6.5 Implementing Custom GNN Module for Mini-batch Training

(中文版)

Note

This tutorial has similar content to this section for the homogeneous graph case.

If you were familiar with how to write a custom GNN module for updating the entire graph for homogeneous or heterogeneous graphs (see Chapter 3: Building GNN Modules), the code for computing on MFGs is similar, with the exception that the nodes are divided into input nodes and output nodes.

For example, consider the following custom graph convolution module code. Note that it is not necessarily among the most efficient implementations - they only serve for an example of how a custom GNN module could look like.

```
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))
```

If you have a custom message passing NN module for the full graph, and you would like to make it work for MFGs, you only need to rewrite the forward function as follows. Note that the corresponding statements from the full-graph implementation are commented; you can compare the original statements with the new statements.

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

    # h is now a pair of feature tensors for input and output nodes, instead of
    # a single feature tensor.
    # 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))
```

In general, you need to do the following to make your NN module work for MFGs.

- Obtain the features for output nodes from the input features by slicing the first few rows. The number of rows can be obtained by `block.number_of_dst_nodes`.
- Replace `g.ndata` with either `block.srcdata` for features on input nodes or `block.dstdata` for features on output nodes, if the original graph has only one node type.
- Replace `g.nodes` with either `block.srcnodes` for features on input nodes or `block.dstnodes` for features on output nodes, if the original graph has multiple node types.
- Replace `g.num_nodes` with either `block.number_of_src_nodes` or `block.number_of_dst_nodes` for the number of input nodes or output nodes respectively.

Heterogeneous graphs

For heterogeneous graph the way of writing custom GNN modules is similar. For instance, consider the following module that work on full graph.

```
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}
```

For `CustomHeteroGraphConv`, the principle is to replace `g.nodes` with `g.srcnodes` or `g.dstnodes` depend on whether the features serve for input or output.

```
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}
```

Writing modules that work on homogeneous graphs,

bipartite graphs, and MFGs

All message passing modules in DGL work on homogeneous graphs, unidirectional bipartite graphs (that have two node types and one edge type), and a MFG with one edge type. Essentially, the input graph and feature of a builtin DGL neural network module must satisfy either of the following cases.

- If the input feature is a pair of tensors, then the input graph must be unidirectional bipartite.
- If the input feature is a single tensor and the input graph is a MFG, DGL will automatically set the feature on the output nodes as the first few rows of the input node features.
- If the input feature must be a single tensor and the input graph is not a MFG, then the input graph must be homogeneous.

For example, the following is simplified from the PyTorch implementation of `dgl.nn.pytorch.SAGEConv` (also available in MXNet and Tensorflow) (removing normalization and dealing with only mean aggregation etc.).

```python
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)))
```

Chapter 3: Building GNN Modules also provides a walkthrough on `dgl.nn.pytorch.SAGEConv`, which works on unidirectional bipartite graphs, homogeneous graphs, and MFGs.