Chapter 8
Unsupervised Learning: Graph Vector

Graph data involves rich and complex potential relationships and plays an important role in many real-world applications, being used extensively in areas such as social networks, recommendation systems, science, and NLP. As AI continues to gain popularity, a growing number of machine learning tasks need to analyze and process graph data. One effective method for graph analysis is to map a graph's elements to a low-dimensional vector space while retaining the graph's structure and property information. This low-dimensional vector is called a graph vector (or "graph embedding"), which is described below.

8.1 Graph Vector Overview

A graph is a data structure comprising a set of vertices, which are interconnected by lines called edges, and relationships between the vertices, as shown in Fig. 8.1. The graph is called a directed graph if each edge has a direction (in this case, the edges are similarly known as directed edges). Conversely, the graph is called an undirected graph if the edges have no direction. Two graphs are isomorphic if they have the same number of vertices and edges and if the second graph can be obtained by permuting all vertices in the first graph one by one to the names of the vertices in the second graph. For example, a pentagon with five vertices and a five-pointed star with five vertices are considered isomorphic. The number of edges associated with the vertex represents the degree of a vertex. A common storage representation of a graph is the adjacency matrix, which can be represented by the vertex set V and the edge matrix E, as shown in Formula (8.1).

L. Chen, *Deep Learning and Practice with MindSpore*, Cognitive Intelligence and Robotics, https://doi.org/10.1007/978-981-16-2233-5_8

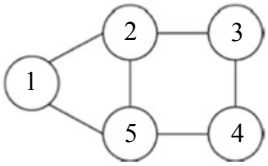


Fig. 8.1 Common storage representation of a graph

$$\boldsymbol{V} = (v_1, v_2, v_3, v_4, v_5)\boldsymbol{E} = \begin{bmatrix} 0\,1\,0\,0\,1 \\ 1\,0\,1\,0\,1 \\ 0\,1\,0\,1\,0 \\ 0\,0\,1\,0\,1 \\ 1\,1\,0\,1\,0 \end{bmatrix} \tag{8.1}$$

Graphs are used in all sorts of real-world applications (e.g., in communication networks, social networks, e-commerce networks, and traffic networks). Because they contain rich information, comprising potentially billions of vertices and relationships between the vertices (edges), graph analysis is of particular importance.

However, these graphs are usually high-dimensional ones that contain massive volumes of information, making it difficult to directly process them. An important method for analyzing and processing such graphs is the graph embedding method (GEM), which uses a low dimension, dense vector to represent a graph's vertex and reflect its structure information. The following key features [1] are paramount in a good GEM [1].

1. Neighborhood awareness: The distance between hidden vectors on vertices reflects the distance between the vertices on the graph.
2. Low dimension: This feature is necessary to facilitate subsequent calculations.
3. Adaptation: Adding a vertex (or edge) should not cause all calculation processes to be repeated.
4. Continuity: Continuous representations have smooth decision boundaries and enable refined representations of the graph members.

Depending on the application scenario, the GEM can be divided into vertex embedding, edge embedding, mixed embedding, and whole graph embedding. In the first category, vertex embedding, algorithms such as the classical DeepWalk and Node2Vec, and graph-based neural network ones such as graph convolutional networks (GCNs) and graph attention networks (GATs), are used.

The classical GEMs have two disadvantages: First, during the learning embedding process, parameters are not shared between vertices, and calculation efficiency is low. Second, because learning is directly performed on a particular structure graph, there is a lack of generalization ability, and new or dynamic graphs cannot be processed. Although CNNs are well known for processing Euclidean data, non-Euclidean data is difficult to process. CNNs and GEMs have promoted the development of the graph neural network (GNN) model, which captures the dependence of graphs through message transfer between vertices of graphs. Despite the original GNN being difficult to train and offering suboptimal results, researchers have made significant improvements in its network architecture, optimization methods, and parallel computing, enabling it to achieve good learning capabilities. Over the last few years, GNN has become a popular graph analysis method [2] due to advantages such as excellent performance and high interpretability. In addition, the algorithms represented by GCNs and graph attention networks are gaining significant attention.

This chapter explores the vertex embedding algorithm, with each section centering on the following topics:

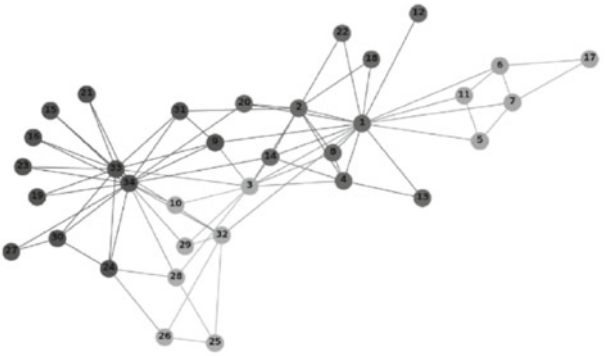
1. Section 8.2 focuses on the classical graph embedding method DeepWalk.
2. Section 8.3 examines the classical graph embedding method Large-scale Information Network Embedding (LINE).
3. Section 8.4 discusses the classical graph embedding method Node2Vec.
4. Sections 8.5 and 8.6 cover the algorithms based on graph neural networks, including GCN and GAT.
5. Section 8.7 delves into the application of graph neural networks in the recommendation system.

8.2 DeepWalk Algorithm

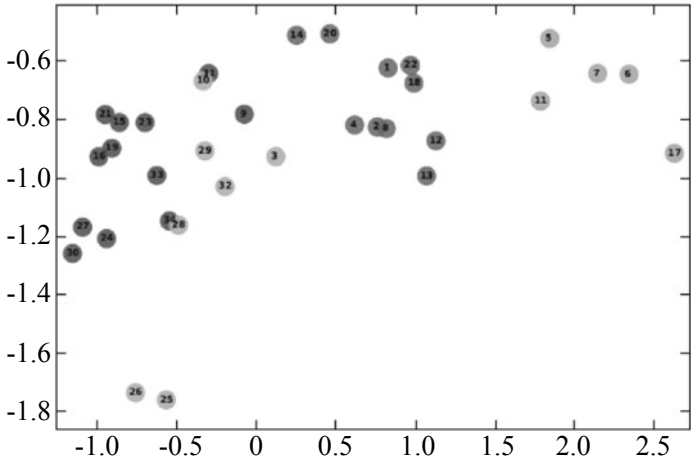
The sparsity of graph representation data (such as the adjacency matrix) makes it a challenging task to design algorithms. We need to eliminate the adverse impacts of data sparsity during network application (such as network classification, recommendation, and anomaly detection) in order to develop high-quality machine learning algorithms. Establishing a method to map the complex and high-dimensional sparse graph data to the low-dimensional dense vector is therefore of the utmost importance. Because machine learning cannot directly deal with natural languages, we must convert words into vectors composed of numeric values so that we can subsequently establish models for analysis. In the field of NLP, one of the most prominent algorithms is Word2Vec, which was inspirational in Bryan Perozzi's proposal of the DeepWalk algorithm [1] in 2014. DeepWalk, a classical unsupervised learning algorithm in graph embedding, performs well in the absence of information and is readily usable by statistical models.

8.2.1 Principles of the DeepWalk Algorithm

The DeepWalk algorithm learns the low-dimensional vector representation of a vertex in a graph by truncating the local information of the random walk (which is often used as a similarity measure in content recommendation and community discovery). In the field of natural language, we can consider the vertex as a word, and the sequence of the vertices obtained through the random walk is like a sentence. The input of the DeepWalk algorithm is a connected graph (either directed or undirected), and the output is a vector representation of all vertices in the graph. Figure 8.2 provides an example of low-dimensional vector representation of the DeepWalk learning vertex, where (a) shows that the input is a graph and (b) shows that the output is a two-dimensional vector representation of each vertex in the input graph.



(a) Input (graph)



(b) Output (vector representation)

Fig. 8.2 Example of low-dimensional vector representation of the DeepWalk learning vertex

The vector dimension is determined to be 2 because the two-dimensional vector is easy to visualize. In the figure, the vertices shown in the same color are similar to each other. The more vertices that two vertices have in common, the shorter the distance between the two-dimensional vectors corresponding to the two vertices.

In order to perform model learning using the natural language modeling algorithm, datasets are required, which are the corpus of several sentences and the vocabulary of several words. Conversely, in the DeepWalk algorithm, the corpus is a set of random walk vertex sequences with limited length, and the vocabulary is the vertex of the graph.

The DeepWalk algorithm is divided into two parts—the input (graph) and the output (vector representation)—as shown in the following figure.

1. **Generating a vertex sequence through random walk**

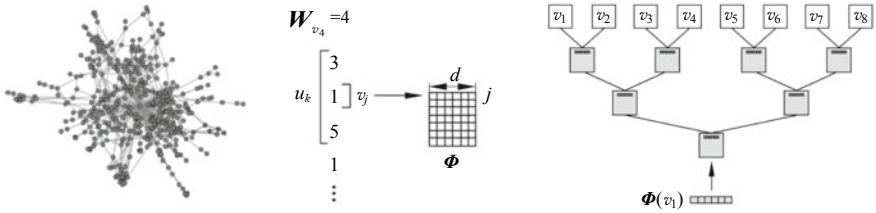
We define a random walk $\boldsymbol{W}_{v_i}$ with vertex v_i as its root vertex. For graph G, we first perform an even, random sampling on a vertex v_i, which is the root vertex of the random walk $\boldsymbol{W}_{v_i}$. We then perform uniform sampling on the neighbors of the currently sampled vertex until the number of vertices in the random walk reaches the maximum length t. Note that the lengths of vertex sequences in the random walk can be different. Random walk not only captures community information, but also has the following two advantages:

1. Parallel local exploration for the vertex is easy to implement.
2. Global recalculation is not required when minor changes occur locally, thereby facilitating online learning.

2. **Skip-Gram**

Skip-Gram is a Word2Vec algorithm in NLP [3] and can learn the random walk $\boldsymbol{W}_{v_i}$ to obtain a vector representation. Given a keyword, Skip-Gram calculates the probability of maximizing the occurrence of surrounding words; that is, it predicts the context. This is explained in more detail in Sect. 7.1. Skip-Gram traverses all possible collocations appearing in the random walk window w. For each collocation, the occurrence probability of neighbor vertices is maximized by each vertex v_i and its representation vector $\boldsymbol{\Phi}(v_j) \in \mathbb{R}^d$. The label dimension is equal to the number |V| of vertices (similar to the one-hot vector), and the number of vertices is generally large. Using Softmax to calculate this probability directly would consume a large amount of computing resources for learning, so instead we can use the hierarchical Softmax [4, 5], which approximates the probability to accelerate the training. Hierarchical Softmax takes the prediction problem and, by assigning vertices to leaf nodes of a binary tree, converts it into maximizing the probability of a path. If we assume that the path to the vertex u_k is a sequence $(b_0, b_1, \ldots, b_{\lceil \log|V| \rceil})$ regarding the tree node, we can obtain the following:

$$P(u_k|\boldsymbol{\Phi}(v_j)) = \prod_{l=1}^{\lceil \log|V| \rceil} P(b_l|\boldsymbol{\Phi}(v_j)) \tag{8.2}$$



(a) Random walk (b) Representation mapping (c) Hierarchical Softmax

Fig. 8.3 Process of the DeepWalk algorithm

where

$P(u_k|\boldsymbol{\Phi}(v_j))$ is the probability that the vertex u_k is the context of the vertex v_j;
$\boldsymbol{\Phi}(v_j)$ is the vector representation of the vertex v_j; and
$P(b_l|\boldsymbol{\Phi}(v_j))$ is the probability that the l^{th} node in the path of the vertex u_k is selected along the binary tree starting from the vertex v_j.

For $P(u_k|\boldsymbol{\Phi}(v_j))$, the calculation of time complexity decreases from $O(|V|)$ to $O(\log|V|)$.

Figure 8.3 and the subsequent description provide details about the process used in the DeepWalk algorithm.

1. In Fig. 8.3a, a random walk sequence $\boldsymbol{W}_{v_4}$ with v_4 as the root vertex is obtained.
2. In Fig. 8.3b, a sample is generated on a sequence $\boldsymbol{W}_{v_4}$ by continuously sliding the window (with a length of $2w + 1$). If we assume that the vertex in the window is [1, 3, 5] and the sample is {(1, 3), (1, 5)}, we can conclude that the center vertex v_1 is mapped to its vector representation $\boldsymbol{\Phi}(v_1)$.
3. In Fig. 8.3c, Hierarchical Softmax decomposes $P(v_3|\boldsymbol{\Phi}(v_1))$ and $P(v_5|\boldsymbol{\Phi}(v_1))$ into probability distribution that corresponds to the path from the root to v_3 and v_5. It maximizes the two probabilities by updating $\boldsymbol{\Phi}$, which is the vertex representation matrix that needs to be calculated.

8.2.2 Implementation of the DeepWalk Algorithm

This section builds on the theoretical information provided earlier by outlining the pseudocode necessary to implement DeepWalk and Skip-Gram.

Algorithm 8.1 Pseudocode for Implementing DeepWalk

Input: Graph $G(V, \boldsymbol{E})$, window size w, embedding size d, number γ of times that each vertex is walked, and walk length t

Output: Vertex representation matrix $\boldsymbol{\Phi} \in \square^{|V| \times d}$

(1) Initialization: Sample Φ from $\boldsymbol{U}^{|V| \times d}$

(2) Establish a binary tree T from V

(3) For each iteration $i = 0$ to γ, execute:

(4) Disorder the vertex sequence $O = \text{Shuffle}(V)$

(5) For each vertex $v_i \in O$, execute:

(6) Generate random walk with v_i as the root vertex, where $\boldsymbol{W}_{v_i} = \text{RandomWalk}(G, v_i, t)$

(7) Learn embedding by using Skip-Gram $(\boldsymbol{\Phi}, \boldsymbol{W}_{v_i}, w)$

(8) End

Algorithm 8.2 Pseudocode for Implementing Skip-Gram

Input: Vertex representation matrix $\boldsymbol{\Phi}$, random walk sequence $\boldsymbol{W}_{v_i}$, and window size w

Output: New vertex representation matrix $\boldsymbol{\Phi}$

(1) For each vertex $v_j \in \boldsymbol{W}_{v_i}$, execute:

(2) For each vertex in the window $u_k \in \boldsymbol{W}_{v_i}[j - w: j + w]$, execute:

(3) Calculate $J(\boldsymbol{\Phi}) = -\log P(u_k | \boldsymbol{\Phi}(v_j))$

(4) Update vector representation $\boldsymbol{\Phi} = \boldsymbol{\Phi} - \alpha * \frac{\partial J}{\partial \boldsymbol{\Phi}}$

(5) End

(6) End

8.3 LINE Algorithm

DeepWalk performs well on many datasets because it is a graph embedding method based on random walk. However, the DeepWalk algorithm considers the similarities between points based on only the explicit connections between such points (e.g., points 6 and 7 in Fig. 8.4); it ignores the possibility that similarities may exist between points that are not connected in the information network. In Fig. 8.4, for example, there is no direct connection between points 5 and 6. But there are similarities between them, because they share points 1, 2, 3, and 4.

We can use an analogy here: If two people have many mutual friends, we can assume that those two people probably have common hobbies and habits. By carefully

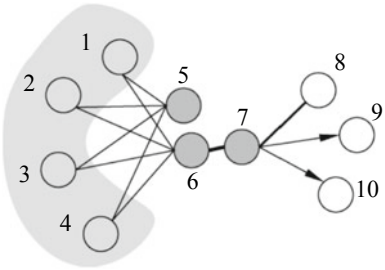


Fig. 8.4 Information network[1]

designing the loss function and considering the similarities between points 6 and 7 as well as those between points 5 and 6, we can ensure that the vector representation obtained by the LINE algorithm [6] retains information about both the local and global network architectures.

The LINE algorithm has strong universality and can be used for both directed and undirected graphs. Furthermore, it can be used for both weighted and unweighted graphs. The following sections provide a brief overview of the LINE algorithm and pseudocode for its implementation.

8.3.1 Principles of the LINE Algorithm

Directly connected points always exhibit similarities between them. If there is a direct connection between two vertices, the weight w_{ij} of the edge connecting the two vertices represents a first-order similarity, which is a direct similarity between the pairs of vertices. Conversely, if no direct connection exists between two vertices, the first-order similarity is 0. Figure 8.4 shows a first-order similarity, between points 6 and 7.

The first-order similarity applies only to undirected graphs. For each undirected edge (i, j) in an information network, the joint probability between the vertices v_i and v_j is defined as follows:

$$p_1(v_i, v_j) = \frac{1}{1 + \exp(-\boldsymbol{\Phi}(v_i)^{\mathrm{T}} \bullet \boldsymbol{\Phi}(v_j))} \tag{8.3}$$

where

$p_1(v_i, v_j)$ is the joint probability between the vertices v_i and v_j;
$\boldsymbol{\Phi}(v_i)$ is the low-dimensional vector representation of the vertex v_i, and $\boldsymbol{\Phi}(v_i) \in \mathbb{R}^d$.

[1] Source: http://www.www2015.it/documents/proceedings/proceedings/p1067.pdf.

This formula defines probability distribution $p\ (\bullet, \bullet)$ in $|V| \times |V|$, where V denotes the number of vertices. $\hat{p}_1 = \frac{w_{ij}}{W}$ is the empirical probability, where $W = \sum_{(i,j)\in E} w_{ij}$. To preserve the first-order similarity, we can use Kullback–Leibler (KL) divergence to define the following objective function:

$$D_{\mathrm{KL}}\big(\hat{p}_1(\bullet, \bullet), p_1(\bullet, \bullet)\big) = \sum_{(i,j)\in E} \hat{p}_1(v_i, v_j)(\log \hat{p}_1(v_i, v_j) - \log p_1(v_i, v_j)) \quad (8.4)$$

where

$D_{\mathrm{KL}}(\hat{p}_1\ (\bullet, \bullet), p_1(\bullet, \bullet))$ is the KL divergence of the empirical joint probability distribution and the ground truth joint probability distribution;
$\hat{p}_1\ (\bullet, \bullet)$ is the empirical joint probability distribution between vertices; and
$p_1\ (\bullet, \bullet)$ is the ground truth joint probability distribution between vertices.

After removing constant terms from Formula (8.4), we can obtain the first-order similarity objective function:

$$\mathrm{O}_1 = -\sum_{(i,j)\in E} w_{ij} \log p_1(v_i, v_j) \quad (8.5)$$

where

O_1 is the first-order similarity objective function of the LINE algorithm;
w_{ij} is the edge weight between the vertices v_i and v_j; and
$p_1(v_i, v_j)$ is the ground truth joint probability between the vertices v_i and v_j.

As mentioned earlier, the first-order similarity represents the similarity between the points that have direct connections. However, the information network may contain many other points that have no direct connections. For such cases, the second-order similarity is defined to cover the similarity between the neighbor network architectures of the vertices u and v. If we use $\boldsymbol{p}_u = \big(w_{u,1}, w_{u,2}, \ldots, w_{u,|V|}\big)$ to indicate the first-order similarity between the vertex u and all other vertices, we can conclude that the similarity between $\boldsymbol{p}_u$ and $\boldsymbol{p}_v$ is the second-order similarity between the vertices u and v. Similar to the first-order similarity, the second-order similarity between the vertices u and v is 0 if no vertex is connected to both u and v. Figure 8.4 shows a second-order similarity, between points 5 and 6.

For the second-order similarity, each vertex exists not only as itself but also as the context of other vertices, meaning that two additional vectors are required: $\boldsymbol{\Phi}(v_i)$ and $\boldsymbol{\Phi}(v_i)'$. $\boldsymbol{\Phi}(v_i)$ is the vector representation of the vertex v_i when it is regarded as itself, and $\boldsymbol{\Phi}(v_i)'$ is the vector representation of the vertex v_i when it is regarded as the context of other vertices. The second-order similarity can be used for both directed graphs and undirected graphs. In the information network, one undirected edge can be regarded as two directed edges, so for any directed edge (i, j), the probability that the vertex $v_j$becomes the context of v_i is defined as:

$$p_2(v_j|v_i) = \frac{\exp(\boldsymbol{\Phi}(v_j)'^{\mathrm{T}} \bullet \boldsymbol{\Phi}(v_i))}{\sum_{k=1}^{|V|} \exp(\boldsymbol{\Phi}(v_k)'^{\mathrm{T}} \bullet \boldsymbol{\Phi}(v_i))} \tag{8.6}$$

where

$p_2(v_j \mid v_i)$ is the probability that the vertex v_j becomes the context of v_i;
$\boldsymbol{\Phi}(v_i)'^{\mathrm{T}}$ is the low-dimensional vector representation of v_j as the context; and.
$|V|$ is the number of vertices in the information network.

For each vertex v_i, Formula (8.6) defines conditional distribution $p_2(\bullet \mid v_i)$. Its empirical distribution $\hat{p}_2(\bullet|v_i)$ is defined as $\hat{p}_2\left(v_j|v_i\right) = \frac{w_{ij}}{d_i}$, where $d_i = \sum_{k\in N(i)} w_{ik}$ (which is the out-degree of the vertex v_i). To preserve the second-order similarity, we can use KL divergence to define the following objective function:

$$D_{\mathrm{KL}}\left(\hat{p}_1(\bullet,\bullet), p_1(\bullet,\bullet)\right) = \sum_{(i,j)\in E} d_i \hat{p}_2(v_j|v_i)(\log \hat{p}_2(v_j|v_i) - \log p_2(v_j|v_i)) \tag{8.7}$$

where

$D_{\mathrm{KL}}\left(\hat{p}_1(\bullet\ ,\ \bullet), p_1(\bullet\ ,\ \bullet)\right)$ is the KL divergence of the empirical joint probability distribution and the ground truth probability distribution between the vertices used as contexts;
$\hat{p}_2\ (v_j \mid v_i)$ is the empirical probability that the vertex v_j becomes the context of the vertex v_i; and.
d_i is the out-degree of the vertex.

After removing constant terms from Formula (8.7), we can obtain the second-order similarity objective function:

$$\mathrm{O}_2 = -\sum_{(i,j)\in E} w_{ij} \log p_2(v_j|v_i) \tag{8.8}$$

To preserve both the first- and second-order similarities, the LINE algorithm minimizes O_1 and O_2 and then concatenates the low-dimensional vectors obtained based on O_1 and O_2. This makes it possible to obtain the low-dimensional vector representation $\boldsymbol{\Phi}(v_i)$ of each vertex v_i.

If O_2 is minimized directly, we must calculate the sum of all vertices when calculating the conditional distribution $p_2(\bullet \mid v_i)$, resulting in the time complexity of minimizing O_2 reaching $O(|V|^2)$. Here, the objective function that defines negative sampling becomes:

$$\log\sigma(\boldsymbol{\Phi}(v_j)'^{\mathrm{T}} \bullet \boldsymbol{\Phi}(v_i)) + \sum_{n=1}^{K} E_{n\sim P_{\mathrm{n}}(v)}[\log\sigma(-\boldsymbol{\Phi}(v_n)'^{\mathrm{T}} \bullet \boldsymbol{\Phi}(v_i))] \tag{8.9}$$

where

σ is the sigmoid function, and $\sigma(x) = \frac{1}{1+\exp(x)}$;
K is the number of negative samples in each data sampling; and
$P_n(v) \propto d_v^{3/4}$.

If we replace $\log p_2(v_j|\, v_i)$ with the objective function for negative sampling, the objective function of the second-order similarity becomes:

$$O_2 = -\sum_{(i,j)\in E} w_{ij} \left\{ \log\sigma(\boldsymbol{\Phi}(v_j)'^{\mathrm{T}} \bullet \boldsymbol{\Phi}(v_i)) + \sum_{n=1}^{K} E_{v_n j P_n(v)}[\log\sigma(-\boldsymbol{\Phi}(v_n)'^{\mathrm{T}} \bullet \boldsymbol{\Phi}(v_i))] \right\} \tag{8.10}$$

In addition, when O_1 is minimized directly, $u_{ik} = \infty$, where $i = 1, 2, \ldots, |V|$; and $k = 1, 2, \ldots, d$. To avoid $u_{ik} = \infty$, we need to change the objective function by performing negative sampling:

$$O_1 = -\sum_{(i,j)\in E} w_{ij} \left\{ \log\sigma(\boldsymbol{\Phi}(v_j)^T \bullet \boldsymbol{\Phi}(v_i)) + \sum_{n=1}^{K} E_{v_n \sim P_n(v)}[\log\sigma(-\boldsymbol{\Phi}(v_n)^T \bullet \boldsymbol{\Phi}(v_i))] \right\} \tag{8.11}$$

Regardless of whether O_1 or O_2 is minimized, the objective function includes w_{ij}, which appears in the gradient when we use the gradient descent method for minimization. For different edges, w_{ij} may vary significantly, making it difficult to select an appropriate learning rate. If we select a higher learning rate, gradient explosion may occur on the edge with a larger w_{ij}. Conversely, if we select a lower learning rate, gradient disappearance may occur on the edge with a smaller w_{ij}. To overcome this conundrum, we therefore need to perform edge sampling for optimization, by using the Alias method to sample the original weighted edges. The probability of each sampled edge is proportional to the weight of the edge in the original graph, and the sampled edge weight is used as a binary edge (the weight is 0 or 1). This solves the problem of w_{ij} differing for different edges.

8.3.2 Implementation of the LINE Algorithm

This section builds on the theoretical information provided earlier by outlining the pseudocode necessary to implement the LINE algorithm.

Algorithm 8.3 Pseudocode for Implementing the LINE Algorithm

Input: Graph $G(V, E)$, embedded dimension d, negative sample number K, and initial learning rate lr

Output: Dimension is $\Phi^{|V|\times d}$ low-dimensional dense vector representation

(1) Minimize the first-order similarity loss function O_1:

(2) Initialize the representation vector $\boldsymbol{\Phi}(v_i)$, $i = 1, 2, \ldots, |V|$, where $\boldsymbol{\Phi}(v_i) \in R^d$

(3) Establish an edge sampling table based on graph $G(V, \boldsymbol{E})$ by using the Alias method

(4) Establish a negative sample table based on negative sample number K and graph $G(V, \boldsymbol{E})$

(5) When $(i, j) \in \boldsymbol{E}$, cyclically execute:

(6) Perform edge sampling based on the edge sampling table

(7) Perform negative sampling based on the negative sampling table

(8) End of cyclic execution

(9) Optimize O_1 by using the asynchronous gradient descent

(10) Minimize the second-order similarity loss function O_2

(11) Initialize the representation vectors $\boldsymbol{\Phi}(v_i)$ and $\boldsymbol{\Phi}(v_i)'$, where $i = 1, 2, \ldots, |V|$, $\boldsymbol{\Phi}(v_i) \in R^d$, and $\boldsymbol{\Phi}(v_i)' \in R^d$

(12) Repeat steps (2) to (8)

(13) Optimize O_2 by using the asynchronous gradient descent

(14) Concatenate $\boldsymbol{\Phi}(v_i)$ obtained based on O_1 and $\boldsymbol{\Phi}(v_i)$ obtained based on O_2 to obtain the low-dimensional vector representation of the graph

8.4 Node2Vec Algorithm

The DeepWalk and LINE algorithms described in Sects. 8.2 and 8.3, respectively, depend on the strict concept of network neighborhood and both are insensitive to the network-specific connection mode. The DeepWalk algorithm samples the vertex neighborhood by using depth-first search (DFS) random walk, whereas the LINE algorithm samples vertices by using breadth-first search (BFS). As shown in Fig. 8.5, the Node2Vec algorithm—a graph embedding method and an extension of the DeepWalk algorithm—integrates both DFS and BFS. In Fig. 8.5, the neighborhood for DFS is composed of vertices sampled in ascending order of distance to the source vertex u (e.g., s_4, s_5, and s_6), whereas that for BFS is limited to the vertices adjacent to the source vertex u (e.g., s_1, s_2, and s_3).

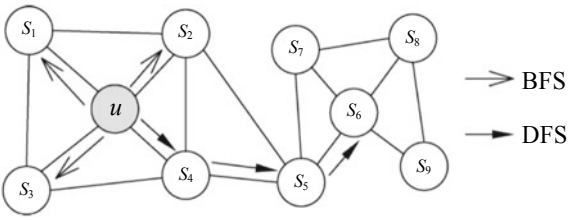


Fig. 8.5 DFS and BFS policies from source vertex u[2]

8.4.1 Principles of the Node2Vec Algorithm

The Node2Vec algorithm, proposed by Aditya Grover in 2016 [7], is used to learn the continuous vector representations of a graph's vertices. Compared with both the DeepWalk and LINE algorithms described earlier, Node2Vec can effectively explore different neighborhoods (homogeneity and structural equivalency) by designing a biased random walk process for vertices, allowing it to learn more comprehensive representations of the vertices. The Node2Vec algorithm functions in a similar way to the DeepWalk algorithm and can be divided into two processes: biased random walk and learning vector representations.

1. **Biased random walk**

The biased random walk process is implemented by assigning different sampling probabilities to different vertices. If we assume that the source vertex is u, the random walk length is l, the *ith* vertex is c_i, and the start vertex is $c_0 = u$, we can use the following formula to calculate the sampling probability of the vertex c_i:

$$P(c_i = x | c_{i-1} = v) = \begin{cases} \frac{\pi_{vx}}{Z}, & (v, x) \in \boldsymbol{E} \\ 0, & \text{else} \end{cases} \tag{8.12}$$

where

π_{vx} is the unnormalized transition probability between the vertex v and the vertex x; and
Z is a normalization constant.

To implement the biased random walk process, Node2Vec introduces p and q—two parameters that are used in calculating the transition probability in Formula (8.12). If we assume that the walk is performed through edge (t, v) to the vertex v, we can calculate the transition probability of the edge (t, v) on the basis of v, allowing us to determine the next vertex for the walk. We use the following unnormalized transition probability:

[2] Source: https://cs.stanford.edu/~jure/pubs/node2vec-kdd16.pdf.

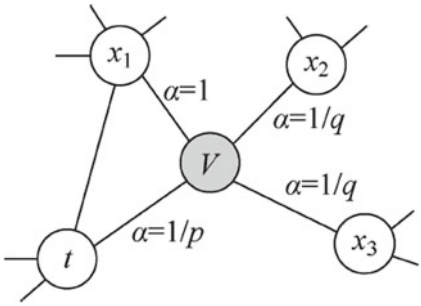


Fig. 8.6 Example of random walk

$$\pi_{vx} = \alpha_{pq}(t, x) \bullet w_{vx}\alpha_{pq}(t, x) = \begin{cases} \frac{1}{p} & d_{tx} = 0 \\ 1 & d_{tx} = 1 \\ \frac{1}{q} & d_{tx} = 2 \end{cases} \tag{8.13}$$

where d_{tx} is the shortest path distance from the vertex t to the vertex v.

d_{tx} must be one of {0, 1, 2} so that the two parameters p and q can adequately control the walk process, as shown in Fig. 8.6. The return parameter p controls the probability of subsequently accessing the vertex in the previous step. If the current vertex is v, and p is greater than max(q, 1), the probability of accessing the vertex t in the previous step will decrease. Conversely, the probability will increase if p is less than min(q, 1). The "in–out" parameter q controls whether the walk process is more like BFS or DFS. A larger q indicates that the vertex for the random walk is closer to the vertex t, meaning that the walk process is more like BFS. Conversely, a smaller q indicates that the vertex is farther away, which is more like DFS.

Each sampling step in biased random walk is based on the transition probability π_{vx}. We can pre-calculate this probability by using the Alias sampling method [8, 9] and then use it directly in the sampling process. In this case, the sampling complexity is $O(1)$, meaning that the walk process of the algorithm is faster.

2. Vector representations of vertices

The biased random walk process enables us to obtain a set of vertex sequences for random walk. Now we will introduce learning vector representations of vertices.

For a given graph $G = (V, \boldsymbol{E})$, the mapping function from a vertex to a vector representation is f: $V \rightarrow \mathbb{R}^{\mathrm{d}}$, where d is the dimension of a representation vector. For each vertex $u \in V$, an algorithm is used to learn the vertex vector representation. This enables us to subsequently optimize the following formula of the objective function:

$$f^* = \underset{f}{\operatorname{argmax}} \sum_{u \in V} \log P(N_s(u) | f(u)) \tag{8.14}$$

where

$f(u)$ is the vector representation of the vertex u; and
$N_S(u)$ is the neighborhood of the vertex u under the neighborhood sampling strategy S.

However, due to the complexity involved in solving the optimization problem, we introduce the following two assumptions:

1. Conditional independence assumption:

$$P(N_S(u)|f(u)) = \prod_{n_i \in N_S(u)} P(n_i|f(u)) \tag{8.15}$$

where n_i is the vertex in the neighborhood of the vertex u under the neighborhood sampling strategy S.

2. Feature space symmetry assumption:

$$P(n_i|f(u)) = \frac{\exp(f(n_i) \bullet f(u))}{\sum_{v \in V} \exp(f(v) \bullet f(u))} \tag{8.16}$$

Given these two assumptions, we can simplify the objective function in Formula (8.14) as follows:

$$\begin{aligned} \frac{\partial \mathrm{E}}{\partial u_j} &= t_j - y_j = e_j f^* \\ &= \underset{f}{\operatorname{argmax}} \sum_{u \in V} [-\log Z_u + \sum_{n_i \in N_S(u)} f(n_i) \bullet f(u)] \end{aligned} \tag{8.17}$$

where Z_u is the partition function of each vertex, and $Z_u = \sum_{v \in V} \exp(f(u) \bullet (f(v))$.

By using negative sampling, we are able to minimize the calculation costs of the partition function. And to optimize Formula (8.17), we can use SGD—similar to training a neural network—in order to continuously learn the parameters of mapping f to obtain the vector representation of each vertex.

8.4.2 Implementation of the Node2Vec Algorithm

Similar to the DeepWalk algorithm, the Node2Vec algorithm is mainly used to generate random walk sequences and learn vector representations. In this section, we describe how to implement the Node2Vec algorithm through pseudocode.

Algorithm 8.4 Pseudocode for Implementing Node2Vec

Input: Graph $G(V, \boldsymbol{E}, W)$, embedding dimension d, number of times γ that each vertex is walked, walk length l, window size w, return parameter p, and in-out parameter q

Output: Embedded mapping function f

(1) Calculate the transition probability π = PreprocessModifiedWeights (G, p, q)

(2) Initialize walk to null

(3) For each iteration iter = 1 to γ, execute:

(4) For each vertex $u \in V$

(5) Generate random walk = node2vecWalk (G', u, l)

(6) Save the random walk to "walks"

(7) Learn mapping f = stochastic gradient descent (w, d, walks)

node2vecWalk (graph $G' = (V, \boldsymbol{E}, \pi)$, start vertex u, walk length l)

(1) Initialize walk to [u]

(2) For the walk from the source vertex walk_iter = 1 to l, execute:

(3) Obtain the current vertex curr = walk[−1]

(4) Obtain the neighbor of the current vertex V_curr = GetNeighbors (curr, G')

(5) Sample vertex s = AliasSample (V_{curr}, π)

(6) Save the sampled vertex s to the end of walk

(7) Return walk

8.5 GCN Algorithm

In the field of computer vision, CNNs achieve good results because discrete convolution can effectively extract spatial features. For low-dimensional matrices (such as images or videos) where pixels are ordered, CNNs calculate weighted summation of center and adjacent pixels to extract spatial features. But when given high-dimensional graph data that lacks an ordered structure, CNNs find it difficult to process the data. In order to solve this problem, Bruna et al. proposed GCNs, which aggregate the vertex information of irregular graph data.

The approaches that apply convolution to graph domain can be divided into spectral and non-spectral. The GCN is a spectral approach that, by leveraging the spectral graph theory, implements convolution operations on topologies, and uses the Laplacian matrix to move convolution operations in the spatial domain to the spectral domain. By representing any vector on a graph as a linear combination of Laplacian eigenmatrices, the features of the graph can be extracted in the spectral domain. This results in the GCN being more effective than the non-spectral approach, which directly extracts the features in the spatial domain. Furthermore, because the GCN model can extract information about the entire graph in one go, and the parameters of

the filter can be shared at all positions in the graph [10], there is no need to calculate the parameters of the filter for each vertex. This in turn significantly reduces the complexity of the model.

Building on the first generation of GCN, Defferranrd et al. proposed to replace the convolution kernel with the Chebyshev polynomial summation [11]. This method enables us to obtain a smooth filter in the frequency domain while reducing the model complexity. Subsequently, numerous approaches for replacing the convolution kernel with mathematical transformation have emerged. In the model described in the following section, Kipf and Welling limit the filter to run in the first-order neighborhood around each vertex, thereby reducing the calculation costs, increasing the network efficiency, and improving the model accuracy.

8.5.1 Principles of the GCN Algorithm

The formula of the two-layer GCN model selected by Kipf and Welling is as follows:

$$Z = f(\mathbf{X}, \mathbf{A}) = \text{Softmax}\Big(\hat{\boldsymbol{A}}\text{ReLU}\Big(\hat{\boldsymbol{A}}\boldsymbol{X}\boldsymbol{W}^{(0)}\Big)\boldsymbol{W}^{(1)}\Big) \tag{8.18}$$

where $\hat{A} = \tilde{\boldsymbol{D}}^{-\frac{1}{2}}\tilde{\boldsymbol{A}}\tilde{D}^{-\frac{1}{2}}$, $\tilde{\boldsymbol{D}}_{ii} = \sum_j \tilde{\boldsymbol{A}}i_j$, and $\tilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}_N$;

$\boldsymbol{A}$ is the adjacency matrix of a graph;
$\boldsymbol{W}^l$ represents $\boldsymbol{W}^{(0)}$ and $\boldsymbol{W}^{(1)}$, which are weight parameters; and
$\boldsymbol{X}$ is the vertex eigenmatrix of the graph.

The following explains the origin of Formula (8.18), starting with the formula:

$$\boldsymbol{L} = \boldsymbol{I}_{\text{N}} - \text{D}^{-1/2}\boldsymbol{L}\boldsymbol{D}^{-1/2} \tag{8.19}$$

We use this formula to define the symmetric normalized Laplacian matrix, where $\boldsymbol{D}$ is the vertex degree matrix.

The Laplacian matrix is then decomposed to obtain the following:

$$\boldsymbol{L} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^{\text{T}} \tag{8.20}$$

where

$\boldsymbol{U}$ is the normalized Laplacian eigenvector matrix (i.e., a spectral matrix), and
$\boldsymbol{\Lambda}$ is the corresponding eigenvalue matrix (a diagonal matrix).

The spectral convolution on the graph may be defined as the product of the signal $x \in \mathbb{R}$ and the filter diag(θ) ($\theta \in \mathbb{R}$) in the Fourier domain, that is:

$$q_\vartheta \star \boldsymbol{x} = \boldsymbol{U} q_\vartheta \boldsymbol{U}^T \boldsymbol{x} \tag{8.21}$$

where

$\boldsymbol{U}^{\mathrm{T}}\boldsymbol{x}$ is the graph Fourier transform of $\boldsymbol{x}$; and
g_θ is the function of the eigenvector $\mathcal{L}$, that is, $g_\theta(\boldsymbol{\Lambda})$.

Computing the Laplacian eigenmatrix involves substantial overheads if the graph data is large, so to reduce the calculation complexity, we can use the K-order truncation of Chebyshev polynomial and thereby approximate $g_\theta(\boldsymbol{\Lambda})$.

$$g_{\theta'}(\Lambda) \approx \sum_{k=0}^{K} \boldsymbol{\theta}_k^{'} T_K(\tilde{\boldsymbol{\Lambda}}) \tag{8.22}$$

where

$\tilde{\boldsymbol{\Lambda}}$ is the eigenvector matrix after scaling is performed based on the maximal eigenvalue $\lambda_{\max}$, of $\mathcal{L}$, and $\tilde{\Lambda} = 2/\lambda_{\max} \bullet \Lambda - I_{\mathrm{N}}$; and
$\boldsymbol{\theta}'$ is the Chebyshev parameter vector, where $\boldsymbol{\theta}' \in \mathbb{R}^K$. The Chebyshev polynomial is defined by recursion: $T_k(\boldsymbol{x}) = 2\boldsymbol{x}T_{k\text{-}1} - T_{k\text{-}2}(\boldsymbol{x})$, where $T_0(\boldsymbol{x}) = 1$ and $T_1(\boldsymbol{x}) = \boldsymbol{x}$.

By replacing g_θ with $g_\theta{}'$, we can obtain:

$$\begin{aligned} g_{\theta'} \star \boldsymbol{x} &\approx z\boldsymbol{U} \sum_{k=0}^{K} \theta' T_k(\tilde{\boldsymbol{\Lambda}}) \boldsymbol{U}^T \boldsymbol{x} \\ &= \sum_{k=0}^{K} \boldsymbol{\theta}' \boldsymbol{U} T_k(\tilde{\boldsymbol{\Lambda}}) \boldsymbol{U}^T \boldsymbol{x} \end{aligned} \tag{8.23}$$

$T_k(\tilde{\Lambda})$ is a k-order polynomial of $\boldsymbol{\Lambda}$, and $\boldsymbol{U}\ \tilde{\Lambda}^{\,k} \boldsymbol{U}^{\mathrm{T}} = (\boldsymbol{U}\ \tilde{\Lambda}\ \boldsymbol{U}^{\mathrm{T}})^{\,k} = \tilde{\boldsymbol{L}}^{\,k}$, where $\tilde{\boldsymbol{L}} = \frac{2}{\lambda_{\max}} \boldsymbol{L} - \boldsymbol{I}_N$. Formula (8.22) can therefore be expressed as follows:

$$g_{\theta'} \star \boldsymbol{x} \approx \sum_{k=0}^{K} \boldsymbol{\theta}' T_k(\tilde{\boldsymbol{L}}) \boldsymbol{x} \tag{8.24}$$

With the Chebyshev polynomial approximation, the spectral convolution is no longer dependent on the entire graph. Instead, it is related to only the k-order vertices (i.e., the kth-order neighborhood) of the center vertex.

After we perform the Chebyshev polynomial approximation, we can consider each convolution operation as aggregating the k-order neighbor information for each center vertex. Even so, the calculation amount remains high after approximation because the graph structure data is large. In order to reduce the calculation costs, we can further simplify the calculation by letting $k = 1$, meaning that the information of only first-order neighbors is aggregated at any given time. In this case, the spectral convolution can be approximated as a linear function of $\tilde{\boldsymbol{L}}$. As mentioned earlier, only the dependence between the center vertex and the first-order neighbor is established. In order to solve this problem, a stacked GCN must be used to establish the dependence of k-order neighbors. We are able to obtain the first-order neighbor information of the second-order graph convolution after superimposing the first-order neighbor of the second-order graph convolution. This means that the center vertex will obtain the second-order neighbor information through the first-order neighbor of the second-order graph convolution, and so on. Furthermore, the Chebyshev polynomial does not limit the dependence of k-order neighbors when this dependence is established.

We can further simplify the calculation. In the linear model of the GCN, we can obtain the following first-order linear approximate expression of spectral convolution by defining $\lambda_{\max} \approx 2$:

$$\begin{aligned} g_{\theta'} \star \boldsymbol{x} &\approx \boldsymbol{\theta}_0^{'}\boldsymbol{x} + \boldsymbol{\theta}_1^{'}(\boldsymbol{L} - \boldsymbol{I}_N)\boldsymbol{x} \\ &= \boldsymbol{\theta}_0^{'}\,\boldsymbol{x} - \boldsymbol{\theta}_1^{'}\boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{x} \end{aligned} \tag{8.25}$$

Formula (8.25) includes only two parameters: $\boldsymbol{\theta}_0^{'}$ and $\boldsymbol{\theta}_1^{'}$. So to establish the dependence of k-order neighbors, we can use a k-layer filter.

We limit the number of parameters to avoid overfitting and minimize the matrix multiplication of each layer to reduce the calculation complexity. If we let $\boldsymbol{\theta} = \boldsymbol{\theta}_0^{'} = -\boldsymbol{\theta}_1^{'}$, we can express Formula (8.25) as follows:

$$g_\theta \star \boldsymbol{x} \approx \vartheta\left(\boldsymbol{I}_N + \boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}}\right)\boldsymbol{x} \tag{8.26}$$

The eigenvalue range of $\boldsymbol{I}_N + \boldsymbol{I}_N + \boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}}$ is [0, 2], meaning that when the operation is repeated continuously (in very deep networks), gradient explosion or disappearance may occur. To avoid this problem, the renormalization trick is introduced:

$$\boldsymbol{I}_N + \boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}} \rightarrow \widetilde{\boldsymbol{D}}^{-\frac{1}{2}}\widetilde{\boldsymbol{A}}\widetilde{\boldsymbol{D}}^{-\frac{1}{2}} \tag{8.27}$$

where $\tilde{\mathbf{A}} = \mathbf{A} + \boldsymbol{I}_N$, $\widetilde{\boldsymbol{D}}_{ii} = \sum_j \tilde{\mathbf{A}}$ When the representation of each vertex in a graph is not a separate scalar but is instead a vector of size C, we can use its variants for processing:

$$\boldsymbol{Z} = \widetilde{\boldsymbol{D}}^{-\frac{1}{2}} \widetilde{\boldsymbol{A}} \widetilde{\boldsymbol{D}}^{-\frac{1}{2}} \boldsymbol{X} \boldsymbol{\vartheta} \tag{8.28}$$

where

$\boldsymbol{\theta}$ is a parameter matrix, and $\boldsymbol{\theta} \in \mathbb{R}^{\mathrm{C} \times \mathrm{F}}$; and
$\boldsymbol{Z}$ is the corresponding convolution result, and $\boldsymbol{Z} \in \mathbb{R}^{\mathrm{N} \times \mathrm{F}}$

In this case, the vertex representation of each vertex is updated to a new F-dimensional vector that includes the information of the corresponding first-order neighbor.

We are now able to obtain the layer-by-layer propagation expression of the graph CNN:

$$\boldsymbol{H}^{(l+1)} = \sigma\left(\widetilde{\boldsymbol{D}}^{-\frac{1}{2}} \widetilde{\boldsymbol{A}} \widetilde{\boldsymbol{D}}^{-\frac{1}{2}} \boldsymbol{H}^{(l)} \boldsymbol{W}^{(l)}\right) \tag{8.29}$$

where the input of the l-layer network is $\boldsymbol{H}^{(l)}$, and $\boldsymbol{H}^{(l)} \in \mathbb{R}^{\mathrm{N} \times \mathrm{F}}$ (initial input is $H^{(0)} =$ X);

N is the number of vertices in the graph, and each vertex is represented by a d-dimensional eigenvector;
$\boldsymbol{W}^{(l)}$ is the weight parameter that needs to be trained, and $\boldsymbol{W}^{(l)} \in \mathbb{R}^{d \times d}$; and
σ is the activation function.

Through this derivation, we are able to obtain the GCN architecture:

$$\boldsymbol{Z} = f(\boldsymbol{X}, \boldsymbol{A}) = \mathrm{Softmax}\left(\hat{\boldsymbol{A}} \mathrm{ReLU}\left(\hat{\boldsymbol{A}} \boldsymbol{X} \boldsymbol{W}^{(0)}\right) \boldsymbol{W}^{(1)}\right) \tag{8.30}$$

8.5.2 Implementation of the GCN Algorithm

This section describes how to implement the GCN algorithm through pseudocode.

Algorithm 8.5 Pseudocode for Implementing GCN

Input: Graph G (V, $\boldsymbol{E}$), vertex eigenmatrix $\boldsymbol{X}$, adjacency matrix $\boldsymbol{A}$, learning rate lr, and labeled vertex set y_L

Output: Output eigenmatrix $\boldsymbol{Z}$

(1) Randomly initialize the weight parameter $\boldsymbol{W}$

(2) Input the vertex feature $\boldsymbol{X}$ and the adjacency matrix $\boldsymbol{A}$

(3) Extract the vertex degree matrix $\boldsymbol{D}$

(4) Calculate the Laplacian matrix $\hat{\boldsymbol{A}}$ based on the matrix $\boldsymbol{D}$ and the matrix $\boldsymbol{A}$

(5) Perform a graph convolution operation on $\boldsymbol{X}$ using $\hat{\boldsymbol{A}}$ and $\boldsymbol{W}$ to obtain a feature map

(6) Use the ReLU function for activation to obtain the output matrix $\boldsymbol{Z}_0$

(7) Perform the graph convolution operation based on the output $\boldsymbol{Z}_0$ of the upper layer to obtain a new feature map

(8) Use the Softmax function for activation to obtain the new output eigenmatrix $\boldsymbol{Z}$

(9) Use the loss function to calculate the loss between the output eigenmatrix $\boldsymbol{Z}$ and y_L

(10) Update the weight parameter $\boldsymbol{W}$ by using gradient descent

(11) Repeat steps (4) to (9)

(12) End

8.6 GAT Algorithm

For the spectral approaches represented by the GCN, each calculation relies on the Laplacian matrix eigenvector and graph structure. This makes it difficult to apply a GCN model on other graphs once it has been trained on a particular graph structure. Furthermore, because the GCN model lacks the inductive ability, it has a limited scope for application.

For the non-spectral approaches, convolution is defined directly on the graph to operate adjacent vertices in space. However, such approaches are problematic, in that it is challenging to define an operation that can handle neighbors of different sizes while also ensuring that CNN parameters can be shared. To address this challenge, researchers have made a series of improvements [12–15]. For example, in 2017, Hamilton et al. proposed a classical inductive learning algorithm called GraphSAGE. In GraphSAGE, sampling is performed based on the neighborhood of a fixed size, and each vertex is represented by an aggregate of its neighbors. This means that a vertex not present during the training can still be appropriately represented by its neighbor vertices if it subsequently appears at a later stage. GraphSAGE has shown promising results in several large-scale induction benchmark tests.

In practice, however, the impacts of neighbor vertices on the target vertices are different. The methods referred to earlier do not take into account that fact that different neighbors are of the same importance. In 2018, Petar et al., taking inspiration from the attention mechanism widely used in deep learning models, proposed the graph attention network (GAT) [16], a graph data vertex classification model based on the attention mechanism [16]. The attention mechanism imitates human intuition, focusing on salient parts helpful for the target task while ignoring other invalid information. Similarly, the GAT pays attention to its neighbors and determines the weights of the neighbor vertices through the self-attention strategy. Different neighbor vertices have different impacts on the target vertices, allowing the hidden representation of each target vertex to be calculated more effectively.

8.6.1 Principles of the GAT Algorithm

This section focuses on the graph attention layer, which is an important component of the graph attention network. Here we make the following assumptions:

- The input of the current attention layer is a set of vertex features: $\boldsymbol{h} = \{\vec{\boldsymbol{h}}_1, \vec{\boldsymbol{h}}_2, \ldots, \vec{\boldsymbol{h}}_N\}, \vec{\boldsymbol{h}}_i \in \mathbb{R}^F$, where N is the number of vertices, and F is the number of features of each vertex.
- The output of the attention layer is a new set of vertex features: $\boldsymbol{h}' = \{\vec{\boldsymbol{h}}'_1, \vec{\boldsymbol{h}}'_2, \ldots, \vec{\boldsymbol{h}}'_N\}, \vec{\boldsymbol{h}}'_i \in \mathbb{R}^{F'}$.

Because we need at least one nonlinear transformation so that we can convert input features into higher-level features, we perform linear transformation on each vertex, and then use the self-attention mechanism a to calculate the attention correlation coefficient e_{ij}. This coefficient indicates the importance of the feature of vertex j to vertex i.

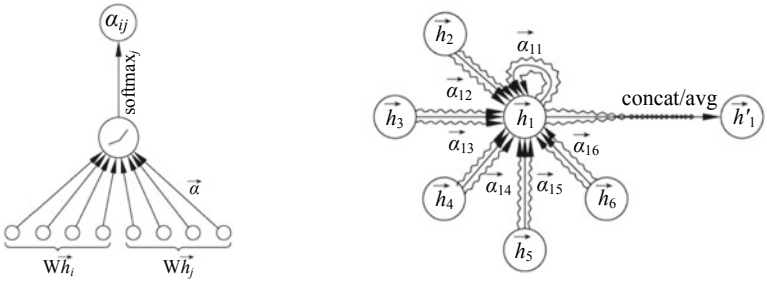
$$e_{ij} = a\left(\boldsymbol{W}\vec{\boldsymbol{h}}_i, \boldsymbol{W}\vec{\boldsymbol{h}}_j\right) \tag{8.31}$$

where

$\boldsymbol{W}$ is the weight matrix, and $\boldsymbol{W} \in \mathbb{R}^{F' \times F}$; and
a is the self-attention mechanism a: $\mathbb{R}^{F'} \times \mathbb{R}^{F'} \to \mathbb{R}$.

We then introduce the attention mechanism into the graph structure through masked attention: e_{ij} is calculated only for vertices of $j \in N_i$, where N_i is a neighbor vertex of vertex i (and includes vertex i itself). To normalize all neighbor vertices j of vertex i, making it easier to compare the coefficients of different vertices, we use the Softmax function:

$$a_{ij} = \text{Softmax}_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in N_i} \exp(e_{ik})} \tag{8.32}$$



(a) Calculation of the attention coefficient

(b) Aggregation of the multi-head graph attention layer ($K = 3$)

Fig. 8.7 Network architecture of the attention mechanism[3]

The attention mechanism a may be a single-layer feedforward neural network, which is determined by $\vec{a} \in \mathbb{R}^{2F'}$ and LeakyReLU nonlinear activation function (slope $a = 0.2$ when the input is negative). Figure 8.7a shows the network architecture of the attention mechanism.

In order to calculate the attention coefficient, we use the following formula:

$$a_{ij} = \frac{\exp(\text{LeakyReLU}(\vec{a}^T[\boldsymbol{W}\vec{\boldsymbol{h}}_i || \boldsymbol{W}\vec{\boldsymbol{h}}_j]))}{\sum_{k \in N_i} \exp(\text{LeakyReLU}(\vec{a}^T[\boldsymbol{W}\vec{\boldsymbol{h}}_i || \boldsymbol{W}\vec{\boldsymbol{h}}_k]))} \tag{8.33}$$

where $\bullet^T$ indicates the transpose operation, and $\|$ indicates the concatenation operation.

By performing the preceding calculation, we are able to obtain the normalized attention coefficient, which we can subsequently use to calculate the output features of each vertex.

$$\vec{\boldsymbol{h}}_i^{'} = \sigma\left(\sum_{j \in N_i} a_{ij} \boldsymbol{W}\vec{\boldsymbol{h}}_j\right) \tag{8.34}$$

Similar to the Transformer model proposed by Vaswani et al., extending the self-attention mechanism to the multi-head attention mechanism improves the calculation stability. The calculation process involves performing K calculations separately based

[3] Source: https://arxiv.org/pdf/1710.10903.pdf.

on the self-attention mechanism, and then concatenating the obtained features to obtain the final vertex representation:

$$\vec{\boldsymbol{h}}_i^{'} = ||_{k=1}^{K} \sigma\left(\sum_{j \in N_i} a_{ij}^k \boldsymbol{W}^k \vec{\boldsymbol{h}}_j\right) \tag{8.35}$$

where

a_{ij}^k is the normalized attention coefficient obtained through the Kth calculation based on the self-attention mechanism; and
W^k is the corresponding input weight matrix for linear transformation.

When the multi-head attention mechanism is used at the last layer of the network, concatenation is less effective and so is replaced with an averaging operation. After nonlinear activation, the vertex representation of the multi-head attention layer is obtained:

$$\vec{\boldsymbol{h}}_i^{'} = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j \in N_i} a_{ij}^k \boldsymbol{W}^k \vec{\boldsymbol{h}}_j\right) \tag{8.36}$$

Figure 8.7b illustrates the aggregation process of the multi-head graph attention layer ($K = 3$). Different arrow styles in the graph represent separate attention calculation processes. The representation of a target vertex in the graph attention network is the weighted sum of its first-order neighbor vertices including the target vertex, which is a calculation process on a local graph.

The GAT has a number of advantages, such as high efficiency, flexibility, and portability. To achieve high efficiency, the GAT implements parallel calculation for the local graph vertex neighbor pair. To realize flexibility, it assigns different weights to vertices of different degrees. And in terms of portability, the model can be extended to unknown graphs, representing the ability of inductive learning. The GAT considers the different importance of neighbor vertices to target vertices and has achieved good results in some practical scenarios.

8.6.2 Implementation of the GAT Algorithm

This section describes how to implement the GAT algorithm through pseudocode.

Algorithm 8.6 Pseudocode for Implementing the GAT Algorithm

Input: Graph $G(V, \boldsymbol{E})$, vertex feature $\boldsymbol{h}$, vertex label y, learning rate lr, and number K of the units in the multi-head attention mechanism
Output: Weight parameter $\boldsymbol{W}$ and $\vec{\mathbf{a}}$
(1) Randomly initialize the weight parameter matrix $\boldsymbol{W}$ and $\vec{\mathbf{a}}$
(2) Repeatedly execute:
(3) For the input layer and the hidden layer:
(4) Execute the self-attention mechanism for each iteration iter = 0 to K
(5) Calculate the attention cross-correlation coefficient α_{ij}
(6) Calculate the output feature $\boldsymbol{h}'$ of the vertex under the self-attention mechanism
(7) Concatenate K output eigenvectors to obtain the output feature $\boldsymbol{h}'$ of the vertex under the multi-head attention mechanism
(8) Use the output feature $\boldsymbol{h}'$ as the input feature at the next layer
(9) For the output layer:
(10) Repeat steps (3) to (5)
(11) Average K output eigenvectors to obtain the output feature $\boldsymbol{h}'$ of the vertex under the multi-head attention mechanism
(12) Update the weight parameter matrix $\boldsymbol{W}$ and $\vec{\mathbf{a}}$ based on the difference between the vertex label y and the output feature $\boldsymbol{h}'$
(13) Proceed until the stop condition is met

8.7 Application: Recommendation System

We are currently in the midst of an information boom driven by the unprecedented popularity of the Internet and the near-ubiquitous use of mobile terminals. In today's fast-paced world, people want information at their fingertips. But given the vast amounts of information now available, it has become critical to ensure that people can obtain what they want, when they need it. In order to meet such demands, the recommendation system is developed.

The existing recommendation systems employ collaborative filtering, indicating that similar users like the same items and, conversely, the same user likes similar items. This type of filtering is divided into memory-based methods and model-based methods.

Memory-based methods can be further divided into user-based and item-based collaborative filtering. User-based collaborative filtering recommends items to similar users, whereas item-based collaborative filtering recommends similar items to users. In order to implement these two methods, we need to define the similarity

between items or users. Although these methods are simple, easy to understand, and easy to implement, a significant amount of time is required to calculate the similarities between each pair of items or users, and to find similar items or users, especially when there are a huge number of items or users.

For highly efficient recommendation systems, one of the most successful methods for implementing collaborative filtering is the model-based matrix factorization. In this model, a user and an item (a user-item pair) are modeled as implicit vectors in the same space based on interactions between the user and item. For an unknown user-item pair, the preferences are calculated based on the vectors of the corresponding user and item (usually through the vector inner product operation). Two popular models in the matrix factorization family are SVD and SVD $++$. The SVD model, which is based on the general matrix factorization, improves the stability of the model training by introducing user bias, item bias, and global bias variables. SVD $++$, as an extension of SVD, improves the effectiveness by introducing an auxiliary feature: the interaction history between the user and the item.

8.7.1 Recommendation System in Industry Applications

The recommendation system dates back to as early as the twentieth century, but it wasn't until the last decade when the industry began adopting it more widely. For example, it is estimated that Amazon sells more than 35% of its listed items through recommendation systems. In addition, by using recommendation systems, Google generated revenue of $43 billion in Internet advertising in 2014, and in 2015, Google Play and Apple's App Store earned $10 billion and $21 billion, respectively. Huawei—ranked highly in the Fortune Global 500 list—also applies the recommendation system throughout its business operations. The recommendation system involves an extensive range of content, so extensive in fact that we could write an entire book dedicated exclusively to this topic. So, given the space limitations in this book, we will focus on the recommendation system only from the perspective of industry researchers.

Industry recommendation systems consist of three steps: candidate set generation, matching prediction, and sorting. The number of items that these systems recommend may be millions or more, but matching predictions and sorting on such large candidate sets cannot be performed within an acceptable timeframe. As a result, it is necessary to generate a smaller candidate set (typically ranging from hundreds to thousands) based on the current recommendation scenario, the features of the item, and even the user's preferences. After the candidate set is generated, the matching prediction model predicts the current user's preference for each item in the candidate set. Ultimately, the sorting step combines the results of the matching prediction model with business rules to generate the final sorting results.

An important part of industry recommendation systems is click through rate (CTR) prediction, which first appeared in online advertisement scenarios and belongs to the matching prediction step described earlier. In online advertisement scenarios using the cost per click (CPC) model, revenue is generated for the platform each time a user clicks on an advertisement. The amount of revenue is specified in a contract between the advertiser and platform. In most cases, the platform uses CTR $\times$ bid sorting rules for candidate advertisements, where CTR is an estimate of the current user's CTR for the advertisement, and bid represents the amount of money the advertiser will pay to the platform if the user clicks on the advertisement. The sorting rules arrange candidate advertisements according to the expected benefits; that is, they are sorted based on the revenue they generate for the platform each time they are displayed. Such rules are also used for real-time bidding advertisements, and similar rules are used in game and video sorting scenarios. Game sorting is generally based on CTR $\times$ LTV, where life time value (LTV) is the average fee a user pays for the game. Video sorting generally uses CTR $\times$ WT, where watch time (WT) is the average time a user spends watching the video. Given its wide scope of application, CTR prediction is extremely important in the industry recommendation system.

The recommendation system models used by most enterprises have evolved from the wide model—using logistic regression (LR) or factorization machine (FM)—to the deep learning model, and then to the reinforcement learning model in addition to the deep learning model where the graph structure is considered.

To understand the graph neural network-based model, we first need to understand the input data form used in the recommendation system. This input data form differs significantly from that used in NLP and computer vision. The recommendation system covers many discrete features, such as gender, city, and day of the week. Because these features have no numerical meaning, they are typically represented by one-hot encoding. In this encoding method, all possible values are represented by a high-dimensional vector with a value of 0–1, where the corresponding bit is 1 and all other bits are 0. The dimension of the one-hot vector is the number of all possible values. For example, "Friday" can be represented as [0, 0, 0, 0, 1, 0, 0], the gender "male" can be represented as [0, 1], and the city "Shanghai" can be represented as [0, 0, 1, …, 0]. From the preceding information, we can see that the input data of the recommendation system is usually high-dimensional and sparse.

8.7.2 Graph Neural Network Model in a Recommendation System

Ultra-large recommendation systems face several challenges due to the high-dimensional and sparse nature of their input data:

1. Storage: The data is structured and all features are arranged in a certain order, with many of them duplicated. For example, if there are 10,000 male users, the

system needs to store 10,000 male-represented vectors, such as [0, 1]. As the number of features, users, and items increases, the amount of duplicated data becomes larger.

2. Sparsity: For movie-recommending scenarios like MovieLens [17], the data is usually represented by a "user-item" score matrix. As the number of users and items increase, the dimensions and sparsity of the score matrix also increase. This is because most users do not score most items, and the collaborative filtering algorithm relies on the score matrix.
3. Scalability: The ability to process ever-increasing volumes of data and the exponential growth of collaborative filtering calculation make it extremely difficult for recommendation systems to scale easily [18].

The graph data structure makes it possible to address these challenges.

1. For repeated storage of features, male can be represented as a vertex on a graph, with all male users having an edge from the user's vertex to the male vertex. This means that information about only the edge needs to be maintained. For higher-dimensional features, the effectiveness of graph structure storage is more pronounced.
2. In response to the sparsity challenge, graph structure storage is vertex-centric, and only the in-edge and out-edge are maintained.
3. To facilitate scalability, the graph structure makes it easy to add new vertices and edges, requiring the model to be updated only for the new additions.

PinSage [18], jointly published by Pinterest and Stanford University, is the industry's first commercial end-to-end recommendation model based on the graph neural network. Pinterest, an image-based social networking site, displays images in the form of a waterfall stream, where new images are automatically loaded at the bottom of the page without needing users to change the current page. Users can pin images of interest on the pinboard and can save and share the images, while other users can follow and forward the images. The main items recommended on Pinterest are images (called Pins), which may include images of food, clothes, products, etc. Users group images they like into Boards. Pinterest data can be modeled to construct a bipartite graph, which includes two types of vertices (Pins and Boards). In the bipartite graph, shown in Fig. 8.8, there is no connection edge between vertices of

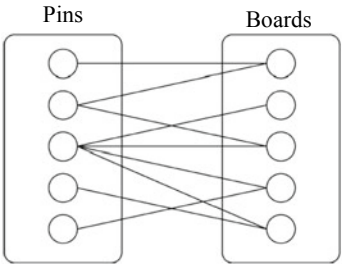


Fig. 8.8 Pinterest bipartite graph

the same type, and the vertex features include images and textual annotations (title and description).

In the traditional GCN, the entire graph is used for training. However, in industrial recommendation scenarios such as Pinterest, there are billions of vertices and tens of billions of edges, making it difficult to perform operations if the entire graph is used for training. To solve this problem, PinSage took inspiration from GraphSage to make improvements to the GCN. GraphSage can be considered as a GCN based on random walk and is an inductive variant of the GCN. It learns the vertex representation by sampling the neighbor information of the aggregated vertex in order to avoid operating the entire Laplacian matrix of the graph. This means that GraphSage can be generalized to an unknown vertex if one exists, and its neighbor information can be used to learn the representation of the vertex. The key improvements PinSage made to the GCN are as follows:

1. Local graph convolution is performed by dynamically constructing a new computational graph through random walk (short random walks) sampling of vertex neighbors. Because the importance of different neighbors to the target vertex is different, the neighbor will have an importance score during the information aggregation.
2. Distributed training is performed based on mini-batch. The CPU is used to sample vertex neighbors to obtain the features required for defining local convolution. Through tensor calculation and hardware acceleration, the distributed stochastic gradient descent calculation is performed for each pre-calculated small graph. The convolution operation can be performed separately, and the parameters of each convolutional layer are shared.
3. Repeated calculation of vertex neighbors is eliminated by using related technologies during inference.

The PinSage algorithm uses the local graph convolution to learn the vertex embedding of the web-level graph containing billions of objects, whereby high-quality vertex embedding facilitates subsequent recommendations. The PinSage algorithm can be summarized into two parts.

The first part is convolution, which is shown in Algorithm 8.7. The vertex embedding calculation, vertex neighbors, weights of the vertex neighbors, and aggregate function are used as the input. Through information aggregation, the neighbor embedding (line 1 of the pseudocode) is calculated. Then, the neighbor and vertex embeddings are used to update the current vertex embedding (line 2 of the pseudocode). Finally, the resulting vertex embedding is normalized (line 3 of the pseudocode).

The method for sampling vertex neighbors during information aggregation has two advantages: (1) the number of neighbors is fixed and the memory used for calculation is controllable; and (2) different importance of the neighbors to the vertex is used for information aggregation. Each time the convolution operation (Algorithm 8.7) is used to obtain the new embedding of a vertex, more information about the local graph structure around the vertex can be obtained by superposing several such convolutions.

Algorithm 8.7 Convolution.

Input: Embedding $\mathbf{z}_u$ of the current vertex u, neighbor embedding set $\{\mathbf{z}_v|v\in N(u)\}$, neighbor weight set α, and aggregate function $\gamma(\bullet)$

Output: New embedding $\mathbf{z}_u^{\mathrm{NEW}}$ of the vertex u.

(1) Aggregate neighbor information

$$\boldsymbol{n}_u \leftarrow \gamma\left(\left\{\mathrm{ReLU}\left(\boldsymbol{Q}\boldsymbol{h}_v + q\right)|v \in N(u)\right\}, \alpha\right)$$

(2) Calculate and update the vertex embedding

$$z_u^{NEW} \leftarrow \mathrm{ReLU}\left(\boldsymbol{W} \bullet \mathrm{Concat}\left(z_u, \boldsymbol{n}_u\right) + \boldsymbol{w}\right)$$

(3) Normalize the vertex embedding

$$z_u^{NEW} \leftarrow z_u^{NEW} / \left\| z_u^{NEW} \right\|_2$$

The second part of the PinSage algorithm is mini-batch, shown in Algorithm 8.8, which stacks convolutions into a mini-batch of vertices M to generate the embedding. The mini-batch vertex neighbor sampling process is performed to obtain the neighbor of each vertex (lines 2–8 of the pseudocode). Then, K convolutions are used to iteratively generate K representations of the target vertex (lines 9–16 of the pseudocode). Finally, the vertex embedding is obtained through learning (based on the previously obtained embedding) by using a fully connected neural network (lines 17–19 of the pseudocode). $\boldsymbol{G}_1$, $\boldsymbol{G}_2$, and g are the parameters of the fully connected layer.

Algorithm 8.8 Mini-batch.

Input: Small-batch vertex set $M \subset V$, depth parameter K, and neighbor function N: $V \to 2^V$

Output: Embedding z_u, where $\forall u \in M$

(1) Mini-batch1: Neighbor sampling of mini-batch vertices

(2) $S^{(K)} \leftarrow M$

(3) For each depth parameter $k = K, K\text{-}1, \ldots, 1$, execute:

(4) $S^{(k-1)} \leftarrow S^{(k)}$

(5) For each vertex $u \in S^{(k)}$, execute:

(6) $S^{(k-1)} \leftarrow S^{(k-1)} \cup N(u)$

(7) End

(8) End

(9) Generate the vertex embedding

(10) $\boldsymbol{h}_u^{(0)} \leftarrow \boldsymbol{x}_u, \forall u \in S^{(0)}$

(11) For each depth parameter $k = 1, 2, \ldots, K$, execute:

(12) For each vertex $u \in S^{(k)}$, execute:

(13) $\mathcal{H} \leftarrow \{\boldsymbol{h}_v^{(k-1)}, \forall v \in N(u)\}$

(14) $\boldsymbol{h}_u^{(k)} \leftarrow \text{Convolve}^{(k)}(\boldsymbol{h}_u^{(k-1)}, \mathcal{H})$

(15) End

(16) End

(17) For each vertex $u \in M$, execute:

(18) $\boldsymbol{z}_u \leftarrow \boldsymbol{G}_2 \cdot RELU(\boldsymbol{G}_1 \boldsymbol{h}_u^{(K)} + g)$

(19) End

PinSage has achieved positive results and encouraged the use of the graph convolution algorithm in commercial recommendation systems. In the future, graph neural networks can be expanded to solve the learning problems of other large-scale graph representations and generate greater value in real-world scenarios.

References

1. B. Perozzi, R. Al-Rfou, S. Skiena, DeepWalk: online learning of social representations, in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining,* 20, 701–710
2. J. Zhou , G. Cui, Z. Zhang et al., *Graph Neural Networks: A Review of Methods and Applications* [EB/OL]. (2019–07–10) [2019–10–28]. https://arxiv.org/pdf/1812.08434.pdf
3. T. Mikolov, K. Chen, G. Corrado et al., *Efficient Estimation of Word Representations in Vector Space* [EB/OL]. (2013–09–07) [2019–10–28] https://arxiv.org/pdf/1301.3781.pdf%5D

4. A. Mnih, G.E. Hinton, A scalable hierarchical distributed language model. Adv. Neur. Inf. Proc. Syst. 1081–1088 (2009)
5. F. Morin, Y. Bengio, Hierarchical probabilistic neural network language model, in *Proceedings of the International Workshop on Artificial Intelligence and Statistics, 5,* 246–252
6. J. Tang, M. Qu, M. Wang et al. Line: large-scale information network embedding, in *Proceedings of the 24th International Conference on World Wide Web. International World Wide Web Conferences Steering Committee*, vol. 24 (2015), pp. 1067–1077.
7. A. Grover, J. Leskovec, Node2Vec: scalable feature learning for networks, in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, vol. 22 (2016), pp. 855–864
8. J.R. Norris, *Markov Chains* (Cambridge University Press, Cambridge, 1998)
9. A.J. Walker, New fast method for generating discrete random numbers with arbitrary frequency distributions. Electron. Lett. **10**(8), 127–128 (1974)
10. D.K. Duvenaud, D. Maclaurin, J. Iparraguirre et al., Convolutional networks on graphs for learning molecular fingerprints. Adv. Neu. Inf. Proc. Syst. 2224–2232 (2015)
11. T.N. Kipf, M. Welling, Semi-supervised classification with graph convolutional networks, in *International Conference on Learning Representations* (2017)
12. J. Atwood, D. Towsley, Diffusion-convolutional neural networks. Adv. Neu. Inf. Proc. Syst. 1993–2001 (2016)
13. M. Niepert, M. Ahmed, M. Kutzkov, Learning convolutional neural networks for graphs, in *Proceedings of The 33rd International Conference on Machine Learning*, vol. 48 (2016) 2014–2023
14. F. Monti, D. Boscaini, J. Masci et al., Geometric deep learning on graphs and manifolds using mixture model CNNs, in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition,* (2017), pp. 5115–5124
15. W. Hamilton, Z. Ying, J. Leskovec, Inductive Representation Learning on Large Graphs. Adv. Neu. Inf. Proc. Syst. 1024–1034 (2017)
16. F.M. Harper, J.A. Konstan, The movieLens datasets: history and context. Acm Trans. Interact. Intell. Syst. **5**(4), 19 (2016)
17. L. Sharma, A. Gera, A survey of recommendation system: research challenges. Int. J. Eng. Trends Technol. **4**(5), 1989–1992 (2013)
18. R. Ying, R. He, K. Chen et al. Graph convolutional neural networks for web-scale recommender systems, in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (ACM, 2018), pp. 974–983