7 Graph Neural Networks

7.1 Introduction

Graphs are the mathematical representation of networks. Networks describe interactions between entities; for example, a social network of friends, bonds between atoms in a molecule or protein, the internet of web pages, the cellular communication network between users, a financial transaction network between bank clients, protein-to-protein interaction networks, or the neural networks between neurons in our brains. Specifically, the human brain consists of around 100 billion neurons (for comparison, the cat brain consists of around one billion neurons) with 100 trillion connections between neurons. Each neuron is connected to 5,000–200,000 other neurons, and there are around 10,000 different types of neurons. Perhaps most importantly, around 1,000 neurons are generated each day of our lives. Modeling such networks requires a dynamic graph structure.

Each node in a network may have an associated feature vector, as shown in Figure 7.1. For example, in a graph representing a molecule or protein, the nodes are the atoms, the bonds between atoms are the edges, and each node has an associated feature vector of atom properties. In a social network, each node may represent a user. The edges are connections between users. Each node may have a feature vector, including the user's age, gender, status, country, occupation, interests, likes, etc.

Network data is often messy or incomplete, and therefore we would like to be

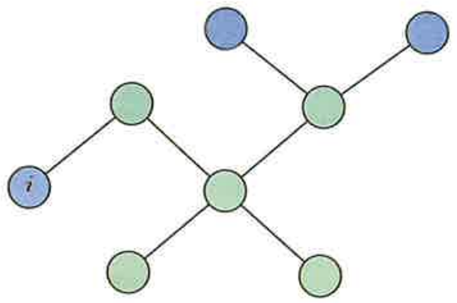

Figure 7.1 A graph with nodes. Each node $i \in \mathcal{V}$ is associated with a feature vector $v_i \in \mathcal{R}^n$.

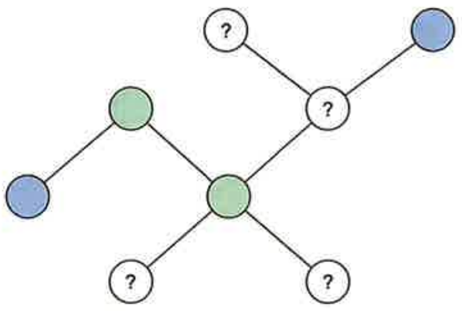

Figure 7.2 Graph node classification. The goal is to classify the uncolored nodes to one of two classes, green or blue.

able to perform operations on graphs, such as completing missing information in the graph. Common tasks on networks include node classification for predicting the type of nodes, as shown in Figure 7.2, link prediction for predicting whether two nodes are connected, finding clusters for detecting communities, and measuring the similarity between nodes for embedding node features into a low-dimensional space. Specifically, we will use deep learning for performing three key operations on graphs:

1. Node prediction: predicting a property of a graph node.
2. Link prediction: predicting a property of a graph edge. For example, in a social network, we can predict whether two people will become friends.
3. Graph or sub-graph prediction: predicting a property of the entire graph or a sub-graph. For example, given a graph representation of a protein, we can predict its function as an enzyme or not. Given a molecule represented as a graph, we can predict whether it will bind to a given receptor.

Notice that if we only have node information and the task is edge prediction, we may pool the information from the graph nodes. Similarly, if we only have edge information and the task is node prediction, we may pool information from the graph edges.

A fundamental property common to neural network representations that work well is that they all share weights. Chapter 5 on convolutional neural networks (CNNs) describes neural networks applied to images of fixed size and regular grids, sharing weights across space, as shown on the left of Figure 7.3, by using a CNN or ResNet or ODENet. Chapter 6, on sequence models, describes neural networks applied to sequences, sharing weights across time, as shown in the center of Figure 7.3, by using a recurrent neural network (RNN), long short-term memory (LSTM), or gated recurrent unit (GRU). In this chapter, we describe graph neural networks (GNNs), applied to networks or general graphs sharing weights across neighborhoods, as shown in the right of Figure 7.3. A key insight in GNNs is that, similarly to CNNs or RNNs, nodes in the graph may aggregate information from neighboring nodes.

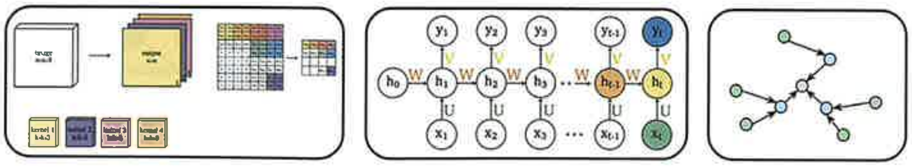

Figure 7.3 Neural network representations sharing weights. A CNN shares weights across space (left); an RNN shares weights across time (center); and a GNN shares weights across neighborhoods (right).

7.2 Definitions

We begin with basic graph definitions. A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ contains a set of n vertices (or nodes) $\mathcal{V}$ and a set of m edges $\mathcal{E}$ between vertices. The edges of the graph can either be undirected or directed.

A common duality of modeling problems in computer science is using graph theory by a graph representation or linear algebra by a matrix representation. Moving back and forth between graph theory and linear algebra allows us to apply algorithms from both.

Two basic graph representations are an adjacency matrix and adjacency list. An adjacency matrix A of dimensions $n \times n$ is defined such that:

$$A_{i,j} = \begin{cases} 1, & \text{if there is an edge between vertices } i \text{ and } j \\ 0, & \text{otherwise} \end{cases} \tag{7.1}$$

If the edges have weights then the 1s in the adjacency matrix are replaced with edge weights $w_{i,j}$. For an undirected graph the matrix A is symmetric.

The adjacency matrix of the example graph in Figure 7.4 with 9 nodes and 11 edges is the 9×9 matrix:

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{7.2}$$

where the number of 1s in matrix A is twice the number of edges in the graph.

Notice that different permutations of the node labels result in different adjacency matrices. In contrast, an adjacency list of the edges in the graph is invariant to node permutations. Storing an adjacency matrix takes $O(n^2)$ mem-

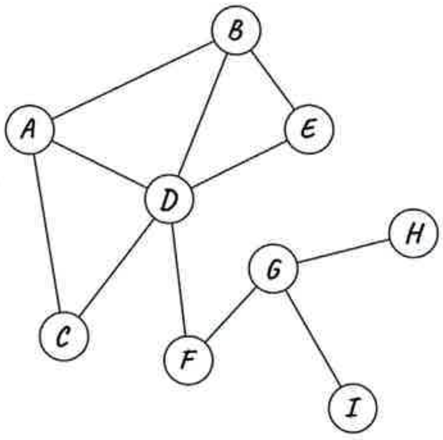

Figure 7.4 Example graph with 9 nodes and 11 edges.

ory, where n is the number of nodes in the graph; storing an adjacency list takes only $O(m+n)$, where m is the number of edges in the graph.

The degree of a node represents the number of edges incident to that node, and the average degree of a graph is the average degree over all its nodes $\frac{1}{n}\sum_{i=1}^{n} d_i$, which equals $\frac{2m}{n}$ for an undirected graph and $\frac{m}{n}$ for a directed graph. In a complete undirected graph, there is an edge between every two vertices for a total of $\frac{n(n-1)}{2}$ edges.

The degree matrix D of the adjacency matrix A is a diagonal matrix such that:

$$D_{i,i} = \text{degree}(v_i) = d_i = \sum_{j=1}^{n} A_{i,j} \tag{7.3}$$

The neighbors of a node $i \in \mathcal{V}$ are its adjacent nodes $\mathcal{N}(i)$, and the degree of a node is its number of neighbors $d_i = |\mathcal{N}(i)|$. The degree matrix of the graph in Figure 7.4 is the 9×9 diagonal matrix:

$$D = \begin{bmatrix} 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.4}$$

In a regular graph, each node has the same number of neighbors, which is the degree of a node.

The graph Laplacian matrix L is the difference between the degree matrix and adjacency matrix $L = D - A$. The Laplacian matrix of the example graph in Figure 7.4 is given by the matrix:

$$L = D - A = \begin{bmatrix} 3 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 3 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 2 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & 5 & -1 & -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & -1 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 3 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix} \tag{7.5}$$

The adjacency matrix and the degree matrix are symmetric, and therefore, the Laplacian matrix is symmetric. Normalizing the Laplacian matrix makes diagonal elements equal 1 and scales off-diagonal entries. The graph symmetric normalized Laplacian matrix is:

$$L^{\text{sym}} = D^{-\frac{1}{2}} L D^{-\frac{1}{2}} = I - D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \tag{7.6}$$

where $D^{-\frac{1}{2}}$ is a diagonal matrix with entries $D_{i,i}^{-\frac{1}{2}} = \frac{1}{\sqrt{\mathrm{d}_i}}$. Nodes without neighbors are not normalized to avoid division by zero. The symmetric normalized Laplacian matrix elements are given by:

$$L_{i,j}^{\text{sym}} = \begin{cases} 1, & \text{if } i = j \text{ and } \mathrm{d}_i \neq 0 \\ -\frac{1}{\sqrt{\mathrm{d}_i \mathrm{d}_j}}, & \text{if } i \neq j \text{ and node } i \text{ is adjacent to node } j \\ 0, & \text{otherwise} \end{cases} \tag{7.7}$$

The symmetric normalized Laplacian matrix of the example graph in Figure 7.4 is given by the matrix:

$$L^{\text{sym}} = \begin{bmatrix} 1 & -\frac{1}{3} & -\frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{15}} & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{3} & 1 & 0 & -\frac{1}{\sqrt{15}} & -\frac{1}{\sqrt{6}} & 0 & 0 & 0 & 0 \\ -\frac{1}{\sqrt{6}} & 0 & 1 & -\frac{1}{\sqrt{10}} & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{\sqrt{15}} & -\frac{1}{\sqrt{15}} & -\frac{1}{\sqrt{10}} & 1 & -\frac{1}{\sqrt{10}} & -\frac{1}{\sqrt{10}} & 0 & 0 & 0 \\ 0 & -\frac{1}{\sqrt{6}} & 0 & -\frac{1}{\sqrt{10}} & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{\sqrt{10}} & 0 & 1 & -\frac{1}{\sqrt{6}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\frac{1}{\sqrt{6}} & 1 & -\frac{1}{\sqrt{3}} & -\frac{1}{\sqrt{3}} \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{\sqrt{3}} & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{\sqrt{3}} & 0 & 1 \end{bmatrix} \tag{7.8}$$

which is a symmetric matrix.

The random walk normalized Laplacian matrix is a transition matrix for a random walk on a graph with non-negative weights and is defined as:

$$L^{\text{rw}} = D^{-1} L = I - D^{-1} A \tag{7.9}$$

where D^{-1} is a diagonal matrix with entries $D^{-1}_{i,i} = \frac{1}{\mathrm{d}_i}$. The random walk normalized Laplacian matrix elements are given by:

$$L^{\mathrm{rw}}_{i,j} = \begin{cases} 1, & \text{if } i = j \text{ and } \mathrm{d}_i \neq 0 \\ -\frac{1}{\mathrm{d}_i}, & \text{if } i \neq j \text{ and node } i \text{ is adjacent to node } j \\ 0, & \text{otherwise} \end{cases} \tag{7.10}$$

The random walk normalized Laplacian matrix of the example graph in Figure 7.4 is given by the matrix:

$$L^{\mathrm{rw}} = \begin{bmatrix} 1 & -\frac{1}{3} & -\frac{1}{3} & -\frac{1}{3} & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{3} & 1 & 0 & -\frac{1}{3} & -\frac{1}{3} & 0 & 0 & 0 & 0 \\ -\frac{1}{2} & 0 & 1 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{5} & -\frac{1}{5} & -\frac{1}{5} & 1 & -\frac{1}{5} & -\frac{1}{5} & 0 & 0 & 0 \\ 0 & -\frac{1}{2} & 0 & -\frac{1}{2} & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{2} & 0 & 1 & -\frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\frac{1}{3} & 1 & -\frac{1}{3} & -\frac{1}{3} \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix} \tag{7.11}$$

which is not symmetric and each row sums to zero. The matrices L^{rw} and L^{sym} are similar and therefore have the same eigenvalues.

A graph with n nodes has n eigenvectors with eigenvalues that are non-negative since the Laplacian matrix L has non-negative eigenvalues. A sub-graph of a graph is a subset of edges and all their nodes in the graph. If there is at least one path between each pair of nodes in the sub-graph, it is a connected component. The number of zero eigenvalues of the Laplacian matrix of a graph is the number of its connected components.

A walk on a graph begins with a node $i \in \mathcal{V}$ and ends with a node $j \in \mathcal{V}$ and traverses a sequence of edges and nodes between nodes i and j. If the nodes are distinct, the walk is a path; if the edges are distinct, the walk is a trail. In the matrix, A^k, which is the adjacency matrix to the power of k, each entry $A^k_{i,j}$ is the number of walks of length k in the graph between the node in row i and the node in column j.

Graph nodes may consist of features x. For example, a binary feature x for the graph shown in Figure 7.4 may be defined by appending a column to the

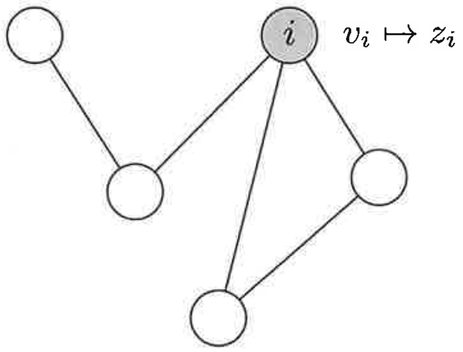

Figure 7.5 Graph node embedding. A node $i \in V$ with associated feature vector $v_i \in \mathcal{R}^n$ which is embedded into a low-dimensional space $z_i \in \mathcal{R}^d$ by an embedding $f : v_i \mapsto z_i$.

adjacency matrix:

$$\begin{bmatrix} A & B & C & D & E & F & G & H & I & x \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{7.12}$$

or, for example, a graph in which the nodes are papers, the edges are the other papers they cite, and the features are the paper abstract or the language embedding of the abstract.

Most graphs are sparse, with fewer edges than square nodes $m \ll n^2$; therefore, adjacency lists are an alternative representation for efficient storage. A linked list represents each vertex and all its edges and adjacent vertices.

7.3 Embeddings

An example of embedding a node in a graph into $\mathcal{R}^n$ is an embedding such that similar nodes in the graph along with their features are embedded to nearby nodes in the embedding space. Our embedding objective may not be limited to similarity and may be defined with respect to other properties of the graph and embedding space. We define an encoder f of a node $i \in V$, such that $f(i)$ is the embedding of the node feature vector v_i as shown in Figure 7.5.

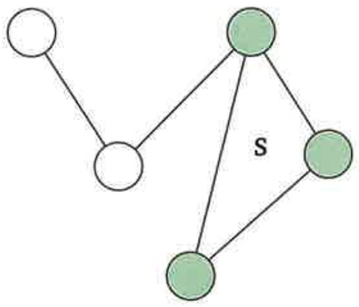

Figure 7.6 Sub-graph embedding by taking the sum of the embeddings of the nodes in the sub-graph.

If each node $i \in \mathcal{V}$ has an associated feature vector v_i then a node embedding, maintaining similarity, maps node feature vectors v_i to vectors z_i in a low-dimensional space such that the similarity between nodes i and j, denoted by $s(i,j)$, is maintained in the embedding space. For example, we may optimize for the similarity between nodes i and j, such that their similarity $s(i,j)$ is maintained after the embedding $f(i)^T f(j)$, where f denotes the encoder which embeds node feature vectors.

A shallow node embedding uses an $n \times 1$-dimensional one-hot encoding e_i of each node i to look up the embedded node. The one-hot encoding e_i of a node $i \in V$ is an $n \times 1$ zero vector except for a single 1 in position i. An embedding matrix W of dimensionality $d \times n$, where d is the dimensionality of a node feature vector v_i and n is the number of nodes, is formed such that each column of W is the embedding of a different node. Multiplying the $d \times n$ embedding matrix W by the $n \times 1$ one-hot encoded vector e_i representing a node i results in We_i, which is the $d \times 1$ ith column of the matrix W representing the node in the embedding space. This results in a problem with shallow embeddings: they do not share weights. As demonstrated earlier, the success of neural networks stems from representations sharing weights across space in CNNs or across time in RNNs. This motivates the sharing of weights by aggregating graph neighborhoods in GNNs, as described in Section 7.5.

We may embed a sub-graph $S \in \mathcal{G}$, either by taking the sum of the embeddings of the nodes in the sub-graph $\sum_{i \in S} f(i)$, or by taking a representative node j of the sub-graph and setting the sub-graph embedding to be $f(j)$ as shown in Figures 7.6 and 7.7.

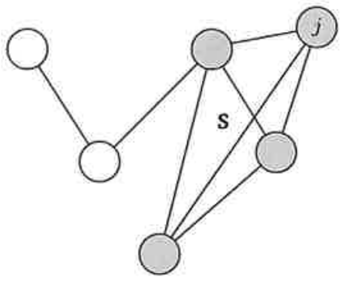

Figure 7.7 Sub-graph embedding by taking a representative node of the sub-graph.

7.4 Node Similarity

7.4.1 Adjacency-based Similarity

In node embeddings we define pairwise node similarity and optimize an embedding to approximate similarities. Going beyond shallow node embeddings, we can define different measures of similarity. For example, we define the similarity between nodes i and j to be the weight on the edge between them, $s(i,j) = A_{i,j}$, where A is the weighted adjacency matrix. We then find the matrix W with dimensions $d \times n$ which minimizes the loss:

$$\mathcal{L} = \sum_{(i,j)\in\mathcal{V}\times\mathcal{V}} \|f(i)^T f(j) - A_{i,j}\|^2 \tag{7.13}$$

over all pairs of nodes in the graph.

7.4.2 Multi-hop Similarity

The first ring of neighbors of a node, as shown in Figure 7.8, is the node's neighborhood. Let A denote the adjacency matrix of 1-hop neighbors, A^2 denote the adjacency matrix of 2-hop neighbors and in general A^k the adjacency matrix of k-hop neighbors. Then, we can minimize the loss:

$$L = \sum_{(i,j)\in\mathcal{V}\times\mathcal{V}} \|f(i)^T f(j) - A^k_{i,j}\|^2 \tag{7.14}$$

7.4.3 Overlap Similarity

Another measure of similarity is the overlap between node neighborhoods, as shown in Figure 7.9. Suppose nodes i and j share common nodes in the social network of mutual friends. We can then minimize the loss function measuring the overlap between neighborhoods:

$$L = \sum_{(i,j)\mathcal{V}\times\mathcal{V}} \|f(i)^T f(j) - S_{i,j}\|^2 \tag{7.15}$$

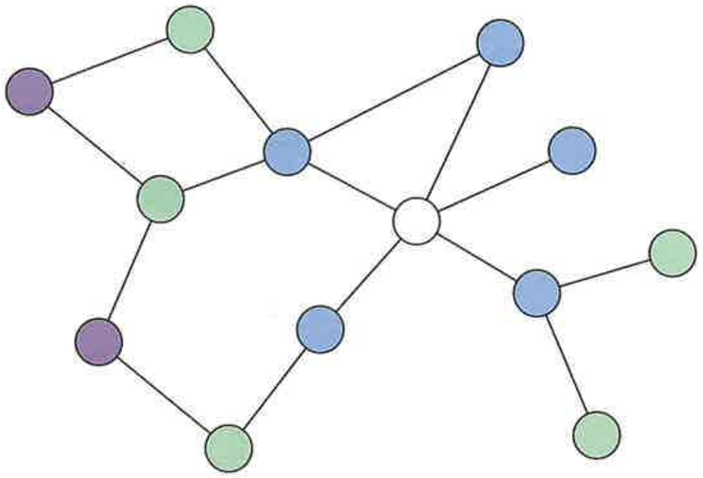

Figure 7.8 Graph neighborhoods. Given a root node shown in white, the 1-hop ring of neighbors is shown in blue, the 2-hop neighbors are shown in green, and the 3-hop neighbors are in purple.

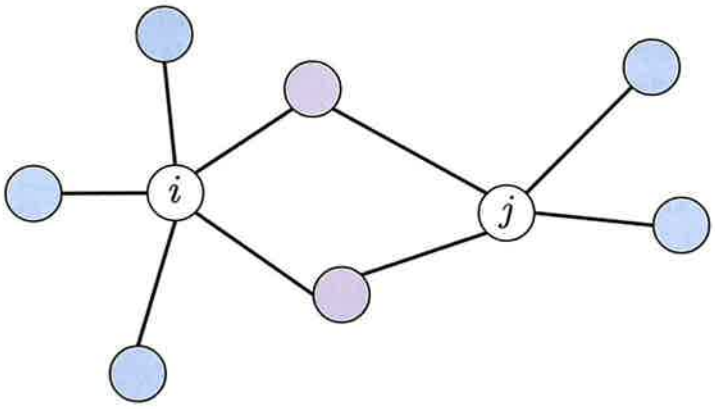

Figure 7.9 Mutual nodes (shown in purple) are the overlap between node neighborhoods of nodes i and j.

where $S_{i,j}$ measures the overlap between the neighbors $\mathcal{N}(i)$ of node i and neighbors $\mathcal{N}(j)$ of node j. The overlap may be measured using the overlap coefficient $\frac{|\mathcal{N}(i) \cap \mathcal{N}(j)|}{\min\{|\mathcal{N}(i)|, |\mathcal{N}(j)|\}}$ or Jaccard similarity $\frac{|\mathcal{N}(i) \cap \mathcal{N}(j)|}{|\mathcal{N}(i) \cup \mathcal{N}(j)|}$.

7.4.4 Random Walk Embedding

We can define an embedding using a random walk from nodes in the graph, as shown in Figure 7.10. A random walk in a graph begins with a node $i \in V$ and

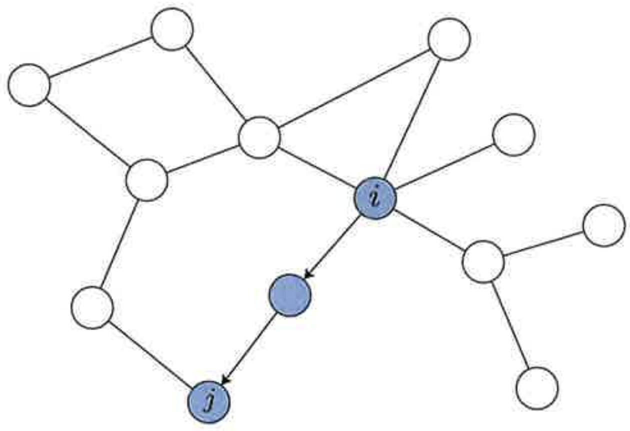

Figure 7.10 Graph random walk (shown in blue) consists of the nodes on a random path starting from node $i \in V$ and ending in node $j \in V$.

repeatedly walks to one of its neighbors $\mathcal{N}(i)$ with probability $\frac{1}{d(i)}$ for t steps until reaching an end at node j on the graph.

Running random walks may start from each graph node i multiple times. We collect all the nodes visited for each node in the walk, and then optimize the embedding defined by:

$$f(i)^T f(j) \propto P(i \text{ and } j \text{ co-occur on the random walk}) = p(i|j) \tag{7.16}$$

which is the probability that we reach node j starting a random walk from node i.

Using the loss function:

$$\mathcal{L} = \sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{N}(i)} -\log p(j|f(i)) \tag{7.17}$$

where $p(j|f(i))$ is given by the softmax:

$$p(j|f(i)) = \frac{\exp(f(i)^T f(j))}{\sum_{j \in \mathcal{N}(i)} \exp(f(i)^T f(j))} \tag{7.18}$$

DeepWalk (Perozzi et al., 2014) uses a skip-gram model of random walks on a graph to classify nodes of a graph. Node2vec (Grover and Leskovec, 2016) uses a random walk on a graph based on both the current node i and the previous nodes that led to node i. Instead of moving from node i to another node with probability $\frac{1}{d(i)}$, node2vec defines a random walk with probability based on the length of the shortest path between the previous node and the next node. LINE (Tang et al., 2015) embeds graph nodes into a low-dimensional space applied to the task of node classification and link prediction.

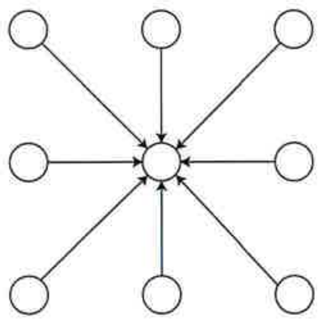

Figure 7.11 Regular graph structure representing neighboring pixels of an image.

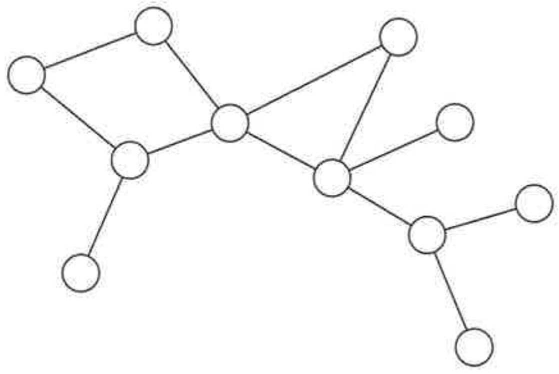

Figure 7.12 Irregular graph structure of real-world graphs representing networks.

7.4.5 Graph Neural Network Properties

A CNN has a regular grid structure, as shown in Figure 7.11, which is suitable for images; however, it is not suitable for real-world graphs, which have irregular structure, as shown in Figure 7.12.

A naive approach for representing a general graph is to concatenate each node's feature vectors to the adjacency matrix and encode each node by the corresponding row of the adjacency matrix and its features. A fully connected network architecture given a node's row in the adjacency matrix and features is unsuitable for graph representation. Having such a vector representation be the input to a fully connected neural network has numerous limitations. In such a naive network, the number of parameters is linear in the size of the graph, the network is dependent on the order of the nodes, and it does not accommodate dynamic graphs. We want to be able to add or remove nodes to real-world graphs, such as the social network, without changing the network architecture. The desired properties of our graph neural network architecture are that the number of parameters is independent of the graph size, scaling to graphs with billions of nodes, that the

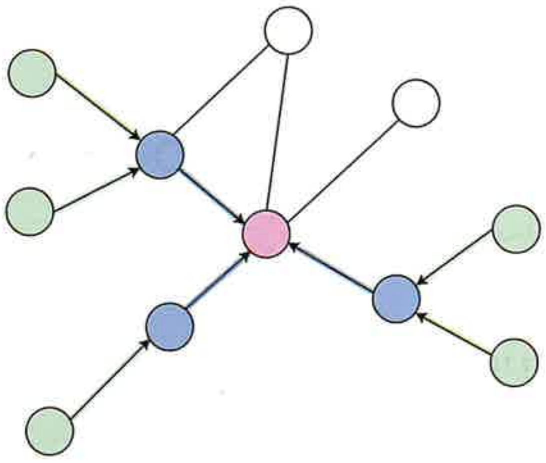

Figure 7.13 Each node aggregates information from its ring of neighbors.

network is invariant to node ordering, that the operations be local depending on neighborhoods, that the model accommodates any graph structure, and that once we learn the properties of one graph, we can transfer them to a new unseen graph.

7.5 Neighborhood Aggregation in Graph Neural Networks

We consider GNNs that take into account neighbors of each node, aggregating information from neighboring nodes similar to breadth-first search (BFS); and the other graph neural network which considers chains from a node, similar to depth-first search (DFS). In the first architecture, we consider each node in the graph and pick up the graph from that node as the root, allowing all other nodes to dangle, building a computation graph where that node is the root. Once we determine the node computation graph, we will propagate and transform information from its neighbors, its neighbors' neighbors, and so on, as shown in Figure 7.13, where each node consists of a vector containing the features of the node.

Most GNNs are based on aggregating information into each node from its neighboring nodes in a layer ℓ and combining that information with the node features in that layer:

$$h_i^\ell = \text{combine}^\ell\{h_i^{\ell-1}, \text{aggregate}^\ell\{h_j^{\ell-1}, j \in \mathcal{N}(i)\}\} \tag{7.19}$$

where h_i^ℓ is the feature representation of node i at layer ℓ.

Consider the graph shown in Figure 7.14. We generate embeddings based on local neighborhoods and aggregate information from neighbors using the neural

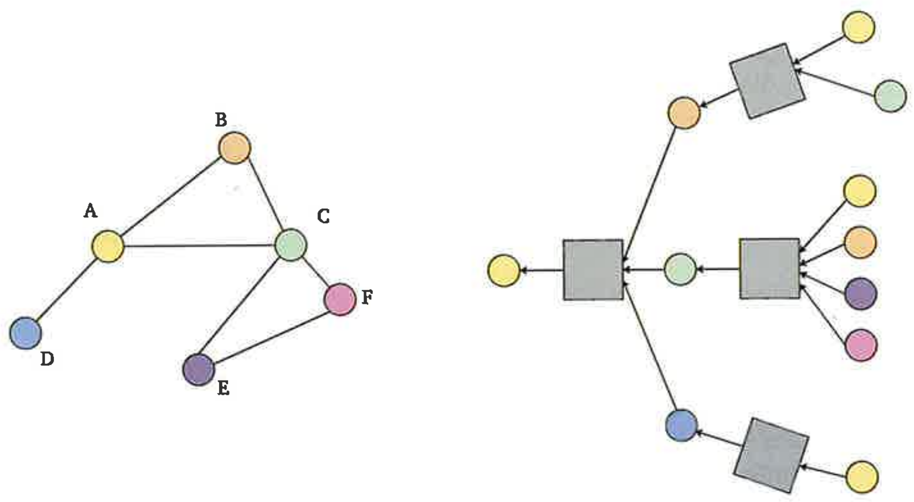

Figure 7.14 A computational graph is constructed for each node, aggregating its neighbors and in turn from each neighbor.

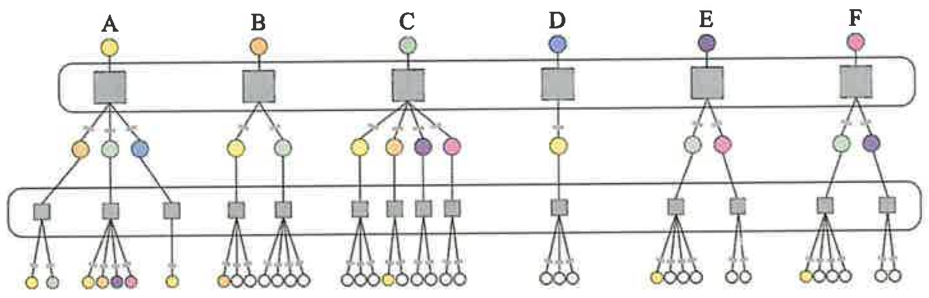

Figure 7.15 Computational graphs starting from each node in the graph, sharing weights between different computational graphs for all 1-hop neighbors, 2-hop neighbors, and 3-hop neighbors. The roots of the computational graphs for each node form the last layer of the GNN, whereas the leaves and their node features form the first layer.

network. Consider node A; its neighbors are nodes B, C, and D, and in turn B's neighbors are nodes A and C, C's neighbors are nodes A, B, E, and F, and D's neighbor is node A.

Next, we consider each node in turn and generate a computation graph for each node where that node is the root. Finally, we will share the aggregation parameters across all nodes for every layer of neighbors, as shown in Figure 7.15.

The gray boxes in each layer in Figure 7.15 represent aggregation parameters, denoted in the special case below by shared matrices W^ℓ and B^ℓ for layer ℓ, such that the aggregation boxes in each layer are identical and shared across nodes.

In summary, the nodes have embeddings at each layer, and the network shares aggregation parameters across all nodes in a layer.

We denote the feature vector of a node i by $h_i^0 = x_i$. A feature vector h_i^ℓ will be an aggregation of the feature vectors $h_j^{\ell-1}$ of the neighbors $j \in \mathcal{N}(i)$ of i and the feature vector $h_i^{\ell-1}$ of the previous layer embedding. An example of a choice of aggregation and combination function is:

$$h_i^\ell = \sigma\left(W^\ell \sum_{j\in\mathcal{N}(i)} \frac{h_j^{\ell-1}}{|\mathcal{N}(i)|} + B^\ell h_i^{\ell-1}\right) \tag{7.20}$$

where h_i^ℓ is the ℓth layer embedding of i, σ is a non-linear activation function and $\sum_{j\in\mathcal{N}(i)} \frac{h_j^{\ell-1}}{|\mathcal{N}(i)|}$ is the average of neighbors in the previous layer embedding. We have two types of weight matrices: W^ℓ is a matrix of weights for neighborhood embeddings, and B^ℓ is a matrix of weights for self-embedding. These matrices are shared for each layer ℓ across all nodes.

7.5.1 Supervised Node Classification Using a GNN

For the task of node classification, given m labeled nodes i with labels y^i we train a GNN by minimizing the objective:

$$\frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(y^i, \hat{y}^i) \tag{7.21}$$

where the predictions $\hat{y}^i$ are the softmax of the node representations at the last layer.

7.6 Graph Neural Network Variants

7.6.1 Graph Convolution Network

A graph convolution network (GCN; Kipf and Welling (2017)) has a similar formulation using a single matrix for both the neighborhood and self-embeddings normalized by the product of square roots of node degrees:

$$\begin{aligned} h_i^\ell &= \sigma\left(W^\ell \sum_{j\in i\cup\mathcal{N}(i)} \frac{\hat{A}_{i,j}}{\sqrt{\hat{d}_j \hat{d}_i}} h_j^{\ell-1}\right) \\ &= \sigma\left(\sum_{j\in\mathcal{N}(i)} \frac{\hat{A}_{i,j}}{\sqrt{\hat{d}_j \hat{d}_i}} W^\ell h_j^{\ell-1} + \frac{1}{\hat{d}_i} W^\ell h_i^{\ell-1}\right) \end{aligned} \tag{7.22}$$

where $\hat{A} = A + I$ is the adjacency matrix including self-loops, $\hat{d}_i$ is the degree in the graph with self-loops, and σ is a non-linear activation function. Aggregation is defined by the term on the left and the combination on the right.

An equivalent formulation (Wu et al., 2019) is given by:

$$H^{\ell+1} = \hat{D}^{-\frac{1}{2}} \hat{A} \hat{D}^{-\frac{1}{2}} H^{\ell} W^{\ell} \tag{7.23}$$

where $\hat{D}_{i,i} = \sum_j \hat{A}_{i,j}$.

7.6.2 GraphSAGE

GraphSAGE (Hamilton et al., 2017) concatenates the neighborhood embedding and self-embedding:

$$h_i^{\ell} = \sigma([W^{\ell} \text{aggregate}(\{h_j^{\ell-1}, j \in \mathcal{N}(i)\}), B^{\ell} h_i^{\ell-1}]) \tag{7.24}$$

The graph neighborhood aggregation function can be the mean, pooling, or an LSTM sequence model:

$$\text{Mean aggregation: } \sum_{j \in \mathcal{N}(i)} \frac{h_j^{\ell-1}}{|\mathcal{N}(i)|} \tag{7.25}$$

$$\text{Pooling: } \gamma(\{Q h_j^{\ell-1}, j \in \mathcal{N}(i)\}) \tag{7.26}$$

$$\text{LSTM: } \text{LSTM}([h_j^{\ell-1}, j \in \pi(\mathcal{N}(i))]) \tag{7.27}$$

and the network learns the parameters for aggregating information.

In the training process, we have an output embedding after L layers $e_i = h_i^L$ and we learn the weight matrices W^{ℓ} for the neighborhood embedding and B^{ℓ} for self-embedding. We define a neighborhood aggregation function and a loss function on embedding and train on a set of nodes generating embeddings for nodes.

This is useful since once we train the GNN, and compute the aggregation parameters, namely the weight matrices, we can generalize to new nodes. We generate a computation graph for a new node and transfer the weight matrices to the new node and compute a forward pass for prediction. In addition, given an entire new graph, we can transfer the aggregation weight matrices computed on one graph to a new graph and compute the forward pass to perform prediction.

7.6.3 Gated Graph Neural Networks

The second architecture, similar to DFS, shares weights across all the layers in each computation graph, instead of sharing weights across neighborhoods. In gated graph neural networks (Li et al., 2016) nodes aggregate messages from neighbors using a neural network, and similar to RNNs parameter sharing is across layers:

$$m_i^{\ell} = W \sum_{j \in \mathcal{N}(i)} h_j^{\ell-1} \tag{7.28}$$

$$h_i^{\ell} = \text{GRU}(h_i^{\ell-1}, m_i^{\ell}) \tag{7.29}$$

7.6.4 Graph Attention Networks

In graph attention networks (GATs) (Veličković et al., 2018) we use attention-based neighborhood aggregation. The attention function adaptively controls the contribution of neighbor j to node i:

$$h_i^\ell = \sigma\left(\sum_{j \in i \cup \mathcal{N}(i)} \alpha_{i,j} W h_j^{\ell-1}\right) \tag{7.30}$$

where $\alpha_{i,j}$ are the attention coefficients that define a distribution over node i and its neighbors $k \in \mathcal{N}(i)$ using the softmax:

$$\alpha_{i,j} = \frac{\exp(e_{i,j})}{\sum_{k \in i \cup \mathcal{N}(i)} \exp(e_{i,k})} \tag{7.31}$$

and $e_{i,j}$ is a function of $h_i^{\ell-1}$ and $h_j^{\ell-1}$:

$$e_{i,j} = \text{ReLU}(v^T(Wh_i^{\ell-1}||Wh_j^{\ell-1}) \tag{7.32}$$

where $||$ is the concatenation operation, and v and W are a learned vector and weight matrix. When using multiple attention heads, h_i^ℓ is the aggregation of multiple contributions, each of the form of Equation 7.30.

7.6.5 Message-Passing Networks

In a similar fashion to using aggregation and combination, a message-passing graph neural network is defined by messages between nodes across edges (aggregation) and node updates (combination):

$$h_i^\ell = \text{update}^\ell(h_i^{\ell-1}, \sum_{j \in \mathcal{N}(i)} \text{message}^\ell(h_i^{\ell-1}, h_j^{\ell-1}, e_{i,j})) \tag{7.33}$$

7.7 Applications

Graph neural networks are used in a wide range of applications, including (1) image retrieval; (2) computer vision for scene understanding (Santoro et al., 2017); (3) computer graphics for 3D shape analysis (Monti et al., 2017) and for learning point-cloud representations (Wang, Sun, Liu, Sarma, Bronstein and Solomon, 2019); (4) social networks for link prediction; (5) recommender systems and few-shot learning (Garcia and Bruna, 2018); (6) combinatorial optimization (Ma et al., 2020); (7) physics for learning the dynamics and interactions of physical objects (Battaglia et al., 2016; Chang et al., 2017; Watters et al., 2017; Sanchez-Gonzalez et al., 2018; Van Steenkiste et al., 2018); (8) chemistry for molecule classification (Duvenaud et al., 2015; Gilmer et al., 2017), defining a graph in which molecules are nodes and edges represent bonds between molecules, and molecule design (Jin et al., 2018); (9) biology for drug discovery, protein function prediction, and protein–protein interactions; (10) for representing computer programs; (11) in natural language processing; (12) for traffic applications such as ride hailing and flight classification; and (13) in stock trading.

7.8 Software Libraries, Benchmarks, and Visualization

PyTorch Geometric (Fey and Lenssen, 2019) is a library for deep learning on graphs in PyTorch. DGL (Wang, Yu, Gan, Zhaoogle, Gai, Ye, Li, Zhou, Huang, Zheng, Lin, Ma, Deng, Guo, Zhang and Huang, 2020) is an optimized library for deep learning on graphs in PyTorch and MXNet. OGB (Liu et al., 2020) is a collection of benchmark datasets, data-loaders and evaluators for deep learning on graphs.

7.9 Summary

Graph neural networks (Hamilton et al., 2017; Kipf and Welling, 2017; Veličković et al., 2018; Xu et al., 2019) are applied to irregular structures such as networks represented by graphs. They commonly share weights across neighborhoods, similar to how CNNs share weights across space and RNNs share weights across time. Graph neural networks are used for three main tasks: (1) predicting properties of particular nodes, (2) predicting edges between nodes, and (3) predicting properties of sub-graphs or entire graphs.