超大规模数据集类的创建

在前面的学习中我们只接触了数据可全部储存于内存的数据集，这些数据集对应的数据集类在创建对象时就将所有数据都加载到内存。然而在一些应用场景中，**数据集规模超级大，我们很难有足够大的内存完全存下所有数据**。因此需要**一个按需加载样本到内存的数据集类**。在此上半节内容中，我们将学习为一个包含上千万个图样本的数据集构建一个数据集类。

`Dataset`基类简介

在PyG中，我们通过继承`torch_geometric.data.Dataset`基类来自定义一个按需加载样本到内存的数据集类。此基类与Torchvision的`Dataset`类的概念密切相关，这与第6节中介绍的`torch_geometric.data.InMemoryDataset`基类是一样的。

继承`torch_geometric.data.InMemoryDataset`基类要实现的方法，继承此基类同样要实现，此外还需要实现以下方法：

- `len()`：返回数据集中的样本的数量。
- `get()`：实现加载单个图的操作。注意：在内部，`__getitem__()`返回通过调用`get()`来获取`Data`对象，并根据`transform`参数对它们进行选择性转换。

下面让我们通过一个简化的例子看**继承`torch_geometric.data.Dataset`基类的规范**：

```
import os.path as osp

import torch
from torch_geometric.data import Dataset, download_url

class MyOwnDataset(Dataset):
    def __init__(self, root, transform=None, pre_transform=None):
        super(MyOwnDataset, self).__init__(root, transform, pre_transform)

```

```
    @property
    def raw_file_names(self):
        return ['some_file_1', 'some_file_2', ...]

    @property
    def processed_file_names(self):
        return ['data_1.pt', 'data_2.pt', ...]

    def download(self):
        # Download to `self.raw_dir`.
        path = download_url(url, self.raw_dir)
        ...

    def process(self):
        i = 0
        for raw_path in self.raw_paths:
            # Read data from `raw_path`.
            data = Data(...)

            if self.pre_filter is not None and not
self.pre_filter(data):
                continue

            if self.pre_transform is not None:
                data = self.pre_transform(data)

            torch.save(data,
osp.join(self.processed_dir, 'data_{}.pt'.format(i)))
            i += 1

    def len(self):
        return len(self.processed_file_names)

    def get(self, idx):
        data = torch.load(osp.join(self.processed_dir,
'data_{}.pt'.format(idx)))
        return data

```

其中，每个`Data`对象在`process()`方法中单独被保存，并在`get()`中通过指定索引进行加载。

跳过download/process

对于无需下载数据集原文件的情况，我们不重写（override）`download`方法即可跳过下载。对于无需对数据集做预处理的情况，我们不重写`process`方法即可跳过预处理。

无需定义Dataset类

通过下面的方式，我们可以不用定义一个`Dataset`类，而直接生成一个`Dataloader`对象，直接用于训练：

```
from torch_geometric.data import Data, DataLoader

data_list = [Data(...), ..., Data(...)]
loader = DataLoader(data_list, batch_size=32)
```

我们也可以通过下面的方式将一个列表的`Data`对象组成一个`batch`：

```
from torch_geometric.data import Data, Batch

data_list = [Data(...), ..., Data(...)]
loader = Batch.from_data_list(data_list, batch_size=32)
```

图样本封装成批（BATCHING）与`DataLoader`类

内容来源：ADVANCED MINI-BATCHING

合并小图组成大图

图可以有任意数量的节点和边，它不是规整的数据结构，因此对图数据封装成批的操作与对图像与序列等数据封装成批的操作不同。PyTorch Geometric中采用的将多个图封装成批的方式是，将小图作为连通组件（connected component）的形式合并，构建一个大图。于是小图的邻接矩阵存储在大图邻接矩阵的对角线上。大图的邻接矩阵、属性矩阵、预测目标矩阵分别为：

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 & & \\ & \ddots & \\ & & \mathbf{A}_n \end{bmatrix}, \qquad \mathbf{X} = \begin{bmatrix} \mathbf{X}_1 \\ \vdots \\ \mathbf{X}_n \end{bmatrix}, \qquad \mathbf{Y} = \begin{bmatrix} \mathbf{Y}_1 \\ \vdots \\ \mathbf{Y}_n \end{bmatrix}.$$

此方法有以下关键的优势:

- 依靠消息传递方案的GNN运算不需要被修改，因为消息仍然不能在属于不同图的两个节点之间交换。
- 没有额外的计算或内存的开销。例如，这个批处理程序的工作完全不需要对节点或边缘特征进行任何填充。请注意，邻接矩阵没有额外的内存开销，因为它们是以稀疏的方式保存的，只保留非零项，即边。

通过`torch_geometric.data.DataLoader`类，多个小图被封装成一个大图。`torch_geometric.data.DataLoader`是PyTorch的`DataLoader`的子类，它覆盖了`collate()`函数，改函数定义了一列表的样本是如何封装成批的。因此，所有可以传递给PyTorch `DataLoader`的参数也可以传递给PyTorch Geometric的`DataLoader`，例如，`num_workers`。

小图的属性增值与拼接

将小图存储到大图中时需要对小图的属性做一些修改，一个最显著的例子就是要对节点序号增值。在最一般的形式中，PyTorch Geometric的`DataLoader`类会自动对`edge_index`张量增值，增加的值为当前被处理图的前面的图的累积节点数量。比方说，现在对第k个图的`edge_index`张量做增值，前面$k-1$个图的累积节点数量为n，那么对第k个图的`edge_index`张量的增值n。增值后，对所有图的`edge_index`张量（其形状为`[2, num_edges]`）在第二维中连接起来。

然而，有一些特殊的场景中（如下所述），基于需求我们希望能修改这一行为。PyTorch Geometric允许我们通过覆盖`torch_geometric.data.__inc__()`和`torch_geometric.data.__cat_dim__()`函数来实现我们希望的行为。在未做修改的情况下，它们在`Data`类中的定义如下。

```
def __inc__(self, key, value):
    if 'index' in key or 'face' in key:
        return self.num_nodes
    else:
        return 0

def __cat_dim__(self, key, value):
    if 'index' in key or 'face' in key:
        return 1
    else:
        return 0
```

我们可以看到，`__inc__()`定义了两个连续的图的属性之间的增量大小，而`__cat_dim__()`定义了同一属性的图形张量应该在哪个维度上被连接起来。PyTorch Geometric为存储在`Data`类中的每个属性调用此二函数，并以它们各自的`key`和值`item`作为参数。

在下面的内容中，我们将学习一些对`__inc__()`和`__cat_dim__()`的修改可能是绝对必要的案例。

图的匹配（Pairs of Graphs）

如果你想在一个`Data`对象中存储多个图，例如用于图匹配等应用，我们需要确保所有这些图的正确封装成批行为。例如，考虑将两个图，一个源图G_s和一个目标图G_t，存储在一个`Data`类中，即

```
class PairData(Data):
    def __init__(self, edge_index_s, x_s, edge_index_t, x_t):
        super(PairData, self).__init__()
        self.edge_index_s = edge_index_s
        self.x_s = x_s
        self.edge_index_t = edge_index_t
        self.x_t = x_t

```

在这种情况中，`edge_index_s`应该根据源图G_s的节点数做增值，即`x_s.size(0)`，而`edge_index_t`应该根据目标图G_t的节点数做增值，即`x_t.size(0)`。

```

```

```
class PairData(Data):
    def __init__(self, edge_index_s, x_s, edge_index_t, x_t):
        super(PairData, self).__init__()
        self.edge_index_s = edge_index_s
        self.x_s = x_s
        self.edge_index_t = edge_index_t
        self.x_t = x_t

    def __inc__(self, key, value):
        if key == 'edge_index_s':
            return self.x_s.size(0)
        if key == 'edge_index_t':
            return self.x_t.size(0)
        else:
            return super().__inc__(key, value)
```

我们可以通过设置一个简单的测试脚本来测试我们的PairData批处理行为。

```
edge_index_s = torch.tensor([
    [0, 0, 0, 0],
    [1, 2, 3, 4],
])
x_s = torch.randn(5, 16)  # 5 nodes.
edge_index_t = torch.tensor([
    [0, 0, 0],
    [1, 2, 3],
])
x_t = torch.randn(4, 16)  # 4 nodes.

data = PairData(edge_index_s, x_s, edge_index_t, x_t)
data_list = [data, data]
loader = DataLoader(data_list, batch_size=2)
batch = next(iter(loader))

print(batch)
# Batch(edge_index_s=[2, 8], x_s=[10, 16],
edge_index_t=[2, 6], x_t=[8, 16])

print(batch.edge_index_s)
```

```
# tensor([[0, 0, 0, 0, 5, 5, 5, 5], [1, 2, 3, 4, 6, 7, 8, 9]])

print(batch.edge_index_t)
# tensor([[0, 0, 0, 4, 4, 4], [1, 2, 3, 5, 6, 7]])
```

到目前为止，一切看起来都很好! `edge_index_s`和`edge_index_t`被正确地封装成批了，即使在为G_s和G_t含有不同数量的节点时也是如此。然而，由于PyTorch Geometric无法识别`PairData`对象中实际的图，所以`batch`属性（将大图每个节点映射到其各自对应的小图）没有正确工作。此时就需要`DataLoader`的`follow_batch`参数发挥作用。在这里，我们可以指定我们要为哪些属性维护批信息。

```
loader = DataLoader(data_list, batch_size=2, follow_batch=['x_s', 'x_t'])
batch = next(iter(loader))

print(batch)
# Batch(edge_index_s=[2, 8], x_s=[10, 16], x_s_batch=[10],
#           edge_index_t=[2, 6], x_t=[8, 16], x_t_batch=[8])
print(batch.x_s_batch)
# tensor([0, 0, 0, 0, 0, 1, 1, 1, 1, 1])

print(batch.x_t_batch)
# tensor([0, 0, 0, 0, 1, 1, 1, 1])
```

可以看到，`follow_batch=['x_s', 'x_t']`现在成功地为节点特征`x_s'`和x_t'分别创建了名为`x_s_batch`和`x_t_batch`的赋值向量。这些信息现在可以用来在一个单一的`Batch'对象中对多个图进行聚合操作，例如，全局池化。

二部图（Bipartite Graphs）

二部图的邻接矩阵定义两种类型的节点之间的连接关系。一般来说，不同类型的节点数量不需要一致，于是二部图的邻接矩阵$A \in \{0,1\}^{N \times M}$可能为平方矩阵，即可能有$N \neq M$。对二部图的封装成批过程中，`edge_index`中边的源节点与目标节点做的增值操作应是不同的。我们将二部图中两类节点的特征特征张量分别存储为`x_s`和`x_t`。

```
class BipartiteData(Data):
    def __init__(self, edge_index, x_s, x_t):
        super(BipartiteData, self).__init__()
        self.edge_index = edge_index
        self.x_s = x_s
        self.x_t = x_t
```

为了对二部图实现正确的封装成批，我们需要告诉PyTorch Geometric，它应该在`edge_index`中独立地为边的源节点和目标节点做增值操作。

```
def __inc__(self, key, value):
    if key == 'edge_index':
        return torch.tensor([[self.x_s.size(0)],
[self.x_t.size(0)]])
    else:
        return super().__inc__(key, value)
```

其中，`edge_index[0]`（边的源节点）根据`x_s.size(0)`做增值运算，而`edge_index[1]`（边的目标节点）根据`x_t.size(0)`做增值运算。我们可以再次通过运行一个简单的测试脚本来测试我们的实现。

```
edge_index = torch.tensor([
    [0, 0, 1, 1],
    [0, 1, 1, 2],
])
x_s = torch.randn(2, 16)  # 2 nodes.
x_t = torch.randn(3, 16)  # 3 nodes.

data = BipartiteData(edge_index, x_s, x_t)
data_list = [data, data]
loader = DataLoader(data_list, batch_size=2)
batch = next(iter(loader))

print(batch)
# Batch(edge_index=[2, 8], x_s=[4, 16], x_t=[6, 16])

print(batch.edge_index)
# tensor([[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 1, 2, 3, 4,
4, 5]])
```

可以看到我们得到我们期望的结果。

在新的维度上做拼接

有时，`Data`对象的属性需要在一个新的维度上做拼接（如经典的封装成批），例如，图级别属性或预测目标。具体来说，形状为`[num_features]`的属性列表应该被返回为`[num_examples, num_features]`，而不是`[num_examples * num_features]`。PyTorch Geometric通过在`__cat_dim__()`中返回一个`None`的连接维度来实现这一点。

```
 class MyData(Data):
     def __cat_dim__(self, key, item):
         if key == 'foo':
             return None
         else:
             return super().__cat_dim__(key, item)

edge_index = torch.tensor([
   [0, 1, 1, 2],
   [1, 0, 2, 1],
])
foo = torch.randn(16)

data = MyData(edge_index=edge_index, foo=foo)
data_list = [data, data]
loader = DataLoader(data_list, batch_size=2)
batch = next(iter(loader))

print(batch)
# Batch(edge_index=[2, 8], foo=[2, 16])
```

正如我们期望的，`batch.foo`现在由两个维度来表示，一个批维度，一个特征维度。

创建超大规模数据集类实践

PCQM4M-LSC是一个分子图的量子特性回归数据集，它包含了3,803,453个图。

注意以下代码依赖于`ogb`包，通过`pip install ogb`命令可安装此包。`ogb`文档可见于Get Started | Open Graph Benchmark (stanford.edu)。

我们定义的数据集类如下：

```python
import os
import os.path as osp

import pandas as pd
import torch
from ogb.utils import smiles2graph
from ogb.utils.torch_util import
replace_numpy_with_torchtensor
from ogb.utils.url import download_url, extract_zip
from rdkit import RDLogger
from torch_geometric.data import Data, Dataset
import shutil

RDLogger.DisableLog('rdApp.*')

class MyPCQM4MDataset(Dataset):

    def __init__(self, root):
        self.url = 'https://dgl-data.s3-
accelerate.amazonaws.com/dataset/OGB-
LSC/pcqm4m_kddcup2021.zip'
        super(MyPCQM4MDataset, self).__init__(root)

        filepath = osp.join(root, 'raw/data.csv.gz')
        data_df = pd.read_csv(filepath)
        self.smiles_list = data_df['smiles']
        self.homolumogap_list = data_df['homolumogap']

    @property
    def raw_file_names(self):
        return 'data.csv.gz'

    def download(self):
        path = download_url(self.url, self.root)
        extract_zip(path, self.root)
        os.unlink(path)
        shutil.move(osp.join(self.root,
'pcqm4m_kddcup2021/raw/data.csv.gz'),
osp.join(self.root, 'raw/data.csv.gz'))

    def len(self):
```

```
        return len(self.smiles_list)

    def get(self, idx):
        smiles, homolumogap = self.smiles_list[idx], self.homolumogap_list[idx]
        graph = smiles2graph(smiles)
        assert(len(graph['edge_feat']) == graph['edge_index'].shape[1])
        assert(len(graph['node_feat']) == graph['num_nodes'])

        x = torch.from_numpy(graph['node_feat']).to(torch.int64)
        edge_index = torch.from_numpy(graph['edge_index']).to(torch.int64)
        edge_attr = torch.from_numpy(graph['edge_feat']).to(torch.int64)
        y = torch.Tensor([homolumogap])
        num_nodes = int(graph['num_nodes'])
        data = Data(x, edge_index, edge_attr, y, num_nodes=num_nodes)
        return data

    # 获取数据集划分
    def get_idx_split(self):
        split_dict = replace_numpy_with_torchtensor(torch.load(osp.join(self.root, 'pcqm4m_kddcup2021/split_dict.pt')))
        return split_dict

if __name__ == "__main__":
    dataset = MyPCQM4MDataset('dataset2')
    from torch_geometric.data import DataLoader
    from tqdm import tqdm
    dataloader = DataLoader(dataset, batch_size=256, shuffle=True, num_workers=4)
    for batch in tqdm(dataloader):
        pass

```

在生成一个该数据集类的对象时，程序首先会检查指定的文件夹下是否存在`data.csv.gz`文件，如果不在，则会执行`download`方法，这一过程是在运行`super`类的`__init__`方法中发生的。然后程序继续执行`__init__`方法的剩余部分，读取`data.csv.gz`文件，获取存储图信息的`smiles`格式的字符串，以及回归预测的目标`homolumogap`。我们将由`smiles`格式的字符串转成图的过程在`get()`方法中实现，这样我们在生成一个`DataLoader`变量时，通过指定`num_workers`可以实现并行执行生成多个图。

参考资料

- `Dataset`类官方文档：`torch_geometric.data.Dataset`
- 将图样本封装成批（BATCHING）：ADVANCED MINI-BATCHING
- 分子图的量子特性回归数据集：PCQM4M-LSC
- Get Started | Open Graph Benchmark (stanford.edu)