

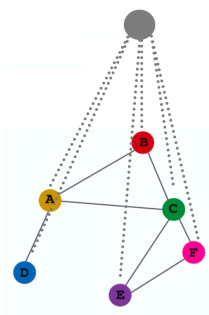
GNN training in practice

Graph manipulation

The raw input graph is often not an optimal computation graph for learning node, edge, or graph embeddings. This may be because the graph is too large, too dense, too sparse, unreliable, noisy, derived from sampling, or other reasons. To address these issues, various *graph manipulation* techniques have been developed to help graph neural networks learn more informative and representative embeddings.

Virtual node(s)

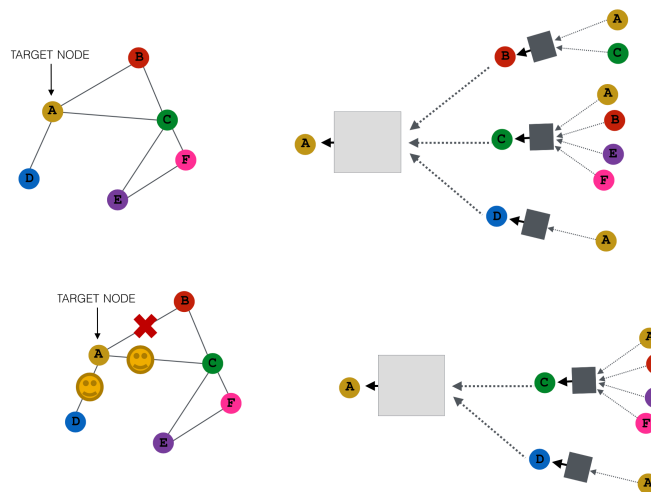
In sparse graphs, message passing in graph neural networks becomes highly inefficient. One way to improve the model's expressivity is to increase the number of message passing layers. However, in small-world networks with small diameters, this quickly leads to the undesirable effect of *over-smoothing*, where node embeddings converge to nearly identical values. A simple yet effective solution is to introduce a *virtual node* (or a hierarchy of such nodes) that connects to all nodes in the graph. This reduces the distance between any two nodes to at most two, enabling even a shallow graph neural network to efficiently exchange information across the entire graph.



Neighbor sampling

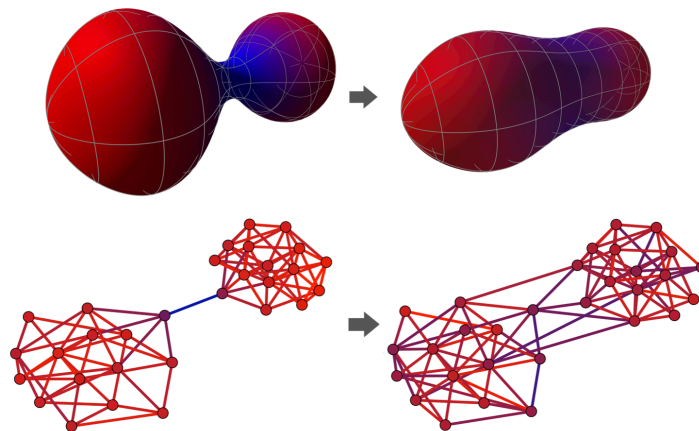
In large, dense graphs, message passing in graph neural networks can become prohibitively expensive. For example, in scale-free networks with high-degree hubs, these nodes must aggregate information from an enormous neighborhood, which is both inefficient and computationally demanding. A simple yet effective technique is to *sample* only a subset of

neighbors for each node during every message passing iteration. In subsequent iterations, different neighbors can be sampled, which not only reduces computational cost but also serves as a form of regularization. Notably, this technique is equivalent to randomly *removing edges* from the graph.



Edge rewiring

In spatial graphs with large diameters, it may be necessary to increase the number of message passing layers to allow information to propagate across distant nodes. However, certain regions of the graph can then act as bottlenecks, preventing effective information flow. This phenomenon is known as the *over-squashing problem*. A simple yet effective solution is to *rewire* some *edges* in the graph to alleviate these bottlenecks. For instance, well-embedded edges can be rewired to run parallel to high-betweenness edges, helping to distribute the information more evenly across the graph. Similar to neighbor sampling, this rewiring can also be viewed as a form of regularization.



Feature augmentation

Nodes, edges, or entire graphs may lack informative features or external metadata that a

graph neural network could leverage. Moreover, certain structural properties cannot be learned by a plain, vanilla graph neural network. To overcome these limitations, various *feature augmentation* techniques can be employed to enrich node and edge representations, incorporate structural information, and ultimately enable the network to learn more expressive and discriminative embeddings.

Let each node i be associated with a d -dimensional feature vector $x_i \in \mathbb{R}^d$. These features may represent, for example, user metadata or profile image in social networks, gene expression profiles in biological networks, or web page content in information networks.

When node features are not available, they must be engineered. Even when features exist but are limited or uninformative, they can be augmented using the approaches described below.

One-hot encoding

One-hot encoding assigns each node i a unique vector x_i consisting of all zeros except for a single one in the i -th position. This enables a graph neural network to distinguish between nodes effectively. However, this approach is limited to transductive learning settings, and since these vectors are n -dimensional, it is practical only for relatively small graphs.

$$x_i = [\dots, 0, 1, 0, \dots] \in \{0, 1\}^n$$

Constant features

A common approach suitable for inductive learning settings is to assign each node i a *constant vector* x_i of all ones. Although this makes all nodes initially identical, a graph neural network can still learn meaningful representations from the graph structure. Since these vectors are d -dimensional, this approach scales to graphs of arbitrary size.

$$x_i = [1, 1, 1, \dots] = [1]^d$$

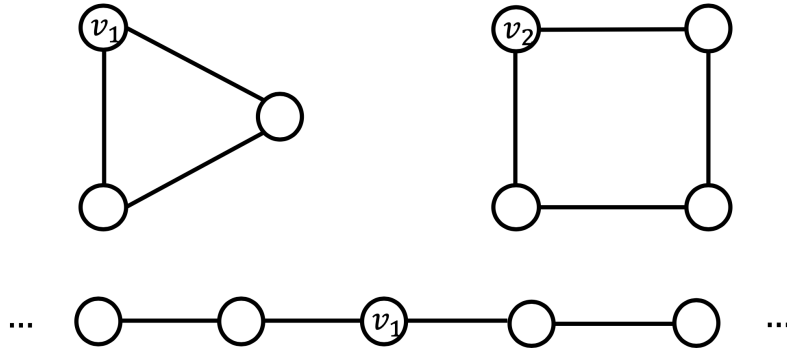
Random features

An alternative approach is to assign each node i a *random vector* x_i , sampled from a uniform distribution $\mathcal{U}(0, 1)$, a normal distribution $\mathcal{N}(0, 1)$, or another. This allows the graph neural network to distinguish between nodes, but since the vectors are random, the network must still rely primarily on the graph structure to learn meaningful representations. If the feature vectors are resampled before each training epoch, the network can be applied in both transductive and inductive learning settings.

$$x_i = [0, 0.1, -0.1, \dots] \in \mathbb{R}^d$$

Closed loops

Certain structural properties cannot be captured by a plain graph neural network. For example, a vanilla network cannot determine the length of the cycles that a node belongs to, although this information is extremely important in some applications.



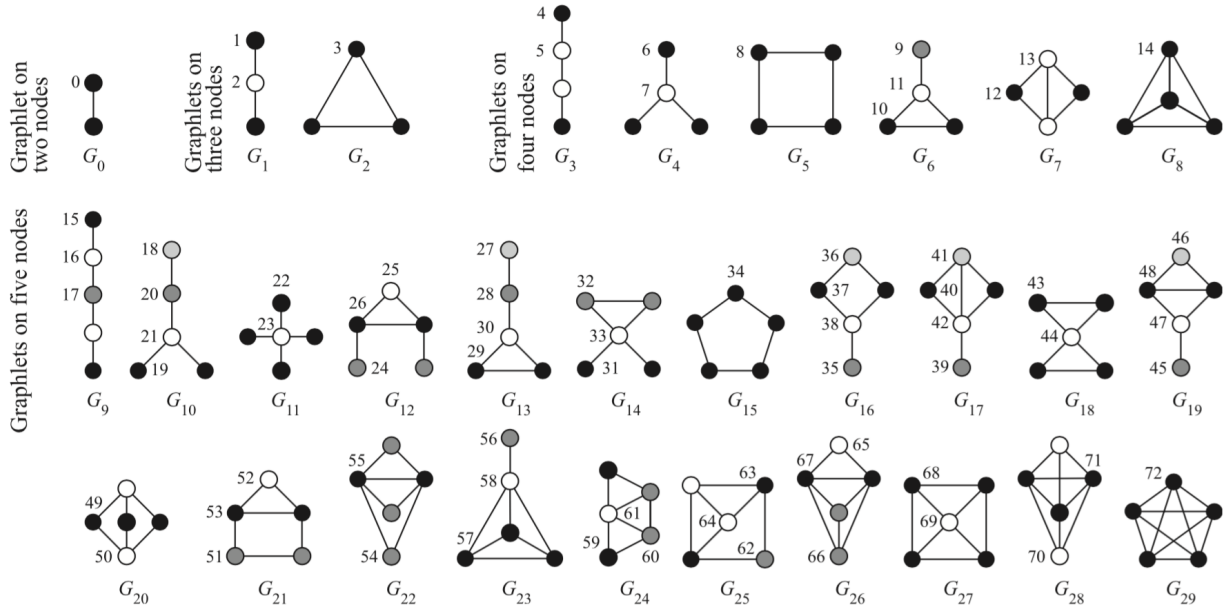
However, these properties can be precomputed using network analysis algorithms, allowing the construction of a feature vector x_i for each node i that the graph neural network can leverage. In particular, let x_{ij} denote the number of *closed loops* of length $j \in \{0, \dots, d-1\}$ that start and end at node i . Then, the feature vector of node i can be written as

$$x_i = [1, l_i, k_i, \Delta_i, \dots] \in \mathbb{N}^d,$$

where l_i is the number of self-loops on node i , k_i is its degree, and Δ_i is the number of triangles that include node i , etc.

Network analysis

Node feature vectors can also be constructed by precomputing various *network analysis* measures that capture the micro-, meso-, or macro-scale structure of the graph. Examples include node positions or centrality measures, distance profiles, graphlet orbits, community memberships, blockmodeling, and others.



These techniques are generally far less computationally demanding than training a graph neural network, yet they can significantly enhance the learning process.

Positional encoding

An effective approach to capture node *positional encoding* also used within transformers is to use the eigenvectors of the graph Laplacian matrix, defined as

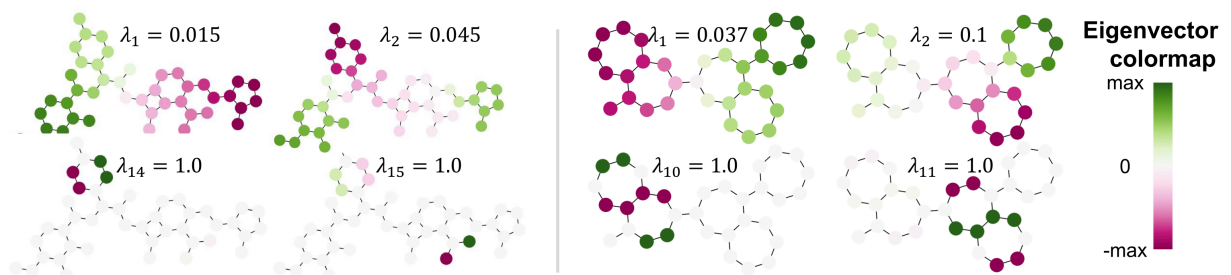
$$L = D - A,$$

where $A \in \{0, 1\}^{n \times n}$ is the adjacency matrix of a connected undirected graph, and $D = \text{diag}(k_1, \dots, k_n) \in \mathbb{N}^{n \times n}$ is a diagonal matrix containing the degrees of the nodes.

Let λ_i denote the i -th eigenvalue of L and v_i the corresponding eigenvector, satisfying

$$Lv_i = \lambda_i v_i.$$

The eigenvectors encode structural information about the graph. In particular, eigenvectors $v_1, v_2, \dots$ associated with small eigenvalues $\lambda_1 \leq \lambda_2 \leq \dots$ capture global structure on the graph, while those corresponding to larger eigenvalues capture local symmetries. (Note that $\lambda_0 = 0$ corresponds to the eigenvector v_0 consisting of all ones.)



The feature vector x_i of node i can thus be defined using the first d eigenvectors as

$$x_i = [v_{1i}, v_{2i}, v_{3i}, \dots] \in \mathbb{R}^d.$$

Prediction heads

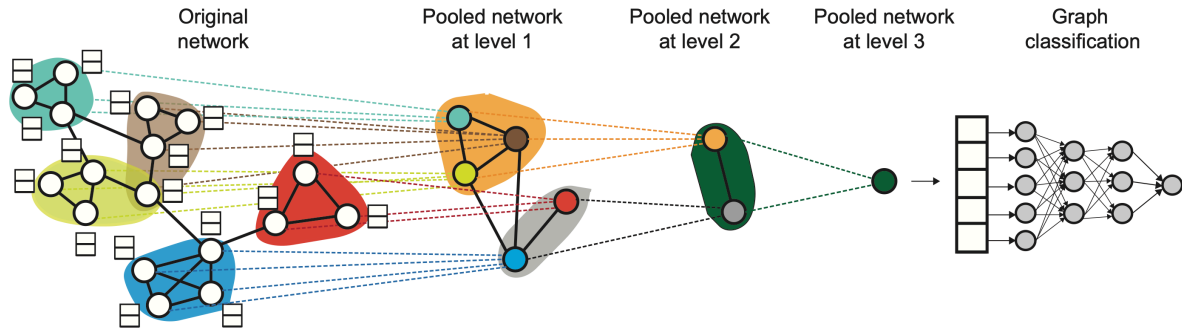
Let z_i denote the embedding of node i from the last layer of the graph neural network.

Node-level tasks

For *node-level tasks*, the embeddings z_i can be directly used for node classification, regression, or clustering. Specifically, the embeddings are fed into an appropriate machine learning model, such as a multilayer perceptron, and both the graph neural network and the downstream model are trained in an end-to-end manner.

Graph-level tasks

For *graph-level tasks*, any permutation-invariant aggregation or pooling of the node embeddings z_i can be used to obtain a graph-level representation. Common aggregation functions include summation, averaging, and element-wise maximum, while more advanced approaches employ differential pooling ([Ying et al., 2018](#)), attention-based pooling ([Lee et al., 2019](#)), or introduce a virtual node that aggregates global information during message passing. The resulting graph-level embedding can then be fed into a downstream model for tasks such as graph classification, regression, or clustering.



Edge-level tasks

For *edge-level tasks* in directed networks, the embeddings z_i and z_j can be simply concatenated. In undirected networks, z_i and z_j must be combined using some symmetric function, such as the dot product, Euclidean distance, summation, averaging, element-wise maximum, element-wise Hadamard product, bilinear form $z_i^T W z_j$, or other, and subsequently used for binary classification, regression, or clustering.

However, for multi-class classification, regression, or clustering over multiple classes $c \in \{1, \dots, C\}$, one can compute class-specific interactions as

$$z_i^T W^{(1)} z_j, z_i^T W^{(2)} z_j, \dots, z_i^T W^{(C)} z_j,$$

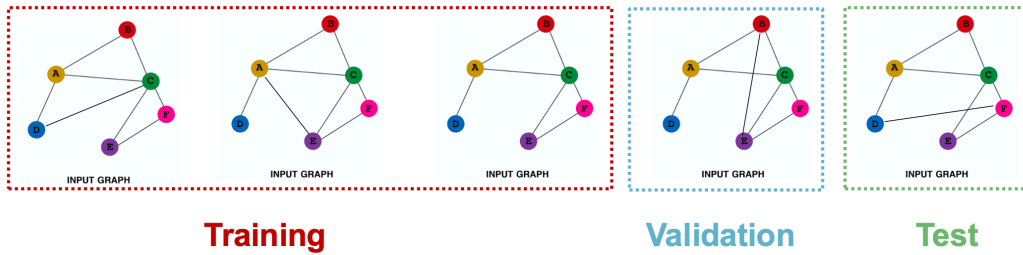
where $W^{(c)} \in \mathbb{R}^{d \times d}$ are learnable weight matrices. The resulting C -dimensional vector can then be used as input to a downstream model for prediction.

Graph splitting

For training graph neural networks, nodes, edges, or entire graphs are divided into training, validation, and test sets. These splits can be generated *randomly*, although standard benchmarks often provide *fixed splits* to ensure consistency and comparability across studies. For temporal datasets, the splits are usually constructed *chronologically*, such that the model is trained on past data and evaluated on its ability to predict future outcomes.

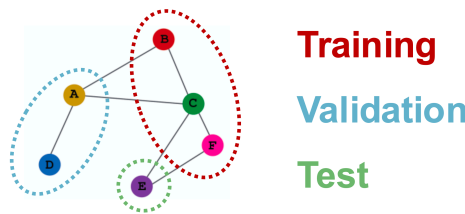
Graph-level tasks

Graph-level tasks are well defined only in inductive learning settings, where training, validation, and testing are performed on distinct, independent graphs.



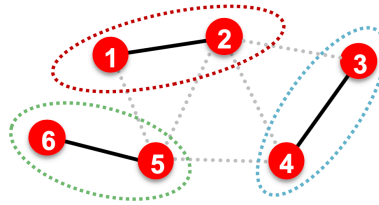
Node-level tasks

In transductive node-level tasks, training, validation, and testing are performed on the same graph. However, each split has access only to the labels of the nodes within its own subset.



In inductive node-level tasks, training, validation, and testing are performed either on different graphs or on disjoint subgraphs of a single graph. It is important to note that randomly partitioning the nodes into disjoint sets may remove many edges and consequently lead to disconnected subgraphs. A more effective approach is to sample each subgraph using some exploration strategy such as breadth-first search traversal or random walks (with restarts).

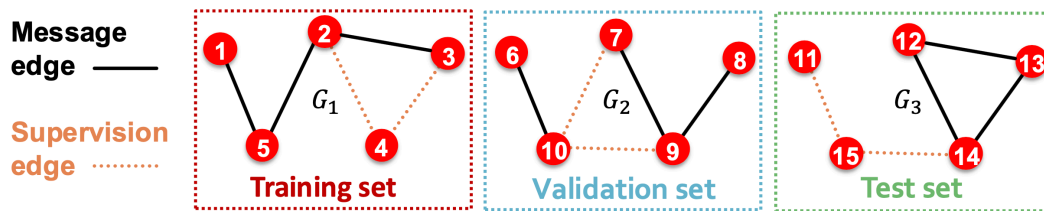
Training
Validation
Test



Edge-level tasks

Consider link prediction as an example of a self-supervised edge-level task. In this setting, we first sample disconnected pairs of nodes to serve as negative examples. Next, we remove an equal number of edges from the graph to serve as positive examples, also referred to as supervision edges. The remaining edges, called message passing edges, are then used to train the graph neural network.

In inductive link prediction, training, validation, and testing are performed on either distinct graphs or disjoint subgraphs of a single graph. Within each graph, edges are divided into message passing edges and supervision edges.



In the more common transductive link prediction setting, training, validation, and testing are performed on the same graph. The edges are typically partitioned into four sets as shown below. During training, the training message passing edges are used to predict the training supervision edges. During validation, both the training message passing and supervision edges are used to predict the validation (supervision) edges. Finally, during testing, all three of these sets are used to predict the test (supervision) edges.

