5.2 Edge Classification/Regression

(中文版)

Sometimes you wish to predict the attributes on the edges of the graph. In that case, you would like to have an *edge classification/regression* model.

Here we generate a random graph for edge prediction as a demonstration.

```
src = np.random.randint(0, 100, 500)
dst = np.random.randint(0, 100, 500)
# make it symmetric
edge_pred_graph = dgl.graph((np.concatenate([src, dst]), np.concatenate([dst, src])))
# synthetic node and edge features, as well as edge labels
edge_pred_graph.ndata['feature'] = torch.randn(100, 10)
edge_pred_graph.edata['feature'] = torch.randn(1000, 10)
edge_pred_graph.edata['label'] = torch.randn(1000)
# synthetic train-validation-test splits
edge_pred_graph.edata['train_mask'] = torch.zeros(1000, dtype=torch.bool).bernoulli(0.6)
```

Overview

From the previous section you have learned how to do node classification with a multilayer GNN. The same technique can be applied for computing a hidden representation of any node. The prediction on edges can then be derived from the representation of their incident nodes.

The most common case of computing the prediction on an edge is to express it as a parameterized function of the representation of its incident nodes, and optionally the features on the edge itself.

Model Implementation Difference from Node Classification

Assuming that you compute the node representation with the model from the previous section, you only need to write another component that computes the edge prediction with the `apply_edges()` method.

For instance, if you would like to compute a score for each edge for edge regression, the following code computes the dot product of incident node representations on each edge.

```
import dgl.function as fn
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']
```

One can also write a prediction function that predicts a vector for each edge with an MLP. Such vector can be used in further downstream tasks, e.g. as logits of a categorical distribution.

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

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

    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(self.apply_edges)
            return graph.edata['score']
```

Training loop

Given the node representation computation model and an edge predictor model, we can easily write a full-graph training loop where we compute the prediction on all edges.

The following example takes `SAGE` in the previous section as the node representation computation model and `DotPredictor` as an edge predictor model.

```
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, x):
        h = self.sage(g, x)
        return self.pred(g, h)
```

In this example, we also assume that the training/validation/test edge sets are identified by boolean masks on edges. This example also does not include early stopping and model saving.

```
node_features = edge_pred_graph.ndata['feature']
edge_label = edge_pred_graph.edata['label']
train_mask = edge_pred_graph.edata['train_mask']
model = Model(10, 20, 5)
opt = torch.optim.Adam(model.parameters())
for epoch in range(10):
    pred = model(edge_pred_graph, node_features)
    loss = ((pred[train_mask] - edge_label[train_mask]) ** 2).mean()
    opt.zero_grad()
    loss.backward()
    opt.step()
    print(loss.item())
```

Heterogeneous graph

Edge classification on heterogeneous graphs is not very different from that on homogeneous graphs. If you wish to perform edge classification on one edge type, you only need to compute the node representation for all node types, and predict on that edge type with `apply_edges()` method.

For example, to make `DotProductPredictor` work on one edge type of a heterogeneous graph, you only need to specify the edge type in `apply_edges` method.

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

You can similarly write a `HeteroMLPPredictor`.

```python
class HeteroMLPPredictor(nn.Module):
    def __init__(self, in_features, out_classes):
        super().__init__()
        self.W = nn.Linear(in_features * 2, out_classes)

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

    def forward(self, graph, h, etype):
        # h contains the node representations for each edge type computed from
        # the GNN for heterogeneous graphs defined in the node classification
        # section (Section 5.1).
        with graph.local_scope():
            graph.ndata['h'] = h   # assigns 'h' of all node types in one shot
            graph.apply_edges(self.apply_edges, etype=etype)
            return graph.edges[etype].data['score']
```

The end-to-end model that predicts a score for each edge on a single edge type will look like this:

```python
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, x, etype):
        h = self.sage(g, x)
        return self.pred(g, h, etype)
```

Using the model simply involves feeding the model a dictionary of node types and features.

```
model = Model(10, 20, 5, hetero_graph.etypes)
user_feats = hetero_graph.nodes['user'].data['feature']
item_feats = hetero_graph.nodes['item'].data['feature']
label = hetero_graph.edges['click'].data['label']
train_mask = hetero_graph.edges['click'].data['train_mask']
node_features = {'user': user_feats, 'item': item_feats}
```

Then the training loop looks almost the same as that in homogeneous graph. For instance, if you wish to predict the edge labels on edge type `click`, then you can simply do

```
opt = torch.optim.Adam(model.parameters())
for epoch in range(10):
    pred = model(hetero_graph, node_features, 'click')
    loss = ((pred[train_mask] - label[train_mask]) ** 2).mean()
    opt.zero_grad()
    loss.backward()
    opt.step()
    print(loss.item())
```

Predicting Edge Type of an Existing Edge on a Heterogeneous Graph

Sometimes you may want to predict which type an existing edge belongs to.

For instance, given the heterogeneous graph example, your task is given an edge connecting a user and an item, to predict whether the user would `click` or `dislike` an item.

This is a simplified version of rating prediction, which is common in recommendation literature.

You can use a heterogeneous graph convolution network to obtain the node representations. For instance, you can still use the RGCN defined previously for this purpose.

To predict the type of an edge, you can simply repurpose the `HeteroDotProductPredictor` above so that it takes in another graph with only one edge type that “merges” all the edge types to be predicted, and emits the score of each type for every edge.

In the example here, you will need a graph that has two node types `user` and `item`, and one single edge type that “merges” all the edge types from `user` and `item`, i.e. `click` and `dislike`. This can be conveniently created using the following syntax:

```
dec_graph = hetero_graph['user', :, 'item']
```

which returns a heterogeneous graphs with node type `user` and `item`, as well as a single edge type combining all edge types in between, i.e. `click` and `dislike`.

Since the statement above also returns the original edge types as a feature named `dgl.ETYPE`, we can use that as labels.

```
edge_label = dec_graph.edata[dgl.ETYPE]
```

Given the graph above as input to the edge type predictor module, you can write your predictor module as follows.

```
class HeteroMLPPredictor(nn.Module):
    def __init__(self, in_dims, n_classes):
        super().__init__()
        self.W = nn.Linear(in_dims * 2, n_classes)

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

    def forward(self, graph, h):
        # h contains the node representations for each edge type computed from
        # the GNN for heterogeneous graphs defined in the node classification
        # section (Section 5.1).
        with graph.local_scope():
            graph.ndata['h'] = h   # assigns 'h' of all node types in one shot
            graph.apply_edges(self.apply_edges)
            return graph.edata['score']
```

The model that combines the node representation module and the edge type predictor module is the following:

```
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 = HeteroMLPPredictor(out_features, len(rel_names))
    def forward(self, g, x, dec_graph):
        h = self.sage(g, x)
        return self.pred(dec_graph, h)
```

The training loop then simply be the following:

```python
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):
    logits = model(hetero_graph, node_features, dec_graph)
    loss = F.cross_entropy(logits, edge_label)
    opt.zero_grad()
    loss.backward()
    opt.step()
    print(loss.item())
```

DGL provides Graph Convolutional Matrix Completion as an example of rating prediction, which is formulated by predicting the type of an existing edge on a heterogeneous graph. The node representation module in the model implementation file is called `GCMCLayer`. The edge type predictor module is called `BiDecoder`. Both of them are more complicated than the setting described here.