5.3 Link Prediction

(中文版)

In some other settings you may want to predict whether an edge exists between two given nodes or not. Such task is called a *link prediction* task.

Overview

A GNN-based link prediction model represents the likelihood of connectivity between two nodes u and v as a function of $\boldsymbol{h}_u^{(L)}$ and $\boldsymbol{h}_v^{(L)}$, their node representation computed from the multi-layer GNN.

$$y_{u,v} = \phi(\boldsymbol{h}_u^{(L)}, \boldsymbol{h}_v^{(L)})$$

In this section we refer to $y_{u,v}$ the *score* between node u and node v.

Training a link prediction model involves comparing the scores between nodes connected by an edge against the scores between an arbitrary pair of nodes. For example, given an edge connecting u and v, we encourage the score between node u and v to be higher than the score between node u and a sampled node v' from an arbitrary *noise* distribution $v' \sim P_n(v)$. Such methodology is called *negative sampling*.

There are lots of loss functions that can achieve the behavior above if minimized. A non-exhaustive list include:

- Cross-entropy loss: $\mathcal{L} = -\log \sigma(y_{u,v}) - \sum_{v_i \sim P_n(v), i=1,\ldots,k} \log[1 - \sigma(y_{u,v_i})]$
- BPR loss: $\mathcal{L} = \sum_{v_i \sim P_n(v), i=1,\ldots,k} -\log \sigma(y_{u,v} - y_{u,v_i})$
- Margin loss: $\mathcal{L} = \sum_{v_i \sim P_n(v), i=1,\ldots,k} \max(0, M - y_{u,v} + y_{u,v_i})$, where M is a constant hyperparameter.

You may find this idea familiar if you know what implicit feedback or noise-contrastive estimation is.

The neural network model to compute the score between u and v is identical to the edge regression model described above.

Here is an example of using dot product to compute the scores on edges.

```
class DotProductPredictor(nn.Module):
    def forward(self, graph, h):
        # h contains the node representations computed from the GNN defined
        # in the node classification section (Section 5.1).
        with graph.local_scope():
            graph.ndata['h'] = h
            graph.apply_edges(fn.u_dot_v('h', 'h', 'score'))
            return graph.edata['score']
```

Training loop

Because our score prediction model operates on graphs, we need to express the negative examples as another graph. The graph will contain all negative node pairs as edges.

The following shows an example of expressing negative examples as a graph. Each edge (u, v) gets k negative examples (u, v_i) where v_i is sampled from a uniform distribution.

```
def construct_negative_graph(graph, k):
    src, dst = graph.edges()

    neg_src = src.repeat_interleave(k)
    neg_dst = torch.randint(0, graph.num_nodes(), (len(src) * k,))
    return dgl.graph((neg_src, neg_dst), num_nodes=graph.num_nodes())
```

The model that predicts edge scores is the same as that of edge classification/regression.

```
class Model(nn.Module):
    def __init__(self, in_features, hidden_features, out_features):
        super().__init__()
        self.sage = SAGE(in_features, hidden_features, out_features)
        self.pred = DotProductPredictor()
    def forward(self, g, neg_g, x):
        h = self.sage(g, x)
        return self.pred(g, h), self.pred(neg_g, h)
```

The training loop then repeatedly constructs the negative graph and computes loss.

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

node_features = graph.ndata['feat']
n_features = node_features.shape[1]
k = 5
model = Model(n_features, 100, 100)
opt = torch.optim.Adam(model.parameters())
for epoch in range(10):
    negative_graph = construct_negative_graph(graph, k)
    pos_score, neg_score = model(graph, negative_graph, node_features)
    loss = compute_loss(pos_score, neg_score)
    opt.zero_grad()
    loss.backward()
    opt.step()
    print(loss.item())
```

After training, the node representation can be obtained via

```
node_embeddings = model.sage(graph, node_features)
```

There are multiple ways of using the node embeddings. Examples include training downstream classifiers, or doing nearest neighbor search or maximum inner product search for relevant entity recommendation.

Heterogeneous graphs

Link prediction on heterogeneous graphs is not very different from that on homogeneous graphs. The following assumes that we are predicting on one edge type, and it is easy to extend it to multiple edge types.

For example, you can reuse the `HeteroDotProductPredictor` above for computing the scores of the edges of an edge type for link prediction.

```
class HeteroDotProductPredictor(nn.Module):
    def forward(self, graph, h, etype):
        # h contains the node representations for each node type computed from
        # the GNN defined in the previous section (Section 5.1).
        with graph.local_scope():
            graph.ndata['h'] = h
            graph.apply_edges(fn.u_dot_v('h', 'h', 'score'), etype=etype)
            return graph.edges[etype].data['score']
```

To perform negative sampling, one can construct a negative graph for the edge type you are performing link prediction on as well.

```
def construct_negative_graph(graph, k, etype):
    utype, _, vtype = etype
    src, dst = graph.edges(etype=etype)
    neg_src = src.repeat_interleave(k)
    neg_dst = torch.randint(0, graph.num_nodes(vtype), (len(src) * k,))
    return dgl.heterograph(
        {etype: (neg_src, neg_dst)},
        num_nodes_dict={ntype: graph.num_nodes(ntype) for ntype in graph.ntypes})
```

The model is a bit different from that in edge classification on heterogeneous graphs since you need to specify edge type where you perform link prediction.

```
class Model(nn.Module):
    def __init__(self, in_features, hidden_features, out_features, rel_names):
        super().__init__()
        self.sage = RGCN(in_features, hidden_features, out_features, rel_names)
        self.pred = HeteroDotProductPredictor()
    def forward(self, g, neg_g, x, etype):
        h = self.sage(g, x)
        return self.pred(g, h, etype), self.pred(neg_g, h, etype)
```

The training loop is similar to that of homogeneous graphs.

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

k = 5
model = Model(10, 20, 5, hetero_graph.etypes)
user_feats = hetero_graph.nodes['user'].data['feature']
item_feats = hetero_graph.nodes['item'].data['feature']
node_features = {'user': user_feats, 'item': item_feats}
opt = torch.optim.Adam(model.parameters())
for epoch in range(10):
    negative_graph = construct_negative_graph(hetero_graph, k, ('user', 'click', 'item'))
    pos_score, neg_score = model(hetero_graph, negative_graph, node_features, ('user', 'click',
'item'))
    loss = compute_loss(pos_score, neg_score)
    opt.zero_grad()
    loss.backward()
    opt.step()
    print(loss.item())
```