3.2 DGL NN Module Forward Function

(中文版)

In NN module, `forward()` function does the actual message passing and computation. Compared with PyTorch's NN module which usually takes tensors as the parameters, DGL NN module takes an additional parameter `dgl.DGLGraph`. The workload for `forward()` function can be split into three parts:

- Graph checking and graph type specification.
- Message passing.
- Feature update.

The rest of the section takes a deep dive into the `forward()` function in SAGEConv example.

Graph checking and graph type specification

```
def forward(self, graph, feat):
    with graph.local_scope():
        # Specify graph type then expand input feature according to graph type
        feat_src, feat_dst = expand_as_pair(feat, graph)
```

`forward()` needs to handle many corner cases on the input that can lead to invalid values in computing and message passing. One typical check in conv modules like `GraphConv` is to verify that the input graph has no 0-in-degree nodes. When a node has 0 in-degree, the `mailbox` will be empty and the reduce function will produce all-zero values. This may cause silent regression in model performance. However, in `SAGEConv` module, the aggregated representation will be concatenated with the original node feature, the output of `forward()` will not be all-zero. No such check is needed in this case.

DGL NN module should be reusable across different types of graph input including: homogeneous graph, heterogeneous graph (1.5 Heterogeneous Graphs), subgraph block (Chapter 6: Stochastic Training on Large Graphs).

The math formulas for SAGEConv are:

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

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

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

One needs to specify the source node feature `feat_src` and destination node feature `feat_dst` according to the graph type. `expand_as_pair()` is a function that specifies the graph type and expand `feat` into `feat_src` and `feat_dst`. The detail of this function is shown below.

```
def expand_as_pair(input_, g=None):
    if isinstance(input_, tuple):
        # Bipartite graph case
        return input_
    elif g is not None and g.is_block:
        # Subgraph block case
        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:
        # Homogeneous graph case
        return input_, input_
```

For homogeneous whole graph training, source nodes and destination nodes are the same. They are all the nodes in the graph.

For heterogeneous case, the graph can be split into several bipartite graphs, one for each relation. The relations are represented as `(src_type, edge_type, dst_dtype)`. When it identifies that the input feature `feat` is a tuple, it will treat the graph as bipartite. The first element in the tuple will be the source node feature and the second element will be the destination node feature.

In mini-batch training, the computing is applied on a subgraph sampled based on a bunch of destination nodes. The subgraph is called as `block` in DGL. In the block creation phase, `dst nodes` are in the front of the node list. One can find the `feat_dst` by the index `[0:g.number_of_dst_nodes()]`.

After determining `feat_src` and `feat_dst`, the computing for the above three graph types are the same.

Message passing and reducing

```
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'))
    # divide in_degrees
    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 GCN does not require 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)
```

The code actually does message passing and reducing computing. This part of code varies module by module. Note that all the message passing in the above code are implemented using `update_all()` API and `built-in` message/reduce functions to fully utilize DGL's performance optimization as described in 2.2 Writing Efficient Message Passing Code.

Update feature after reducing for output

```
# activation
if self.activation is not None:
    rst = self.activation(rst)
# normalization
if self.norm is not None:
    rst = self.norm(rst)
return rst
```

The last part of `forward()` function is to update the feature after the `reduce function`. Common update operations are applying activation function and normalization according to the option set in the object construction phase.