6.3 Training GNN for Link Prediction with Neighborhood Sampling

(中文版)

Define a neighborhood sampler and data loader with negative sampling

You can still use the same neighborhood sampler as the one in node/edge classification.

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

`as_edge_prediction_sampler()` in DGL also supports generating negative samples for link prediction. To do so, you need to provide the negative sampling function. `Uniform` is a function that does uniform sampling. For each source node of an edge, it samples `k` negative destination nodes.

The following data loader will pick 5 negative destination nodes uniformly for each source node of an edge.

```
sampler = dgl.dataloading.as_edge_prediction_sampler(
    sampler, negative_sampler=dgl.dataloading.negative_sampler.Uniform(5))
dataloader = dgl.dataloading.DataLoader(
    g, train_seeds, sampler,
    batch_size=args.batch_size,
    shuffle=True,
    drop_last=False,
    pin_memory=True,
    num_workers=args.num_workers)
```

For the builtin negative samplers please see Negative Samplers for Link Prediction.

You can also give your own negative sampler function, as long as it takes in the original graph `g` and the minibatch edge ID array `eid`, and returns a pair of source ID arrays and destination ID arrays.

The following gives an example of custom negative sampler that samples negative destination nodes according to a probability distribution proportional to a power of degrees.

```
class NegativeSampler(object):
    def __init__(self, g, k):
        # caches the probability distribution
        self.weights = g.in_degrees().float() ** 0.75
        self.k = k

    def __call__(self, g, eids):
        src, _ = g.find_edges(eids)
        src = src.repeat_interleave(self.k)
        dst = self.weights.multinomial(len(src), replacement=True)
        return src, dst
sampler = dgl.dataloading.as_edge_prediction_sampler(
    sampler, negative_sampler=NegativeSampler(g, 5))
dataloader = dgl.dataloading.DataLoader(
    g, train_seeds, sampler,
    batch_size=args.batch_size,
    shuffle=True,
    drop_last=False,
    pin_memory=True,
    num_workers=args.num_workers)
```

Adapt your model for minibatch training

As explained in 5.3 Link Prediction, link prediction is trained via comparing the score of an edge (positive example) against a non-existent edge (negative example). To compute the scores of edges you can reuse the node representation computation model you have seen in edge classification/regression.

```
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
```

For score prediction, since you only need to predict a scalar score for each edge instead of a probability distribution, this example shows how to compute a score with a dot product of incident node representations.

```
class ScorePredictor(nn.Module):
    def forward(self, edge_subgraph, x):
        with edge_subgraph.local_scope():
            edge_subgraph.ndata['x'] = x
            edge_subgraph.apply_edges(dgl.function.u_dot_v('x', 'x', 'score'))
            return edge_subgraph.edata['score']
```

When a negative sampler is provided, DGL's data loader will generate three items per minibatch:

- A positive graph containing all the edges sampled in the minibatch.
- A negative graph containing all the non-existent edges generated by the negative sampler.
- A list of *message flow graphs* (MFGs) generated by the neighborhood sampler.

So one can define the link prediction model as follows that takes in the three items as well as the input features.

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

    def forward(self, positive_graph, negative_graph, blocks, x):
        x = self.gcn(blocks, x)
        pos_score = self.predictor(positive_graph, x)
        neg_score = self.predictor(negative_graph, x)
        return pos_score, neg_score
```

Training loop

The training loop simply involves iterating over the data loader and feeding in the graphs as well as the input features to the model defined above.

```
def compute_loss(pos_score, neg_score):
    # an example hinge loss
    n = pos_score.shape[0]
    return (neg_score.view(n, -1) - pos_score.view(n, -1) + 1).clamp(min=0).mean()

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

for input_nodes, positive_graph, negative_graph, blocks in dataloader:
    blocks = [b.to(torch.device('cuda')) for b in blocks]
    positive_graph = positive_graph.to(torch.device('cuda'))
    negative_graph = negative_graph.to(torch.device('cuda'))
    input_features = blocks[0].srcdata['features']
    pos_score, neg_score = model(positive_graph, negative_graph, blocks, input_features)
    loss = compute_loss(pos_score, neg_score)
    opt.zero_grad()
    loss.backward()
    opt.step()
```

DGL provides the unsupervised learning GraphSAGE that shows an example of link prediction on homogeneous graphs.

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 `dgl.DGLGraph.apply_edges()`.

```
class ScorePredictor(nn.Module):
    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(
                    dgl.function.u_dot_v('x', 'x', 'score'), 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()

    def forward(self, positive_graph, negative_graph, blocks, x):
        x = self.rgcn(blocks, x)
        pos_score = self.pred(positive_graph, x)
        neg_score = self.pred(negative_graph, x)
        return pos_score, neg_score
```

Data loader definition is also very similar to that of edge classification/regression. The only difference is that you need to give the negative 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, negative_sampler=dgl.dataloading.negative_sampler.Uniform(5))
dataloader = dgl.dataloading.DataLoader(
    g, train_eid_dict, sampler,
    batch_size=1024,
    shuffle=True,
    drop_last=False,
    num_workers=4)
```

If you want to give your own negative sampling function, the function should take in the original graph and the dictionary of edge types and edge ID tensors. It should return a dictionary of edge types and source-destination array pairs. An example is given as follows:

```
class NegativeSampler(object):
    def __init__(self, g, k):
        # caches the probability distribution
        self.weights = {
            etype: g.in_degrees(etype=etype).float() ** 0.75
            for etype in g.canonical_etypes}
        self.k = k

    def __call__(self, g, eids_dict):
        result_dict = {}
        for etype, eids in eids_dict.items():
            src, _ = g.find_edges(eids, etype=etype)
            src = src.repeat_interleave(self.k)
            dst = self.weights[etype].multinomial(len(src), replacement=True)
            result_dict[etype] = (src, dst)
        return result_dict
```

Then you can give the dataloader a dictionary of edge types and edge IDs as well as the negative sampler. For instance, the following iterates over all edges of the heterogeneous graph.

```
train_eid_dict = {
    etype: g.edges(etype=etype, form='eid')
    for etype in g.canonical_etypes}
sampler = dgl.dataloading.as_edge_prediction_sampler(
    sampler, negative_sampler=NegativeSampler(g, 5))
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, positive_graph, negative_graph, blocks in dataloader:
    blocks = [b.to(torch.device('cuda')) for b in blocks]
    positive_graph = positive_graph.to(torch.device('cuda'))
    negative_graph = negative_graph.to(torch.device('cuda'))
    input_features = blocks[0].srcdata['features']
    pos_score, neg_score = model(positive_graph, negative_graph, blocks, input_features)
    loss = compute_loss(pos_score, neg_score)
    opt.zero_grad()
    loss.backward()
    opt.step()
```