6.2 Training GNN for Edge Classification with Neighborhood Sampling

(中文版)

Training for edge classification/regression is somewhat similar to that of node classification/regression with several notable differences.

Define a neighborhood sampler and data loader

You can use the same neighborhood samplers as node classification.

```
sampler = dgl.dataloading.MultiLayerFullNeighborSampler(2)
```

To use the neighborhood sampler provided by DGL for edge classification, one need to instead combine it with `as_edge_prediction_sampler()`, which iterates over a set of edges in minibatches, yielding the subgraph induced by the edge minibatch and *message flow graphs* (MFGs) to be consumed by the module below.

For example, the following code creates a PyTorch DataLoader that iterates over the training edge ID array `train_eids` in batches, putting the list of generated MFGs onto GPU.

```
sampler = dgl.dataloading.as_edge_prediction_sampler(sampler)
dataloader = dgl.dataloading.DataLoader(
    g, train_eid_dict, sampler,
    batch_size=1024,
    shuffle=True,
    drop_last=False,
    num_workers=4)
```

Note

See the Stochastic Training Tutorial for the concept of message flow graph.

For a complete list of supported builtin samplers, please refer to the neighborhood sampler API reference.

If you wish to develop your own neighborhood sampler or you want a more detailed explanation of the concept of MFGs, please refer to 6.4 Implementing Custom Graph Samplers.

Removing edges in the minibatch from the original graph for neighbor sampling

When training edge classification models, sometimes you wish to remove the edges appearing in the training data from the computation dependency as if they never existed. Otherwise, the model will "know" the fact that an edge exists between the two nodes, and potentially use it for advantage.

Therefore in edge classification you sometimes would like to exclude the edges sampled in the minibatch from the original graph for neighborhood sampling, as well as the reverse edges of the sampled edges on an undirected graph. You can specify `exclude='reverse_id'` in calling `as_edge_prediction_sampler()`, with the mapping of the edge IDs to their reverse edges IDs. Usually doing so will lead to much slower sampling process due to locating the reverse edges involving in the minibatch and removing them.

```
n_edges = g.num_edges()
sampler = dgl.dataloading.as_edge_prediction_sampler(
    sampler, exclude='reverse_id', reverse_eids=torch.cat([
        torch.arange(n_edges // 2, n_edges), torch.arange(0, n_edges // 2)]))
dataloader = dgl.dataloading.DataLoader(
    g, train_eid_dict, sampler,
    batch_size=1024,
    shuffle=True,
    drop_last=False,
    num_workers=4)
```

Adapt your model for minibatch training

The edge classification model usually consists of two parts:

- One part that obtains the representation of incident nodes.
- The other part that computes the edge score from the incident node representations.

The former part is exactly the same as that from node classification and we can simply reuse it. The input is still the list of MFGs generated from a data loader provided by DGL, as well as the input features.

```
class StochasticTwoLayerGCN(nn.Module):
    def __init__(self, in_features, hidden_features, out_features):
        super().__init__()
        self.conv1 = dglnn.GraphConv(in_features, hidden_features)
        self.conv2 = dglnn.GraphConv(hidden_features, out_features)

    def forward(self, blocks, x):
        x = F.relu(self.conv1(blocks[0], x))
        x = F.relu(self.conv2(blocks[1], x))
        return x
```

The input to the latter part is usually the output from the former part, as well as the subgraph of the original graph induced by the edges in the minibatch. The subgraph is yielded from the same data loader. One can call `dgl.DGLGraph.apply_edges()` to compute the scores on the edges with the edge subgraph.

The following code shows an example of predicting scores on the edges by concatenating the incident node features and projecting it with a dense layer.

```
class ScorePredictor(nn.Module):
    def __init__(self, num_classes, in_features):
        super().__init__()
        self.W = nn.Linear(2 * in_features, num_classes)

    def apply_edges(self, edges):
        data = torch.cat([edges.src['x'], edges.dst['x']], 1)
        return {'score': self.W(data)}

    def forward(self, edge_subgraph, x):
        with edge_subgraph.local_scope():
            edge_subgraph.ndata['x'] = x
            edge_subgraph.apply_edges(self.apply_edges)
            return edge_subgraph.edata['score']
```

The entire model will take the list of MFGs and the edge subgraph generated by the data loader, as well as the input node features as follows:

```
class Model(nn.Module):
    def __init__(self, in_features, hidden_features, out_features, num_classes):
        super().__init__()
        self.gcn = StochasticTwoLayerGCN(
            in_features, hidden_features, out_features)
        self.predictor = ScorePredictor(num_classes, out_features)

    def forward(self, edge_subgraph, blocks, x):
        x = self.gcn(blocks, x)
        return self.predictor(edge_subgraph, x)
```

DGL ensures that that the nodes in the edge subgraph are the same as the output nodes of the last MFG in the generated list of MFGs.

Training Loop

The training loop is very similar to node classification. You can iterate over the dataloader and get a subgraph induced by the edges in the minibatch, as well as the list of MFGs necessary for computing their incident node representations.

```
model = Model(in_features, hidden_features, out_features, num_classes)
model = model.cuda()
opt = torch.optim.Adam(model.parameters())

for input_nodes, edge_subgraph, blocks in dataloader:
    blocks = [b.to(torch.device('cuda')) for b in blocks]
    edge_subgraph = edge_subgraph.to(torch.device('cuda'))
    input_features = blocks[0].srcdata['features']
    edge_labels = edge_subgraph.edata['labels']
    edge_predictions = model(edge_subgraph, blocks, input_features)
    loss = compute_loss(edge_labels, edge_predictions)
    opt.zero_grad()
    loss.backward()
    opt.step()
```

For heterogeneous graphs

The models computing the node representations on heterogeneous graphs can also be used for computing incident node representations for edge classification/regression.

```
class StochasticTwoLayerRGCN(nn.Module):
    def __init__(self, in_feat, hidden_feat, out_feat, rel_names):
        super().__init__()
        self.conv1 = dglnn.HeteroGraphConv({
                rel : dglnn.GraphConv(in_feat, hidden_feat, norm='right')
                for rel in rel_names
            })
        self.conv2 = dglnn.HeteroGraphConv({
                rel : dglnn.GraphConv(hidden_feat, out_feat, norm='right')
                for rel in rel_names
            })

    def forward(self, blocks, x):
        x = self.conv1(blocks[0], x)
        x = self.conv2(blocks[1], x)
        return x
```

For score prediction, the only implementation difference between the homogeneous graph and the heterogeneous graph is that we are looping over the edge types for `apply_edges()`.

```
class ScorePredictor(nn.Module):
    def __init__(self, num_classes, in_features):
        super().__init__()
        self.W = nn.Linear(2 * in_features, num_classes)

    def apply_edges(self, edges):
        data = torch.cat([edges.src['x'], edges.dst['x']], 1)
        return {'score': self.W(data)}

    def forward(self, edge_subgraph, x):
        with edge_subgraph.local_scope():
            edge_subgraph.ndata['x'] = x
            for etype in edge_subgraph.canonical_etypes:
                edge_subgraph.apply_edges(self.apply_edges, etype=etype)
            return edge_subgraph.edata['score']

class Model(nn.Module):
    def __init__(self, in_features, hidden_features, out_features, num_classes,
                 etypes):
        super().__init__()
        self.rgcn = StochasticTwoLayerRGCN(
            in_features, hidden_features, out_features, etypes)
        self.pred = ScorePredictor(num_classes, out_features)

    def forward(self, edge_subgraph, blocks, x):
        x = self.rgcn(blocks, x)
        return self.pred(edge_subgraph, x)
```

Data loader definition is also very similar to that of node classification. The only difference is that you need `as_edge_prediction_sampler()`, and you will be supplying a dictionary of edge types and edge ID tensors instead of a dictionary of node types and node ID tensors.

```
sampler = dgl.dataloading.MultiLayerFullNeighborSampler(2)
sampler = dgl.dataloading.as_edge_prediction_sampler(sampler)
dataloader = dgl.dataloading.DataLoader(
    g, train_eid_dict, sampler,
    batch_size=1024,
    shuffle=True,
    drop_last=False,
    num_workers=4)
```

Things become a little different if you wish to exclude the reverse edges on heterogeneous graphs. On heterogeneous graphs, reverse edges usually have a different edge type from the edges themselves, in order to differentiate the “forward” and “backward” relationships (e.g. `follow` and `followed by` are reverse relations of each other, `purchase` and `purchased by` are reverse relations of each other, etc.).

If each edge in a type has a reverse edge with the same ID in another type, you can specify the mapping between edge types and their reverse types. The way to exclude the edges in the minibatch as well as their reverse edges then goes as follows.

```
sampler = dgl.dataloading.as_edge_prediction_sampler(
    sampler, exclude='reverse_types',
    reverse_etypes={'follow': 'followed by', 'followed by': 'follow',
                    'purchase': 'purchased by', 'purchased by': 'purchase'})
dataloader = dgl.dataloading.DataLoader(
    g, train_eid_dict, sampler,
    batch_size=1024,
    shuffle=True,
    drop_last=False,
    num_workers=4)
```

The training loop is again almost the same as that on homogeneous graph, except for the implementation of `compute_loss` that will take in two dictionaries of node types and predictions here.

```
model = Model(in_features, hidden_features, out_features, num_classes, etypes)
model = model.cuda()
opt = torch.optim.Adam(model.parameters())

for input_nodes, edge_subgraph, blocks in dataloader:
    blocks = [b.to(torch.device('cuda')) for b in blocks]
    edge_subgraph = edge_subgraph.to(torch.device('cuda'))
    input_features = blocks[0].srcdata['features']
    edge_labels = edge_subgraph.edata['labels']
    edge_predictions = model(edge_subgraph, blocks, input_features)
    loss = compute_loss(edge_labels, edge_predictions)
    opt.zero_grad()
    loss.backward()
    opt.step()
```

GCMC is an example of edge classification on a bipartite graph.