Chapter 4

The Graph Neural Network Model

The first part of this book discussed approaches for learning low-dimensional embeddings of the nodes in a graph. However, the approaches we discussed primarily used a *shallow* encoding approach to generate representations of nodes. We simply trained a unique embedding for each node using an embedding lookup, and our encoder function did not depend on the input graph structure.

In this chapter we turn our focus to more complex encoder models. We will introduce the *graph neural network (GNN)* formalism, which is a general framework for defining deep neural networks on graph data. The key idea is that we want to generate representations of nodes that actually depend on the structure of the graph, as well as any feature information we might have.

The primary challenge in developing complex encoders for graph-structured data is that our usual deep learning toolbox does not apply. For example, convolutional neural networks (CNNs) are well-defined only over grid-structured inputs (e.g., images), while recurrent neural networks (RNNs) are well-defined only over sequences. To define a deep neural network over general graphs, we need to generalize both of these approaches and design a framework without such topological restrictions.

Can we simply use the adjacency matrix as input to a deep neural network? One reasonable idea for defining a deep neural network over graphs would be to simply use the adjacency matrix as input to a deep neural network. For example, to generate an embedding of an entire graph we could simply flatten the adjacency matrix and feed the result to a multilayer perceptron:

$$\mathbf{z}_{\mathcal{G}} = \text{MLP}(\mathbf{A}[1] \oplus \mathbf{A}[2] \oplus ... \oplus \mathbf{A}[|\mathcal{V}|]), \tag{4.1}$$

where $\mathbf{A}[i] \in \mathbf{R}^{|\mathcal{V}|}$ denotes a row of the adjacency matrix and we use $\oplus$ to denote vector concatenation. We saw some approaches using a variation of this idea in Chapter 2 when we discussed the autoencoder approaches SDNE and DNGR.

The issue with such approaches is that they *depend on the arbitrary ordering of nodes that we used in the adjacency matrix.* More formally, the issue is that such a model is not permutation equivariant with respect to the node ordering, which means that the output changes when we permute the rows/columns of the adjacency matrix. One key desideratum for designing neural networks over graphs is that they are permutation equivariant, and this is another challenge of learning over graphs compared to objects such as images or sequences.

4.1 Neural Message Passing

The basic graph neural network (GNN) model can be motivated in a variety of ways. The same fundamental GNN model has been derived as a generalization of convolutions to non-Euclidean data [Bruna et al., 2014], as a differentiable variant of belief propagation [Dai et al., 2016], as well as by analogy to classic graph isomorphism tests [Hamilton et al., 2017b]. Regardless of the motivation, the defining feature of a GNN is that it uses a form of *neural message passing* in which vector messages are exchanged between nodes in the graph and updated using neural networks.

In the rest of this chapter, we will detail the foundations of this neural message passing framework. We will focus on the message passing framework itself and defer discussions of training and optimizing GNN models to Chapter 5. The bulk of this chapter will describe how we can take an input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, along with a set of node features $\mathbf{X} \in \mathbb{R}^{d \times |\mathcal{V}|}$, and use this information to generate node embeddings $\mathbf{z}_u$. However, we will also discuss how the GNN framework can be used to generate embeddings for subgraphs and entire graphs.

4.1.1 Overview of the Message Passing Framework

During each message-passing iteration in a GNN, the embedding $\mathbf{h}_u^{(k)}$ corresponding to each node $u \in \mathcal{V}$ is updated according to information aggregated from u's graph neighborhood $\mathcal{N}(v)$ (Figure 4.1). This message-passing update can be expressed as follows:

$$\begin{aligned}\mathbf{h}_u^{(k+1)} &= \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})\right) && (4.2)\\ &= \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}\right), && (4.3)\end{aligned}$$

where UPDATE and AGGREGATE are arbitrary differentiable functions (i.e., neural networks) and $\mathbf{m}_{\mathcal{N}(u)}$ is the "message" that is aggregated from u's graph neighborhood $\mathcal{N}(u)$. We use superscripts to distinguish the embeddings and

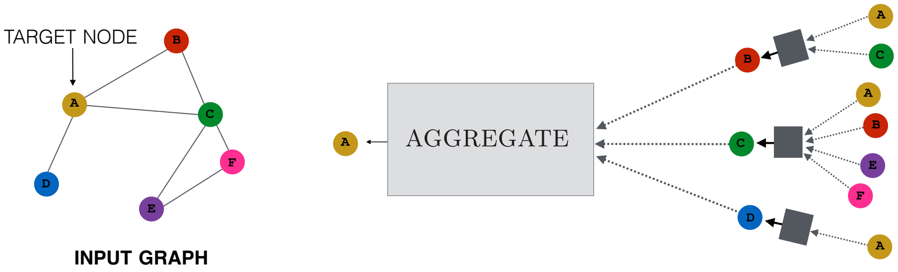

Figure 4.1: Overview of how a single node aggregates messages from its local neighborhood. The model aggregates messages from A's local graph neighbors (i.e., B, C, and D), and in turn, the messages coming from these neighbors are based on information aggregated from their respective neighborhoods, and so on. This visualization shows a two-layer version of a message-passing model.

functions at different iterations of message passing.[1]

At each iteration of the GNN, the AGGREGATE function takes as input the set of embeddings of the nodes in u's graph neighborhood $\mathcal{N}(u)$ and generates a message $\mathbf{m}_{\mathcal{N}(u)}$ based on this aggregated neighborhood information. The update function UPDATE then combines the message $\mathbf{m}_{\mathcal{N}(u)}$ with the previous embedding $\mathbf{h}_u$ of node u to generate the updated embedding. The initial embeddings at $k = 0$ are set to the input features for all the nodes, i.e., $\mathbf{h}_u^{(0)} = \mathbf{x}_u, \forall u \in \mathcal{V}$. After running K iterations of the GNN message passing, we can use the output of the final layer to define the embeddings for each node, i.e.,

$$\mathbf{z}_u = \mathbf{h}_u^{(K)}, \forall u \in \mathcal{V}. \tag{4.4}$$

Note that since the AGGREGATE function takes a *set* as input, GNNs defined in this way are permutation equivariant by design.

4.1.2 Motivations and Intuitions

The basic intuition behind the GNN message-passing framework is straightforward: at each iteration, every node aggregates information from its local neighborhood, and as these iterations progress each node embedding contains more and more information from further reaches of the graph. To be precise: after the first iteration ($k = 1$), every node embedding contains information from its 1-hop neighborhood, i.e., every node embedding contains information about the features of its immediate graph neighbors, which can be reached by a path of length 1 in the graph; after the second iteration ($k = 2$) every node embedding contains information from its 2-hop neighborhood; and in general, after k iterations every node embedding contains information about its k-hop neighborhood.

[1] The different iterations of message passing are also sometimes known as the different "layers" of the GNN.

But what kind of "information" do these node embeddings actually encode? Generally, this information comes in two forms. On the one hand there is *structural* information about the graph. For example, after k iterations of GNN message passing, the embedding $\mathbf{h}_u^{(k)}$ of node u might encode information about the degrees of all the nodes in u's k-hop neighborhood. This structural information can be useful for many tasks. For instance, when analyzing molecular graphs, we can use degree information to infer atom types and different *structural motifs* such as benzene rings (Figure 1.5).

In addition to structural information, the other key kind of information captured by GNN node embedding is *feature-based.* We generally assume that all the nodes in our graph have features or attributes associated with them, which we use to initialize the GNN message passing. After t iterations of GNN message passing, the embeddings for each node thus also encode information about all the features in their t-hop neighborhood. This local feature-aggregation behaviour of GNNs is analogous to the behavior of the convolutional kernels in convolutional neural networks (CNNs). However, whereas CNNs aggregate feature information from spatially-defined patches in an image, GNNs aggregate information based on local graph neighborhoods. We will explore the connection between GNNs and convolutions in more detail in Chapter 6.

4.1.3 The Basic GNN

So far, we have discussed the GNN framework in a relatively abstract fashion as a series of message-passing iterations using UPDATE and AGGREGATE functions (Equation 4.2). In order to translate the abstract GNN framework defined in Equation 4.2 into something we can implement, we must give concrete instantiations to these UPDATE and AGGREGATE functions. We begin here with the most basic GNN framework, which is a simplification of the original GNN model proposed by Scarselli et al. [2009].

The basic GNN message passing is defined as

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}_{\text{self}}^{(k)}\mathbf{h}_u^{(k-1)} + \mathbf{W}_{\text{neigh}}^{(k)} \sum_{v\in\mathcal{N}(u)} \mathbf{h}_v^{(k-1)} + \mathbf{b}^{(k)}\right), \tag{4.5}$$

where $\mathbf{W}_{\text{self}}^{(k)}, \mathbf{W}_{\text{neigh}}^{(tk} \in \mathbb{R}^{d^{(k)}\times d^{(k-1)}}$ are trainable parameter matrices and σ denotes an elementwise non-linearity (e.g., a tanh or ReLU). The bias term $\mathbf{b} \in \mathbb{R}^{d^{(k)}}$ is often ommitted for notational simplicity but can be important to achieve strong performance.

The message passing in the basic GNN framework is analgous to a standard multi-layer perceptron (MLP) or Elman-style recurrent neural network (Elman RNN), as it relies on linear operations followed by a single elementwise non-linearity. We first sum the messages incoming from the neighbors; then, we combine the neighborhood information with the node's previous embedding using a linear combination; and finally, we apply an elementwise non-linearity.

We can equivalently define the basic GNN through the UPDATE and AGGREGATE functions:

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v\in\mathcal{N}(u)} \mathbf{h}_v, \tag{4.6}$$

$$\text{UPDATE}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = \sigma\left(\mathbf{W}_{\text{self}}\mathbf{h}_u + \mathbf{W}_{\text{neigh}}\mathbf{m}_{\mathcal{N}(u)}\right), \tag{4.7}$$

where we recall that we use

$$\mathbf{m}_{\mathcal{N}(u)} = \mathbf{m}_{\mathcal{N}(u)} \tag{4.8}$$

as a shorthand to denote the message that has been aggregated from u's graph neighborhood. Note also that we ommitted the superscript denoting the iteration in the above equations, which we will often do for notational brevity. In general, the parameters $\mathbf{W}_{\text{self}}, \mathbf{W}_{\text{neigh}}$ and $\mathbf{b}$ can be shared across the GNN message passing iterations or trained separately.

Node vs. graph-level equations In the description of the basic GNN model above, we defined the core message-passing operations at the node level. We will use this convention for the bulk of this Chapter and this book as a whole. However, it is important to note that many GNNs can also be succinctly defined using graph-level equations. In the case of a basic GNN, we can write the graph-level definition of the model as follows:

$$\mathbf{H}^{(t)} = \sigma\left(\mathbf{A}\mathbf{H}^{(t-1)}\mathbf{W}^{(t)}_{\text{neigh}} + \mathbf{H}^{(t-1)}\mathbf{W}^{(t)}_{\text{self}}\right), \tag{4.9}$$

where $\mathbf{H}^{(t)} \in \mathbb{R}^{|V|\times d}$ denotes the matrix of node representations at layer t in the GNN (with each node corresponding to a row in the matrix), $\mathbf{A}$ is the graph adjacency matrix, and we have omitted the bias term for notational simplicity. While this graph-level representation is not easily applicable to all GNN models—such as the attention-based models we discuss below—it is often more succinct and also highlights how many GNNs can be efficiently implemented using a small number of sparse matrix operations.

4.1.4 Message Passing with Self-loops

As a simplification of the neural message passing approach it is common to add self-loops to the input graph and omit the explicit update step. In this approach we define the message passing simply as

$$\mathbf{h}_u^{(k)} = \text{AGGREGATE}(\{\mathbf{h}_v^{(t-1)}, \forall v \in \mathcal{N}(u) \cup \{u\}\}), \tag{4.10}$$

where now the aggregation is taken over the set $\mathcal{N}(u) \cup \{u\}$, i.e., the node's neighbors as well as the node itself. The benefit of this approach is that we no longer need to define an explicit update function, as the update is implicitly defined through the aggregation method. Simplifying the message passing in this way can often alleviate overfitting, but it also severely limits the expressivity of the GNN, as the information coming the node's neighbours cannot be differentiated from the information from the node itself.

In the case of the basic GNN, adding self-loops is equivalent to sharing parameters between the $\mathbf{W}_{\text{self}}$ and $\mathbf{W}_{\text{neigh}}$ matrices, which gives the following graph-level update:

$$\mathbf{H}^{(t)} = \sigma\left((\mathbf{A}+\mathbf{I})\mathbf{H}^{(t-1)}\mathbf{W}^{(t)}\right). \tag{4.11}$$

In the following chapters we will refer to this as the *self-loop* GNN approach.

4.2 Generalized Neighborhood Aggregation

The basic GNN model outlined in Equation (4.5) can achieve strong performance, and its theoretical capacity is well-understood (see Chapter 6). However, just like a simple MLP or Elman RNN, the basic GNN can be improved upon and generalized in many ways. Here, we discuss how the AGGREGATE operator can be generalized and improved upon, with the following section (Section 4.3) providing an analogous discussion for the UPDATE operation.

4.2.1 Neighborhood Normalization

The most basic neighborhood aggregation operation (Equation 4.6) simply takes the sum of the neighbor embeddings. One issue with this approach is that it can be unstable and highly sensitive to node degrees. For instance, suppose node u has 100× as many neighbors as node u' (i.e., a much higher degree), then we would reasonably expect that $\|\sum_{v\in\mathcal{N}(u)}\mathbf{h}_v\| >> \|\sum_{v'\in\mathcal{N}(u')}\mathbf{h}_{v'}\|$ (for any reasonable vector norm $\|\cdot\|$), and this drastic difference in magnitude can lead to numerical instabilities as well as difficulties for optimization.

One simple solution to this problem is to simply normalize the aggregation operation based upon the degrees of the nodes involved. The simplest approach is to just take an average rather than sum:

$$\mathbf{m}_{\mathcal{N}(u)} = \frac{\sum_{v\in\mathcal{N}(u)}\mathbf{h}_v}{|\mathcal{N}(u)|}, \tag{4.12}$$

but researchers have also found success with other normalization factors, such as the following approach employed by Kipf and Welling [2016]:

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v\in\mathcal{N}(u)}\frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}. \tag{4.13}$$

The intuition behind the Kipf normalization is that the model should down-weight messages coming from neighbors with very high degrees, since the "signal" coming from these high-degree neighbors may be less precise and informative. For example, in a citation graph—the kind of data that Kipf and Welling

[2016] analyzed—information from very high-degree nodes (i.e., papers) may not be very useful for inferring community membership in the graph, since these papers can be cited thousands of times across diverse subfields. Kipf normalization can also be motivated based on spectral graph theory. In particular, combining the Kipf normalized aggregation (Equation 4.13) along with the basic GNN update function (Equation 4.7) results in a first-order approximation of a spectral graph convolution, and we expand on this connection in Chapter 6.

Graph convolutional networks (GCNs)

One of the most popular baseline graph neural network models—the graph convolutional network (GCN)—employs the Kipf normalized aggregation as well as the self-loop update approach. The GCN model thus defines the message passing function as

$$\mathbf{h}_v^{(k)} = \sigma\left(\mathbf{W}^{(k)} \sum_{v \in \mathcal{N}(u)\cup\{u\}} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}\right). \tag{4.14}$$

This approach was first outlined by Kipf and Welling [2016] and has proved to be one of the most popular and effective baseline GNN architectures.

To normalize or not to normalize? Proper normalization can be essential to achieve stable and strong performance when using a GNN. It is important to note, however, that normalization can also lead to a loss of information. For example, after normalization, it can be hard (or even impossible) to use the learned embeddings to distinguish between nodes of different degrees, and various other structural graph features can be obscured by normalization. In fact, a basic GNN using the normalized aggregation operator in Equation (4.12) is provably less powerful than the basic sum aggregator in Equation (4.6) (see Chapter 6). The use of normalization is thus an application-specific question. Usually, normalization is most helpful in tasks where node feature information is far more useful than structural information, or where there is a very wide range of node degrees that can lead to instabilities during optimization.

4.2.2 Set Aggregators

Neighborhood normalization can be a useful tool to improve GNN performance, but can we do more to improve the AGGREGATE operator? Is there perhaps something more sophisticated than just summing over the neighbor embeddings?

The neighborhood aggregation operation is fundamentally a set function. We are given a set of neighbor embeddings $\{\mathbf{h}_v, \forall v \in \mathcal{N}(u)\}$ and must map this set to a single vector $\mathbf{m}_{\mathcal{N}(u)}$. The fact that $\{\mathbf{h}_v, \forall v \in \mathcal{N}(u)\}$ is a *set* is in fact quite important: there is no natural ordering of a nodes' neighbors, and any aggregation function we define must thus be *permutation invariant*.

Set pooling

One principled approach to define an aggregation function is based on the theory of permutation invariant neural networks. For example, Zaheer et al. [2017] shows that an aggregation function with the following form is a *universal set function approximator*:

$$\mathbf{m}_{\mathcal{N}(u)} = \mathbf{MLP}_\theta \left(\sum_{v \in N(u)} \mathbf{MLP}_\phi(\mathbf{h}_v) \right), \tag{4.15}$$

where as usual we use $\mathbf{MLP}_\theta$ to denote an arbitrarily deep multi-layer perceptron parameterized by some trainable parameters θ. In other words, the theoretical results in Zaheer et al. [2017] show that any permutation-invariant function that maps a set of embeddings to a single embedding can be approximated to an arbitrary accuracy by a model following Equation (4.15).

Note that the theory presented in Zaheer et al. [2017] employs a sum of the embeddings after applying the first MLP (as in Equation 4.15). However, it is possible to replace the sum with an alternative reduction function, such as an element-wise maximum or minimum, as in Qi et al. [2017], and it is also common to combine models based on Equation (4.15) with the normalization approaches discussed in Section 4.2.1, as in the `GraphSAGE-pool` approach [Hamilton et al., 2017b].

Set pooling approaches based on Equation (4.15) often lead to small increases in performance, though they also introduce an increased risk of overfitting, depending on the depth of the MLPs used. If set pooling is used, it is common to stick with MLPs that have only a single hidden layer, since these models are sufficient to satisfy the theory, but are not so overparameterized so as to risk catastrophic overfitting.

Janossy pooling

Set pooling approaches to neighborhood aggregation essentially just add extra layers of MLPs on top of the more basic aggregation architectures discussed in Section 4.1.3. This idea is simple, but is known to increase the theoretical capacity of GNNs. However, there is another alternative approach, termed *Janossy pooling*, that is also provably more powerful than simply taking a sum or mean of the neighbor embeddings [Murphy et al., 2018].

Recall that the challenge of neighborhood aggregation is that we must use a *permutation-invariant* function, since there is no natural ordering of a node's neighbors. In the set pooling approach (Equation 4.15), we achieved this permutation invariance by relying on a sum, mean, or element-wise max to reduce the set of embeddings to a single vector. We made the model more powerful by combining this reduction with feed-forward neural networks (i.e., MLPs). Janossy pooling employs a different approach entirely: instead of using a permutation-invariant reduction (e.g., a sum or mean), we apply a *permutation-sensitive function* and average the result over many possible permutations.

Let $\pi_i \in \Pi$ denote a function (i.e., permutation) that maps the set $\{\mathbf{h}_v, \forall v \in \mathcal{N}(u)\}$ to a specific sequence $(\mathbf{h}_{v_1}, \mathbf{h}_{v_2}, ..., \mathbf{h}_{v_{|\mathcal{N}(u)|}})_{\pi_i}$. In other words, π_i takes the unordered set of neighbor embeddings and places these embeddings in a sequence based on some arbitrary ordering. The Janossy pooling approach then performs neighborhood aggregation by

$$\mathbf{m}_{\mathcal{N}(u)} = \mathbf{MLP}_\theta \left(\frac{1}{|\Pi|} \sum_{\pi \in \Pi} \rho_\phi \left(\mathbf{h}_{v_1}, \mathbf{h}_{v_2}, \mathbf{h}_{v_{|\mathcal{N}(u)|}} \right)_{\pi_i} \right), \tag{4.16}$$

where Π denotes a set of permutations and ρ_ϕ is a permutation-sensitive function, e.g., a neural network that operates on sequences. In practice ρ_ϕ is usually defined to be an LSTM, since LSTMs are known to be a powerful neural network architecture for sequences.

If the set of permutations Π in Equation (4.16) is equal to all possible permutations, then the aggregator in Equation (4.16) is also a universal function approximator for sets, like Equation (4.15). However, summing over all possible permutations is generally intractable. Thus, in practice, Janossy pooling employs one of two approaches:

1. Sample a random subset of possible permutations during each application of the aggregator, and only sum over that random subset.

2. Employ a *canonical* ordering of the nodes in the neighborhood set; e.g., order the nodes in descending order according to their degree, with ties broken randomly.

Murphy et al. [2018] includes a detailed discussion and empirical comparison of these two approaches, as well as other approximation techniques (e.g., truncating the length of sequence), and their results indicate that Janossy-style pooling can improve upon set pooling in a number of synthetic evaluation setups.

4.2.3 Neighborhood Attention

In addition to more general forms of set aggregation, a popular strategy for improving the aggregation layer in GNNs is to apply *attention* [Bahdanau et al., 2014]. The basic idea is to assign an attention weight or importance to each neighbor, which is used to weigh this neighbor's influence during the aggregation step. The first GNN model to apply this style of attention was Veličković et al. [2017]'s Graph Attention Network (GAT), which uses attention weights to define a weighted sum of the neighbors:

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \alpha_{u,v} \mathbf{h}_v, \tag{4.17}$$

where $\alpha_{u,v}$ denotes the attention on neighbor $v \in \mathcal{N}(u)$ when we are aggregating information at node u. In the original GAT paper, the attention weights are defined as

$$\alpha_{u,v} = \frac{\exp\left(\mathbf{a}^\top [\mathbf{W}\mathbf{h}_u \oplus \mathbf{W}\mathbf{h}_v]\right)}{\sum_{v' \in \mathcal{N}(u)} \exp\left(\mathbf{a}^\top [\mathbf{W}\mathbf{h}_u \oplus \mathbf{W}\mathbf{h}_{v'}]\right)}, \tag{4.18}$$

where $\mathbf{a}$ is a trainable attention vector, $\mathbf{W}$ is a trainable matrix, and $\oplus$ denotes the concatenation operation.

The GAT-style attention computation is known to work well with graph data. However, in principle any standard attention model from the deep learning literature at large can be used [Bahdanau et al., 2014]. Popular variants of attention include the bilinear attention model

$$\alpha_{u,v} = \frac{\exp\left(\mathbf{h}_u^\top \mathbf{W} \mathbf{h}_v\right)}{\sum_{v' \in \mathcal{N}(u)} \exp\left(\mathbf{h}_u^\top \mathbf{W} \mathbf{h}_{v'}\right)}, \tag{4.19}$$

as well as variations of attention layers using MLPs, e.g.,

$$\alpha_{u,v} = \frac{\exp\left(\mathbf{MLP}(\mathbf{h}_u, \mathbf{h}_v)\right)}{\sum_{v' \in \mathcal{N}(u)} \exp\left(\mathbf{MLP}(\mathbf{h}_u, \mathbf{h}_{v'})\right)}, \tag{4.20}$$

where the MLP is restricted to a scalar output.

In addition, while it is less common in the GNN literature, it is also possible to add multiple attention "heads", in the style of the popular Transformer architecture [Vaswani et al., 2017]. In this approach, one computes K distinct attention weights $\alpha_{u,v,k}$, using independently parametrized attention layers. The messages aggregated using the different attention weights are then transformed and combined in the aggregation step, usually with a linear projection followed by a concatenation operation, e.g.,

$$\mathbf{m}_{\mathcal{N}(u)} = [\mathbf{a}_1 \oplus \mathbf{a}_2 \oplus ... \oplus \mathbf{a}_K] \tag{4.21}$$

$$\mathbf{a}_k = \mathbf{W}_i \sum_{v \in \mathcal{N}(u)} \alpha_{u,v,k} \mathbf{h}_v \tag{4.22}$$

where the attention weights $\alpha_{u,v,k}$ for each of the K attention heads can be computed using any of the above attention mechanisms.

Graph attention and transformers GNN models with multi-headed attention (Equation 4.21) are closely related to the Transformer architecture [Vaswani et al., 2017]. Transformers are a popular architecture for both natural language processing (NLP) and computer vision, and—in the case of NLP—they have been an important driver behind large state-of-the-art NLP systems, such as BERT [Devlin et al., 2018] and XLNet [Yang et al., 2019]. The basic idea behind Transformers is to define neural network layers entirely based on the attention operation. At each layer in a Transformer, a new hidden representation is generated for every position in the input data (e.g., every word in a sentence) by using multiple attention heads to compute attention weights between all pairs of positions in the input, which are then aggregated with weighted sums based on these attention weights (in a manner analogous to Equation 4.21). In fact, the basic Transformer layer is exactly equivalent to a GNN layer using multi-headed attention (i.e., Equation 4.21) if we assume that the GNN receives a fully-connected graph as input.

This connection between GNNs and Transformers has been exploited in numerous works. For example, one implementation strategy for designing GNNs is to simply start with a Transformer model and then apply a binary adjacency mask on the attention layer to ensure that information is only aggregated between nodes that are actually connected in the graph. This style of GNN implementation can benefit from the numerous well-engineered libraries for Transformer architectures that exist. However, a downside of this approach, is that Transformers must compute the pairwise attention between all positions/nodes in the input, which leads to a quadratic $O(|\mathcal{V}|^2)$ time complexity to aggregate messages for all nodes in the graph, compared to a $O(|\mathcal{V}||\mathcal{E}|)$ time complexity for a more standard GNN implementation.

Adding attention is a useful strategy for increasing the representational capacity of a GNN model, especially in cases where you have prior knowledge to indicate that some neighbors might be more informative than others. For example, consider the case of classifying papers into topical categories based on citation networks. Often there are papers that span topical boundaries, or that are highly cited across various different fields. Ideally, an attention-based GNN would learn to ignore these papers in the neural message passing, as such "promiscuous" neighbors would likely be uninformative when trying to identify the topical category of a particular node. In Chapter 6, we will discuss how attention can influence the inductive bias of GNNs from a signal processing perspective.

4.3 Generalized Update Methods

The AGGREGATE operator in GNN models has generally received the most attention from researchers—in terms of proposing novel architectures and variations. This was especially the case after the introduction of the `GraphSAGE` framework, which introduced the idea of generalized neighbourhood aggregation. However, the core GNN message passing involves two key steps: aggregation and updating. In many ways the UPDATE operator plays an equally important role in defining the power and inductive bias of the GNN model, and in this section, we turn our attention to various different generalizations of the UPDATE operator.

In the basic GNN model, the updated hidden representation $\mathbf{h}_u^{(t)}$ of a node u is computed at each step of the message passing via a linear combination of the aggregated message $\mathbf{m}_{\mathcal{N}(u)}$ coming from the node u's neighbours as well as the node's previous hidden representation $\mathbf{h}_u^{(t-1)}$. While simple and effective in many settings, this basic update approach has serious drawbacks.

Most prominently, by linearly combining $\mathbf{m}_{\mathcal{N}(u)}$ with the $\mathbf{h}_u^{(t-1)}$, the information from the various iterations of message passing becomes "entangled" after each round of message passing. When using this simple linear combination the GNN has no easy way to disentangle local information that is specific to a particular round of message passing, from more non-local information being received in later message passing iterations. This linear entanglement can lead to (among other things) an issue known as *over-smoothing*. After a large number of iterations of message passing, the representations for all the nodes in the graph become very similar to one another.

Lastly, the simple style of update used in the basic GNN approach can lead to numerical instabilities (e.g., exploding gradients) due to the repeated application of matrix multiplications at each layer. This issue is analagous to the numerical instabilities observed for basic recurrent neural network (RNN) models, such as the Elman RNN.

Over-smoothing and neighbourhood influence The issue of *over-smoothing* in GNNs can be formalized by defining the influence of each nodes input feature $\mathbf{h}_u^{(0)} = \mathbf{x}_u$ on the final layer embedding of all the other nodes in the graph, i.e, $\mathbf{h}_v^{(K)}, \forall v \in \mathcal{V}$. In particular, for any pair of nodes u and v we can quantify the influence of node u on node v in the GNN by examining the magnitude of the corresponding Jacobian matrix [Xu et al., 2018b]:

$$I_K(u,v) = \mathbf{1}^\top \left(\frac{\partial \mathbf{h}_v^{(K)}}{\partial \mathbf{h}_u^{(0)}} \right) \mathbf{1}, \tag{4.23}$$

where $\mathbf{1}$ is a vector of all ones. The $I_K(u,v)$ value uses the sum of the entries in the Jacobian matrix $\frac{\partial \mathbf{h}_v^{(K)}}{\partial \mathbf{h}_u^{(0)}}$ as a measure of how much the initial embedding of node u influences the final embedding of node v in the GNN.

Given this definition of influence, Xu et al. [2018b] prove the following:

Theorem 3. *For any GNN model using a self-loop update approach and an aggregation function of the form*

$$\textsc{aggregate}(\{\mathbf{h}_v, \forall v \in \mathcal{N}(u) \cup \{u\}\}) = \frac{1}{f_n(|\mathcal{N}(u) \cup \{u\}|)} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \mathbf{h}_v, \tag{4.24}$$

where $f : \mathbb{R}^+ \to \mathbb{R}^+$ is an arbitrary differentiable normalization function, we have that

$$I_K(u,v) \propto p_{\mathcal{G},K}(u|v), \tag{4.25}$$

where $p_{\mathcal{G},K}(u|v)$ denotes the probability of visiting node v on a length-K random walk starting from node u.

This theorem is a direct consequence of Theorem 1 in Xu et al. [2018b]. It states that when using a K-layer GCN-style model, the influence of node u and node v is proportional the probability of reaching node v on a K-step random walk starting from node u. An important consequence of this, however, is that as $K \to \infty$ the influence of every node approaches the stationary distribution of random walks over the graph, meaning that local neighborhood information is lost. Moreover, in many real-world graphs—which contain high-degree nodes and resemble so-called "expander" graphs—it only takes $k = O(\log(|\mathcal{V}|)$ steps for the random walk starting from any node to converge to an almost-uniform distribution [Hoory et al., 2006].

Theorem 1 applies directly to models using a self-loop update approach, but the result can also be extended in asympotic sense for the basic GNN update (i.e., Equation 4.7) as long as $\|\mathbf{W}^{(k)}_{\text{self}}\| < \|\mathbf{W}^{(k)}_{\text{neigh}}\|$ at each layer k. Thus, when using simple GNN models—and especially those with the self-loop update approach—building deeper models can actually hurt performance. As more layers are added we lose information about local neighborhood structures and our learned embeddings become over-smoothed, approaching an almost-uniform distribution.

4.3.1 Concatenation and Skip-Connections

The core issue with the basic GNN update function is that it uses a linear combination to update the hidden node representation at each iteration of message passing. A natural way to alleviate this issue is to introduce various different concatenation or skip-connections, which try to directly preserve information from previous rounds of message passing during the update step.

These concatenation and skip-connection methods can be used in conjunction with most other GNN update approaches. Thus, for the sake of generality, we will use $\text{UPDATE}_{\text{base}}$ to denote the base update function that we are building upon (e.g., we can assume that $\text{UPDATE}_{\text{base}}$ is given by Equation 4.7), and we will define various skip-connection updates on top of this base function.

One of the simplest skip connection updates employs a concatenation to preserve more node-level information during message passing:

$$\text{UPDATE}_{\text{concat}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = [\text{UPDATE}_{\text{base}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) \oplus \mathbf{h}_u], \tag{4.26}$$

where we simply concatenate the output of the base update function with the nodes previous-layer representation. Again, the key intuition here is that we encourage the model to disentangle information during message passing—separating the information coming from the neighbors (i.e., $\mathbf{m}_{\mathcal{N}(u)}$) from the current representation of each node (i.e., $\mathbf{h}_u$).

The concatenation based skip connection was proposed in the `GraphSAGE` framework and was the first work to highlight the possible benefits of these kinds of modifications to the update function [Hamilton et al., 2017a]. However, in addition to concatenation, we can also employ other forms of skip-connections, such as the linear interpolation method proposed by Pham et al. [2017]:

$$\text{UPDATE}_{\text{interpolate}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = \boldsymbol{\alpha}_1 \odot \text{UPDATE}_{\text{base}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) + \boldsymbol{\alpha}_2 \odot \mathbf{h}_u, \tag{4.27}$$

where $\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2 \in [0,1]^d$ are gating vectors with $\boldsymbol{\alpha}_2 = \mathbf{1} - \boldsymbol{\alpha}_1$ and $\odot$ denotes elementwise multiplication. In this approach, the final updated representation is a linear interpolation between the previous representation and the representation that was updated based on the neighborhood information. The gating parameters $\boldsymbol{\alpha}_1$ can be learned jointly with the model in a variety of ways. For example, Pham et al. [2017] generates $\boldsymbol{\alpha}_1$ as the output of a separate single-layer GNN, which takes the current hidden-layer representations as features. But other simpler approaches could also be employed, such as simply directly learning $\boldsymbol{\alpha}_1$ parameters for each message passing layer or using an MLP on the current node representations to generate these gating parameters.

In general, these concatenation and residual connections are simple strategies that can help to alleviate the over-smoothing issue in GNNs, while also improving the numerical stability of optimization. Indeed, similar to the utility of residual connections in convolutional neural networks (CNNs) [He et al., 2016], applying these approaches to GNNs can facilitate the training of much deeper models. In practice these techniques tend to be most useful for node classification tasks with moderately deep GNNs (e.g., 2-5 layers), and they excel on tasks that exhibit homophily, i.e., where the prediction for each node is strongly related to the features of its local neighborhood.

4.3.2 Gated Updates

In the previous section we discussed skip-connection and residual connection approaches that bear strong analogy to techniques used in computer vision to build deeper CNN architectures. In a parallel line of work, researchers have also drawn inspiration from the gating methods used to improve the stability and learning ability of recurrent neural networks (RNNs). In particular, one way to view the GNN message passing algorithm is that the aggregation function is receiving an "observation" from the neighbors, which is then used to update the "hidden state' of each node. In this view, we can directly apply methods used to update the hidden state of RNN architectures based on observations. For instance, one of the earliest GNN architectures [Li et al., 2015] defines the update function as

$$\mathbf{h}_u^{(k)} = \text{GRU}(\mathbf{h}_u^{(k-1)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}), \tag{4.28}$$

where GRU denotes the update equation of the gated recurrent unit (GRU) cell [Cho et al., 2014]. Other approaches have employed updates based on the LSTM architecture [Selsam et al., 2018].

In general, any update function defined for RNNs can be employed in the context of GNNs. We simply replace the hidden state argument of the RNN update function (usually denoted $\mathbf{h}^{(t)}$ with the node's hidden state, and we replace the observation vector (usually denoted $\mathbf{x}^{(t)}$) with the message aggregated from the local neighborhood. Importantly, the parameters of this RNN-style update are always shared across nodes (i.e., we use the same LSTM or GRU cell to update each node). In practice, researchers usually share the parameters of the update function across the message passing layers of the GNN as well.

These gated updates are very effective at facilitating deep GNN architectures (e.g., more than 10 layers) and preventing over-smoothing. Generally, they are most useful for GNN applications where the prediction task requires complex reasoning over the global structure of the graph, such as applications for program analysis [Li et al., 2015] or combinatorial optimization [Selsam et al., 2018].

4.3.3 Jumping Knowledge Connections

In the preceding sections we have been implicitly assuming that we are using the output of the final layer of the GNN. In other words, we have been assuming that the node representations $\mathbf{z}_u$ that we use in a downstream task are equal to final layer node embeddings in the GNN:

$$\mathbf{z}_u = \mathbf{h}_u^{(K)}, \forall u \in \mathcal{V}. \tag{4.29}$$

This assumption is made by many GNN approaches, and the limitations of this strategy motivated much of the need for residual and gated updates to limit over-smoothing.

However, a complimentary strategy to improve the quality of the final node representations is to simply leverage the representations *at each layer of message passing*, rather than only using the final layer output. In this approach we define the final node representations $\mathbf{z}_u$ as

$$\mathbf{z}_u = f_{\text{JK}}(\mathbf{h}_u^{(0)} \oplus \mathbf{h}_u^{(1)} \oplus ... \oplus \mathbf{h}_u^{(K)}), \tag{4.30}$$

where f_{JK} is an arbitrary differentiable function. This strategy is known as adding *jumping knowledge (JK) connections* and was first proposed and analyzed by [Xu et al., 2018b]. In many applications the function f_{JK} can simply be defined as the identity function, meaning that we just concatenate the node embeddings from each layer, but Xu et al. [2018b] also explore other options such as max-pooling approaches and LSTM attention layers. While simple, this approach often leads to consistent improvements across a wide-variety of tasks and is a generally useful strategy to employ.

4.4 Edge Features and Multi-relational GNNs

So far our discussion of GNNs and neural message passing has implicitly assumed that we have simple graphs. However, there are many applications where the graphs in question are multi-relational or otherwise heterogenous (e.g., knowledge graphs). In this section, we review some of the most popular strategies that have been developed to accommodate such data.

4.4.1 Relational Graph Neural Networks

The first approach proposed to address this problem is commonly known as the *Relational Graph Convolutional Network (RGCN)* approach [Schlichtkrull et al., 2017]. In this approach we augment the aggregation function to accommodate multiple relation types by specifying a separate transformation matrix per relation type:

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{\tau \in \mathcal{R}} \sum_{v \in \mathcal{N}_\tau(u)} \frac{\mathbf{W}_\tau \mathbf{h}_v}{f_n(\mathcal{N}(u), \mathcal{N}(v))}, \tag{4.31}$$

where f_n is a normalization function that can depend on both the neighborhood of the node u as well as the neighbor v being aggregated over. Schlichtkrull et al. [2017] discuss several normalization strategies to define f_n that are analagous to those discussed in Section 4.2.1. Overall, the multi-relational aggregation in RGCN is thus analagous to the basic a GNN approach with normalization, but we separately aggregate information across different edge types.

Parameter sharing

One drawback of the naive RGCN approach is the drastic increase in the number of parameters, as now we have one trainable matrix per relation. In certain applications—such as applications on knowledge graphs with many distinct relations—this increase in parameters can lead to overfitting and slow learning. Schlichtkrull et al. [2017] propose a scheme to combat this issue by parameter sharing with basis matrices, where

$$\mathbf{W}_\tau = \sum_{i=1}^{b} \alpha_{i,\tau} \mathbf{B}_i. \tag{4.32}$$

In this basis matrix approach, all the relation matrices are defined as linear combinations of b basis matrices $\mathbf{B}_1, ..., \mathbf{B}_b$, and the only relation-specific parameters are the b combination weights $\alpha_{1,\tau}, ..., \alpha_{b,\tau}$ for each relation τ. In this basis sharing approach, we can thus rewrite the full aggregation function as

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{\tau \in \mathcal{R}} \sum_{v \in \mathcal{N}_\tau(u)} \frac{\boldsymbol{\alpha_\tau} \times_1 \mathcal{B} \times_2 \mathbf{h}_v}{f_n(\mathcal{N}(u), \mathcal{N}(v))}, \tag{4.33}$$

where $\mathcal{B} = (\mathbf{B}_1, ..., \mathbf{B}_b)$ is a tensor formed by stacking the basis matrices, $\boldsymbol{\alpha_\tau} = (\alpha_{1,\tau}, ..., \alpha_{b,\tau})$ is a vector containing the basis combination weights for relation τ, and $\times_i$ denotes a tensor product along mode i. Thus, an alternative view of the parameter sharing RGCN approach is that we are learning an embedding for each relation, as well a tensor that is shared across all relations.

Extensions and variations

The RGCN architecture can be extended in many ways, and in general, we refer to approaches that define separate aggregation matrices per relation as *relational graph neural networks*. For example, a variation of this approach without parameter sharing is deployed by Zitnik et al. [2018] to model a multi-relational dataset relating drugs, diseases and proteins, and a similar strategy is leveraged by Marcheggiani and Titov [2017] to analyze linguistic dependency graphs. Other works have found success combining the RGCN-style aggregation with attention [Teru and Hamilton, 2019].

In a recent extension, Vashishth et al. [2019] also proposed to augment this general strategy by leveraging the multi-relational decoders discussed in Chapter 3. In particular, they define the aggregation as

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{\tau \in \mathcal{R}} \sum_{v \in \mathcal{N}_\tau(u)} \frac{\mathbf{W}_\tau \times \text{DEC}(\mathbf{h}_v, \tau, \cdot)}{f_n(\mathcal{N}(u), \mathcal{N}(v))}, \tag{4.34}$$

where we use $\text{DEC}(\mathbf{h}_v, \tau, \cdot) : \mathbb{R}^d \times \mathcal{R} \times \mathbb{R}^d \to \mathbb{R}^d$ to denote a *partial* application of one of the multi-relational decoders defined in Chapter 3. For example, if we were using the TransE decoder then

$$\text{DEC}_{\text{TransE}}(\mathbf{h}_v, \tau, \cdot) = \mathbf{h}_v + \mathbf{r}_\tau. \tag{4.35}$$

Similarly, with the RESCAL decoder we would have

$$\text{DEC}_{\text{RESCAL}}(\mathbf{h}_v, \tau, \cdot) = \mathbf{R}_\tau \mathbf{h}_v. \tag{4.36}$$

In general, the partial application of these multi-relational decoders is defined as the transformation that is (implicitly) applied to one of the input node embeddings before computing the final score. Vashishth et al. [2019] discuss several instantiations of this approach and note performance improvements compared to a basic RGCN.

4.4.2 Attention and Feature Concatenation

The relational GNN approach, where we define a separate aggregation parameter per relation, is applicable for multi-relational graphs and cases where we have discrete edge features. To accommodate cases where we have more general forms of edge features, we can leverage these features in attention or by concatenating this information with the neighbor embeddings during message passing. For example, given any base aggregation approach $\text{AGGREGATE}_{\text{base}}$ one simple strategy to leverage edge features is to define a new aggregation function as

$$\mathbf{m}_{\mathcal{N}(u)} = \text{AGGREGATE}_{\text{base}}(\{\mathbf{h}_v \oplus \mathbf{e}_{(u,\tau,v)}, \forall v \in \mathcal{N}(u)\}), \tag{4.37}$$

where $\mathbf{e}_{(u,\tau,v)}$ denotes an arbitrary vector-valued feature for the edge (u, τ, v). This approach is simple and general, and has seen recent success with attention-based approaches as the base aggregation function [Sinha et al., 2019].

4.5 Graph Pooling

The neural message passing approach produces a set of *node* embeddings, but what if we want to make predictions at the *graph* level? In other words, we have been assuming that the goal is to learn node representations $\mathbf{z}_u, \forall u \in \mathcal{V}$, but what if we to learn an embedding $\mathbf{z}_{\mathcal{G}}$ for the entire graph $\mathcal{G}$? This task is often referred to as *graph pooling*, since our goal is to pool together the node embeddings in order to learn an embedding of the entire graph.

Set pooling approaches

Similar to the AGGREGATE operator, the task of graph pooling can be viewed as a problem of learning over sets. We want to design a pooling function f_p, which maps a set of node embeddings $\{\mathbf{z}_1, ..., \mathbf{z}_{|V|}\}$ to an embedding $\mathbf{z}_{\mathcal{G}}$ that represents the full graph. Indeed, any of the approaches discussed in Section 4.2.2 for learning over sets of neighbor embeddings can also be employed for pooling at the graph level, and the theoretical analysis of Zaheer et al. [2017] on set-based learning still applies here.

In practice, there are two approaches that are most commonly applied for learning graph-level embeddings via set pooling. The first approach is simply to take a sum (or mean) of the node embeddings:

$$\mathbf{z}_{\mathcal{G}} = \frac{\sum_{v \in \mathcal{V}} \mathbf{z}_c}{f_n(|\mathcal{V}|)}, \tag{4.38}$$

where f_n is some normalizing function (e.g., the identity function). While quite simple, pooling based on the sum or mean of the node embeddings is often sufficient for applications involving small graphs (e.g., predicting properties from molecular graphs).

The second popular set-based approach uses a combination of LSTMs and attention to pool in a more complex manner, in a manner inspired by the work of Vinyals et al. [2015]. In this pooling approach we iterate a series of attention-based aggregations defined by the following set of equations, which are iterated for $t = 1, ..., T$ steps:

$$\begin{align}
\mathbf{q}_i &= \text{LSTM}(\mathbf{o}_{t-1}, \mathbf{q}_{t-1}), \tag{4.39}\\
e_{v,t} &= f_a(\mathbf{z}_v, \mathbf{q}_t), \forall v \in \mathcal{V}, \tag{4.40}\\
a_{v,t} &= \frac{\exp(e_{v,i})}{\sum_{u \in \mathcal{V}} e_{u,t}}, \forall v \in \mathcal{V}, \tag{4.41}\\
\mathbf{o}_t &= \sum_{v \in \mathcal{V}} a_{v,t} \mathbf{z}_v. \tag{4.42}
\end{align}$$

In the above equations, the $\mathbf{q}_t$ vector represents a *query vector* for the attention at each iteration t. In Equation (4.40), the query vector is used to compute an attention score over each node using an attention function $f_a : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ (e.g., a dot product), and this attention score is then normalized in Equation (4.41). Finally, in Equation (4.42) a weighted sum of the node embeddings is computed based on the attention weights, and this weighted sum is used to update the query vector using an LSTM update (Equation 4.39). Generally the $\mathbf{q}_0$ and $\mathbf{o}_0$ vectors are initialized with all-zero values, and after iterating Equations 4.39-4.42 for T iterations, an embedding for the full graph is computed as

$$\mathbf{z}_{\mathcal{G}} = \mathbf{o}_1 \oplus \mathbf{o}_2 \oplus ... \oplus \mathbf{o}_T. \tag{4.43}$$

This approach represents a sophisticated architecture for attention-based pooling over a set, and it has become a popular pooling method in many graph-level classification tasks.

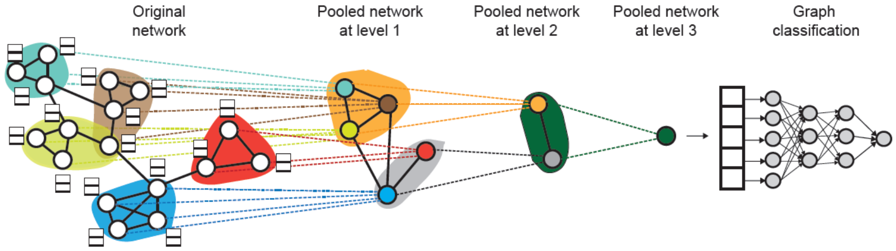
Original network
Pooled network at level 1
Pooled network at level 2
Pooled network at level 3
Graph classification

Figure 4.2: Illustration of the graph coarsening approach. At each hierarchical level, the nodes in the graph are assigned to different clusters based on their learned embeddings. A new graph is then formed based on these clusters, where the nodes in the graph at layer l correspond to clusters in layer $l-1$. GNNs are used to process the graphs at the different hierarchical layers. Image taken from Ying et al. [2018b].

Graph coarsening approaches

One limitation of the set pooling approaches is that they do not exploit the structure of the graph. While it is reasonable to consider the task of graph pooling as simply a set learning problem, there can also be benefits from exploiting the graph topology at the pooling stage. One popular strategy to accomplish this is to perform graph *clustering or coarsening* as a means to pool the node representations.

In these style of approaches, we assume that we have some clustering function

$$\mathbf{f}_c \rightarrow \mathcal{G} \times \mathbb{R}^{|V| \times d} \rightarrow \mathbb{R}^{+|V| \times c}, \tag{4.44}$$

which maps all the nodes in the graph to an assignment over c clusters. In particular, we presume that this function outputs an assignment matrix $\mathbf{S} = f_c(\mathcal{G}, \mathbf{Z})$, where $\mathbf{S}[u, i] \in \mathbb{R}^+$ denotes the strength of the association between node u and cluster i. One simple example of f_c function would be spectral clustering approach described in Chapter 1, where the cluster assignment is based on the spectral decomposition of the graph adjacency matrix. Note that using a simple spectral clustering approach would ignore the node embedding (i.e., $\mathbf{Z}$) input to the cluster function $f_c(\mathcal{G}, \mathbf{Z})$ and generate a hard cluster assignment with a binary assignment matrix $\mathbf{S} \in \mathbb{Z}^{|V| \times c}$. However, as a more complex definition of f_c one can actually employ another GNN to learn the cluster assignments, where the node embeddings $\mathbf{Z}$ are used as input to the GNN [Ying et al., 2018b]. In this GNN approach, one would generally apply a softmax function after the final GNN layer to output probabilistic cluster assignments.

Regardless of the approach used to generate the cluster assignment matrix $\mathbf{S}$, the key idea of graph coarsening approaches is that we then use this matrix to *coarsen* the graph. In particular, we use the assignment matrix to compute a new coarsened adjacency matrix

$$\mathbf{A}^{\text{new}} = \mathbf{S}^\top \mathbf{A} \mathbf{S} \in \mathbb{R}^{+c \times c} \tag{4.45}$$

and a new set of node features

$$\mathbf{X}^{\text{new}} = \mathbf{S}^{\top}\mathbf{X} \in \mathbb{R}^{c \times d}. \tag{4.46}$$

Thus, this new adjacency matrix now represents the strength of association (i.e., the edges) between the clusters in the graph, and the new feature matrix represents the aggregated embeddings for all the nodes assigned to each cluster. We can then run a GNN on this coarsened graph and repeat the entire coarsening process for a number of iterations, where the size of the graph is decreased at each step. The final representation of the graph is then computed by a set pooling over the embeddings of the nodes in a sufficiently coarsened graph. Figure 4.2 illustrates this idea.

This coarsening based approach is inspired by the pooling approaches used in convolutional neural networks (CNNs), and it relies on the intuition that we can build hierarchical GNNs that operate on different granularities of the input graph. In practice, these coarsening approaches can lead to strong performance, but they can also be unstable and difficult to train. For example, in order to have the entire learning process be end-to-end differentiable the clustering functions f_c must be differentiable, which rules out most off-the-shelf clustering algorithms such as spectral clustering. Cangea et al. [2018] provides an insightful comparison and discussion of different approaches in this domain, highlighting the benefits of learning sparse cluster assignments and some of the challenges in implementing these coarsening approaches.