图数据常规操作

图一般被用来建模和描述目标（节点）间成对的关系（边）。在Pytorch Gemometric（以后均简称pyg）中，一个图是由torch_geometric.data.Data的一个实例来描述的，设此图有N个节点，每个节点有n个特征，M条边，每条边有m个特征，默认情况下拥有如下的属性：

1. **data.x**: 节点的特征矩阵，形状：[N，n]
2. **data.edge_index**: 用COO格式储存的图数据，形状：[2,M]（如不理解没事，后面我会在栗子中详细介绍），数据类型是torch.longCOO就是坐标,coordinate.思想也很简单，每列分别存储：行坐标，纵坐标，值。
3. **data.edge_attr**: 边特征矩阵，形状[M，m]
4. **data.y**: 要训练的目标（可以是任意形状），如节点级目标[节点数，*]，图级目标[1,*],此处*代表样本数量。
5. **data.pos**: 节点位置矩阵，形状：[N,num_dimensions],在有些图中，节点是具有坐标属性，比如3D点云，每个节点都是3维空间中的一个坐标，类似的也可以是其它维度的。这些属性都不是必须属性，实际上Data对象并不局限于这些属性。例如：我们可以通过data.face来扩展它，用以保存形状为[3，num_Faces]（数据类型为torch.long）的三维网格中三角形的连通性。

```
import torch
from torch_geometric.data import Data

# 以列为单位,比如第一列:0,1表示节点0到节点1的边:0->1,
# 同理1,2表示边1->2
# 这样储存可以很方便储存稀疏的图。虽然这个图只有2条边，但是为了统一无向图和有向图的存储，
# 因此需要每条边存2个方向。
# 竖向是边
edge_index = torch.tensor([[0, 1, 1, 2],
                           [1, 0, 2, 1]], dtype=torch.long)
# 特征
x = torch.tensor([[-1], [0], [1]], dtype=torch.float)

data = Data(x=x,edge_indx=edge_index)


# print(data)
# Data(edge_indx=[2, 4], x=[3, 1])
# 2维4条边,3个节点,每个节点1个特征

# import torch
# from torch_geometric.data import Data
# #以行为单位更符合人类的逻辑，但是在构建数据时，记得先转置，再contiguous
# edge_index = torch.tensor([[0, 1],
#                            [1, 0],
#                            [1, 2],
#                            [2, 1]], dtype=torch.long)
# x = torch.tensor([[-1], [0], [1]], dtype=torch.float)
#
# data = Data(x=x, edge_index=edge_index.t().contiguous())
# >>> Data(edge_index=[2, 4], x=[3, 1])

print(data.keys)
# ['x', 'edge_indx']
#两种方式都可以，
```

```
print(data['x'])
print(data.x)
# >>> tensor([[-1.0],
#             [0.0],
#             [1.0]])
for key, item in data:
    print("{} found in data".format(key))
# edge_indx found in data
# x found in data
#以下几个属性是初始化图之后就自动生成的，不需要人为赋值
data.num_nodes
data.num_edges
data.num_node_features
#是否包含孤立节点
data.contains_isolated_nodes()
#是否包含自连接节点，即自己到自己的边
data.contains_self_loops()
#是否有向图
data.is_directed()
```

标准数据集

载入标准数据集

```
# 加载ENZYMES数据集
from torch_geometric.datasets import TUDataset

dataset = TUDataset(root='/tmp/ENZYMES', name='ENZYMES')

dataset.num_classes  #图分类任务,有6个类别
dataset.num_node_features  #每个节点3个特征
data = dataset[0]
#有600张图,每张图用dataset[i]读出
# >>> Data(edge_index=[2, 168], x=[37, 3], y=[1])
# 168/2 条边,37个节点,3个特征
data.is_undirected() # 无向图

dataset = dataset.shuffle()
# 打乱数据集
# 等价于
perm = torch.randperm(len(dataset))
dataset = dataset[perm]
```

```
# Cora数据集
from torch_geometric.datasets import Planetoid

dataset = Planetoid(root='/tmp/Cora', name='Cora')


data = dataset[0]
# >>> Data(edge_index=[2, 10556], test_mask=[2708], train_mask=[2708], val_mask=
[2708], x=[2708, 1433], y=[2708])
# 10556条边,train_mask 指示了哪些节点用于训练 (140 nodes)
data.is_undirected()
# >>> True
```

```
data.train_mask.sum().item()
# >>> 140
# train_mask 指示了哪些节点用于训练 (140 nodes)
data.val_mask.sum().item()
# >>> 500
# val_mask 指示了哪些节点用于验证，比如用来 只是模型提前结束训练 (500 nodes)
data.test_mask.sum().item()
# >>> 1000
# test_mask 指示了哪些节点用于测试(1000 nodes)
```

小批量训练

神经网络通常使用分批训练。PyG 通过创建稀疏块对角邻接矩阵实现并行化(从edge_index 中定义而来)，并在节点维度上合并特征矩阵和目标矩阵，从而在批处理上实现了并行化。这个组合允许在一个批次中，有不同数量的节点和边。

X1，Xn 是样本的特征矩阵Y1是标签矩阵，A1 A2分别 表示X1 X2的邻接矩阵，通过把他们用对角线的形式拼接在一起，组成一个新的邻接矩阵A。他实际上包含了A1~An的这些图的信息，并且是相互独立没有连接的.

简而言之通过这么处理可以把一个batch的数据存在3个大矩阵当中，而不用每个样本存储一个矩阵，提高计算的效率。

`torch_geometric.data.DataLoader` 会自动完成这个过程

```
from torch_geometric.datasets import TUDataset
from torch_geometric.data import DataLoader

dataset = TUDataset(root='/tmp/ENZYMES', name='ENZYMES', use_node_attr=True)
loader = DataLoader(dataset, batch_size=32, shuffle=True)
for batch in loader:
    print(batch)
    batch.num_graphs
#所有num_graphs都是32个
# Batch(batch=[896], edge_index=[2, 3462], x=[896, 21], y=[32])
# Batch(batch=[1030], edge_index=[2, 3762], x=[1030, 21], y=[32])
# Batch(batch=[1170], edge_index=[2, 4136], x=[1170, 21], y=[32])
# Batch(batch=[913], edge_index=[2, 3532], x=[913, 21], y=[32])
# Batch(batch=[847], edge_index=[2, 3346], x=[847, 21], y=[32])
# Batch(batch=[1248], edge_index=[2, 4612], x=[1248, 21], y=[32])
# Batch(batch=[995], edge_index=[2, 3932], x=[995, 21], y=[32])
# Batch(batch=[978], edge_index=[2, 3814], x=[978, 21], y=[32])
# Batch(batch=[1164], edge_index=[2, 4254], x=[1164, 21], y=[32])
# Batch(batch=[1096], edge_index=[2, 4188], x=[1096, 21], y=[32])
# Batch(batch=[1057], edge_index=[2, 4120], x=[1057, 21], y=[32])
# Batch(batch=[964], edge_index=[2, 3666], x=[964, 21], y=[32])
# Batch(batch=[1041], edge_index=[2, 4012], x=[1041, 21], y=[32])
# Batch(batch=[1072], edge_index=[2, 4112], x=[1072, 21], y=[32])
# Batch(batch=[1134], edge_index=[2, 4240], x=[1134, 21], y=[32])
# Batch(batch=[1144], edge_index=[2, 4514], x=[1144, 21], y=[32])
# Batch(batch=[953], edge_index=[2, 3774], x=[953, 21], y=[32])
# Batch(batch=[1077], edge_index=[2, 4118], x=[1077, 21], y=[32])
# Batch(batch=[801], edge_index=[2, 2970], x=[801, 21], y=[24])
```

torch_geometric.data.Batch 继承自 torch_geometric.data.Data 并且包含了一个额外的属性：data.batch 是一个列向量，它将整个图的所有节点映射到不同的批次中。

$$\text{batch} = [0 \quad \cdots \quad 0 \quad 1 \quad \cdots \quad n-2 \quad n-1 \quad \cdots \quad n-1]^\top$$

你可以使用它，例如，平均每个图的节点维数的节点特征：

```
from torch_geometric.datasets import TUDataset
from torch_geometric.data import DataLoader
from torch_scatter import scatter_mean

dataset = TUDataset(root='/tmp/ENZYMES', name='ENZYMES', use_node_attr=True)
loader = DataLoader(dataset, batch_size=32, shuffle=True)
for data in loader:
    x = scatter_mean(data.x, data.batch, dim=0)
    print(x.size())
```

数据转换

变换是一个在torchvision中变换和增强图像的常用方法。PyG 有自己的转换方法，以Data 对象作为输入，返回一个新的变换过的Data 对象。可以在torch_geometric.transforms.Compose对变换操作进行叠加组合（类似于有torch的Compose），Compose会在将已处理的数据集保存到磁盘上之前(pre_transform)或访问数据集中的图之前(transform)被应用。

还可以使用transform 来进行随机扩展（增广）Data对象

import torch_geometric.transforms as T

实现一个完整的图学习

训练一个GCN

```
import torch
import torch.nn.functional as F
from torch_geometric.nn import GCNConv
from torch_geometric.datasets import Planetoid

dataset = Planetoid(root='/tmp/Cora', name='Cora')


class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # 定义两个卷积层，输入是2个int类型数据，分别代表输入特征和输出特征。
        self.conv1 = GCNConv(dataset.num_node_features, 16)
        self.conv2 = GCNConv(16, dataset.num_classes)

    # 前向传播（nn.Module会自动根据前向传播来实现反向传播）
    def forward(self, data):
        # 还记得data.x吗？它是节点特征矩阵，edge_index则 是我们要输入的图。如忘记了，不妨再去上面翻翻。
        x, edge_index = data.x, data.edge_index
        # 开始执行图卷积操作，在GCNConv的forward方法中，会自动适用输入的图，类死于edge_index这样格式，或者邻接矩阵都可以。
        x = self.conv1(x, edge_index)
```

```
        x = F.relu(x)
        # 丢弃权重默认为0.5
        x = F.dropout(x, training=self.training)
        # 表面上看第二层和第一层卷积没什么区别，
        # 但是别忘了，在上面我们定义他们的时候，他们的输入输出维度是不同的。所以第二层执行完以后列的维度就是dataset.num_classes了，这样就完成了多个特征到指定类别特征的映射。
        x = self.conv2(x, edge_index)
        # 在列上进行归一化
        return F.log_softmax(x, dim=1)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = Net().to(device)
data = dataset[0].to(device)
#设置优化器，优化器会帮我们管理模型的所有参数的更新。
optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=5e-4)

model.train()
for epoch in range(200):
    optimizer.zero_grad()
    out = model(data)
    #计算交叉熵损失，因为已经做过归一化了。就直接使用NLLloss了。
    loss = F.nll_loss(out[data.train_mask], data.y[data.train_mask])
    print(f"Loss:{loss.item()}")
    #反向传播
    loss.backward()
    #更新参数
    optimizer.step()

#切换到测试模式
model.eval()
_, pred = model(data).max(dim=1)
correct = int(pred[data.test_mask].eq(data.y[data.test_mask]).sum().item())
acc = correct / int(data.test_mask.sum())
print('Accuracy: {:.4f}'.format(acc))
```

创建消息传播神经网络

消息传播网络是GNN的通用框架之一，所谓通用框架是对多种变体GNN网络结构的一般总结，也就是GNN编程的通用范式。

这里简单介绍三类通用框架：

1. 消息传播神经网络（Message Passing Neural Network,MPNN）:基本思路是：节点的表示向量都是通过消息函数M（Message）和更新函数U（update）进行K轮消息传播机制的迭代后得到的。这个说法其实和我们理解的一般的CNN是很相似的。图卷积最常用的GCN（简化的谱域卷积模型），R-GCN（GCN的进阶版。可以处理异构图，即有多种节点关系的图），GraphSAGE（大名鼎鼎的SAGE）,都可以归结到此类模型。
2. 非局部神经网络（Non-Local Neural Network,NLNN）：NLNN是对注意力机制的一般化总结，GAT（图注意力模型，现在用的比较多，效果也比较不错，这个也会补），可以看作是NLNN的一个特例。
3. 图网络（Graph Network）相比较于MPNN和NLNN，对GNN做出了更一般化的总结，这个框架包含了节点状态，边状态，图状态的三个元素，因此可以分别对应节点任务，边任务，图任务。可以说是对GNN的一个全面包含的框架。

MPNN

将卷积算子推广到不规则域通常表示为邻域聚合或消息传递方案。用$x_i^{(k-1)} \in R^F$表示节点i在k-1层的节点特征，表$e_{j,i} \in R^D$表示（可选）从节点j到节点i的边特征，消息传递图神经网络可以描述为：
$\mathbf{x}_i^{(k)} = \gamma^{(k)} \left(\mathbf{x}_i^{(k-1)}, \square_{j \in \mathcal{N}(i)} \phi^{(k)} \left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{j,i} \right) \right)$

方框□表示一种可微的排列不变的函数。例如：求和，求均值，取最大值等。ϕ表示可微函数，例如MLPs（多层感知机）。

“消息传递”的基本类

PyG提供了消息传递的基本类，这帮助我们创建多种消息传递神经网络，并且自动实现消息传递。

用户只需要定义 ϕ函数，如message(),γ函数，如update()，以及聚合模式：aggr=“add”, aggr=“mean” or aggr=“max”.
这是通过以下方法完成的：

1. MessagePassing(aggr=“add”, flow=“source_to_target”, node_dim=-2)：定义聚合模式，消息传递的流方向，（soource to target 或者target to source）.此外，node_dim 属性指示了沿着哪个轴传播。
2. MessagePassing.propagate(edge_index, size=None, **kwargs): 开始调用消息的初试传递。获取边缘索引和所有构建消息和更新节点嵌入所需的附加数据。注意propagate() 不仅为形状对称的[N,N]邻接矩阵交换消息，也可以用于一般稀疏矩阵,例如形状为[N,M]的二分图矩阵只需要传递一个size=[N,M],如果不传递size参数，会默认传入为对称矩阵。对于具有两个独立节点集和指标集，且每个集合都有自己的信息的二部图，这种分裂可以通过将信息作为元组传递来标记,例如x=(x_N,x_M).
3. MessagePassing.message(...):
   构造消息到节点 i 类似于 ϕ对于每个边缘 (j,i)∈E若flow="source_to_target"和(i,j)∈E若flow=“target_to_source”。可以接受最初传递给的任何参数propagate()。此外，propagate()可以将传递给的张量映射到各个节点i 和 j通过在变量名称（例如.eg和）之后附加_i或。请注意，我们通常是指_j x_ix_ji 作为聚集信息的中央节点，并指 j 作为相邻节点，因为这是最常见的表示法

实现GCN层

GCN层的数学定义：

$\mathbf{x}_i^{(k)} = \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{\deg(i)} \cdot \sqrt{\deg(j)}} \cdot \left(\mathbf{\Theta} \cdot \mathbf{x}_j^{(k-1)} \right)$

首先通过权重矩阵 $\mathbf{\Theta}$对相邻节点特征进行变换，按他们的程度归一化，最后累加起来。该公式可以分为以下步骤：

1. 为邻接矩阵添加自循环（加上单位矩阵I）
2. 线性变换节点特征矩阵
3. 计算归一化系数
4. 正则化 ϕ中的节点特征
5. 累加邻居节点的特征（add的聚合操作）

步骤1-3通常在消息传递发生之前计算。可以使用MessagePassing基类轻松地处理步骤4-5。

```
import torch
from torch_geometric.nn import MessagePassing
from torch_geometric.utils import add_self_loops,degree

class GCNConv(MessagePassing):
    def __init__(self,in_channels,out_channels):
```

```
        super(GCNConv,self).__init__(aggr='add')
        self.linear = torch.nn.Linear(in_channels,out_channels)
    def forward(self,x,edge_index):
        # x维度[N, in_channels]  N为节点数,in_channels输入特征个数
        # edge_index维度[2, E]  E为边数

        # Step 1: Add self-loops to the adjacency matrix.加自环
        edge_index,_ = add_self_loops(edge_index,num_nodes=x.size(0))

        # Step 2: Linearly transform node feature matrix.线性变换节点特征矩阵
        x = self.linear(x)
        # Step 3: Compute normalization.计算归一化系数
        row,col = edge_index
        deg = degree(col,x.size(0),dtype=x.dtype)
        deg_inv_sqrt = deg.pow(-0.5)
        norm = deg_inv_sqrt[row]*deg_inv_sqrt[col]
        # Step 4-5: Start propagating messages.正则化节点特征,累加邻居节点的特征（add的聚合操作）。
        return self.propagate(edge_index,x=x,norm=norm)
    def message(self, x_j,norm):
        # x_j维度[E, out_channels]  E为边数,out_channels输出特征个数
        # Step 4: Normalize node features. view是把张量变形，比如此处就是把norm列规定为1
        return norm.view(-1,1)*x_j

    # x_j表示提升的张量，其中包含每个边缘的源节点特征，即每个节点的邻居。
    # 节点功能可以通过在变量名称后添加_i或_j来自动提升。
    # 任何张量都可以通过这种方式转换，只要它们具有源或目标节点特征即可。
conv = GCNConv(16, 32)
x = conv(x, edge_index)
```

实现图边卷积

图边卷积或者点云的数学定义：

$$\mathbf{x}_i^{(k)} = \max_{j \in \mathcal{N}(i)} h_\Theta \left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)} - \mathbf{x}_i^{(k-1)} \right)$$

h_Θ表示MLP,类似于GCN层，我们同样使用MessagePassing class 来实现这个层，这次会使用max聚合

```
import torch
from torch.nn import Sequential as Seq,Linear,ReLU
from torch_geometric.nn import MessagePassing

class EdgeConv(MessagePassing):
    def __init__(self,in_channels,out_channels):
        super(EdgeConv,self).__init__(aggr='max')
        self.mlp =
Seq(Linear(2*in_channels,out_channels),ReLU,Linear(out_channels,out_channels))

    def forward(self,x,edge_index):
        return self.propagate(edge_index,x = x)
    def message(self, x_i,x_j):
        tmp = torch.cat([x_i,x_j - x_i],dim=1)
        return self.mlp(tmp)
```

在message()函数内部，我们用于self.mlp转换每个边的目标节点特征x_i和相对源节点特征x_j - x_i
(j,i)∈E

边缘卷积实际上是一种动态卷积，它使用特征空间中的最近邻居来重新计算每一层的图形。幸运的是，PyG带有GPU加速的分批k-NN图形

生成方法，名为torch_geometric.nn.pool.knn_graph()

```
import torch
from torch.nn import Sequential as Seq,Linear,ReLU
from torch_geometric.nn import MessagePassing
from  torch_geometric.nn import knn_graph

class EdgeConv(MessagePassing):
    def __init__(self,in_channels,out_channels):
        super(EdgeConv,self).__init__(aggr='max')
        self.mlp =
Seq(Linear(2*in_channels,out_channels),ReLU,Linear(out_channels,out_channels))

    def forward(self,x,edge_index):
        return self.propagate(edge_index,x = x)
    def message(self, x_i,x_j):
        tmp = torch.cat([x_i,x_j - x_i],dim=1)
        return self.mlp(tmp)
class DynamicEdgeConv(EdgeConv):
    def __init__(self,in_channels,out_channels,k=6):
        super(DynamicEdgeConv,self).__init__(in_channels,out_channels)
        self.k = k
    def forward(self,x,batch=None):
        edge_index = knn_graph(x,self.k,batch,loop=False,flow=self.flow)
        return super(DynamicEdgeConv,self).forward(x,edge_index)
conv = DynamicEdgeConv(3, 128, k=6)
x = conv(x, batch)
```

构建自己的数据集

为数据集提供了两个抽象类：torch_geometric.data.Dataset和
torch_geometric.data.InMemoryDataset

后者继承自前者，若整个数据集都能放进内存，则应该使用后者。

按照tochvision约定，每个数据集都会设置一个根文件夹。在此文件夹下，又设有raw_dir用来存放下载的原始数据，processede_dir用来保存处理过的数据集。
可以为每个数据设置transform,pre_transform,pre_filter函数，默认是None.

- transform:在访问之前动态转换数据对象，一般用来做数据增强。
- pre_transform:用于将数据对象保存到磁盘之前的转换，建议用它来做繁重的数据预处理（只需执行一次）
- pre_filter:可以在保存数据之前手动过滤掉特定的数据对象

创建内存数据集

为了创建**torch_geometric.data.InMemoryDataset**，需要实现四种基本方法：

- torch_geometric.data.InMemoryDataset.raw_file_names()：储存源文件的列表。如果这些文件都在raw_dir中找到了，就会跳过下载。

- torch_geometric.data.InMemoryDataset.processed_file_names()：储存处理后的文件列表，若全部找到就跳过预处理。
- torch_geometric.data.InMemoryDataset.download()：将raw_file_names中的文件列表，下载到raw_dir
- torch_geometric.data.InMemoryDataset.process()：处理原始数据，并保存到processed_dir

更详细的可以查看torch_geometric.data.

这四个函数中，最关键的是process()，在这个方法中，我们需要读取并且创建一系列的Data对象，然后将其保存到processed_dir. 由于保存一个巨大的python列表相当耗时，因此在保存之前，一般会将上述列表整理为一个巨大的Data对象，通过**torch_geometric.data.InMemoryDataset.collate()**来实现，collate()方法，可以理解为用空间换时间。
这个整理后的数据对象将所有的例子连接成一个大的数据对象data，然后返回了slices字典以从该对象中重构单个示例。最后，将构造函数中的这2个对象，加载到属性self.data 和slef.slices。

```
import torch
from torch_geometric.data import InMemoryDataset

class MyOwnDataset(InMemoryDataset):
    def __init__(self,root,transform = None,pre_transform=None):
        super(MyOwnDataset,self).__init__(root,transform,pre_transform)
        self.data,self.slices = torch.load(self.processed_paths[0])
    @property
    def raw_file_names(self):
        return ['some_file_1','some_file_2',...]
    @property
    def processed_file_names(self):
        return ['data.pt']
    def download(self):
        # Download to 'self.raw_dir'
        pass
    def process(self):
        data_list=[]
        if self.pre_filter is not None:
            data_list=[data for data in data_list if self.pre_filter(data)]
        if self.pre_transform is not None:
            data_list=[self.pre_transform(data) for data in data_list]

        data,slice = self.collate(data_list)
        torch.save((data,slice),self.processed_paths[0])
```

创建更大的数据集

为了创建不适合内存的数据集，可以使用torch_geometric.data.Dataset，它和torchvision的datasets是相当接近的。只是需要额外实现2个方法：

- torch_geometric.data.Dataset.len()：返回数据集中的样本数，
- torch_geometric.data.Dataset.get()：实现加载的单个图的逻辑。

在内部，torch_geometric.data.Dataset.**getitem**() 从torch_geometric.data.Dataset.get()获取数据对象，并根据transform中规定的转换方法进行转换。
同样看一个简单示例：

```
import os.path as osp

import torch
from torch_geometric.data import Dataset
```

```
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`.

    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
```

每个图形数据对象分别保存在中process()，并手动加载到中get()