9
Beyond GNNs: More Deep Models on Graphs

9.1 Introduction

There are many traditional deep models, such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), deep autoencoders, and generative adversarial networks (GANs). These models have been designed for different types of data. For example, CNNs can process regular grid-like data such as images, while RNNs can deal with sequential data such as text. They have also been designed in different settings. For instance, a large number of labeled data is needed to train good CNNs and RNNs (or the supervised setting), while autoencoders and GANs can extract complex patterns with only unlabeled data (or the unsupervised setting). These different architectures enable deep learning techniques to apply to many fields such as computer vision, natural language processing, data mining, and information retrieval. We have introduced various graph neural networks (GNNs) for simple and complex graphs in the previous chapters. However, these models have been developed only for certain graph tasks such as node classification and graph classification; and they often require labeled data for training. Thus, efforts have been made to adopt more deep architectures to graph-structured data. Autoencoders have been extended to graph-structured data for node representation learning (Wang et al., 2016; Kipf and Welling, 2016b; Pan et al., 2018). Deep generative models, such as variational autoencoder and generative adversarial networks, have also been adapted to graph-structured data for node representation learning (Kipf and Welling, 2016b; Pan et al., 2018; Wang et al., 2018a) and graph generation (Simonovsky and Komodakis, 2018; De Cao and Kipf, 2018). These deep graph models have facilitated a broader range of graph tasks under different settings beyond the capacity of GNNs; and have greatly advanced deep learning techniques on graphs. This chapter aims to cover more deep models on graphs, including deep autoencoders, variational autoencoders, recurrent neural networks, and generative adversarial networks.

9.2 Autoencoders on Graphs

Autoencoders, which have been introduced in Section 3.5, can be regarded as unsupervised learning models to obtain compressed low-dimensional representations for input data samples. Autoencoders have been adopted to learn low-dimensional node representations (Wang et al., 2016; Kipf and Welling, 2016b; Pan et al., 2018). In (Wang et al., 2016), the neighborhood information of each node is utilized as the input to be reconstructed; hence the learned low-dimensional representation can preserve the structural information of the nodes. Both the encoder and decoder are modeled with feedforward neural networks as introduced in Section 3.5. In (Kipf and Welling, 2016b; Pan et al., 2018), the graph neural network model, which utilizes both the input node features and graph structure, is adopted as the encoder to encode node into low-dimensional representations. These encoded node representations are then employed to reconstruct the graph structural information. Next, we briefly introduce these two types of graph autoencoders for learning low-dimensional node representations.

In (Wang et al., 2016), for each node $v_i \in \mathcal{V}$, its corresponding row in the adjacency matrix of the graph $\mathbf{a}_i = \mathbf{A}_i$ is served as the input of the encoder to obtain its low-dimensional representation as:

$$\mathbf{z}_i = f_{enc}(\mathbf{a}_i; \mathbf{\Theta}_{enc}),$$

where f_{enc} is the encoder, which is modeled with a feedworward neural network parameterized by $\mathbf{\Theta}_{enc}$. Then $\mathbf{z}_i$ is utilized as the input to the decoder, which aims to reconstruct $\mathbf{a}_i$ as:

$$\tilde{\mathbf{a}}_i = f_{dec}(\mathbf{z}_i; \mathbf{\Theta}_{dec}),$$

where f_{dec} is the decoder and $\mathbf{\Theta}_{dec}$ denotes its parameters. The reconstruction loss can thus be built by constraining $\tilde{\mathbf{a}}_i$ to be similar to $\mathbf{a}_i$ for all nodes in $\mathcal{V}$ as:

$$\mathcal{L}_{enc} = \sum_{v_i \in \mathcal{V}} \|\mathbf{a}_i - \tilde{\mathbf{a}}_i\|_2^2.$$

Minimizing the above reconstruction loss can "compress" the neighborhood information into the low-dimensional representation $\mathbf{z}_i$. The pairwise similarity between the neighborhood of nodes (i.e. the similarity between the input) is not explicitly be captured. However, as the autoencoder (the parameters) is shared by all the nodes, the encoder is expected to map those nodes who have similar inputs to similar node representations, which implicitly preserves the similarity. The above reconstruction loss might be problematic due to the inherent sparsity of the adjacency matrix $\mathbf{A}$. A large portion of the elements in $\mathbf{a}_i$ is 0, which might lead the optimization process to easily overfit to reconstructing the 0 elements. To solve this issue, more penalty is imposed to the reconstruction error of the non-zero elements by modifying the reconstruction loss as:

$$\mathcal{L}_{enc} = \sum_{v_i \in \mathcal{V}} \| (\mathbf{a}_i - \tilde{\mathbf{a}}_i) \odot \mathbf{b}_i \|_2^2,$$

where $\odot$ denotes the Hadamard product, $\mathbf{b}_i = \{b_{i,j}\}_{j=1}^{|\mathcal{V}|}$ with $b_{i,j} = 1$ when $A[i,j] = 0$ and $b_{i,j} = \beta > 1$ when $A[i,j] \neq 0$. β is a hyperparameter to be tuned. Furthermore, to directly enforce connected nodes to have similar low-dimensional representations, a regularization loss is introduced as:

$$\mathcal{L}_{con} = \sum_{v_i, v_j \in \mathcal{V}} \mathbf{A}_{i,j} \cdot \|\mathbf{z}_i - \mathbf{z}_j\|_2^2.$$

Finally, regularization loss on the parameters of encoder and decoder is also included in the objective, which leads to the following loss to be minimized:

$$\mathcal{L} = \mathcal{L}_{enc} + \lambda \cdot \mathcal{L}_{con} + \eta \cdot \mathcal{L}_{reg},$$

where $\mathcal{L}_{reg}$ denotes the regularization on the parameters, which can be expressed as:

$$\mathcal{L}_{reg} = \|\mathbf{\Theta}_{enc}\|_2^2 + \|\mathbf{\Theta}_{dec}\|_2^2. \tag{9.1}$$

The graph autoencoder model introduced above can only utilize the graph structure but not be able to incorporate node features when they are available. In (Kipf and Welling, 2016b), the graph neural network model is adopted as the encoder, which utilizes both the graph structural information and node features. Specifically, the encoder is modeled as:

$$\mathbf{Z} = f_{GNN}(\mathbf{A}, \mathbf{X}; \mathbf{\Theta}_{GNN}),$$

where f_{GNN} is the encoder which is a graph neural network model. In (Kipf and Welling, 2016b), the GCN-Filter is adopted to build the encoder. The decoder is to reconstruct the graph, which includes the adjacency matrix $\mathbf{A}$ and the attribute matrix $\mathbf{X}$. In (Kipf and Welling, 2016b), only the adjacency matrix $\mathbf{A}$ is used as the target for reconstruction. Specifically, the adjacency matrix can

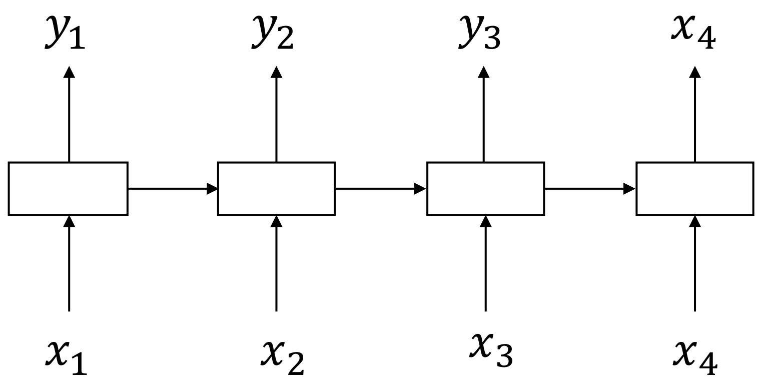

Figure 9.1 A illustrative sequence

be reconstructed from the encoded representations $\mathbf{Z}$ as:

$$\hat{\mathbf{A}} = \sigma(\mathbf{Z}\mathbf{Z}^T),$$

where $\sigma(\cdot)$ is the sigmoid function. The low-dimensional representations $\mathbf{Z}$ can be learned by minimizing the reconstruction error between $\hat{\mathbf{A}}$ and $\mathbf{A}$. The objective can be modeled as:

$$-\sum_{v_i, v_j \in \mathcal{V}} \left(\mathbf{A}_{i,j} \log \hat{\mathbf{A}}_{i,j} + \left(1 - \mathbf{A}_{i,j}\right) \log \left(1 - \hat{\mathbf{A}}_{i,j}\right)\right),$$

which can be viewed as the cross-entropy loss between $\mathbf{A}$ and $\hat{\mathbf{A}}$.

9.3 Recurrent Neural Networks on Graphs

Recurrent neural networks in Section 3.4 have been originally designed to deal with sequential data and have been generalized to learning representations for graph-structured data in recent years. In (Tai et al., 2015), Tree-LSTM is introduced to generalize the LSTM model to tree-structured data. A tree can be regarded as a special graph, which does not have any loops. In (Liang et al., 2016), Graph-LSTM is proposed to further extend the Tree-LSTM to generic graphs. In this section, we first introduce the Tree-LSTM and then discuss Graph-LSTM.

As shown in Figure 9.1, a sequence can be regarded as a specific tree where each node (except for the first one) has only a single child, i.e., its previous node. The information flows from the first node to the last node in the sequence.

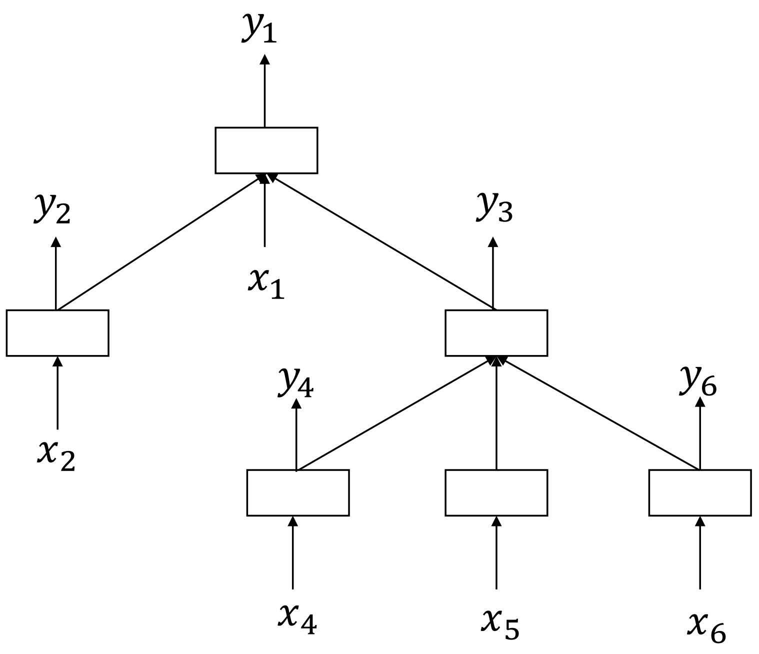

Figure 9.2 A illustrative tree

Hence, as introduced in Section 3.4.2 and illustrated in Figure 3.13, the LSTM model composes the hidden state of a given node in a sequence by using the input at this node and also the hidden state from its previous node. However, in comparison, as shown in Figure 9.2, in a tree, a node can have an arbitrary number of child nodes. In a tree, the information is assumed to always flow from the child nodes to the parent node. Hence, when composing the hidden state for a node, we need to utilize its input and the hidden states of its child nodes. Based on this intuition, the Tree-LSTM model is proposed to deal with tree-structured data. To introduce the Tree-LSTM model, we follow the same notations as those in Section 3.4.2. Specifically, for node v_k in the tree, we use $\mathbf{x}^{(k)}$ as its input, $\mathbf{h}^{(k)}$ as its hidden state, $\mathbf{C}^{(k)}$ as its cell memory and $\mathcal{N}_c(v_k)$ as the set of its child nodes. Given a tree, the Tree-LSTM model composes the hidden state of node v_k as:

$$\tilde{\mathbf{h}}^{(k)} = \sum_{v_j \in \mathcal{N}_c(v_k)} \mathbf{h}^{(j)} \tag{9.2}$$

$$\mathbf{f}_{kj} = \sigma(\mathbf{W}_f \cdot \mathbf{x}^{(k)} + \mathbf{U}_f \cdot \mathbf{h}^{(j)} + \mathbf{b}_f) \quad \textbf{for } v_j \in \mathcal{N}_c(v_k) \tag{9.3}$$

$$\mathbf{i}_k = \sigma(\mathbf{W}_i \cdot \mathbf{x}^{(k)} + \mathbf{U}_i \cdot \tilde{\mathbf{h}}^{(k)} + \mathbf{b}_i) \tag{9.4}$$

$$\mathbf{o}_k = \sigma(\mathbf{W}_o \cdot \mathbf{x}^{(k)} + \mathbf{U}_o \cdot \tilde{\mathbf{h}}^{(k)} + \mathbf{b}_o) \tag{9.5}$$

$$\tilde{\mathbf{C}}^{(k)} = tanh(\mathbf{W}_c \cdot \mathbf{x}^{(k)} + \mathbf{U}_c \cdot \tilde{\mathbf{h}}^{(k)} + \mathbf{b}_c) \tag{9.6}$$

$$\mathbf{C}^{(k)} = \mathbf{i}_t \odot \tilde{\mathbf{C}}^{(k)} + \sum_{v_j \in \mathcal{N}_c(v_k)} \mathbf{f}_{kj} \odot \mathbf{C}^{(j)} \tag{9.7}$$

$$\mathbf{h}^{(k)} = \mathbf{o}_t \odot tanh(\mathbf{C}^{(k)}). \tag{9.8}$$

We next briefly describe the operation procedure of the Tree-LSTM model. The hidden states of the child nodes of v_k are aggregated to generate $\tilde{\mathbf{h}}^{(k)}$ as shown in Eq. (9.2). The aggregated hidden state $\tilde{\mathbf{h}}^{(k)}$ is utilized to generate the input gate, output gate and candidate cell memory in Eq. (9.4), Eq. (9.5) and Eq. (9.6), respectively. In Eq. (9.3), for each child $v_j \in \mathcal{N}_c(v_k)$, a corresponding forget gate is generated to control the information flow from this child node to v_k, when updating the memory cell for v_k in Eq. (9.7). Finally, in Eq. (9.8), the hidden state for node v_k is updated.

Unlike trees, there are often loops in generic graphs. Hence, there is no natural ordering for nodes in the generic graphs as that in trees. Breadth-First Search (BFS) and Depth-First Search (DFS) are the possible ways to define an ordering for the nodes as proposed in (Liang et al., 2016). Furthermore, the ordering of nodes can also be defined according to the specific application at hand. After obtaining an ordering for the nodes, we can use similar operations, as shown from Eq. (9.2) to Eq. (9.8) to update the hidden state and cells for these nodes following the obtained ordering. The major difference is that in undirected graphs, Eq. (9.2) aggregates hidden states from all neighbors $\mathcal{N}(v_k)$ of node v_k, while Tree-LSTM in Eq. (9.2) only aggregates information from the child nodes of v_k. Furthermore, the hidden states of some nodes in neighbors $\mathcal{N}(v_k)$ may not have been updated. In this case, the pre-updated hidden states are utilized in the aggregation process.

9.4 Variational Autoencoders on Graphs

Variational Autoencoder (VAE) is a kind of generative models, which aims to model the probability distribution of a given dataset $\mathcal{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$. It is also a latent variable model, which generates samples from latent variables. Given a latent variable $\mathbf{z}$ sampled from a standard normal distribution $p(\mathbf{z})$, we want to learn a latent model, which can generate similar samples as these in the given data with the following probability:

$$p(\mathbf{x}|\mathbf{z};\mathbf{\Theta}) = \mathcal{N}(\mathbf{x}|f(\mathbf{z};\mathbf{\Theta}),\sigma^2 \cdot \mathbf{I}), \tag{9.9}$$

where $\mathbf{\Theta}$ is the parameter to be learned and $\mathbf{x}$ is a generated sample in the same domain as the given data. For example, if the input data samples are images, we want to generate images as well. $f(\mathbf{z};\Theta)$ is a deterministic function, which maps the latent variable $\mathbf{z}$ to the mean of the probability of the generative model in Eq. (9.9). Note that, in practice, the probability of the generated samples is not necessary to be Gaussian but can be other distributions according to the applications. Here, for convenience, we use Gaussian distribution as an example, which is adopted when generating images in computer vision tasks. To ensure that the generative model in Eq. (9.9) is representative of the given data $\mathcal{X}$, we need to maximize the following log likelihood for each sample $\mathbf{x}_i$ in $\mathcal{X}$:

$$\log p(\mathbf{x}_i) = \log \int p(\mathbf{x}_i|\mathbf{z};\mathbf{\Theta})p(\mathbf{z})d\mathbf{z} \text{ for } \mathbf{x}_i \in \mathcal{X}. \tag{9.10}$$

However, the integral in Eq. (9.10) is intractable. Furthermore, the true posterior $p(\mathbf{z}|\mathbf{x};\Theta)$ is also intractable, which hinders the possibility of using the EM algorithm. To remedy this issue, an inference model $q(\mathbf{z}|\mathbf{x};\mathbf{\Phi})$ parameterized with $\mathbf{\Phi}$, which is an approximation of the intractable true posterior $p(\mathbf{z}|\mathbf{x};\Theta)$, is introduced (Kingma and Welling, 2013). Usually, $q(\mathbf{z}|\mathbf{x};\mathbf{\Phi})$ is modeled as a Gaussian distribution $q(\mathbf{z}|\mathbf{x};\mathbf{\Phi}) = \mathcal{N}(\mu(\mathbf{x};\mathbf{\Phi}),\Sigma(\mathbf{x};\mathbf{\Phi}))$, where the mean and covariance matrix are learned through some deterministic function parameterized with $\mathbf{\Phi}$. Then, the log-likelihood in Eq. (9.10) can be rewritten as:

$$\log p(\mathbf{x}_i) = D_{KL}(q(\mathbf{z}|\mathbf{x};\mathbf{\Phi})\|p(\mathbf{z}|\mathbf{x};\mathbf{\Theta})) + \mathcal{L}(\mathbf{\Theta},\mathbf{\Phi};\mathbf{x}_i),$$

where $\mathcal{L}(\mathbf{\Theta},\mathbf{\Phi};\mathbf{x}_i)$ is called the variational lower bound of the log-likelihood of $\mathbf{x}_i$ as the KL divergence of the approximate and the true posterior (the first term on the right hand side) is non-negative. Specifically, the variational lower bound can be written as:

$$\mathcal{L}(\mathbf{\Theta},\mathbf{\Phi};\mathbf{x}_i) = \underbrace{\mathbb{E}_{q(\mathbf{z}|\mathbf{x_i};\mathbf{\Phi})}\left[\log p(\mathbf{x}_i|\mathbf{z};\mathbf{\Theta})\right]}_{reconstruction} - \underbrace{D_{KL}(q(\mathbf{z}|\mathbf{x_i};\mathbf{\Phi})\|p(\mathbf{z}))}_{regularization}. \tag{9.11}$$

Instead of maximizing the log-likelihood in Eq. (9.10) for samples in $\mathcal{X}$, we aim to differentiate and maximize the variational lower bound in Eq. (9.11) with respect to both $\mathbf{\Theta}$ and $\mathbf{\Phi}$. Note that minimizing the negation of the variational lower bound $-\mathcal{L}(\mathbf{\Theta}, \mathbf{\Phi}; \mathbf{x}_i)$ resembles the process of the classic autoencoder as introduced in Section 3.5, which gives the name "Variational Autoencoder" to the model. Specifically, the first term on the right hand side of Eq. (9.11) can be regarded as the reconstruction process, where $q(\mathbf{z}|\mathbf{x_i}; \mathbf{\Phi})$ is the encoder (the inference model) and $p(\mathbf{x}_i|\mathbf{z}; \mathbf{\Theta})$ is the decoder (the generative model). Different from the classical autoencoder, where the encoder maps a given input to a representation, this encoder $q(\mathbf{z}|\mathbf{x_i}; \mathbf{\Phi})$ maps an input $\mathbf{x}_i$ into a latent Gaussian distribution. Maximizing the first term on the right hand side of Eq. (9.11) can be viewed as minimizing the distance between the input $\mathbf{x}_i$ and the decoded mean $f(\mathbf{z}; \mathbf{\Theta})$ of $p(\mathbf{x}_i|\mathbf{z}; \mathbf{\Theta})$. Meanwhile, the second term on the right hand side of Eq. (9.11) can be regarded as a regularization term, which enforces the approximated posterior $q(\mathbf{z}|\mathbf{x_i}; \mathbf{\Phi})$ to be close to the prior distribution $p(\mathbf{z})$. After training, the generative model $p(\mathbf{x}_i|\mathbf{z}; \mathbf{\Theta})$ can be utilized to generate samples that are similar to the ones in the given data while the latent variable can be sampled from the standard Gaussian distribution $p(\mathbf{z})$.

9.4.1 Variational Autoencoders for Node Representation Learning

In (Kipf and Welling, 2016b), the variational autoencoder is adopted to learn node representations on graphs. The inference model is to encode each node into a multivariate Gaussian distribution and the joint distribution of all nodes are shown below:

$$\begin{aligned} q(\mathbf{Z}|\mathbf{X}, \mathbf{A}; \mathbf{\Phi}) &= \prod_{v_i \in \mathcal{V}} q\left(\mathbf{z}_i|\mathbf{X}, \mathbf{A}; \mathbf{\Phi}\right) \\ \text{with } q\left(\mathbf{z}_i|\mathbf{X}, \mathbf{A}; \mathbf{\Phi}\right) &= \mathcal{N}\left(\mathbf{z}_i|\boldsymbol{\mu}_i, \operatorname{diag}\left(\boldsymbol{\sigma}_i^2\right)\right), \end{aligned} \tag{9.12}$$

where $\boldsymbol{\mu}_i$ and $\boldsymbol{\sigma}_i$ are the mean and variance learned through deterministic graph neural network models as follows:

$$\begin{aligned} \boldsymbol{\mu} &= \text{GNN}(\mathbf{X}, \mathbf{A}; \mathbf{\Phi}_\mu), \\ \log \boldsymbol{\sigma} &= \text{GNN}(\mathbf{X}, \mathbf{A}; \mathbf{\Phi}_\sigma), \end{aligned}$$

where $\boldsymbol{\mu}$ is a matrix with $\boldsymbol{\mu}_i$ as its i-th row while $\boldsymbol{\sigma}$ is a vector with $\boldsymbol{\sigma}_s$ as its s-th element. The parameters $\mathbf{\Phi}_\mu$ and $\mathbf{\Phi}_\sigma$ can be summarized as $\mathbf{\Phi}$ in Eq. (9.12). Specifically, in (Kipf and Welling, 2016b), the GCN-Filter is adopted as the graph neural network model to build the inference model. The generative model, which is to generate (reconstruct) the adjacent matrix of the graph, is modeled with the inner product between the latent variables $\mathbf{Z}$ as follows:

$$p(\mathbf{A}|\mathbf{Z}) = \prod_{i=1}^{N}\prod_{j=1}^{N} p\left(\mathbf{A}[i,j]|\mathbf{z}_i,\mathbf{z}_j\right),$$
$$\text{with } p\left(\mathbf{A}[i,j]=1|\mathbf{z}_i,\mathbf{z}_j\right) = \sigma\left(\mathbf{z}_i^\top \mathbf{z}_j\right),$$

where $\mathbf{A}[i,j]$ is the ij-th element of the adjacency matrix $\mathbf{A}$ and $\sigma(\cdot)$ is the sigmoid function. Note that there are no parameters in the generative model. The variational parameters in the inference model are learned by optimizing the variational lower bound as shown below:

$$\mathcal{L} = \mathbb{E}_{q(\mathbf{Z}|\mathbf{X},\mathbf{A};\mathbf{\Phi})}[\log p(\mathbf{A}|\mathbf{Z})] - \mathrm{KL}[q(\mathbf{Z}|\mathbf{X},\mathbf{A};\mathbf{\Phi})\|p(\mathbf{Z})],$$

where $p(\mathbf{Z}) = \prod_i p(\mathbf{z_i}) = \prod_i \mathcal{N}(\mathbf{z}_i|0,\mathbf{I})$ is a Gaussian prior enforced to the latent variables $\mathbf{Z}$.

9.4.2 Variational Autoencoders for Graph Generation

In the task of graph generation, we are given a set of graph $\{\mathcal{G}_i\}$ and try to generate graphs that are similar to them. Variational Autoencoder has been adopted to generate small graphs such as molecular graphs (Simonovsky and Komodakis, 2018). Specifically, given a graph $\mathcal{G}$, the inference model $q(\mathbf{z}|\mathcal{G};\mathbf{\Phi})$ aims to map it to a latent distribution. Meanwhile, the decoder can be represented by a generative model $p(\mathcal{G}|\mathbf{z};\mathbf{\Theta})$. Both $\mathbf{\Phi}$ and $\mathbf{\Theta}$ are parameters, which can be learned by optimizing the following variational lower bound of the log-likelihood $\log p(\mathcal{G};\mathbf{\Theta}))$ of $\mathcal{G}$:

$$\mathcal{L}(\Phi,\Theta;\mathcal{G}) = \mathbb{E}_{q(\mathbf{z}|\mathcal{G};\mathbf{\Phi})}\left[\log p(\mathcal{G}|\mathbf{z};\mathbf{\Theta})\right] - D_{\mathrm{KL}}\left[q(\mathbf{z}|\mathcal{G};\mathbf{\Phi})\|p(\mathbf{z})\right], \tag{9.13}$$

where $p(\mathbf{z}) = \mathcal{N}(\mathbf{0},\mathbf{I})$ is a Gaussian prior on $\mathbf{z}$. Next, we describe the details about the encoder (inference model) $q(\mathbf{z}|\mathcal{G};\mathbf{\Phi})$, the decoder (generative model) $p(\mathcal{G}|\mathbf{z};\mathbf{\Theta})$ and finally discuss how to evaluate $\mathbb{E}_{q(\mathbf{z}|\mathcal{G};\mathbf{\Phi})}\left[\log p(\mathcal{G}|\mathbf{z};\mathbf{\Theta})\right]$.

Encoder: The Inference Model

In (Simonovsky and Komodakis, 2018), the goal is to generate small graphs with few nodes. For example, molecular graphs are usually quite small. Furthermore, these graphs are assumed to be associated with node and edge attributes. In the case of molecular graphs, the node and edge attributes indicate the type of nodes and edges, which are encoded as 1-hot vectors. Specifically, a graph $\mathcal{G} = \{\mathbf{A},\mathbf{F},\mathbf{E}\}$ can be represented by its adjacency matrix $\mathbf{A} \in \{0,1\}^{N\times N}$, node attributes $\mathbf{F} \in \{0,1\}^{N\times t_n}$ and also edge attributes $\mathbf{E} \in \{0,1\}^{N\times N\times t_e}$. Usually, in molecular graphs, the number of nodes N is in the order of tens. $\mathbf{F}$ is the matrix indicting the attribute (type) of each node, where t_n is the number of node types. Specifically, the i-th row of $\mathbf{F}$ is a one-hot vector indicating the type of i-th node. Similarly, $\mathbf{E}$ is a tensor indicating the types of edges. Note that, the graph is typically not complete and thus, we do not have $N \times N$ edges. Hence, in $\mathcal{E}$, the "one-hot" vectors corresponding to the non-existing edges are $\mathbf{0}$ vectors. To fully utilize the given graph information, the graph neural network model with pooling layers is utilized to model the encoder as:

$$\begin{aligned} &q(\mathbf{z}|\mathcal{G};\mathbf{\Phi}) = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2), \\ &\boldsymbol{\mu} = pool(\text{GNN}_{\mu}(\mathcal{G})); \quad \log \boldsymbol{\sigma} = pool(\text{GNN}_{\sigma}(\mathcal{G})), \end{aligned}$$

where the mean and variance are learned by graph neural network models. In detail, the ECC-Filter as introduced in Section 5.3.2 is utilized to build the graph neural network model to learn the node representations while the gated global pooling introduced in Section 5.4.1 is adopted to pool the node representations to generate the graph representation.

Decoder: The Generative Model

The generative model aims to generate a graph $\mathcal{G}$ given a latent representation $\mathbf{z}$. In other words, it is to generate the three matrices $\mathbf{A}$, $\mathbf{E}$, and $\mathbf{F}$. In (Simonovsky and Komodakis, 2018), the size of the graphs to be generated is limited to a small number k. In detail, the generative model is asked to output a probabilistic fully connected graph with k nodes $\widetilde{\mathcal{G}} = \{\widetilde{\mathbf{A}}, \widetilde{\mathbf{E}}, \widetilde{\mathbf{F}}\}$. In this probabilistic graph, the existence of nodes and edges are modeled as Bernoulli variables while the types of nodes and edges are modeled as Multinomial variables. Specifically, the predicted fully connected adjacency matrix $\widetilde{\mathbf{A}} \in [0,1]^{k \times k}$ contains both the node existence probabilities at the diagonal elements $\widetilde{\mathbf{A}}[i,i]$ and the edge existence probabilities at the off-diagonal elements $\widetilde{\mathbf{A}}[i,j]$. The edge type probabilities are contained in the tensor $\widetilde{\mathbf{E}} \in \mathbb{R}^{k \times k \times t_e}$. The node type probabilities are expressed in the matrix $\widetilde{\mathbf{F}} \in \mathbb{R}^{k \times t_n}$. Different architectures can be used for modelling the generative model. In (Simonovsky and Komodakis, 2018), a simple feedforward network model, which takes the latent variable $\mathbf{z}$ as input and outputs three matrices in its last layer, is adopted. Sigmoid function is applied to obtain $\widetilde{\mathbf{A}}$, which demonstrates the probability of the existence of nodes and edges. Edge-wise and node-wise softmax functions are applied to obtain $\widetilde{\mathbf{E}}$ and $\widetilde{\mathbf{F}}$, respectively. Note that the obtained probabilistic graph $\widetilde{\mathcal{G}}$ can be regarded as the generative model, which can be expressed as:

$$p(\mathcal{G}|\mathbf{z};\mathbf{\Theta}) = p(\mathcal{G}|\widetilde{\mathcal{G}}),$$

where

$$\widetilde{\mathcal{G}} = \mathrm{MLP}(\mathbf{z}; \boldsymbol{\Theta}).$$

The *MLP*() denotes the feedforward network model.

Reconstruction loss

To optimize Eq. (9.13), it is remained to evaluate $\mathbb{E}_{q(\mathbf{z}|\mathcal{G};\boldsymbol{\Phi})}\left[\log p(\mathcal{G}|\mathbf{z};\boldsymbol{\Theta})\right]$, which can be regarded evaluating how close the input graph $\mathcal{G}$ and the reconstructed probabilistic graph $\widetilde{\mathcal{G}}$ are. Since there is no particular node ordering in graphs, comparing two graphs is difficult. In (Simonovsky and Komodakis, 2018), the max pooling matching algorithm (Cho et al., 2014b) is adopted to find correspondences $\mathbf{P} \in \{0,1\}^{k\times N}$ between the input graph $\mathcal{G}$ and the $\widetilde{\mathcal{G}}$. It is based on the similarity between the nodes from the two graphs, where N denotes the number of nodes in $\mathcal{G}$ and k is the number of nodes in $\widetilde{\mathcal{G}}$. Specifically, $\mathbf{P}_{i,j} = 1$ only when the i-th node in $\widetilde{\mathcal{G}}$ is aligned with the j-th node in the original graph $\mathcal{G}$, otherwise $\mathbf{P}_{i,j} = 0$. Given the alignment matrix $\mathbf{P}$, the information in the two graphs can be aligned to be comparable. In particular, the input adjacency matrix can be mapped to the predicted graph as $\mathbf{A}' = \mathbf{P}\mathbf{A}\mathbf{P}^T$, while the predicted node types and edge types can be mapped to the input graph as $\widetilde{\mathbf{F}}' = \mathbf{P}^T\widetilde{\mathbf{F}}$ and $\widetilde{\mathbf{E}}'_{:,:,l} = \mathbf{P}^T\widetilde{\mathbf{E}}_{:,:,l}\mathbf{P}$. Then, $\mathbb{E}_{q(\mathbf{z}|\mathcal{G};\boldsymbol{\Phi})}\left[\log p(\mathcal{G}|\mathbf{z};\boldsymbol{\Theta})\right]$ is estimated with a single latent variable $\mathbf{z}$ sampled from $q(\mathbf{z}|\mathcal{G})$ as follows:

$$\mathbb{E}_{q(\mathbf{z}|\mathcal{G};\boldsymbol{\Phi})}\left[\log p(\mathcal{G}|\mathbf{z};\boldsymbol{\Theta})\right] \sim \log p(\mathcal{G}|\mathbf{z};\boldsymbol{\Theta}) = \log p\left(\mathbf{A}',\mathbf{E},\mathbf{F}|\widetilde{\mathbf{A}},\widetilde{\mathbf{E}}'\widetilde{\mathbf{F}}'\right),$$

where $p\left(\mathbf{A}',\mathbf{E},\mathbf{F}|\widetilde{\mathbf{A}},\widetilde{\mathbf{E}}'\widetilde{\mathbf{F}}'\right)$ can be modeled as:

$$\begin{aligned} &p\left(\mathbf{A}',\mathbf{E},\mathbf{F}|\widetilde{\mathbf{A}},\widetilde{\mathbf{E}}'\widetilde{\mathbf{F}}'\right) \\ &= \lambda_A \log p(\mathbf{A}'|\widetilde{\mathbf{A}}) + \lambda_E \log p(\mathbf{E}|\widetilde{\mathbf{E}}') + \lambda_F \log p(\mathbf{F}|\widetilde{\mathbf{F}}'), \end{aligned} \tag{9.14}$$

where $\lambda_{\mathbf{A}}, \lambda_E$ and λ_F are hyperparameters. Specifically, the three terms are the log-likelihood of the $\mathbf{A}'$, $\mathbf{E}'$ and $\mathbf{F}'$, respectively, which can be modeled as the negation of the cross-entropy between $\mathbf{A}'$ and $\widetilde{\mathbf{A}}$; $\mathbf{E}$ and $\widetilde{\mathbf{E}}'$; $\mathbf{E}$ and $\widetilde{\mathbf{E}}'$, respectively. In detail, they can be formally stated as:

$$
\begin{aligned}
\log p(\mathbf{A}'|\widetilde{\mathbf{A}}) =& \frac{1}{k}\sum_{i=1}^{k}\left[\mathbf{A}'_{i,i}\log\widetilde{\mathbf{A}}_{i,i} + (1-\mathbf{A}'_{i,i})\log(1-\widetilde{\mathbf{A}}_{i,i})\right] \\
&+ \frac{1}{k(k-1)}\sum_{i\neq j}^{k}\left[\mathbf{A}'_{i,j}\log\widetilde{\mathbf{A}}_{i,j} + (1-\mathbf{A}'_{i,j})\log(1-\widetilde{\mathbf{A}}_{i,j})\right], \\
\log p(\mathbf{E}|\widetilde{\mathbf{E}}') =& \frac{1}{\|\mathbf{A}\|_1 - N}\sum_{i\neq j}^{N}\log\left(\mathbf{E}_{i,j,:}^{\top}\widetilde{\mathbf{E}}_{i,j,:}\right), \\
\log p(\mathbf{F}|\widetilde{\mathbf{F}}') =& \frac{1}{N}\sum_{i=1}^{N}\log\left(\mathbf{F}_{i,:}^{\top}\widetilde{\mathbf{F}}'_{i,:}\right).
\end{aligned}
$$

9.5 Generative Adversarial Networks on Graphs

The generative adversarial nets (GANs) are a framework to estimate the complex data distribution via an adversarial process where the generative model is pitted against an adversary: a discriminative model that learns to tell whether a sample is from the original data or generated by the generative model (Goodfellow et al., 2014a). In detail, the generative model $G(\mathbf{z};\mathbf{\Theta})$ maps a noise variable $\mathbf{z}$ sampled from a prior noise distribution $p(\mathbf{z})$ to the data space with $\mathbf{\Theta}$ as its parameters. While, the discriminative model $D(\mathbf{x};\mathbf{\Phi})$ is modeled as a binary classifier with the parameters $\mathbf{\Phi}$, which tells whether a given data sample $\mathbf{x}$ is sampled from the data distribution $p_{data}(\mathbf{x})$ or generated by the generative model G. Specifically, $D(\mathbf{x};\mathbf{\Phi})$ maps $\mathbf{x}$ to a scalar indicating the probability that $\mathbf{x}$ comes from the given data rather than being generated by the generative model. During the training procedure, the two models are competing against each other. The generative model tries to learn to generate fake samples that are good enough to fool the discriminator, while the discriminator tries to improve itself to identify the samples generated by the generative model as fake samples. The competition drives both models to improve themselves until the generated samples are indistinguishable from the real ones. This competition can be modeled as a two-player minimax game as:

$$
\min_{\mathbf{\Theta}}\max_{\mathbf{\Phi}} \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}\left[\log D(\mathbf{x};\mathbf{\Phi})\right] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}\left[\log(1 - D(G(\mathbf{z};\mathbf{\Theta})))\right].
$$

The parameters of the generative model and the discriminative model are optimized alternatively. In this section, we will use node representation learning and graph generation tasks as examples to describe how the GAN frameworks can be applied to graph-structured data.

9.5.1 Generative Adversarial Networks for Node Representation Learning

In (Wang et al., 2018a), the GAN framework is adapted for node representation learning. Given a node v_i, the generative model aims to approximate the distribution of its neighbors. It can be denoted as $p(v_j|v_i)$ that is defined over the entire set of nodes $\mathcal{V}$. The set of its real neighbors $\mathcal{N}(v_i)$ can be regarded as the observed samples drawn from $p(v_j|v_i)$. The generator, which is denoted as $G(v_j|v_i;\mathbf{\Theta})$, tries to generate (more precisely, select) the node that is most likely connected with node v_i from $\mathcal{V}$. $G(v_j|v_i;\mathbf{\Theta})$ can be regarded as the probability of sampling v_j as a fake neighbor of node v_i. The discriminator, which we denoted as $D(v_j, v_i;\mathbf{\Phi})$, tries to tell whether a given pair of nodes (v_j, v_i) are connected or not in the graph. The output of the discriminator can be regarded as the probability of an edge existing between the two nodes v_j and v_i. The generator G and the discriminator D compete against each other: the generator G tries to fit the underlying probability distribution $p_{true}(v_j|v_i)$ perfectly such that the generated (selected) node v_j is relevant enough to the node v_i to fool the discriminator. While the discriminator tries to differentiate the nodes generated by the generator from the real neighbors of node v_i. Formally, the two models are playing the following minimax game:

$$\begin{aligned}\min_{\mathbf{\Theta}}\max_{\mathbf{\Phi}} V(G,D) = \sum_{v_i\in\mathcal{V}} \Big(&\mathbb{E}_{v_j\sim p_{\text{true}}(v_j|v_i)}\left[\log D\left(v_j, v_i;\mathbf{\Phi}\right)\right]\\ &+\mathbb{E}_{v_j\sim G(v_j|v_i;\mathbf{\Theta})}\left[\log\left(1-D\left(v_j,v_i;\mathbf{\Phi}\right)\right)\right]\Big).\end{aligned}$$

The parameters of the generator G and the discriminator D can be optimized by alternatively maximizing and minimizing the objective function $V(G,D)$. Next, we describe the details of the design of the generator and the discriminator.

The Generator

A straightforward way to model the generator is to use a softmax function over all nodes $\mathcal{V}$ as:

$$G(v_j|v_i;\mathbf{\Theta}) = \frac{\exp\left(\boldsymbol{\theta}_j^\top\boldsymbol{\theta}_i\right)}{\sum\limits_{v_k\in\mathcal{V}}\exp\left(\boldsymbol{\theta}_k^\top\boldsymbol{\theta}_i\right)}, \tag{9.15}$$

where $\boldsymbol{\theta}_i\in\mathbb{R}^d$ denotes the d-dimensional representation for the node v_i specific to the generator, and $\mathbf{\Theta}$ includes the representations for all nodes (They are also the parameters of the generator). Note that, in this formulation, the relevance between nodes are measured by the inner product of the representations of the two nodes. This idea is reasonable as we expect the low-dimensional representations to be closer if the two nodes are more relevant to each other. Once the parameters $\mathbf{\Theta}$ are learned, given a node v_i, the generator G can be used to sample nodes according to the distribution $G(v_j|v_i;\mathbf{\Theta})$. As we mentioned before, instead of generating fake nodes, the procedure of the generator should be more precisely described as selecting a node from the entire set $\mathcal{V}$.

While the softmax function in Eq. (9.15) provides an intuitive way to model the probability distribution, it suffers from severe computational issue. Specifically, the computational cost of the denominator of Eq. (9.15) is prohibitively expensive due to the summation over all nodes in $\mathcal{V}$. To solve this issue, hierarchical softmax (Morin and Bengio, 2005; Mikolov et al., 2013) introduced in Section 4.2.1 can be adopted.

The Discriminator

The discriminator is modeled as a binary classifier, which aims to tell whether a given pair of node (v_j, v_i) are connected with an edge in the graph or not. In detail, $D\left(v_j, v_i;\mathbf{\Phi}\right)$ models the probability of the existence of an edge between nodes v_j and v_i as:

$$D\left(v_j, v_i;\mathbf{\Phi}\right) = \sigma\left(\boldsymbol{\phi}_j^\top \boldsymbol{\phi}_i\right) = \frac{1}{1+\exp\left(-\boldsymbol{\phi}_j^\top \boldsymbol{\phi}_i\right)}, \tag{9.16}$$

where $\boldsymbol{\phi}_i \in \mathbb{R}^d$ is the low-dimensional representation of node v_i specific to the discriminator. We use the notation $\mathbf{\Phi}$ to denote the union of representations of all nodes, which are the parameters of the discriminator to be learned. After training, the node representations from both the generator and discriminator or their combination can be utilized for the downstream tasks.

9.5.2 Generative Adversarial Networks for Graph Generation

The framework of generative adversarial networks has been adapted for graph generation in (De Cao and Kipf, 2018). Specifically, the GAN framework is adopted to generate molecular graphs. As similar to Section 9.4.2, a molecular graph $\mathcal{G}$ with N nodes is represented by two objects: 1) A matrix $\mathbf{F} \in \{0,1\}^{N\times t_e}$, which indicates the type of all nodes. The i-th row of the matrix $\mathbf{F}$ corresponds to the i-th node and t_n is the number of node types (or different atoms); and 2) A tensor $\mathbf{E} \in \{0,1\}^{N\times N\times t_e}$ indicating the type of all edges where t_e is the number of edge types (or different bonds). The generator's goal is not only to generate molecular graphs similar to a given set of molecules but also to optimize some specific properties such as the solubility of these generated molecules. Hence, in addition to the generator and the discriminator in the GAN framework, there is also a judge. It measures how good a generated graph is in terms of the specific property (to assign a reward). The judge is a network pre-trained on some external molecules with ground truth. It is only used to provide guidance for generating desirable graphs. During the training procedure, the generator and discriminator are trained through competing against each other. However, the judge network is fixed and serves as a black box. Specifically, the generator and the discriminator are playing the following two-player minimax game:

$$\min_{\mathbf{\Theta}} \max_{\mathbf{\Phi}} \mathbb{E}_{\mathcal{G}\sim p_{data}(\mathcal{G})} \left[\log D(\mathcal{G}; \mathbf{\Phi})\right] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})} \left[\log(1 - D(G(\mathbf{z}; \mathbf{\Theta}))) - \lambda J(G(\mathbf{z}; \mathbf{\Theta}))\right],$$

where $p_{data}(\mathcal{G})$ denotes the true distribution of the given molecular graphs and $J()$ is the judge network. The judge network produces a scalar indicating some specific property of the input required to be maximized. Next, we describe the generator, the discriminator, and the judge network in the framework.

The Generator

The generator $G(\mathbf{z}; \mathbf{\Theta})$ is similar to the one we introduced in Section 9.4.2, where a fully connected probabilistic graph is generated given a latent representation $\mathbf{z}$ sampled from a noise distribution $p(\mathbf{z}) = \mathcal{N}(0, \mathbf{I})$. Specifically, the generator $G(\mathbf{z}; \mathbf{\Theta})$ maps a latent representation $\mathbf{z}$ to two continuous dense objects. They are used to describe the generated graph with k nodes – $\tilde{\mathbf{E}} \in \mathbb{R}^{k\times k\times t_e}$, which indicates the probability distributions of the type of edges; and $\tilde{\mathbf{F}} \in \mathbb{R}^{k\times t_n}$, which denotes the probability distribution of the types of nodes. To generate molecular graphs, discrete matrices of $\mathbf{E}$ and $\mathbf{F}$ are sampled from $\tilde{\mathbf{E}}$ and $\tilde{\mathbf{F}}$, respectively. During the training procedure, the continuous probabilistic graph $\tilde{\mathbf{E}}$ and $\tilde{\mathbf{F}}$ can be utilized such that the gradient can be successfully obtained through back-propagation.

The Discriminator and the Judge Network

Both the discriminator and the judge network receive a graph $\mathcal{G} = \{\mathbf{E}, \mathbf{F}\}$ as input, and output a scalar value. In (De Cao and Kipf, 2018), the graph neural network model is adopted to model these two components. In detail, the graph representation of the input graph $\mathcal{G}$ is obtained as:

$$\mathbf{h}_{\mathcal{G}} = pool(\text{GNN}(\mathbf{E}, \mathbf{F})),$$

where GNN() denotes several stacked graph filtering layers and pool() indicates the graph pooling operation. Specifically, in (De Cao and Kipf, 2018), the gated global pooling operation introduced in Section 5.4.1 is adopted as the pooling operation to generate the graph representation $\mathbf{h}_{\mathcal{G}}$. The graph representation is then fed into a few more fully connected layers to produce a scalar value. In particular, in the discriminator, the produced scalar value between 0 and 1 measures the probability that the generated graph is a “real” molecular graph from the given set of graphs. Meanwhile, the judge network outputs a scalar value that indicates the specific property of the graph. The discriminator needs to be trained alternatively with the generator. However, the judge network is pre-trained with the additional source of molecular graphs, and then it is treated as a fixed black box during the training of the GAN framework.

9.6 Conclusion

This chapter introduces more deep learning techniques on graphs. They include deep autoencoders, variational autoencoders (VAEs), recurrent neural networks (RNNs), and generative adversarial networks (GANs). Specifically, we introduce the graph autoencoders and recurrent neural networks, which are utilized to learn node representations. We then introduced two deep generative models: variational autoencoder and generative adversarial networks. We use the tasks of node representation learning and graph generation to illustrate how to adapt them to graphs.

9.7 Further Reading

Deep graph models beyond GNNs have greatly enriched deep learning methods on graphs and tremendously extended its application areas. In this chapter, we only introduce representative algorithms in one or two application areas. There are more algorithms and applications. In (Jin et al., 2018), variational autoencoder is utilized with graph neural networks for molecular graph generation. In (Ma et al., 2018b), additional constraints are introduced to variational graph autoencoders to generate semantically valid molecule graphs. In (You et al., 2018a), the GAN framework is combined with reinforcement learning techniques for molecule generation, where graph neural networks are adopted to model the policy network. Furthermore, recurrent neural networks are also utilized for graph generation (You et al., 2018b; Liao et al., 2019), where sequence of nodes and the connections between these nodes are generated.