Graph Neural Networks
Script based on CS224W: Winter 2021

Unity05

December 27, 2021

Contents

1 Message Passing Graph Neural Networks

1.1 Introduction

The idea behind message passing GNNs is **k - hop neighborhood aggregation**. One single GNN layer can be looked at as one single hop.

A GNN layer mainly consists of two parts: **Message Computation** and **Aggregation**.

1.1.1 Message Computation

Each node computes a message based on it's embedding in the previous layer.

$$m_u^{(l)} = \phi^{(l)} \left(h_u^{(l-1)} \right)$$

$m_u^{(l)}$	...	message of node u in layer l
$\phi^{(l)}$	...	message computation function of layer l
$h_u^{(l-1)}$	...	node u's embedding in layer l - 1

1.1.2 Aggregation

Node v's new embedding is computed by aggregating its own message as well as all of its neighbor node's messages.

$$h_v^{(l)} = \sigma \left(\square^{(l)} \left(\{ m_u^{(l)} \mid u \in N(v) \}, m_v^{(l)} \right) \right)$$

σ	...	nonlinear activation function
$h_v^{(l)}$	...	node v's new embedding in layer l - 1
$\square^{(l)}$	...	aggregation function of layer l
$m_u^{(l)}$	...	message of node u in layer l
$N(v)$	...	neighborhood of node v

1.2 Popular GNN Layers

1.2.1 Graph Convolutional Networks (GCN)

Message Computation
Embeddings are passed through a linear layer (transformation with weight matrix).
Normalized by node degrees.

$$m_u^{(l)} = \left(W^{(l)} \cdot h_u^{(l-1)} \right)$$

Aggregation

$$h_v^{(l)} = \sigma \left(\sum_{u \in N(v) \cup \{v\}} \frac{1}{\sqrt{deg(u)} \cdot \sqrt{deg(v)}} \cdot m_u^{(l)} \right)$$

1.2.2 Graph Attention Networks (GAT)

Message Computation
No difference to GCN.

$$m_u^{(l)} = \left(W^{(l)} \cdot h_u^{(l-1)} \right)$$

Aggregation
Weighted summation of messages normalized by attention weights.

$$h_v^{(l)} = \sigma \left(\sum_{u \in N(v) \cup \{v\}} \alpha_{vu} \cdot m_u^{(l)} \right)$$

Computation of α_{vu} with attention mechanism a:

$$\alpha_{vu} = \frac{exp(e_{vu})}{\sum_{k \in N(v)} exp(e_{vk})}$$

$$e_{vu} = a \left(W^{(l)} h_u^{(l-1)}, W^{(l)} h_v^{(l-1)} \right)$$

1.2.3 Graph Isomorphism Network (GIN)

We don't split up in Message Computation and Aggregation as MLP models both functions ϕ and f.

$$h_v^{(l)} = MLP^{(l)} \left(\left(1 + \epsilon^{(l)}\right) \cdot h_v^{(l-1)} + \sum_{u \in N(v)} h_u^{(l-1)} \right)$$

1.3 Expressiveness of GNNs

To assess the expressiveness of a GNN we have to look at its **computational graph**. It is important to keep in mind that the computational graph can only distinguish different node features not node IDs. Therefore, they are identical to subtrees rooted at the respective nodes.

Hence, we want GNNs to **map subtrees injectively**. This means we need to retain information about neighborhood structure. As the same feature yields to the same message we achieve this by using **injectiv aggregation functions**.

1.3.1 Mean - Pool

Cannot distinguish different multi-sets with the same embedding proportion.

1.3.2 Max - Pool

Cannot distinguish different sets with the same embeddings.

1.3.3 Injective Multi-Set Function

Any multi-set function can be expressed as:

$$\phi \left(\sum_{x \in X} f(x) \right)$$

According to the **universal approximation theorem**, any function ϕ and f can be modeled with an MLP. As MLPs can model compositions of functions, we end up with:

$$MLP \left(\sum_{x \in X} x \right)$$

Transferring this to the domain of GNNs it yields **GIN(1.2.3) as the most expressive GNN**.

2 GNN Training

2.1 General Information

2.1.1 Training Settings

- Supervised Learning vs. Unsupervised Learning

- What do we want to predict?

  - Node - Level

  - Edge - Level

  - Graph - Level

To make a **k-way prediction** $\hat{y}$ on any level, we need **prediction heads**. K-way prediction means classification with k classes or regression with k targets.

2.1.2 Node - Level

After running the GNN, we have embedding $h_v \in \mathbb{R}^d$ for node v. For node - level prediction we can therefore simply use a matrix transformation:

$$\hat{y}_v = Head_{node}(h_v) = W h_v$$

$$W \in \mathbb{R}^{k \times d}$$

2.1.3 Edge - Level

For edge - level predictions, the prediction head takes **two embeddings** as input. Examples:

- $\hat{y}_{uv} = Linear(Concat(h_u, h_v))$
  not commutative $\rightarrow$ problematic for undirected graphs

- 1-way prediction: $\hat{y}_{uv} = h_u \bullet h_v$

  k-way prediction ($\approx$ Multi-Head Attention):

$$\hat{y}_{uv}^{(i)} = h_u^T W^{(i)} h_v$$

$$\hat{y}_{uv} = Concat(\{\hat{y}_{uv}^{(i)} | 1 \leqslant i \leqslant k\})$$

2.1.4 Graph - Level

For graph - level predictions, the prediction head takes **all embeddings** as input. It therefore basically is very similar to an **aggregation function**.

- **Global Pooling**: similar to GNN aggregation functions (like Mean, Max and Sum) $\rightarrow$ same problems occur.

- **Hierarchical Pooling**: aggregate embeddings inside **clusters** of the graph and repeat until the result is a single embedding.

3 Graph Generation

3.1 Network Properties

3.1.1 Degree Distribution

The degree distribution $P(k)$ is the probability that a random node has degree k. Computation:

$$N_k = \#\text{nodes with degree k}$$

$$P(k) = \frac{N_k}{N}$$

3.1.2 Clustering Coefficient

How connected the neighborhood of node i with degree k_i is. It basically measures, how many edges out of all possible edges exist.

$$C_i = \frac{2e_i}{k_i (k_i - 1)}$$

$$C = \frac{1}{N} \sum_i^N C_i$$

C_i ...   clustering coefficient

e_i ...   total number of edges in neighborhood of node i

3.1.3 Connectivity

Connectivity is the size of the **largest connected component**. This largest component is called the **giant component**. Can be found by i.e. BFS.

3.1.4 Path Length

Diameter

$$max_{u,v} d(u,v)$$

$d(u,v)$ ... length of shortest path between u and v

Average Path Length

$$\overline{h} = \frac{1}{2E_{max}} \sum_{i,j \neq i} h_{ij}$$

E_{max} ...   max number of edges $\left(\frac{n(n-1)}{2} \right)$

h_{ij} ... $d(i,j)$

3.1.5 Expansion

Expansion α is defined as

$$\alpha = \min_{S \subseteq V} \frac{\#\text{edges leaving S}}{min(|S|, |V \setminus S|)}$$

and can be interpreted as the minimum average number of edges that ties a node u of set S to the set $V \setminus S$. This corresponds to a kind of **robustness of the graph**.

3.2 Erdös-Renyi Random Graphs

An Erdös-Renyi Random Graph G_{np} is an undirected graph with **n nodes**. Each **edge (u,v)** has probability p to exist.

3.2.1 Degree Distribution

The degree distribution $P(k)$ for each node is **binomial**. We use $n-1$ instead of n as we don't allow self loops.

$$P(k) = \binom{n-1}{k} p^k (1-p)^{n-1-k}$$

3.2.2 Clustering Coefficient

As the # edge distribution is binomial:

$$\mathbf{E}[e_i] = p \cdot \frac{k_i(k_i - 1)}{2}$$

$$\implies \mathbf{E}[C_i] = \frac{2\mathbf{E}[e_i]}{k_i(k_i - 1)} = \frac{p \cdot k_i(k_i - 1)}{k_i(k_i - 1}} = p = \frac{\overline{k}}{n-1} \approx \frac{\overline{k}}{n}$$

3.3 Small-World Model

Small world models try to achieve **low diameter** while retaining a **high clustering coefficient**.

- start with regular grid $\Rightarrow$ high clustering coefficient

- change target of some edges randomly $\Rightarrow$ low diameter

3.4 Kronecker Graphs

3.4.1 Kronecker Product

Let A and B be matrices.

$$C = A \otimes B = \begin{pmatrix} a_{1,1}B & \cdots & a_{1,m}B \\ \vdots & \vdots & \vdots \\ a_{n,1}B & \cdots & a_{n,m}B \end{pmatrix}$$

For **recursive graph structurs** we apply the Kronecker Product **iteratively**. Different matrices K_1, K_1', ... are possible.

$$K_1^m = K_m = K_{m-1} \otimes K_1$$

3.4.2 Stochastic Kronecker Graphs

Idea: use **probability matrix** and compute the Kronecker Product.

Problem: n^2 probabilities.

Solution: Select m edges randomly based on the probability matrix by **iteratively sampling a pair** column/row (u, v).
When going $1 \leqslant i \leqslant q$ layers deep, the graph has $n = 2^q$ nodes and one can formalize the algorithm as:

Algorithm 1 Generating Stochastic Kronecker Graph Edge

  Θ $\triangleright$ probability matrix
  $x = 0, y = 0$ $\triangleright$ edge location
  for $i \leftarrow 1$ to q **do**
      pick random (u, v) with probability Θ_{uv}
      $x \leftarrow x + u \cdot 2^{q-i}$
      $y \leftarrow y + v \cdot 2^{q-1}$
  end for

3.5 GraphRNN

We want $p_{model}(x|\theta)$ to model the data distribution $p_{data}(x)$. $\rightarrow$ **Maximum Likelihood**

$$\theta^* = \arg\max_{\theta} \mathbf{E}_{x \sim p_{data}} \log p_{model}(x|\theta)$$

GraphRNN consists of two RNNs

- Node - Level RNN

- Edge - Level RNN

which are connected the way as illustrated in fig. 1. We look at each edge level RNN
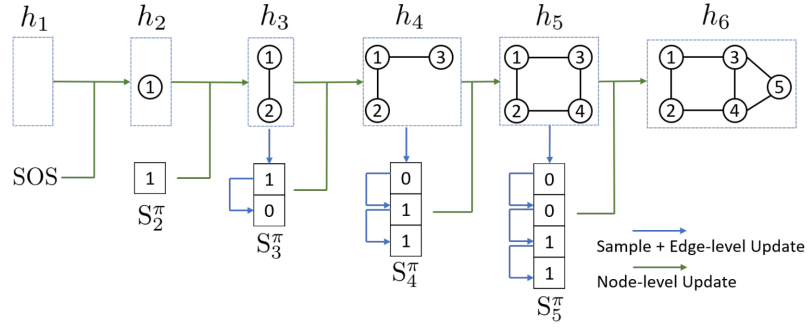Cell's output as **a probability to sample from**.

Figure 1: GraphRNN Schema [1]

It is worth noting that the graph state is updated as:

$$h_i = f_{trans}(h_{i-1}, S_{i-1}^\pi)$$

$$
\begin{array}{lll}
f_{trans} & \ldots & \text{transition module (NN)} \\
S_{i-1}^\pi & \ldots & \text{last entry in adjacency vector of node } \pi(v_{i-1})
\end{array}
$$

Improvement: Only train on $BFS(G, \pi)$.
$\rightarrow$ less BFS orderings than node permutations
$\rightarrow$ edge - level RNN only considers edges to the nodes in the previous level of BFS

References

[1] Jiaxuan You Rex Ying Xiang Ren William L. Hamilton Jure Leskovec. "GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models". In: *arXiv preprint arXiv:1802.08773v3* (2018).