6.4 Implementing Custom Graph Samplers

Implementing custom samplers involves subclassing the `dgl.dataloading.Sampler` base class and implementing its abstract `sample` method. The `sample` method should take in two arguments:

```
def sample(self, g, indices):
    pass
```

The first argument `g` is the original graph to sample from while the second argument `indices` is the indices of the current mini-batch – it generally could be anything depending on what indices are given to the accompanied `DataLoader` but are typically seed node or seed edge IDs. The function returns the mini-batch of samples for the current iteration.

Note

The design here is similar to PyTorch's `torch.utils.data.DataLoader`, which is an iterator of dataset. Users can customize how to batch samples using its `collate_fn` argument. Here in DGL, `dgl.dataloading.DataLoader` is an iterator of `indices` (e.g., training node IDs) while `Sampler` converts a batch of indices into a batch of graph- or tensor-type samples.

The code below implements a classical neighbor sampler:

```
class NeighborSampler(dgl.dataloading.Sampler):
    def __init__(self, fanouts : list[int]):
        super().__init__()
        self.fanouts = fanouts

    def sample(self, g, seed_nodes):
        output_nodes = seed_nodes
        subgs = []
        for fanout in reversed(self.fanouts):
            # Sample a fixed number of neighbors of the current seed nodes.
            sg = g.sample_neighbors(seed_nodes, fanout)
            # Convert this subgraph to a message flow graph.
            sg = dgl.to_block(sg, seed_nodes)
            seed_nodes = sg.srcdata[NID]
            subgs.insert(0, sg)
            input_nodes = seed_nodes
        return input_nodes, output_nodes, subgs
```

To use this sampler with `DataLoader`:

```
graph = ...  # the graph to be sampled from
train_nids = ...  # an 1-D tensor of training node IDs
sampler = NeighborSampler([10, 15])  # create a sampler
dataloader = dgl.dataloading.DataLoader(
    graph,
    train_nids,
    sampler,
    batch_size=32,    # batch_size decides how many IDs are passed to sampler at once
    ...               # other arguments
)
for i, mini_batch in enumerate(dataloader):
    # unpack the mini batch
    input_nodes, output_nodes, subgs = mini_batch
    train(input_nodes, output_nodes, subgs)
```

Sampler for Heterogeneous Graphs

To write a sampler for heterogeneous graphs, one needs to be aware that the argument `g` will be a heterogeneous graph while `indices` could be a dictionary of ID tensors. Most of DGL's graph sampling operators (e.g., the `sample_neighbors` and `to_block` functions in the above example) can work on heterogeneous graph natively, so many samplers are automatically ready for heterogeneous graph. For example, the above `NeighborSampler` can be used on heterogeneous graphs:

```
hg = dgl.heterograph({
    ('user', 'like', 'movie') : ...,
    ('user', 'follow', 'user') : ...,
    ('movie', 'liked-by', 'user') : ...,
})
train_nids = {'user' : ..., 'movie' : ...}  # training IDs of 'user' and 'movie' nodes
sampler = NeighborSampler([10, 15])  # create a sampler
dataloader = dgl.dataloading.DataLoader(
    hg,
    train_nids,
    sampler,
    batch_size=32,    # batch_size decides how many IDs are passed to sampler at once
    ...               # other arguments
)
for i, mini_batch in enumerate(dataloader):
    # unpack the mini batch
    # input_nodes and output_nodes are dictionary while subgs are a list of
    # heterogeneous graphs
    input_nodes, output_nodes, subgs = mini_batch
    train(input_nodes, output_nodes, subgs)
```

Exclude Edges During Sampling

The examples above all belong to *node-wise sampler* because the `indices` argument to the `sample` method represents a batch of seed node IDs. Another common type of samplers is *edge-wise sampler* which, as name suggested, takes in a batch of seed edge IDs to construct mini-batch data. DGL provides a utility `dgl.dataloading.as_edge_prediction_sampler()` to turn a node-wise sampler to an edge-wise sampler. To prevent information leakge, it requires the node-wise sampler to have an additional third argument `exclude_eids`. The code below modifies the `NeighborSampler` we just defined to properly exclude edges from the sampled subgraph:

```
class NeighborSampler(Sampler):
    def __init__(self, fanouts):
        super().__init__()
        self.fanouts = fanouts

    # NOTE: There is an additional third argument. For homogeneous graphs,
    #   it is an 1-D tensor of integer IDs. For heterogeneous graphs, it
    #   is a dictionary of ID tensors. We usually set its default value to be None.
    def sample(self, g, seed_nodes, exclude_eids=None):
        output_nodes = seed_nodes
        subgs = []
        for fanout in reversed(self.fanouts):
            # Sample a fixed number of neighbors of the current seed nodes.
            sg = g.sample_neighbors(seed_nodes, fanout, exclude_edges=exclude_eids)
            # Convert this subgraph to a message flow graph.
            sg = dgl.to_block(sg, seed_nodes)
            seed_nodes = sg.srcdata[NID]
            subgs.insert(0, sg)
            input_nodes = seed_nodes
        return input_nodes, output_nodes, subgs
```

Further Readings

See 6.8 Feature Prefetching for how to write a custom graph sampler with feature prefetching.