节点预测与边预测任务实践

引言

在此节的上半部分中我们学习了在PyG中如何自定义一个**数据完全存于内存的数据集类**。在这下半部分中，我们将实践节点预测与边预测任务。

通过完整的此章内容的学习，希望小伙伴们能够**掌握应对实际中节点预测问题或边预测问题的能力**。

节点预测任务实践

之前我们介绍过由2层`GATConv`组成的神经网络，现在我们重定义一个GAT神经网络，使其能够通过参数定义`GATConv`的层数，以及每一层`GATConv`的`out_channels`。我们的神经网络模型定义如下：

```
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
```

```
19
```

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。

完整的代码可见于`codes/node_classification.py`

边预测任务实践

边预测任务，如果是预测两个节点之间是否存在边。拿到一个图数据集，我们有节点特征矩阵`x`，和哪些节点之间存在边的信息`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`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负样本训练集采样。

下面我们使用Cora数据集作为例子进行边预测任务说明。

首先是**获取数据集并进行分析**：

```
1  import os.path as osp
2
3  from torch_geometric.utils import negative_sampling
4  from torch_geometric.datasets import Planetoid
5  import torch_geometric.transforms as T
6  from torch_geometric.utils import
   train_test_split_edges
7
8  dataset = 'Cora'
```

```
path = osp.join(osp.dirname(osp.realpath(__file__)),
'..', 'data', dataset)
dataset = Planetoid(path, dataset,
transform=T.NormalizeFeatures())
data = dataset[0]
data.train_mask = data.val_mask = data.test_mask =
data.y = None
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
```

我们观察到三个集合中正样本边的数量之和不等于原始边的数量。这是因为原始边的数量统计的是双向边的数量，在验证集正样本边和测试集正样本边中只需对一个方向的边做预测精度的衡量，对另一个方向的预测精度衡量属于重复，但在训练集还是保留双向的边（其实也可以可以不要，编者注）。

接下来**构建神经网络模型**：

```
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)
```

```
9
10      def encode(self, x, edge_index):
11          x = self.conv1(x, edge_index)
12          x = x.relu()
13          return self.conv2(x, edge_index)
14
15      def decode(self, z, pos_edge_index,
    neg_edge_index):
16          edge_index = torch.cat([pos_edge_index,
    neg_edge_index], dim=-1)
17          return (z[edge_index[0]] *
    z[edge_index[1]]).sum(dim=-1)
18
19      def decode_all(self, z):
20          prob_adj = z @ z.t()
21          return (prob_adj >
    0).nonzero(as_tuple=False).t()
22
```

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生成节点表征是一样的；其二是解码（decode），它边两端节点的表征生成边为真的几率（odds）。`decode_all(self, z)`用于推断（inference）阶段，我们要对输入节点所有的节点对预测存在边的几率。

定义单个epoch的训练过程

```
1  def get_link_labels(pos_edge_index, neg_edge_index):
2      num_links = pos_edge_index.size(1) +
    neg_edge_index.size(1)
3      link_labels = torch.zeros(num_links,
    dtype=torch.float)
4      link_labels[:pos_edge_index.size(1)] = 1.
5      return link_labels
6
7  def train(data, model, optimizer):
8      model.train()
9
10     neg_edge_index = negative_sampling(
11         edge_index=data.train_pos_edge_index,
12         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

```

通常在图上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。为了类平衡，在每一个`epoch`的训练过程中，我们只需要用到与正样本一样数量的负样本。综合以上两点原因，我们在每一个`epoch`的训练过程中都采样与正样本数量一样的负样本，这样我们既做到了类平衡，又增加了训练负样本的丰富性。`get_link_labels`函数用于生成完整训练集的标签。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结点对中采样。

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，但在此阶段我们应该要完成对所有结点的编码，因此我们假设此处正样本训练集涉及到了所有的结点，这样就能实现对所有结点的编码。

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

```

注意在验证与测试过程中，我们依然只用正样本边训练集做节点特征编码。

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}')

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


if __name__ == "__main__":
    main()

```

完整的代码可见于`codes/link_prediction.py`。

结语

在此篇文章中，我们介绍了定义一个数据可全部存储于内存的数据集类的方法，并且实践了节点预测任务于边预测任务。我们要**重点关注`InMemoryDataset`子类的运行流程与实现四个函数的规范。**在图神经网络的实现中，我们可以用`torch_geometric.nn.Sequential`容器对神经网络的多个模块顺序相连。

作业

- 实践问题一：对节点预测任务，尝试用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`。
- 实践问题二：对边预测任务，尝试用用`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，这样采样得到的负样本可能包含验证集正样本或测试集正样本，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？以及为什么在验证与测试阶段我们只根据`data.train_pos_edge_index`做结点表征的编码？

```
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