5.2 边分类/回归

(English Version)

有时用户希望预测图中边的属性值，这种情况下，用户需要构建一个边分类/回归的模型。

以下代码生成了一个随机图用于演示边分类/回归。

```
src = np.random.randint(0, 100, 500)
dst = np.random.randint(0, 100, 500)
# 同时建立反向边
edge_pred_graph = dgl.graph((np.concatenate([src, dst]), np.concatenate([dst, src])))
# 建立点和边特征，以及边的标签
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)
# 进行训练、验证和测试集划分
edge_pred_graph.edata['train_mask'] = torch.zeros(1000, dtype=torch.bool).bernoulli(0.6)
```

概述

上一节介绍了如何使用多层GNN进行节点分类。同样的方法也可以被用于计算任何节点的隐藏表示。 并从边的两个端点的表示，通过计算得出对边属性的预测。

对一条边计算预测值最常见的情况是将预测表示为一个函数，函数的输入为两个端点的表示， 输入还可以包括边自身的特征。

与节点分类在模型实现上的差别

如果用户使用上一节中的模型计算了节点的表示，那么用户只需要再编写一个用 `apply_edges()` 方法计算边预测的组件即可进行边分类/回归任务。

例如，对于边回归任务，如果用户想为每条边计算一个分数，可按下面的代码对每一条边计算它的两端节点隐藏表示的点积来作为分数。

```
import dgl.function as fn
class DotProductPredictor(nn.Module):
    def forward(self, graph, h):
        # h是从5.1节的GNN模型中计算出的节点表示
        with graph.local_scope():
            graph.ndata['h'] = h
            graph.apply_edges(fn.u_dot_v('h', 'h', 'score'))
            return graph.edata['score']
```

用户也可以使用MLP(多层感知机)对每条边生成一个向量表示(例如，作为一个未经过归一化的类别的分布)， 并在下游任务中使用。

```
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是从5.1节的GNN模型中计算出的节点表示
        with graph.local_scope():
            graph.ndata['h'] = h
            graph.apply_edges(self.apply_edges)
            return graph.edata['score']
```

模型的训练

给定计算节点和边上表示的模型后，用户可以轻松地编写在所有边上进行预测的全图训练代码。

以下代码用了 第2章：消息传递范式 中定义的 `SAGE` 作为节点表示计算模型以及前一小节中定义的 `DotPredictor` 作为边预测模型。

```
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)
```

在训练模型时可以使用布尔掩码区分训练、验证和测试数据集。该例子里省略了训练早停和模型保存部分的代码。

```
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())
```

异构图上的边预测模型的训练

例如想在某一特定类型的边上进行分类任务，用户只需要计算所有节点类型的节点表示，然后同样通过调用 `apply_edges()` 方法计算预测值即可。唯一的区别是在调用 `apply_edges` 时需要指定边的类型。

```
class HeteroDotProductPredictor(nn.Module):
    def forward(self, graph, h, etype):
        # h是从5.1节中对每种类型的边所计算的节点表示
        with graph.local_scope():
            graph.ndata['h'] = h    #一次性为所有节点类型的 'h'赋值
            graph.apply_edges(fn.u_dot_v('h', 'h', 'score'), etype=etype)
            return graph.edges[etype].data['score']
```

同样地，用户也可以编写一个 `HeteroMLPPredictor`。

```
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, etype):
        # h是从5.1节中对异构图的每种类型的边所计算的节点表示
        with graph.local_scope():
            graph.ndata['h'] = h    #一次性为所有节点类型的 'h'赋值
            graph.apply_edges(self.apply_edges, etype=etype)
            return graph.edges[etype].data['score']
```

在某种类型的边上为每一条边预测的端到端模型的定义如下所示：

```
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)
```

使用模型时只需要简单地向模型提供一个包含节点类型和数据特征的字典。

```
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}
```

训练部分和同构图的训练基本一致。例如，如果用户想预测边类型为 `click` 的边的标签，只需要按下例编写代码。

```
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())
```

在异构图中预测已有边的类型

预测图中已经存在的边属于哪个类型是一个非常常见的任务类型。例如，根据 本章的异构图样例数据， 用户的任务是给定一条连接 `user` 节点和 `item` 节点的边，预测它的类型是 `click` 还是 `dislike` 。 这个例子是评分预测的一个简化版本，在推荐场景中很常见。

边类型预测的第一步仍然是计算节点表示。可以通过类似 节点分类的RGCN模型 这一章中提到的图卷积网络获得。第二步是计算边上的预测值。 在这里可以复用上述提到的 `HeteroDotProductPredictor` 。 这里需要注意的是输入的图数据不能包含边的类型信息， 因此需要将所要预测的边类型(如 `click` 和 `dislike`)合并成一种边的图， 并为每条边计算出每种边类型的可能得分。下面的例子使用一个拥有 `user` 和 `item` 两种节点类型和一种边类型的图。该边类型是通过合并所有从 `user` 到 `item` 的边类型(如 `like` 和 `dislike`)得到。 用户可以很方便地用关系切片的方式创建这个图。

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

这个方法会返回一个异构图，它具有 `user` 和 `item` 两种节点类型，以及把它们之间的所有边的类型进行合并后的单一边类型。

由于上面这行代码将原来的边类型存成边特征 `dgl.ETYPE`，用户可以将它作为标签使用。

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

将上述图作为边类型预测模块的输入，用户可以按如下方式编写预测模块：

```
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是从5.1节中对异构图的每种类型的边所计算的节点表示
        with graph.local_scope():
            graph.ndata['h'] = h    #一次性为所有节点类型的 'h'赋值
            graph.apply_edges(self.apply_edges)
            return graph.edata['score']
```

结合了节点表示模块和边类型预测模块的模型如下所示：

```
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)
```

训练部分如下所示：

```
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())
```

读者可以进一步参考 Graph Convolutional Matrix Completion 这一示例来了解如何预测异构图中的边类型。 模型实现文件中 的节点表示模块称作 `GCMCLayer` 。边类型预测模块称作 `BiDecoder` 。 虽然这两个模块都比上述的示例代码要复杂，但其基本思想和本章描述的流程是一致的。