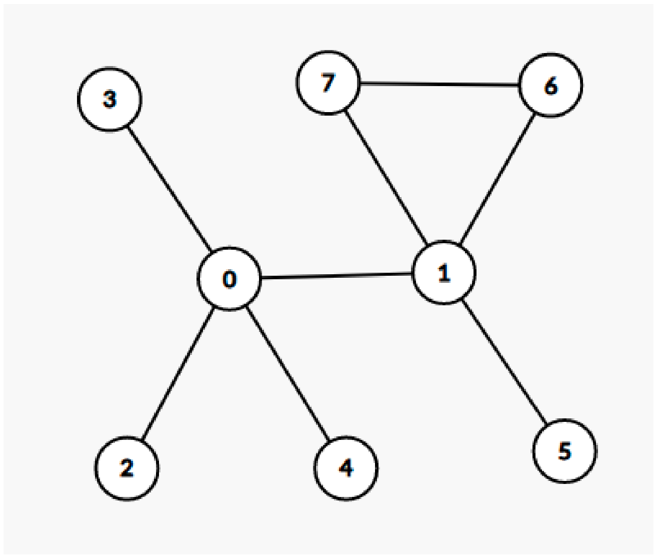
🚃

Topics in Deep Learning - Assignment 2

Question 1: Random Walk Encodings



1.1 Computing the Random Walk Embeddings

DeepWalk - Random Walk Generation

```
random-walks = [[5, 1, 7, 6], [5, 1, 0, 3]]
skipgram-context-pais = [(5, 1), (1, 7), (7, 6), (5, 1), (1, 0), (0, 3)]
```

We randomly walks over the graph, sample the neighborhoods and then proceed to train a skipgram over the generated sequences based on the skipgram context pairs.

Node2Vec - Random Walk Generation

```
random-walks = [[1, 0, 3, 0], [1, 6, 7, 6]]
skipgram-context-pais = [(1, 0), (0, 3), (3, 0), (1, 6), (1, 0), (0, 3)]
```

We walk on the graph balancing BFS and DBS link behavior, similar to above, but downweighting going back to the parent node, and upweighting going to some 1st neighbor of the previous node. This helps reduce redundancy of random walks while still promoting BFS like behaviour to generate local features.

Struct2Vec - Random Walk Generation

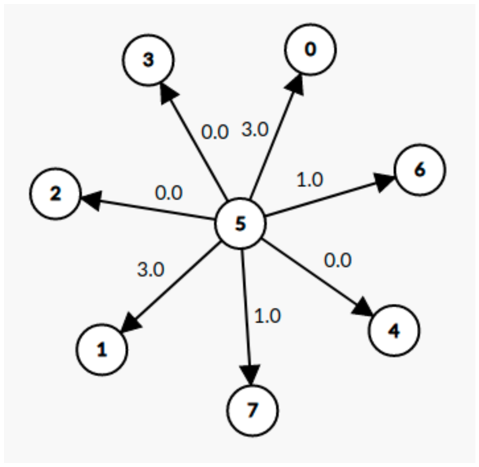
```
random-walks = [[1, 3, 3, 0], [6, 6, 0, 3], [1, 2, 6, 2], [6, 5, 6, 6]]
skipgram-context-pais = [(1, 3), (3, 0), (6, 0), (0, 3), (1, 2),
                         (2, 6), (6, 2), (6, 5), (5, 6)]
```

The walk is now down on a 3 layer fully connected graph. The neighbors of one node are shown below as an illustration. We take probability 20% of switching between 2 layers.

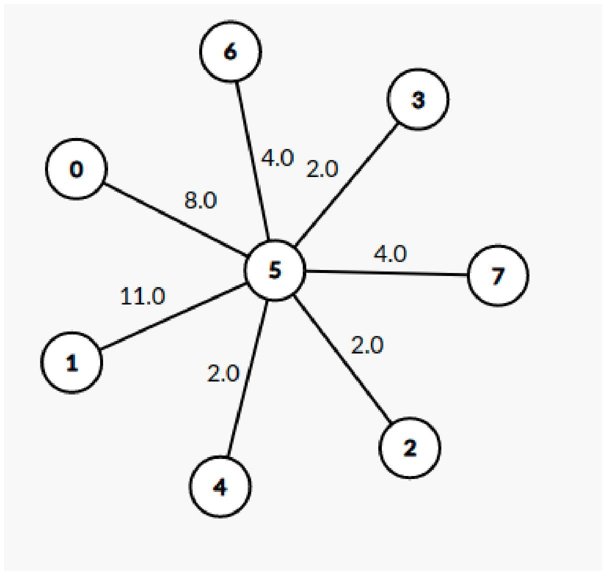
We are taking radius

The weights are calculated based on similarity of degree sequences. Eg. weight of connection between Nodes 5 and 1 is weighted due to the following weights over the different iterations:

- Radius 0: Degree Sequence of 5 is `(1)` and 1 is `(4)` leading to similarity of 3.0, cumulative similarity which is the edge weight is **3.0**.
- Radius 1: Degree Sequence of 5 is `(4)` and 1 is `(1,2,2,4)` leading to similarity of 3.0, cumulative similarity which is the edge weight is **6.0**.
- Radius 2: Degree Sequence of 5 is `(2,2,4)` and 1 is `(1,1,1)` leading to similarity of 5.0, cumulative similarity which is the edge weight is **11.0**.



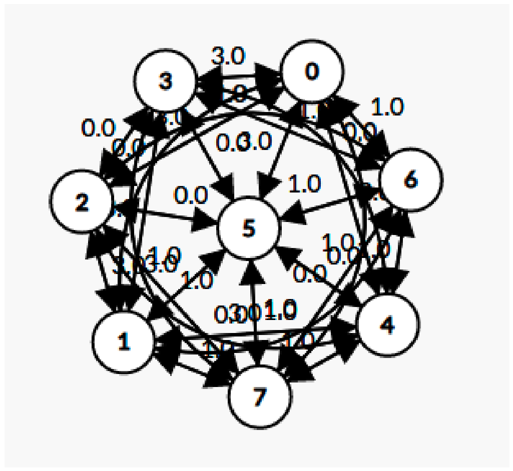
Layer 0: Weights from node 5



Layer 2: Weights from node 5

We make a 3 layer fully-connected graph, the weights from node 5 are shown above. The graph at each layer is fully connected, which is shown to the right.

The results of the random walks are shown above, we note that if the same node is encountered more than once, it's because we are switching layers.



SkipGram Model

Now that we have a lot of context pairs, we will try to get our neural net to take one hot vectors as input and give outputs such that all context pawhenirs have high cosine similarity and and we will pick random samples (typically called negative samples, but in truth they are random samples) and try to minimize their similarity.

1.2 Comparison of the Algorithms

Even though this graph is small, we want nodes `2`, `3`, `4` to be identical on the similarity metric, nodes `5` to be similar, `6` and `7` to be less similar. And we want `0` and `1` to be at a distance. The comments on the node embeddings follow from these realizations.

- **DeepWalk**: DeepWalk does not encode structure, just encodes the actual neighborhoods. This fails to node that Node `5` is more similar to `2`, `3`, and `4` than node `6` and `7` are. Though most of the embeddings will be reasonable and 6 and 7 will be together. The structure information is lost but the neighborhoods are preserved, so `0` will get closer to `2`, `3`, `4` and `1` will to `5`, `6`, `7`. This does not seem like our primary desire.
- **Node2Vec**: Node2Vec is similar to DeepWalk, but helps in exploring new regions of the graph, balancing exploration on small and large ranges. Hothis case the backpropagation equations would simplify as the network would be a simple linear system.wever, since all neighbors are in atmost 4 hop distance, and nodes are very few, exploration balancing is not an issue. The deficiencies of this algorithm are similar to those of DeepWalk.
- **Struct2Vec**: It is the best at encoding structure which generates the neighborhood structure at multiple levels and generates in true form the targets listed above. `2`, `3`, `4`, `5` are similar to each other on the first level, and similarly on higher levels we keep getting large edge weights between elements we expect to be structurally similar.

1.3 Transition Matrix and Steady State

Following is the transition matrix of the given graph:

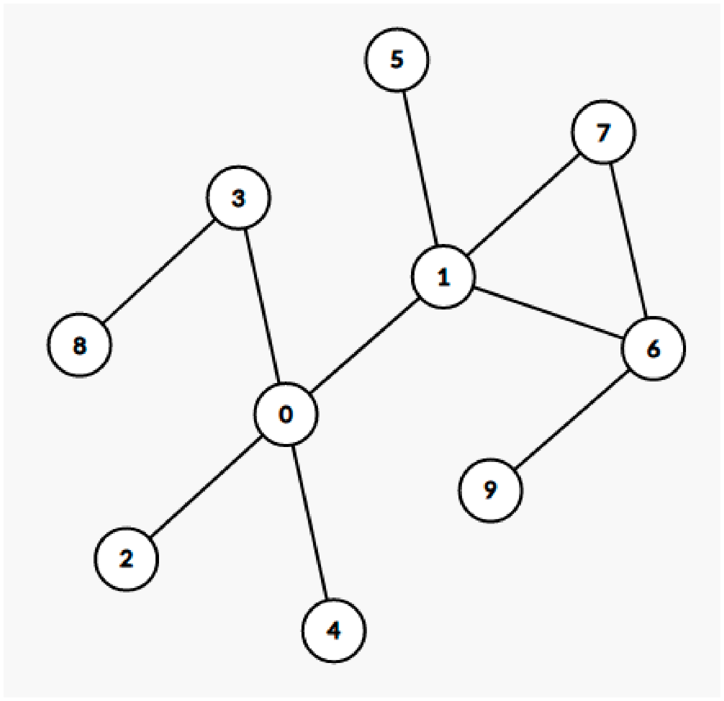
$$\begin{bmatrix} 0. & 0.25 & 1. & 1. & 1. & 0. & 0. & 0. \\ 0.25 & 0. & 0. & 0. & 0. & 1. & 0.5 & 0.5 \\ 0.25 & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0.25 & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0.25 & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0.25 & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0.25 & 0. & 0. & 0. & 0. & 0. & 0.5 \\ 0. & 0.25 & 0. & 0. & 0. & 0. & 0.5 & 0. \end{bmatrix}$$

The +1 eigenvector of our graph is the following. It does not directly correspond to a steady state distribution due to the bipartite nature of the graph (since it's tree like except for one node).

$$\begin{bmatrix} -0.2181 & 0.1233 & -0.1508 & 0.2059 & 0.0450 & 0.0000 & -0.8142 & 0.4781 \end{bmatrix}$$

Question 2: Graph Neural Networks

To be able to show backpropagation over neural networks, we must assume that there exist no non-linearities in our network. In this case the backpropagation equations would simplify as the network would be a simple linear system.



2.1 Iterations on GCN

The weight matrices be: $\begin{bmatrix} 1.0 & 2.0 \\ 1.5 & 1.0 \end{bmatrix} \begin{bmatrix} 0.3 & 0.6 \\ 1.9 & 2.2 \end{bmatrix} \begin{bmatrix} 0.8 & 2.3 \\ 1.2 & 0.9 \end{bmatrix}$ for the 3 GNN layers, taking 2 feature inputs and 2 feature outputs.

Forward and Backward pass over single layer GCN

Mean of neighbor features will be $\begin{bmatrix}1.0 & 1.0\end{bmatrix}$ for all nodes. We also take a simplifying assumption that $W = B$, so the mean is done over all nodes including self loops, meaning that matrix used for self is the same as that for others.

Let the expected classifcation output for all nodes be $\begin{bmatrix}1.0 & 0.0\end{bmatrix}$, let's use Mean Squared Error as the loss function.

$$y_{u_i} = \sum_{v \in \mathcal{N}(u) \cap \{u\}} w_{ij} (x_v)_j$$

$$\frac{L_{u_i}}{\partial w_{ij}} = \frac{L_{u_i}}{\partial y_j} \frac{\partial y_j}{\partial w_{ij}} = 2(y_{u_i} - \hat{y_{u_i}}) \sum_{v \in \mathcal{N}(u) \cap \{u\}} (x_v)_j$$

So the final derivative computation gives us:

$$\frac{L}{\partial w_{ij}} = \sum_{u \in G} 2(y_{u_i} - \hat{y_{u_i}}) \sum_{v \in \mathcal{N}(u) \cap \{u\}} (x_v)_j$$

For `i = 0` the losses will be 3 and for `i = 1` it will be 2.5. $(x_v)_j$ is always 1. The later term evaluates to be twice the sum of degree of the all nodes. Multiplying by a small learning rate $\epsilon = 10^{-3}$, we get the following update.

$$W \leftarrow \begin{bmatrix}1.0 & 2.0 \\ 1.5 & 1.0\end{bmatrix} - \epsilon \begin{bmatrix}3.0 & 3.0 \\ 2.5 & 2.5\end{bmatrix}$$

In case a non-linearity like sigmoid is met, we know that $\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$, and similarly for other non-linearities, and we will use the chain rule to compute back.

Similarly future passes can be computed. The computational results are as follows:

$$\begin{bmatrix}0.996 & 1.993 \\ 1.496 & 0.993\end{bmatrix}, \begin{bmatrix}0.994 & 1.989 \\ 1.494 & 0.989\end{bmatrix}, \begin{bmatrix}0.994 & 1.989 \\ 1.494 & 0.989\end{bmatrix}$$

after 1, 2 and 3 iterations of backpropagation.

Simulating 2 and 3 Layer GCN Forward Pass

Initial Node Embeddings are.

$$\begin{bmatrix}1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0\end{bmatrix}$$

First layer of operations leads to the following embeddings.

$$\begin{bmatrix}2.5 & 3.018 & 3.018 & 3.018 & 3.018 & 2.5 & 2.875 & 2.5 & 2.5 & 2.271 \\ 3.0 & 3.621 & 3.621 & 3.621 & 3.621 & 3.0 & 3.449 & 3.0 & 3.0 & 2.725\end{bmatrix}$$

The second layer leads to these.

$$\begin{bmatrix} 6.45 & 8.454 & 8.454 & 8.454 & 8.454 & 7.118 & 8.284 & 7.118 & 7.118 & 5.957 \\ 8.1 & 10.616 & 10.616 & 10.616 & 10.616 & 8.939 & 10.403 & 8.939 & 8.939 & 7.481 \end{bmatrix}$$

And the final layer gives us these embeddings.

$$\begin{bmatrix} 14.88 & 20.273 & 20.273 & 20.273 & 20.273 & 17.962 & 21.036 & 17.962 & 17.962 & 14.673 \\ 22.125 & 30.144 & 30.144 & 30.144 & 30.144 & 26.707 & 31.278 & 26.707 & 26.707 & 21.817 \end{bmatrix}$$

2.2 Comparing the Different Networks

Graph Convolutional Networks are best at modelling the local features of a neighborhood and learning features of a node based on those around.

Graph Isomorphic Networks are best at capturing structural differences in the graph. Since we are focussing only on the structure right now and our goal is not to merely aggregate neighborhood information.

GraphSAGE is similar to GCN, but works on aggregating information from some sampled set to perform inductive learning, which is useful in the context of large graph but only hurts when dealing with a static graph with 8 nodes.

GCN is inferior to GIN in most structural tasks, and better at neighborhood data aggregation. Due to the reasons listed above, we consider GIN to be superior to it.

2.3 Over Smoothing and Receptive Fields

Definitions of Smoothing and Receptive Fields

The Receptive Field of a Graph Neural Net is the set of node that it has aggregated information from, directly or indirectly. On the `i`-th layer of a Graph Network, all nodes at distance `i` or less are part of it's receptive field.

The problem of over-smoothing is that the receptive fields of Graph Neural Networks very quickly grow as big as the entire graph.

Receptive Field size for our Graph

For our graph, the optimal receptive field size seems to be 3.

- This is because the diameter of the graph is 5, which means that all nodes are in 5 hops to each other. So any value near (i.e. 4) or equal and above 5 will cause significant over smoothing.
- And 0 and 1 seem to not suffice for aggregating sufficient structure information, since differences in nodes `3` and `7` are not captured by it.
- So we are left with receptive fields of size 2-3, both might be apt, but we err on the side of capturing all information and not worrying above over smoothing on an initial guess. On testing, both the values converge to optimal results for simple binary classification tasks.

Skip Connections

Yes skip connections help, in all GNNs since we can afford to over smooth in the further layers, but also have the old features. This would help capture both the local and global features of the graph and add them together through skip connections. So if the discriminative power of the neural nets is not enough to distinguish any two nodes using features of later layers, it can resort to using features of earlier layers.

Question 3: GNN Implementations

3.1 Build GNN Abstraction

The code for this is in `q3/base.py`.

3.2 GCN Implementation

The following are results on running the simulations over the different types of normalization. Symmetric seems to be the best of these, though these results are susceptible to change if we try different hyperparameters, training times, or even random seeds.

Row Normalization	67.8%
Column Normalization	67.6%
Symmetric Normalization	67.9%

- Row normalization helps normalize the values aggregated at each node, so that the magnitude of vectors at each node are not dependent on the degree (connectedness) of that node.
- Column normalization helps in a similar normalization, but not on the stored magnitude of the vectors but on the incoming messages. This is similar except that the final layer is unnormalized, and the initial embeddings are messaged post normalization.
- Symmetric normalization normalizes based on the geometric mean of degrees of both the sender and the receiver of the nodes. This ensures that neither

To replicate these results, run `cd q3/` *and then run the following for each of the normalization:*

- `python main.py --task citeseer --normalization row`
- `python main.py --task citeseer --normalization column`
- `python main.py --task citeseer --normalization symmetric`

3.3 GCN vs. GIN Comparative

The aim of Graph Isomorphism networks is to increase the discriminative power of Graph Neural Networks over GCN and make it equal to that of the WL test.

Update equations for Graph Convolutional Networks (GCN):

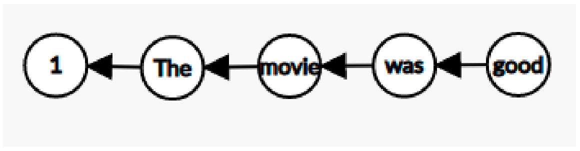
$$h_u^{(t)} = f\left(W \frac{1}{|\mathcal{N}(u)|} \sum_{v \in \mathcal{N}(u)} h_v^{(t-1)} + B h_u^{(t-1)} + b\right)$$

Update equations for Graph Isomorphism Networks (GIN):

$$h_u^{(t)} = f_\theta\left(\sum_{v \in \mathcal{N}(u)} h_v^{(t-1)} + (1+\epsilon)h_u^{(t-1)} + b\right)$$

Graph Information Networks achieve a **higher performance at 65.1%**, significantly under-reaching the other method. It seems that higher discriminative power over subgraph isomorphisms doesn't particularly help and very simply aggregating neighbor messages with linear transforms and single non-linearity is a better idea.

3.4 RNN as a Graph Network



73.2% training accuracy was achieved in 500 training iterations. The accuracy keeps changing, and sufficient iterations could not be completed on hardware due to the `for` loop in aggregation slowing down the compute, a **68% test accuracy** approximate was estimated.

3.5 Packaged Upload

The entire module is uploaded, run using the following commands:

```
git clone git@github.com:AnimeshSinha1309/graphml-assignments
cd graphml-assignments/assignment2/q3
python -m pip install -r requirements.txt
python main.py --help
```