前置作業

套件安裝:

Torch geometric 的安裝若依照助教中提供的 code 無法順利安裝，需要查詢自己 python 和 Pytorch 的版本，依照對應的版本先進行下載對應的 torch-scatter 和 torch-sparse 再 pip install 進去；若一開始就下 pip install -q torch-geometric 發現無法順利使用 geometric 這個套件。

作業實作

Dataset 處理:

Dataset 的地方是最困擾我的部分，因為多數網路上的資料使用的資料集都是 pytorch 上本身就已經有的數據了，在使用時只要 import 就可以使用，像是 data.x、data.train_mask 等都可以直接呼叫，也可以直接把它有的資料集下載到自己的電腦，但是作業中的資料集是使用自己的，並非 pytorch 本身提供，要透過讀檔來獲取資料，然後建立成 graph。

在 synthetic/5000 這個資料夾裡包含多個圖資料的 txt 檔，本來覺得是需要使用 `from torch_geometric.data import Dataset` 這個套件自己創建資料集，後來找到可以使用 `from torch_geometric.data import InMemoryDataset` 來創建，也有順利創建出來，如下圖: (這邊先設定讀取 0.txt 到 5.txt 的資料)

```
Data(x=[0], edge_index=[2, 19982], y=[0])
Data(x=[0], edge_index=[2, 19981], y=[0])
Data(x=[0], edge_index=[2, 19980], y=[0])
Data(x=[0], edge_index=[2, 19982], y=[0])
Data(x=[0], edge_index=[2, 19984], y=[0])
Data(x=[0], edge_index=[2, 19981], y=[0])
```

可以得到 txt 檔中起始節點與終端節點的關係(edge_index)，但是 Data 中的 data.x 每個節點的特徵以及 data.y 每個節點的標籤，我不知道要怎麼在多個圖檔下做設定(做到建模型的部分我發現我遇到最大的困難是我不確定要怎麼得到 node features 和 node label，這導致後面模型要 training 和做 loss function 的時候沒辦法做下去)。

後來決定改成使用 `from torch_geometric.data import Data` 來進行資料 graph 的製作，先單獨用 0.txt 這個圖下去做實驗。

同時資料集有提供正確的 BC value，但是我不知道要怎麼把它應用在模型上面，本來以為 0_score.txt 可以用來當成 Data 裡的 y 值(如 TA1 範例中提及)，可是後來發現 y 值算是屬性，用來把資料做區分(在後面 masking 的時候會使用到)，0_score.txt 這個檔案的數據並不算是屬性，後來看到原始論文提供的程式碼中有做 initial node features 的設定(設置成[Dv, 1, 1]，node_feat_dim = 3(每一節點設定成一個具三維的 feature))，所以我把 0_score.txt 裡的 bc value 當成每一節點的 Dv 然後下去設定 data.x，建立節點特徵矩陣，可以得到結果如下圖:

```
data.x
```

```
tensor([[9.4175e-02, 1.0000e+00, 1.0000e+00],
        [5.3971e-02, 1.0000e+00, 1.0000e+00],
        [4.4344e-02, 1.0000e+00, 1.0000e+00],
        ...,
        [1.9542e-05, 1.0000e+00, 1.0000e+00],
        [6.0578e-05, 1.0000e+00, 1.0000e+00],
        [1.0908e-04, 1.0000e+00, 1.0000e+00]])
```

其中 data.y 節點標籤的地方先假設全部節點都在同一個圖，所以有一樣的標籤(都設為 0)。

最後建置出來 0.txt 的資料如下圖:

```
#dataset information
print('nodes:',data.num_nodes)
print('edges:',data.num_edges)
print(f'Average node degree: {(2*data.num_edges) / data.num_nodes:.2f}')
print('edge_index:',data.edge_index)
print(data.y)
```

```
nodes: 5000
edges: 19982
Average node degree: 7.99
edge_index: tensor([[   0,    0,    0,  ..., 4844, 4870, 4937],
        [   4,    5,    8,  ..., 4849, 4928, 4953]])
tensor([0, 0, 0,  ..., 0, 0, 0], dtype=torch.int32)
```

一開始一直找不到 Dv 是什麼意思，先隨便假設，到很後面才發現 paper 有提到 dv denotes the degrees of node v，所以後來再對 Data 的建置做微調，把 data.x 設定成[Dv, 1, 1]，dv 的計算如下:

dvnodes = (2*num_edges) / num_nodes

然後把正確的 bc value 當成 data.y 來做設定，調整結果如下圖:

```
nodes: 5000
edges: 19982
Average node degree: 7.99
edge_index: tensor([[   0,    0,    0,  ..., 4844, 4870, 4937],
        [   4,    5,    8,  ..., 4849, 4928, 4953]])
data.x:  tensor([[7.9928, 1.0000, 1.0000],
        [7.9928, 1.0000, 1.0000],
        [7.9928, 1.0000, 1.0000],
        ...,
        [7.9928, 1.0000, 1.0000],
        [7.9928, 1.0000, 1.0000],
        [7.9928, 1.0000, 1.0000]])
data.y:  tensor([9.4175e-02, 5.3971e-02, 4.4344e-02,  ..., 1.9542e-05, 6.0578e-05,
        1.0908e-04])
```

數據集分割:

這邊把節點數據 60%用來訓練、20%測試、20%驗證，先分別對 train_mask、val_mask、test_mask 做設定，然後把他們分別指定到 Data 數據集裡(data.XXX_mask)。

```
print(data.train_mask)
```

```
tensor([ True,  True,  True,  ...,  True, False, False])
```

```
print(data.val_mask)
```

```
tensor([False, False, False,  ..., False,  True, False])
```

```
print(data.test_mask)
```

```
tensor([False, False, False,  ..., False, False,  True])
```

模型設計:

原始論文把模型分成 Encoder. Decoder 兩個部分，所以作業的模型也是按照這個架構下去做設計。

Encoder:

作者選擇使用 GNN 來當作 encoder。而依照對論文的理解，作者把 encoder 再細分成三個步驟:

(1) Aggregate node neighbors:
Use a weighted sum aggregator to aggregate neighbors

(2) Combine self embedding and neighbor embedding:
使用 GRU

(3) Layer aggregation:
使用 max-pooling

在步驟(1)和(2)，論文中提到是用當層 neighborhood embedding 作為 input state，前一層的 embedding 作為 hidden state，Figure2 的地方似乎依據節點數重複很多次的(1)和(2)然後依序產生$h_v^{(1)}$一直到$h_v^{(L)}$，過程感覺是使用 GCN 來做 aggregate node neighbors 並結合 GRU 的方式，但我不知道要怎麼把每一 node 做這樣的結合，所以決定把這兩個步驟合併，改成只單獨使用 GCN 產生 node embeddings 然後直接再做 max-pooling。

```
class GCNMaxPooling(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super().__init__(aggr='max') #使用 Max pooling layer aggregator.
        self.lin = Linear(in_channels, out_channels)
        self.bias = torch.nn.Parameter(torch.zeros(out_channels))
```

所以在_init_的地方，使用 max pooling layer aggregator。

定義 forward 的地方依照下面的步驟去做設定:

1. Add self-loops to the adjacency matrix.
2. Linearly transform node feature matrix.
3. Normalize node features.
4. Propagate the messages.
5. Add bias and apply activation function.

參考程式碼中，作者在每個步驟有使用不同的方法做比較，而(3)layer aggregation，作者還考量了 mean pooling、min pooling、LSTM 等，我使用和作者一樣的方式(max-pooling)，把產生的結果當成 decoder 的 input，結果如下:

```
h = model(data.x, data.edge_index)
print('encoder GCN model output',h)
print(f'Embedding shape: {list(h.shape)}')
```

```
aggr_out: tensor([[ 0.9849,  1.0512, -0.2051,  ..., -0.3314,  0.2979, -0.5618],
        [ 0.9788,  1.0286, -0.2049,  ..., -0.3260,  0.2918, -0.5610],
        [ 0.9774,  1.0232, -0.2049,  ..., -0.3247,  0.2904, -0.5609],
        ...,
        [ 0.1941,  0.1997, -0.0409,  ..., -0.0638,  0.0568, -0.1120],
        [ 0.1945,  0.2012, -0.0409,  ..., -0.0638,  0.0572, -0.1120],
        [ 0.1947,  0.2019, -0.0409,  ..., -0.0638,  0.0573, -0.1120]],
       grad_fn=<ScatterReduceBackward0>)
encoder GCN model output tensor([[0.9849, 1.0512, 0.0000,  ..., 0.0000, 0.2979, 0.0000],
        [0.9788, 1.0286, 0.0000,  ..., 0.0000, 0.2918, 0.0000],
        [0.9774, 1.0232, 0.0000,  ..., 0.0000, 0.2904, 0.0000],
        ...,
        [0.1941, 0.1997, 0.0000,  ..., 0.0000, 0.0568, 0.0000],
        [0.1945, 0.2012, 0.0000,  ..., 0.0000, 0.0572, 0.0000],
        [0.1947, 0.2019, 0.0000,  ..., 0.0000, 0.0573, 0.0000]],
       grad_fn=<ReluBackward0>)
Embedding shape: [5000, 128]
```

最後會產生一個 128 dimension 的 embedding。

Decoder:

作者使用 two-layered MLP 作為 decoder。提供的程式碼是把它定義在同一個 class 裡面，我把它獨立出來再定義一個 decoder 的 class，如下圖:

```
#建立two layer MLP當成decoder
class TwoLayerMLP(torch.nn.Module):
    def __init__(self, in_features, hidden_features, out_features):
        super(TwoLayerMLP, self).__init__()
        self.linear1 = Linear(in_features, hidden_features)
        self.activation = ReLU()
        self.linear2 = Linear(hidden_features, out_features)

    def forward(self, x):
        out = self.linear1(x)
        out = self.activation(out)
        out = self.linear2(out)
        return out
```

TwoLayerMLP model 裡面，把 encoder 得到的 dimension=128 的 embedding 當成 input，把 hidden_feature 設定成 4 (同 paper 裡程式碼 aux_feat_dim = 4)，out_feature 設為 1，最終得到 1 維的 ranking score。

```
tensor([[ 0.3152],
        [ 0.3152],
        [ 0.3152],
        ...,
        [-0.0805],
        [-0.0805],
        [-0.0805]], grad_fn=<AddmmBackward0>)
```

Training:

最後關於 training 部分，因為 encoder、decoder 分別是用不同的 class，所以在做 training loss 的時候傳入 model(encoder)以及 modeltwo(decoder)，optimizer 使用 Adam、learning rate 和原論文一樣設定成 0.0001。

```
for epoch in range(20): #maximum_episodes = 10000
    loss = train(model, modeltwo, data, optimizer)
    acc = test(model, modeltwo, data)
    print(f"Epoch: {epoch + 1}, Loss: {loss:.4f}, Accuracy: {acc:.4f}")
```

但是在進到 train()的時候發現問題一直無限增值，train()定義如下圖:

```
def train(model, modeltwo, data, optimizer):
    model.train()
    optimizer.zero_grad()
    output = model(data.x, data.edge_index)
#     print(output)
    finaloutput = modeltwo(output)
    print(finaloutput)

#     Finalll=finaloutput.tolist()
#     print(type(Finalll))

    print('OUT TRAIN MASK',finaloutput[data.train_mask])
#     print('******************')
    print('Y TRAIN MASK',data.y[data.train_mask])
    data.y[data.train_mask] = data.y[data.train_mask].squeeze(1)
    loss = criterion(finaloutput[data.train_mask], data.y[data.train_mask])
#     loss = F.nll_loss(finaloutput[data.train_mask], data.y[data.train_mask])
    loss.backward()
    optimizer.step()
    return loss.item()
```

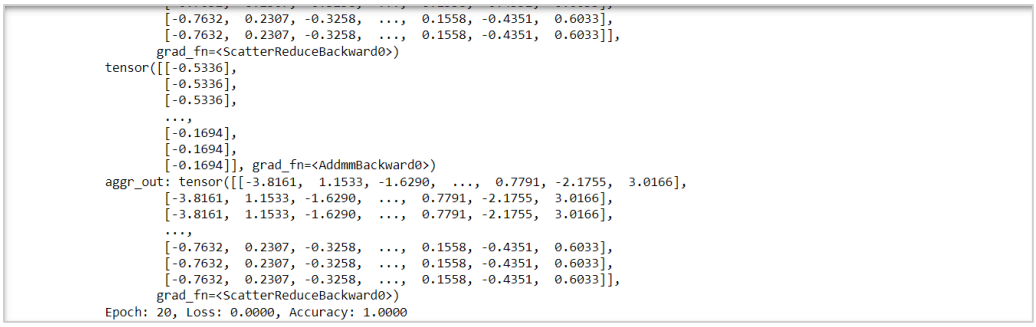
最一開始是程式跑到 loss(這邊使用 CrossEntropy 當作 loss function)的時候出現:

TypeError: only integer tensors of a single element can be converted to an index

原本以為是 finaloutput 的問題，但助教範例中 output.[data.train_mask]的 type 是 float 和我所做出的 finaloutput.[data.train_mask]的 type 相同，後來發現是 data.y 的問題，也是做到這一步才發現在一開始定義 data.x, data.y 的地方很重要，關係到 train_mask 的設計以及 loss function 加上後續的執行，所以後來詢問了助教關於 `from torch_geometric.data import Data` 設定以及定義上的問題，才有前面所提到的 data.x, data.y 的更動。後來再做其他調整後還有遇到像資料集設定等錯誤，但做了很多次調整，最後還是回到

RuntimeError: expected scalar type Long but found Float

後來 restart kernal 之後可能因為中間有改過沒有重新更新到，最後程式有順利運作沒有錯誤，跑出結果如下圖(部分截圖):



我覺得是 data.x 或 data.y 裡面的定義我還沒了解清楚以及模型中的參數要怎麼去做設置，所以沒有做適當的設定才造成無法判別模型 loss 以及 accuracy，但就前面研究要怎麼正確使用套件建立 graph、參考很多其他人建 GNN 的方式等摸索就已經花了我非常多的時間去了解，雖然多了一個星期，很可惜目前只能做到這個程度，下一步做 test 和 baseline 的比較來不及完成。