5.1 节点分类/回归

(English Version)

对于图神经网络来说，最常见和被广泛使用的任务之一就是节点分类。 图数据中的训练、验证和测试集中的每个节点都具有从一组预定义的类别中分配的一个类别，即正确的标注。 节点回归任务也类似，训练、验证和测试集中的每个节点都被标注了一个正确的数字。

概述

为了对节点进行分类，图神经网络执行了 第2章：消息传递范式 中介绍的消息传递机制，利用节点自身的特征和其邻节点及边的特征来计算节点的隐藏表示。 消息传递可以重复多轮，以利用更大范围的邻居信息。

编写神经网络模型

DGL提供了一些内置的图卷积模块，可以完成一轮消息传递计算。 本章中选择 `dgl.nn.pytorch.SAGEConv` 作为演示的样例代码(针对MXNet和PyTorch后端也有对应的模块)，它是GraphSAGE模型中使用的图卷积模块。

对于图上的深度学习模型，通常需要一个多层的图神经网络，并在这个网络中要进行多轮的信息传递。 可以通过堆叠图卷积模块来实现这种网络架构，具体如下所示。

```
# 构建一个2层的GNN模型
import dgl.nn as dglnn
import torch.nn as nn
import torch.nn.functional as F
class SAGE(nn.Module):
    def __init__(self, in_feats, hid_feats, out_feats):
        super().__init__()
        # 实例化SAGEConve，in_feats是输入特征的维度，out_feats是输出特征的维度，aggregator_type是聚合函数的类型
        self.conv1 = dglnn.SAGEConv(
            in_feats=in_feats, out_feats=hid_feats, aggregator_type='mean')
        self.conv2 = dglnn.SAGEConv(
            in_feats=hid_feats, out_feats=out_feats, aggregator_type='mean')

    def forward(self, graph, inputs):
        # 输入是节点的特征
        h = self.conv1(graph, inputs)
        h = F.relu(h)
        h = self.conv2(graph, h)
        return h
```

请注意，这个模型不仅可以做节点分类，还可以为其他下游任务获取隐藏节点表示，如： 5.2 边分类/回归、 5.3 链接预测 和 5.4 整图分类。

关于DGL内置图卷积模块的完整列表，读者可以参考 apinn。

有关DGL神经网络模块如何工作，以及如何编写一个自定义的带有消息传递的GNN模块的更多细节，请参考 第3章：构建图神经网络（GNN）模块 中的例子。

模型的训练

全图(使用所有的节点和边的特征)上的训练只需要使用上面定义的模型进行前向传播计算，并通过在训练节点上比较预测和真实标签来计算损失，从而完成后向传播。

本节使用DGL内置的数据集 `dgl.data.CiteseerGraphDataset` 来展示模型的训练。 节点特征和标签存储在其图上，训练、验证和测试的分割也以布尔掩码的形式存储在图上。这与在 第4章：图数据处理管道 中的做法类似。

```
node_features = graph.ndata['feat']
node_labels = graph.ndata['label']
train_mask = graph.ndata['train_mask']
valid_mask = graph.ndata['val_mask']
test_mask = graph.ndata['test_mask']
n_features = node_features.shape[1]
n_labels = int(node_labels.max().item() + 1)
```

下面是通过使用准确性来评估模型的一个例子。

```
def evaluate(model, graph, features, labels, mask):
    model.eval()
    with torch.no_grad():
        logits = model(graph, features)
        logits = logits[mask]
        labels = labels[mask]
        _, indices = torch.max(logits, dim=1)
        correct = torch.sum(indices == labels)
        return correct.item() * 1.0 / len(labels)
```

用户可以按如下方式实现模型的训练。

```
model = SAGE(in_feats=n_features, hid_feats=100, out_feats=n_labels)
opt = torch.optim.Adam(model.parameters())

for epoch in range(10):
    model.train()
    # 使用所有节点(全图)进行前向传播计算
    logits = model(graph, node_features)
    # 计算损失值
    loss = F.cross_entropy(logits[train_mask], node_labels[train_mask])
    # 计算验证集的准确度
    acc = evaluate(model, graph, node_features, node_labels, valid_mask)
    # 进行反向传播计算
    opt.zero_grad()
    loss.backward()
    opt.step()
    print(loss.item())

    # 如果需要的话，保存训练好的模型。本例中省略。
```

DGL的GraphSAGE样例 提供了一个端到端的同构图节点分类的例子。用户可以在 `GraphSAGE` 类中看到模型实现的细节。 这个模型具有可调节的层数、dropout概率，以及可定制的聚合函数和非线性函数。

异构图上的节点分类模型的训练

如果图是异构的，用户可能希望沿着所有边类型从邻居那里收集消息。 用户可以使用 `dgl.nn.pytorch.HeteroGraphConv` 模块(针对MXNet和PyTorch后端也有对应的模块)在所有边类型上执行消息传递， 并为每种边类型使用一种图卷积模块。

下面的代码定义了一个异构图卷积模块。模块首先对每种边类型进行单独的图卷积计算，然后将每种边类型上的消息聚合结果再相加， 并作为所有节点类型的最终结果。

```
# Define a Heterograph Conv model

class RGCN(nn.Module):
    def __init__(self, in_feats, hid_feats, out_feats, rel_names):
        super().__init__()
        # 实例化HeteroGraphConv，in_feats是输入特征的维度，out_feats是输出特征的维度，aggregate是聚合函数的类型
        self.conv1 = dglnn.HeteroGraphConv({
            rel: dglnn.GraphConv(in_feats, hid_feats)
            for rel in rel_names}, aggregate='sum')
        self.conv2 = dglnn.HeteroGraphConv({
            rel: dglnn.GraphConv(hid_feats, out_feats)
            for rel in rel_names}, aggregate='sum')

    def forward(self, graph, inputs):
        # 输入是节点的特征字典
        h = self.conv1(graph, inputs)
        h = {k: F.relu(v) for k, v in h.items()}
        h = self.conv2(graph, h)
        return h
```

`dgl.nn.HeteroGraphConv` 接收一个节点类型和节点特征张量的字典作为输入，并返回另一个节点类型和节点特征的字典。

本章的 异构图训练的样例数据 中已经有了 `user` 和 `item` 的特征，用户可用如下代码获取。

```
model = RGCN(n_hetero_features, 20, n_user_classes, hetero_graph.etypes)
user_feats = hetero_graph.nodes['user'].data['feature']
item_feats = hetero_graph.nodes['item'].data['feature']
labels = hetero_graph.nodes['user'].data['label']
train_mask = hetero_graph.nodes['user'].data['train_mask']
```

然后，用户可以简单地按如下形式进行前向传播计算：

```
node_features = {'user': user_feats, 'item': item_feats}
h_dict = model(hetero_graph, {'user': user_feats, 'item': item_feats})
h_user = h_dict['user']
h_item = h_dict['item']
```

异构图上模型的训练和同构图的模型训练是一样的，只是这里使用了一个包括节点表示的字典来计算预测值。 例如，如果只预测 `user` 节点的类别，用户可以从返回的字典中提取 `user` 的节点嵌入。

```
opt = torch.optim.Adam(model.parameters())

for epoch in range(5):
    model.train()
    # 使用所有节点的特征进行前向传播计算，并提取输出的user节点嵌入
    logits = model(hetero_graph, node_features)['user']
    # 计算损失值
    loss = F.cross_entropy(logits[train_mask], labels[train_mask])
    # 计算验证集的准确度。在本例中省略。
    # 进行反向传播计算
    opt.zero_grad()
    loss.backward()
    opt.step()
    print(loss.item())

    # 如果需要的话，保存训练好的模型。本例中省略。
```

DGL提供了一个用于节点分类的RGCN的端到端的例子 RGCN 。用户可以在 RGCN模型实现文件 中查看异构图卷积 `RelGraphConvLayer` 的具体定义。