A Gentle Introduction to Graph Neural Networks (Basics, DeepWalk, and GraphSage)

黃功詳 Steeve Huang

Feb 10 · 8 min read

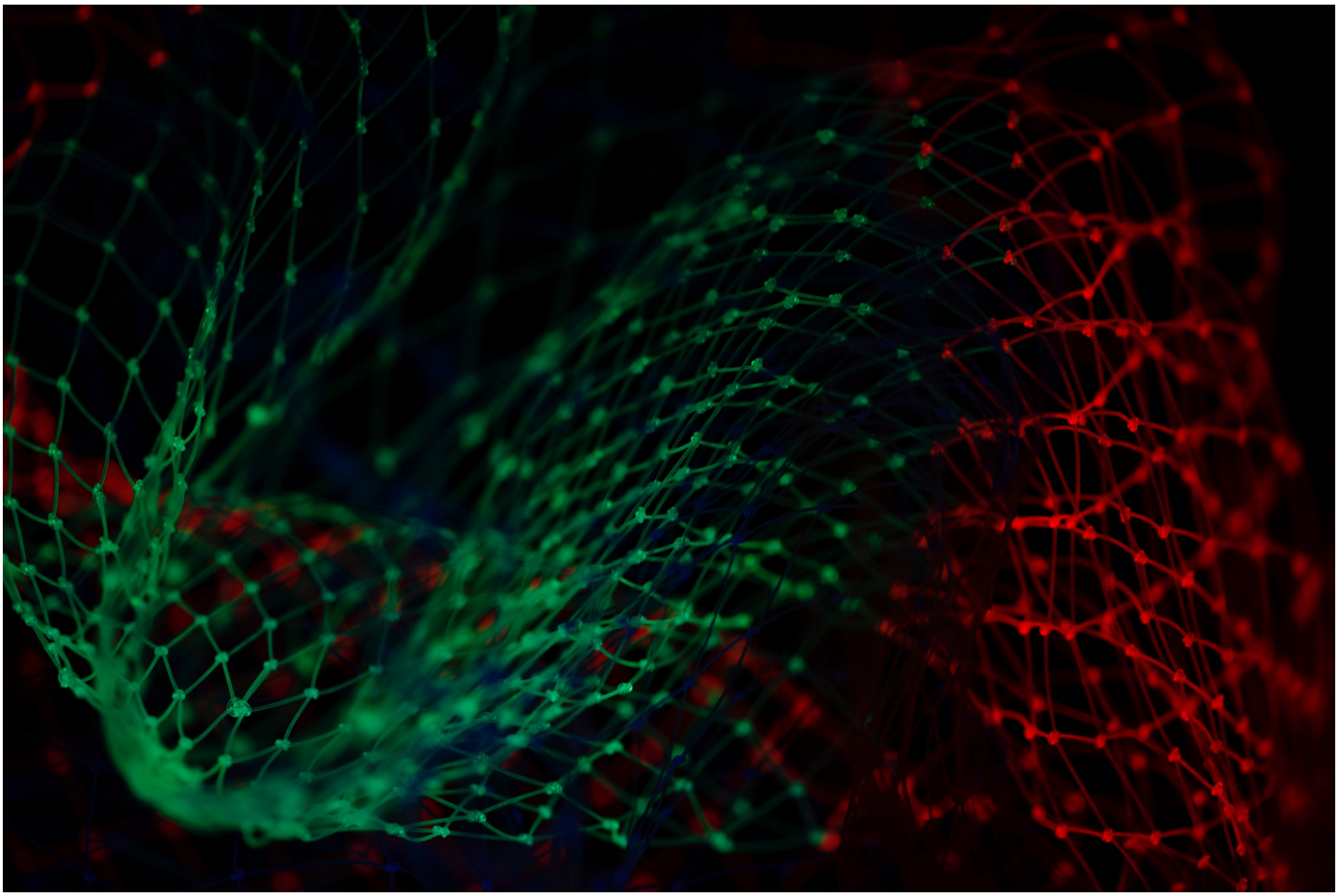


Image from Pexels

Recently, Graph Neural Network (GNN) has gained increasing popularity in various domains, including social network, knowledge graph, recommender system, and even life science. The power of GNN in modeling the dependencies between nodes in a graph enables the breakthrough in the research area related to graph analysis. This

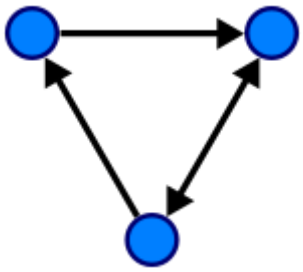
article aims to introduce the basics of Graph Neural Network and two more advanced algorithms, DeepWalk and GraphSage.

Graph

Before we get into GNN, let's first understand what is *Graph*. In Computer Science, a graph is a data structure consisting of two components, vertices and edges. A graph G can be well described by the set of vertices V and edges E it contains.

$$G = (V, E)$$

Edges can be either directed or undirected, depending on whether there exist directional dependencies between vertices.



A Directed Graph (wiki)

The vertices are often called nodes. In this article, these two terms are interchangeable.

Graph Neural Network

Graph Neural Network is a type of Neural Network which directly operates on the Graph structure. A typical application of GNN is node classification. Essentially, every node in the graph is associated with a label, and we want to predict the label of the nodes without ground-truth. This section will illustrate the algorithm described in the paper, the first proposal of GNN and thus often regarded as the original GNN.

In the node classification problem setup, each node v is characterized by its feature x_v and associated with a ground-truth label t_v. Given a partially labeled graph G, the goal is to leverage these labeled nodes to predict the labels of the unlabeled. It learns to represent each node with a d dimensional vector (state) h_v which contains the information of its neighborhood. Specifically,

$$\mathbf{h}_v = f(\mathbf{x}_v, \mathbf{x}_{co[v]}, \mathbf{h}_{ne[v]}, \mathbf{x}_{ne[v]})$$

$$\mathbf{h}_v = f\left(\mathbf{x}_v, \mathbf{x}_{co[v]}, \mathbf{h}_{ne[v]}, \mathbf{x}_{ne[v]}\right)$$

https://arxiv.org/pdf/1812.08434

where *x_co[v]* denotes the features of the edges connecting with *v*, *h_ne[v]* denotes the embedding of the neighboring nodes of *v*, and *x_ne[v]* denotes the features of the neighboring nodes of *v*. The function *f* is the transition function that projects these inputs onto a d-dimensional space. Since we are seeking a unique solution for h_v, we can apply Banach fixed point theorem and rewrite the above equation as an iteratively update process. Such operation is often referred to as **message passing** or **neighborhood aggregation**.

$$\mathbf{H}^{t+1} = F(\mathbf{H}^t, \mathbf{X})$$

https://arxiv.org/pdf/1812.08434

H and X denote the concatenation of all the h and x, respectively.

The output of the GNN is computed by passing the state *h_v* as well as the feature *x_v* to an output function g.

$$\mathbf{o}_v = g(\mathbf{h}_v, \mathbf{x}_v)$$

https://arxiv.org/pdf/1812.08434

Both f and g here can be interpreted as feed-forward fully-connected Neural Networks. The L1 loss can be straightforwardly formulated as the following:

$$loss = \sum_{i=1}^{p} (\mathbf{t}_i - \mathbf{o}_i)$$

https://arxiv.org/pdf/1812.08434

which can be optimized via gradient descent.

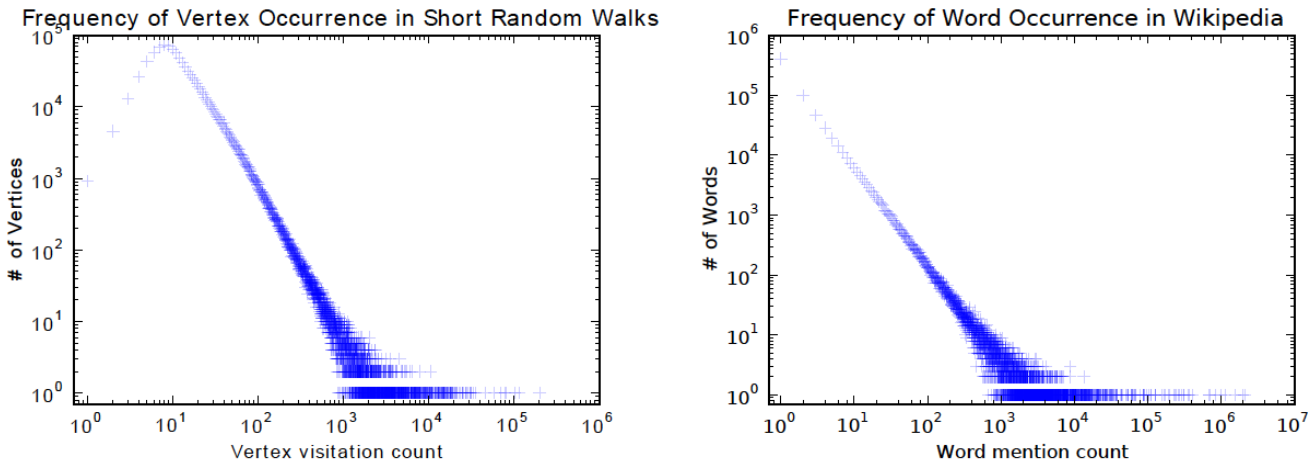
However, there are three main limitations with this original proposal of GNN pointed out by this paper:

1. If the assumption of "fixed point" is relaxed, it is possible to leverage Multi-layer Perceptron to learn a more stable representation, and removing the iterative update process. This is because, in the original proposal, different iterations use the same parameters of the transition function f, while the different parameters in different layers of MLP allow for hierarchical feature extraction.
2. It cannot process edge information (e.g. different edges in a knowledge graph may indicate different relationship between nodes)
3. Fixed point can discourage the diversification of node distribution, and thus may not be suitable for learning to represent nodes.

Several variants of GNN have been proposed to address the above issue. However, they are not covered as they are not the focus in this post.

DeepWalk

DeepWalk is the first algorithm proposing node embedding learned in an unsupervised manner. It highly resembles word embedding in terms of the training process. The motivation is that the distribution of both nodes in a graph and words in a corpus follow a power law as shown in the following figure:



http://www.perozzi.net/publications/14_kdd_deepwalk.pdf

The algorithm contains two steps:

1. Perform random walks on nodes in a graph to generate node sequences

2. Run skip-gram to learn the embedding of each node based on the node sequences generated in step 1

At each time step of the random walk, the next node is sampled uniformly from the neighbor of the previous node. Each sequence is then truncated into sub-sequences of length $2|w| + 1$, where w denotes the window size in skip-gram. If you are not familiar with skip-gram, my previous blog post shall brief you how it works.

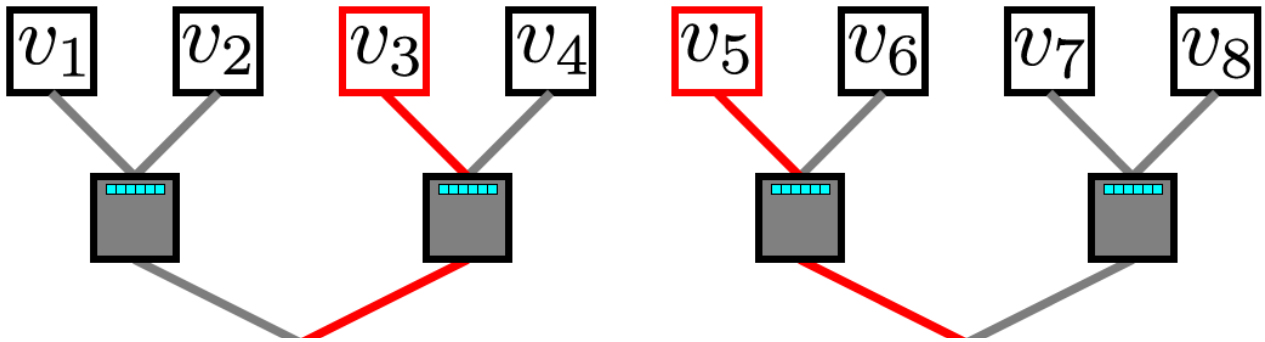
In this paper, hierarchical softmax is applied to address the costly computation of softmax due to the huge number of nodes. To compute the softmax value of each of the individual output element, we must compute all the e^xk for all the element k.

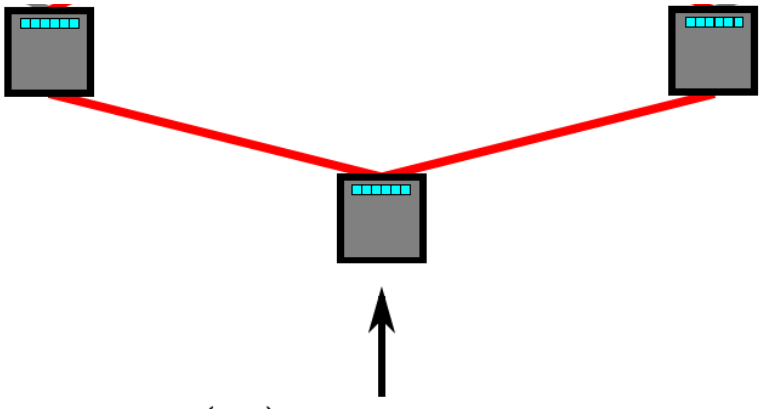
$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}}$$

The definition of Softmax

Therefore, the computation time is $O(|V|)$ for the original softmax, where V denotes the set of vertices in the graph.

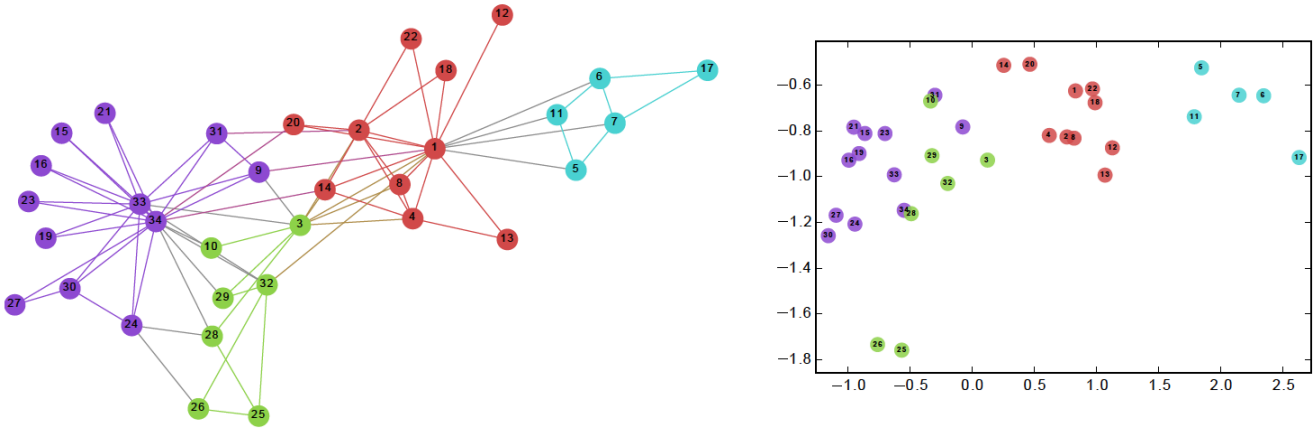
Hierarchical softmax utilizes a binary tree to deal with the problem. In this binary tree, all the leaves (v1, v2, … v8 in the above graph) are the vertices in the graph. In each of the inner node, there is a binary classifier to decide which path to choose. To compute the probability of a given vertex v_k, one simply compute the probability of each of the sub-path along the path from the root node to the leave v_k. Since the probability of each node' children sums to 1, the property that the sum of the probability of all the vertices equals 1 still holds in the hierarchical softmax. The computation time for an element is now reduced to $O(\log|V|)$ as the longest path for a binary tree is bounded by $O(\log(n))$ where n is the number of leaves.





Hierarchical Softmax (http://www.perozzi.net/publications/14_kdd_deepwalk.pdf)

After a DeepWalk GNN is trained, the model has learned a good representation of each node as shown in the following figure. Different colors indicate different labels in the input graph. We can see that in the output graph (embedding with 2 dimensions), nodes having the same labels are clustered together, while most nodes with different labels are separated properly.



(a) Input: Karate Graph (b) Output: Representation

http://www.perozzi.net/publications/14_kdd_deepwalk.pdf

However, the main issue with DeepWalk is that it lacks the ability of generalization. Whenever a new node comes in, it has to re-train the model in order to represent this node (**transductive**). Thus, such GNN is not suitable for dynamic graphs where the nodes in the graphs are ever-changing.

GraphSage

GraphSage provides a solution to address the aforementioned problem, learning the embedding for each node in an **inductive** way. Specifically, each node is represented by the aggregation of its neighborhood. Thus, even if a new node unseen during training

time appears in the graph, it can still be properly represented by its neighboring nodes. Below shows the algorithm of GraphSage.

Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity σ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

1 $\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;
2 **for** $k = 1...K$ **do**
3 **for** $v \in \mathcal{V}$ **do**
4 $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
5 $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
6 **end**
7 $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
8 **end**
9 $\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

https://www-cs-faculty.stanford.edu/people/jure/pubs/graphsage-nips17.pdf

The outer loop indicates the number of update iteration, while *h^k_v* denotes the latent vector of node *v* at update iteration *k*. At each update iteration, *h^k_v* is updated based on an aggregation function, the latent vectors of *v* and *v*'s neighborhood in the previous iteration, and a weight matrix *W^k*. The paper proposed three aggregation function:

1. Mean aggregator:

The mean aggregator takes the average of the latent vectors of a node and all its neighborhood.

$$\mathbf{h}_v^k \leftarrow \sigma(\mathbf{W} \cdot \text{MEAN}(\{\mathbf{h}_v^{k-1}\} \cup \{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$$

https://www-cs-faculty.stanford.edu/people/jure/pubs/graphsage-nips17.pdf

Compared with the original equation, it removes the concatenation operation at line 5 in the above pseudo code. This operation can be viewed as a "skip-connection", which later in the paper proved to largely improve the performance of the model.

2. LSTM aggregator:

Since the nodes in the graph don't have any order, they assign the order randomly by permuting these nodes.

3. Pooling aggregator:

This operator performs an element-wise pooling function on the neighboring set. Below shows an example of max-pooling:

$$\text{AGGREGATE}_k^{\text{pool}} = \max(\{\sigma\left(\mathbf{W}_{\text{pool}}\mathbf{h}_{u_i}^k + \mathbf{b}\right), \forall u_i \in \mathcal{N}(v)\})$$

https://www-cs-faculty.stanford.edu/people/jure/pubs/graphsage-nips17.pdf

, which can be replaced with mean-pooling or any other symmetric pooling function. It points out that pooling aggregator performs the best, while mean-pooling and max-pooling aggregator have similar performance. The paper uses max-pooling as the default aggregation function.

The loss function is defined as the following:

$$J_{\mathcal{G}}(\mathbf{z}_u) = -\log\left(\sigma(\mathbf{z}_u^\top \mathbf{z}_v)\right) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \log\left(\sigma(-\mathbf{z}_u^\top \mathbf{z}_{v_n})\right)$$

https://www-cs-faculty.stanford.edu/people/jure/pubs/graphsage-nips17.pdf

where *u* and *v* co-occur in a fixed-length random walk, while *v_n* are the negative samples that don't co-occur with *u*. Such loss function encourages nodes closer to have similar embedding, while those far apart to be separated in the projected space. Via this approach, the nodes will gain more and more information about their neighborhoods.

GraphSage enables representable embedding to be generated for unseen nodes by aggregating its nearby nodes. It allows node embedding to be applied to domains involving dynamic graph, where the structure of the graph is ever-changing. Pinterest, for example, has adopted an extended version of GraphSage, PinSage, as the core of their content discovery system.

Conclusion

You have learned the basics of Graph Neural Networks, DeepWalk, and GraphSage. The power of GNN in modeling complex graph structures is truly astonishing. In view of its effectiveness, I believe, in the near future, GNN will play an important role in AI's development. If you like my article, don't forget to follow me on Medium and Twitter, where I frequently shared the most advanced development of AI, ML, and DL.

Machine Learning | Deep Learning | Artificial Intelligence | Data Science | Programming