Stanford MS&E 236 / CS 225: Lecture 6
Graph neural networks

Ellen Vitercik*

April 23, 2024

In the last class, we reviewed the basics of graphs. Many discrete optimization problems can be formulated as graph problems. We also saw a handful of famous ("greedy") graph problems for NP-hard problems, including minimum vertex cover, maximum independent set, and maximum cut. In the next two classes, we will see how we can learn better algorithms for these problems using machine learning (perhaps not in the worst case, but for non-worst-case families of graphs). To do so, we will use *graph neural networks (GNNs)*, which will be the focus of this class. Next class, we will focus on how to train GNNs using reinforcement learning (RL), so today, we will also cover the basic building blocks of RL: Markov decision processes.

1 Graph notation

We use $G = (V, E)$ to denote a graph, where $V = \{1, \ldots, n\}$ is a set of nodes/vertices and E is a set of edges/links. The edge between $i, j \in V$ is denoted (i, j). The *neighborhood* of the node $i \in V$ is the set of nodes it is connected to by an edge, denoted as

$$N(i) = \{j : (i, j) \in E\}.$$

2 Graph neural networks

Given a graph or distribution over graphs, there are many different predicted tasks you may wish to perform, which we can coarsely categorize into *node*, *edge*, and *graph* prediction, exemplified as follows.

Node prediction. For example, given a graph that represents a social network, we may wish to predict the political affiliation of each of its users.

Edge prediction. In the same example about social networks, we may wish to predict whether two users—who are not currently friends on the network—know each other in real life.

Graph prediction. For example, given a graph that represents a molecule, we can predict the molecule's properties, like if it is soluble in water.

*These notes are course material and have not undergone formal peer review. Please feel free to send me any typos or comments.

Algorithm 1 Message-passing graph neural network [4, 5]

Input: Graph $G = (V, E)$ with initial node embeddings $\{\boldsymbol{h}_i^{(0)} = \boldsymbol{x}_i : i \in V\}$

1: **for** $\ell \in \{0, 1, \dots, L-1\}$ **do**

2: **for** $i \in V$ **do**

3: Node i aggregates its neighbors' embeddings and computes the vector

$$\boldsymbol{z}_i^{(\ell)} \leftarrow \mathsf{Aggregate}_\ell \left(\left\{ \boldsymbol{h}_j^{(\ell)} : j \in N(i) \right\} \right)$$

4: Node i updates its own embedding using its previous embedding and $\boldsymbol{z}_i^{(\ell)}$:

$$\boldsymbol{h}_i^{(\ell+1)} \leftarrow \mathsf{Update}_\ell \left(\boldsymbol{z}_i^{(\ell)}, \boldsymbol{h}_i^{(\ell)} \right)$$

Output: $\left\{ \boldsymbol{h}_1^{(L)}, \dots, \boldsymbol{h}_n^{(L)} \right\}$

To make these types of predictions, the first task will be to associate each node with some observed data $\boldsymbol{x}_i \in \mathbb{R}^d$. For example, if the graph represents a road network, features could include weather and traffic conditions. Recalling that $|V| = n$, these features form a matrix $X \in \mathbb{R}^{n \times d}$. Everything we discuss today can be extended to edge data as well. In addition to X, another important matrix is the *adjacency matrix* $A \in \{0, 1\}^{n \times n}$, which is defined as

$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{else.} \end{cases}$$

The first, most basic thing we might try to perform predictions on graphs would be to concatenate A and X into one long vector and use that vector as the input of a standard neural network. This approach has two issues. First, the neural network will only work on graphs of a fixed size, and ideally, we would rather not retrain our network for each size graph we encounter (this was also a motivation for pointer networks earlier this quarter). Moreover, if we permute the graph's nodes, we will likely get a completely different answer (this would likely be the case for pointer networks as well).

These shortcomings are addressed with GNNs. At a high level, for each node $i \in V$, a GNN computes an embedding $\boldsymbol{h}_i = f(G, X, i)$. As we will see, individual or pairs of embeddings can be used for node and edge classification. Moreover, the aggregated embeddings $\{\boldsymbol{h}_1, \dots, \boldsymbol{h}_n\}$ can be used for graph classification. As we will see, f will be designed in such a way that it is *permutation invariant* (meaning that graph-level predictions will remain the same even if the nodes are permuted), *permutation equivariant* (meaning that node- and edge-level predictions will remain the same under permutation), and can be applied to graphs of any size.

Message-passing graph neural networks (MPNN) achieve these desiderata by performing message passing over a series of L steps, or *layers*. For each layer $\ell \in [L]$ and node $i \in V$, the GNN computes an embedding $\boldsymbol{h}_i^{(\ell)}$, and the node's final embedding is $\boldsymbol{h}_i^{(L)}$.

The basic form of an MPNN is described by Algorithm 1, with functions $\mathsf{Aggregate}_\ell$ and Update_ℓ defined as follows. The simplest type of aggregation function is a summation, where

$$\mathsf{Aggregate}_\ell \left(\left\{ \boldsymbol{h}_j^{(\ell)} : j \in N(i) \right\} \right) = \sum_{j \in N(i)} \boldsymbol{h}_j^{(\ell)}.$$

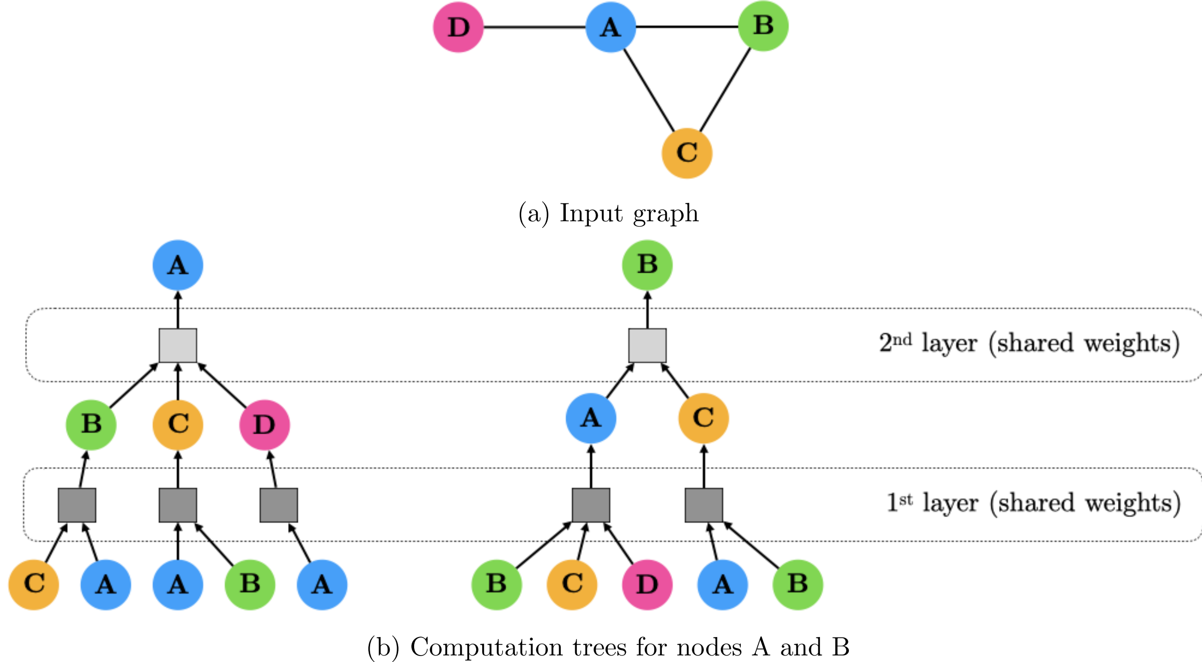

Figure 1: Illustration of Example 2.1. Figure 1a illustrates an input graph and Figure 1b illustrates the nodes involved in computing the final representations $\boldsymbol{h}_A^{(2)}$ and $\boldsymbol{h}_B^{(2)}$ of nodes A and B under a 2-layer GNN.

However, neighborhood sizes can have huge variances, causing summation aggregation functions to have numerical issues. A simple fix is to take the average

$$\mathsf{Aggregate}_\ell\left(\left\{\boldsymbol{h}_j^{(\ell)} : j \in N(i)\right\}\right) = \frac{1}{|N(i)|} \sum_{j \in N(i)} \boldsymbol{h}_j^{(\ell)},$$

and variants thereof [2, 3]. Another option is to take the element-wise maximum, where

$$\boldsymbol{z}_i^{(\ell)}[k] = \max_{j \in N(i)} \boldsymbol{h}_j^{(\ell)}[k]. \tag{1}$$

As we will see, aggregation functions can have a large influence on how well GNNs perform on discrete optimization tasks [6]. Max-aggregation (Equation (1)) often performs well for these tasks (for reasons we have a theoretical understanding of [7, 8]).

The simplest type of update function is a single-layer neural network

$$\mathsf{Update}_\ell\left(\boldsymbol{z}_i^{(\ell)}, \boldsymbol{h}_i^{(\ell)}\right) = \sigma\left(W_{\text{self}}^{(\ell)} \boldsymbol{h}_i^{(\ell)} + W_{\text{neigh}}^{(\ell)} \boldsymbol{z}_i^{(\ell)} + \boldsymbol{b}^{(\ell)}\right),$$

where $W_{\text{self}}^{(\ell)}$, $W_{\text{neigh}}^{(\ell)}$, and $\boldsymbol{b}^{(\ell)}$ are trainable parameters, and σ is a non-linearity (e.g., ReLU). More generally, the update functions are often shallow, feed-forward neural networks.

To better understand GNN computations, it is useful to unroll their computation trees, as illustrated by the following example.

Example 2.1. Figure 1a is a simple input graph, and Figure 1b illustrates the nodes involved in computing the final representations $\boldsymbol{h}_A^{(2)}$ and $\boldsymbol{h}_B^{(2)}$ of nodes A and B under a 2-layer

GNN. In the computation tree for node A in Figure 1b, $\boldsymbol{h}_A^{(2)}$ is computed by combining the embeddings of its neighbors from the previous layers, $\boldsymbol{h}_B^{(1)}$, $\boldsymbol{h}_C^{(1)}$, and $\boldsymbol{h}_D^{(1)}$. The second layer's aggregation and update functions are represented by the light grey box. Meanwhile, node B computed the embedding $\boldsymbol{h}_B^{(1)}$ by combining its neighbors' previous embeddings, $\boldsymbol{h}_A^{(0)}$ and $\boldsymbol{h}_C^{(0)}$. Similarly, node C computed the embedding $\boldsymbol{h}_C^{(1)}$ by combining $\boldsymbol{h}_A^{(0)}$ and $\boldsymbol{h}_B^{(0)}$. The first layer's aggregation and update functions are represented by the dark grey boxes.

A key insight, illustrated by Figure 1b, is that in each layer, the aggregation and update functions are shared across all nodes. This means that we can easily add new nodes and compute their embeddings. Moreover, since these functions do not depend on the nodes' indices, they result in permutation equivariant embeddings. To obtain a permutation-invariant graph embedding, we can apply any permutation-invariant function to the node embeddings, such as a summation

$$\sum_{i=1}^{n} \boldsymbol{h}_i^{(L)}.$$

Finally, given the node embeddings, we can perform node, edge, and graph classification using standard methods, exemplified as follows.

Node classification. Suppose we aim to predict which of C classes a node belongs to. A standard approach is to train weights $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_C \in \mathbb{R}^d$ for each class, and define

$$y_i[k] = \frac{\exp\left(\boldsymbol{w}_k \cdot \boldsymbol{h}_i^{(L)}\right)}{\sum_{j \in [C]} \exp\left(\boldsymbol{w}_j \cdot \boldsymbol{h}_i^{(L)}\right)}$$

to be the probability that the node $i \in V$ belongs to the class $k \in [C]$.

Edge classification. Suppose we aim to predict whether there should be an edge between two nodes $i, j \in V$. A simple approach could be to train the GNN so that $\sigma\left(\boldsymbol{h}_i^{(L)} \cdot \boldsymbol{h}_j^{(L)}\right)$ (with sigmoid function σ) is the predicted probability that the edge exists.

Graph classification. Finally, we can train any ML model g to make predictions

$$\boldsymbol{y} = g\left(\sum_{i \in V} \boldsymbol{h}_i^{(L)}\right).$$

Clearly, we have only scratched the surface of the GNN architecture, but this overview will be sufficient for our purposes. To learn more, I highly recommend checking out the book chapter on GNNs by Bishop and Bishop [1] and Stanford's CS224W class.

References

[1] Christopher M. Bishop and Hugh Bishop. *Deep learning: foundations and concepts.* Springer Cham, 2023.

[2] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[3] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

[4] Christian Merkwirth and Thomas Lengauer. Automatic generation of complementary descriptors with molecular graph networks. *Journal of chemical information and modeling*, 45(5):1159–1168, 2005.

[5] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20 (1):61–80, 2008.

[6] Petar Veličković, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural execution of graph algorithms. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

[7] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about? In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020.

[8] Keyulu Xu, Mozhi Zhang, Jingling Li, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. How neural networks extrapolate: From feedforward to graph neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.