2.1 Built-in Functions and Message Passing APIs

(中文版)

In DGL, **message function** takes a single argument `edges`, which is an `EdgeBatch` instance. During message passing, DGL generates it internally to represent a batch of edges. It has three members `src`, `dst` and `data` to access features of source nodes, destination nodes, and edges, respectively.

reduce function takes a single argument `nodes`, which is a `NodeBatch` instance. During message passing, DGL generates it internally to represent a batch of nodes. It has member `mailbox` to access the messages received for the nodes in the batch. Some of the most common reduce operations include `sum`, `max`, `min`, etc.

update function takes a single argument `nodes` as described above. This function operates on the aggregation result from `reduce function`, typically combining it with a node's original feature at the the last step and saving the result as a node feature.

DGL has implemented commonly used message functions and reduce functions as **built-in** in the namespace `dgl.function`. In general, DGL suggests using built-in functions **whenever possible** since they are heavily optimized and automatically handle dimension broadcasting.

If your message passing functions cannot be implemented with built-ins, you can implement user-defined message/reduce function (aka. **UDF**).

Built-in message functions can be unary or binary. DGL supports `copy` for unary. For binary funcs, DGL supports `add`, `sub`, `mul`, `div`, `dot`. The naming convention for message built-in funcs is that `u` represents `src` nodes, `v` represents `dst` nodes, and `e` represents `edges`. The parameters for those functions are strings indicating the input and output field names for the corresponding nodes and edges. The list of supported built-in functions can be found in DGL Built-in Function. For example, to add the `hu` feature from src nodes and `hv` feature from dst nodes then save the result on the edge at `he` field, one can use built-in function `dgl.function.u_add_v('hu', 'hv', 'he')`. This is equivalent to the Message UDF:

```
def message_func(edges):
    return {'he': edges.src['hu'] + edges.dst['hv']}
```

Built-in reduce functions support operations `sum`, `max`, `min`, and `mean`. Reduce functions usually have two parameters, one for field name in `mailbox`, one for field name in node features, both are strings. For example, `dgl.function.sum('m', 'h')` is equivalent to the Reduce UDF that sums up the message `m`:

```
import torch
def reduce_func(nodes):
     return {'h': torch.sum(nodes.mailbox['m'], dim=1)}
```

For advanced usage of UDF, see User-defined Functions.

It is also possible to invoke only edge-wise computation by `apply_edges()` without invoking message passing. `apply_edges()` takes a message function for parameter and by default updates the features of all edges. For example:

```
import dgl.function as fn
graph.apply_edges(fn.u_add_v('el', 'er', 'e'))
```

For message passing, `update_all()` is a high-level API that merges message generation, message aggregation and node update in a single call, which leaves room for optimization as a whole.

The parameters for `update_all()` are a message function, a reduce function and an update function. One can call update function outside of `update_all` and not specify it in invoking `update_all()`. DGL recommends this approach since the update function can usually be written as pure tensor operations to make the code concise. For example:

```
def update_all_example(graph):
    # store the result in graph.ndata['ft']
    graph.update_all(fn.u_mul_e('ft', 'a', 'm'),
                     fn.sum('m', 'ft'))
    # Call update function outside of update_all
    final_ft = graph.ndata['ft'] * 2
    return final_ft
```

This call will generate the messages `m` by multiply src node features `ft` and edge features `a`, sum up the messages `m` to update node features `ft`, and finally multiply `ft` by 2 to get the result `final_ft`. After the call, DGL will clean the intermediate messages `m`. The math formula for the above function is:

$$final\_ft_i = 2 * \sum_{j \in \mathcal{N}(i)} (ft_j * a_{ji})$$

DGL's built-in functions support floating point data types, i.e. the feature must be `half` (`float16`) / `float` / `double` tensors. `float16` data type support is disabled by default as it has a minimum GPU compute capacity requirement of `sm_53` (Pascal, Volta, Turing and Ampere architectures).

User can enable float16 for mixed precision training by compiling DGL from source (see Mixed Precision Training tutorial for details).