pgl

Release 2.1.3

PaddlePaddle

Aug 23, 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.

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.

To write a sum aggregator, users only need to write the following codes.

```
import pgl
import paddle
import numpy as np


num_nodes = 5
edges = [(0, 1), (1, 2), (3, 4)]
feature = np.random.randn(5, 100).astype(np.float32)

g = pgl.Graph(num_nodes=num_nodes,
    edges=edges,
    node_feat={
        "h": feature
    })
g.tensor()

def send_func(src_feat, dst_feat, edge_feat):
    return src_feat

def recv_func(msg):
    return msg.reduce_sum(msg["h"])

msg = g.send(send_func, src_feat=g.node_feat)

ret = g.recv(recv_func, msg)
```

CHAPTER

ONE

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.

1.1 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.

1.2 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

TWO

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

THREE

MODEL ZOO

The following graph learning models have been implemented in the framework. You can find more examples and the details here.

Model	feature
ERNIESage	ERNIE SAmple aggreGatE for Text and Graph
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

FOUR

SYSTEM REQUIREMENTS

PGL requires:

- paddle >= 2.0.0
- cython

PGL only supports Python 3

CHAPTER

FIVE

INSTALLATION

You can simply install it via pip.

```
pip install pgl
```

CHAPTER

SIX

THE TEAM

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

E-mail: nlp-gnn[at]baidu.com

CHAPTER

SEVEN

LICENSE

PGL uses Apache License 2.0.

CHAPTER

EIGHT

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.

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.

To write a sum aggregator, users only need to write the following codes.

```
import pgl
import paddle
import numpy as np


num_nodes = 5
edges = [(0, 1), (1, 2), (3, 4)]
feature = np.random.randn(5, 100).astype(np.float32)

g = pgl.Graph(num_nodes=num_nodes,
    edges=edges,
    node_feat={
        "h": feature
    })
g.tensor()

def send_func(src_feat, dst_feat, edge_feat):
    return src_feat

def recv_func(msg):
    return msg.reduce_sum(msg["h"])

msg = g.send(send_func, src_feat=g.node_feat)

ret = g.recv(recv_func, msg)
```

8.1 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.

8.1.1 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.

8.1.2 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.

8.2 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.

8.3 Model Zoo

The following graph learning models have been implemented in the framework. You can find more examples and the details here.

Model	feature
ERNIESage	ERNIE SAmple aggreGatE for Text and Graph
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.

8.4 System requirements

PGL requires:

- paddle >= 2.0.0
- cython

PGL only supports Python 3

8.5 Installation

You can simply install it via pip.

```
pip install pgl
```

8.6 The Team

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

E-mail: nlp-gnn[at]baidu.com

8.7 License

PGL uses Apache License 2.0.

CHAPTER

NINE

QUICK START

9.1 Quick Start Instructions

9.1.1 Install PGL

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

```
paddlepaddle >= 2.0.0
cython
```

We can simply install pgl by pip.

```
pip install pgl
```

9.1.2 Introduction

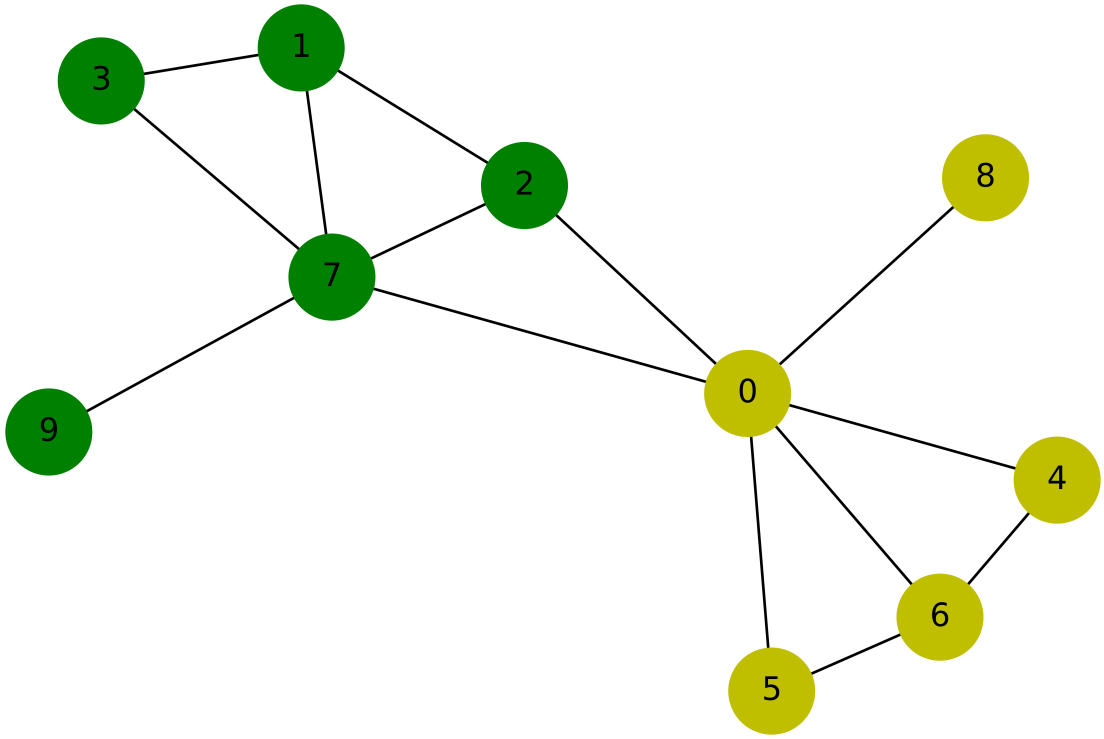
Paddle Graph Learning (PGL) is an efficient and flexible graph learning framework based on PaddlePaddle.

To let users get started quickly, the main purpose of this tutorial is:

- Understand how a graph network is calculated based on PGL.
- Use PGL to implement a simple graph neural network model, which is used to classify the nodes in the graph.

9.1.3 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 numpy as np

import paddle
import paddle.nn as nn
import paddle.nn.functional as F
from paddle.optimizer import Adam
import pgl


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 = pgl.Graph(edges = edge_list,
                  num_nodes = num_node,
                  node_feat = {'nfeat':feature},
```

(continues on next page)

(continued from previous page)

```
                edge_feat ={'efeat': edge_feature})

    return g
```

```
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)
```

```
There are 10 nodes in the graph.
There are 14 edges in the graph.
```

9.1.4 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 you 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:

```
class GCN(nn.Layer):
    """Implement of GCN
    """

    def __init__(self,
                 input_size,
                 num_class,
                 num_layers=2,
                 hidden_size=16,
                 **kwargs):
        super(GCN, self).__init__()
        self.num_class = num_class
        self.num_layers = num_layers
        self.hidden_size = hidden_size
        self.gcns = nn.LayerList()
        for i in range(self.num_layers):
            if i == 0:
                self.gcns.append(
                    pgl.nn.GCNConv(
                        input_size,
                        self.hidden_size,
                        activation="relu",
                        norm=True))
            else:
                self.gcns.append(
                    pgl.nn.GCNConv(
                        self.hidden_size,
                        self.hidden_size,
```

(continues on next page)

(continued from previous page)

```
                        activation="relu",
                        norm=True))

        self.output = nn.Linear(self.hidden_size, self.num_class)
    def forward(self, graph, feature):
        for m in self.gcns:
            feature = m(graph, feature)
        logits = self.output(feature)
        return logits
```

9.1.5 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")
```

9.1.6 Step 4: training

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

```
g = g.tensor()
y = paddle.to_tensor(y)
gcn = GCN(16, 2)
criterion = paddle.nn.loss.CrossEntropyLoss()
optim = Adam(learning_rate=0.01,
             parameters=gcn.parameters())
```

```
gcn.train()
for epoch in range(30):
    logits = gcn(g, g.node_feat['nfeat'])
    loss = criterion(logits, y)
    loss.backward()
    optim.step()
    optim.clear_grad()
    print("epoch: %s | loss: %.4f" % (epoch, loss.numpy()[0]))
```

```
epoch: 0 | loss: 0.7915
epoch: 1 | loss: 0.6991
epoch: 2 | loss: 0.6377
epoch: 3 | loss: 0.6056
epoch: 4 | loss: 0.5844
epoch: 5 | loss: 0.5643
epoch: 6 | loss: 0.5431
epoch: 7 | loss: 0.5214
epoch: 8 | loss: 0.5001
epoch: 9 | loss: 0.4812
epoch: 10 | loss: 0.4683
epoch: 11 | loss: 0.4565
epoch: 12 | loss: 0.4449
epoch: 13 | loss: 0.4343
epoch: 14 | loss: 0.4248
```

(continues on next page)

(continued from previous page)

```
epoch: 15 | loss: 0.4159
epoch: 16 | loss: 0.4081
epoch: 17 | loss: 0.4016
epoch: 18 | loss: 0.3963
epoch: 19 | loss: 0.3922
epoch: 20 | loss: 0.3892
epoch: 21 | loss: 0.3869
epoch: 22 | loss: 0.3854
epoch: 23 | loss: 0.3845
epoch: 24 | loss: 0.3839
epoch: 25 | loss: 0.3837
epoch: 26 | loss: 0.3838
epoch: 27 | loss: 0.3840
epoch: 28 | loss: 0.3843
epoch: 29 | loss: 0.3846
```

9.2 Quick Start with HeterGraph

9.2.1 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.

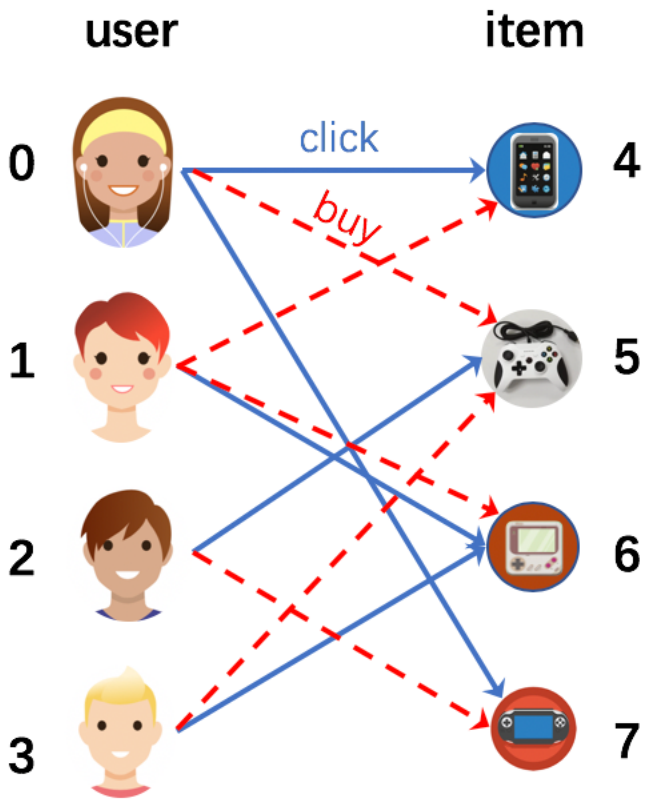
The goal of this tutorial:

- 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.

9.2.2 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”.



9.2.3 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)],
    }

clicked = [(j, i) for i, j in edges['click']]
bought = [(j, i) for i, j in edges['buy']]
edges['clicked'] = clicked
edges['bought'] = bought
```

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
import paddle
import paddle.nn as nn
```

(continues on next page)

(continued from previous page)

```
import pgl
seed = 0
np.random.seed(0)
paddle.seed(0)

num_nodes = len(node_types)

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

labels = np.array([0, 1, 0, 1, 0, 1, 1, 0])
```

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

```
g = pgl.HeterGraph(edges=edges,
                   node_types=node_types,
                   node_feat=node_features)
```

9.2.4 MessagePassing on Heterogeneous Graph

After building the heterogeneous graph, we can easily carry out the message passing mode. In this case, we have two different types of edges.

```
class HeterMessagePassingLayer(nn.Layer):
    def __init__(self, in_dim, out_dim, etypes):
        super(HeterMessagePassingLayer, self).__init__()
        self.in_dim = in_dim
        self.out_dim = out_dim
        self.etypes = etypes

        self.weight = []
        for i in range(len(self.etypes)):
            self.weight.append(
                 self.create_parameter(shape=[self.in_dim, self.out_dim]))

    def forward(self, graph, feat):
        def send_func(src_feat, dst_feat, edge_feat):
            return src_feat

        def recv_func(msg):
            return msg.reduce_mean(msg["h"])

        feat_list = []
        for idx, etype in enumerate(self.etypes):
            h = paddle.matmul(feat, self.weight[idx])
            msg = graph[etype].send(send_func, src_feat={"h": h})
            h = graph[etype].recv(recv_func, msg)
            feat_list.append(h)

        h = paddle.stack(feat_list, axis=0)
        h = paddle.sum(h, axis=0)

        return h
```

Create a simple GNN by stacking two HeterMessagePassingLayer.

```
class HeterGNN(nn.Layer):
    def __init__(self, in_dim, hidden_size, etypes, num_class):
        super(HeterGNN, self).__init__()
        self.in_dim = in_dim
        self.hidden_size = hidden_size
        self.etypes = etypes
        self.num_class = num_class

        self.layers = nn.LayerList()
        self.layers.append(
                HeterMessagePassingLayer(self.in_dim, self.hidden_size, self.etypes))
        self.layers.append(
                HeterMessagePassingLayer(self.hidden_size, self.hidden_size, self.
↪etypes))

        self.linear = nn.Linear(self.hidden_size, self.num_class)

    def forward(self, graph, feat):
        h = feat
        for i in range(len(self.layers)):
            h = self.layers[i](graph, h)

        logits = self.linear(h)

        return logits
```

9.2.5 Training

```
model = HeterGNN(8, 8, g.edge_types, 2)

criterion = paddle.nn.loss.CrossEntropyLoss()

optim = paddle.optimizer.Adam(learning_rate=0.05,
                              parameters=model.parameters())

g.tensor()
labels = paddle.to_tensor(labels)
for epoch in range(10):
    #print(g.node_feat["features"])
    logits = model(g, g.node_feat["features"])
    loss = criterion(logits, labels)
    loss.backward()
    optim.step()
    optim.clear_grad()

    print("epoch: %s | loss: %.4f" % (epoch, loss.numpy()[0]))
```

```
epoch: 0 | loss: 1.3536
epoch: 1 | loss: 1.1593
epoch: 2 | loss: 0.9971
epoch: 3 | loss: 0.8670
epoch: 4 | loss: 0.7591
epoch: 5 | loss: 0.6629
epoch: 6 | loss: 0.5773
epoch: 7 | loss: 0.5130
```

(continues on next page)

(continued from previous page)

```
epoch: 8 | loss: 0.4782
epoch: 9 | loss: 0.4551
```

9.3 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.

9.3.1 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.

9.3.2 Dependencies

- paddlepaddle >= 2.0.0
- pgl >= 2.0

9.3.3 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
```

9.3.4 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.

9.3.5 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

9.4 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.

9.4.1 Simple example to build GCN

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

```
import paddle
import paddle.nn as nn

class CustomGCNConv(nn.Layer):
    def __init__(self, input_size, output_size):
        super(GCNConv, self).__init__()
        self.input_size = input_size
        self.output_size = output_size
        self.linear = nn.Linear(input_size, output_size)
        self.norm = norm
        self.activation = activation

    def forward(self, graph, feature):
        norm = GF.degree_norm(graph)

        feature = self.linear(feature)

        output = graph.send_recv(feature, "sum")

        output = output * norm
        output = nn.functional.relu(output)
        return output
```

9.4.2 Datasets

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

9.4.3 Dependencies

- paddlepaddle==2.0.0
- pgl==2.1

9.4.4 Performance

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

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

9.4.5 How to run

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

```
# Run on GPU
CUDA_VISIBLE_DEVICES=0 python train.py --dataset cora

# Run on CPU
CUDA_VISIBLE_DEVICES= python train.py --dataset cora
```

Hyperparameters

- dataset: The citation dataset "cora", "citeseer", "pubmed".

9.5 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.

9.5.1 Simple example to build single head GAT

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

```
import paddle.fluid as fluid

class CustomGATConv(nn.Layer):
    def __init__(self,
                 input_size, hidden_size,
                 ):

        self.hidden_size = hidden_size
        self.num_heads = num_heads

        self.linear = nn.Linear(input_size, hidden_size)
        self.weight_src = self.create_parameter(shape=[ hidden_size ])
        self.weight_dst = self.create_parameter(shape=[ hidden_size ])

        self.leaky_relu = nn.LeakyReLU(negative_slope=0.2)

    def send_attention(self, src_feat, dst_feat, edge_feat):
        alpha = src_feat["src"] + dst_feat["dst"]
        alpha = self.leaky_relu(alpha)
```

(continues on next page)

(continued from previous page)

```
        return {"alpha": alpha, "h": src_feat["h"]}

    def reduce_attention(self, msg):
        alpha = msg.reduce_softmax(msg["alpha"])
        feature = msg["h"]
        feature = feature * alpha
        feature = msg.reduce(feature, pool_type="sum")
        return feature

    def forward(self, graph, feature):
        feature = self.linear(feature)
        attn_src = paddle.sum(feature * self.weight_src, axis=-1)
        attn_dst = paddle.sum(feature * self.weight_dst, axis=-1)
        msg = graph.send(
            self.send_attention,
            src_feat={"src": attn_src,
                      "h": feature},
            dst_feat={"dst": attn_dst})
        output = graph.recv(reduce_func=self.reduce_attention, msg=msg)
        return output
```

9.5.2 Datasets

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

9.5.3 Dependencies

- paddlepaddle==2.0.0
- pgl==2.1

9.5.4 Performance

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

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

9.5.5 How to run

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

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

Hyperparameters

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

9.6 RGCN: Modeling Relational Data with Graph Convolutional Networks

RGCN is a graph convolutional networks applied in heterogeneous graph.

Its message-passing equation is as follows:

:math:‘‘ h*{i}^{(l+1)}=sigmaleft(sum*{r in mathcal{R}} sum*{j *in mathcal{N}*{r}(i)} W*{r}^{(l)} h*{j}^{(l)}right) $$

From the equation above, we can see that there are two parts in the computation.

1, Message aggregation within each relation r (edge_type).

2, Reduction that merges the results from multiple relationships.

9.6.1 Datasets

Here, we use MUTAG dataset to reproduce this model. The dataset can be downloaded from here.

9.6.2 Dependencies

- paddlepaddle>=2.0
- pgl>=2.1

9.6.3 How to run

To train a RGCN model on MUTAG dataset, you can just run

```
export CUDA_VISIBLE_DEVICES=0

python train.py --data_path /your/path/to/mutag_data
```

If you want to train a RGCN model with multiple GPUs, you can just run with fleetrun API with `CUDA_VISIBLE_DEVICES`

```
CUDA_VISIBLE_DEVICES=0,1 fleetrun train.py --data_path /your/path/to/mutag_data
```

Hyperparameters

- data_path: The directory of your dataset.
- epochs: Number of epochs default (10)
- input_size: Input dimension.
- hidden_size: The hidden size for the RGCN model.

- num_class: The number of classes to be predicted.
- num_layers: The number of RGCN layers to be applied.
- num_bases: Number of basis decomposition
- seed: Random seed.
- lr: Learning rate.

9.6.4 Performance

We train the RGCN model for 10 epochs and report the besst accuracy on the test dataset.

Dataset	Accuracy	Reported in paper
MUTAG	77.94%	73.23%

9.7 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) with the **public train/dev/test split**, 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.

9.7.1 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)	•
SSGC (Zhu 2021)	0.834(0.000)	0.796(0.000)	0.734(0.000)	Weight decay is important, 1e-4 for Citeseer/ 5e-6 for Cora / 5e-6 for Pubmed

9.7.2 How to run the experiments?

```
# Device choose
# use GPU
export CUDA_VISIBLE_DEVICES=0
# use CPU
export CUDA_VISIBLE_DEVICES=

# Experimental API
# If you want to try MultiGPU-FullBatch training. Run the following code instead.
# This will only speed up models that have more computation on edges.
# For example, the TransformerConv in [Yun 2020](https://arxiv.org/abs/2009.03509).

CUDA_VISIBLE_DEVICES=0,1 multi_gpu_train.py --conf config/transformer.yaml

# GCN
python train.py --conf config/gcn.yaml  --dataset cora
python train.py --conf config/gcn.yaml  --dataset pubmed
python train.py --conf config/gcn.yaml  --dataset citeseer

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

# SGC
python train.py --conf config/sgc.yaml --dataset cora
python train.py --conf config/sgc.yaml --dataset pubmed
python train.py --conf config/sgc.yaml --dataset citeseer

# APPNP
python train.py --conf config/appnp.yaml --dataset cora
python train.py --conf config/appnp.yaml --dataset pubmed
python train.py --conf config/appnp.yaml --dataset citeseer

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

# TransformConv + Gated Residual
python train.py --conf config/transformer.yaml --dataset cora
python train.py --conf config/transformer.yaml --dataset pubmed
python train.py --conf config/transformer.yaml --dataset citeseer

# SSGC
python train.py --conf config/sgc.yaml --dataset cora
python train.py --conf config/sgc.yaml --dataset pubmed
python train.py --conf config/sgc.yaml --dataset citeseer
```

9.8 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.

9.8.1 Datasets

The reddit dataset should be downloaded from the following links and placed in the directory `pgl.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

9.8.2 Dependencies

- paddlepaddle>=2.0
- pgl

9.8.3 How to run

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

```
python train.py  --epoch 10  --normalize --symmetry
```

If you want to train a GraphSAGE model with multiple GPUs, you can just run with fleetrun API with `CUDA_VISIBLE_DEVICES`

```
CUDA_VISIBLE_DEVICES=0,1 fleetrun train.py --epoch 10 --normalize --symmetry
```

If you want to train a GraphSAGE model with CPU Parameters, you can just run with fleetrun API with `train_distributed_cpu.py`

```
fleetrun --worker_num 2 --server_num 2 train_distributed_cpu.py --epoch 10 --
↪normalize --symmetry
```

Hyperparameters

- epoch: Number of epochs default (10)
- 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: The max neighbors for each layers hop neighbor sampling. (default: [25, 10])
- hidden_size: The hidden size of the GraphSAGE models.

9.8.4 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%

9.9 API Reference

9.9.1 pgl.heter_graph

9.9.2 pgl.graph

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.

9.9.3 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.

9.9.4 pgl.nn

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

9.9.5 pgl.nn.functional

Graph Level Function

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

9.9.6 pgl.dataset

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

9.9.7 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

TEN

THE TEAM

10.1 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

ELEVEN

LICENSE

PGL uses Apache License 2.0.

PYTHON MODULE INDEX

p

INDEX

R

S

T

V

Y