6.1 Training GNN for Node Classification with Neighborhood Sampling

(中文版)

To make your model been trained stochastically, you need to do the followings:

- Define a neighborhood sampler.
- Adapt your model for minibatch training.
- Modify your training loop.

The following sub-subsections address these steps one by one.

Define a neighborhood sampler and data loader

DGL provides several neighborhood sampler classes that generates the computation dependencies needed for each layer given the nodes we wish to compute on.

The simplest neighborhood sampler is `MultiLayerFullNeighborSampler` which makes the node gather messages from all of its neighbors.

To use a sampler provided by DGL, one also need to combine it with `DataLoader`, which iterates over a set of indices (nodes in this case) in minibatches.

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

```
import dgl
import dgl.nn as dglnn
import torch
import torch.nn as nn
import torch.nn.functional as F

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

Iterating over the DataLoader will yield a list of specially created graphs representing the computation dependencies on each layer. They are called *message flow graphs* (MFGs) in DGL.

```
input_nodes, output_nodes, blocks = next(iter(dataloader))
print(blocks)
```

The iterator generates three items at a time. `input_nodes` describe the nodes needed to compute the representation of `output_nodes`. `blocks` describe for each GNN layer which node representations are to be computed as output, which node representations are needed as input, and how does representation from the input nodes propagate to the output nodes.

> **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.

Adapt your model for minibatch training

If your message passing modules are all provided by DGL, the changes required to adapt your model to minibatch training is minimal. Take a multi-layer GCN as an example. If your model on full graph is implemented as follows:

```
class TwoLayerGCN(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, g, x):
        x = F.relu(self.conv1(g, x))
        x = F.relu(self.conv2(g, x))
        return x
```

Then all you need is to replace `g` with `blocks` generated above.

```
class StochasticTwoLayerGCN(nn.Module):
    def __init__(self, in_features, hidden_features, out_features):
        super().__init__()
        self.conv1 = dgl.nn.GraphConv(in_features, hidden_features)
        self.conv2 = dgl.nn.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 DGL `GraphConv` modules above accepts an element in `blocks` generated by the data loader as an argument.

The API reference of each NN module will tell you whether it supports accepting a MFG as an argument.

If you wish to use your own message passing module, please refer to 6.5 Implementing Custom GNN Module for Mini-batch Training.

Training Loop

The training loop simply consists of iterating over the dataset with the customized batching iterator. During each iteration that yields a list of MFGs, we:

1. Load the node features corresponding to the input nodes onto GPU. The node features can be stored in either memory or external storage. Note that we only need to load the input nodes' features, as opposed to load the features of all nodes as in full graph training.

   If the features are stored in `g.ndata`, then the features can be loaded by accessing the features in `blocks[0].srcdata`, the features of source nodes of the first MFG, which is identical to all the necessary nodes needed for computing the final representations.

2. Feed the list of MFGs and the input node features to the multilayer GNN and get the outputs.
3. Load the node labels corresponding to the output nodes onto GPU. Similarly, the node labels can be stored in either memory or external storage. Again, note that we only need to load the output nodes' labels, as opposed to load the labels of all nodes as in full graph training.

   If the features are stored in `g.ndata`, then the labels can be loaded by accessing the features in `blocks[-1].dstdata`, the features of destination nodes of the last MFG, which is identical to the nodes we wish to compute the final representation.

4. Compute the loss and backpropagate.

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

for input_nodes, output_nodes, blocks in dataloader:
    blocks = [b.to(torch.device('cuda')) for b in blocks]
    input_features = blocks[0].srcdata['features']
    output_labels = blocks[-1].dstdata['label']
    output_predictions = model(blocks, input_features)
    loss = compute_loss(output_labels, output_predictions)
    opt.zero_grad()
    loss.backward()
    opt.step()
```

DGL provides an end-to-end stochastic training example GraphSAGE implementation.

For heterogeneous graphs

Training a graph neural network for node classification on heterogeneous graph is similar.

For instance, we have previously seen how to train a 2-layer RGCN on full graph. The code for RGCN implementation on minibatch training looks very similar to that (with self-loops, non-linearity and basis decomposition removed for simplicity):

```
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
```

Some of the samplers provided by DGL also support heterogeneous graphs. For example, one can still use the provided `MultiLayerFullNeighborSampler` class and `DataLoader` class for stochastic training. For full-neighbor sampling, the only difference would be that you would specify a dictionary of node types and node IDs for the training set.

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

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

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

for input_nodes, output_nodes, blocks in dataloader:
    blocks = [b.to(torch.device('cuda')) for b in blocks]
    input_features = blocks[0].srcdata     # returns a dict
    output_labels = blocks[-1].dstdata     # returns a dict
    output_predictions = model(blocks, input_features)
    loss = compute_loss(output_labels, output_predictions)
    opt.zero_grad()
    loss.backward()
    opt.step()
```

DGL provides an end-to-end stochastic training example RGCN implementation.