6.5 为小批次训练实现定制化的GNN模块

(English Version)

如果用户熟悉如何定制用于更新整个同构图或异构图的GNN模块(参见 第3章：构建图神经网络（GNN）模块)，那么在块上计算的代码也是类似的，区别只在于节点被划分为输入节点和输出节点。

以下面的自定义图卷积模块代码为例。注意，该代码并不一定是最高效的实现， 此处只是将其作为自定义GNN模块的一个示例。

```
class CustomGraphConv(nn.Module):
    def __init__(self, in_feats, out_feats):
        super().__init__()
        self.W = nn.Linear(in_feats * 2, out_feats)

    def forward(self, g, h):
        with g.local_scope():
            g.ndata['h'] = h
            g.update_all(fn.copy_u('h', 'm'), fn.mean('m', 'h_neigh'))
            return self.W(torch.cat([g.ndata['h'], g.ndata['h_neigh']], 1))
```

如果用户已有一个用于整个图的自定义消息传递模块，并且想将其用于块，则只需要按照如下的方法重写forward函数。 注意，以下代码在注释里保留了整图实现的语句，用户可以将用于块的语句和原先用于整图的语句进行比较。

```python
class CustomGraphConv(nn.Module):
    def __init__(self, in_feats, out_feats):
        super().__init__()
        self.W = nn.Linear(in_feats * 2, out_feats)

    # h现在是输入和输出节点的特征张量对，而不是一个单独的特征张量

    # def forward(self, g, h):
    def forward(self, block, h):
        # with g.local_scope():
        with block.local_scope():
            # g.ndata['h'] = h
            h_src = h
            h_dst = h[:block.number_of_dst_nodes()]
            block.srcdata['h'] = h_src
            block.dstdata['h'] = h_dst

            # g.update_all(fn.copy_u('h', 'm'), fn.mean('m', 'h_neigh'))
            block.update_all(fn.copy_u('h', 'm'), fn.mean('m', 'h_neigh'))

            # return self.W(torch.cat([g.ndata['h'], g.ndata['h_neigh']], 1))
            return self.W(torch.cat(
                [block.dstdata['h'], block.dstdata['h_neigh']], 1))
```

通常，需要对用于整图的GNN模块进行如下调整以将其用于块作为输入的情况：

- 切片取输入特征的前几行，得到输出节点的特征。切片行数可以通过 `block.number_of_dst_nodes` 获得。
- 如果原图只包含一种节点类型，对输入节点特征，将 `g.ndata` 替换为 `block.srcdata`；对于输出节点特征，将 `g.ndata` 替换为 `block.dstdata`。
- 如果原图包含多种节点类型，对于输入节点特征，将 `g.nodes` 替换为 `block.srcnodes`；对于输出节点特征，将 `g.nodes` 替换为 `block.dstnodes`。
- 对于输入节点数量，将 `g.num_nodes` 替换为 `block.number_of_src_nodes`；对于输出节点数量，将 `g.num_nodes` 替换为 `block.number_of_dst_nodes`。

异构图上的模型定制

为异构图修改GNN模块的方法是类似的。例如，以下面用于全图的GNN模块为例：

```
class CustomHeteroGraphConv(nn.Module):
    def __init__(self, g, in_feats, out_feats):
        super().__init__()
        self.Ws = nn.ModuleDict()
        for etype in g.canonical_etypes:
            utype, _, vtype = etype
            self.Ws[etype] = nn.Linear(in_feats[utype], out_feats[vtype])
        for ntype in g.ntypes:
            self.Vs[ntype] = nn.Linear(in_feats[ntype], out_feats[ntype])

    def forward(self, g, h):
        with g.local_scope():
            for ntype in g.ntypes:
                g.nodes[ntype].data['h_dst'] = self.Vs[ntype](h[ntype])
                g.nodes[ntype].data['h_src'] = h[ntype]
            for etype in g.canonical_etypes:
                utype, _, vtype = etype
                g.update_all(
                    fn.copy_u('h_src', 'm'), fn.mean('m', 'h_neigh'),
                    etype=etype)
                g.nodes[vtype].data['h_dst'] = g.nodes[vtype].data['h_dst'] + \
                    self.Ws[etype](g.nodes[vtype].data['h_neigh'])
            return {ntype: g.nodes[ntype].data['h_dst'] for ntype in g.ntypes}
```

对于 `CustomHeteroGraphConv`，原则是将 `g.nodes` 替换为 `g.srcnodes` 或 `g.dstnodes` (根据需要输入还是输出节点的特征来选择)。

```
class CustomHeteroGraphConv(nn.Module):
    def __init__(self, g, in_feats, out_feats):
        super().__init__()
        self.Ws = nn.ModuleDict()
        for etype in g.canonical_etypes:
            utype, _, vtype = etype
            self.Ws[etype] = nn.Linear(in_feats[utype], out_feats[vtype])
        for ntype in g.ntypes:
            self.Vs[ntype] = nn.Linear(in_feats[ntype], out_feats[ntype])

    def forward(self, g, h):
        with g.local_scope():
            for ntype in g.ntypes:
                h_src, h_dst = h[ntype]
                g.dstnodes[ntype].data['h_dst'] = self.Vs[ntype](h[ntype])
                g.srcnodes[ntype].data['h_src'] = h[ntype]
            for etype in g.canonical_etypes:
                utype, _, vtype = etype
                g.update_all(
                    fn.copy_u('h_src', 'm'), fn.mean('m', 'h_neigh'),
                    etype=etype)
                g.dstnodes[vtype].data['h_dst'] = \
                    g.dstnodes[vtype].data['h_dst'] + \
                    self.Ws[etype](g.dstnodes[vtype].data['h_neigh'])
            return {ntype: g.dstnodes[ntype].data['h_dst']
                    for ntype in g.ntypes}
```

实现能够处理同构图、二分图和块的模块

DGL中所有的消息传递模块(参见 apinn)都能够处理同构图、 单向二分图(包含两种节点类型和一种边类型)和包含一种边类型的块。 本质上，内置的DGL神经网络模块的输入图及特征必须满足下列情况之一：

- 如果输入特征是一个张量对，则输入图必须是一个单向二分图
- 如果输入特征是一个单独的张量且输入图是一个块，则DGL会自动将输入节点特征前一部分设为输出节点的特征。
- 如果输入特征是一个单独的张量且输入图不是块，则输入图必须是同构图。

例如，下面的代码是 `dgl.nn.pytorch.SAGEConv` 的简化版(DGL同样支持它在MXNet和TensorFlow后端里的实现)。 代码里移除了归一化，且只考虑平均聚合函数的情况。

```
import dgl.function as fn
class SAGEConv(nn.Module):
    def __init__(self, in_feats, out_feats):
        super().__init__()
        self.W = nn.Linear(in_feats * 2, out_feats)

    def forward(self, g, h):
        if isinstance(h, tuple):
            h_src, h_dst = h
        elif g.is_block:
            h_src = h
            h_dst = h[:g.number_of_dst_nodes()]
        else:
            h_src = h_dst = h

        g.srcdata['h'] = h_src
        g.dstdata['h'] = h_dst
        g.update_all(fn.copy_u('h', 'm'), fn.sum('m', 'h_neigh'))
        return F.relu(
            self.W(torch.cat([g.dstdata['h'], g.dstdata['h_neigh']], 1)))
```

第3章：构建图神经网络（GNN）模块 提供了对 `dgl.nn.pytorch.SAGEConv` 代码的详细解读，其适用于单向二分图、同构图和块。