节点预测与边预测任务实践

引言

在此小节我们将利用在上一小节6-1-数据完整存于内存的数据集类中构造的PlanetoidPubMed数据集类，来实践节点预测与边预测任务。

注：边预测任务实践中的代码来源于link_pred.py。

节点预测任务实践

之前我们学习过由2层GATConv组成的图神经网络，现在我们重定义一个GAT图神经网络，使其能够通过参数来定义GATConv的层数，以及每一层GATConv的out_channels。我们的图神经网络定义如下：

```
class GAT(torch.nn.Module):
    def __init__(self, num_features, hidden_channels_list, num_classes):
        super(GAT, self).__init__()
        torch.manual_seed(12345)
        hns = [num_features] + hidden_channels_list
        conv_list = []
        for idx in range(len(hidden_channels_list)):
            conv_list.append((GATConv(hns[idx], hns[idx+1]), 'x, edge_index -> x'))
            conv_list.append(ReLU(inplace=True),)

        self.convseq = Sequential('x, edge_index', conv_list)
        self.linear = Linear(hidden_channels_list[-1], num_classes)

```

```
    def forward(self, x, edge_index):
        x = self.convseq(x, edge_index)
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.linear(x)
        return x

```

由于我们的神经网络由多个GATConv顺序相连而构成，因此我们使用了torch_geometric.nn.Sequential容器，详细内容可见于官方文档。

我们的图神经网络类定义如下：

```
class GAT(torch.nn.Module):
    def __init__(self, num_features, hidden_channels_list, num_classes):
        super(GAT, self).__init__()
        torch.manual_seed(12345)
        hns = [num_features] + hidden_channels_list
        conv_list = []
        for idx in range(len(hidden_channels_list)):
            conv_list.append((GATConv(hns[idx], hns[idx+1]), 'x, edge_index -> x'))
            conv_list.append(ReLU(inplace=True),)

        self.convseq = Sequential('x, edge_index', conv_list)
        self.linear = Linear(hidden_channels_list[-1], num_classes)

    def forward(self, x, edge_index):
        x = self.convseq(x, edge_index)
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.linear(x)
        return x

```

我们通过hidden_channels_list参数来设置每一层GATConv的outchannel，所以hidden_channels_list长度即为GATConv的层数。通过修改hidden_channels_list，我们就可构造出不同的图神经网络。

完整的代码可见于codes/node_classification.py。请小伙伴们自行完成代码中图神经网络类的训练、验证和测试。

边预测任务实践

边预测任务，目标是预测两个节点之间是否存在边。拿到一个图数据集，我们有节点属性x，边端点edge_index。edge_index存储的便是正样本。为了构建边预测任务，我们需要生成一些负样本，即采样一些不存在边的节点对作为负样本边，正负样本数量应平衡。此外要将样本分为训练集、验证集和测试集三个集合。

PyG中为我们提供了现成的采样负样本边的方法，train_test_split_edges(data, val_ratio=0.05, test_ratio=0.1)，其

- 第一个参数为torch_geometric.data.Data对象，
- 第二参数为验证集所占比例，
- 第三个参数为测试集所占比例。

该函数将自动地采样得到负样本，并将正负样本分成训练集、验证集和测试集三个集合。它用train_pos_edge_index、train_neg_adj_mask、val_pos_edge_index、val_neg_edge_index、test_pos_edge_index和test_neg_edge_index，六个属性取代edge_index属性。

注意train_neg_adj_mask与其他属性格式不同，其实该属性在后面并没有派上用场，后面我们仍然需要进行一次训练集负样本采样。

下面我们使用Cora数据集作为例子，进行边预测任务说明。

获取数据集并进行分析

首先是**获取数据集并进行分析**:

```
import os.path as osp

from torch_geometric.utils import negative_sampling
from torch_geometric.datasets import Planetoid
import torch_geometric.transforms as T
from torch_geometric.utils import train_test_split_edges


dataset = Planetoid('dataset', 'Cora',
transform=T.NormalizeFeatures())
```

```
data = dataset[0]
data.train_mask = data.val_mask = data.test_mask = data.y = None # 不再有用
data = train_test_split_edges(data)

print(data.edge_index.shape)
# torch.Size([2, 10556])

for key in data.keys:
    print(key, getattr(data, key).shape)

# x torch.Size([2708, 1433])
# val_pos_edge_index torch.Size([2, 263])
# test_pos_edge_index torch.Size([2, 527])
# train_pos_edge_index torch.Size([2, 8976])
# train_neg_adj_mask torch.Size([2708, 2708])
# val_neg_edge_index torch.Size([2, 263])
# test_neg_edge_index torch.Size([2, 527])
# 263 + 527 + 8976 = 9766 != 10556
# 263 + 527 + 8976/2 = 5278 = 10556/2
```

我们观察到训练集、验证集和测试集中正样本边的数量之和不等于原始边的数量。这是因为，现在所用的Cora图是无向图，在统计原始边数量时，每一条边的正向与反向各统计了一次，训练集也包含边的正向与反向，但验证集与测试集都只包含了边的一个方向。

为什么训练集要包含边的正向与反向，而验证集与测试集都只包含了边的一个方向？这是因为，训练集用于训练，训练时一条边的两个端点要互传信息，只考虑一个方向的话，只能由一个端点传信息给另一个端点，而验证集与测试集的边用于衡量检验边预测的准确性，只需考虑一个方向的边即可。

边预测图神经网络的构造

接下来**构造神经网络**：

```
import torch
from torch_geometric.nn import GCNConv

class Net(torch.nn.Module):
```

```
    def __init__(self, in_channels, out_channels):
        super(Net, self).__init__()
        self.conv1 = GCNConv(in_channels, 128)
        self.conv2 = GCNConv(128, out_channels)

    def encode(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = x.relu()
        return self.conv2(x, edge_index)

    def decode(self, z, pos_edge_index, neg_edge_index):
        edge_index = torch.cat([pos_edge_index, neg_edge_index],
dim=-1)
        return (z[edge_index[0]] * z[edge_index[1]]).sum(dim=-1)

    def decode_all(self, z):
        prob_adj = z @ z.t()
        return (prob_adj > 0).nonzero(as_tuple=False).t()

```

用于做边预测的神经网络主要由两部分组成：其一是编码（encode），它与我们前面介绍的节点表征生成是一样的；其二是解码（decode），它根据边两端节点的表征生成边为真的几率（odds）。`decode_all(self, z)`用于推理（inference）阶段，我们要对所有的节点对预测存在边的几率。

边预测图神经网络的训练

定义单个epoch的训练过程

```
def get_link_labels(pos_edge_index, neg_edge_index):
    num_links = pos_edge_index.size(1) + neg_edge_index.size(1)
    link_labels = torch.zeros(num_links, dtype=torch.float)
    link_labels[:pos_edge_index.size(1)] = 1.
    return link_labels

def train(data, model, optimizer):
    model.train()

    neg_edge_index = negative_sampling(
```

```
        edge_index=data.train_pos_edge_index,
        num_nodes=data.num_nodes,
        num_neg_samples=data.train_pos_edge_index.size(1))

    optimizer.zero_grad()
    z = model.encode(data.x, data.train_pos_edge_index)
    link_logits = model.decode(z, data.train_pos_edge_index,
neg_edge_index)
    link_labels = get_link_labels(data.train_pos_edge_index,
neg_edge_index).to(data.x.device)
    loss = F.binary_cross_entropy_with_logits(link_logits,
link_labels)
    loss.backward()
    optimizer.step()

    return loss

```

通常，存在边的节点对的数量往往少于不存在边的节点对的数量。我们在每一个epoch的训练过程中，都进行一次训练集负样本采样。采样到的样本数量与训练集正样本相同，但不同epoch中采样到的样本是不同的。这样做，我们既能实现类别数量平衡，又能实现增加训练集负样本的多样性。在负样本采样时，我们传递了train_pos_edge_index为参数，于是negative_sampling()函数只会在训练集中不存在边的节点对中采样。get_link_labels()函数用于生成完整训练集的标签。

注：在训练阶段，我们应该只见训练集，对验证集与测试集都是不可见的。所以我们没有使用所有的边，而是只用了训练集正样本边。

定义单个epoch验证与测试过程

```
@torch.no_grad()
def test(data, model):
    model.eval()

    z = model.encode(data.x, data.train_pos_edge_index)

    results = []
    for prefix in ['val', 'test']:
        pos_edge_index = data[f'{prefix}_pos_edge_index']
        neg_edge_index = data[f'{prefix}_neg_edge_index']
        link_logits = model.decode(z, pos_edge_index,
neg_edge_index)
```

```
        link_probs = link_logits.sigmoid()
        link_labels = get_link_labels(pos_edge_index, neg_edge_index)
        results.append(roc_auc_score(link_labels.cpu(), link_probs.cpu()))
    return results

```

注：在验证与测试阶段，我们也应该只见训练集，对验证集与测试集都是不可见的。所以在验证与测试阶段，我们依然只用训练集正样本边。

运行完整的训练、验证与测试

```
def main():
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    dataset = 'Cora'
    path = osp.join(osp.dirname(osp.realpath(__file__)), '..', 'data', dataset)
    dataset = Planetoid(path, dataset, transform=T.NormalizeFeatures())
    data = dataset[0]
    ground_truth_edge_index = data.edge_index.to(device)
    data.train_mask = data.val_mask = data.test_mask = data.y = None
    data = train_test_split_edges(data)
    data = data.to(device)

    model = Net(dataset.num_features, 64).to(device)
    optimizer = torch.optim.Adam(params=model.parameters(), lr=0.01)

    best_val_auc = test_auc = 0
    for epoch in range(1, 101):
        loss = train(data, model, optimizer)
        val_auc, tmp_test_auc = test(data, model)
        if val_auc > best_val_auc:
            best_val_auc = val_auc
            test_auc = tmp_test_auc
        print(f'Epoch: {epoch:03d}, Loss: {loss:.4f}, Val: {val_auc:.4f}, '
              f'Test: {test_auc:.4f}')
```

```
25
26      z = model.encode(data.x, data.train_pos_edge_index)
27      final_edge_index = model.decode_all(z)
28
29
30  if __name__ == "__main__":
31      main()
32
```

完整的代码可见于codes/link_prediction.py。

结语

在完整的第6节内容中，我们学习了

- PyG中规定的使用数据的一般过程；
- InMemoryDataset基类；
- 一个简化的InMemory数据集类；
- 一个InMemory数据集类实例，以及使用该数据集类时会发生的一些过程；
- 节点预测任务实践；
- 边预测任务实践。

我们需要重点关注**InMemory数据集类的运行流程**与其**四个方法的定义的规范**，同时我们还应该重点关注边预测任务中的**数据集划分**，**训练集负样本采样**，以及**训练、验证与测试三个阶段使用的边**。

作业

- 实践问题一：尝试使用PyG中的不同的网络层去代替GCNConv，以及不同的层数和不同的out_channels，来实现节点分类任务。
- 实践问题二：在边预测任务中，尝试用torch_geometric.nn.Sequential容器构造图神经网络。

- 思考问题三：如下方代码所示，我们以`data.train_pos_edge_index`为实际参数来进行训练集负样本采样，但这样采样得到的负样本可能包含一些验证集的正样本与测试集的正样本，即可能将真实的正样本标记为负样本，由此会产生冲突。但我们还是这么做，这是为什么？

```
neg_edge_index = negative_sampling(
    edge_index=data.train_pos_edge_index,
    num_nodes=data.num_nodes,
    num_neg_samples=data.train_pos_edge_index.size(1))
```

参考资料

- `Sequential`官网文档：torch_geometric.nn.Sequential
- 边预测任务实践中的代码来源于link_pred.py