Part II
Foundations of Graph Neural Networks

Chapter 4
Graph Neural Networks for Node Classification

Jian Tang and Renjie Liao

Abstract Graph Neural Networks are neural architectures specifically designed for graph-structured data, which have been receiving increasing attention recently and applied to different domains and applications. In this chapter, we focus on a fundamental task on graphs: node classification. We will give a detailed definition of node classification and also introduce some classical approaches such as label propagation. Afterwards, we will introduce a few representative architectures of graph neural networks for node classification. We will further point out the main difficulty—the oversmoothing problem—of training deep graph neural networks and present some latest advancement along this direction such as continuous graph neural networks.

4.1 Background and Problem Definition

Graph-structured data (e.g., social networks, the World Wide Web, and protein-protein interaction networks) are ubiquitous in real-world, covering a variety of applications. A fundamental task on graphs is node classification, which tries to classify the nodes into a few predefined categories. For example, in social networks, we want to predict the political bias of each user; in protein-protein interaction networks, we are interested in predicting the function role of each protein; in the World Wide Web, we may have to classify web pages into different semantic categories. To make effective prediction, a critical problem is to have very effective node representations, which largely determine the performance of node classification.

Graph neural networks are neural network architectures specifically designed for learning representations of graph-structured data including learning node represen-

Jian Tang
Mila-Quebec AI Institute, HEC Montreal, e-mail: jian.tang@hec.ca

Renjie Liao
University of Toronto, e-mail: rjliao@cs.toronto.edu

tations of big graphs (e.g., social networks and the World Wide Web) and learning representations of entire graphs (e.g., molecular graphs). In this chapter, we will focus on learning node representations for large-scale graphs and will introduce learning the whole-graph representations in other chapters. A variety of graph neural networks have been proposed (Kipf and Welling, 2017b; Veličković et al, 2018; Gilmer et al, 2017; Xhonneux et al, 2020; Liao et al, 2019b; Kipf and Welling, 2016; Veličković et al, 2019). In this chapter, we will comprehensively revisit existing graph neural networks for node classification including supervised approaches (Sec. 4.2), unsupervised approaches (Sec. 4.3), and a common problem of graph neural networks for node classification—over-smoothing (Sec. 4.4).

Problem Definition. Let us first formally define the problem of learning node representations for node classification with graph neural networks. Let $\mathscr{G} = (\mathscr{V}, \mathscr{E})$ denotes a graph, where $\mathscr{V}$ is the set of nodes and $\mathscr{E}$ is the set of edges. $A \in R^{N \times N}$ represents the adjacency matrix, where N is the total number of nodes, and $X \in R^{N \times C}$ represents the node attribute matrix, where C is the number of features for each node. The goal of graph neural networks is to learn effective node representations (denoted as $H \in R^{N \times F}$, F is the dimension of node representations) by combining the graph structure information and the node attributes, which are further used for node classification.

Table 4.1: Notations used throughout this chapter.

Concept	Notation
Graph	$\mathscr{G} = (\mathscr{V}, \mathscr{E})$
Adjacency matrix	$A \in \mathbb{R}^{N \times N}$
Node attributes	$X \in \mathbb{R}^{N \times C}$
Total number of GNN layers	K
Node representations at the k-th layer	$H^k \in \mathbb{R}^{N \times F}$, $k \in \{1, 2, \cdots, K\}$

4.2 Supervised Graph Neural Networks

In this section, we revisit several representative methods of graph neural networks for node classification. We will focus on the supervised methods and introduce the unsupervised methods in the next section. We will start by introducing a general framework of graph neural networks and then introduce different variants under this framework.

4.2.1 General Framework of Graph Neural Networks

The essential idea of graph neural networks is to iteratively update the node representations by combining the representations of their neighbors and their own representations. In this section, we introduce a general framework of graph neural networks in (Xu et al, 2019d). Starting from the initial node representation $H^0 = X$, in each layer we have two important functions:

- **AGGREGATE**, which tries to aggregate the information from the neighbors of each node;
- **COMBINE**, which tries to update the node representations by combining the aggregated information from neighbors with the current node representations.

Mathematically, we can define the general framework of graph neural networks as follows:

Initialization: $H^0 = X$
For $k = 1, 2, \cdots, K$,

$$a_v^k = \textbf{AGGREGATE}^k\{H_u^{k-1} : u \in N(v)\} \tag{4.1}$$

$$H_v^k = \textbf{COMBINE}^k\{H_v^{k-1}, a_v^k\}, \tag{4.2}$$

where $N(v)$ is the set of neighbors for the v-th node. The node representations H^K in the last layer can be treated as the final node representations.

Once we have the node representations, they can be used for downstream tasks. Take the node classification as an example, the label of node v (denoted as $\hat{y}_v$) can be predicted through a Softmax function, i.e.,

$$\hat{y}_v = \text{Softmax}(WH_v^\top), \tag{4.3}$$

where $W \in \mathbb{R}^{|\mathscr{L}| \times F}$, $|\mathscr{L}|$ is the number of labels in the output space.

Given a set of labeled nodes, the whole model can be trained by minimizing the following loss function:

$$O = \frac{1}{n_l} \sum_{i=1}^{n_l} \text{loss}(\hat{y}_i, y_i), \tag{4.4}$$

where y_i is the ground truth label of node i, n_l is the number of labeled nodes, $loss(\cdot, \cdot)$ is a loss function such as cross-entropy loss function. The whole neural networks can be optimized by minimizing the objective function O with backpropagation.

Above we present a general framework of graph neural networks. Next, we will introduce a few most representative instantiations or variants of graph neural networks in the literature.

4.2.2 Graph Convolutional Networks

We will start from the graph convolutional networks (GCN) (Kipf and Welling, 2017b), which is now the most popular graph neural network architecture due to its simplicity and effectiveness in a variety of tasks and applications. Specifically, the node representations in each layer is updated according to the following propagation rule:

$$H^{k+1} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^k W^k). \tag{4.5}$$

$\tilde{A} = A + \mathbf{I}$ is the adjacency matrix of the given undirected graph $\mathscr{G}$ with self-connections, which allows to incorporate the node features itself when updating the node representations. $\mathbf{I} \in \mathbb{R}^{N\times N}$ is the identity matrix. $\tilde{D}$ is a diagonal matrix with $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$. $\sigma(\cdot)$ is an activation function such as ReLU and Tanh. The ReLU active function is widely used, which is defined as $\text{ReLU}(x) = \max(0,x)$. $W^k \in \mathbb{R}^{F\times F'}$ (F,F' are the dimensions of node representations in the k-th, (k+1)-th layer respectively) is a laywise linear transformation matrix, which will be trained during the optimization.

We can further dissect equation equation 4.5 and understand the AGGREGATE and COMBINE function defined in GCN. For a node i, the node updating equation can be reformulated as below:

$$H_i^k = \sigma\Big(\sum_{j\in\{N(i)\cup i\}} \frac{\tilde{A}_{ij}}{\sqrt{\tilde{D}_{ii}\tilde{D}_{jj}}} H_j^{k-1} W^k\Big) \tag{4.6}$$

$$H_i^k = \sigma\Big(\sum_{j\in N(i)} \frac{A_{ij}}{\sqrt{\tilde{D}_{ii}\tilde{D}_{jj}}} H_j^{k-1} W^k + \frac{1}{\tilde{D}_i} H_i^{k-1} W^k\Big) \tag{4.7}$$

In the Equation equation 4.7, we can see that the AGGREGATE function is defined as the weighted average of the neighbor node representations. The weight of the neighbor j is determined by the weight of the edge between i and j (i.e. A_{ij} normalized by the degrees of the two nodes). The COMBINE function is defined as the summation of the aggregated messages and the node representation itself, in which the node representation is normalized by its own degree.

Connections with Spectral Graph Convolutions. Next, we discuss the connections between GCNs and traditional spectral filters defined on graphs (Defferrard et al, 2016). The spectral convolutions on graphs can be defined as a multiplication of a node-wise signal $\mathbf{x} \in \mathbb{R}^N$ with a convolutional filter $g_\theta = \text{diag}(\theta)$ ($\theta \in \mathbb{R}^N$ is the parameter of the filter) in the Fourier domain. Mathematically,

$$g_\theta \star \mathbf{x} = U g_\theta U^T \mathbf{x}. \tag{4.8}$$

U represents the matrix of the eigenvectors of the normalized graph Laplacian matrix $L = I_N - D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$. $L = U\Lambda U^T$, Λ is a diagonal matrix of eigenvalues, and $U^T\mathbf{x}$ is the graph Fourier transform of the input signal $\mathbf{x}$. In practice, g_θ can be understood as a function of the eigenvalues of the normalized graph Laplacian matrix L (i.e. $g_\theta(\Lambda)$). In practice, directly calculating Eqn. equation 4.8 is very computationally expensive, which is quadratic to the number of nodes N. According to (Hammond et al, 2011), this problem can be circumvented by approximating the function $g_\theta(\Lambda)$ with a truncated expansion of Chebyshev polynomials $T_k(x)$ up to K^{th} order:

$$g_{\theta'}(\Lambda) = \sum_{k=0}^{K} \theta'_k T_k(\tilde{\Lambda}), \tag{4.9}$$

where $\tilde{\Lambda} = \frac{2}{\lambda_{max}}\Lambda - \mathbf{I}$, and λ_{max} is the largest eigenvalue of L. $\theta' \in \mathbb{R}^K$ is the vector of Chebyshev coefficients. $T_k(x)$ are Chebyshev polynomials which are recursively defined as $T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$, with $T_0(x) = 1$ and $T_1(x) = x$. By combining Eqn. equation 4.9 and Eqn. equation 4.8, the convolution of a signal x with a filter $g_{\theta'}$ can be reformulated as below:

$$g_{\theta'} \star \mathbf{x} = \sum_{k=0}^{K} \theta'_k T_k(\tilde{L})\mathbf{x}, \tag{4.10}$$

where $\tilde{L} = \frac{2}{\lambda_{max}}L - \mathbf{I}$. From this equation, we can see that each node only depends on the information within the K^{th}-order neighborhood. The overall complexity of evaluating Eqn. equation 4.10 is $\mathcal{O}(|\mathcal{E}|)$ (i.e. linear to the number of edges in the original graph $\mathcal{G}$), which is very efficient.

To define a neural network based on graph convolutions, one can stack multiple convolution layers defined according to Eqn. equation 4.10 with each layer followed by a nonlinear transformation. At each layer, instead of being limited to the explicit parametrization by the Chebyshev polynomials defined in Eqn. equation 4.10, the authors of GCNs proposed to limit the number of convolutions to $K = 1$ at each layer. By doing this, at each layer, it only defines a linear function over the graph Laplacian matrix L. However, by stacking multiple such layers, we are still capable of covering a rich class of convolution filter functions on graphs. Intuitively, such a model is capable of alleviating the problem of overfitting local neighborhood structures for graphs whose node degree distribution has a high variance such as social networks, the World Wide Web, and citation networks.

At each layer, we can further approximate $\lambda_{max} \approx 2$, which could be accommodated by the neural network parameters during training. Based on al these simplifications, we have

$$g_{\theta'} \star \mathbf{x} \approx \theta'_0\mathbf{x} + \theta'_1\mathbf{x}(L - I_N)\mathbf{x} = \theta'_0\mathbf{x} - \theta'_1 D^{-\frac{1}{2}}AD^{-\frac{1}{2}}, \tag{4.11}$$

where θ'_0 and θ'_1 are too free parameters, which could be shared over the entire graph. In practice, we can further reduce the number of parameters, which allows to

reduce overfitting and meanwhile minimize the number of operations per layer. As a result, the following expression can be further obtained:

$$g_\theta \star \mathbf{x} \approx \theta(\mathbf{I} + D^{-\frac{1}{2}}AD^{-\frac{1}{2}})\mathbf{x}, \tag{4.12}$$

where $\theta = \theta'_0 = -\theta'_1$. One potential issue is the matrix $I_N + D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$, whose eigenvalues lie in the interval of $[0,2]$. In a deep graph convolutional neural network, repeated application of the above function will likely lead to exploding or vanishing gradients, yielding numerical instabilites. As a result, we can further renormalize this matrix by converting $\mathbf{I} + D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ to $\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$, where $\tilde{A} = A + \mathbf{I}$, and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$.

In the above, we only consider the case that there is only one feature channel and one filter. This can be easily generalized to an input signal with C channels $X \in \mathbb{R}^{N\times C}$ and F filters (or number of hidden units) as follows:

$$H = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}XW, \tag{4.13}$$

where $W \in R^{C\times F}$ is a matrix of filter parameters. H is the convolved signal matrix.

4.2.3 Graph Attention Networks

In GCNs, for a target node i, the importance of a neighbor j is determined by the weight of their edge A_{ij} (normalized by their node degrees). However, in practice, the input graph may be noisy. The edge weights may not be able to reflect the true strength between two nodes. As a result, a more principled approach would be to automatically learn the importance of each neighbor. Graph Attention Networks (a.k.a. GAT(Veličković et al, 2018)) is built on this idea and try to learn the importance of each neighbor based on the **Attention** mechanism (Bahdanau et al, 2015; Vaswani et al, 2017). Attention mechanism has been wide used in a variety of tasks in natural language understanding (e.g. machine translation and question answering) and computer vision (e.g. visual question answering and image captioning). Next, we will introduce how attention is used in graph neural networks.

Graph Attention Layer. The graph attention layer defines how to transfer the hidden node representations at layer $k-1$ (denoted as $H^{k-1} \in \mathbb{R}^{N\times F}$) to the new node representations $H^k \in \mathbb{R}^{N\times F'}$. In order to guarantee sufficient expressive power to transform the lower-level node representations to higher-level node representations, a shared linear transformation is applied to every node, denoted as $W \in \mathbb{R}^{F\times F'}$. Afterwards, self-attention is defined on the nodes, which measures the attention coefficients for any pair of nodes through a shared attentional mechanism $a : \mathbb{R}^{F'} \times \mathbb{R}^{F'} \to R$

$$e_{ij} = a(WH_i^{k-1}, WH_j^{k-1}). \tag{4.14}$$

e_{ij} indicates the relationship strength between node i and j. Note in this subsection we use H_i^{k-1} to represent a column-wise vector instead of a row-wise vector. For each node, we can theoretically allow it to attend to every other node on the graph, which however will ignore the graph structural information. A more reasonable solution would be only to attend to the neighbors for each node. In practice, the first-order neighbors are only used (including the node itself). And to make the coefficients comparable across different nodes, the attention coefficients are usually normalized with the softmax function:

$$\alpha_{ij} = \text{Softmax}_j(\{e_{ij}\}) = \frac{\exp(e_{ij})}{\sum_{l \in N(i)} \exp(e_{il})}. \tag{4.15}$$

We can see that for a node i, α_{ij} essentially defines a multinomial distribution over the neighbors, which can also be interpreted as the transition probability from node i to each of its neighbors.

In the work by Veličković et al (2018), the attention mechanism a is defined as a single-layer feedforward neural network including a linear transformation with the weight vector $W_2 \in \mathbb{R}^{1 \times 2F'}$) and a LeakyReLU nonlinear activation function (with negative input slope $\alpha = 0.2$). More specifically, we can calculate the attention coefficients with the following architecture:

$$\alpha_{ij} = \frac{\exp(\text{LeakyReLU}(W_2[WH_i^{k-1}||WH_j^{k-1}]))}{\sum_{l \in N(i)} \exp(\text{LeakyReLU}(W_2[WH_i^{k-1}||WH_l^{k-1}]))}, \tag{4.16}$$

where $||$ represents the operation of concatenating two vectors. The new node representation is a linear combination of the neighboring node representations with the weights determined by the attention coefficients (with a potential nonlinear transformation), i.e.

$$H_i^k = \sigma\left(\sum_{j \in N(i)} \alpha_{ij} W H_j^{k-1}\right). \tag{4.17}$$

Multi-head Attention.

In practice, instead of only using one single attention mechanism, *multi-head attention* can be used, each of which determines a different similarity function over the nodes. For each attention head, we can independently obtain a new node representation according to Eqn. equation 4.17. The final node representation will be a concatenation of the node representations learned by different attention heads. Mathematically, we have

$$H_i^k = \Big\|_{t=1}^{T} \sigma\left(\sum_{j \in N(i)} \alpha_{ij}^t W^t H_j^{k-1}\right), \tag{4.18}$$

where T is the total number of attention heads, α_{ij}^t is the attention coefficient calculated from the t-th attention head, W^t is the linear transformation matrix of the t-th attention head.

One thing that mentioned in the paper by Veličković et al (2018) is that in the final layer, when trying to combine the node representations from different attention heads, instead of using the operation concatenation, other pooling techniques could be used, e.g. simply taking the average node representations from different attention heads.

$$H_i^k = \sigma\left(\frac{1}{T}\sum_{t=1}^{T}\sum_{j\in N(i)}\alpha_{ij}^t W^t H_j^{k-1}\right). \tag{4.19}$$

4.2.4 Neural Message Passing Networks

Another very popular graph neural network architecture is the Neural Message Passing Network (MPNN) (Gilmer et al, 2017), which is originally proposed for learning molecular graph representations. However, MPNN is actually very general, provides a general framework of graph neural networks, and could be used for the task of node classification as well. The essential idea of MPNN is formulating existing graph neural networks as a general framework of neural message passing among nodes. In MPNNs, there are two important functions including **Message** and **Updating** function:

$$m_i^k = \sum_{i\in N(j)} M_k(H_i^{k-1}, H_j^{k-1}, e_{ij}), \tag{4.20}$$

$$H_i^k = U_k(H_i^{k-1}, m_i^k). \tag{4.21}$$

$M_k(\cdot,\cdot,\cdot)$ defines the message between node i and j in the k-th layer, which depends on the two node representations and the information of their edge. U_k is the node updating function in the k-th layer which combines the aggregated messages from the neighbors and the node representation itself. We can see that the MPNN framework is very similar to the general framework we introduced in Section 4.2.1. The **AGGREGATE** function defined here is simply a summation of all the messages from the neighbors. The **COMBINE** function is the same as the node **Updating** function.

4.2.5 Continuous Graph Neural Networks

The above graph neural networks iteratively update the node representations with different kinds of graph convolutional layers. Essentially, these approaches model

the discrete dynamics of node representations with GNNs. Xhonneux et al (2020) proposed the continuous graph neural networks (CGNNs), which generalizes existing graph neural networks with discrete dynamics to continuous settings, i.e., trying to model the continuous dynamics of node representations. The key idea is how to characterize the continuous dynamics of node representations, i.e. the derivatives of node representation w.r.t. time. The CGNN model is inspired by the diffusion-based models on graphs such as PageRank and epidemic models on social networks. The derivatives of the node representations are defined as a combination of the node representation itself, the representations of its neighbors, and the initial status of the nodes. Specifically, two different variants of node dynamics are introduced. The first model assumes that different dimensions of node presentations (a.k.a. feature channels) are independent; the second model is more flexible, which allows different feature channels to interact with each other. Next, we give a detailed introduction to each of the two models.

Note: in this part, instead of using the original adjacency matrix A, we use the following regularized matrix for characterizing the graph structure:

$$A := \frac{\alpha}{2}\left(I + D^{-\frac{1}{2}}AD^{-\frac{1}{2}}\right), \tag{4.22}$$

where $\alpha \in (0,1)$ is a hyperparameter. D is the degree matrix of the original adjacency matrix A. With the new regularized adjacency matrix A, the eigenvalues of A will lie in the interval $[0,\alpha]$, which will make A^k converges to 0 when we increase the power of k.

Model 1: Independent Feature Channels. As different nodes in a graph are interconnected, a natural solution to model the dynamic of each feature channel should be taking the graph structure into consideration, which allows the information to propagate across different nodes. We are motivated by existing diffusion-based methods on graphs such as PageRank (Page et al, 1999) and label propagation (Zhou et al, 2004), which defines the discrete propagation of node representations (or signals on nodes) with the following step-wise propagation equations:

$$H^{k+1} = AH^k + H^0, \tag{4.23}$$

where $H^0 = X$ or the output of an encoder on the input feature X. Intuitively, at each step, the new node representation is a linear combination of its neighboring node representations as well as the initial node features. Such a mechanism allows to model the information propagation on the graph without forgetting the initial node features. We can unroll Eqn. equation 4.23 and explicitly derive the node representations at the k-th step:

$$H^k = \left(\sum_{i=0}^{k} A^i\right) H^0 = (A - \mathbf{I})^{-1}(A^{k+1} - \mathbf{I})H^0. \tag{4.24}$$

As the above equation effectively models the discrete dynamics of node representations, the CGNN model further extended it to the continuous setting, which

replaces the discrete time step k to a continuous variable $t \in \mathbb{R}_0^+$. Specifically, it has been shown that Eqn. equation 4.24 is a discretization of the following ordinary differential equation (ODE):

$$\frac{dH^t}{dt} = \log A H^t + X, \tag{4.25}$$

with the initial value $H^0 = (\log A)^{-1}(A-I)X$, where X is the initial node features or the output of an encoder applied to it. We do not provide the proof here. More details can be referred to the original paper (Xhonneux et al, 2020). In Eqn. equation 4.25, as $\log A$ is intractable to compute in practice, it is approximated with the first-order of the Taylor expansion, i.e. $\log A \approx A - I$. By integrating all these information, we have the following ODE equation:

$$\frac{dH^t}{dt} = (A - \mathbf{I})H^t + X, \tag{4.26}$$

with the initial value $H^0 = X$, which is the first variant of the CGNN model.

The CGNN model is actually very intuitive, which has a nice connection with traditional epidemic model, which aims at studying the dynamics of infection in a population. For the epidemic model, it usually assumes that the infection of people will be affected by three different factors including the infection from neighbors, the natural recovery, and the natural characteristics of people. If we treat H^t as the number of people infected at time t, then these three factors can be naturally modeled by the three terms in Eqn. equation 4.26: AH^t for the infection from neighbors, $-H^t$ for the natural recovery, and the last one X for the natural characteristics of people.

Model 2: Modeling the Interaction of Feature Channels. The above model assumes different node feature channels are independent with each other, which is a very strong assumption and limits the capacity of the model. Inspired by the success of a linear variant of graph neural networks (i.e., Simple GCN (Wu et al, 2019a)), a more powerful discrete node dynamic model is proposed, which allows different feature channels to interact with each other as,

$$H^{k+1} = AH^k W + H^0, \tag{4.27}$$

where $W \in \mathbb{R}^{F \times F}$ is a weight matrix used to model the interactions between different feature channels. Similarly, we can also extend the above discrete dynamics into continuous case, yielding the following equation:

$$\frac{dH^t}{dt} = (A - \mathbf{I})H^t + H^t(W - \mathbf{I}) + X, \tag{4.28}$$

with the initial value being $H^0 = X$. This is the second variant of CGNN with trainable weights. Similar form of ODEs defined in Eqn. equation 4.28 has been studied in the literature of control theory, which is known as Sylvester differential equation (Locatelli and Sieniutycz, 2002). The two matrices $A - \mathbf{I}$ and $W - \mathbf{I}$ characterize

the natural solution of the system while X is the information provided to the system to drive the system into the desired state.

Discussion. The proposed continuous graph neural networks (CGNN) has multiple nice properties: (1) Recent work has shown that if we increase the number of layers K in the discrete graph neural networks, the learned node representations tend to have the problem of over-smoothing (will introduce in detail later) and hence lose the power of expressiveness. On the contrary, the continuous graph neural networks are able to train very deep graph neural networks and are experimentally robust to arbitrarily chosen integration time; (2) For some of the tasks on graphs, it is critical to model the long-range dependency between nodes, which requires training deep GNNs. Existing discrete GNNs fail to train very deep GNNs due to the over-smoothing problem. The CGNNs are able to effectively model the long-range dependency between nodes thanks to the stability w.r.t. time. (3) The hyperparameter α is very important, which controls the rate of diffusion. Specifically, it controls the rate at which high-order powers of regularized matrix A vanishes. In the work proposed by (Xhonneux et al, 2020), the authors proposed to learn a different value of α for each node, which hence allows to choose the best diffusion rates for different nodes.

4.2.6 Multi-Scale Spectral Graph Convolutional Networks

Recall the one-layer graph convolution operator used in GCNs (Kipf and Welling, 2017b) $H = LHW$, where $L = D^{-\frac{1}{2}}\tilde{A}D^{-\frac{1}{2}}$. Here we drop the superscript of the layer index to avoid the clash with the notation of the matrix power. There are two main issues with this simple graph convolution formulation. First, one such graph convolutional layer would only propagate information from any node to its nearest neighbors, *i.e.*, neighboring nodes that are one-hop away. If one would like to propagate information to M-hop away neighbors, one has to either stack M graph convolutional layers or compute the graph convolution with M-th power of the graph Laplacian, *i.e.*, $H = \sigma(L^M HW)$. When M is large, the solution of stacking layers would make the whole GCN model very deep, thus causing problems in learning like the vanishing gradient. This is similar to what people experienced in training very deep feedforward neural networks. For the matrix power solution, naively computing the M-th power of the graph Laplacian is also very costly (*e.g.*, the time complexity is $O(N^{3(M-1)})$ for graphs with N nodes). Second, there are no learnable parameters in GCNs associated with the graph Laplacian L (corresponding to the connectivities/structures). The only learnable parameter W is a linear transform applied to every node simultaneously which is not aware of the structures. Note that we typically associate learnable weights on edges while applying the convolution applied to regular graphs like grids (*e.g.*, applying 2D convolution to images). This would greatly improve the expressiveness of the model. However, it is not clear that how

one can add learnable parameters to the graph Laplacian L since its size varies from graph to graph.

Algorithm 1 : Lanczos Algorithm

Input: S, x, M, ε
Initialization: $\beta_0 = 0$, $\mathbf{q}_0 = 0$, and $\mathbf{q}_1 = \mathbf{x}/\|\mathbf{x}\|$
For $j = 1, 2, \ldots, K$:
    $\mathbf{z} = S\mathbf{q}_j$
    $\gamma_j = \mathbf{q}_j^\top \mathbf{z}$
    $\mathbf{z} = \mathbf{z} - \gamma_j \mathbf{q}_j - \beta_{j-1}\mathbf{q}_{j-1}$
    $\beta_j = \|\mathbf{z}\|_2$
    If $\beta_j < \varepsilon$, quit
    $\mathbf{q}_{j+1} = \mathbf{z}/\beta_j$

$Q = [\mathbf{q}_1, \mathbf{q}_2, \cdots, \mathbf{q}_M]$
Construct T following Eq. (4.29)
Eigen decomposition $T = BRB^\top$
Return $V = QB$ and R. =0

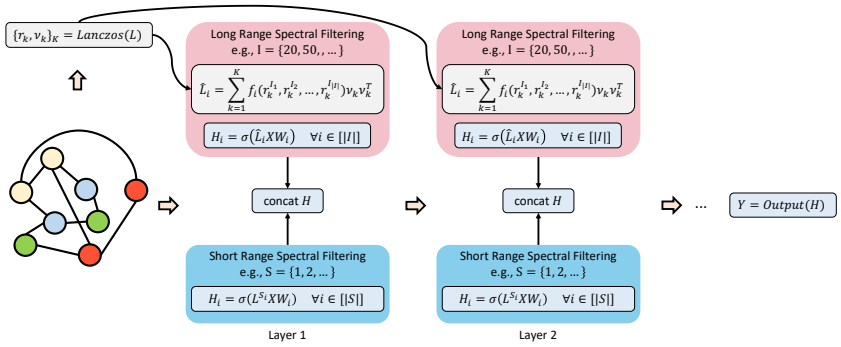

Fig. 4.1: The inference procedure of Lanczos Networks. The approximated top eigenvalues $\{r_k\}$ and eigenvectors $\{\mathbf{v}_k\}$ are computed by the Lanczos algorithm. Note that this step is only needed once per graph. The long range/scale (top blocks) graph convolutions are efficiently computed by the low-rank approximation of the graph Laplacian. One can control the ranges (*i.e.*, the exponent of eigenvalues) as hyperparameters. Learnable spectral filters are applied to the approximated top eigenvalues $\{r_k\}$. The short range/scale (bottom blocks) graph convolution is the same as GCNs. Adapted from Figure 1 of (Liao et al, 2019b).

To overcome these two problems, authors propose Lanczos Networks in (Liao et al, 2019b). Given the graph Laplacian matrix L[1] and node features X, one first

[1] Here we assume a symmetric graph Laplacian matrix. If it is non-symmetric (*e.g.*, for directed graphs), one can resort to the Arnoldi algorithm.

uses the M-step Lanczos algorithm (Lanczos, 1950) (listed in Alg. 1) to compute an orthogonal matrix Q and a symmetric tridiagonal matrix T, such that $Q^\top LQ = T$. We denote $Q = [\mathbf{q}_1, \cdots, \mathbf{q}_M]$ where column vector $\mathbf{q}_i$ is the i-th Lanczos vector. Note that M could be much smaller than the number of nodes N. T is illustrated as below,

$$T = \begin{bmatrix} \gamma_1 & \beta_1 & & \\ \beta_1 & \ddots & \ddots & \\ & \ddots & \ddots & \beta_{M-1} \\ & & \beta_{M-1} & \gamma_M \end{bmatrix}. \tag{4.29}$$

After obtaining the tridiagonal matrix T, we can compute the Ritz values and Ritz vectors which approximate the top eigenvalues and eigenvectors of L by diagonalizing the matrix T as $T = BRB^\top$, where the $K \times K$ diagonal matrix R contains the Ritz values and $B \in \mathbb{R}^{K \times K}$ is an orthogonal matrix. Here top means ranking the eigenvalues by their magnitudes in a descending order. This can be implemented via the general eigendecomposition or some fast decomposition methods specialized for tridiagonal matrices. Now we have a low rank approximation of the graph Laplacian matrix $L \approx VRV^\top$, where $V = QB$. Denoting the column vectors of V as $\{\mathbf{v}_1, \cdots, \mathbf{v}_M\}$, we can compute multi-scale graph convolution as

$$\begin{aligned} H &= \hat{L}HW \\ \hat{L} &= \sum_{m=1}^{M} f_\theta(r_m^{I_1}, r_m^{I_2}, \cdots, r_m^{I_u}) v_m v_m^\top, \end{aligned} \tag{4.30}$$

where $\{I_1, \cdots, I_u\}$ is the set of scale/range parameters which determine how many hops (or how far) one would like to propagate the information over the graph. For example, one could easily set $\{I_1 = 50, I_2 = 100\}$ ($u = 2$ in this case) to consider the situations of propagating 50 and 100 steps respectively. Note that one only needs to compute the scalar power rather than the original matrix power. The overall complexity of the Lanczos algorithm in our context is $O(MN^2)$ which makes the whole algorithm much more efficient than naively computing the matrix power. Moreover, f_θ is a learnable spectral filter parameterized by θ and can be applied to graphs with varying sizes since we decouple the graph size and the input size of f_θ. f_θ directly acts on the graph Laplacian and greatly improves the expressiveness of the model.

Although Lanczos algorithm provides an efficient way to approximately compute arbitrary powers of the graph Laplacian, it is still a low-rank approximation which may lose certain information (*e.g.*, the high frequency one). To alleviate the problem, one can further do vanilla graph convolution with small scale parameters like $H = L^S HW$ where S could be small integers like 2 or 3. The resultant representation can be concatenated with the one obtained from the longer scale/range graph convolution in Eq. (4.30). Relying on the above design, one could add nonlinearities and stack multiple such layers to build a deep graph convolutional network (namely Lanczos Networks) just like GCNs. The overall inference procedure of Lanczos Networks is shown in Fig. 4.1. This method demonstrates strong empirical

performances on a wide variety of tasks/benchmarks including molecular property prediction in quantum chemistry and document classification in citation networks. It just requires slight modifications to the implementation of the original GCNs. Nevertheless, if the input graph is extremely large (*e.g.*, some large social network), the Lanczos algorithm itself would be a computational bottleneck. How to improve this model in such a problem context would be an open question.

Here we only introduce a few representative architectures of graph neural networks for node classification. There are also many other well-known architectures including gated graph neural networks (Li et al, 2016b)—which is mainly designed for output sequences—and GraphSAGE (Hamilton et al, 2017b)—which is mainly designed for inductive setting of node classification.

4.3 Unsupervised Graph Neural Networks

In this section, we review a few representative GNN-based methods for unsupervised learning on graph-structured data, including variational graph auto-encoders (Kipf and Welling, 2016) and deep graph infomax (Veličković et al, 2019).

4.3.1 Variational Graph Auto-Encoders

Following variational auto-encoders (VAEs) (Kingma and Welling, 2014; Rezende et al, 2014) , variational graph auto-encoders (VGAEs) (Kipf and Welling, 2016) provide a framework for unsupervised learning on graph-structured data. In the following, we first review the model and then discuss its advantages and disadvantages.

4.3.1.1 Problem Setup

Suppose we are given an undirected graph $\mathscr{G} = (\mathscr{V}, \mathscr{E})$ with N nodes. Each node is associated with a node feature/attribute vector. We compactly denote all node features as a matrix $X \in \mathbb{R}^{N \times C}$. The adjacency matrix of the graph is A. We assume self-loops are added to the orignal graph $\mathscr{G}$ so that the diagonal entries of A are 1. This is a convention in graph convolutional networks (GCNs) (Kipf and Welling, 2017b) and makes the model consider a node's old representation while updating its new representation. We also assume each node is associated with a latent variable (the collection of all latent variables is again compactly denoted as a matrix $Z \in \mathbb{R}^{N \times F}$). We are interested in inferring the latent variables of nodes in the graph and decoding the edges.

4.3.1.2 Model

Similar to VAEs, the VGAE model consists of an encoder $q_\phi(Z|A,X)$, a decoder $p_\theta(A|Z)$, and a prior $p(Z)$.

Encoder The goal of the encoder is to learn a distribution of latent variables associated with each node conditioning on the node features X and the adjacency matrix A. We could instantiate $q_\phi(Z|A,X)$ as a graph neural network where the learnable parameters are ϕ. In particular, VGAE assumes an node-independent encoder as below,

$$q_\phi(Z|X,A) = \prod_{i=1}^{N} q_\phi(\mathbf{z}_i|X,A) \tag{4.31}$$

$$q_\phi(\mathbf{z}_i|X,A) = \mathcal{N}(\mathbf{z}_i|\mu_i, \text{diag}(\sigma_i^2)) \tag{4.32}$$

$$\boldsymbol{\mu}, \boldsymbol{\sigma} = \text{GCN}_\phi(X,A) \tag{4.33}$$

where $\mathbf{z}_i$, $\boldsymbol{\mu}_i$, and $\boldsymbol{\sigma}_i$ are the i-th rows of the matrices Z, $\boldsymbol{\mu}$, and $\boldsymbol{\sigma}$ respectively. Basically, we assume a multivariate Normal distribution with the diagonal covariance as the variational approximated distribution of the latent vector per node (*i.e.*, $\mathbf{z}_i$). The mean and diagonal covariance are predict by the encoder network, *i.e.*, a GCN as described in Section 4.2.2. For example, the original paper uses a two-layer GCN as follows,

$$\boldsymbol{\mu} = \tilde{A}HW_{\boldsymbol{\mu}} \tag{4.34}$$

$$\boldsymbol{\sigma} = \tilde{A}HW_{\boldsymbol{\sigma}} \tag{4.35}$$

$$H = \text{ReLU}(\tilde{A}XW_0), \tag{4.36}$$

where $\tilde{A} = D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ is the symmetrically normalized adjacency matrix and D is the degree matrix. Learnable parameters are thus $\phi = [W_{\boldsymbol{\mu}}, W_{\boldsymbol{\sigma}}, W_0]$.

Decoder Given sampled latent variables, the decoder aims at predicting the connectivities among nodes. The original paper adopts a simple dot-product based predictor as below,

$$p(A|Z) = \prod_{i=1}^{N}\prod_{j=1}^{N} p(A_{ij}|\mathbf{z}_i,\mathbf{z}_j) \tag{4.37}$$

$$p(A_{ij}|\mathbf{z}_i,\mathbf{z}_j) = \boldsymbol{\sigma}(\mathbf{z}_i^\top \mathbf{z}_j), \tag{4.38}$$

where A_{ij} denotes the (i,j)-th element and $\boldsymbol{\sigma}(\cdot)$ is the logistic sigmoid function. This decoder again assumes conditional independence among all possible edges for tractability. Note that there are no learnable parameters associated with this decoder. The only way to improve the performance of the decoder is to learn good latent representations.

Prior The prior distributions over the latent variables are simply set to independent zero-mean Gaussians with unit variances,

$$p(Z) = \prod_{i=1}^{N} \mathcal{N}(\mathbf{z}_i | \mathbf{0}, \mathbf{I}). \tag{4.39}$$

This prior is fixed throughout the learning as what typical VAEs do.

Objective & Learning To learn the encoder and the decoder, one typically maximize the evidence lower bound (ELBO) as in VAEs,

$$\mathscr{L}_{\text{ELBO}} = \mathbb{E}_{q_\phi(Z|X,A)} [\log p(A|Z)] - \text{KL}(q_\phi(Z|X,A) \| p(Z)), \tag{4.40}$$

where $\text{KL}(q\|p)$ is the Kullback-Leibler divergence between distributions q and p. Note that we can not directly maximize the log likelihood since the introduction of latent variables Z induces a high-dimensional integral which is intractable. We instead maximize the ELBO in Eq. (4.40) which is a lower bound of the log likelihood. However, the first expectation term is again intractable. One often resorts to the Monte Carlo estimation by sampling a few Z from the encoder $q_\phi(Z|X,A)$ and evaluating the term using the samples. To maximize the objective, one can perform stochastic gradient descent along with the reparameterization trick (Kingma and Welling, 2014). Note that the reparameterization trick is necessary since we need to back-propagate through the sampling in the aforementioned Monte Carlo estimation term to compute the gradient w.r.t. the parameters of the encoder.

4.3.1.3 Discussion

The VGAE model is popular in the literature mainly due to its simplicity and good empirical performances. For example, since there are no learnable parameters for the prior and the decoder, the model is quite light-weight and the learning process is fast. Moreover, the VGAE model is versatile in way that once we learned a good encoder, *i.e.*, good latent representations, we can use them for predicting edges (, link prediction), node attributes, and so on. On the other side, VGAE model is still limited in the following ways. First, it can not serve as a good generative model for graphs as what VAEs do for images since the decoder is not learnable. One could simply design some learnable decoder. However, it is not clear that the goal of learning good latent representations and generating graphs with good qualities are always well-aligned. More exploration along this direction would be fruitful. Second, the independence assumption is exploited for both the encoder and the decoder which might be very limited. More structural dependence (*e.g.*, auto-regressive) would be desirable to improve the model capacity. Third, as discussed in the original paper, the prior may be potentially a poor choice. At last, for link prediction in practice, one may need to add the weighting of edges vs. non-edges in the decoder term and carefully tune it since graphs may be very sparse.

4.3.2 Deep Graph Infomax

Following Mutual Information Neural Estimation (MINE) (Belghazi et al, 2018) and Deep Infomax (Hjelm et al, 2018), Deep Graph Infomax (Veličković et al, 2019) is an unsupervised learning framework that learns graph representations via the principle of mutual information maximization.

4.3.2.1 Problem Setup

Following the original paper, we will explain the model under the single-graph setup, *i.e.*, the node feature matrix X and the graph adjacency matrix A of a single graph are provided as input. Extensions to other problem setups like transductive and inductive learning settings will be discussed in Section 4.3.2.3. The goal is to learn the node representations in an unsupervised way. After node representations are learned, one can apply some simple linear (logistic regression) classifier on top of the representations to perform supervised tasks like node classification.

4.3.2.2 Model

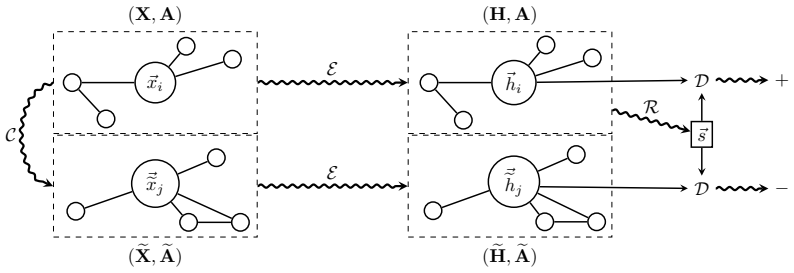

Fig. 4.2: The overall process of Deep Graph Infomax. The top path shows how the positive sample is processed, whereas the bottom shows process corresponding to the negative sample. Note that the graph representation is shared for both positive and negative samples. Subgraphs of positive and negative samples do not necessarily need to be different. Adapted from Figure 1 of (Veličković et al, 2019).

The main idea of the model is to maximize the local mutual information between a node representation (capturing local graph information) and the graph representation (capturing global graph information). By doing so, the learned node representation should capture the global graph information as much as possible. Let us denote the graph encoder as ε which could be any GNN discussed before, *e.g.*, a two-layer GCN. We can obtain all node representations as $H = \varepsilon(X,A)$ where the

representation $\mathbf{h}_i$ of any node i should contain some local information near node i. Specifically, k-layer GCN should be able to leverage node information that is k-hop away. To get the global graph information, one could use a readout layer/function to process all node representations, *i.e.*, $\mathbf{s} = \mathscr{R}(H)$, where the readout function $\mathscr{R}$ could be some learnable pooling function or simply an average operator.

Objective Given the local node representation $\mathbf{h}_i$ and the global graph representation $\mathbf{s}$, the natural next-step is to compute their mutual information. Recall the definition of mutual information is as follows,

$$\mathrm{MI}(\mathbf{h},\mathbf{s}) = \int\int p(\mathbf{h},\mathbf{s}) \log\left(\frac{p(\mathbf{h},\mathbf{s})}{p(\mathbf{h})p(\mathbf{s})}\right) \mathrm{d}\mathbf{h}\mathrm{d}\mathbf{s}. \tag{4.41}$$

However, maximizing the local mutual information alone is not enough to learn useful representations as shown in (Hjelm et al, 2018). To develop a more practical objective, authors in (Veličković et al, 2019) instead use a noise-contrastive type objective following Deep Infomax (Hjelm et al, 2018),

$$\mathscr{L} = \frac{1}{N+M}\left(\sum_{i=1}^{N}\mathbb{E}_{(X,A)}\left[\log \mathscr{D}(\mathbf{h}_i,\mathbf{s})\right] + \sum_{j=1}^{M}\mathbb{E}_{(\tilde{X},\tilde{A})}\left[\log\left(1-\mathscr{D}(\tilde{\mathbf{h}}_j,\mathbf{s})\right)\right]\right). \tag{4.42}$$

where $\mathscr{D}$ is a binary classifier which takes both the node representation $\mathbf{h}_i$ and the graph representation $\mathbf{s}$ as input and predicts whether the pair $(\mathbf{h}_i,\mathbf{s})$ comes from the joint distribution $p(\mathbf{h},\mathbf{s})$ (positive class) or the product of marginals $p(\mathbf{h}_i)p(\mathbf{s})$ (negative class). We denote $\tilde{\mathbf{h}}_j$ as the j-th node representation from the negative sample. The numbers of positive and negative samples are N and M respectively. We will explain how to draw positive and negative samples shortly. The overall objective is thus the negative binary cross-entropy for training a probabilistic classifier. Note that this objective is the same type of distance as used in generative adversarial networks (GANs) (Goodfellow et al, 2014b) which is shown to be proportional to the Jensen-Shannon divergence (Goodfellow et al, 2014b; Nowozin et al, 2016). As verified by (Hjelm et al, 2018), maximizing the Jensen-Shannon divergence based mutual information estimator behaves similarly (*i.e.*, they have an approximately monotonic relationship) to directly maximizing the mutual information. Therefore, maximizing the objective in Eq. (4.42) is expected to maximize the mutual information. Moreover, the freedom of choosing negative samples makes the method more likely to learn useful representations than maximizing the vanilla mutual information.

Negative Sampling To generate the positive samples, one can directly sample a few nodes from the graph to construct the pairs $(\mathbf{h}_i,\mathbf{s})$. For negative samples, one can generate them via corrupting the original graph data, denoting as $(\tilde{X},\tilde{A}) = \mathscr{C}(X,A)$. In practice, one can choose various forms of this corruption function $\mathscr{C}$. For example, authors in (Veličković et al, 2019) suggest to keep the adjacency matrix to be the same and corrupt the node feature X by row-wise shuffling. Other possibilities of the corruption function include randomly sampling subgraphs and applying Dropout (Srivastava et al, 2014) to node features.

Once positive and negative samples were collected, one can learn the representations via maximizing the objective in Eq. (4.42). We summarize the training process of Deep Graph Infomax as follows:

1. Sample negative examples via the corruption function $(\tilde{X},\tilde{A}) \sim \mathscr{C}(X,A)$.
2. Compute node representations of positive samples $H = \{\mathbf{h}_1,\cdots,\mathbf{h}_N\} = \varepsilon(X,A)$.
3. Compute node representations of negative samples $\tilde{H} = \{\tilde{\mathbf{h}}_1,\cdots,\tilde{\mathbf{h}}_M\} = \varepsilon(\tilde{X},\tilde{A})$.
4. Compute graph representation via the readout function $\mathbf{s} = \mathscr{R}(H)$.
5. Update parameters of ε, $\mathscr{D}$, and $\mathscr{R}$ via gradient ascent to maximize Eq. (4.42).

4.3.2.3 Discussion

Deep Graph Infomax is an efficient unsupervised representation learning method for graph-structured data. The implementation of the encoder, the readout, and the binary cross-entropy type of loss are all straightforward. The mini-batch training does not necessarily need to store the whole graph since the readout can be applied to a set of subgraphs as well. Therefore, the method is memory-efficient. Also, the processing of positive and negative samples can be done in parallel. Moreover, authors prove that minimizing the cross-entropy type of classification error can be used to maximize the mutual information under certain conditions, *e.g.*, the readout function is injective and input feature comes from a finite set. However, the choice of the corruption function seems to be crucial to ensure satisfying empirical performances. There seems no such a universally good corruption function. One needs to do trial–and-error to obtain a proper one depending on the task/dataset.

4.4 Over-smoothing Problem

Training deep graph neural networks by stacking multiple layers of graph neural networks usually yields inferior results, which is a common problem observed in many different graph neural network architectures. This is mainly due to the problem of over-smoothing, which is first explicitly studied in (Li et al, 2018b). (Li et al, 2018b) showed that the graph convolutional network (Kipf and Welling, 2017b) is a special case of Laplacian smoothing:

$$Y = (1-\gamma I)X + \gamma \tilde{A}_{rw} X, \tag{4.43}$$

where $\tilde{A}_{rw} = \tilde{D}^{-1}\tilde{A}$, which defines the transitional probabilities between nodes on graphs. The GCN corresponds to a special case of Laplacian smoothing with $\gamma = 1$ and the symmetric matrix $\tilde{A}_{sym} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$ is used. The Laplacian smoothing will push nodes belonging to the same clusters to take similar representations, which are beneficial for downstream tasks such as node classification. However, when the GCNs go deep, the node representations suffer from the problem of over-smoothing, i.e., all the nodes will have similar representations. As a result, the performance on

downstream tasks suffer as well. This phenomenon has later been pointed out by a few other later work as well such as (Zhao and Akoglu, 2019; Li et al, 2018b; Xu et al, 2018a; Li et al, 2019c; Rong et al, 2020b).

PairNorm (Zhao and Akoglu, 2019). Next, we will present a method called PairNorm for alleviating the problem of over-smoothing when GNNs go deep. The essential idea of PairNorm is to keep the total pairwise squared distance (TPSD) of node representations unchanged, which is the same as that of the original node feature X. Let $\tilde{H}$ be the output of the node representations by the graph convolution, which will be the input of PairNorm, and $\hat{H}$ is the output of PairNorm. The goal of PairNorm is to normalize the $\tilde{H}$ such that after normalization $\text{TPSD}(\hat{H}) = \text{TPSD}(X)$. In other words,

$$\sum_{(i,j)\in\mathscr{E}} ||\hat{H}_i - \hat{H}_j||^2 + \sum_{(i,j)\notin\mathscr{E}} ||\hat{H}_i - \hat{H}_j||^2 = \sum_{(i,j)\in\mathscr{E}} ||X_i - X_j||^2 + \sum_{(i,j)\notin\mathscr{E}} ||X_i - X_j||^2. \tag{4.44}$$

In practice, instead of measuring the TPSD of original node features X, (Zhao and Akoglu, 2019) proposed to maintain a constant TPSD value C across different graph convolutional layers. The value C will be a hyperparameter of the PairNorm layer, which can be tuned for each data set. To normalize $\tilde{H}$ into $\hat{H}$ with a constant TPSD, we must first calculate the $\text{TPSD}(\tilde{H})$. However, this is very computationally expensive, which is quadratic to the number of nodes N. We notice that the TPSD can be reformulated as:

$$\text{TPSD}(\tilde{H}) = \sum_{(i,j)\in[N]} ||\tilde{H}_i - \tilde{H}_j||^2 = 2N^2 \left(\frac{1}{N}\sum_{i=1}^{N} ||\tilde{H}_i||_2^2 - ||\frac{1}{N}\sum_{i=1}^{N} \tilde{H}_i||_2^2 \right) \tag{4.45}$$

We can further simply the above equation by substracting the row-wise mean from each $\tilde{H}_i$. In other words, $\tilde{H}_i^c = \tilde{H}_i - \frac{1}{N}\sum_{i=1}^{N} \tilde{H}_i$, which denotes the centered representation. A nice property of centering the node representation is that it will not change the TPSD and meanwhile push the second term $||\frac{1}{N}\sum_{i=1}^{N} \tilde{H}_i||_2^2$ to zero. As a result, we have

$$\text{TPSD}(\tilde{H}) = \text{TPSD}(\tilde{H}^c) = 2N||\tilde{H}^c||_F^2. \tag{4.46}$$

To summarize, the proposed PairNorm can be divded into two steps: center-and-scale,

$$\tilde{H}_i^c = \tilde{H}_i - \frac{1}{N}\sum_{i=1}^{N} \tilde{H}_i \qquad \text{(Center)} \tag{4.47}$$

$$\hat{H}_i = s \cdot \frac{\tilde{H}_i^c}{\sqrt{\frac{1}{N}\sum_{i=1}^{N} ||\tilde{H}_i^c||_2^2}} = s\sqrt{N} \cdot \frac{\tilde{H}_i^c}{\sqrt{||\tilde{H}^c||_F^2}} \qquad \text{(Scale)}, \tag{4.48}$$

where s is a hyperparameter determining C. At the end, we have

$$\text{TPSD}(\hat{H}) = 2N||\hat{H}||_F^2 = 2N\sum_i ||s \cdot \frac{\tilde{H}_i^c}{\sqrt{\frac{1}{N}\sum_{i=1}^N ||\tilde{H}_i^c||_2^2}}||_2^2 = 2N^2 s^2 \tag{4.49}$$

which is a constant across different graph convolutional layers.

4.5 Summary

In this chapter, we give a comprehensive introduction to different architectures of graph neural networks for node classification. These neural networks can be generally classified into two categories including supervised and unsupervised approaches. For supervised approaches, the main difference among different architectures lie in how to propagate messages between nodes, how to aggregate the messages from neighbors, and how to combine the aggregated messages from neighbors with the node representation itself. For the unsupervised approaches, the main difference comes from designing the objective function. We also discuss a common problem of training deep graph neural networks—over-smoothing, and introduce a method to tackle it. In the future, promising directions on graph neural networks include theoretical analysis for understanding the behaviors of graph neural networks, and applying them to a variety of fields and domains such as recommender systems, knowledge graphs, drug and material discovery, computer vision, and natural language understanding.

Editor's Notes: Node classification task is one of the most important tasks in Graph Neural Networks. The node representation learning techniques introduced in this chapter are the corner stone for all other tasks for the rest of the book, including graph classification task (Chapter 9), link prediction (Chapter 10), graph generation task (Chapter 11), and so on. Familiar with the learning methodologies and design principles of node representation learning is the key to deeply understanding other fundamental research directions like Theoretical analysis (Chapter 5), Scalability (Chapter 6), Explainability (Chapter 7), and Adversarial Robustness (Chapter 8).