6.2 针对边分类任务的邻居采样训练方法

(English Version)

边分类/回归的训练与节点分类/回归的训练类似，但还是有一些明显的区别。

定义邻居采样器和数据加载器

用户可以使用 和节点分类一样的邻居采样器。

```
sampler = dgl.dataloading.MultiLayerFullNeighborSampler(2)
```

想要用DGL提供的邻居采样器做边分类，需要将其与 `EdgeDataLoader` 结合使用。 `EdgeDataLoader` 以小批次的形式对一组边进行迭代， 从而产生包含边小批次的子图以及供下文中模块使用的 `块` 。

例如，以下代码创建了一个PyTorch数据加载器，该PyTorch数据加载器以批的形式迭代训练边ID的数组 `train_eids` ，并将生成的块列表放到GPU上。

```
dataloader = dgl.dataloading.EdgeDataLoader(
    g, train_eid_dict, sampler,
    batch_size=1024,
    shuffle=True,
    drop_last=False,
    num_workers=4)
```

有关DGL的内置采样器的完整列表，用户可以参考 neighborhood sampler API reference。

如果用户希望开发自己的邻居采样器，或者想要对块的概念有更详细的了解，请参考 6.4 定制用户自己的邻居采样器。

小批次邻居采样训练时删边

用户在训练边分类模型时，有时希望从计算依赖中删除出现在训练数据中的边，就好像这些边根本不存在一样。 否则，模型将 “知道” 两个节点之间存在边的联系，并有可能利用这点 “作弊” 。

因此，在基于邻居采样的边分类中，用户有时会希望从采样得到的小批次图中删去部分边及其对应的反向边。 用户可以在实例化 `EdgeDataLoader` 时设置 `exclude='reverse_id'`，同时将边ID映射到其反向边ID。 通常这样做会导致采样过程变慢很多，这是因为DGL要定位并删除包含在小批次中的反向边。

```
n_edges = g.num_edges()
dataloader = dgl.dataloading.EdgeDataLoader(
    g, train_eid_dict, sampler,

    # 下面的两个参数专门用于在邻居采样时删除小批次的一些边和它们的反向边
    exclude='reverse_id',
    reverse_eids=torch.cat([
        torch.arange(n_edges // 2, n_edges), torch.arange(0, n_edges // 2)]),

    batch_size=1024,
    shuffle=True,
    drop_last=False,
    num_workers=4)
```

调整模型以适用小批次训练

边分类模型通常由两部分组成：

- 获取边两端节点的表示。
- 用边两端节点表示为每个类别打分。

第一部分与 随机批次训练节点分类 完全相同，用户可以简单地复用它。输入仍然是DGL的数据加载器生成的块列表和输入特征。

```
class StochasticTwoLayerGCN(nn.Module):
    def __init__(self, in_features, hidden_features, out_features):
        super().__init__()
        self.conv1 = dglnn.GraphConv(in_features, hidden_features)
        self.conv2 = dglnn.GraphConv(hidden_features, out_features)

    def forward(self, blocks, x):
        x = F.relu(self.conv1(blocks[0], x))
        x = F.relu(self.conv2(blocks[1], x))
        return x
```

第二部分的输入通常是前一部分的输出，以及由小批次边导出的原始图的子图。 子图是从相同的数据加载器产生的。用户可以调用 `dgl.DGLGraph.apply_edges()` 计算边子图中边的得分。

以下代码片段实现了通过合并边两端节点的特征并将其映射到全连接层来预测边的得分。

```
class ScorePredictor(nn.Module):
    def __init__(self, num_classes, in_features):
        super().__init__()
        self.W = nn.Linear(2 * in_features, num_classes)

    def apply_edges(self, edges):
        data = torch.cat([edges.src['x'], edges.dst['x']], 1)
        return {'score': self.W(data)}

    def forward(self, edge_subgraph, x):
        with edge_subgraph.local_scope():
            edge_subgraph.ndata['x'] = x
            edge_subgraph.apply_edges(self.apply_edges)
            return edge_subgraph.edata['score']
```

模型接受数据加载器生成的块列表、边子图以及输入节点特征进行前向传播，如下所示：

```
class Model(nn.Module):
    def __init__(self, in_features, hidden_features, out_features, num_classes):
        super().__init__()
        self.gcn = StochasticTwoLayerGCN(
            in_features, hidden_features, out_features)
        self.predictor = ScorePredictor(num_classes, out_features)

    def forward(self, edge_subgraph, blocks, x):
        x = self.gcn(blocks, x)
        return self.predictor(edge_subgraph, x)
```

DGL保证边子图中的节点与生成的块列表中最后一个块的输出节点相同。

模型的训练

模型的训练与节点分类的随机批次训练的情况非常相似。用户可以遍历数据加载器以获得由小批次边组成的子图，以及计算其两端节点表示所需的块列表。

```
model = Model(in_features, hidden_features, out_features, num_classes)
model = model.cuda()
opt = torch.optim.Adam(model.parameters())

for input_nodes, edge_subgraph, blocks in dataloader:
    blocks = [b.to(torch.device('cuda')) for b in blocks]
    edge_subgraph = edge_subgraph.to(torch.device('cuda'))
    input_features = blocks[0].srcdata['features']
    edge_labels = edge_subgraph.edata['labels']
    edge_predictions = model(edge_subgraph, blocks, input_features)
    loss = compute_loss(edge_labels, edge_predictions)
    opt.zero_grad()
    loss.backward()
    opt.step()
```

异构图上的模型训练

在异构图上，计算节点表示的模型也可以用于计算边分类/回归所需的两端节点的表示。

```
class StochasticTwoLayerRGCN(nn.Module):
    def __init__(self, in_feat, hidden_feat, out_feat, rel_names):
        super().__init__()
        self.conv1 = dglnn.HeteroGraphConv({
                rel : dglnn.GraphConv(in_feat, hidden_feat, norm='right')
                for rel in rel_names
            })
        self.conv2 = dglnn.HeteroGraphConv({
                rel : dglnn.GraphConv(hidden_feat, out_feat, norm='right')
                for rel in rel_names
            })

    def forward(self, blocks, x):
        x = self.conv1(blocks[0], x)
        x = self.conv2(blocks[1], x)
        return x
```

在同构图和异构图上做评分预测时，代码实现的唯一不同在于调用 `apply_edges()` 时需要在特定类型的边上进行迭代。

```
class ScorePredictor(nn.Module):
    def __init__(self, num_classes, in_features):
        super().__init__()
        self.W = nn.Linear(2 * in_features, num_classes)

    def apply_edges(self, edges):
        data = torch.cat([edges.src['x'], edges.dst['x']], 1)
        return {'score': self.W(data)}

    def forward(self, edge_subgraph, x):
        with edge_subgraph.local_scope():
            edge_subgraph.ndata['x'] = x
            for etype in edge_subgraph.canonical_etypes:
                edge_subgraph.apply_edges(self.apply_edges, etype=etype)
            return edge_subgraph.edata['score']

class Model(nn.Module):
    def __init__(self, in_features, hidden_features, out_features, num_classes,
                 etypes):
        super().__init__()
        self.rgcn = StochasticTwoLayerRGCN(
            in_features, hidden_features, out_features, etypes)
        self.pred = ScorePredictor(num_classes, out_features)

    def forward(self, edge_subgraph, blocks, x):
        x = self.rgcn(blocks, x)
        return self.pred(edge_subgraph, x)
```

数据加载器的定义也与节点分类的非常相似。唯一的区别是用户需要使用 `EdgeDataLoader` 而不是 `NodeDataLoader`， 并且提供边类型和边ID张量的字典，而不是节点类型和节点ID张量的字典。

```
sampler = dgl.dataloading.MultiLayerFullNeighborSampler(2)
dataloader = dgl.dataloading.EdgeDataLoader(
    g, train_eid_dict, sampler,
    batch_size=1024,
    shuffle=True,
    drop_last=False,
    num_workers=4)
```

如果用户希望删除异构图中的反向边，情况会有所不同。在异构图上， 反向边通常具有与正向边本身不同的边类型，以便区分 `向前` 和 `向后` 关系。 例如， `关注` 和 `被关注` 是一对相反的关系， `购买` 和 `被买下` 也是一对相反的关系。

如果一个类型中的每个边都有一个与之对应的ID相同、属于另一类型的反向边， 则用户可以指定边类型及其反向边类型之间的映射。删除小批次中的边及其反向边的方法如下。

```
dataloader = dgl.dataloading.EdgeDataLoader(
    g, train_eid_dict, sampler,

    # 下面的两个参数专门用于在邻居采样时删除小批次的一些边和它们的反向边
    exclude='reverse_types',
    reverse_etypes={'follow': 'followed by', 'followed by': 'follow',
                    'purchase': 'purchased by', 'purchased by': 'purchase'}

    batch_size=1024,
    shuffle=True,
    drop_last=False,
    num_workers=4)
```

除了 `compute_loss` 的代码实现有所不同，异构图的训练循环与同构图中的训练循环几乎相同， 计算损失函数接受节点类型和预测的两个字典。

```
model = Model(in_features, hidden_features, out_features, num_classes, etypes)
model = model.cuda()
opt = torch.optim.Adam(model.parameters())

for input_nodes, edge_subgraph, blocks in dataloader:
    blocks = [b.to(torch.device('cuda')) for b in blocks]
    edge_subgraph = edge_subgraph.to(torch.device('cuda'))
    input_features = blocks[0].srcdata['features']
    edge_labels = edge_subgraph.edata['labels']
    edge_predictions = model(edge_subgraph, blocks, input_features)
    loss = compute_loss(edge_labels, edge_predictions)
    opt.zero_grad()
    loss.backward()
    opt.step()
```

GCMC 是一个在二分图上做边分类的代码示例。