3.3 Heterogeneous GraphConv Module

(中文版)

`HeteroGraphConv` is a module-level encapsulation to run DGL NN module on heterogeneous graphs. The implementation logic is the same as message passing level API `multi_update_all()`, including:

- DGL NN module within each relation r.
- Reduction that merges the results on the same node type from multiple relations.

This can be formulated as:

$$h_{dst}^{(l+1)} = \underset{r\in\mathcal{R},r_{dst}=dst}{AGG}(f_r(g_r, h_{r_{src}}^l, h_{r_{dst}}^l))$$

where f_r is the NN module for each relation r, AGG is the aggregation function.

HeteroGraphConv implementation logic:

```
import torch.nn as nn

class HeteroGraphConv(nn.Module):
    def __init__(self, mods, aggregate='sum'):
        super(HeteroGraphConv, self).__init__()
        self.mods = nn.ModuleDict(mods)
        if isinstance(aggregate, str):
            # An internal function to get common aggregation functions
            self.agg_fn = get_aggregate_fn(aggregate)
        else:
            self.agg_fn = aggregate
```

The heterograph convolution takes a dictionary `mods` that maps each relation to an nn module and sets the function that aggregates results on the same node type from multiple relations.

```
def forward(self, g, inputs, mod_args=None, mod_kwargs=None):
    if mod_args is None:
        mod_args = {}
    if mod_kwargs is None:
        mod_kwargs = {}
    outputs = {nty : [] for nty in g.dsttypes}
```

Besides input graph and input tensors, the `forward()` function takes two additional dictionary parameters `mod_args` and `mod_kwargs`. These two dictionaries have the same keys as `self.mods`. They are used as customized parameters when calling their corresponding NN modules in `self.mods` for different types of relations.

An output dictionary is created to hold output tensor for each destination type `nty`. Note that the value for each `nty` is a list, indicating a single node type may get multiple outputs if more than one relations have `nty` as the destination type. `HeteroGraphConv` will perform a further aggregation on the lists.

```
if g.is_block:
    src_inputs = inputs
    dst_inputs = {k: v[:g.number_of_dst_nodes(k)] for k, v in inputs.items()}
else:
    src_inputs = dst_inputs = inputs

for stype, etype, dtype in g.canonical_etypes:
    rel_graph = g[stype, etype, dtype]
    if rel_graph.num_edges() == 0:
        continue
    if stype not in src_inputs or dtype not in dst_inputs:
        continue
    dstdata = self.mods[etype](
        rel_graph,
        (src_inputs[stype], dst_inputs[dtype]),
        *mod_args.get(etype, ()),
        **mod_kwargs.get(etype, {}))
    outputs[dtype].append(dstdata)
```

The input `g` can be a heterogeneous graph or a subgraph block from a heterogeneous graph. As in ordinary NN module, the `forward()` function need to handle different input graph types separately.

Each relation is represented as a `canonical_etype`, which is `(stype, etype, dtype)`. Using `canonical_etype` as the key, one can extract out a bipartite graph `rel_graph`. For bipartite graph, the input feature will be organized as a tuple `(src_inputs[stype], dst_inputs[dtype])`. The NN module for each relation is called and the output is saved. To avoid unnecessary call, relations with no edges or no nodes with the src type will be skipped.

```
rsts = {}
for nty, alist in outputs.items():
    if len(alist) != 0:
        rsts[nty] = self.agg_fn(alist, nty)
```

Finally, the results on the same destination node type from multiple relations are aggregated using `self.agg_fn` function. Examples can be found in the API Doc for `HeteroGraphConv`.