3.2 编写DGL NN模块的forward函数

(English Version)

在NN模块中， `forward()` 函数执行了实际的消息传递和计算。与通常以张量为参数的PyTorch NN模块相比， DGL NN模块额外增加了1个参数 `dgl.DGLGraph` 。 `forward()` 函数的内容一般可以分为3项操作：

- 检测输入图对象是否符合规范。
- 消息传递和聚合。
- 聚合后，更新特征作为输出。

下文展示了SAGEConv示例中的 `forward()` 函数。

输入图对象的规范检测

```
def forward(self, graph, feat):
    with graph.local_scope():
        # 指定图类型，然后根据图类型扩展输入特征
        feat_src, feat_dst = expand_as_pair(feat, graph)
```

`forward()` 函数需要处理输入的许多极端情况，这些情况可能导致计算和消息传递中的值无效。 比如在 `GraphConv` 等conv模块中，DGL会检查输入图中是否有入度为0的节点。 当1个节点入度为0时， `mailbox` 将为空，并且聚合函数的输出值全为0， 这可能会导致模型性能不佳。但是，在 `SAGEConv` 模块中，被聚合的特征将会与节点的初始特征拼接起来， `forward()` 函数的输出不会全为0。在这种情况下，无需进行此类检验。

DGL NN模块可在不同类型的图输入中重复使用，包括：同构图、异构图（1.5 异构图）和子图块（第6章：在大图上的随机（批次）训练）。

SAGEConv的数学公式如下：

$$h_{\mathcal{N}(dst)}^{(l+1)} = \text{aggregate}\left(\{h_{src}^{l}, \forall src \in \mathcal{N}(dst)\}\right)$$

$$h_{dst}^{(l+1)} = \sigma\left(W \cdot \text{concat}(h_{dst}^{l}, h_{\mathcal{N}(dst)}^{l+1}) + b\right)$$

$$h_{dst}^{(l+1)} = \text{norm}(h_{dst}^{l+1})$$

源节点特征 `feat_src` 和目标节点特征 `feat_dst` 需要根据图类型被指定。 用于指定图类型并将 `feat` 扩展为 `feat_src` 和 `feat_dst` 的函数是 `expand_as_pair()` 。 该函数的细节如下所示。

```
def expand_as_pair(input_, g=None):
    if isinstance(input_, tuple):
        # 二分图的情况
        return input_
    elif g is not None and g.is_block:
        # 子图块的情况
        if isinstance(input_, Mapping):
            input_dst = {
                k: F.narrow_row(v, 0, g.number_of_dst_nodes(k))
                for k, v in input_.items()}
        else:
            input_dst = F.narrow_row(input_, 0, g.number_of_dst_nodes())
        return input_, input_dst
    else:
        # 同构图的情况
        return input_, input_
```

对于同构图上的全图训练，源节点和目标节点相同，它们都是图中的所有节点。

在异构图的情况下，图可以分为几个二分图，每种关系对应一个。关系表示为 `(src_type, edge_type, dst_dtype)` 。 当输入特征 `feat` 是1个元组时，图将会被视为二分图。元组中的第1个元素为源节点特征，第2个元素为目标节点特征。

在小批次训练中，计算应用于给定的一堆目标节点所采样的子图。子图在DGL中称为区块(`block`)。 在区块创建的阶段， `dst nodes` 位于节点列表的最前面。通过索引 `[0:g.number_of_dst_nodes()]` 可以找到 `feat_dst` 。

确定 `feat_src` 和 `feat_dst` 之后，以上3种图类型的计算方法是相同的。

消息传递和聚合

```
import dgl.function as fn
import torch.nn.functional as F
from dgl.utils import check_eq_shape

if self._aggre_type == 'mean':
    graph.srcdata['h'] = feat_src
    graph.update_all(fn.copy_u('h', 'm'), fn.mean('m', 'neigh'))
    h_neigh = graph.dstdata['neigh']
elif self._aggre_type == 'gcn':
    check_eq_shape(feat)
    graph.srcdata['h'] = feat_src
    graph.dstdata['h'] = feat_dst
    graph.update_all(fn.copy_u('h', 'm'), fn.sum('m', 'neigh'))
    # 除以入度
    degs = graph.in_degrees().to(feat_dst)
    h_neigh = (graph.dstdata['neigh'] + graph.dstdata['h']) / (degs.unsqueeze(-1) + 1)
elif self._aggre_type == 'pool':
    graph.srcdata['h'] = F.relu(self.fc_pool(feat_src))
    graph.update_all(fn.copy_u('h', 'm'), fn.max('m', 'neigh'))
    h_neigh = graph.dstdata['neigh']
else:
    raise KeyError('Aggregator type {} not recognized.'.format(self._aggre_type))

# GraphSAGE中gcn聚合不需要fc_self
if self._aggre_type == 'gcn':
    rst = self.fc_neigh(h_neigh)
else:
    rst = self.fc_self(h_self) + self.fc_neigh(h_neigh)
```

上面的代码执行了消息传递和聚合的计算。这部分代码会因模块而异。请注意，代码中的所有消息传递均使用 `update_all()` API和 DGL内置的消息/聚合函数来实现，以充分利用 2.2 编写高效的消息传递代码 里所介绍的性能优化。

聚合后，更新特征作为输出

```
# 激活函数
if self.activation is not None:
    rst = self.activation(rst)
# 归一化
if self.norm is not None:
    rst = self.norm(rst)
return rst
```

`forward()` 函数的最后一部分是在完成消息聚合后更新节点的特征。 常见的更新操作是根据构造函数中设置的选项来应用激活函数和进行归一化。