4.3 Process data

(中文版)

One can implement the data processing code in function `process()`, and it assumes that the raw data is located in `self.raw_dir` already. There are typically three types of tasks in machine learning on graphs: graph classification, node classification, and link prediction. This section will show how to process datasets related to these tasks.

The section focuses on the standard way to process graphs, features and masks. It will use builtin datasets as examples and skip the implementations for building graphs from files, but add links to the detailed implementations. Please refer to 1.4 Creating Graphs from External Sources to see a complete guide on how to build graphs from external sources.

Processing Graph Classification datasets

Graph classification datasets are almost the same as most datasets in typical machine learning tasks, where mini-batch training is used. So one can process the raw data to a list of `dgl.DGLGraph` objects and a list of label tensors. In addition, if the raw data has been split into several files, one can add a parameter `split` to load specific part of the data.

Take `QM7bDataset` as example:

```python
from dgl.data import DGLDataset

class QM7bDataset(DGLDataset):
    _url = 'http://deepchem.io.s3-website-us-west-1.amazonaws.com/' \
           'datasets/qm7b.mat'
    _sha1_str = '4102c744bb9d6fd7b40ac67a300e49cd87e28392'

    def __init__(self, raw_dir=None, force_reload=False, verbose=False):
        super(QM7bDataset, self).__init__(name='qm7b',
                                          url=self._url,
                                          raw_dir=raw_dir,
                                          force_reload=force_reload,
                                          verbose=verbose)

    def process(self):
        mat_path = self.raw_path + '.mat'
        # process data to a list of graphs and a list of labels
        self.graphs, self.label = self._load_graph(mat_path)

    def __getitem__(self, idx):
        """ Get graph and label by index

        Parameters
        ----------
        idx : int
            Item index

        Returns
        -------
        (dgl.DGLGraph, Tensor)
        """
        return self.graphs[idx], self.label[idx]

    def __len__(self):
        """Number of graphs in the dataset"""
        return len(self.graphs)
```

In `process()`, the raw data is processed to a list of graphs and a list of labels. One must implement `__getitem__(idx)` and `__len__()` for iteration. DGL recommends making `__getitem__(idx)` return a tuple `(graph, label)` as above. Please check the QM7bDataset source code for details of `self._load_graph()` and `__getitem__`.

One can also add properties to the class to indicate some useful information of the dataset. In `QM7bDataset`, one can add a property `num_tasks` to indicate the total number of prediction tasks in this multi-task dataset:

```python
@property
def num_tasks(self):
    """Number of labels for each graph, i.e. number of prediction tasks."""
    return 14
```

After all these coding, one can finally use `QM7bDataset` as follows:

```
import dgl
import torch

from dgl.dataloading import GraphDataLoader

# load data
dataset = QM7bDataset()
num_tasks = dataset.num_tasks

# create dataloaders
dataloader = GraphDataLoader(dataset, batch_size=1, shuffle=True)

# training
for epoch in range(100):
    for g, labels in dataloader:
        # your training code here
        pass
```

A complete guide for training graph classification models can be found in 5.4 Graph Classification.

For more examples of graph classification datasets, please refer to DGL's builtin graph classification datasets:

- gindataset
- minigcdataset
- qm7bdata
- tudata

Processing Node Classification datasets

Different from graph classification, node classification is typically on a single graph. As such, splits of the dataset are on the nodes of the graph. DGL recommends using node masks to specify the splits. The section uses builtin dataset CitationGraphDataset as an example:

In addition, DGL recommends re-arrange the nodes and edges so that nodes near to each other have IDs in a close range. The procedure could improve the locality to access a node's neighbors, which may benefit follow-up computation and analysis conducted on the graph. DGL provides an API called `dgl.reorder_graph()` for this purpose. Please refer to `process()` part in below example for more details.

```
from dgl.data import DGLBuiltinDataset
from dgl.data.utils import _get_dgl_url

class CitationGraphDataset(DGLBuiltinDataset):
    _urls = {
        'cora_v2' : 'dataset/cora_v2.zip',
        'citeseer' : 'dataset/citeseer.zip',
        'pubmed' : 'dataset/pubmed.zip',
    }

    def __init__(self, name, raw_dir=None, force_reload=False, verbose=True):
        assert name.lower() in ['cora', 'citeseer', 'pubmed']
        if name.lower() == 'cora':
            name = 'cora_v2'
        url = _get_dgl_url(self._urls[name])
        super(CitationGraphDataset, self).__init__(name,
                                                   url=url,
                                                   raw_dir=raw_dir,
                                                   force_reload=force_reload,
                                                   verbose=verbose)

    def process(self):
        # Skip some processing code
        # === data processing skipped ===

        # build graph
        g = dgl.graph(graph)
        # splitting masks
        g.ndata['train_mask'] = train_mask
        g.ndata['val_mask'] = val_mask
        g.ndata['test_mask'] = test_mask
        # node labels
        g.ndata['label'] = torch.tensor(labels)
        # node features
        g.ndata['feat'] = torch.tensor(_preprocess_features(features),
                                       dtype=F.data_type_dict['float32'])
        self._num_tasks = onehot_labels.shape[1]
        self._labels = labels
        # reorder graph to obtain better locality.
        self._g = dgl.reorder_graph(g)

    def __getitem__(self, idx):
        assert idx == 0, "This dataset has only one graph"
        return self._g

    def __len__(self):
        return 1
```

For brevity, this section skips some code in `process()` to highlight the key part for processing node classification dataset: splitting masks. Node features and node labels are stored in `g.ndata`. For detailed implementation, please refer to CitationGraphDataset source code.

Note that the implementations of `__getitem__(idx)` and `__len__()` are changed as well, since there is often only one graph for node classification tasks. The masks are `bool tensors` in PyTorch and TensorFlow, and `float tensors` in MXNet.

The section uses a subclass of `CitationGraphDataset`, `dgl.data.CiteseerGraphDataset`, to show the usage of it:

```
# load data
dataset = CiteseerGraphDataset(raw_dir='')
graph = dataset[0]

# get split masks
train_mask = graph.ndata['train_mask']
val_mask = graph.ndata['val_mask']
test_mask = graph.ndata['test_mask']

# get node features
feats = graph.ndata['feat']

# get labels
labels = graph.ndata['label']
```

A complete guide for training node classification models can be found in 5.1 Node Classification/Regression.

For more examples of node classification datasets, please refer to DGL's builtin datasets:

- citationdata
- corafulldata
- amazoncobuydata
- coauthordata
- karateclubdata
- ppidata
- redditdata
- sbmdata
- sstdata
- rdfdata

Processing dataset for Link Prediction datasets

The processing of link prediction datasets is similar to that for node classification's, there is often one graph in the dataset.

The section uses builtin dataset KnowledgeGraphDataset as an example, and still skips the detailed data processing code to highlight the key part for processing link prediction datasets:

```
# Example for creating Link Prediction datasets
class KnowledgeGraphDataset(DGLBuiltinDataset):
    def __init__(self, name, reverse=True, raw_dir=None, force_reload=False, verbose=True):
        self._name = name
        self.reverse = reverse
        url = _get_dgl_url('dataset/') + '{}.tgz'.format(name)
        super(KnowledgeGraphDataset, self).__init__(name,
                                                    url=url,
                                                    raw_dir=raw_dir,
                                                    force_reload=force_reload,
                                                    verbose=verbose)

    def process(self):
        # Skip some processing code
        # === data processing skipped ===

        # splitting mask
        g.edata['train_mask'] = train_mask
        g.edata['val_mask'] = val_mask
        g.edata['test_mask'] = test_mask
        # edge type
        g.edata['etype'] = etype
        # node type
        g.ndata['ntype'] = ntype
        self._g = g

    def __getitem__(self, idx):
        assert idx == 0, "This dataset has only one graph"
        return self._g

    def __len__(self):
        return 1
```

As shown in the code, it adds splitting masks into `edata` field of the graph. Check KnowledgeGraphDataset source code to see the complete code. The following code uses a subclass of `KnowledgeGraphDataset`, `dgl.data.FB15k237Dataset`, to show the usage of it:

```
from dgl.data import FB15k237Dataset

# load data
dataset = FB15k237Dataset()
graph = dataset[0]

# get training mask
train_mask = graph.edata['train_mask']
train_idx = torch.nonzero(train_mask, as_tuple=False).squeeze()
src, dst = graph.edges(train_idx)
# get edge types in training set
rel = graph.edata['etype'][train_idx]
```

A complete guide for training link prediction models can be found in 5.3 Link Prediction.

For more examples of link prediction datasets, please refer to DGL's builtin datasets:

- kgdata
- bitcoinotcdata