CS224W Homework 1

Due: Oct 16 11:59 p.m.

1 GNN Expressiveness (28 points)

For Q1.1, write down number of layers needed. For Q1.2, write down the transition matrix M and the limiting distribution r. For Q1.3 and 1.4, write down the transition matrix w.r.t A and D. For Q1.5, write down your proof in a few sentences (equations if necessary). For Q1.6, describe the message function, aggregate function, and update rule in a few sentences or equations.

Graph Neural Networks (GNNs) are a class of neural network architectures used for deep learning on graph-structured data. Broadly, GNNs aim to generate high-quality embeddings of nodes by iteratively aggregating feature information from local graph neighborhoods using neural networks; embeddings can then be used for recommendations, classification, link prediction, or other downstream tasks. Two important types of GNNs are GCNs (graph convolutional networks) and GraphSAGE (graph sampling and aggregation).

Let $G = (V, E)$ denote a graph with node feature vectors X_u for $u \in V$. To generate the embedding for a node u, we use the neighborhood of the node as the computation graph. At every layer l, for each pair of nodes $u \in V$ and its neighbor $v \in V$, we compute a message function via neural networks, and apply a convolutional operation that aggregates the messages from the node's local graph neighborhood (Figure 1.1), and updates the node's representation at the next layer. By repeating this process through K GNN layers, we capture feature and structural information from a node's local K-hop neighborhood. For each of the message computation, aggregation, and update functions, the learnable parameters are shared across all nodes in the same layer.

We initialize the feature vector for node X_u based on its individual node attributes. If we already have outside information about the nodes, we can embed that as a feature vector. Otherwise, we can use a constant feature (vector of 1) or the degree of the node as the feature vector.

These are the key steps in each layer of a GNN:

- **Message computation**: We use a neural network to learn a message function between nodes. For each pair of nodes u and its neighbor v, the neural network message function can be expressed as $M(h_u^k, h_v^k, e_{u,v})$. In GCN and GraphSAGE, this can simply be $\sigma(Wh_v + b)$, where W and b

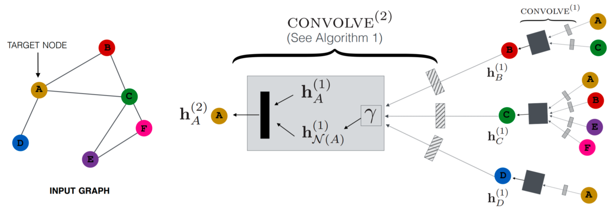

Figure 1.1: GNN architecture

are the weights and bias of a neural network linear layer. Here h_u^k refers to the hidden representation of node u at layer k, and $e_{u,v}$ denotes available information about the edge (u, v), like the edge weight or other features. For GCN and GraphSAGE, the neighbors of u are simply defined as nodes that are connected to u. However, many other variants of GNNs have different definitions of neighborhood.

- **Aggregation**: At each layer, we apply a function to aggregate information from all of the neighbors of each node. The aggregation function is usually permutation invariant, to reflect the fact that nodes' neighbors have no canonical ordering. In a GCN, the aggregation is done by a weighted sum, where the weight for aggregating from v to u corresponds to the (u, v) entry of the normalized adjacency matrix $D^{-1/2}AD^{-1/2}$.

- **Update**: We update the representation of a node based on the aggregated representation of the neighborhood. For example, in GCNs, a multi-layer perceptron (MLP) is used; Graph-SAGE combines a skip layer with the MLP.

- **Pooling**: The representation of an entire graph can be obtained by adding a pooling layer at the end. The simplest pooling methods are just taking the mean, max, or sum of all of the individual node representations. This is usually done for the purposes of graph classification.

We can formulate the Message computation, Aggregation, and Update steps for a GCN as a layer-wise propagation rule given by:

$$h^{k+1} = \sigma(D^{-1/2}AD^{-1/2}h^kW^k) \tag{1}$$

where h^k represents the matrix of activations in the k-th layer, $D^{-1/2}AD^{-1/2}$ is the normalized adjacency of graph G, W_k is a layer-specific learnable matrix, and σ is a non-linearity function. Dropout and other forms of regularization can also be used.

We provide the pseudo-code for GraphSAGE embedding generation below. This will also be relevant to the questions below.

Algorithm 1: Pseudo-code for forward propagation in GraphSAGE

```
Input  : Graph $G(V, E)$; input features $\{x_v, \forall v \in V\}$; depth $K$;
         non-linearity $\sigma$; weight matrices $\{W^k, \forall k \in [1, K]\}$;
         neighborhood function $\mathcal{N} : v \to 2^V$;
         aggregator functions $\{\text{AGGREGATE}_k, \forall k \in [1, K]\}$
Output: Vector representations $z_v$ for all $v \in V$

$h_v^0 \leftarrow x_v, \forall v \in V$ ;
for $k = 1...K$ do
    for $v \in V$ do
        $h_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{h_u^{k-1}, \forall u \in \mathcal{N}(v)\})$ // aggregation
        $h_v^k \leftarrow \sigma\left(W^k \cdot \text{CONCAT}(h_v^{k-1}, h_{\mathcal{N}(v)}^k)\right)$ // MLP with skip
        connection
    $h_v^k \leftarrow h_v^k / \|h_v^k\|_2, \forall v \in V$ // update step
$z_v \leftarrow h_v^K, \forall v \in V$
```

In this question, we investigate the effect of the number of message passing layers on the expressive power of Graph Convolutional Networks. In neural networks, expressiveness refers to the set of functions (usually the loss function for classification or regression tasks) a neural network is able to compute, which depends on the structural properties of a neural network architecture.

.

1.1 Effect of Depth on Expressiveness (4 points)

Consider the following 2 graphs in figure 1.2, where all nodes have 1-dimensional initial feature vector $x = [1]$. We use a simplified version of GNN, with no non-linearity, no learned linear transformation, and sum aggregation. Specifically, at every layer, the embedding of node v is updated as the sum over the embeddings of its neighbors (N_v) and its current embedding h_v^t to get h_v^{t+1}. We run the GNN to compute node embeddings for the 2 red nodes respectively. Note that the 2 red nodes have different 5-hop neighborhood structure (note this is not the minimum number of hops for which the neighborhood structure of the 2 nodes differs). How many layers of message passing are needed so that these 2 nodes can be distinguished (i.e., have different GNN embeddings)? Explain your answer in a few sentences.

.

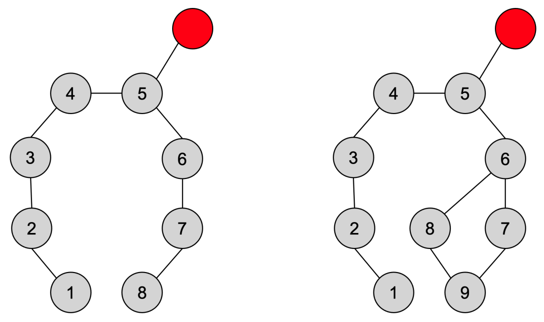

Figure 1.2: Figure for Question 1.1

★ **Solution** ★

Let A and B be the left and right graphs respectively.
And let node 0 be the node of interest (red node) in both graphs.
The numbers of layers of message passing needed to obtain different GNN embeddings is **3**. Since:

$\mathrm{h(A)}_v^0 = [1]$
$h(A)_0^1 = \sum\{h_5^0\}$
$h(A)_0^2 = \sum\{h_4^1, h_6^1\}$
$h(A)_0^3 = \sum\{h_3^2, h_7^2\}$

$h(B)_v^0 = [1]$
$h(B)_0^1 = \sum\{h_5^0\}$
$h(B)_0^2 = \sum\{h_4^1, h_6^1\}$
$h(B)_0^3 = \sum\{h_3^2, h_7^2, h_8^2\}$

1.2 Random Walk Matrix (4 points)

Consider the graph shown below (figure 1.3).

1. Assume that the current distribution over nodes is $r = [0, 0, 1, 0]$, and after the random walk, the distribution is $M \cdot r$. What is the random walk transition matrix M, where each column of M corresponds with the node ID of the node that you are transitioning from?

2. What is the limiting distribution r, namely the eigenvector of M that has an eigenvalue of 1 ($r = Mr$)? Write your answer in fraction form or round it to the nearest thousandth place and in the following form, e.g. $[1.200, 0.111, 0.462, 0.000]$. Note that before reporting you should normalize r **Hint: r is a probability distribution representing the random walk probabilities for each node after a large number of timesteps**.

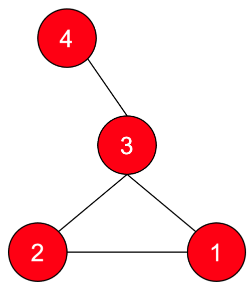

Figure 1.3: Figure for Question 1.2

★ Solution ★

1. The matrix M is:

$$M = \begin{bmatrix} 0 & 1/2 & 1/3 & 0 \\ 1/2 & 0 & 1/3 & 0 \\ 1/2 & 1/2 & 0 & 1 \\ 0 & 0 & 1/3 & 0 \end{bmatrix} \tag{2}$$

Therefore, the new random walk distribution is given by:

$$r = \begin{bmatrix} 0 & 1/2 & 1/3 & 0 \\ 1/2 & 0 & 1/3 & 0 \\ 1/2 & 1/2 & 0 & 1 \\ 0 & 0 & 1/3 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 0 \\ 1/3 \end{bmatrix} \tag{3}$$

2.
After 100 random walk iterations, the limiting distribution r, converges to:
$[0.25, 0.25, 0.375, 0.125]$

1.3 Learning BFS with GNN (7 points)

Next, we investigate the expressive power of GNN for learning simple graph algorithms. Consider breadth-first search (BFS), where at every step, nodes that are connected to already visited nodes become visited. Suppose that we use GNN to learn to execute the BFS algorithm. Suppose that the embeddings are 1-dimensional. Initially, all nodes have input feature 0, except a source node which has input feature 1. At every step, nodes reached by BFS have embedding 1, and nodes not reached by BFS have embedding 0. Describe a message function, an aggregation function, and an update rule for the GNN such that it learns the task perfectly.

★ **Solution** ★

First Approach: Using **OR** Operator

Message Function:
For this particular case. It's simply the embedding of the node:

$$m_{u\to v}^{t} = h_{u\to v}^{t} \tag{4}$$

Aggregation Function:

$$M_u^t = \bigvee_{v\in N(u)} (h_v^t) \tag{5}$$

Update Rule:

$$h_u^{t+1} = h_u^t \bigvee M_u^t \tag{6}$$

! Important: This approach won't be valid in practice. Since OR operation is neither differentiable nor a continuos function. Despite the fact to be permutation invariant.

Second Approach

Message Function:
For this particular case. It's simply the embedding of the node:

$$m_{u\to v}^{t} = h_{u\to v}^{t} \tag{7}$$

Aggregation Function:

$$M_u^t = max_{v\in N(u)}(h_u^t) \tag{8}$$

Update Rule:

$$h_u^{t+1} = max(h_u^t, M_u^t) \tag{9}$$