SGL

Release beta

DAIR @PKU

Jul 29, 2022

GET STARTED

1 Library Highlights **3**

2 Related Publications **5**

3 License **7**

3.1 Overview . . . 7
3.2 Installation . . . 8
3.3 Quick Start . . . 9
3.4 sgl.data . . . 13
3.5 sgl.datasets . . . 13
3.6 sgl.operators.graph_op . . . 13
3.7 sgl.operators.message_op . . . 14
3.8 sgl.models . . . 14
3.9 sgl.tasks . . . 14
3.10 sgl.search . . . 14

Python Module Index **15**

Index **17**

SGL is a Graph Neural Network (GNN) toolkit targeting scalable graph learning, which supports deep graph learning on extremely large datasets. SGL allows users to easily implement scalable graph neural networks and evaluate its performance on various downstream tasks like node classification, node clustering, and link prediction. Further, SGL supports auto neural architecture search functionality based on OpenBox. SGL is designed and developed by the graph learning team from the DAIR Lab at Peking University.

CHAPTER

ONE

LIBRARY HIGHLIGHTS

- **High scalability**: Follow the scalable design paradigm **SGAP** in PaSca, SGL scale to graph data with billions of nodes and edges.
- **Auto neural architecture search**: Automatically choose decent neural architectures according to specific tasks, and pre-defined objectives (e.g., inference time).
- **Ease of use**: User-friendly interfaces of implementing existing scalable GNNs and executing various downstream tasks.

CHAPTER

TWO

RELATED PUBLICATIONS

PaSca: a Graph Neural Architecture Search System under the Scalable Paradigm [PDF] Wentao Zhang, Yu Shen, Zheyu Lin, Yang Li, Xiaosen Li, Wen Ouyang, Yangyu Tao, Zhi Yang, and Bin Cui. The world wide web conference. (WWW 2022, CCF-A)

Node Dependent Local Smoothing for Scalable Graph Learning [PDF] Wentao Zhang, Mingyu Yang, Zeang Sheng, Yang Li, Wen Ouyang, Yangyu Tao, Zhi Yang, Bin Cui. Thirty-fifth Conference on Neural Information Processing Systems. (NeurIPS 2021, CCF-A, Spotlight Presentation, Acceptance Rate: < 3%).

Graph Attention Multi-Layer Perceptron [PDF] Wentao Zhang, Ziqi Yin, Zeang Sheng, Wen Ouyang, Xiaosen Li, Yangyu Tao, Zhi Yang, Bin Cui. arXiv:2108.10097, 2021. (arXiv preprint).

CHAPTER

THREE

LICENSE

The entire codebase is under MIT license.

3.1 Overview

SGL is a Graph Neural Network (GNN) toolkit targeting scalable graph learning, which supports deep graph learning on extremely large datasets. SGL allows users to easily implement scalable graph neural networks and evaluate its performance on various downstream tasks like node classification, node clustering, and link prediction. Further, SGL supports auto neural architecture search functionality based on OpenBox. SGL is designed and developed by the graph learning team from the DAIR Lab at Peking University.

3.1.1 Main Functionalities

- A handy platform for **implementing and evaluating scalable GNNs**.
- Scalable learning on various graph-related tasks, including **node classification**, **node clustering**, and **link prediction**.
- **Auto neural architecture search** on given tasks, datasets and objectives.

3.1.2 Training paradigm

The main design goal of SGL is to support scalable graph learning. SGL adopts the scalable training paradigm, **SGAP** (Scalable Graph Architecture Paradigm), in PaSca. **SGAP** split the conventional GNN training process into three independent stages — **Preprocessing**, **Training**, and **Postprocessing**, which can be represented as follows:

- **Preprocessing:** $\mathbf{M} = graph\_propagate(\mathbf{A}, \mathbf{X})$**;** $\mathbf{X}' = message\_aggregate(\mathbf{M})$
  - **SGAP** propagates and aggregates information at the graph level.
- **Training:** $\mathbf{Y} = model\_train(\mathbf{X}')$
  - **SGAP** feeds the propagated and aggregated information into a machine learning model (e.g., SVM, MLP) for training.
- **Postprocessing:** $\mathbf{M}' = graph\_propagate(\mathbf{A}, \mathbf{Y})$**;** $\mathbf{Y}' = message\_aggregate(\mathbf{M}')$
  - **SGAP** again propagates and aggregates the outputs of the previous stage at the graph level.

Note: The first $message\_aggregate$ operation in the **Preprocessing** stage will be transferred to the **Training** stage if it contains learnable parameters; and the second $message\_aggregate$ operation in the **Postprocessing** stage is prohibited to contain learnable parameters.

Compared to conventional GNN training process, **SGAP** has mainly two advantages:

1. The time- and resource-consuming propagation operation is only executed two times during the full training process; while the number of executing propagation in the conventional GNN training process equals to the number of training epochs, which is usually far greater than two.
2. The dependencies between training examples have been fully taken care of in the **Preprocessing** stage. Thus, the training examples can be freely split to small batches to feed into the model in the **Training** stage, which boosts the efficiency and the scalability of the training process.

3.1.3 Model construction paradigm

Corresponding to its training paradigm, **SGAP**, SGL needs to define the behaviors of two $graph\_propagate$ for each GNN model. To fulfill this goal, SGL designs three important modules:

- **Graph Operator**: to carry out the functionality of $graph\_propagate$. It receives the adjacency matrix **A** and the node representation matrix **X**, and outputs a list of propagated information matrices of different propagation depths.
- **Message Operator**: to carry out the functionality of $message\_aggregate$. It receives a list of propagated information matrices and aggregates the matrices according to pre-defined behaviors. The final output of each **Message Operator** is a single matrix.
- **Base Model**: to carry out the functionality of $model\_training$. It can be not only deep learning models like MLP, but also traditional machine learning methods like SVM and random forest.

To construct a GNN model in SGL, the users only need to fill in some blanks with pre-/user-defined **Graph Operators**, **Message Operators** and **Base Models**. Please refer to models part for the detailed API for constructing models. SGL also provides simple interfaces for defining new **Graph Operators** and **Message Operators**, please refer to operators part for more details.

3.2 Installation

Some datasets in SGL are constructed based on PyG.

Please follow the instructions below to install PyG first before installing SGL: https://pytorch-geometric.readthedocs.io/en/latest/notes/installation.html.

3.2.1 Install from pip

When PyG has been installed, SGL can be installed from PyPI by:

```
pip install sgl-dair
```

3.3 Quick Start

In this short tutorial, we will qucikly go through the basic and the advanced usage of SGL. The tutorial is composed of following parts:

- *Basic usage*
  - *Execute graph-related tasks*
    - *Import datasets*
    - *Import GNN models*
    - *Execute tasks*
  - *Auto neural architrcture search (TODO)*
- *Advanced usage*
  - *Adopt user-defined datasets*
  - *Build models under SGAP paradigm*
  - *Implement new Graph Operators*
  - *Implement new Message Operators*

3.3.1 Basic usage

In this part, we will introduce the basic usage of SGL, including how to excute graph-related tasks and how to use the NAS (Neural Architecture Search) functionality.

Execute graph-related tasks

SGL provides user-friendly interfaces to execute graph-related tasks, including node classification, node clustering, and link prediction. In this tutorial, we will go through an example of excuting a node classification task with a SGC on the PubMed dataset.

Import datasets

Here we import the PubMed dataset via the following code:

```
from sgl.datasets import Planetoid
dataset = Planetoid(name="pubmed", root="./", split="official")
```

The `Planetoid` class contains three popular graph datasets: Cora, Citeseer, and PubMed.

- The 1st argument `name` indicates which dataset among the three to choose;
- The 2nd argument `root` indicates where to put the dataset files;
- The 3rd argumnet `split` indicates the train/validation/test split.

SGL has integated many graph datasets other than the Planetoid datasets. Please refer to datasets part for the detailed information of each dataset.

Import GNN models

Here we import the GNN model SGC as follows:

```
from sgl.models.homo import SGC
model = SGC(prop_steps=3, feat_dim=dataset.num_features, output_dim=dataset.num_classes)
```

SGL supports not only GNNs designed for homogeneous graphs but also GNNs designed for heterogeneous graphs. The two different categories of models reside in `sgl.models.homo` and `sgl.models.hetero`, respectively. The GNN model SGC is designed for homogeneous graphs, and thus can be imported from `sgl.models.homo`. `SGC` class has three main arguments:

- The 1st argument `prop_steps` stands for the propagation depth;
- The 2nd argument `feat_dim` stands for the dimension of the input feature;
- The 3rd argument `output_dim` stands for the dimension of the output representation.

Please refer to the models part for more details of SGC and other GNN models.

Execute tasks

The node classification task can be executed by the following code:

```
from sgl.tasks import NodeClassification
device = "cuda:0"
test_acc = NodeClassification(dataset, model, lr=0.1, weight_decay=5e-5, epochs=200,␣
↪device=device).test_acc
```

The users have to input the adopted dataset, the adopted GNN model, and several hyperparameters before executing a task.

The possible output of the above code might be:

```
Preprocessing done in 0.1280s
Epoch: 001 loss_train: 1.0985 acc_train: 0.3333 acc_val: 0.2300 acc_test: 0.2110 time: 1.
↪3086s
Epoch: 002 loss_train: 1.0289 acc_train: 0.3667 acc_val: 0.7100 acc_test: 0.6920 time: 0.
↪0030s
Epoch: 003 loss_train: 0.9554 acc_train: 0.8667 acc_val: 0.7220 acc_test: 0.7300 time: 0.
↪0040s
Epoch: 004 loss_train: 0.8918 acc_train: 0.9333 acc_val: 0.7220 acc_test: 0.7300 time: 0.
↪0030s
Epoch: 005 loss_train: 0.8354 acc_train: 0.9167 acc_val: 0.7400 acc_test: 0.7220 time: 0.
↪0020s
Epoch: 006 loss_train: 0.7835 acc_train: 0.9333 acc_val: 0.7380 acc_test: 0.7180 time: 0.
↪0030s
Epoch: 007 loss_train: 0.7358 acc_train: 0.9167 acc_val: 0.7280 acc_test: 0.7240 time: 0.
↪0020s
Epoch: 008 loss_train: 0.6929 acc_train: 0.9333 acc_val: 0.7320 acc_test: 0.7320 time: 0.
↪0030s
Epoch: 009 loss_train: 0.6546 acc_train: 0.9333 acc_val: 0.7360 acc_test: 0.7340 time: 0.
↪0030s
Epoch: 010 loss_train: 0.6198 acc_train: 0.9333 acc_val: 0.7360 acc_test: 0.7360 time: 0.
↪0030s
```

(continues on next page)

(continued from previous page)

```
......
Epoch: 191 loss_train: 0.1886 acc_train: 1.0000 acc_val: 0.8020 acc_test: 0.7920 time: 0.
↪0030s
Epoch: 192 loss_train: 0.1886 acc_train: 1.0000 acc_val: 0.8020 acc_test: 0.7930 time: 0.
↪0030s
Epoch: 193 loss_train: 0.1885 acc_train: 1.0000 acc_val: 0.8020 acc_test: 0.7920 time: 0.
↪0030s
Epoch: 194 loss_train: 0.1884 acc_train: 1.0000 acc_val: 0.8020 acc_test: 0.7920 time: 0.
↪0030s
Epoch: 195 loss_train: 0.1884 acc_train: 1.0000 acc_val: 0.8020 acc_test: 0.7920 time: 0.
↪0030s
Epoch: 196 loss_train: 0.1883 acc_train: 1.0000 acc_val: 0.8020 acc_test: 0.7920 time: 0.
↪0020s
Epoch: 197 loss_train: 0.1882 acc_train: 1.0000 acc_val: 0.8020 acc_test: 0.7930 time: 0.
↪0040s
Epoch: 198 loss_train: 0.1882 acc_train: 1.0000 acc_val: 0.8020 acc_test: 0.7920 time: 0.
↪0030s
Epoch: 199 loss_train: 0.1881 acc_train: 1.0000 acc_val: 0.8020 acc_test: 0.7920 time: 0.
↪0030s
Epoch: 200 loss_train: 0.1880 acc_train: 1.0000 acc_val: 0.8020 acc_test: 0.7910 time: 0.
↪0030s
Optimization Finished!
Total time elapsed: 1.9751s
Best val: 0.8020, best test: 0.7920
```

Please refer to the tasks part for more details of executing graph-related tasks.

Auto neural architrcture search (TODO)

3.3.2 Advanced usage

In this part, we will introduce the advanced usage of SGL, including adopting user-defined datasets, building models under SGAP paradigm, implementing new graph operators and message operators.

Adopt user-defined datasets

SGL designs two base classes, `NodeDataset` and `HeteroNodeDataset`, for the homogeneous graph datasets and the heterogeneous graph datasets, respectively. We will take implementing a homogeneous graph dataset as an example below to explain how to adopt user-defined datasets.

To implement a new homogeneous graph dataset, one has to first to inherit the base class `NodeDataset`, whose detailed introduction can be found in the data part. Then, there exist two important virtual functions to implement:

- `download`: download the raw files of the dataset from the Interent and store them in pre-defined places;
- `process`: process the raw files fetched by `download` and store the processed file defined by the data class `Graph`.

The data class `Graph` is designed to store the critical data for the homogeneous graph; the corresponding data class for the heterogeneous graph is `HeteroGraph`. To instantiate `Graph`, one needs to at least provide the following information:

- `row`: the row index of the edges in the graph;
- `col`: the column index of the edges in the graph;
- `edge_weight`: the weight of the edges in the graph;

- `edge_type`: the type of the edges in the graph;
- `num_node`: the total number of nodes in the graph;
- `node_type`: the type of the nodes in the graph.

The datasets in the datasets part all follow the same construction scheme.

Please refer to the data part for more detailed introduction of the two base classes, `NodeDataset` and `HeteroNodeDataset`.

Build models under SGAP paradigm

SGL adopts the SGAP (Scalable Graph Architecture Paradigm) as its training paradigm. Corresponding to that, the model construction paradigm differs from the conventional message passing paradigm. The detailed introduciton of the model construction paradigm of SGL is provided in overview. Below will explain how to build a SGC in SGL.

As introduced in overview, a GNN model in SGL is composed of five parts:

- *pre_graph_op*, *pre_msg_op*: **Graph Operator** and **Message Operator** for the Preprocessing stage;
- *base_model*: **Base Model** for the Training stage;
- *post_graph_op*, *post_msg_op*: **Graph Operator** and **Message Operator** for the Postprocessing stage.

Thus, users only have to assign each module with pre-/user-defined Graph Operator/Message operator/Base Model when building models after inheriting the base class `BaseSGAPModel`. The behaviors of the adopted different Graph Operators, Message Operators and Base Models determine the behaviors of the built GNN models. The code of building SGC is provided below:

```
from sgl.models.base_model import BaseSGAPModel
from sgl.models.simple_models import LogisticRegression
from sgl.operators.graph_op import LaplacianGraphOp
from sgl.operators.message_op import LastMessageOp


class SGC(BaseSGAPModel):
    def __init__(self, prop_steps, feat_dim, output_dim):
        super(SGC, self).__init__(prop_steps, feat_dim, output_dim)

        self._pre_graph_op = LaplacianGraphOp(prop_steps, r=0.5)
        self._pre_msg_op = LastMessageOp()
        self._base_model = LogisticRegression(feat_dim, output_dim)
```

Note: The *LaplacianGraphOp*, *LastMessageOp*,and *LogisticRegreesion* are pre-defined Graph Operator, Message Operator, and Base Model, respectively.

Note: SGC does not have the Postprocessing stage in its training process. Thus, the modules used for the Postprocessing stage do not exist in the construction of SGC.

In the following parts of this tutorial, we will introduce ways to implement new Graph Operators and Message Operators.

Implement new Graph Operators

As introduced in overview, the behaviors of the Graph Operators can be represented as follows: $\mathbf{M} = graph\_propagate(\mathbf{A}, \mathbf{X})$. Thus, the critical part of implementing new Graph Operators is to determine the value of the matrix $\mathbf{A}$.

In SGL, users only need to implement the virtual function *construct_adj*, which takes in the original adjacency matrix of the graph and outputs the desired propagation matrix after inheriting the base class `GraphOp`. Below is the implementation of the PPR (Personalized PageRank) Graph Operator:

```
class PprGraphOp(GraphOp):
    def __init__(self, prop_steps, r=0.5, alpha=0.15):
        super(PprGraphOp, self).__init__(prop_steps)
        self.__r = r
        self.__alpha = alpha

    def _construct_adj(self, adj):
        adj_normalized = adj_to_symmetric_norm(adj, self.__r)
        adj_normalized = (1 - self.__alpha) * adj_normalized + self.__alpha * sp.eye(adj.
↪shape[0])
        return adj_normalized.tocsr()
```

Please refer to operators part for more detailed introduction.

Implement new Message Operators

Similar to implementing new Graph Operators, implementing new Message Operators is easy in SGL. The users need to determine the behaviors of the new Message Operators represented in $\mathbf{X}' = message\_aggregate(\mathbf{M})$.

Practically speaking, users have to implement the virtual function *combine* function after inheriting the base class `MessageOp`. The code below provides the implementation of the ConcatMessageOp in SGL:

```
class ConcatMessageOp(MessageOp):
    def __init__(self, start, end):
        super(ConcatMessageOp, self).__init__(start, end)
        self._aggr_type = "concat"

    def _combine(self, feat_list):
        return torch.hstack(feat_list[self._start:self._end])
```

Please refer to operators part for more detailed introduction.

3.4 sgl.data

3.5 sgl.datasets

3.6 sgl.operators.graph_op

class sgl.operators.**GraphOp**(*prop_steps*)

3.7 sgl.operators.message_op

class sgl.operators.**MessageOp**(*start=None*, *end=None*)

class sgl.operators.message_op.**ConcatMessageOp**(*start*, *end*)

Bases: *MessageOp*

class sgl.operators.message_op.**IterateLearnableWeightedMessageOp**(*start*, *end*, *combination_type*, **args*)

Bases: *MessageOp*

class sgl.operators.message_op.**LastMessageOp**

Bases: *MessageOp*

class sgl.operators.message_op.**LearnableWeightedMessageOp**(*start*, *end*, *combination_type*, **args*)

Bases: *MessageOp*

class sgl.operators.message_op.**MaxMessageOp**(*start*, *end*)

Bases: *MessageOp*

class sgl.operators.message_op.**MeanMessageOp**(*start*, *end*)

Bases: *MessageOp*

class sgl.operators.message_op.**MinMessageOp**(*start*, *end*)

Bases: *MessageOp*

class sgl.operators.message_op.**OverSmoothDistanceWeightedOp**

Bases: *MessageOp*

class sgl.operators.message_op.**ProjectedConcatMessageOp**(*start*, *end*, *feat_dim*, *hidden_dim*, *num_layers*)

Bases: *MessageOp*

class sgl.operators.message_op.**SimpleWeightedMessageOp**(*start*, *end*, *combination_type*, **args*)

Bases: *MessageOp*

class sgl.operators.message_op.**SumMessageOp**(*start*, *end*)

Bases: *MessageOp*

3.8 sgl.models

3.9 sgl.tasks

3.10 sgl.search

PYTHON MODULE INDEX

S

sgl.operators.graph_op, 13
sgl.operators.message_op, 14

INDEX

C

ConcatMessageOp (*class in sgl.operators.message_op*), 14

G

GraphOp (*class in sgl.operators*), 13

I

IterateLearnableWeightedMessageOp (*class in sgl.operators.message_op*), 14

L

LastMessageOp (*class in sgl.operators.message_op*), 14
LearnableWeightedMessageOp (*class in sgl.operators.message_op*), 14

M

MaxMessageOp (*class in sgl.operators.message_op*), 14
MeanMessageOp (*class in sgl.operators.message_op*), 14
MessageOp (*class in sgl.operators*), 14
MinMessageOp (*class in sgl.operators.message_op*), 14
module
    sgl.operators.graph_op, 13
    sgl.operators.message_op, 14

O

OverSmoothDistanceWeightedOp (*class in sgl.operators.message_op*), 14

P

ProjectedConcatMessageOp (*class in sgl.operators.message_op*), 14

S

sgl.operators.graph_op
    module, 13
sgl.operators.message_op
    module, 14
SimpleWeightedMessageOp (*class in sgl.operators.message_op*), 14
SumMessageOp (*class in sgl.operators.message_op*), 14