6.1 针对节点分类任务的邻居采样训练方法

(English Version)

为了随机(批次)训练模型，需要进行以下操作：

- 定义邻居采样器。
- 调整模型以进行小批次训练。
- 修改模型训练循环部分。

以下小节将逐一介绍这些步骤。

定义邻居采样器和数据加载器

DGL提供了几个邻居采样类，这些类会生成需计算的节点在每一层计算时所需的依赖图。

最简单的邻居采样器是 `MultiLayerFullNeighborSampler` ，它可获取节点的所有邻居。

要使用DGL提供的采样器，还需要将其与 `NodeDataLoader` 结合使用，后者可以以小批次的形式对一个节点的集合进行迭代。

例如，以下代码创建了一个PyTorch的 DataLoader，它分批迭代训练节点ID数组 `train_nids` ， 并将生成的子图列表放到GPU上。

```
import dgl
import dgl.nn as dglnn
import torch
import torch.nn as nn
import torch.nn.functional as F

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

对DataLoader进行迭代，将会创建一个特定图的列表，这些图表示每层的计算依赖。在DGL中称之为 *块*。

```
input_nodes, output_nodes, blocks = next(iter(dataloader))
print(blocks)
```

上面的dataloader一次迭代会生成三个输出。 `input nodes` 代表计算 `output nodes` 的表示所需的节点。 `块` 包含了每个GNN层要计算哪些节点表示作为输出，要将哪些节点表示作为输入，以及来自输入节点的表示如何传播到输出节点。

完整的内置采样方法清单，用户可以参考 neighborhood sampler API reference。

如果用户希望编写自己的邻居采样器，或者想要关于块的更深入的介绍，读者可以参考 6.4 定制用户自己的邻居采样器。

调整模型以进行小批次训练

如果用户的消息传递模块全使用的是DGL内置模块，则模型在进行小批次训练时只需做很小的调整。 以多层GCN为例。如果用户模型在全图上是按以下方式实现的：

```
class TwoLayerGCN(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, g, x):
        x = F.relu(self.conv1(g, x))
        x = F.relu(self.conv2(g, x))
        return x
```

然后，用户所需要做的就是用上面生成的块(`block`)来替换图(`g`)。

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

上面的DGL `GraphConv` 模块接受的一个参数是数据加载器生成的 `块` 中的一个元素。

用户可以查阅 NN模块的API参考 来查看DGL的内置模型模块是否支持接受 `块` 作为参数。

如果希望使用自定义的消息传递模块，用户可以参考 6.5 为小批次训练实现定制化的GNN模块。

模型的训练

这里的模型的训练循环仅包含使用定制的批处理迭代器遍历数据集的内容。在每个生成块列表的迭代中：

1. 将与输入节点相对应的节点特征加载到GPU上。节点特征可以存储在内存或外部存储中。请注意，用户只需要加载输入节点的特征，而不是像整图训练那样加载所有节点的特征。

   如果特征存储在 `g.ndata` 中，则可以通过 `blocks[0].srcdata` 来加载第一个块的输入节点的特征， 这些节点是计算节点最终表示所需的所有必需的节点。

2. 将块列表和输入节点特征传入多层GNN并获取输出。
3. 将与输出节点相对应的节点标签加载到GPU上。同样，节点标签可以存储在内存或外部存储器中。 再次提醒下，用户只需要加载输出节点的标签，而不是像整图训练那样加载所有节点的标签。

   如果特征存储在 `g.ndata` 中，则可以通过访问 `blocks[-1].dstdata` 中的特征来加载标签，它是最后一个块的输出节点的特征，这些节点与用户希望计算最终表示的节点相同。

4. 计算损失并反向传播。

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

for input_nodes, output_nodes, blocks in dataloader:
    blocks = [b.to(torch.device('cuda')) for b in blocks]
    input_features = blocks[0].srcdata['features']
    output_labels = blocks[-1].dstdata['label']
    output_predictions = model(blocks, input_features)
    loss = compute_loss(output_labels, output_predictions)
    opt.zero_grad()
    loss.backward()
    opt.step()
```

DGL提供了一个端到端的随机批次训练示例 GraphSAGE的实现。

异构图上模型的训练

在异构图上训练图神经网络进行节点分类的方法也是类似的。

例如，在 异构图上的节点分类模型的训练 中介绍了如何在整图上训练一个2层的RGCN模型。RGCN小批次训练的代码与它非常相似(为简单起见，这里删除了自环、非线性和基分解)：

```
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
```

DGL提供的一些采样方法也支持异构图。例如，用户仍然可以使用`MultiLayerFullNeighborSampler`类和`NodeDataLoader`类进行随机批次训练。对于全邻居采样，唯一的区别是用户需要为训练集指定节点类型和节点ID的字典。

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

模型的训练与同构图几乎相同。不同之处在于，`compute_loss`的实现会包含两个字典：节点类型和预测结果。

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

for input_nodes, output_nodes, blocks in dataloader:
    blocks = [b.to(torch.device('cuda')) for b in blocks]
    input_features = blocks[0].srcdata     # returns a dict
    output_labels = blocks[-1].dstdata     # returns a dict
    output_predictions = model(blocks, input_features)
    loss = compute_loss(output_labels, output_predictions)
    opt.zero_grad()
    loss.backward()
    opt.step()
```

DGL提供了端到端随机批次训练的 RGCN的实现。