Deep Learning Midterm HW

Liam Carpenter

November 2021

Problem 1.2

1

A simple example of a polynomial that is invariant to both these groups is a polynomial of the form

$$\sum_{i=1}^{n} (x_i)^k, \forall k \in \mathbb{Z}$$

Any permutation, and therefore cyclic shift results in the same polynomial, and therefore is invariant to both cyclic and symmetric group action. This is because the order of summation does not matter for the final result.

2

A polynomial of the form

$$p(x_1, \dots, x_d) = x_1^2 x_2 + x_2^2 x_3 + \dots + x_{d-1}^2 x_d + x_d^2 x_1$$

is invariant to the cyclic group but not the symmetric group. This comes from the fact that in a cyclic shift the pairs x_{k-1}, x_k stay together, albeit are shifted. We note that $\forall g \in C_d, 1 \le k \le d$

$$p(x_1, x_2, \dots, x_d) = p(g.(x_1, x_2, \dots, x_d)) = p(x_k, x_{k+1}, \dots x_d, x_1, \dots, x_{k-1})$$

Therefore for our polynomial we have,

$$p(x_1, \dots, x_d) = x_1^2 x_2 + x_2^2 x_3 + \dots + x_{d-1}^2 x_d + x_d^2 x_1 =$$

$$x_k^2 x_{k+1} + \dots + x_{d-1}^2 x_d + x_d^2 x_1 + x_1^2 x_2 + \dots + x_{k-1}^2 x_k = p(g.(x_1, \dots, x_d))$$

So since the "neighbors" of the input are preserved and this polynomial is just a product of neighbors, it is invariant to the cyclic group. However an arbitrary permutation which breaks this pairing would not be invariant. Seen in a permuation that swap x_1, x_3 which gives elements,

$$x_1^2 x_2 + x_2^x{}_3 + x_3^2 x_4 + \dots$$

and

$$x_3^2x_2 + x_2^2x_3 + x_1^2x_4 + \dots$$

These being distinct terms we can see that the polynomial is not invariant to an arbitrary permutation.

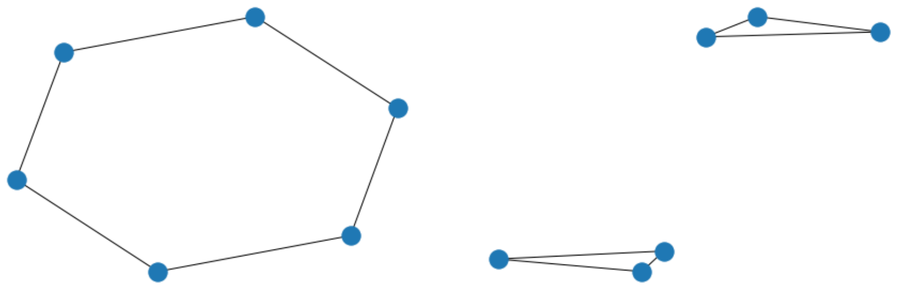
3

Since the cyclic group is a subgroup of the symmetric group it is impossible for a polynomial to be invariant to the symmetric group but not the cyclic group.

Problem 1.3

All of the code from this section can be found in the notebook provided

1



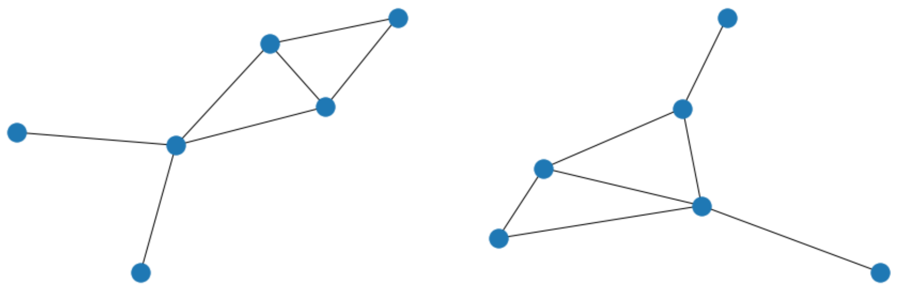
The power of a single layer graph neural network is upper-bounded by the 1-WL test for graph isomorphism. That is if the 1-WL test cannot distinguish whether two graphs are isomorphic, then neither can the single layer GNN. In this example we use a cononical failure case of the 1-WL test to identify whether these two graphs are isomorphic to each other and conclude that similarly the single layer GNN representation of these graphs would be identical.

2

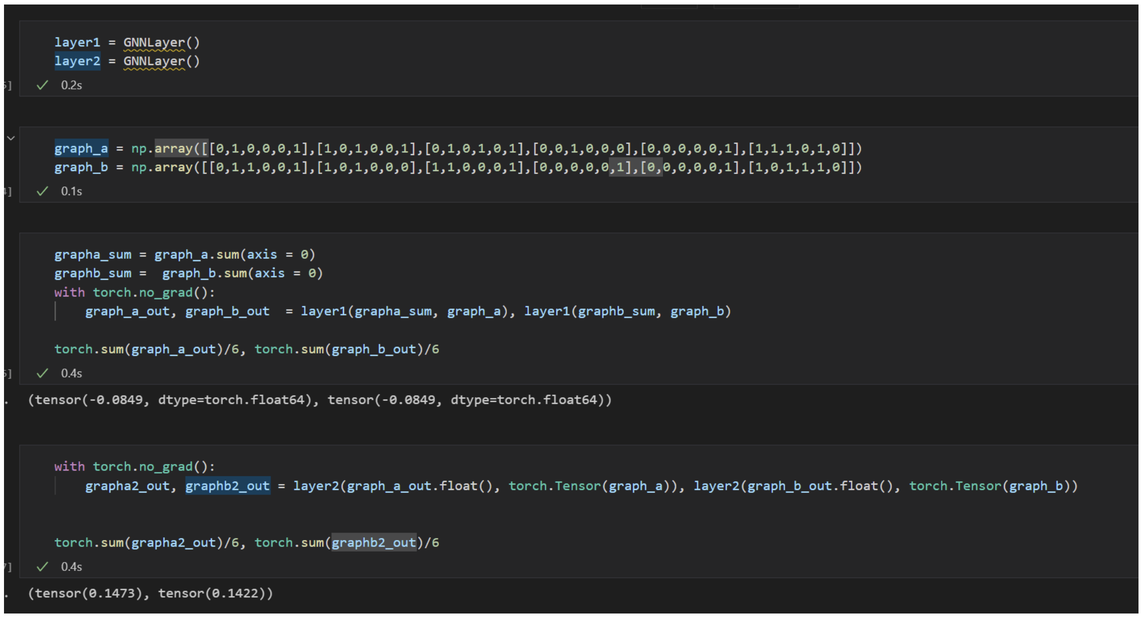
From the example we can come up with a simple classification task that the 1 layer GNN would be provably unable to solve, based on the 1-WL upper bound. We define a function $f : G \to 1, 0$ such that $f(G) = 1$ if the graph contains a triangle, $f(G) = 0$ if G does not contain a triangle.

3

Performing classification on graphs of the following types,



results in exactly the same representation for a single layer GNN but different representations for a 2 layer GNN. Intuitively the one hop neighbor information of these graphs are exactly the same, however the two hop representations of them are different from the node with four edges. In one of the graphs this node has only one neighbor with a single edge and the other has two neighbors with a single edge. From this we can perform construct a function $f : G \to \{0, 1\}$ such that $f(G) = 1$ if each node of G has at most one neighbor with only a single edge, and $f(G) = 0$ if G has a node with multiple neighbors with only a single edge. Here is an example of the computational verification that a single layer representation is the same but a two layer representation is different



```
layer1 = GNNLayer()
layer2 = GNNLayer()
```

✓ 0.2s

```
graph_a = np.array([[0,1,0,0,0,1],[1,0,1,0,0,1],[0,1,0,1,0,1],[0,0,1,0,0,0],[0,0,0,0,0,1],[1,1,1,0,1,0]])
graph_b = np.array([[0,1,1,0,0,1],[1,0,1,0,0,0],[1,1,0,0,0,1],[0,0,0,0,0,1],[0,0,0,0,0,1],[1,0,1,1,1,0]])
```

✓ 0.1s

```
grapha_sum = graph_a.sum(axis = 0)
graphb_sum =  graph_b.sum(axis = 0)
with torch.no_grad():
    graph_a_out, graph_b_out  = layer1(grapha_sum, graph_a), layer1(graphb_sum, graph_b)

torch.sum(graph_a_out)/6, torch.sum(graph_b_out)/6
```

✓ 0.4s

```
(tensor(-0.0849, dtype=torch.float64), tensor(-0.0849, dtype=torch.float64))
```

```
with torch.no_grad():
    grapha2_out, graphb2_out = layer2(graph_a_out.float(), torch.Tensor(graph_a)), layer2(graph_b_out.float(), torch.Tensor(graph_b))


torch.sum(grapha2_out)/6, torch.sum(graphb2_out)/6
```

✓ 0.4s

```
(tensor(0.1473), tensor(0.1422))
```

Problem 2

1

With the graph neural network we would like a function that is invariant to the labeling of the labels of our choice of labels in the output function. This choice of loss function however does not satisfy this property. A permutation of our labels, by multiplying the labels by -1 will result in a different loss than not doing so. If the target of an input graph is, $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ we would like the loss function to score $\begin{pmatrix} -1 \\ -1 \end{pmatrix}$ and $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ equally, since the labeling is arbitrary. The given loss function would not do this, it would attribute higher loss to the negated version.

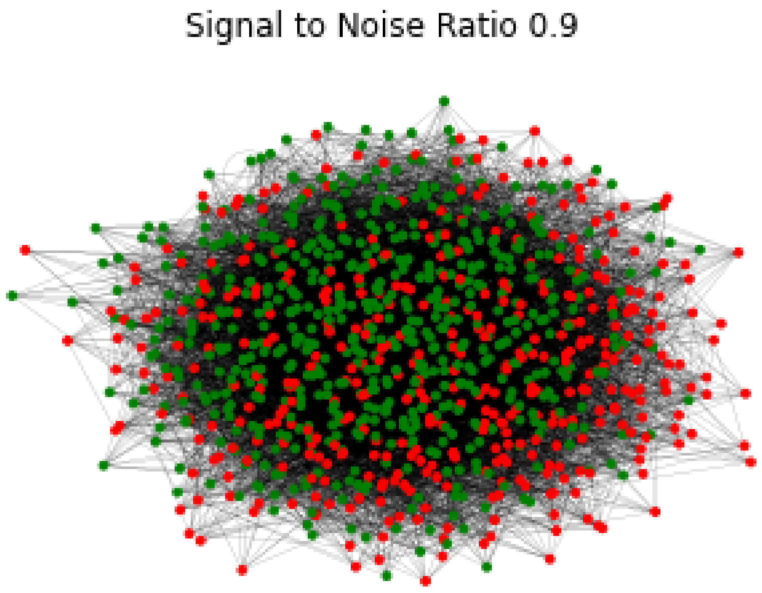
2

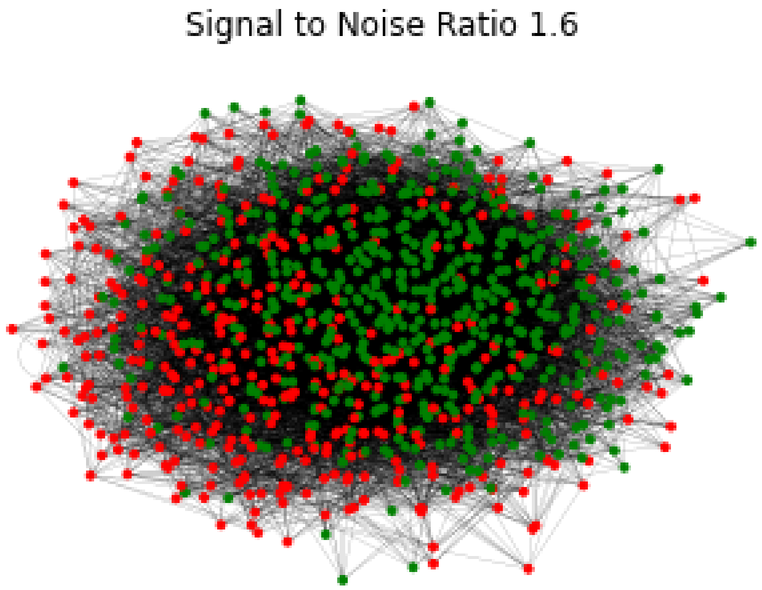
This loss function will select the loss function that is minimal over the original labels and the inverted labels. This selection of loss function gives us the property that we desire, mainly invariance to the selection of labeling of our nodes. This is a desireable property because at the end of the day we are trying to perform community detection not just node labeling and the communities are defined up to a permutation of their labels.

3

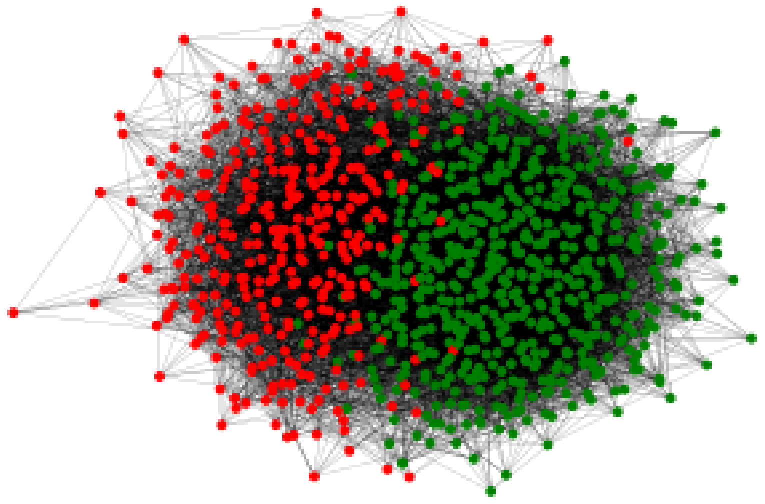
The input into our neural network is the adjacency information of our graphs. That is which edges are connected and which edges are not. Importantly we use this information to classify the nodes of the graph. So if we set $p = q$ then a node connects to another node in its class with the same probability that it connects to a node not in its class. This eliminates any information that can be drawn from edge connections and makes the task unlearnable.

4

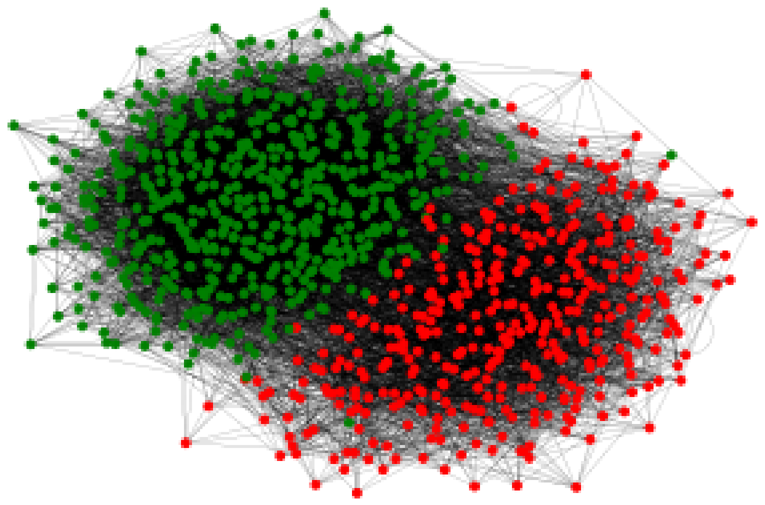




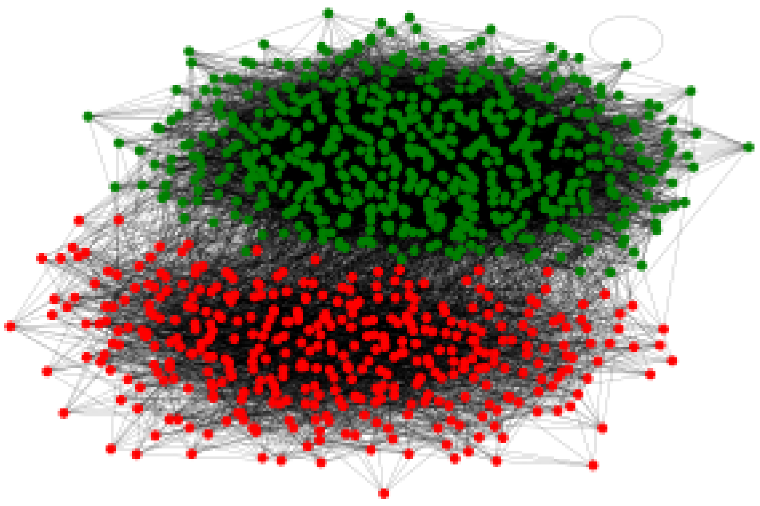
Signal to Noise Ratio 2.5



Signal to Noise Ratio 3.6



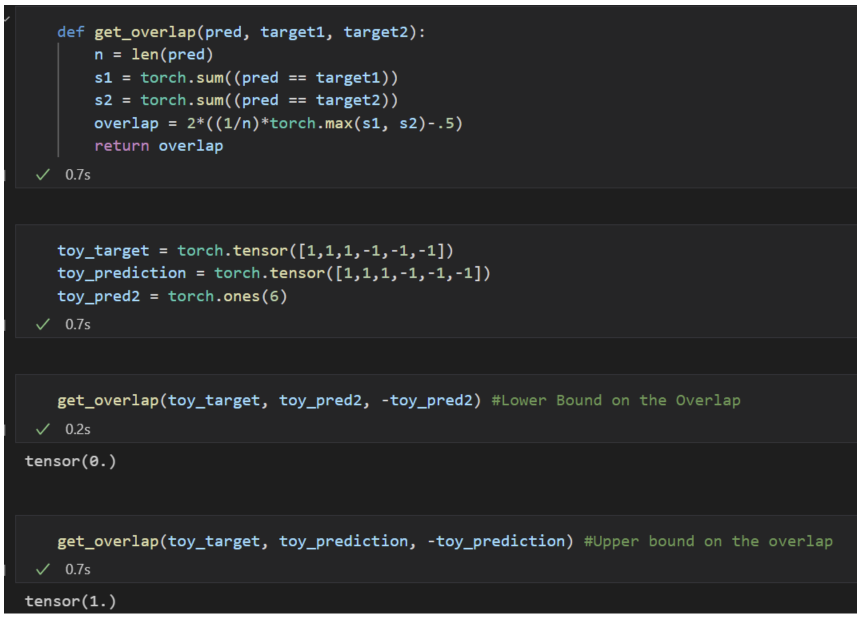
Signal to Noise Ratio 4.9



We can see that as we increase probability of connection for same class and decrease probability of connection for different classes, the groupings of like classes become more and more distinct. Likely leading to an easier classification task.

5

The overlap of our model classifies how much better than average our model does. If our model is purely random in its guesses we get an expected overlap of 0. A perfect model results in an overlap of one. Running these two examples gives verification for these bounds.



```python
def get_overlap(pred, target1, target2):
    n = len(pred)
    s1 = torch.sum((pred == target1))
    s2 = torch.sum((pred == target2))
    overlap = 2*((1/n)*torch.max(s1, s2)-.5)
    return overlap
```

✓ 0.7s

```python
toy_target = torch.tensor([1,1,1,-1,-1,-1])
toy_prediction = torch.tensor([1,1,1,-1,-1,-1])
toy_pred2 = torch.ones(6)
```

✓ 0.7s

```python
get_overlap(toy_target, toy_pred2, -toy_pred2) #Lower Bound on the Overlap
```

✓ 0.2s

```
tensor(0.)
```

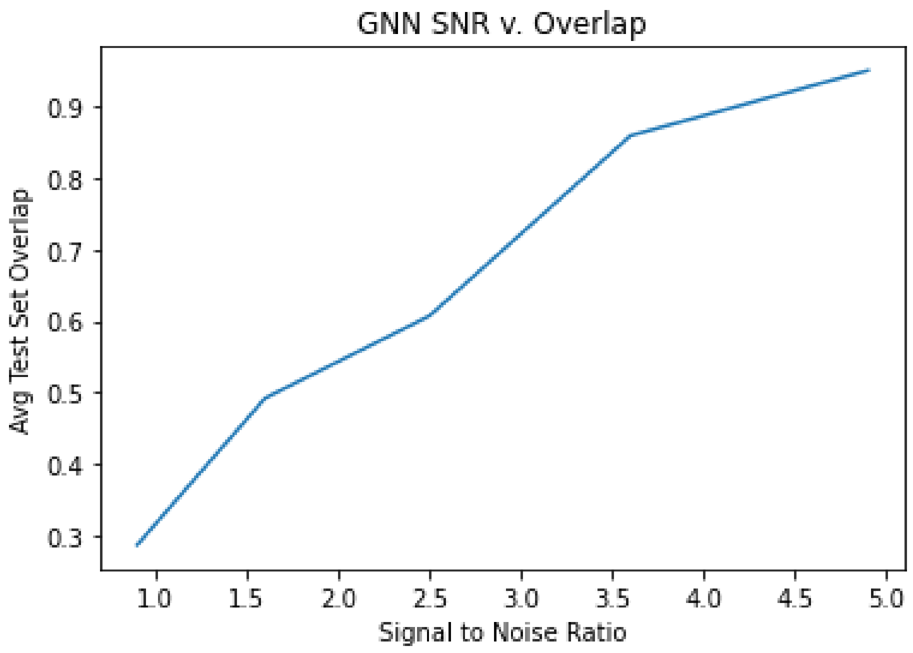
```python
get_overlap(toy_target, toy_prediction, -toy_prediction) #Upper bound on the overlap
```

✓ 0.7s

```
tensor(1.)
```

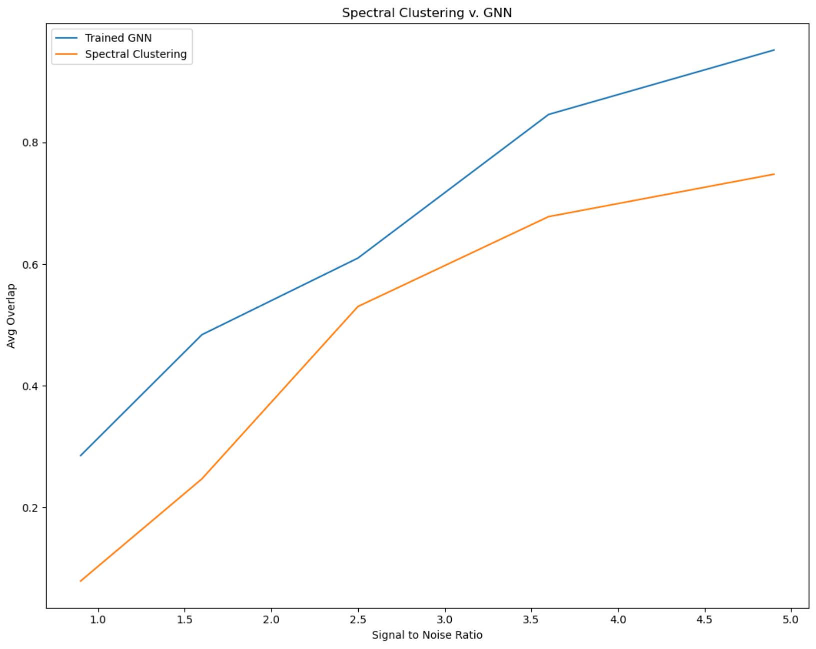
6

We can see from our figure that as the signal to noise ratio increases our model performs better and better. This aligns with our intuition that more information in the adjacency matrix improves the performance of the model.



7

Our graph neural network seems to react to the increase in signal to noise similarly to the baseline estimator, however there is a notable increase in performance consistent throughout the various signal to noise ratio's we tested them on.


Spectral Clustering v. GNN
Trained GNN
Spectral Clustering
Avg Overlap
0.2
0.4
0.6
0.8
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Signal to Noise Ratio