5
Graph Neural Networks

5.1 Introduction

Graph Neural Networks (GNNs) are a set of methods that aim to apply deep neural networks to graph-structured data. The classical deep neural networks cannot be easily generalized to graph-structured data as the graph structure is not a regular grid. The investigation of graph neural networks can date back to the early of the 21st century, when the first GNN model (Scarselli et al., 2005, 2008) was proposed for both node- and graph-focused tasks. When deep learning techniques gained enormous popularity in many areas, such as computer vision and natural language processing, researchers started to dedicate more efforts to this research area.

Graph neural networks can be viewed as a process of representation learning on graphs. Node-focused tasks target on learning good features for each node such that node-focused tasks can be facilitated. For graph-focused tasks, they aim to learn representative features for the entire graph where learning node features is typically an intermediate step. The process of learning node features usually leverages both the input node features and the graph structure. More specifically, this process can be summarized as follows:

$$\mathbf{F}^{(\text{of})} = h(\mathbf{A}, \mathbf{F}^{(\text{if})}) \tag{5.1}$$

where $\mathbf{A} \in \mathbb{R}^{N \times N}$ denotes the adjacency matrix of the graph with N nodes (i.e., the graph structure) and $\mathbf{F}^{(\text{if})} \in \mathbb{R}^{N \times d_{\text{if}}}$ and $\mathbf{F}^{(\text{of})} \in \mathbb{R}^{N \times d_{\text{of}}}$ denote the input and output feature matrices where d_{if} and d_{of} are their dimensions, respectively. In this book, we generally refer to the process that takes node features and graph structure as input and outputs a new set of node features as *graph filtering* operation. The superscripts (or subscripts) "if" and "of" in Eq. (5.1) denote *the input of filtering* and *the output of filtering*, receptively. Correspondingly, the operator $h(\cdot, \cdot)$ is called as a *graph filter*. Figure 5.1 illustrates a typical

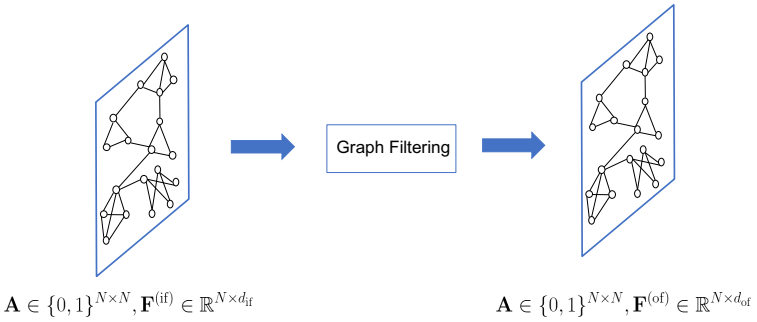

Figure 5.1 Graph filtering operation

graph filtering process where the filtering operation does not change the graph structure, but it only refines the node features.

For node-focused tasks, the graph filtering operation is sufficient, and multiple graph filtering operations are usually stacked consecutively to generate final node features. However, other operations are necessary for graph-focused tasks to generate the features for the entire graph from the node features. Similar to the classical CNNs, pooling operations are proposed to summarize node features to generate graph-level features. The classical CNNs are applied to data residing on regular grids. However, the graph structure is irregular, which calls for dedicated pooling operations in graph neural networks. Intuitively, pooling operations on graphs should utilize the graph structure information to guide the pooling process. In fact, pooling operations often take a graph as input and then produce a coarsened graph with fewer nodes. Thus, the key to pooling operations is to generate the graph structure (or the adjacency matrix) and the node features for the coarsened graph. In general, as shown in Figure 5.2, a graph pooling operation can be described as follows:

$$\mathbf{A}^{(\text{op})}, \mathbf{F}^{(\text{op})} = \text{pool}(\mathbf{A}^{(\text{ip})}, \mathbf{F}^{(\text{ip})}) \tag{5.2}$$

where $\mathbf{A}^{(\text{ip})} \in \mathbb{R}^{N_{\text{ip}}\times N_{\text{ip}}}$, $\mathbf{F}^{(\text{ip})} \in \mathbb{R}^{N_{\text{ip}}\times d_{\text{ip}}}$ and $\mathbf{A}^{(\text{op})} \in \mathbb{R}^{N_{\text{op}}\times N_{\text{op}}}$, $\mathbf{F}^{(\text{op})} \in \mathbb{R}^{N_{\text{op}}\times d_{\text{op}}}$ are the adjacency matrices and feature matrices before and after the pooling operation, respectively. Similarly the superscripts (or subscripts) "ip" and "op" are used to indicate *the input of pooling* and *the output of pooling*, receptively. Note that N_{op} denotes the number of nodes in the coarsened graph and $N_{\text{op}} < N_{\text{ip}}$.

The architecture of a typical graph neural network model consists of graph filtering and/or graph pooling operations. For node-focused tasks, GNNs only utilize graph filtering operations. They are often composed with multiple con-

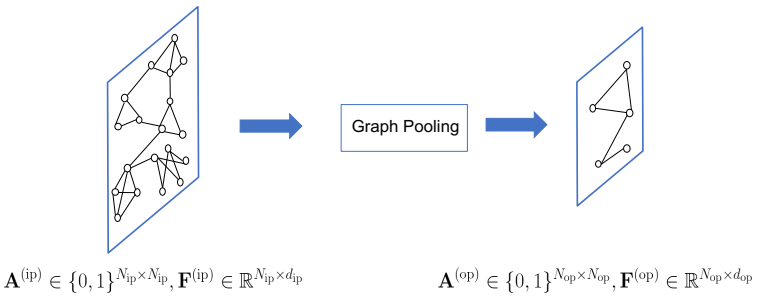

Figure 5.2 Graph pooling operation

secutive graph filtering layers where the output of the previous layer is the input for the following consecutive layer. For graph-focused tasks, GNNs require both the graph filtering and the graph pooling operations. Pooling layers usually separate the graph filtering layers into blocks. In this chapter, we first briefly introduce general architectures for GNNs and then provide the details of representative graph filtering and graph pooling operations.

5.2 The General GNN Frameworks

In this section, we introduce the general frameworks of GNNs for both node-focused and graph-focused tasks. We first introduce some notations that we use through the following sections. We denote a graph as $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$. The adjacency matrix of the graph with N nodes is denoted as $\mathbf{A}$. The associated features are represented as $\mathbf{F} \in \mathbb{R}^{N \times d}$. Each row of $\mathbf{F}$ corresponds to a node, and d is the dimension of the features.

5.2.1 A General Framework for Node-focused Tasks

A general framework for node-focused tasks can be regarded as a composition of graph filtering and non-linear activation layers. A GNN framework with L graph filtering layers and $L-1$ activation layers (see Section 3.2.2 for representative activation functions) is shown in Figure 5.3, where $h_i()$ and $\alpha_i()$ denote the i-th graph filtering layer and activation layer, respectively. We use $\mathbf{F}^{(i)}$ to denote the output of the i-th graph filtering layer. Specifically, $\mathbf{F}^{(0)}$ is initialized to be the associated features $\mathbf{F}$. Furthermore, we use d_i to indicate the dimension of the output of the i-th graph filtering layer. Since the graph structure is unchanged, we have $\mathbf{F}^{(i)} \in \mathbb{R}^{N \times d_i}$. The i-th graph filtering layer can

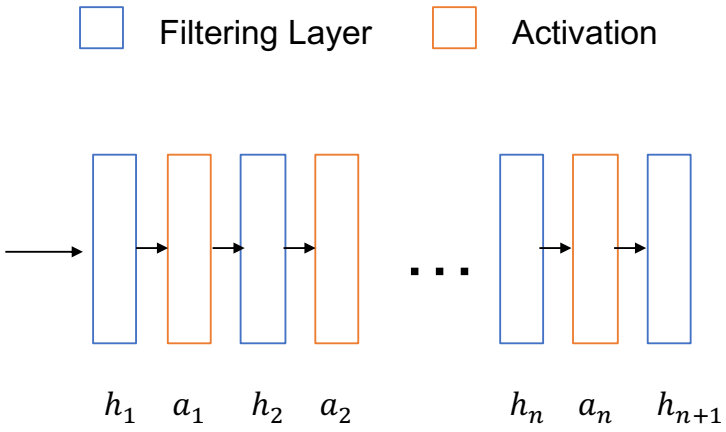

Figure 5.3 A general GNN architecture for node-focused tasks

be described as:

$$\mathbf{F}^{(i)} = h_i\left(\mathbf{A}, \alpha_{i-1}\left(\mathbf{F}^{(i-1)}\right)\right)$$

where $\alpha_{i-1}()$ is the element-wise activation function following the $(i-1)$-th graph filtering layer. Note that we abuse the notation a little bit to use α_0 to denote the identity function as we do not apply the activation on the input features. The final output $\mathbf{F}^{(L)}$ is leveraged as the input to some specific layers according to the downstream node-focused tasks.

5.2.2 A General Framework for Graph-focused Tasks

A general GNN framework for graph-focused tasks consists of three types of layers, i.e., the graph filtering layer, the activation layer, and the graph pooling layer. The graph filtering layer and the activation layer in the framework have similar functionalities as those in the node-focused framework. They are used to generate better node features. The graph pooling layer is utilized to summarize the node features and generate higher-level features that can capture the information of the entire graph. Typically, a graph pooling layer follows a series of graph filtering and activation layers. A coarsened graph with more abstract and higher-level node features is generated after the graph pooling layer. These layers can be organized into a *block* as shown in Figure 5.4 where h_i, α_i and p denote the i-th filtering layer, the i-th activation layer and the pooling layer in this block. The input of the block is the adjacency matrix $\mathbf{A}^{(\text{ib})}$ and the features $\mathbf{F}^{(\text{ib})}$ of a graph $\mathcal{G}_{ib} = \{\mathcal{V}_{ib}, \mathcal{E}_{ib}\}$ and the output is the newly

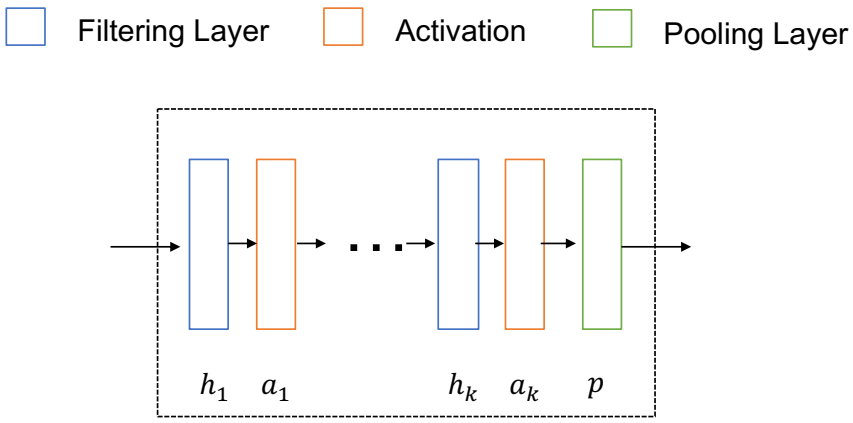

Figure 5.4 A block in GNNs for graph-focused tasks

generated adjacency matrix $\mathbf{A}^{(ob)}$ and the features $\mathbf{F}^{(ob)}$ for the coarsened graph $\mathcal{G}_{ob} = \{\mathcal{V}_{ob}, \mathcal{E}_{ob}\}$. The computation procedure of a block is formally stated as follows:

$$\begin{aligned} \mathbf{F}^{(i)} &= h_i(\mathbf{A}^{(ib)}, \alpha_{i-1}(\mathbf{F}^{(i-1)})) \quad \text{for} \quad i = 1, \ldots k, \\ \mathbf{A}^{(ob)}, \mathbf{F}^{(ob)} &= p(\mathbf{A}^{(ib)}, \mathbf{F}^{(k)}), \end{aligned} \tag{5.3}$$

where α_i is the activation function for $i \neq 0$ where α_0 is the identity function and $\mathbf{F}^{(0)} = \mathbf{F}^{ib}$. We can summarize the above computation process of a block as follows:

$$\mathbf{A}^{(ob)}, \mathbf{F}^{(ob)} = B(\mathbf{A}^{(ib)}, \mathbf{F}^{(ib)}).$$

The entire GNN framework can consist of one or more of these *blocks* as shown in Figure 5.5. The computation process of the GNN framework with L blocks can be formally defined as follows:

$$\mathbf{A}^{(j)}, \mathbf{F}^{(j)} = B^{(j)}(\mathbf{A}^{(j-1)}, \mathbf{F}^{(j-1)}) \quad \text{for} \quad j = 1, \ldots, L. \tag{5.4}$$

where $\mathbf{F}^{(0)} = \mathbf{F}$ and $\mathbf{A}^{(0)} = \mathbf{A}$ are the initial node features and the adjacency matrix of the original graph, respectively. Note that the output of one block is utilized as the input for the consecutively following block as shown in Eq. (5.4). When there is only one block (or $L = 1$), the GNN framework can be regarded as *flat* since it directly generates graph-level features from the original graph. The GNN framework with pooling layers can be viewed as a hierarchical process when $L > 1$, where the node features are gradually summarized to

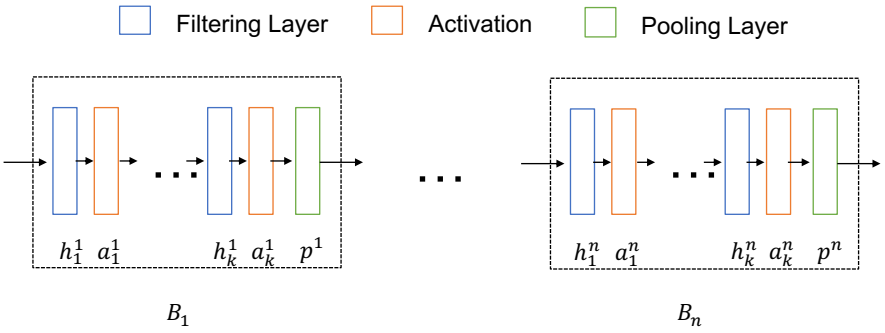

Figure 5.5 Architectures of GNNs for graph-focused tasks

form the graph features by subsequently generating more and more coarsened graphs.

5.3 Graph Filters

There are various perspectives to design graph filters, which can be roughly split into two categories: 1) spatial-based graph filters and 2) spectral-based graph filters. The spatial-based graph filters explicitly leverage the graph structure (i.e., the connections between the nodes) to perform the feature refining process in the graph domain. In contrast, the spectral-based graph filters utilize spectral graph theory to design the filtering operation in the spectral domain. These two categories of graph filters are closely related. Especially, some of the spectral-based graph filters can be regarded as spatial-based filters. In this section, we introduce spectral-based graph filters and explain how some of the spectral-based graph filters can be viewed from a spatial perspective. We then discuss more spatial-based graph filters.

5.3.1 Spectral-based Graph Filters

The spectral-based graph filters are designed in the spectral domain of graph signals. We first introduce the graph spectral filtering and then describe how it can be adopted to design spectral-based graph filters.

Graph Spectral Filtering

As shown in Figure 5.6, the idea of the graph spectral filtering is to modulate the frequencies of a graph signal such that some of its frequency components

are kept/amplified while others are removed/diminished. Hence, given a graph signal $\mathbf{f} \in \mathbb{R}^N$, we need first to apply Graph Fourier Transform (GFT) on it to obtain its graph Fourier coefficients, and then modulate these coefficients before reconstructing the signal in the spatial domain.

As introduced in Chapter 2, for a signal $\mathbf{f} \in \mathbb{R}^N$ defined on a graph $\mathcal{G}$, its Graph Fourier Transform is defined as follows:

$$\hat{\mathbf{f}} = \mathbf{U}^\top \mathbf{f},$$

where $\mathbf{U}$ consists of eigenvectors of the Laplacian matrix of $\mathcal{G}$ and $\hat{\mathbf{f}}$ is the obtained graph Fourier coefficients for the signal $\mathbf{f}$. These graph Fourier coefficients describe how each graph Fourier component contributes to the graph signal $\mathbf{f}$. Specifically, the i-th element of $\hat{\mathbf{f}}$ corresponds to the i-th graph Fourier component $\mathbf{u}_i$ with the frequency λ_i. Note that λ_i is the eigenvalue corresponding to $\mathbf{u}_i$. To modulate the frequencies of the signal $\mathbf{f}$, we filter the graph Fourier coefficients as follows:

$$\hat{\mathbf{f}}'[i] = \hat{\mathbf{f}}[i] \cdot \gamma(\lambda_i), \text{for} \quad i = 1, \ldots, N.$$

where $\gamma(\lambda_i)$ is a function with the frequency λ_i as input which determines how the corresponding frequency component should be modulated. This process can be expressed in a matrix form as follows:

$$\hat{\mathbf{f}}' = \gamma(\mathbf{\Lambda}) \cdot \hat{\mathbf{f}} = \gamma(\mathbf{\Lambda}) \cdot \mathbf{U}^\top \mathbf{f},$$

where $\mathbf{\Lambda}$ is a diagonal matrix consisting of the frequencies (eigenvalues of the Laplacian matrix) and $\gamma(\mathbf{\Lambda})$ is applying the function $\gamma()$ to each element in the diagonal of $\mathbf{\Lambda}$. Formally, $\mathbf{\Lambda}$ and $\gamma(\mathbf{\Lambda})$ can be represented as follows:

$$\mathbf{\Lambda} = \begin{pmatrix} \lambda_1 & & 0 \\ & \ddots & \\ 0 & & \lambda_N \end{pmatrix}; \quad \gamma(\mathbf{\Lambda}) = \begin{pmatrix} \gamma(\lambda_1) & & 0 \\ & \ddots & \\ 0 & & \gamma(\lambda_N) \end{pmatrix}.$$

With the filtered coefficients, we can now reconstruct the signal to the graph domain using the Inverse Graph Fourier Transform (IGFT) as follows:

$$\mathbf{f}' = \mathbf{U}\hat{\mathbf{f}}' = \mathbf{U} \cdot \gamma(\mathbf{\Lambda}) \cdot \mathbf{U}^\top \mathbf{f}, \tag{5.5}$$

where $\mathbf{f}'$ is the obtained filtered graph signal. The filtering process can be regarded as applying the operator $\mathbf{U} \cdot \gamma(\mathbf{\Lambda}) \cdot \mathbf{U}^\top$ to the input graph signal. For convenience, we sometimes refer the function $\gamma(\mathbf{\Lambda})$ as the filter since it controls how each frequency component of the graph signal $\mathbf{f}$ is filtered. For example, in the extreme case, if $\gamma(\lambda_i)$ equals to 0, then, $\hat{\mathbf{f}}'[i] = 0$ and the frequency component $\mathbf{u}_i$ is removed from the graph signal $\mathbf{f}$.

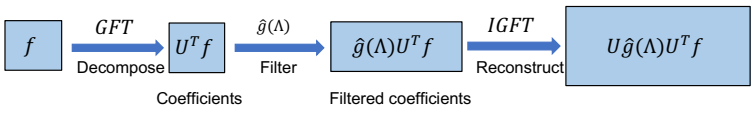

Figure 5.6 The Process of Spectral Filtering

Example 5.1 (Shuman et al., 2013) Suppose that we are given a noisy graph signal $\mathbf{y} = \mathbf{f}_0 + \eta$ defined on a graph $\mathcal{G}$, where η is uncorrelated additive Gaussian noise, we wish to recover the original signal $\mathbf{f}_0$. The original signal $\mathbf{f}_0$ is assumed to be smooth with respect to the underlying graph $\mathcal{G}$. To enforce this prior information of the smoothness of the clean signal $\mathbf{f}_0$, a regularization term of the form $\mathbf{f}^\top \mathbf{L}\mathbf{f}$ is included in the optimization problem as follows:

$$\arg\min_{\mathbf{f}} \ \|\mathbf{f} - \mathbf{y}\|^2 + c\mathbf{f}^\top \mathbf{L}\mathbf{f}, \tag{5.6}$$

where $c > 0$ is a constant to control the smoothness. The objective is convex; hence, the optimal solution $\mathbf{f}'$ can be obtained by setting its derivative to 0 as follows:

$$\begin{aligned} &2(\mathbf{f} - \mathbf{y}) + 2c\mathbf{L}\mathbf{f} = 0 \\ \Rightarrow &(I + c\mathbf{L})\mathbf{f} = \mathbf{y} \\ \Rightarrow &(\mathbf{U}\mathbf{U}^\top + c\mathbf{U}\mathbf{\Lambda}\mathbf{U}^\top)\mathbf{f} = \mathbf{y} \\ \Rightarrow &\mathbf{U}(\mathbf{I} + c\mathbf{\Lambda})\mathbf{U}^\top\mathbf{f} = \mathbf{y} \\ \Rightarrow &\mathbf{f}' = \mathbf{U}(\mathbf{I} + c\mathbf{\Lambda})^{-1}\mathbf{U}^\top\mathbf{y}. \end{aligned} \tag{5.7}$$

Comparing Eq. (5.7) with Eq. (5.5), we can find that the cleaner signal is obtained by filtering the noisy signal $\mathbf{y}$ with the filter $\gamma(\mathbf{\Lambda}) = (\mathbf{I} + c\mathbf{\Lambda})^{-1}$. For a specific frequency λ_l, the filter can be expressed as follows:

$$\gamma(\lambda_l) = \frac{1}{1 + c\lambda_l}, \tag{5.8}$$

which clearly indicates that $\gamma(\lambda_l)$ is a low-pass filter since $\gamma(\lambda_l)$ is large when λ_l is small and it is small when λ_l is large. Hence, solving the optimization problem in Eq.(5.6) is equivalent to applying the low-pass filter in Eq. (5.8) to the noisy signal $\mathbf{y}$.

Spectral-based Graph Filters

We have introduced the graph spectral filtering operator, which can be used to filter certain frequencies in the input signal. For example, if we want to get a smooth signal after filtering, we can design a low-pass filter, where $\gamma(\lambda)$ is large

when λ is small, and it is small when λ is large. In this way, the obtained filtered signal is smooth since it majorly contains the low-frequency part of the input signal. An example of the low-pass filter is shown in Example 5.1. If we know how we want to modulate the frequencies in the input signal, we can design the function $\gamma(\lambda)$ in a corresponding way. However, when utilizing the spectral-based filter as a graph filter in graph neural networks, we often do not know which frequencies are more important. Hence, just like the classical neural networks, the graph filters can be learned in a data-driven way. Specifically, we can model $\gamma(\Lambda)$ with certain functions and then learn the parameters with the supervision from data.

A natural attempt is to give full freedom when designing $\gamma()$ (or a non-parametric model). In detail, the function $\gamma()$ is defined as follows (Bruna et al., 2013):

$$\gamma(\lambda_l) = \theta_l,$$

where θ_l is a parameter to be learned from the data. It can also be represented in a matrix form as follows:

$$\gamma(\Lambda) = \begin{pmatrix} \theta_1 & & 0 \\ & \ddots & \\ 0 & & \theta_N \end{pmatrix}.$$

However, there are some limitations with this kind of filter. First, the number of parameters to be learned is equal to the number of nodes N, which can be extremely large in real-world graphs. Hence, it requires lots of memory to store these parameters and also abundant data to fit them. Second, the filter $\mathbf{U} \cdot \gamma(\Lambda) \cdot \mathbf{U}^\top$ is likely to be a dense matrix. Therefore, the calculation of the i-th element of the output signal $\mathbf{f}'$ could relate to all the nodes in the graph. In other words, the operator is not spatially localized. Furthermore, the computational cost for this operator is quite expensive due to the eigendecomposition of the Laplacian matrix and the matrix multiplication between dense matrices when calculating $\mathbf{U} \cdot \gamma(\Lambda) \cdot \mathbf{U}^\top$.

To address these issues, a polynomial filter operator, which we denoted as Poly-Filter, is proposed in (Defferrard et al., 2016). The function $\gamma()$ can be modeled with a K-order truncated polynomial as follows:

$$\gamma(\lambda_l) = \sum_{k=0}^{K} \theta_k \lambda_l^k. \tag{5.9}$$

In terms of the matrix form, it can be rewritten as below:

$$\gamma(\mathbf{\Lambda}) = \sum_{k=0}^{K} \theta_k \mathbf{\Lambda}^k. \tag{5.10}$$

Clearly, the number of the parameters in Eq. (5.9) and Eq. (5.10) is $K + 1$, which is not dependent on the number of nodes in the graph. Furthermore, we can show that $\mathbf{U} \cdot \gamma(\mathbf{\Lambda}) \cdot \mathbf{U}^\top$ can be simplified to be a polynomial of the Laplacian matrix. It means that: 1) no eigendecomposition is needed; and 2) the polynomial parametrized filtering operator is spatially localized, i.e., the calculation of each element of the output $\mathbf{f}'$ only involves a small number of nodes in the graph. Next, we first show that the Poly-Filter operator can be formulated as a polynomial of the Laplacian matrix and then understand it from a spatial perspective.

By applying this Poly-Filter operator on $\mathbf{f}$, according to Eq. (5.5), we can get the output $\mathbf{f}'$ as follows:

$$\begin{aligned}
\mathbf{f}' &= \mathbf{U} \cdot \gamma(\mathbf{\Lambda}) \cdot \mathbf{U}^\top \mathbf{f} \\
&= \mathbf{U} \cdot \sum_{k=0}^{K-1} \theta_k \mathbf{\Lambda}^k \cdot \mathbf{U}^\top \mathbf{f} \\
&= \sum_{k=0}^{K} \theta_k \mathbf{U} \cdot \mathbf{\Lambda}^k \cdot \mathbf{U}^\top \mathbf{f}.
\end{aligned} \tag{5.11}$$

To further simplify Eq. (5.11), we first show that $\mathbf{U} \cdot \mathbf{\Lambda}^k \cdot \mathbf{U}^\top = \mathbf{L}^k$ as follows:

$$\begin{aligned}
\mathbf{U} \cdot \mathbf{\Lambda}^k \cdot \mathbf{U}^\top &= \mathbf{U} \cdot (\mathbf{\Lambda} \mathbf{U}^\top \mathbf{U})^k \mathbf{U}^\top \\
&= \underbrace{(\mathbf{U} \cdot \mathbf{\Lambda} \cdot \mathbf{U}^\top) \cdots (\mathbf{U} \cdot \mathbf{\Lambda} \cdot \mathbf{U}^\top)}_{k} \\
&= \mathbf{L}^k.
\end{aligned} \tag{5.12}$$

With Eq. (5.12), we can now simplify Eq. (5.11) as follows:

$$\begin{aligned}
\mathbf{f}' &= \sum_{k=0}^{K} \theta_k \mathbf{U} \cdot \mathbf{\Lambda}^k \cdot \mathbf{U}^\top \mathbf{f} \\
&= \sum_{k=0}^{K} \theta_k \mathbf{L}^k \mathbf{f}.
\end{aligned}$$

The polynomials of the Laplacian matrix are all sparse. Meanwhile, the i, j-th ($i \neq j$) element of $\mathbf{L}^k$ is non-zero only when the length of the shortest path between node v_i and node v_j, i.e., $\text{dis}(v_i, v_j)$, is less or equal to k as described in the following lemma.

Lemma 5.2 *Let $\mathcal{G}$ be a graph and $\mathbf{L}$ be its Laplacian matrix. Then, the i, j-th element of the k-th power of the Laplacian matrix $\mathbf{L}^k_{i,j} = 0$ if dis$(v_i, v_j) > k$.*

Proof We prove this Lemma by induction. When $k = 1$, by the definition of the Laplacian matrix $\mathbf{L}$, we naturally have that $\mathbf{L}_{i,j} = 0$ if $d(v_i, v_j) > 1$. Assume for $k = n$, we have that $\mathbf{L}^n_{i,j} = 0$ if dis$(v_i, v_j) > n$. We proceed to prove that for $k = n+1$, we have $\mathbf{L}^{n+1}_{i,j} = 0$ if dis$(v_i, v_j) > n+1$. Specifically, the element $\mathbf{L}^{n+1}_{i,j}$ can be represented using $\mathbf{L}^n$ and $\mathbf{L}$ as:

$$\mathbf{L}^{n+1}_{i,j} = \sum_{h=1}^{N} \mathbf{L}^n_{i,h} \mathbf{L}_{h,j}.$$

We next show that $\mathbf{L}^n_{i,h}\mathbf{L}_{h,j} = 0$ for $h = 1, \ldots, N$, which indicates that $\mathbf{L}^{n+1}_{i,j} = 0$.

If $\mathbf{L}_{h,j} \neq 0$, then dis$(v_h, v_j) \leq 1$, i.e., either $h = j$ or there is an edge between node v_i and node v_j. If we have $d(v_i, v_h) \leq n$, then, with dis$(v_i, v_j) \leq 1$, we have dis$(v_i, v_j) \leq n + 1$, which contradicts the assumption. Hence, dis$(v_i, v_h) > n$ must hold. Thus, we have $\mathbf{L}^n_{i,h} = 0$, which means $\mathbf{L}^n_{i,h}\mathbf{L}_{h,j} = 0$.

If $\mathbf{L}_{h,j} = 0$, then $\mathbf{L}^n_{i,h}\mathbf{L}_{h,j} = 0$ also holds. Therefore, $\mathbf{L}^{n+1}_{i,j} = 0$ if dis$(v_i, v_j) > n + 1$, which completes the proof. □

We now focus on a single element of the output signal $\mathbf{f}'$ to observe how the calculation is related to other nodes in the graph. More specifically, the value of the output signal at the node v_i can be calculated as:

$$\mathbf{f}'[i] = \sum_{v_j \in \mathcal{V}} \left(\sum_{k=0}^{K} \theta_k \mathbf{L}^k_{i,j} \right) \mathbf{f}[j], \tag{5.13}$$

which can be regarded as a linear combination of the original signal on all the nodes according to the weight $\sum_{k=0}^{K} \theta_k \mathbf{L}^k_{i,j}$. According to Lemma 5.2, $\mathbf{L}^k_{i,j} = 0$ when dis$(v_i, v_j) > k$. Hence, not all the nodes are involved in this calculation, but only those nodes that are within K-hop of the node v_i are involved. We can reorganize Eq. (5.13) with only those nodes that are within K-hop neighborhood of node v_i as:

$$\mathbf{f}'[i] = b_{i,i}\mathbf{f}[i] + \sum_{v_j \in \mathcal{N}^K(v_i)} b_{i,j}\mathbf{f}[j], \tag{5.14}$$

where $\mathcal{N}^K(v_i)$ denotes all the nodes that are within K-hop neighborhood of the node v_i and the parameter $b_{i,j}$ is defined as:

$$b_{i,j} = \sum_{k=\text{dis}(v_i,v_j)}^{K} \theta_k \mathbf{L}^k_{i,j},$$

where $\text{dis}(v_i, v_j)$ denotes the length of the shortest path between node v_i and node v_j. We can clearly observe that the Poly-Filter is localized in the spatial domain as it only involves K-hop neighborhoods when calculating the output signal value for a specific node. Furthermore, the Poly-Filter can be also regarded as a spatial-based graph filter as the filtering process can be described based on the spatial graph structure as shown in Eq. (5.14). A similar graph filter operation is proposed in (Atwood and Towsley, 2016). Instead of using the powers of Laplacian matrix, it linearly combines information aggregated from multi-hop neighbors of the center node with powers of the adjacency matrix .

While the Poly-Filter enjoys various advantages, there are still some limitations. One major issue is that the basis of the polynomial (i.e., $1, x, x^2, \dots$) is not orthogonal to each other. Hence, the coefficients are dependent on each other, making them unstable under perturbation during the learning process. In other words, an update in one coefficient may lead to changes in other coefficients. To address this issue, Chebyshev polynomial, which has a set of orthogonal basis, is utilized to model the filter (Defferrard et al., 2016). Next, we briefly discuss the Chebyshev polynomial and then detail the Cheby-Filter based on the Chebyshev polynomial.

Chebyshev Polynomial and Cheby-Filter

The Chebyshev polynomials $T_k(y)$ can be generated by the following recurrence relation:

$$T_k(y) = 2yT_{k-1}(y) - T_{k-2}(y), \tag{5.15}$$

with $T_0(y) = 1$ and $T_1(y) = y$, respectively. For $y \in [-1, 1]$, these Chebyshev polynomials can be represented in the trigonometric expression as:

$$T_k(y) = \cos(k \arccos(y)),$$

which means that each $T_k(y)$ is bounded in $[-1, 1]$. Furthermore, these Chebyshev polynomials satisfy the following relation:

$$\int_{-1}^{1} \frac{T_l(y)T_m(y)}{\sqrt{1-y^2}} dy = \begin{cases} \delta_{l,m}\pi/2 & \text{if } m, l > 0, \\ \pi & \text{if } m = l = 0, \end{cases} \tag{5.16}$$

where $\delta_{l,m} = 1$ only when $l = m$, otherwise $\delta_{l,m} = 0$. Eq. (5.16) indicates that the Chebyshev polynomials are orthogonal to each other. Thus, the Chebyshev polynomials form an orthogonal basis for the Hillbert space of square integrable functions with respect to the measure $dy/\sqrt{1-y^2}$, which is denoted as $L^2([-1, 1], dy/\sqrt{1-y^2})$.

As the domain for the Chebyshev polynomials is $[-1, 1]$, to approximate the

filter with the Chebyshev polynomials, we rescale and shift the eigenvalues of the Laplacian matrix as follows:

$$\tilde{\lambda}_l = \frac{2 \cdot \lambda_l}{\lambda_{max}} - 1,$$

where $\lambda_{max} = \lambda_N$ is the largest eigenvalue of the Laplacian matrix. Clearly, all the eigenvalues are transformed to the range $[-1, 1]$ by this operation. Correspondingly, in the matrix form, the rescaled and shifted diagonal eigenvalue matrix is denoted as:

$$\tilde{\mathbf{\Lambda}} = \frac{2\mathbf{\Lambda}}{\lambda_{max}} - \mathbf{I},$$

where $\mathbf{I}$ is the identity matrix. The Cheby-Filter, which is parametarized with the truncated Chebyshev polynomials can be formulated as follows:

$$\gamma(\mathbf{\Lambda}) = \sum_{k=0}^{K} \theta_k T_k(\tilde{\mathbf{\Lambda}}).$$

The process of applying the Cheby-Filter on a graph signal $\mathbf{f}$ can be defined as:

$$\begin{aligned} \mathbf{f}' &= \mathbf{U} \cdot \sum_{k=0}^{K} \theta_k T_k(\tilde{\mathbf{\Lambda}}) \mathbf{U}^\top \mathbf{f} \\ &= \sum_{k=0}^{K} \theta_k \mathbf{U} T_k(\tilde{\mathbf{\Lambda}}) \mathbf{U}^\top \mathbf{f}. \end{aligned} \tag{5.17}$$

Next, we show that $\mathbf{U}T_k(\tilde{\mathbf{\Lambda}})\mathbf{U}^\top = T_k(\tilde{\mathbf{L}})$ with $\tilde{\mathbf{L}} = \frac{2\mathbf{L}}{\lambda_{max}} - \mathbf{I}$ in the following theorem.

Theorem 5.3 *For a graph $\mathcal{G}$ with Laplacian matrix $\mathbf{L}$, the following equation holds for $k \geq 0$.*

$$\mathbf{U}T_k(\tilde{\mathbf{\Lambda}})\mathbf{U}^\top = T_k(\tilde{\mathbf{L}}),$$

where

$$\tilde{\mathbf{L}} = \frac{2\mathbf{L}}{\mathbf{\Lambda}_{max}} - \mathbf{I}.$$

Proof For $k = 0$, the equation holds as $\mathbf{U}T_0(\tilde{\mathbf{\Lambda}})\mathbf{U}^\top = \mathbf{I} = T_0(\tilde{\mathbf{L}})$.

For $k = 1$,

$$\begin{aligned}\mathbf{U}T_1(\tilde{\mathbf{\Lambda}})\mathbf{U}^\top &= \mathbf{U}\tilde{\mathbf{\Lambda}}\mathbf{U}^\top \\ &= \mathbf{U}(\frac{2\mathbf{\Lambda}}{\mathbf{\Lambda}_{max}} - \mathbf{I})\mathbf{U}^\top \\ &= \frac{2\mathbf{L}}{\lambda_{max}} - \mathbf{I} \\ &= T_1(\tilde{\mathbf{L}}).\end{aligned}$$

Hence, the equation also holds for $k = 1$.

Assume that the equation holds for $k = n - 2$ and $k = n - 1$ with $n \geq 2$, we show that the equation also holds for $k = n$ using the recursive relation in Eq. (5.15) as:

$$\begin{aligned}\mathbf{U}T_n(\tilde{\mathbf{\Lambda}})\mathbf{U}^\top &= \mathbf{U}\left[2\tilde{\mathbf{\Lambda}}T_{n-1}(\tilde{\mathbf{\Lambda}}) - T_{n-2}(\tilde{\mathbf{\Lambda}})\right]\mathbf{U}^\top \\ &= 2\mathbf{U}\tilde{\mathbf{\Lambda}}T_{n-1}(\tilde{\mathbf{\Lambda}})\mathbf{U}^\top - \mathbf{U}T_{n-2}(\tilde{\mathbf{\Lambda}})\mathbf{U}^\top \\ &= 2\mathbf{U}\tilde{\mathbf{\Lambda}}\mathbf{U}\mathbf{U}^\top T_{n-1}(\tilde{\mathbf{\Lambda}})\mathbf{U}^\top - T_{n-2}(\tilde{\mathbf{L}}) \\ &= 2\tilde{\mathbf{L}}T_{n-1}(\tilde{\mathbf{L}}) - T_{n-2}(\tilde{\mathbf{L}}) \\ &= T_n(\tilde{\mathbf{L}}),\end{aligned}$$

which completes the proof. □

With theorem 5.3, we can further simplify Eq. (5.17) as:

$$\begin{aligned}\mathbf{f}' &= \sum_{k=0}^{K} \theta_k \mathbf{U}T_k(\tilde{\mathbf{\Lambda}})\mathbf{U}^\top\mathbf{f} \\ &= \sum_{k=0}^{K} \theta_k T_k(\tilde{\mathbf{L}})\mathbf{f}\end{aligned}$$

Hence, the Cheby-Filter still enjoys the advantages of Poly-Filter while it is more stable under perturbations.

GCN-Filter: Simplified Cheby-Filter Involving 1-hop Neighbors

The Cheby-Filter involves a K-hop neighborhood of a node when calculating the new features for the node. In (Kipf and Welling, 2016a), a simplified Cheby-Filter named GCN-Filter is proposed. It is simplified from the Cheby-Filter by setting the order of Chebyshev polynomials to $K = 1$ and approximating $\lambda_{max} \approx 2$. Under this simplification and approximation, the Cheby-Filter

with $K = 1$ can be simplified as follows:

$$\begin{aligned}\gamma(\mathbf{\Lambda}) &= \theta_0 T_0(\tilde{\mathbf{\Lambda}}) + \theta_1 T_1(\tilde{\mathbf{\Lambda}}) \\ &= \theta_0 \mathbf{I} + \theta_1 \tilde{\mathbf{\Lambda}} \\ &= \theta_0 \mathbf{I} + \theta_1 (\mathbf{\Lambda} - \mathbf{I}).\end{aligned}$$

Correspondingly, applying the GCN-Filter to a graph signal $\mathbf{f}$, we can get the output signal $\mathbf{f}'$ as follows:

$$\begin{aligned}\mathbf{f}' &= \mathbf{U}\gamma(\mathbf{\Lambda})\mathbf{U}^\top \mathbf{f} \\ &= \theta_0 \mathbf{U}\mathbf{I}\mathbf{U}^\top \mathbf{f} + \theta_1 \mathbf{U}(\mathbf{\Lambda} - \mathbf{I})\mathbf{U}^\top \mathbf{f} \\ &= \theta_0 \mathbf{f} - \theta_1 (\mathbf{L} - \mathbf{I})\mathbf{f} \\ &= \theta_0 \mathbf{f} - \theta_1 (\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{\frac{1}{2}})\mathbf{f}. \end{aligned} \tag{5.18}$$

Note that Eq. (5.18) holds as the normalized Laplacian matrix as defined in Definition 2.29 is adopted, i.e. $\mathbf{L} = \mathbf{I} - \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$. A further simplification is applied to Eq. (5.18) by setting $\theta = \theta_0 = -\theta_1$, which leads to

$$\begin{aligned}\mathbf{f}' &= \theta_0 \mathbf{f} - \theta_1 (\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{\frac{1}{2}})\mathbf{f} \\ &= \theta(\mathbf{I} + \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}})\mathbf{f}. \end{aligned} \tag{5.19}$$

Note that the matrix $\mathbf{I} + \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$ has eigenvalues in the range $[0, 2]$, which may lead to numerical instabilities when this operator is repeatedly applied to a specific signal $\mathbf{f}$. Hence, a renormalization trick is proposed to alleviate this problem, which uses $\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}$ to replace the matrix $\mathbf{I} + \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}$ in Eq. (5.19), where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ and $\tilde{\mathbf{D}}_{ii} = \sum_j \tilde{\mathbf{A}}_{i,j}$. The final GCN-Filter after these simplifications is defined as:

$$\mathbf{f}' = \theta\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{f}. \tag{5.20}$$

The i, j-th element of $\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}$ is non-zero only when nodes v_i and v_j are connected. For a single node, this process can be viewed as aggregating information from its 1-hop neighbors where the node itself is also regarded as its 1-hop neighbor. Thus, the GCN-Filter can also be viewed as a spatial-based filter, which only involves directly connected neighbors when updating node features.

Graph Filters for Multi-channel Graph Signals

We have introduced the graph filters for 1-channel graph signals, where each node is associated with a single scalar value. However, in practice, graph signals are typically multi-channel, where each node has a vector of features. A

multi-channel graph signals with d_{in} dimensions can be denoted as $\mathbf{F} \in \mathbb{R}^{N \times d_{in}}$. To extend the graph filters to the multi-channel signals, we utilize the signals from all the input channels to generate the output signal as follows:

$$\mathbf{f}_{out} = \sum_{d=1}^{d_{in}} \mathbf{U} \cdot \gamma_d(\mathbf{\Lambda}) \cdot \mathbf{U}^\top \mathbf{F}_{:,d}$$

where $\mathbf{f}_{out} \in \mathbb{R}^N$ is the 1-channel output signal of the filter and $\mathbf{F}_{:,d} \in \mathbb{R}^N$ denotes the d-th channel of the input signal. Thus, the process can be viewed as applying the graph filter in each input channel and then calculating the summation of their results. Just as the classical convolutional neural networks, in most of the cases, multiple filters are used to filter the input channels and the output is also a multi-channel signal. Suppose that we use d_{out} filters, the process to generate the d_{out}-channel output signal is defined as:

$$\mathbf{F}'_{:,j} = \sum_{d=1}^{d_{in}} \mathbf{U} \cdot \gamma_{j,d}(\mathbf{\Lambda}) \cdot \mathbf{U}^\top \mathbf{F}_{:,d} \quad \text{for } j = 1, \ldots, d_{out}.$$

Specifically, in the case of GCN-Filter in Eq. (5.20), this process for multi-channel input and output can be simply represented as:

$$\mathbf{F}'_{:,j} = \sum_{d=1}^{d_{in}} \theta_{j,d} \tilde{\mathbf{D}}^{-\frac{1}{2}} \tilde{\mathbf{A}} \tilde{\mathbf{D}}^{-\frac{1}{2}} \mathbf{F}_{:,d} \quad \text{for } j = 1, \ldots, d_{out},$$

which can be further rewritten in a matrix form as:

$$\mathbf{F}' = \tilde{\mathbf{D}}^{-\frac{1}{2}} \tilde{\mathbf{A}} \tilde{\mathbf{D}}^{-\frac{1}{2}} \mathbf{F} \mathbf{\Theta} \tag{5.21}$$

where $\mathbf{\Theta} \in \mathbb{R}^{d_{in} \times d_{out}}$ and $\mathbf{\Theta}[d, j] = \theta_{j,d}$ is the parameter corresponding to the j-th output channel and d-th input channel. Specifically, for a single node v_i, the filtering process in Eq. (5.21) can also be formulated in the following form:

$$\mathbf{F}'_i = \sum_{v_j \in \mathcal{N}(v_i) \cup \{v_i\}} \left(\tilde{\mathbf{D}}^{-\frac{1}{2}} \tilde{\mathbf{A}} \tilde{\mathbf{D}}^{-\frac{1}{2}} \right)_{i,j} \mathbf{F}_j \mathbf{\Theta} = \sum_{v_j \in \mathcal{N}(v_i) \cup \{v_i\}} \frac{1}{\sqrt{\tilde{d}_i \tilde{d}_j}} \mathbf{F}_j \mathbf{\Theta}, \tag{5.22}$$

where $\tilde{d}_i = \tilde{\mathbf{D}}_{i,i}$ and we use $\mathbf{F}_i \in \mathbb{R}^{1 \times d_{out}}$ to denote the i-th row of $\mathbf{F}$, i.e., the features for node v_i. The process in Eq. (5.22) can be regarded as aggregating information from 1-hop neighbors of node v_i.

5.3.2 Spatial-based Graph Filters

As shown in Eq. (5.22), for a node v_i, the GCN-Filter performs a spatial information aggregation involving 1-hop neighbors and the matrix $\mathbf{\Theta}$ consisting of parameters for the filters can be regarded as a linear transformation applying to

the input node features. In fact, spatial-based filters in graph neural networks have been proposed even before deep learning became popular (Scarselli et al., 2005). More recently, A variety of spatial-based filters have been designed for graph neural networks. In this section, we review the very first spatial filter (Scarselli et al., 2005, 2008) and then more advanced spatial-based filters.

The filter in the very first graph neural network

The concept of graph neural networks was first proposed in (Scarselli et al., 2008). This GNN model iteratively updates features of one node by utilizing features of its neighbors. Next, we briefly introduce the design of the filter in the very first GNN model. Specifically, the model is proposed to deal with graph data where each node is associated with an input label. For node v_i, its corresponding label can be denoted as l_i. For the filtering process, the input graph feature is denoted as $\mathbf{F}$, where $\mathbf{F}_i$, i.e., the i-th row of $\mathbf{F}$, is the associated features for node v_i. The output features of the filter are represented as $\mathbf{F}'$. The filtering operation for node v_i can be described as:

$$\mathbf{F}'_i = \sum_{v_j \in \mathcal{N}(v_i)} g(l_i, \mathbf{F}_j, l_j),$$

where $g()$ is a parametric function, called *local transition function*, which is spatially localized. The filtering process for node v_i only involves its 1-hop neighbors. Typically $g()$ can be modeled by feedforward neural networks. The function $g()$ is shared by all the nodes in the graph when performing the filtering process. Note that the node label information l_i can be viewed as the initial input information, which is fixed and utilized in the filtering process.

GraphSAGE-Filter

The GraphSAGE model proposed in (Hamilton et al., 2017a) introduced a spatial based filter, which is also based on aggregating information from neighboring nodes. For a single node v_i, the process to generate its new features can be formulated as follows:

$$\mathcal{N}_S(v_i) = \text{SAMPLE}(\mathcal{N}(v_i), S) \tag{5.23}$$

$$\mathbf{f}'_{\mathcal{N}_S(v_i)} = \text{AGGREGATE}(\{\mathbf{F}_j, \forall v_j \in \mathcal{N}_S(v_i)\}) \tag{5.24}$$

$$\mathbf{F}'_i = \sigma(\left[\mathbf{F}_i, \mathbf{f}'_{\mathcal{N}_S(v_i)}\right] \mathbf{\Theta}) \tag{5.25}$$

where SAMPLE() is a function that takes a set as input and randomly samples S elements from the input as output, AGGREGATE() is a function to combine the information from the neighboring nodes where $\mathbf{f}'_{\mathcal{N}_S(v_i)}$ denotes the output of the AGGREGATE() function and $[\cdot, \cdot]$ is the concatenation operation. Hence,

for a single node v_i, the filter in GraphSAGE first samples S nodes from its neighboring nodes $\mathcal{N}(v_i)$ as shown in Eq. (5.23). Then, the AGGREGATE() function aggregates the information from these sampled nodes and generates the feature $\mathbf{f}'_{\mathcal{N}_S(v_i)}$ as shown in Eq. (5.24). Finally, the generated neighborhood information and the old features of node v_i are combined to generate the new features for node v_i as shown in Eq. (5.25). Various AGGREGATE() functions have been introduced in (Hamilton et al., 2017a) as below.

- **Mean aggregator.** The mean aggregator is to simply take element-wise mean of the vectors in $\{\mathbf{F}_j, \forall v_j \in \mathcal{N}_S(v_i)\}$. The mean aggregator here is very similar to the filter in GCN. When dealing with a node v_i, both of them take the (weighted) average of the neighboring nodes as its new representation. The difference is how the input representation $\mathbf{F}_j$ of node v_i gets involved in the calculation. It is clear that in GraphSAGE, $\mathbf{F}_i$ is concatenated to the aggregated neighboring information $\mathbf{f}'_{\mathcal{N}_S(v_i)}$. However, in the GCN-Filter, the node v_i is treated equally as its neighbors and $\mathbf{F}_i$ is a part of the weighted average process.
- **LSTM aggregator.** The LSTM aggregator is to treat the set of the sampled neighboring nodes $\mathcal{N}_S(v_i)$ of node v_i as a sequence and utilize the LSTM architecture to process the sequence. The output of the last unit of the LSTM serves as the result $\mathbf{f}'_{\mathcal{N}_S(v_i)}$. However, there is no natural order among the neighbors; hence, a random ordering is adopted in (Hamilton et al., 2017a).
- **Pooling operator.** The pooling operator adopts the max pooling operation to summarize the information from the neighboring nodes. Before summarizing the results, input features at each node are first transformed with a layer of a neural network. The process can be described as follows

$$\mathbf{f}'_{\mathcal{N}_S(v_i)} = \max(\{\alpha(\mathbf{F}_j\mathbf{\Theta}_{\text{pool}}), \forall v_j \in \mathcal{N}_S(v_i)\}),$$

  where max() denotes the element-wise max operator, $\mathbf{\Theta}_{\text{pool}}$ denotes a transformation matrix and $\alpha()$ is a non-linear activation function.

The GraphSAGE-Filter is spatially localized as it only involves 1-hop neighbors no matter which aggregator is used. The aggregator is also shared among all the nodes.

GAT-Filter

Self-attention mechanism (Vaswani et al., 2017) is introduced to build spatial graph filters in graph attention networks (GAT) (Veličković et al., 2017). For convenience, we call the graph filter in GAT as GAT-Filter. The GAT-Filter is similar to the GCN-Filter since it also performs an information aggregation

from neighboring nodes when generating new features for each node. The aggregation in GCN-Filter is solely based on the graph structure, while GAT-filter tries to differentiate the importance of the neighbors when performing the aggregation. More specifically, when generating the new features for a node v_i, it attends to all its neighbors to generate an importance score for each neighbor. These importance scores are then adopted as linear coefficients during the aggregation process. Next, we detail the GAT-Filter.

The importance score of node $v_j \in \mathcal{N}(v_i) \cup \{v_i\}$ to the node v_i can be calculated as follows:

$$e_{ij} = a(\mathbf{F}_i\mathbf{\Theta}, \mathbf{F}_j\mathbf{\Theta}), \tag{5.26}$$

where $\mathbf{\Theta}$ is a shared parameter matrix. $a()$ is a shared attention function which is a single-layer feedforward network in (Veličković et al., 2017) as:

$$a(\mathbf{F}_i\mathbf{\Theta}, \mathbf{F}_j\mathbf{\Theta}) = \text{LeakyReLU}(\mathbf{a}^\top[\mathbf{F}_i\mathbf{\Theta}, \mathbf{F}_j\mathbf{\Theta}]),$$

where $[\cdot, \cdot]$ denotes the concatenation operation, $\mathbf{a}$ is a parametrized vector and LeakyReLU is the nonlinear activation function we have introduced in Section 3.2.2. The scores calculated by Eq. (5.26) are then normalized before being utilized as the weights in the aggregation process to keep the output representation in a reasonable scale. The normalization over all neighbors of v_i is performed through a softmax layer as:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{v_k \in \mathcal{N}(v_i) \cup \{v_i\}} \exp(e_{ik})},$$

where α_{ij} is the normalized importance score indicating the importance of node v_j to node v_i. With the normalized importance scores, the new representation $\mathbf{F}'_i$ of node v_i can be computed as:

$$\mathbf{F}'_i = \sum_{v_j \in \mathcal{N}(v_i) \cup \{v_i\}} \alpha_{ij}\mathbf{F}_j\mathbf{\Theta}, \tag{5.27}$$

where $\mathbf{\Theta}$ is the same transforming matrix as in Eq. (5.26). To stabilize the learning process of self-attention, the multi-head attention (Vaswani et al., 2017) is adopted. Specifically, M independent attention mechanisms in the form of Eq. (5.27) with different $\mathbf{\Theta}^m$ and α^m_{ij} are performed in parallel. Their outputs are then concatenated to generate the final representation of node v_i as:

$$\mathbf{F}'_i = \|_{m=1}^{M} \sum_{v_j \in \mathcal{N}(v_i) \cup \{v_i\}} \alpha^m_{ij}\mathbf{F}_j\mathbf{\Theta}^m, \tag{5.28}$$

where we use $\|$ to denote the concatenation operator. Note that the GAT-Filter is spatially localized, as for each node, only its 1-hop neighbors are

utilized in the filtering process to generate the new features. In the original model (Veličković et al., 2017), activation functions are applied to the output of each attention head before the concatenation. The formulation in Eq. (5.28) did not include activation functions for convenience.

ECC-Filter

When there is edge information available in the graph, it can be utilized for designing the graph filters. Specifically, in (Simonovsky and Komodakis, 2017), an edge-conditioned graph filter (ECC-Filter) is designed when edges have various types (the number of types is finite). For a given edge (v_i, v_j), we use $tp(v_i, v_j)$ to denote its type. Then the ECC-Filter is defined as:

$$\mathbf{F}'_i = \frac{1}{|\mathcal{N}(v_i)|} \sum_{v_j \in \mathcal{N}(v_i)} \mathbf{F}_j \mathbf{\Theta}_{tp(v_i,v_j)},$$

where $\mathbf{\Theta}_{tp(v_i,v_j)}$ is the parameter matrix shared by the edges with the type of $tp(v_i, v_j)$.

GGNN-Filter

The GGNN-Filter (Li et al., 2015) adapts the original GNN filter in (Scarselli et al., 2008) with Gated Recurrent Unit (GRU) (see Section 3.4 for details of GRU). The GGNN-Filter is designed for graphs where the edges are directed and also have different types. Specifically, for an edge $(v_i, v_j) \in \mathcal{E}$, we use $tp(v_i, v_j)$ to denote its types. Note that, as the edges are directed, the types of edges (v_i, v_j) and (v_j, v_i) can be different, i.e, $tp(v_i, v_j) \neq tp(v_j, v_i)$. The filtering process of the GGNN-Filter for a specific node v_i can be formulated as follows:

$$\mathbf{m}_i = \sum_{(v_j,v_i)\in\mathcal{E}} \mathbf{F}_j \mathbf{\Theta}^e_{tp(v_j,v_i)} \tag{5.29}$$

$$\mathbf{z}_i = \sigma\left(\mathbf{m}_i \mathbf{\Theta}^z + \mathbf{F}_i \mathbf{U}^z\right) \tag{5.30}$$

$$\mathbf{r}_i = \sigma\left(\mathbf{m}_i \mathbf{\Theta}^r + \mathbf{F}_i \mathbf{U}^r\right) \tag{5.31}$$

$$\widetilde{\mathbf{F}}_i = \tanh\left(\mathbf{m}_i \mathbf{\Theta} + (\mathbf{r}_i \odot \mathbf{F}_i)\, \mathbf{U}\right) \tag{5.32}$$

$$\mathbf{F}'_i = (1 - \mathbf{z}_i) \odot \mathbf{F}_i + \mathbf{z}_i \odot \widetilde{\mathbf{F}}_i \tag{5.33}$$

where $\mathbf{\Theta}^e_{tp(v_j,v_i)}, \mathbf{\Theta}^z, \mathbf{\Theta}^r$ and $\mathbf{\Theta}$ are parameters to be learned. The first step as in Eq. (5.29) is to aggregate information from both the in-neighbors and out-neighbors of node v_i. During this aggregation, the transform matrix $\mathbf{\Theta}^e_{tp(v_j,v_i)}$ is shared by all the nodes connected to v_i via the edge type $tp(v_j, v_i)$. The remaining equations (or Eqs. (5.30)-(5.33)) are GRU steps to update the hidden representations with the aggregated information $\mathbf{m}_i$. $\mathbf{z}_i$ and $\mathbf{r}_i$ are the update and

reset gates, $\sigma(\cdot)$ is the sigmoid function and $\odot$ denotes the Hardmand operation. Hence, the GGNN-Filter can also be written as:

$$\mathbf{m}_i = \sum_{(v_j,v_i)\in\mathcal{E}} \mathbf{F}_j \mathbf{\Theta}^e_{tp(v_j,v_i)} \tag{5.34}$$

$$\mathbf{F}'_i = \text{GRU}(\mathbf{m}_i, \mathbf{F}_i) \tag{5.35}$$

where Eq. (5.35) summarizes Eqs. (5.30) to (5.33).

Mo-Filter

In (Monti et al., 2017), a general framework, i.e., mixture model networks (MoNet), is introduced to perform convolution operations on non-Euclidean data such as graphs and manifolds. Next, we introduce the graph filtering operation in (Monti et al., 2017), which we name as the Mo-Filter. We take node v_i as an example to illustrate its process. For each neighbor $v_j \in \mathcal{N}(v_i)$, a pseudo-coordinate is introduced to denote the relevant relation between nodes v_j and v_i. Specifically, for the center node v_i and its neighbor v_j, the pseudo-coordinate is defined with their degrees as:

$$c(v_i, v_j) = \left(\frac{1}{\sqrt{d_i}}, \frac{1}{\sqrt{d_j}}\right)^\top, \tag{5.36}$$

where d_i and d_j denote the degree of nodes v_i and v_j, respectively. Then a Gaussian kernel is applied on the pseudo-coordinate to measure the relation between the two nodes as:

$$\alpha_{i,j} = \exp\left(-\frac{1}{2}(c(v_i, v_j) - \mu)^\top \mathbf{\Sigma}^{-1}(c(v_i, v_j) - \mu)\right), \tag{5.37}$$

where μ and $\mathbf{\Sigma}$ are the mean vector and the covariance matrix of the Gaussian kernel to be learned. Note that instead of using the original pseudo-coordinate, we can also utilize a feedforward network to first transform $c(v_i, v_j)$. The aggregation process is as:

$$\mathbf{F}'_i = \sum_{v_j\in\mathcal{N}(v_i)} \alpha_{i,j}\mathbf{F}_j. \tag{5.38}$$

In Eq. (5.38), a single Gaussian kernel is used. However, typically, a set of K kernels with different means and covariances are adopted, which results in the following process:

$$\mathbf{F}'_i = \sum_{k=1}^{K} \sum_{v_j\in\mathcal{N}(v_i)} \alpha^{(k)}_{i,j}\mathbf{F}_j,$$

where $\alpha^{(k)}_{i,j}$ is from the k-th Gaussian kernel.

MPNN: A General Framework for Spatial-based Graph Filters

Message Passing Neural Networks (MPNN) is a general GNN framework. Many spatial-based graph filters including GCN-Filter, GraphSAGE-Filter and GAT-Filter, are its special cases (Gilmer et al., 2017). For a node v_i, the MPNN-Filter updates its features as follows:

$$\mathbf{m}_i = \sum_{v_j \in \mathcal{N}(v_i)} M(\mathbf{F}_i, \mathbf{F}_j, \mathbf{e}_{(v_i,v_j)}), \tag{5.39}$$

$$\mathbf{F}'_i = U(\mathbf{F}_i, \mathbf{m}_i), \tag{5.40}$$

where $M()$ is the message function, $U()$ is the update function and $\mathbf{e}_{(v_i,v_j)}$ is edge features if available. The message function $M()$ generates the messages to pass to node v_i from its neighbors. The update function $U()$ then updates the features of node v_i by combining the original features and the aggregated message from its neighbors. The framework can be even more general if we replace the summation operation in Eq. (5.39) with other aggregation operations.

5.4 Graph Pooling

The graph filters refine the node features without changing the graph structure. After the graph filter operation, each node in the graph has a new feature representation. Typically, the graph filter operations are sufficient for node-focused tasks that take advantage of the node representations. However, for graph-focused tasks, a representation of the entire graph is desired. To obtain such representation, we need to summarize the information from the nodes. There are two main kinds of information that are important for generating the graph representation – one is the node features, and the other is the graph structure. The graph representation is expected to preserve both the node feature information and the graph structure information. Similar to the classical convolutional neural networks, graph pooling layers are proposed to generate graph level representations. The early designs of graph pooling layers are usually flat. In other words, they generate the graph-level representation directly from the node representations in a single step. For example, the average pooling layers and max-pooling layers can be adapted to graph neural networks by applying them to each feature channel. Later on, hierarchical graph pooling designs have been developed to summarize the graph information by coarsening the original graph step by step. In the hierarchical graph pooling design, there are often several graph pooling layers, each of which follows a stack of several filters, as shown in Figure 5.5. Typically, a single graph pooling layer

(both in the flat and the hierarchical cases) takes a graph as input and output a coarsened graph. Recall that the process has been summarized by Eq.(5.2) as:

$$\mathbf{A}^{(\text{op})}, \mathbf{F}^{(\text{op})} = \text{pool}(\mathbf{A}^{(\text{ip})}, \mathbf{F}^{(\text{ip})}), \tag{5.41}$$

Next, we first describe representative flat pooling layers and then introduce hierarchical pooling layers.

5.4.1 Flat Graph Pooling

A flat pooling layer directly generates a graph-level representation from the node representations. In flat pooling layers, there is no new graph but a single node being generated. Thus, instead of Eq.(5.41), the pooling process in flat pooling layers can be summarized as:

$$\mathbf{f}_{\mathcal{G}} = \text{pool}(\mathbf{A}^{(\text{ip})}, \mathbf{F}^{(\text{ip})}),$$

where $\mathbf{f}_{\mathcal{G}} \in \mathbb{R}^{1 \times d_{\text{op}}}$ is the graph representation. Next, we introduce some representative flat pooling layers. The max-pooling and average pooling operations in classical CNNs can be adapted to GNNs. Specifically, the operation of graph max-pooling layer can be expressed as:

$$\mathbf{f}_{\mathcal{G}} = \max(\mathbf{F}^{(\text{ip})}),$$

where the max operation is applied to each channel as follows:

$$\mathbf{f}_{\mathcal{G}}[i] = \max(\mathbf{F}^{(\text{ip})}_{:,i}),$$

where $\mathbf{F}^{(\text{ip})}_{:,i}$ denotes the i-th channel of $\mathbf{F}^{(\text{ip})}$. Similarly, graph average pooling operation applies the average pooling operation channel-wisely as:

$$\mathbf{f}_{\mathcal{G}} = ave(\mathbf{F}^{(\text{ip})}).$$

In (Li et al., 2015), an attention-based flat pooling operation, which is named as gated global pooling, is proposed. An attention score measuring the importance of each node is utilized to summarize the node representations for generating the graph representation. Specifically, the attention score for node v_i is computed as:

$$s_i = \frac{\exp\left(h\left(\mathbf{F}^{(\text{ip})}_i\right)\right)}{\sum_{v_j \in \mathcal{V}} \exp\left(h\left(\mathbf{F}^{(\text{ip})}_j\right)\right)},$$

where h is a feedforward network to map $\mathbf{F}_i^{(\text{ip})}$ to a scalar, which is then normalized through softmax. With the learned attention scores, the graph representation can be summarized from the node representations as:

$$\mathbf{f}_{\mathcal{G}} = \sum_{v_i \in \mathcal{V}} s_i \cdot \tanh\left(\mathbf{F}_i^{(\text{ip})} \mathbf{\Theta}_{ip}\right),$$

where $\mathbf{\Theta}_{ip}$ are parameters to be learned and the activation function tanh() can be also replaced with the identity function.

Some flat graph pooling operations are embedded in the design of the filtering layer. A "fake" node is added to the graph that is connected to all the nodes (Li et al., 2015). The representation of this "fake" node can be learned during the filtering process. Its representation captures the information of the entire graph as it is connected to all nodes in the graph. Hence, the representation of the "fake" node can be leveraged as the graph representation for downstream tasks.

5.4.2 Hierarchical Graph Pooling

Flat pooling layers usually ignore the hierarchical graph structure information when summarizing the node representations for the graph representation. Hierarchical graph pooling layers aim to preserve the hierarchical graph structural information by coarsening the graph step by step until the graph representation is achieved. Hierarchical pooling layers can be roughly grouped according to the ways they coarsen the graph. One type of hierarchical pooling layers coarsens the graph by sub-sampling, i.e., selecting the most important nodes as the nodes for the coarsened graph. A different kind of hierarchical pooling layer combines nodes in the input graph to form supernodes that serve as the nodes for the coarsened graph. The main difference between these two types of coarsening methods is that the sub-sampling based methods keep nodes from the original graph while the supernode-based methods generate new nodes for the coarsened graph. Next, we describe some representative techniques in these two categories. Specifically, we elaborate on the process of hierarchical pooling layers in Eq. (5.41) by explaining how the coarsened graph $\mathbf{A}^{(\text{op})}$ and node features $\mathbf{F}^{(\text{op})}$ are generated.

Downsampling-based Pooling

To coarsen the input graph, a set of N_{op} nodes are selected according to some importance measures, and then graph structure and node features for the coarsened graph are formed upon these nodes. There are three key components in a

downsampling based graph pooling layer: 1) developing the measure for downsampling; 2) generating graph structure for the coarsened graph and 3) generating node features for the coarsened graph. Different downsampling based pooling layers usually have distinct designs in these components. Next, we introduce representative downsampling based graph pooling layers.

The gPool layer (Gao and Ji, 2019) is the first to adopt the downsampling strategy to perform graph coarsening for graph pooling. In gPool, the importance measure for nodes is learned from the input node features $\mathbf{F}^{(\text{ip})}$ as:

$$\mathbf{y} = \frac{\mathbf{F}^{(\text{ip})}\mathbf{p}}{\|\mathbf{p}\|}, \tag{5.42}$$

where $\mathbf{F}^{(\text{ip})} \in \mathbb{R}^{N_{\text{ip}} \times d_{\text{ip}}}$ is the matrix denoting the input node features and $\mathbf{p} \in \mathbb{R}^{d_{\text{ip}}}$ is a vector to be learned to project the input features into importance scores. After obtaining the importance scores $\mathbf{y}$, we can rank all the nodes and select the k most important ones as:

$$\text{idx} = \text{rank}(\mathbf{y}, N_{\text{op}}),$$

where N_{op} is the number of nodes in the coarsened graph and idx denotes the indices of the selected top N_{op} nodes. With the selected nodes represented with their indices idx, we proceed to generate the graph structure and node features for the coarsened graph. Specifically, the graph structure for the coarsened graph can be induced from the graph structure of the input graph as:

$$\mathbf{A}^{(\text{op})} = \mathbf{A}^{(\text{ip})}(\text{idx}, \text{idx}),$$

where $\mathbf{A}^{(\text{ip})}(\text{idx}, \text{idx})$ performs row and column extraction from $\mathbf{A}^{(\text{ip})}$ with the selected indices idx. Similarly, the node features can also be extracted from the input node features. In (Gao and Ji, 2019), gating system is adopted to control the information flow from the input features to the new features. Specifically, the selected nodes with a higher importance score can have more information flow to the coarsened graph, which can be modeled as:

$$\begin{aligned}
\tilde{\mathbf{y}} &= \sigma(\mathbf{y}(\text{idx})) \\
\tilde{\mathbf{F}} &= \mathbf{F}^{(\text{ip})}(\text{idx}, :) \\
\mathbf{F}_p &= \tilde{\mathbf{F}} \odot (\tilde{\mathbf{y}}\mathbf{1}_{d_{\text{ip}}}^\top),
\end{aligned}$$

where $\sigma()$ is the sigmoid function mapping the importance score to $(0, 1)$ and $\mathbf{1}_{d_{\text{ip}}} \in \mathbb{R}^{d_{\text{ip}}}$ is a all-ones vector. Note that $\mathbf{y}(\text{idx})$ extracts the corresponding elements from $\mathbf{y}$ according to the indices in idx and $\mathbf{F}^{(\text{ip})}(\text{idx})$ retrieves the the corresponding rows according to idx.

In gPool, the importance score is learned solely based on the input features,

as shown in Eq. (5.42). It ignores the graph structure information. To incorporate the graph structure information when learning the importance score, the GCN-Filter is utilized to learn the importance score in (Lee et al., 2019). Specifically, the importance score can be obtained as follows:

$$\mathbf{y} = \alpha\left(\text{GCN-Filter}(\mathbf{A}^{(\text{ip})}, \mathbf{F}^{(\text{ip})})\right). \tag{5.43}$$

where α is an activation function such as tanh. Note that $\mathbf{y}$ is a vector instead of a matrix. In other words, the number of the output channel of the GCN-Filter is set to 1. This graph pooling operation is named as the SAGPool.

Supernode-based Hierarchical Graph Pooling

The downsampling based hierarchical graph pooling layers try to coarsen the input graph by selecting a subset of nodes according to some importance measures. During the process, the information about the unselected nodes is lost as these nodes are discarded. Supernode-based pooling methods aim to coarsen the input graph by generating supernodes. Specifically, they try to learn to assign the nodes in the input graph into different clusters, where these clusters are treated as supernodes. These supernodes are regarded as the nodes in the coarsened graph. The edges between the supernodes and the features of these supernodes are then generated to form the coarsened graph. There are three key components in a supernode-based graph pooling layer: 1) generating supernodes as the nodes for the coarsened graph; 2) generating graph structure for the coarsened graph; and 3) generating node features for the coarsened graph. Next, we describe some representative supernode based graph pooling layers.

diffpool

The diffpool algorithm generates the supernodes in a differentiable way. In detail, a soft assignment matrix from the nodes in the input graph to the supernodes is learned using GCN-Filter as:

$$\mathbf{S} = \text{softmax}\left(\text{GCN-Filter}(\mathbf{A}^{(\text{ip})}, \mathbf{F}^{(\text{ip})})\right), \tag{5.44}$$

where $\mathbf{S} \in \mathbb{R}^{N_{\text{ip}} \times N_{\text{op}}}$ is the assignment matrix to be learned. Note that as shown in Eq. (5.3), $\mathbf{F}^{(\text{ip})}$ is usually the output of the latest graph filtering layer. However, in (Ying et al., 2018c), the input of the pooling layer is the output of the previous pooling layer, i.e., the input of a learning block $\mathbf{F}^{(\text{ib})}$ (see details on *block* in Section 5.2.2). Furthermore, several GCN-Filters can be stacked to learn the assignment matrix, though only a single filter is utilized in Eq. (5.44). Each column of the assignment matrix can be regarded as a supernode. The softmax function is applied row-wisely; hence, each row is normalized to have

a summation of 1. The j-th element in the i-th row indicates the probability of assigning the i-th node to the j-th supernode. With the assignment matrix $\mathbf{S}$, we can proceed to generate the graph structure and node features for the coarsened graph. Specifically, the graph structure for the coarsened graph can be generated from the input graph by leveraging the soft assignment matrix $\mathbf{S}$ as:

$$\mathbf{A}^{(\text{op})} = \mathbf{S}^\top \mathbf{A}^{(\text{ip})} \mathbf{S} \in \mathbb{R}^{N_{\text{op}} \times N_{\text{op}}}.$$

Similarly, the node features for the supernodes can be obtained by linearly combining the node features of the input graph according to the assignment matrix $\mathbf{S}$ as:

$$\mathbf{F}^{(\text{op})} = \mathbf{S}^\top \mathbf{F}^{(\text{inter})} \in \mathbb{R}^{N_{\text{op}} \times d_{\text{op}}},$$

where $\mathbf{F}^{(\text{inter})} \in \mathbb{R}^{N_{\text{ip}} \times d_{\text{op}}}$ is the intermediate features learned through GCN-Filters as follows:

$$\mathbf{F}^{(\text{inter})} = \text{GCN-Filter}(\mathbf{A}^{(\text{ip})}, \mathbf{F}^{(\text{ip})}). \tag{5.45}$$

Multiple GCN-Filters can be stacked though only one is shown in Eq. (5.45). The process of diffpool can be summarized as:

$$\mathbf{A}^{(\text{op})}, \mathbf{F}^{(\text{op})} = \text{diffpool}(\mathbf{A}^{(\text{ip})}, \mathbf{F}^{(\text{ip})}).$$

EigenPooling

EigenPooling (Ma et al., 2019b) generates the supernodes using spectral clustering methods and focuses on forming graph structure and node features for the coarsened graph. After applying the spectral clustering algorithm, a set of non-overlapping clusters are obtained, which are also regarded as the supernodes for the coarsened graph. The assignment matrix between the nodes of the input graph and the supernodes can be denoted as $\mathbf{S} \in \{0, 1\}^{N_{\text{ip}} \times N_{\text{op}}}$, where only a single element in each row is 1 and all others are 0. More specifically, $\mathbf{S}[i, j] = 1$ only when the i-th node is assigned to the j-th supernode. For the k-th supernode, we use $\mathbf{A}^{(k)} \in \mathbb{R}^{N^{(k)} \times N^{(k)}}$ to describe the graph structure in its corresponding cluster, where $N^{(k)}$ is the number of nodes in this cluster. We define a sampling operator $\mathbf{C}^{(k)} \in \{0, 1\}^{N_{\text{ip}} \times N^{(k)}}$ as:

$$\mathbf{C}^{(k)}[i, j] = 1 \quad \text{if and only if} \quad \Gamma^{(k)}(j) = v_i,$$

where $\Gamma^{(k)}$ denotes the list of nodes in the k-th cluster and $\Gamma^{(k)}(j) = v_i$ means that node v_i corresponds to the j-th node in this cluster. With this sampling operator, the adjacency matrix for the k-th cluster can be formally defined as:

$$\mathbf{A}^{(k)} = (\mathbf{C}^{(k)})^\top \mathbf{A}^{(\text{ip})} \mathbf{C}^{(k)}.$$

Next, we discuss the process of generating graph structure and node features for the coarsened graph. To form the graph structure between the supernodes, only the connections across the clusters in the original graph are considered. To achieve the goal, we first generate the intra-cluster adjacency matrix for the input graph, which only consists of the edges within each cluster as:

$$\mathbf{A}_{int} = \sum_{k=1}^{N_{\text{op}}} \mathbf{C}^{(k)} \mathbf{A}^{(k)} (\mathbf{C}^{(k)})^{\top}.$$

Then, the inter-cluster adjacency matrix, which only consists of the edges across the clusters, can be represented as $\mathbf{A}_{ext} = \mathbf{A} - \mathbf{A}_{int}$. The adjacency matrix for the coarsened graph can be obtained as:

$$\mathbf{A}^{\text{op}} = \mathbf{S}^{\top} \mathbf{A}_{ext} \mathbf{S}.$$

Graph Fourier Transform is adopted to generate node features. Specifically, graph structure and node features of each subgraph (or cluster) are utilized to generate the node features for the corresponding supernode. Next, we take the k-th cluster as an illustrative example to demonstrate the process. Let $\mathbf{L}^{(k)}$ denote the Laplacian matrix for this subgraph and $\mathbf{u}_1^{(k)}, \ldots, \mathbf{u}_{n^{(k)}}^{(k)}$ are its corresponding eigenvectors. The features of the nodes in this subgraph can be extracted from $\mathbf{F}^{(\text{ip})}$ by using the sampling operator $\mathbf{C}^{(k)}$ as follows:

$$\mathbf{F}_{\text{ip}}^{(k)} = (\mathbf{C}^{(k)})^{\top} \mathbf{F}^{(\text{ip})},$$

where $\mathbf{F}_{\text{ip}}^{(k)} \in \mathbb{R}^{N^{(k)} \times d_{\text{ip}}}$ is the input features for nodes in the k-th cluster.

Then, we apply Graph Fourier Transform to generate the graph Fourier coefficients for all channels of $\mathbf{F}_{\text{ip}}^{(k)}$ as:

$$\mathbf{f}_i^{(k)} = (\mathbf{u}_i^{(k)})^{\top} \mathbf{F}_{\text{ip}}^{(k)} \quad \text{for} \quad i = 1, \ldots, N^{(k)},$$

where $\mathbf{f}_i^{(k)} \in \mathbb{R}^{1 \times d_{\text{ip}}}$ consists of the i-th graph Fourier coefficients for all feature channels. The node features for the k-th supernode can be formed by concatenating these coefficients as:

$$\mathbf{f}^{(k)} = [\mathbf{f}_1^{(k)}, \ldots, \mathbf{f}_{N^{(k)}}^{(k)}].$$

We usually only utilize the first few coefficients to generate features of supernodes for two reasons. First, different subgraphs may have varied numbers of nodes; hence, to ensure the same dimension of features, some of the coefficients need to be discarded. Second, the first few coefficients typically capture most of the important information as in reality, the majority of the graph signals are smooth.

5.5 Parameter Learning for Graph Neural Networks

In this section, we use node classification and graph classification as examples of downstream tasks to illustrate how to learn parameters of Graph Neural Networks. Note that we have formally defined the tasks of node classification and graph classification in Definition 2.42 and Definition 2.46, respectively.

5.5.1 Parameter Learning for Node Classification

As introduced in Definition 2.42, the node set of a graph $\mathcal{V}$ can be divided to two disjoint sets, $\mathcal{V}_l$ with labels and $\mathcal{V}_u$ without labels. The goal of node classification is to learn a model based on the labeled nodes $\mathcal{V}_l$ to predict the labels of the unlabeled nodes in $\mathcal{V}_u$. The GNN model usually takes the entire graph as input to generate node representations, which are then utilized to train a node classifier. Specifically, let $GNN_{\text{node}}(,)$ denote a GNN model with several graph filtering layers stacked as introduced in Section 5.2.1. The $GNN_{\text{node}}(,)$ function takes the graph structure and the node features as input and outputs the refined node features as follows:

$$\mathbf{F}^{(\text{out})} = GNN_{\text{node}}(\mathbf{A}, \mathbf{F}; \mathbf{\Theta}_1), \tag{5.46}$$

where $\mathbf{\Theta}_1$ denotes the model parameters, $\mathbf{A} \in \mathbb{R}^{N\times N}$ is the adjacency matrix, $\mathbf{F} \in \mathbb{R}^{N\times d_{\text{in}}}$ is the input features of the original graph and $\mathbf{F}^{(\text{out})} \in \mathbb{R}^{N\times d_{\text{out}}}$ is the produced output features. Then, the output node features are utilized to perform node classification as:

$$\mathbf{Z} = \text{softmax}(\mathbf{F}^{(\text{out})}\mathbf{\Theta}_2), \tag{5.47}$$

where $\mathbf{Z} \in \mathbb{R}^{N\times C}$ is the output logits for all nodes, $\mathbf{\Theta}_2 \in \mathbb{R}^{d_{\text{out}}\times C}$ is the parameter matrix to transform the features $\mathbf{F}_{out}$ into the dimension of the number of classes C. The i-th row of $\mathbf{Z}$ indicates the predicted class distribution of node v_i and the predicted label is usually the one with the largest probability. The entire process can be summarized as:

$$\mathbf{Z} = f_{GNN}(\mathbf{A}, \mathbf{F}; \mathbf{\Theta}), \tag{5.48}$$

where the function f_{GNN} consists of the processes in Eq. (5.46) and Eq. (5.47) and $\mathbf{\Theta}$ includes the parameters $\mathbf{\Theta}_1$ and $\mathbf{\Theta}_2$. The parameters $\mathbf{\Theta}$ in Eq. (5.48) can be learned by minimizing the following objective:

$$\mathcal{L}_{train} = \sum_{v_i\in\mathcal{V}_l} \ell(f_{GNN}(\mathbf{A}, \mathbf{F}; \mathbf{\Theta})_i, y_i), \tag{5.49}$$

where $f_{GNN}(\mathbf{A}, \mathbf{F}; \mathbf{\Theta})_i$ denotes the i-th row of the output, i.e., the logits for node v_i, y_i is the associated label and $\ell(\cdot, \cdot)$ is a loss function such as cross-entropy loss.

5.5.2 Parameter Learning for Graph Classification

As introduced in Definition 2.46, in the task of graph classification, each graph is treated as a sample with an associated label. The training set can be denoted as $\mathcal{D} = \{\mathcal{G}_i, y_i\}$, where y_i is the corresponding label for graph $\mathcal{G}_i$. The task of graph classification is to train a model on the training set $\mathcal{D}$ such that it can perform good predictions on unlabeled graphs. The graph neural network model is usually utilized as a feature encoder, which maps an input graph into a feature representation as follows:

$$\mathbf{f}_{\mathcal{G}} = GNN_{\text{graph}}(\mathcal{G}; \mathbf{\Theta}_1), \tag{5.50}$$

where GNN_{graph} is the graph neural network model to learn graph-level representations. It often consists of graph filtering and graph pooling layers. $\mathbf{f}_{\mathcal{G}} \in \mathbb{R}^{1 \times d_{out}}$ is the produced graph-level representation. This graph-level representation is then utilized to perform the graph classification as:

$$\mathbf{z}_{\mathcal{G}} = \text{softmax}(\mathbf{f}_{\mathcal{G}} \mathbf{\Theta}_2), \tag{5.51}$$

where $\mathbf{\Theta}_2 \in \mathbb{R}^{d_{\text{out}} \times C}$ transforms the graph representation to the dimension of the number of classes C and $\mathbf{z}_{\mathcal{G}} \in \mathbb{R}^{1 \times C}$ denotes the predicted logits for the input graph $\mathcal{G}$. The graph $\mathcal{G}$ is typically assigned to the label with the largest logit. The entire process of graph classification can be summarized as follows:

$$\mathbf{z}_{\mathcal{G}} = f_{GNN}(\mathcal{G}; \mathbf{\Theta}), \tag{5.52}$$

where f_{GNN} is the function includes Eq. (5.50) and Eq. (5.51) and $\mathbf{\Theta}$ includes parameters $\mathbf{\Theta}_1$ and $\mathbf{\Theta}_2$. The parameter $\mathbf{\Theta}$ can be learned by minimizing the following objective:

$$\mathcal{L}_{train} = \sum_{\mathcal{G}_i \in \mathcal{D}} \ell(f_{GNN}(\mathcal{G}_i, \mathbf{\Theta}), y_i),$$

where y_i is the associated label of $\mathcal{G}_i$ and $\ell(\cdot, \cdot)$ is a loss function.

5.6 Conclusion

In this chapter, we introduce the graph neural network (GNN) frameworks for both node-focused and graph-focused tasks. Specifically, we introduce two

major components: 1) the graph filtering layer, which refines the node features; and 2) the graph pooling layer, which aims to coarsen the graph and finally generate the graph-level representation. We categorize graph filters as spectral-based and spatial-based filters, then review representative algorithms for each category and discuss the connections between these two categories. We group graph pooling as flat graph pooling and hierarchical graph pooling and introduce representative methods for each group. Finally, we present how to learn GNN parameters via downstream tasks including node classification and graph classification.

5.7 Further Reading

Besides from the graph neural network models introduced in this chapter, there are also some other attempts to learn graph-level representations for graph classification utilizing neural networks (Yanardag and Vishwanathan, 2015; Niepert et al., 2016; Lee et al., 2018). In addition to representative graph filtering and pooling operations introduced above, there are more graph filtering and and pooling methods (Li et al., 2018c; Gao et al., 2018a; Zhang et al., 2018a; Liu et al., 2019b; Velickovic et al., 2019; Morris et al., 2019; Gao et al., 2020; Yuan and Ji, 2019). Meanwhile, several surveys introduce and summarize the graph neural network models from different perspectives (Zhou et al., 2018a; Wu et al., 2020; Zhang et al., 2018c). As graph neural network research has gained increasing attention, multiple handy libraries have been designed to ease the development of graph neural network models. These packages include *Pytorch Geometric (PyG)* (Fey and Lenssen, 2019), which is developed upon PyTorch; and *Deep Graph Library (DGL)* (Wang et al., 2019e), which has various deep learning frameworks including Pytorch and Tensorflow as its backend.