Graph Neural Networks

Script based on CS224W: Winter 2021 [3]

Unity05

December 31, 2021

Contents

1 Message Passing Graph Neural Networks **4**
- 1.1 Introduction . . . 4
  - 1.1.1 Message Computation . . . 4
  - 1.1.2 Aggregation . . . 4
- 1.2 Popular GNN Layers . . . 5
  - 1.2.1 Graph Convolutional Networks (GCN) [2] . . . 5
  - 1.2.2 Graph Attention Networks (GAT) [6] . . . 5
  - 1.2.3 Graph Isomorphism Network (GIN) [7] . . . 6
- 1.3 Expressiveness of GNNs . . . 6
  - 1.3.1 Mean - Pool . . . 6
  - 1.3.2 Max - Pool . . . 6
  - 1.3.3 Injective Multi-Set Function . . . 6

2 GNN Training Pipeline **7**
- 2.1 Preprocessing . . . 7
  - 2.1.1 Graph Augmentation . . . 7
  - 2.1.2 Dataset Split . . . 7
- 2.2 Training . . . 8
  - 2.2.1 Training Settings . . . 8
  - 2.2.2 Node - Level . . . 8
  - 2.2.3 Edge - Level . . . 8
  - 2.2.4 Graph - Level . . . 9

3 Knowledge Graphs **9**
- 3.1 Heterogeneous Graphs . . . 9
- 3.2 Relational GCN (RGCN) [5] . . . 9
  - 3.2.1 Block Diagonal Matrices . . . 10
  - 3.2.2 Basis Learning . . . 10

4 Graph Generation **10**
- 4.1 Network Properties . . . 10
  - 4.1.1 Degree Distribution . . . 10
  - 4.1.2 Clustering Coefficient . . . 10
  - 4.1.3 Connectivity . . . 11
  - 4.1.4 Path Length . . . 11
  - 4.1.5 Expansion . . . 11
- 4.2 Erdös-Renyi Random Graphs . . . 11
  - 4.2.1 Degree Distribution . . . 12
  - 4.2.2 Clustering Coefficient . . . 12
- 4.3 Small-World Model . . . 12

4.4 Kronecker Graphs [4] 12
4.4.1 Kronecker Product 12
4.4.2 Stochastic Kronecker Graphs 13
4.5 GraphRNN [8] 13

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) [2]

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) [6]

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) [7]

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** [1], 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 Pipeline

2.1 Preprocessing

2.1.1 Graph Augmentation

- **Lack Of Features**
  - Constant node features (low expressiveness, high generalization to new nodes).
  - One - hot node features (high expressiveness, no generalization to new nodes).
  - Almost any node property.
- **Sparse/Dense/Large Graph Structure**
  - Sparse: add virtual edges / node
  - Dense: sample neighbors
  - Large: sample subgraphs

2.1.2 Dataset Split

We distinguish between **Fixed Split** and **Random Split**. The problem is that the **components** of a graph are **not independent** from one another.

Transductive Setting
The model can observe the **whole graph** but we **split the labels** we train / validate / test on. → Not applicable for graph - level predictions.

Inductive Setting
We **break** the graph into **independent graphs** we train / validate / test on.

2.2 Training

2.2.1 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.

2.2.2 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:

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

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

2.2.3 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):

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

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

2.2.4 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) → same problems occur.
- **Hierarchical Pooling**: aggregate embeddings inside **clusters** of the graph and repeat until the result is a single embedding.

3 Knowledge Graphs

3.1 Heterogeneous Graphs

Is knowledge graph → is heterogeneous graph. Heterogeneous graph G:

$$G = (V, E, R, T)$$

$v_i \in V$	...	node
$(v_i, r, v_j) \in E$	...	edges
$T(v_i)$	...	node type
$r \in R$	...	relation type

3.2 Relational GCN (RGCN) [5]

Message Computation

$$m_{u,r}^{(l)} = \left(W_r^{(l)} \cdot h_u^{(l-1)} \right)$$

Aggregation

$$h_v^{(l)} = \sigma \left(W_{root}^{(l)} \cdot h_v^{(l-1)} + \sum_{r \in R} \sum_{i \in N_r(v)} \frac{1}{|N_r(v)|} \cdot m_u^{(l)} \right)$$

Problem: # weight matrices in layer l = $|R|$ ⇒ overfitting

3.2.1 Block Diagonal Matrices

3.2.2 Basis Learning

View matrix W_r as transformation matrix for **change of basis**. W_r is a **learnable linear combination** of multiple transformation matrices.

$$W_r = \sum_{b=1}^{B} a_{rb} \cdot V_b$$

V_b	...	transformation matrix
a_{rb}	...	learnable importance weight of V_b

4 Graph Generation

4.1 Network Properties

4.1.1 Degree Distribution

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

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

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

4.1.2 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

4.1.3 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.

4.1.4 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} ... $d(i,j)$

4.1.5 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**.

4.2 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.

4.2.1 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.

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

4.2.2 Clustering Coefficient

As the # edge distribution is binomial:

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

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

4.3 Small-World Model

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

- start with regular grid ⇒ high clustering coefficient
- change target of some edges randomly ⇒ low diameter

4.4 Kronecker Graphs [4]

4.4.1 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.

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

4.4.2 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 ▷ probability matrix
$x = 0, y = 0$ ▷ 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

4.5 GraphRNN [8]

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**.

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

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

f_{trans} ... transition module (NN)
S^{π}_{i-1} ... last entry in adjacency vector of node $\pi(v_{i-1})$

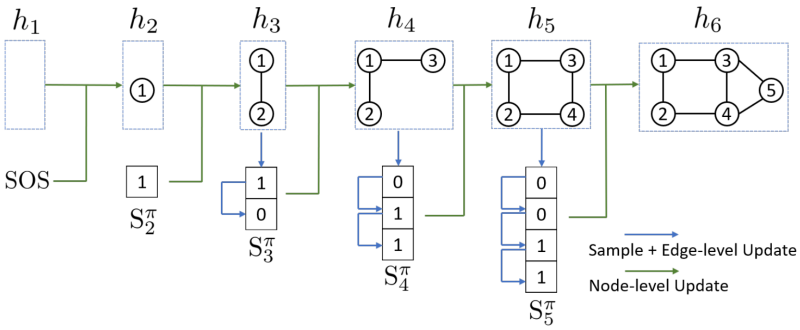

Figure 1: GraphRNN Schema [8]

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

Glossary

Fixed Split We split the dataset into Training Set, Validation Set and Test Set.

Random Split We randomly split the dataset into Training Set, Validation Set and Test Set. We then take the mean performance over multiple random seeds.

References

[1] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[2] Thomas N. Kipf and Max Welling. *Semi-Supervised Classification with Graph Convolutional Networks*. 2017. arXiv: 1609.02907 [cs.LG].

[3] Jure Leskovec. *Stanford CS224W: Machine Learning with Graphs, Winter 2021*. URL: http://snap.stanford.edu/class/cs224w-2020/. Last visited on 2021/12/27.

[4] Jure Leskovec et al. *Kronecker Graphs: An Approach to Modeling Networks*. 2009. arXiv: 0812.4905 [stat.ML].

[5] Michael Schlichtkrull et al. *Modeling Relational Data with Graph Convolutional Networks*. 2017. arXiv: 1703.06103 [stat.ML].

[6] Petar Veličković et al. *Graph Attention Networks*. 2018. arXiv: 1710.10903 [stat.ML].

[7] Keyulu Xu et al. *How Powerful are Graph Neural Networks?* 2019. arXiv: 1810.00826 [cs.LG].

[8] Jiaxuan You et al. *GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models*. 2018. arXiv: 1802.08773 [cs.LG].