to.

```
import numpy as np
import pgl

num_nodes = 5
edges = [ (0, 1), (1, 2), (3, 4)]
graph = pgl.Graph(num_nodes=num_nodes,
            edges=edges)
joint_graph = pgl.Graph.disjoint([graph, graph], merged_graph_index=False)
print(joint_graph.graph_node_id)
>>> [0, 0, 0, 0, 0, 1, 1, 1, 1 ,1]
print(joint_graph.num_graph)
>>> 2
```

(continues on next page)

(continued from previous page)

```
joint_graph = pgl.Graph.disjoint([graph, graph], merged_graph_index=True)
print(joint_graph.graph_node_id)
>>> [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
print(joint_graph.num_graph)
>>> 1
```

dump (*path*)

Dump the graph into a directory.

This function will dump the graph information into the given directory path. The graph can be read back with `pgl.Graph.load`

Parameters **path** – The directory for the storage of the graph.

property edge_feat

Return a dictionary of edge features.

property edges

Return all edges in numpy.ndarray or paddle.Tensor with shape (num_edges, 2).

property graph_edge_id

Return a numpy.ndarray or paddle.Tensor with shape [num_edges] that indicates which graph the edges belongs to.

Examples:

```
import numpy as np
import pgl

num_nodes = 5
edges = [ (0, 1), (1, 2), (3, 4)]
graph = pgl.Graph(num_nodes=num_nodes,
            edges=edges)
joint_graph = pgl.Graph.batch([graph, graph])
print(joint_graph.graph_edge_id)

>>> [0, 0, 0, 1, 1, 1]
```

property graph_node_id

Return a numpy.ndarray or paddle.Tensor with shape [num_nodes] that indicates which graph the nodes belongs to.

Examples:

```
import numpy as np
import pgl

num_nodes = 5
edges = [ (0, 1), (1, 2), (3, 4)]
graph = pgl.Graph(num_nodes=num_nodes,
            edges=edges)
joint_graph = pgl.Graph.batch([graph, graph])
print(joint_graph.graph_node_id)

>>> [0, 0, 0, 0, 0, 1, 1, 1, 1 ,1]
```

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 or paddle.Tensor as the given nodes' indegree.

is_tensor()
Return whether the Graph is in paddle.Tensor or numpy format.

classmethod load(*path*, *mmap_mode='r'*)
Load Graph from path and return a Graph in numpy.

Parameters

- **path** – The directory path of the stored Graph.
- **mmap_mode** – Default `mmap_mode="r"`. If not None, memory-map the graph.

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.

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.

numpy(*inplace=True*)
Convert the Graph into numpy format.

In numpy format, the graph edges and node features are in numpy.ndarray format. But you can't use send and recv in numpy graph.

Parameters **inplace** – (Default True) Whether to convert the graph into numpy inplace.

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 or paddle.Tensor 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
import pgl

num_nodes = 5
edges = [ (0, 1), (1, 2), (3, 4)]
graph = pgl.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]]
```

recv(*reduce_func*, *msg*, *recv_mode='dst'*)
Recv message and aggregate the message by reduce_func

The UDF reduce_func 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.

        Return:
```

(continues on next page)

(continued from previous page)

```
            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_func** – A callable UDF reduce function.

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

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.

send(*message_func*, *src_feat=None*, *dst_feat=None*, *edge_feat=None*, *node_feat=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
```

(continues on next page)

(continued from previous page)

```
        should have a shape of (num_edges, dims).
    '''
    return {'msg': src_feat['h']}
```

Parameters

- **message_func** – UDF function.
- **src_feat** – a dict {name: tensor,} to build src node feat
- **dst_feat** – a dict {name: tensor,} to build dst node feat
- **node_feat** – a dict {name: tensor,} to build both src and dst node feat
- **edge_feat** – a dict {name: tensor,} to build edge feat

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.

send_recv(*feature*, *reduce_func='sum'*)

This method combines the send and recv function.

Now, this method only supports default copy send function, and built-in receive function ('sum', 'mean', 'max', 'min').

Parameters

- **feature** (*Tensor | Tensor List*) – the node feature of a graph.
- **reduce_func** (*str*) – 'sum', 'mean', 'max', 'min' built-in receive function.

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).

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
import pgl

num_nodes = 5
edges = [ (0, 1), (1, 2), (3, 4)]
graph = pgl.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],
       []]
```

tensor(*inplace=True*)
Convert the Graph into paddle.Tensor format.

In paddle.Tensor format, the graph edges and node features are in paddle.Tensor format. You can use send and recv in paddle.Tensor graph.

Parameters **inplace** – (Default True) Whether to convert the graph into tensor inplace.

to_mmap(*path='./tmp'*)
Turn the Graph into Memmap mode which can share memory between processes.

10.4.2 pgl.sampling

Graph Sampling Function

pgl.sampling.**graphsage_sample**(*graph*, *nodes*, *samples*, *ignore_edges=[]*)
Implement of graphsage sample. Reference paper: https://cs.stanford.edu/people/jure/pubs/graphsage-nips17.pdf. :param graph: A pgl graph instance :param nodes: Sample starting from nodes :param samples: A list, number of neighbors in each layer :param ignore_edges: list of edge(src, dst) will be ignored.

Returns A list of subgraphs

pgl.sampling.**random_walk**(*graph*, *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.

pgl.sampling.**subgraph**(*graph*, *nodes*, *eid=None*, *edges=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` object.

10.4.3 pgl.nn: Predefined graph neural networks layers.

Graph Convolution Layers

This package implements common layers to help building graph neural networks.

class pgl.nn.conv.**GCNConv**(*input_size*, *output_size*, *activation=None*, *norm=True*)

Bases: `paddle.fluid.dygraph.layers.Layer`

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

- **input_size** – The size of the inputs.
- **output_size** – The size of outputs
- **activation** – The activation for the output.
- **norm** – If `norm` is True, then the feature will be normalized.

forward(*graph*, *feature*, *norm=None*)

Parameters

- **graph** – *pgl.Graph* instance.
- **feature** – A tensor with shape (num_nodes, input_size)
- **norm** – (default None). If `norm` is not None, then the feature will be normalized by given norm. If `norm` is None and `self.norm` is *true*, then we use *lapacian degree norm*.

Returns A tensor with shape (num_nodes, output_size)

class pgl.nn.conv.**GATConv**(*input_size*, *hidden_size*, *feat_drop=0.6*, *attn_drop=0.6*, *num_heads=1*, *concat=True*, *activation=None*)

Bases: `paddle.fluid.dygraph.layers.Layer`

Implementation of graph attention networks (GAT)

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

Parameters

- **input_size** – The size of the inputs.
- **hidden_size** – The hidden size for gat.
- **activation** – (default None) The activation for the output.
- **num_heads** – (default 1) The head number in gat.
- **feat_drop** – (default 0.6) Dropout rate for feature.
- **attn_drop** – (default 0.6) Dropout rate for attention.
- **concat** – (default True) Whether to concat output heads or average them.

forward(*graph*, *feature*)

Parameters

- **graph** – *pgl.Graph* instance.
- **feature** – A tensor with shape (num_nodes, input_size)

Returns If *concat=True* then return a tensor with shape (num_nodes, hidden_size), else return a tensor with shape (num_nodes, hidden_size * num_heads)

class pgl.nn.conv.**APPNP**(*alpha=0.2*, *k_hop=10*)

Bases: paddle.fluid.dygraph.layers.Layer

Implementation of APPNP of "Predict then Propagate: Graph Neural Networks meet Personalized PageRank" (ICLR 2019).

Parameters

- **k_hop** – K Steps for Propagation
- **alpha** – The hyperparameter of alpha in the paper.

Returns A tensor with shape (num_nodes, hidden_size)

forward(*graph*, *feature*, *norm=None*)

Parameters

- **graph** – *pgl.Graph* instance.
- **feature** – A tensor with shape (num_nodes, input_size)
- **norm** – (default None). If norm is not None, then the feature will be normalized by given norm. If norm is None, then we use *lapacian degree norm*.

Returns A tensor with shape (num_nodes, output_size)

class pgl.nn.conv.**GCNII**(*hidden_size*, *activation=None*, *lambda_l=0.5*, *alpha=0.2*, *k_hop=10*, *dropout=0.6*)

Bases: paddle.fluid.dygraph.layers.Layer

Implementation of GCNII of "Simple and Deep Graph Convolutional Networks"

paper: https://arxiv.org/pdf/2007.02133.pdf

Parameters

- **hidden_size** – The size of inputs and outputs.
- **activation** – The activation for the output.
- **k_hop** – Number of layers for gcnii.
- **lambda_l** – The hyperparameter of lambda in the paper.

- **alpha** – The hyperparameter of alpha in the paper.
- **dropout** – Feature dropout rate.

forward(*graph*, *feature*, *norm=None*)

Parameters

- **graph** – *pgl.Graph* instance.
- **feature** – A tensor with shape (num_nodes, input_size)
- **norm** – (default None). If `norm` is not None, then the feature will be normalized by given norm. If `norm` is None, then we use *lapacian degree norm*.

Returns A tensor with shape (num_nodes, output_size)

class pgl.nn.conv.**TransformerConv**(*input_size*, *hidden_size*, *num_heads=4*, *feat_drop=0.6*, *attn_drop=0.6*, *concat=True*, *skip_feat=True*, *gate=False*, *layer_norm=True*, *activation='relu'*)

Bases: `paddle.fluid.dygraph.layers.Layer`

forward(*graph*, *feature*, *edge_feat=None*)

Defines the computation performed at every call. Should be overridden by all subclasses.

Parameters

- ***inputs** (*tuple*) – unpacked tuple arguments
- ****kwargs** (*dict*) – unpacked dict arguments

reduce_attention(*msg*)

send_attention(*src_feat*, *dst_feat*, *edge_feat*)

send_recv(*graph*, *q*, *k*, *v*, *edge_feat*)

class pgl.nn.conv.**GINConv**(*input_size*, *output_size*, *activation=None*, *init_eps=0.0*, *train_eps=False*)

Bases: `paddle.fluid.dygraph.layers.Layer`

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

- **input_size** – The size of input.
- **output_size** – The size of output.
- **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.

forward(*graph*, *feature*)

Parameters

- **graph** – *pgl.Graph* instance.
- **feature** – A tensor with shape (num_nodes, input_size)

Returns A tensor with shape (num_nodes, output_size)

class pgl.nn.conv.**GraphSageConv**(*input_size*, *hidden_size*, *aggr_func='sum'*)
Bases: paddle.fluid.dygraph.layers.Layer

GraphSAGE is a general inductive framework that leverages node feature information (e.g., text attributes) to efficiently generate node embeddings for previously unseen data.

Paper reference: Hamilton, Will, Zhitao Ying, and Jure Leskovec. "Inductive representation learning on large graphs." Advances in neural information processing systems. 2017.

Parameters

- **input_size** – The size of the inputs.
- **hidden_size** – The size of outputs
- **aggr_func** – (default "sum") Aggregation function for GraphSage ["sum", "mean", "max", "min"].

forward(*graph*, *feature*, *act=None*)

Parameters

- **graph** – *pgl.Graph* instance.
- **feature** – A tensor with shape (num_nodes, input_size)
- **act** – (default None) Activation for outputs and before normalize.

Returns A tensor with shape (num_nodes, output_size)

class pgl.nn.conv.**PinSageConv**(*input_size*, *hidden_size*, *aggr_func='sum'*)
Bases: paddle.fluid.dygraph.layers.Layer

PinSage combines efficient random walks and graph convolutions to generate embeddings of nodes (i.e., items) that incorporate both graph structure as well as node feature information.

Paper reference: Ying, Rex, et al. "Graph convolutional neural networks for web-scale recommender systems." Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2018.

Parameters

- **input_size** – The size of the inputs.
- **hidden_size** – The size of outputs
- **aggr_func** – (default "sum") Aggregation function for GraphSage ["sum", "mean", "max", "min"].

forward(*graph*, *nfeat*, *efeat*, *act=None*)

Parameters

- **graph** – *pgl.Graph* instance.
- **nfeat** – A tensor with shape (num_nodes, input_size)
- **efeat** – A tensor with shape (num_edges, 1) denotes edge weight.
- **act** – (default None) Activation for outputs and before normalize.

Returns A tensor with shape (num_nodes, output_size)

Graph Pooling Layers

This package implements common pooling to help building graph neural networks.

class pgl.nn.pool.**GraphPool**
Bases: paddle.fluid.dygraph.layers.Layer

Implementation of graph pooling

This is an implementation of graph pooling

Parameters

- **graph** – the graph object from (Graph)
- **feature** – A tensor with shape (num_nodes, feature_size).
- **pool_type** – The type of pooling ("sum", "mean" , "min", "max")

Returns A tensor with shape (num_graph, feature_size)

forward(*graph*, *feature*, *pool_type*)
Defines the computation performed at every call. Should be overridden by all subclasses.

Parameters

- ***inputs** (*tuple*) – unpacked tuple arguments
- ****kwargs** (*dict*) – unpacked dict arguments

10.4.4 pgl.nn.functional

Graph Level Function

pgl.nn.functional.graph_op.**degree_norm**(*graph*, *mode='indegree'*)

10.4.5 pgl.dataset: Some benchmark datasets.

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

class pgl.dataset.**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.dataset.**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.dataset.**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.dataset.**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.

class pgl.dataset.**RedditDataset**(*normalize=True*, *symmetry=True*)
Bases: object

10.4.6 pgl.message

The Message Implement for recv function

class pgl.message.**Message**(*msg*, *segment_ids*)
Bases: object

This implement Message for graph.recv.

Parameters

- **msg** – A dictionary provided by send function.
- **segment_ids** – The id that the message belongs to.

edge_expand(*msg*)
This is the inverse method for reduce.

Parameters **feature** (*paddle.Tensor*) – A reduced message.

Returns Returns a paddle.Tensor with the first dim the same as the num_edges.

Examples

```
import numpy as np
import pgl
import paddle

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

def send_func(src_feat, dst_feat, edge_feat):
    return { "out": src_feat["feature"] }

message = graph.send(send_func, src_feat={"feature": graph.node_feat["feature
↪"]})

def recv_func(msg):
```

(continues on next page)

(continued from previous page)

```
    value = msg["out"]
    max_value = msg.reduce_max(value)
    # We want to subscribe the max_value correspond to the destination node.
    max_value = msg.edge_expand(max_value)
    value = value - max_value
    return msg.reduce_sum(value)

out = graph.recv(recv_func, message)
```

reduce(*msg*, *pool_type='sum'*)
This method reduce message by given *pool_type*.

Now, this method only supports default reduce function, with ('sum', 'mean', 'max', 'min').

Parameters

- **feature** (*paddle.Tensor*) – feature with first dim as num_edges.
- **pool_type** (*str*) – 'sum', 'mean', 'max', 'min' built-in receive function.

Returns Returns a paddle.Tensor with the first dim the same as the largest segment_id.

reduce_max(*msg*)
This method reduce message by max.

Parameters **feature** (*paddle.Tensor*) – feature with first dim as num_edges.

Returns Returns a paddle.Tensor with the first dim the same as the largest segment_id.

reduce_mean(*msg*)
This method reduce message by mean.

Parameters **feature** (*paddle.Tensor*) – feature with first dim as num_edges.

Returns Returns a paddle.Tensor with the first dim the same as the largest segment_id.

reduce_min(*msg*)
This method reduce message by min.

Parameters **feature** (*paddle.Tensor*) – feature with first dim as num_edges.

Returns Returns a paddle.Tensor with the first dim the same as the largest segment_id.

reduce_softmax(*msg*)
This method reduce message by softmax.

Parameters **feature** (*paddle.Tensor*) – feature with first dim as num_edges.

Returns Returns a paddle.Tensor with the first dim the same as the largest segment_id.

reduce_sum(*msg*)
This method reduce message by sum.

Parameters **feature** (*paddle.Tensor*) – feature with first dim as num_edges.

Returns Returns a paddle.Tensor with the first dim the same as the largest segment_id.

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

R

S

T

V

Y