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 7, we will discuss how attention can influence the inductive bias of GNNs from a signal processing perspective.

5.3 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 [Hamilton et al., 2017b]. However, GNN message passing involves two key steps: aggregation and updating, and in many ways the UPDATE operator plays an equally important role in defining the power and inductive bias of the GNN model.

So far, we have seen the basic GNN approach—where the update operation involves a linear combination of the node's current embedding with the message from its neighbors—as well as the self-loop approach, which simply involves adding a self-loop to the graph before performing neighborhood aggregation. In this section, we turn our attention to more diverse generalizations of the UPDATE operator.

Over-smoothing and neighbourhood influence One common issue with GNNs—which generalized update methods can help to address—is known as *over-smoothing*. The essential idea of over-smoothing is that after several iterations of GNN message passing, the representations for all the nodes in the graph can become very similar to one another. This tendency is especially common in basic GNN models and models that employ the self-loop update approach. Over-smoothing is problematic because it makes it impossible to build deeper GNN models—which leverage longer-term dependencies in the graph—since these deep GNN models tend to just generate over-smoothed embeddings.

This issue of over-smoothing in GNNs can be formalized by defining the influence of each node's 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., 2018]:

$$I_K(u, v) = \mathbf{1}^\top \left(\frac{\partial \mathbf{h}_v^{(K)}}{\partial \mathbf{h}_u^{(0)}} \right) \mathbf{1}, \tag{5.25}$$

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. [2018] prove the following:

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

$$\text{AGGREGATE}(\{\mathbf{h}_v, \forall v \in \mathcal{N}(u) \cup \{u\}\}) = \frac{1}{f_n(|\mathcal{N}(u) \cup \{u\}|)} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \mathbf{h}_v, \tag{5.26}$$

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

$$I_K(u, v) \propto p_{\mathcal{G},K}(u|v), \tag{5.27}$$

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. [2018]. It states that when we are 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 3 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 5.9) as long as $\|\mathbf{W}_{\text{self}}^{(k)}\| < \|\mathbf{W}_{\text{neigh}}^{(k)}\|$ 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.

5.3.1 Concatenation and Skip-Connections

As discussed above, over-smoothing is a core issue in GNNs. Over-smoothing occurs when node-specific information becomes "washed out" or "lost" after several iterations of GNN message passing. Intuitively, we can expect over-smoothing in cases where the information being aggregated from the node neighbors during message passing begins to dominate the updated node representations. In these cases, the updated node representations (i.e., the $\mathbf{h}_u^{(k+1)}$

vectors) will depend too strongly on the incoming message aggregated from the neighbors (i.e., the $\mathbf{m}_{\mathcal{N}(u)}$ vectors) at the expense of the node representations from the previous layers (i.e., the $\mathbf{h}_u^{(k)}$ vectors). A natural way to alleviate this issue is to use vector concatenations 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 5.9), 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:

$$\text{UPDATE}_{\text{concat}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = [\text{UPDATE}_{\text{base}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) \oplus \mathbf{h}_u], \tag{5.28}$$

where we simply concatenate the output of the base update function with the node's 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, which was one of the first works 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]:

$$\text{UPDATE}_{\text{interpolate}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = \boldsymbol{\alpha}_1 \circ \text{UPDATE}_{\text{base}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) + \boldsymbol{\alpha}_2 \odot \mathbf{h}_u, \tag{5.29}$$

where $\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2 \in [0,1]^d$ are gating vectors with $\boldsymbol{\alpha}_2 = \mathbf{1} - \boldsymbol{\alpha}_1$ and $\circ$ 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] generate $\boldsymbol{\alpha}_1$ as the output of a separate single-layer GNN, which takes the current hidden-layer representations as features. However, 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.

5.3.2 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

$$\mathbf{h}_u^{(k)} = \text{GRU}(\mathbf{h}_u^{(k-1)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}), \tag{5.30}$$

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., 2019].

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., 2019].

5.3.3 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:

$$\mathbf{z}_u = \mathbf{h}_u^{(K)}, \forall u \in \mathcal{V}. \tag{5.31}$$

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.