5.4 整图分类

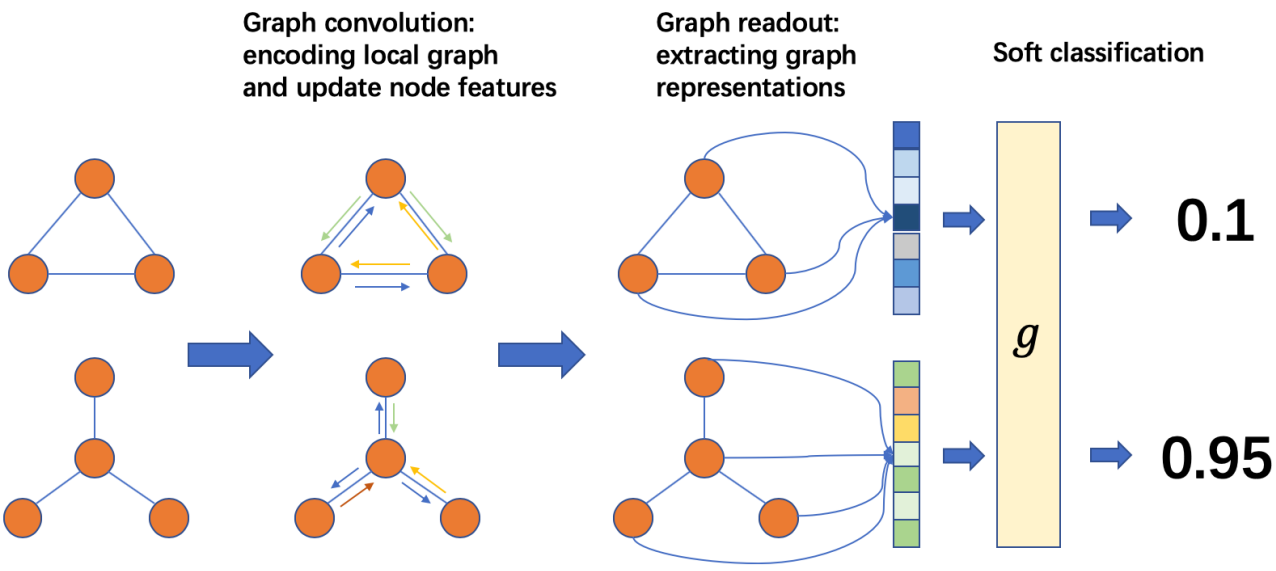
(English Version)

许多场景中的图数据是由多个图组成，而不是单个的大图数据。例如不同类型的人群社区。通过用图刻画同一社区里人与人间的友谊，可以得到多张用于分类的图。 在这个场景里，整图分类模型可以识别社区的类型，即根据结构和整体信息对图进行分类。

概述

整图分类与节点分类或链接预测的主要区别是：预测结果刻画了整个输入图的属性。 与之前的任务类似，用户还是在节点或边上进行消息传递。但不同的是，整图分类任务还需要得到整个图的表示。

整图分类的处理流程如下图所示：



整图分类流程

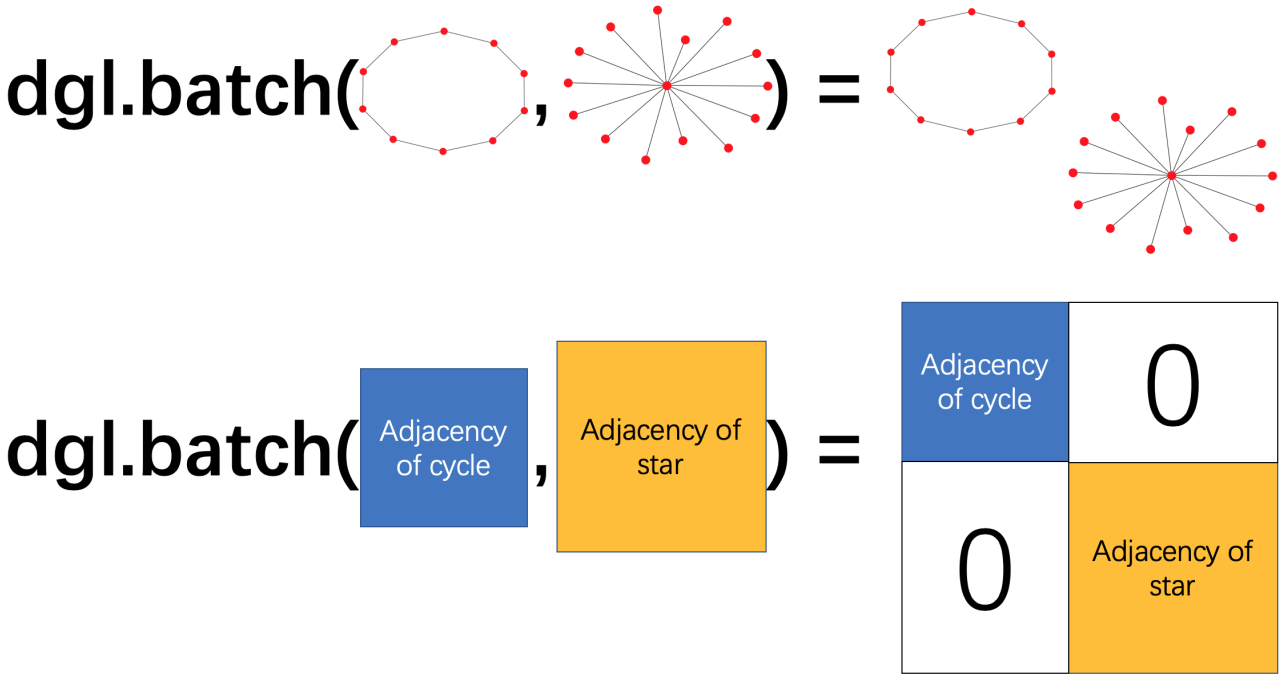
从左至右，一般流程是：

- 准备一个批次的图；
- 在这个批次的图上进行消息传递以更新节点或边的特征；
- 将一张图里的节点或边特征聚合成整张图的图表示；
- 根据任务设计分类层。

批次的图

整图分类任务通常需要在很多图上进行训练。如果用户在训练模型时一次仅使用一张图，训练效率会很低。 借用深度学习实践中常用的小批次训练方法，用户可将多张图组成一个批次，在整个图批次上进行一次训练迭代。

使用DGL，用户可将一系列的图建立成一个图批次。一个图批次可以被看作是一张大图，图中的每个连通子图对应一张原始小图。



批次化的图

需要注意，DGL里对图进行变换的函数会去掉图上的批次信息。用户可以通过 `dgl.DGLGraph.set_batch_num_nodes()` 和 `dgl.DGLGraph.set_batch_num_edges()` 两个函数在变换后的图上重新加入批次信息。

图读出

数据集中的每一张图都有它独特的结构和节点与边的特征。为了完成单个图的预测，通常会聚合并汇总单个图尽可能多的信息。 这类操作叫做“读出”。常见的聚合方法包括：对所有节点或边特征求和、取平均值、逐元素求最大值或最小值。

给定一张图 g，对它所有节点特征取平均值的聚合读出公式如下：

$$h_g = \frac{1}{|\mathcal{V}|}\sum_{v\in\mathcal{V}} h_v$$

其中，h_g 是图 g 的表征， $\mathcal{V}$ 是图 g 中节点的集合， h_v 是节点 v 的特征。

DGL内置了常见的图读出函数，例如 `dgl.readout_nodes()` 就实现了上述的平均值读出计算。

在得到 h_g 后，用户可将其传给一个多层感知机(MLP)来获得分类输出。

编写神经网络模型

模型的输入是带节点和边特征的批次化图。需要注意的是批次化图中的节点和边属性没有批次大小对应的维度。 模型中应特别注意以下几点。

批次化图上的计算

首先，一个批次中不同的图是完全分开的，即任意两个图之间没有边连接。 根据这个良好的性质，所有消息传递函数(的计算)仍然具有相同的结果。

其次，读出函数会分别作用在图批次中的每张图上。假设批次大小为 B，要聚合的特征大小为 D， 则图读出的张量形状为 (B, D)。

```
import dgl
import torch

g1 = dgl.graph(([0, 1], [1, 0]))
g1.ndata['h'] = torch.tensor([1., 2.])
g2 = dgl.graph(([0, 1], [1, 2]))
g2.ndata['h'] = torch.tensor([1., 2., 3.])

dgl.readout_nodes(g1, 'h')
# tensor([3.])  # 1 + 2

bg = dgl.batch([g1, g2])
dgl.readout_nodes(bg, 'h')
# tensor([3., 6.])  # [1 + 2, 1 + 2 + 3]
```

最后，批次化图中的每个节点或边特征张量均通过将所有图上的相应特征拼接得到。

```
bg.ndata['h']
# tensor([1., 2., 1., 2., 3.])
```

模型定义

了解了上述计算规则后，用户可以定义一个非常简单的模型。

```
import dgl.nn.pytorch as dglnn
import torch.nn as nn

class Classifier(nn.Module):
    def __init__(self, in_dim, hidden_dim, n_classes):
        super(Classifier, self).__init__()
        self.conv1 = dglnn.GraphConv(in_dim, hidden_dim)
        self.conv2 = dglnn.GraphConv(hidden_dim, hidden_dim)
        self.classify = nn.Linear(hidden_dim, n_classes)

    def forward(self, g, h):
        # 应用图卷积和激活函数
        h = F.relu(self.conv1(g, h))
        h = F.relu(self.conv2(g, h))
        with g.local_scope():
            g.ndata['h'] = h
            # 使用平均读出计算图表示
            hg = dgl.mean_nodes(g, 'h')
            return self.classify(hg)
```

模型的训练

数据加载

模型定义完成后，用户就可以开始训练模型。由于整图分类处理的是很多相对较小的图，而不是一个大图， 因此通常可以在随机抽取的小批次图上进行高效的训练，而无需设计复杂的图采样算法。

以下例子中使用了 第4章：图数据处理管道 中的整图分类数据集。

```
import dgl.data
dataset = dgl.data.GINDataset('MUTAG', False)
```

整图分类数据集里的每个数据点是一个图和它对应标签的元组。为提升数据加载速度， 用户可以调用GraphDataLoader，从而以小批次遍历整个图数据集。

```
from dgl.dataloading import GraphDataLoader
dataloader = GraphDataLoader(
    dataset,
    batch_size=1024,
    drop_last=False,
    shuffle=True)
```

训练过程包括遍历dataloader和更新模型参数的部分。

```
import torch.nn.functional as F

# 这仅是个例子，特征尺寸是7
model = Classifier(7, 20, 5)
opt = torch.optim.Adam(model.parameters())
for epoch in range(20):
    for batched_graph, labels in dataloader:
        feats = batched_graph.ndata['attr']
        logits = model(batched_graph, feats)
        loss = F.cross_entropy(logits, labels)
        opt.zero_grad()
        loss.backward()
        opt.step()
```

DGL实现了一个整图分类的样例： DGL的GIN样例。 模型训练的代码请参考位于 main.py 源文件中的 `train` 函数。 模型实现位于 gin.py ， 其中使用了更多的模块组件，例如使用 `dgl.nn.pytorch.GINConv` 模块作为图卷积层(DGL同样支持它在MXNet和TensorFlow后端里的实现)、批量归一化等。

异构图上的整图分类模型的训练

在异构图上做整图分类和在同构图上做整图分类略有不同。用户除了需要使用异构图卷积模块，还需要在读出函数中聚合不同类别的节点。

以下代码演示了如何对每种节点类型的节点表示取平均值并求和。

```
class RGCN(nn.Module):
    def __init__(self, in_feats, hid_feats, out_feats, rel_names):
        super().__init__()

        self.conv1 = dglnn.HeteroGraphConv({
            rel: dglnn.GraphConv(in_feats, hid_feats)
            for rel in rel_names}, aggregate='sum')
        self.conv2 = dglnn.HeteroGraphConv({
            rel: dglnn.GraphConv(hid_feats, out_feats)
            for rel in rel_names}, aggregate='sum')

    def forward(self, graph, inputs):
        # inputs是节点的特征
        h = self.conv1(graph, inputs)
        h = {k: F.relu(v) for k, v in h.items()}
        h = self.conv2(graph, h)
        return h

class HeteroClassifier(nn.Module):
    def __init__(self, in_dim, hidden_dim, n_classes, rel_names):
        super().__init__()

        self.rgcn = RGCN(in_dim, hidden_dim, hidden_dim, rel_names)
        self.classify = nn.Linear(hidden_dim, n_classes)

    def forward(self, g):
        h = g.ndata['feat']
        h = self.rgcn(g, h)
        with g.local_scope():
            g.ndata['h'] = h
            # 通过平均读出值来计算单图的表征
            hg = 0
            for ntype in g.ntypes:
                hg = hg + dgl.mean_nodes(g, 'h', ntype=ntype)
            return self.classify(hg)
```

剩余部分的训练代码和同构图代码相同。

```
# etypes是一个列表，元素是字符串类型的边类型
model = HeteroClassifier(10, 20, 5, etypes)
opt = torch.optim.Adam(model.parameters())
for epoch in range(20):
    for batched_graph, labels in dataloader:
        logits = model(batched_graph)
        loss = F.cross_entropy(logits, labels)
        opt.zero_grad()
        loss.backward()
        opt.step()
```