Chapter 9

Deep Generative Models

The traditional graph generation approaches discussed in the previous chapter are useful in many settings. They can be used to efficiently generate synthetic graphs that have certain properties, and they can be used to give us insight into how certain graph structures might arise in the real world. However, a key limitation of those traditional approaches is that they rely on a fixed, hand-crafted generation process. In short, the traditional approaches can generate graphs, but they lack the ability to *learn* a generative model from data.

In this chapter, we will introduce various approaches that address exactly this challenge. These approaches will seek to learn a generative model of graphs based on a set of *training graphs*. These approaches avoid hand-coding particular properties—such as community structure or degree distributions—into a generative model. Instead, the goal of these approaches is to design models that can observe a set of graphs $\{\mathcal{G}_1, ..., \mathcal{G}_n\}$ and learn to generate graphs with similar characteristics as this training set.

We will introduce a series of basic deep generative models for graphs. These models will adapt three of the most popular approaches to building general deep generative models: variational autoencoders (VAEs), generative adversarial networks (GANs), and autoregressive models. We will focus on the simple and general variants of these models, emphasizing the high-level details and providing pointers to the literature where necessary. Moreover, while these generative techniques can in principle be combined with one another—for example, VAEs are often combined with autoregressive approaches—we will not discuss such combinations in detail here. Instead, we will begin with a discussion of basic VAE models for graphs, where we seek to generate an entire graph *all-at-once* in an autoencoder style. Following this, we will discuss how GAN-based objectives can be used in lieu of variational losses, but still in the setting where the graphs are generated all-at-once. These all-at-once generative models are analogous to the ER and SBM generative models from the last chapter, in that we sample all edges in the graph simultaneously. Finally, the chapter will close with a discussion of autoregressive approaches, which allow one to generate a graph *incrementally* instead of all-at-once (e.g., generating a

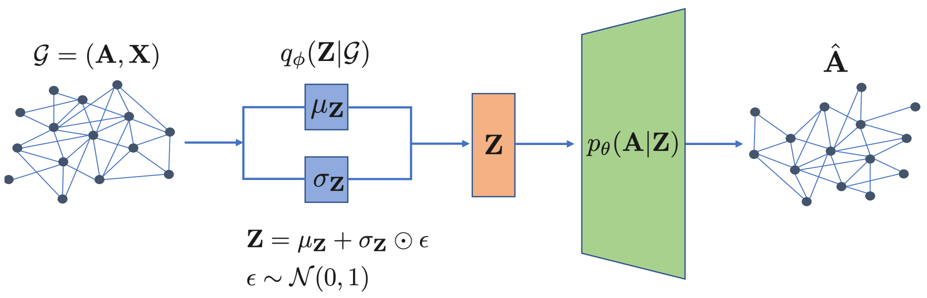
$\mathcal{G} = (\mathbf{A}, \mathbf{X})$
$q_\phi(\mathbf{Z}|\mathcal{G})$
$\mu_\mathbf{Z}$
$\sigma_\mathbf{Z}$
$\mathbf{Z}$
$p_\theta(\mathbf{A}|\mathbf{Z})$
$\hat{\mathbf{A}}$
$\mathbf{Z} = \mu_\mathbf{Z} + \sigma_\mathbf{Z} \odot \epsilon$
$\epsilon \sim \mathcal{N}(0, 1)$

Figure 9.1: Illustration of a standard VAE model applied to the graph setting. An *encoder* neural network maps the input graph $\mathcal{G} = (\mathbf{A}, \mathbf{X})$ to a *posterior distribution* $q_\phi(\mathbf{Z}|\mathcal{G})$ over latent variables $\mathbf{Z}$. Given a sample from this posterior, the *decoder* model $p_\theta(\mathbf{A}|\mathbf{Z})$ attempts to reconstruct the adjacency matrix.

graph node-by-node). These autoregressive approaches bear similarities to the preferential attachment model from the previous chapter in that the probability of adding an edge at each step during generation depends on what edges were previously added to the graph.

For simplicity, all the methods we discuss will only focus on generating graph structures (i.e., adjacency matrices) and not on generating node or edge features. This chapter assumes a basic familiarity with VAEs, GANs, and autoregressive generative models, such as LSTM-based language models. We refer the reader to Goodfellow et al. [2016] for background reading in these areas.

Of all the topics in this book, deep generative models of graphs are both the most technically involved and the most nascent in their development. Thus, our goal in this chapter is to introduce the key methodological frameworks that have inspired the early research in this area, while also highlighting a few influential models. As a consequence, we will often eschew low-level details in favor of a more high-level tour of this nascent sub-area.

9.1 Variational Autoencoder Approaches

Variational autoencoders (VAEs) are one of the most popular approaches to develop deep generative models [Kingma and Welling, 2013]. The theory and motivation of VAEs is deeply rooted in the statistical domain of variational inference, which we briefly touched upon in Chapter 7. However, for the purposes of this book, the key idea behind applying a VAE to graphs can be summarized as follows (Figure 9.1): our goal is to train a *probabilistic decoder* model $p_\theta(\mathbf{A}|\mathbf{Z})$, from which we can sample realistic graphs (i.e., adjacency matrices) $\hat{\mathbf{A}} \sim p_\theta(\mathbf{A}|\mathbf{Z})$ by conditioning on a latent variable $\mathbf{Z}$. In a probabilistic sense, we aim to learn a conditional distribution over adjacency matrices (with the distribution being conditioned on some latent variable).

In order to train a VAE, we combine the probabilistic decoder with a *prob-*

abilistic encoder model $q_\theta(\mathbf{Z}|\mathcal{G})$. This encoder model maps an input graph $\mathcal{G}$ to a *posterior distribution* over the latent variable $\mathbf{Z}$. The idea is that we jointly train the encoder and decoder so that the decoder is able to reconstruct training graphs given a latent variable $\mathbf{Z} \sim q_\theta(\mathbf{Z}|\mathcal{G})$ sampled from the encoder. Then, after training, we can discard the encoder and generate new graphs by sampling latent variables $\mathbf{Z} \sim p(\mathbf{Z})$ from some (unconditional) prior distribution and feeding these sampled latents to the decoder.

In more formal and mathematical detail, to build a VAE for graphs we must specify the following key components:

1. A *probabilistic encoder* model q_ϕ. In the case of graphs, the probabilistic encoder model takes a graph $\mathcal{G}$ as input. From this input, q_ϕ then defines a distribution $q_\phi(\mathbf{Z}|\mathcal{G})$ over *latent representations*. Generally, in VAEs the *reparameterization trick* with Gaussian random variables is used to design a probabilistic q_ϕ function. That is, we specify the latent conditional distribution as $\mathbf{Z} \sim \mathcal{N}(\mu_\phi(\mathcal{G}), \sigma(\phi(\mathcal{G}))$, where μ_ϕ and σ_ϕ are neural networks that generate the mean and variance parameters for a normal distribution, from which we sample latent embeddings $\mathbf{Z}$.

2. A *probabilistic decoder* model p_θ. The decoder takes a latent representation $\mathbf{Z}$ as input and uses this input to specify a conditional distribution over graphs. In this chapter, we will assume that p_θ defines a conditional distribution over the entries of the adjacency matrix, i.e., we can compute $p_\theta(\mathbf{A}[u, v] = 1|\mathbf{Z})$.

3. A *prior distribution* $p(\mathbf{Z})$ over the latent space. In this chapter we will adopt the standard Gaussian prior $\mathbf{Z} \sim \mathcal{N}(\mathbf{0}, \mathbf{1})$, which is commonly used for VAEs.

Given these components and a set of training graphs $\{\mathcal{G}_1, .., \mathcal{G}_n\}$, we can train a VAE model by minimizing the evidence likelihood lower bound (ELBO):

$$\mathcal{L} = \sum_{\mathcal{G}_i \in \{G_1, ..., \mathcal{G}_n\}} \mathbb{E}_{q_\theta(\mathbf{Z}|\mathcal{G}_i)}[p_\theta(\mathcal{G}_i|\mathbf{Z})] - \mathrm{KL}(q_\theta(\mathbf{Z}|\mathcal{G}_i)\|p(\mathbf{Z})). \tag{9.1}$$

The basic idea is that we seek to maximize the reconstruction ability of our decoder—i.e., the likelihood term $\mathbb{E}_{q_\theta(\mathbf{Z}|\mathcal{G}_i)}[p_\theta(\mathcal{G}_i|\mathbf{Z})]$—while minimizing the KL-divergence between our posterior latent distribution $q_\theta(\mathbf{Z}|\mathcal{G}_i)$ and the prior $p(\mathbf{Z})$.

The motivation behind the ELBO loss function is rooted in the theory of variational inference [Wainwright and Jordan, 2008]. However, the key intuition is that we want to generate a distribution over latent representations so that the following two (conflicting) goals are satisfied:

1. The sampled latent representations encode enough information to allow our decoder to reconstruct the input.

2. The latent distribution is as close as possible to the prior.

The first goal ensures that we learn to decode meaningful graphs from the encoded latent representations, when we have training graphs as input. The second goal acts as a regularizer and ensures that we can decode meaningful graphs even when we sample latent representations from the prior $p(\mathbf{Z})$. This second goal is critically important if we want to generate new graphs after training: we can generate new graphs by sampling from the prior and feeding these latent embeddings to the decoder, and this process will only work if this second goal is satisfied.

In the following sections, we will describe two different ways in which the VAE idea can be instantiated for graphs. The approaches differ in how they define the encoder, decoder, and the latent representations. However, they share the overall idea of adapting the VAE model to graphs.

9.1.1 Node-level Latents

The first approach we will examine builds closely upon the idea of encoding and decoding graphs based on node embeddings, which we introduced in Chapter 3. The key idea in this approach is that that the encoder generates latent representations for each node in the graph. The decoder then takes pairs of embeddings as input and uses these embeddings to predict the likelihood of an edge occurring between the two nodes. This idea was first proposed by Kipf and Welling [2016b] and termed the Variational Graph Autoencoder (VGAE).

Encoder model

The encoder model in this setup can be based on any of the GNN architectures we discussed in Chapter 5. In particular, given an adjacency matrix $\mathbf{A}$ and node features $\mathbf{X}$ as input, we use two separate GNNs to generate mean and variance parameters, respectively, conditioned on this input:

$$\mu_{\mathbf{Z}} = \text{GNN}_{\mu}(\mathbf{A}, \mathbf{X}) \qquad \log \sigma_{\mathbf{Z}} = \text{GNN}_{\sigma}(\mathbf{A}, \mathbf{X}). \tag{9.2}$$

Here, $\mu_{\mathbf{Z}}$ is a $|\mathcal{V}| \times d$-dimensional matrix, which specifies a mean embedding value *for each node in the input graph.* The $\log \sigma_{\mathbf{Z}} \in \mathbb{R}^{|V| \times d}$ matrix similarly specifies the log-variance for the latent embedding of each node.[1]

Given the encoded $\mu_{\mathbf{Z}}$ and $\log \sigma_{\mathbf{Z}}$ parameters, we can sample a set of latent node embeddings by computing

$$\mathbf{Z} = \epsilon \circ \exp\left(\log(\sigma_{\mathbf{Z}})\right) + \mu_{\mathbf{Z}}, \tag{9.3}$$

where $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{1})$ is a $|\mathcal{V}| \times d$ dimensional matrix with independently sampled unit normal entries.

[1]Parameterizing the log-variance is often more stable than directly parameterizing the variance.

The decoder model

Given a matrix of sampled node embeddings $\mathbf{Z} \in \mathbb{R}^{|V| \times d}$, the goal of the decoder model is to predict the likelihood of all the edges in the graph. Formally, the decoder must specify $p_\theta(\mathbf{A}|\mathbf{Z})$—the posterior probability of the adjacency matrix given the node embeddings. Again, here, many of the techniques we have already discussed in this book can be employed, such as the various edge decoders introduced in Chapter 3. In the original VGAE paper, Kipf and Welling [2016b] employ a simple dot-product decoder defined as follows:

$$p_\theta(\mathbf{A}[u,v] = 1|\mathbf{z}_u, \mathbf{z}_v) = \sigma(\mathbf{z}_u^\top \mathbf{z}_v), \tag{9.4}$$

where σ is used to denote the sigmoid function. Note, however, that a variety of edge decoders could feasibly be employed, as long as these decoders generate valid probability values.

To compute the reconstruction loss in Equation (9.1) using this approach, we simply assume independence between edges and define the posterior $p_\theta(\mathcal{G}|\mathbf{Z})$ over the full graph as follows:

$$p_\theta(\mathcal{G}|\mathbf{Z}) = \prod_{(u,v) \in \mathcal{V}^2} p_\theta(\mathbf{A}[u,v] = 1|\mathbf{z}_u, \mathbf{z}_v), \tag{9.5}$$

which corresponds to a binary cross-entropy loss over the edge probabilities. To generate discrete graphs after training, we can sample edges based on the posterior Bernoulli distributions in Equation (9.4).

Limitations

The basic VGAE model sketched in the previous sections defines a valid generative model for graphs. After training this model to reconstruct a set of training graphs, we could sample node embeddings $\mathbf{Z}$ from a standard normal distribution and use our decoder to generate a graph. However, the generative capacity of this basic approach is extremely limited, especially when a simple dot-product decoder is used. The main issue is that the decoder has no parameters, so the model is not able to generate non-trivial graph structures without a training graph as input. Indeed, in their initial work on the subject, Kipf and Welling [2016b] proposed the VGAE model as an approach to generate node embeddings, but they did not intend it as a generative model to sample new graphs.

Some papers have proposed to address the limitations of VGAE as a generative model by making the decoder more powerful. For example, Grover et al. [2019] propose to augment the decoder with an “iterative” GNN-based decoder. Nonetheless, the simple node-level VAE approach has not emerged as a successful and useful approach for graph generation. It has achieved strong results on reconstruction tasks and as an autoencoder framework, but as a generative model, this simple approach is severely limited.

9.1.2 Graph-level Latents

As an alternative to the node-level VGAE approach described in the previous section, one can also define variational autoencoders based on graph-level latent representations. In this approach, we again use the ELBO loss (Equation 9.1) to train a VAE model. However, we modify the encoder and decoder functions to work with graph-level latent representations $\mathbf{z}_{\mathcal{G}}$. The graph-level VAE described in this section was first proposed by Simonovsky and Komodakis [2018], under the name GraphVAE.

Encoder model

The encoder model in a graph-level VAE approach can be an arbitrary GNN model augmented with a pooling layer. In particular, we will let $\text{GNN} : \mathbb{Z}^{|\mathcal{V}|\times|\mathcal{V}|} \times \mathbb{R}^{|V|\times m} \rightarrow \mathbb{R}^{||V|\times d}$ denote any k-layer GNN, which outputs a matrix of node embeddings, and we will use $\text{POOL} : \mathbb{R}^{||V|\times d} \rightarrow \mathbb{R}^{d}$ to denote a pooling function that maps a matrix of node embeddings $\mathbf{Z} \in \mathbb{R}^{|V|\times d}$ to a graph-level embedding vector $\mathbf{z}_{\mathcal{G}} \in \mathbb{R}^{d}$ (as described in Chapter 5). Using this notation, we can define the encoder for a graph-level VAE by the following equations:

$$\mu_{\mathbf{z}_{\mathcal{G}}} = \text{POOL}_{\mu}\left(\text{GNN}_{\mu}(\mathbf{A}, \mathbf{X})\right) \qquad \log \sigma_{\mathbf{z}_{\mathcal{G}}} = \text{POOL}_{\sigma}\left(\text{GNN}_{\sigma}(\mathbf{A}, \mathbf{X})\right), \quad (9.6)$$

where again we use two separate GNNs to parameterize the mean and variance of a posterior normal distribution over latent variables. Note the critical difference between this graph-level encoder and the node-level encoder from the previous section: here, we are generating a mean $\mu_{\mathbf{z}_{\mathcal{G}}} \in \mathbb{R}^{d}$ and variance parameter $\log \sigma_{\mathbf{z}_{\mathcal{G}}} \in \mathbb{R}^{d}$ for a single graph-level embedding $\mathbf{z}_{\mathcal{G}} \sim \mathcal{N}(\mu_{\mathbf{z}_{\mathcal{G}}}, \sigma_{\mathbf{z}_{\mathcal{G}}})$, whereas in the previous section we defined posterior distributions for each individual node.

Decoder model

The goal of the decoder model in a graph-level VAE is to define $p_\theta(\mathcal{G}|\mathbf{z}_{\mathcal{G}})$, the posterior distribution of a particular graph structure given the graph-level latent embedding. The original GraphVAE model proposed to address this challenge by combining a basic multi-layer perceptron (MLP) with a Bernoulli distributional assumption [Simonovsky and Komodakis, 2018]. In this approach, we use an MLP to map the latent vector $\mathbf{z}_{\mathcal{G}}$ to a matrix $\tilde{\mathbf{A}} \in [0,1]^{|\mathcal{V}|\times|\mathcal{V}|}$ of edge probabilities:

$$\tilde{\mathbf{A}} = \sigma\left(\text{MLP}(\mathbf{z}_{\mathcal{G}})\right), \quad (9.7)$$

where the sigmoid function σ is used to guarantee entries in $[0,1]$. In principle, we can then define the posterior distribution in an analogous way as the node-level case:

$$p_\theta(\mathcal{G}|\mathbf{z}_{\mathcal{G}}) = \prod_{(u,v)\in\mathcal{V}} \tilde{\mathbf{A}}[u,v]\mathbf{A}[u,v] + (1-\tilde{\mathbf{A}}[u,v])(1-\mathbf{A}[u,v]), \quad (9.8)$$

where $\mathbf{A}$ denotes the true adjacency matrix of graph $\mathcal{G}$ and $\tilde{\mathbf{A}}$ is our predicted matrix of edge probabilities. In other words, we simply assume independent

Bernoulli distributions for each edge, and the overall log-likelihood objective is equivalent to set of independent binary cross-entropy loss function on each edge. However, there are two key challenges in implementing Equation (9.8) in practice:

1. First, if we are using an MLP as a decoder, then **we need to assume a fixed number of nodes.** Generally, this problem is addressed by assuming a *maximum* number of nodes and using a *masking* approach. In particular, we can assume a maximum number of nodes $n_{\max}$, which limits the output dimension of the decoder MLP to matrices of size $n_{\max} \times n_{\max}$. To decode a graph with $|\mathcal{V}| < n_{\max}$ nodes during training, we simply mask (i.e., ignore) all entries in $\tilde{\mathbf{A}}$ with row or column indices greater than $|\mathcal{V}|$. To generate graphs of varying sizes after the model is trained, we can specify a distribution $p(n)$ over graph sizes with support $\{2, ..., n_{\max}\}$ and sample from this distribution to determine the size of the generated graphs. A simple strategy to specify $p(n)$ is to use the empirical distribution of graph sizes in the training data.

2. The second key challenge in applying Equation (9.8) in practice is that **we do not know the correct ordering of the rows and columns in $\tilde{\mathbf{A}}$ when we are computing the reconstruction loss**. The matrix $\tilde{\mathbf{A}}$ is simply generated by an MLP, and when we want to use $\tilde{\mathbf{A}}$ to compute the likelihood of a training graph, we need to implicitly assume some ordering over the nodes (i.e., the rows and columns of $\tilde{\mathbf{A}}$). Formally, the loss in Equation (9.8) requires that we specify a node ordering $\pi \in \Pi$ to order the rows and columns in $\tilde{\mathbf{A}}$.

   This is important because if we simply ignore this issue, then the decoder can overfit to the arbitrary node orderings used during training. There are two popular strategies to address this issue. The first approach—proposed by Simonovsky and Komodakis [2018]—is to apply a graph matching heuristic to try to find the node ordering of $\tilde{\mathbf{A}}$ for each training graph that gives the highest likelihood, which modifies the loss to

$$p_\theta(\mathcal{G}|\mathbf{z}_\mathcal{G}) = \max_{\pi \in \Pi} \prod_{(u,v)\in\mathcal{V}} \tilde{\mathbf{A}}^\pi[u,v]\mathbf{A}[u,v] + (1 - \tilde{\mathbf{A}}^\pi[u,v])(1 - \mathbf{A}[u,v]), \quad (9.9)$$

   where we use $\tilde{\mathbf{A}}^\pi$ to denote the predicted adjacency matrix under a specific node ordering π. Unfortunately, however, computing the maximum in Equation (9.9)—even using heuristic approximations—is computationally expensive, and models based on graph matching are unable to scale to graphs with more than hundreds of nodes. More recently, authors have tended to use heuristic node orderings. For example, we can order nodes based on a depth-first or breadth-first search starting from the highest-degree node. In this approach, we simply specify a particular ordering function π and compute the loss with this ordering:

$$p_\theta(\mathcal{G}|\mathbf{z}_\mathcal{G}) \approx \prod_{(u,v)\in\mathcal{V}} \tilde{\mathbf{A}}^\pi[u,v]\mathbf{A}[u,v] + (1 - \tilde{\mathbf{A}}^\pi[u,v])(1 - \mathbf{A}[u,v]),$$

or we consider a small set of heuristic orderings $\pi_1, ..., \pi_n$ and average over these orderings:

$$p_\theta(\mathcal{G}|\mathbf{z}_\mathcal{G}) \approx \sum_{\pi_i \in \{\pi_1,...,\pi_n\}} \prod_{(u,v)\in\mathcal{V}} \tilde{\mathbf{A}}^{\pi_i}[u,v]\mathbf{A}[u,v]+(1-\tilde{\mathbf{A}}^{\pi_i}[u,v])(1-\mathbf{A}[u,v]).$$

These heuristic orderings do not solve the graph matching problem, but they seem to work well in practice. Liao et al. [2019a] provides a detailed discussion and comparison of these heuristic ordering approaches, as well as an interpretation of this strategy as a variational approximation.

Limitations

As with the node-level VAE approach, the basic graph-level framework has serious limitations. Most prominently, using graph-level latent representations introduces the issue of specifying node orderings, as discussed above. This issue—together with the use of MLP decoders—currently limits the application of the basic graph-level VAE to small graphs with hundreds of nodes or less. However, the graph-level VAE framework can be combined with more effective decoders—including some of the autoregressive methods we discuss in Section 9.3—which can lead to stronger models. We will mention one prominent example of such as approach in Section 9.5, when we highlight the specific task of generating molecule graph structures.

9.2 Adversarial Approaches

Variational autoencoders (VAEs) are a popular framework for deep generative models—not just for graphs, but for images, text, and a wide-variety of data domains. VAEs have a well-defined probabilistic motivation, and there are many works that leverage and analyze the structure of the latent spaces learned by VAE models. However, VAEs are also known to suffer from serious limitations—such as the tendency for VAEs to produce blurry outputs in the image domain. Many recent state-of-the-art generative models leverage alternative generative frameworks, with generative adversarial networks (GANs) being one of the most popular [Goodfellow et al., 2014].

The basic idea behind a general GAN-based generative models is as follows. First, we define a trainable generator network $g_\theta : \mathbb{R}^d \rightarrow \mathcal{X}$. This generator network is trained to generate realistic (but fake) data samples $\tilde{\mathbf{x}} \in \mathcal{X}$ by taking a random seed $\mathbf{z} \in \mathbb{R}^d$ as input (e.g., a sample from a normal distribution). At the same time, we define a discriminator network $d_\phi : \mathcal{X} \rightarrow [0, 1]$. The goal of the discriminator is to distinguish between real data samples $\mathbf{x} \in \mathcal{X}$ and samples generated by the generator $\tilde{\mathbf{x}} \in \mathcal{X}$. Here, we will assume that discriminator outputs the probability that a given input is fake.

To train a GAN, both the generator and discriminator are optimized jointly in an *adversarial game*:

$$\min_\theta \max_\phi \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}[\log(1 - d_\phi(\mathbf{x}))] + \mathbb{E}_{\mathbf{z}\sim p_{\text{seed}}(\mathbf{z})}[\log(d_\phi(g_\theta(\mathbf{z}))], \quad (9.10)$$

where $p_{\text{data}}(\mathbf{x})$ denotes the empirical distribution of real data samples (e.g., a uniform sample over a training set) and $p_{\text{seed}}(\mathbf{z})$ denotes the random seed distribution (e.g., a standard multivariate normal distribution). Equation (9.10) represents a minimax optimization problem. The generator is attempting to minimize the discriminatory power of the discriminator, while the discriminator is attempting to maximize its ability to detect fake samples. The optimization of the GAN minimax objective—as well as more recent variations—is challenging, but there is a wealth of literature emerging on this subject [Brock et al., 2018, Heusel et al., 2017, Mescheder et al., 2018].

A basic GAN approach to graph generation

In the context of graph generation, a GAN-based approach was first employed in concurrent work by Bojchevski et al. [2018] and De Cao and Kipf [2018]. The basic approach proposed by De Cao and Kipf [2018]—which we focus on here—is similar to the graph-level VAE discussed in the previous section. For instance, for the generator, we can employ a simple multi-layer perceptron (MLP) to generate a matrix of edge probabilities given a seed vector $\mathbf{z}$:

$$\tilde{\mathbf{A}} = \sigma\left(\text{MLP}(\mathbf{z})\right), \tag{9.11}$$

Given this matrix of edge probabilities, we can then generate a discrete adjacency matrix $\hat{\mathbf{A}} \in \mathbb{Z}^{|\mathcal{V}|\times|\mathcal{V}|}$ by sampling independent Bernoulli variables for each edge, with probabilities given by the entries of $\tilde{\mathbf{A}}$; i.e., $\hat{\mathbf{A}}[u,v] \sim \text{Bernoulli}(\tilde{\mathbf{A}}[u,v])$. For the discriminator, we can employ any GNN-based graph classification model. The generator model and the discriminator model can then be trained according to Equation (9.10) using standard tools for GAN optimization.

Benefits and limitations of the GAN approach

As with the VAE approaches, the GAN framework for graph generation can be extended in various ways. More powerful generator models can be employed—for instance, leveraging the autoregressive techniques discussed in the next section—and one can even incorporate node features into the generator and discriminator models [De Cao and Kipf, 2018].

One important benefit of the GAN-based framework is that it removes the complication of specifying a node ordering in the loss computation. As long as the discriminator model is permutation invariant—which is the case for almost every GNN—then the GAN approach does not require any node ordering to be specified. The ordering of the adjacency matrix generated by the generator is immaterial if the discriminator is permutation invariant. However, despite this important benefit, GAN-based approaches to graph generation have so far received less attention and success than their variational counterparts. This is likely due to the difficulties involved in the minimax optimization that GAN-based approaches require, and investigating the limits of GAN-based graph generation is currently an open problem.

9.3 Autoregressive Methods

The previous two sections detailed how the ideas of variational autoencoding (VAEs) and generative adversarial networks (GANs) can be applied to graph generation. However, both the basic GAN and VAE-based approaches that we discussed used simple multi-layer perceptrons (MLPs) to generate adjacency matrices. In this section, we will introduce more sophisticated *autoregressive* methods that can decode graph structures from latent representations. The methods that we introduce in this section can be combined with the GAN and VAE frameworks that we introduced previously, but they can also be trained as standalone generative models.

9.3.1 Modeling Edge Dependencies

The simple generative models discussed in the previous sections assumed that edges were generated *independently*. From a probabilistic perspective, we defined the likelihood of a graph given a latent representation $\mathbf{z}$ by decomposing the overall likelihood into a set of independent edge likelihoods as follows:

$$P(\mathcal{G}|\mathbf{z}) = \prod_{(u,v)\in\mathcal{V}^2} P(\mathbf{A}[u,v]|\mathbf{z}). \tag{9.12}$$

Assuming independence between edges is convenient, as it simplifies the likelihood model and allows for efficient computations. However, it is a strong and limiting assumption, since real-world graphs exhibit many complex dependencies between edges. For example, the tendency for real-world graphs to have high clustering coefficients is difficult to capture in an edge-independent model. To alleviate this issue—while still maintaining tractability—autoregressive model relax the assumption of edge independence.

Instead, in the autoregressive approach, we assume that edges are generated sequentially and that the likelihood of each edge can be conditioned on the edges that have been previously generated. To make this idea precise, we will use $\mathbf{L}$ to denote the lower-triangular portion of the adjacency matrix $\mathbf{A}$. Assuming we are working with simple graphs, $\mathbf{A}$ and $\mathbf{L}$ contain exactly the same information, but it will be convenient to work with $\mathbf{L}$ in the following equations. We will then use the notation $\mathbf{L}[v_1,:]$ to denote the row of $\mathbf{L}$ corresponding to node v_1, and we will assume that the rows of $\mathbf{L}$ are indexed by nodes $v_1,...,v_{|\mathcal{V}|}$. Note that due to the lower-triangular nature of $\mathbf{L}$, we will have that $\mathbf{L}[v_i,v_j]=0, \forall j>i$, meaning that we only need to be concerned with generating the first i entries for any row $\mathbf{L}[v_i,:]$; the rest can simply be padded with zeros. Given this notation, the autoregressive approach amounts to the following decomposition of the overall graph likelihood:

$$P(\mathcal{G}|\mathbf{z}) = \prod_{i=1}^{|\mathcal{V}|} P(\mathbf{L}[v_i,:]|\mathbf{L}[v_1,:],...,\mathbf{L}[v_{i-1},:],\mathbf{z}). \tag{9.13}$$

In other words, when we generate row $\mathbf{L}[v_i,:]$, we condition on all the previous generated rows $\mathbf{L}[v_j,:]$ with $j<i$.

9.3.2 Recurrent Models for Graph Generation

We will now discuss two concrete instantiations of the autoregressive generation idea. These two approaches build upon ideas first proposed in Li et al. [2018] and are generally indicative of the strategies that one could employ for this task. In the first model we will review—called GraphRNN [You et al., 2018]—we model autoregressive dependencies using a recurrent neural network (RNN). In the second approach—called graph recurrent attention network (GRAN) [Liao et al., 2019a]—we generate graphs by using a GNN to condition on the adjacency matrix that has been generated so far.

GraphRNN

The first model to employ this autoregressive generation approach was GraphRNN [You et al., 2018]. The basic idea in the GraphRNN approach is to use a hierarchical RNN to model the edge dependencies in Equation (9.13).

The first RNN in the hierarchical model—termed the graph-level RNN—is used to model the state of the graph that has been generated so far. Formally, the graph-level RNN maintains a hidden state $\mathbf{h}_i$, which is updated after generating each row of the adjacency matrix $\mathbf{L}[v_i, :]$:

$$\mathbf{h}_{i+1} = \text{RNN}_{\text{graph}}(\mathbf{h}_i, \mathbf{L}[v_i, L]), \tag{9.14}$$

where we use $\text{RNN}_{\text{graph}}$ to denote a generic RNN state update with $\mathbf{h}_i$ corresponding to the hidden state and $\mathbf{L}[v_i, L]$ to the observation.[2] In You et al. [2018]'s original formulation, a fixed initial hidden state $\mathbf{h}_0 = \mathbf{0}$ is used to initialize the graph-level RNN, but in principle this initial hidden state could also be learned by a graph encoder model or sampled from a latent space in a VAE-style approach.

The second RNN—termed the node-level RNN or RNN_{node}—generates the entries of $\mathbf{L}[v_i, :]$ in an autoregressive manner. RNN_{node} takes the graph-level hidden state $\mathbf{h}_i$ as an initial input and then sequentially generates the binary values of $\mathbf{L}[v_i, ;]$, assuming a conditional Bernoulli distribution for each entry. The overall GraphRNN approach is called hierarchical because the node-level RNN is initialized at each time-step with the current hidden state of the graph-level RNN.

Both the graph-level $\text{RNN}_{\text{graph}}$ and the node-level RNN_{node} can be optimized to maximize the likelihood the training graphs (Equation 9.13) using the *teaching forcing* strategy [Williams and Zipser, 1989], meaning that the ground truth values of $\mathbf{L}$ are always used to update the RNNs during training. To control the size of the generated graphs, the RNNs are also trained to output end-of-sequence tokens, which are used to specify the end of the generation process. Note that—as with the graph-level VAE approaches discussed in Section 9.1—computing the likelihood in Equation (9.13) requires that we assume a particular ordering over the generated nodes.

[2] You et al. [2018] use GRU-style RNNs but in principle LSTMs or other RNN architecture could be employed.

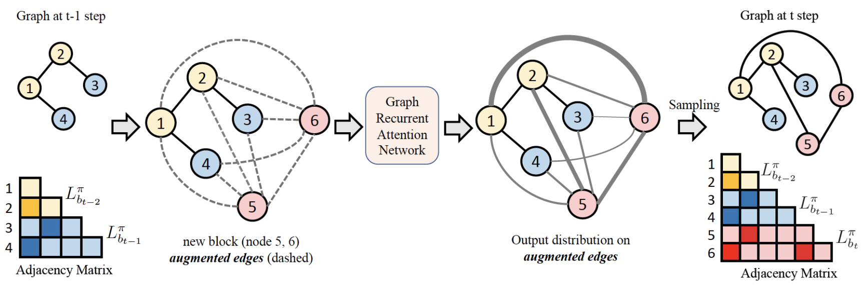

Figure 9.2: Illustration of the GRAN generation approach [Liao et al., 2019a].

After training to maximize the likelihood of the training graphs (Equation 9.13), the GraphRNN model can be used to generate graphs at test time by simply running the hierarchical RNN starting from the fixed, initial hidden state $\mathbf{h}_0$. Since the edge-level RNN involves a stochastic sampling process to generate the discrete edges, the GraphRNN model is able to generate diverse samples of graphs even when a fixed initial embedding is used. However—as mentioned above—the GraphRNN model could, in principle, be used as a decoder or generator within a VAE or GAN framework, respectively.

Graph Recurrent Attention Networks (GRAN)

The key benefit of the GraphRNN approach—discussed above—is that it models dependencies between edges. Using an autoregressive modeling assumption (Equation 9.13), GraphRNN can condition the generation of edges at generation step i based on the state of the graph that has already been generated during generation steps $1, ...i-1$. Conditioning in this way makes it much easier to generate complex motifs and regular graph structures, such as grids. For example, in Figure 9.3, we can see that GraphRNN is more capable of generating grid-like structures, compared to the basic graph-level VAE (Section 9.1). However, the GraphRNN approach still has serious limitations. As we can see in Figure 9.3, the GraphRNN model still generates unrealistic artifacts (e.g., long chains) when trained on samples of grids. Moreover, GraphRNN can be difficult to train and scale to large graphs due to the need to backpropagate through many steps of RNN recurrence.

To address some of the limitations of the GraphRNN approach, Liao et al. [2019a] proposed the GRAN model. GRAN—which stands for *graph recurrent attention networks*—maintains the autoregressive decomposition of the generation process. However, instead of using RNNs to model the autoregressive generation process, GRAN uses GNNs. The key idea in GRAN is that we can model the conditional distribution of each row of the adjacency matrix by running a GNN on the graph that has been generated so far (Figure 9.2):

$$P(\mathbf{L}[v_i,:]|\mathbf{L}[v_1,:],...,\mathbf{L}[v_{i-1},:],\mathbf{z}) \approx \text{GNN}(\mathbf{L}[v_1:v_{i-1},:],\tilde{\mathbf{X}}). \tag{9.15}$$

Here, we use $\mathbf{L}[v_1 : v_{i-1}, :]$ to denote the lower-triangular adjacency matrix of the graph that has been generated up to generation step i. The GNN in Equation (9.15) can be instantiated in many ways, but the crucial requirement is that it generates a vector of edge probabilities $\mathbf{L}[v_i, :]$, from which we can sample discrete edge realizations during generation. For example, Liao et al. [2019a] use a variation of the graph attention network (GAT) model (see Chapter 5) to define this GNN. Finally, since there are no node attributes associated with the generated nodes, the input feature matrix $\tilde{\mathbf{X}}$ to the GNN can simply contain randomly sampled vectors (which are useful to distinguish between nodes).

The GRAN model can be trained in an analogous manner as GraphRNN by maximizing the likelihood of training graphs (Equation 9.13) using teacher forcing. Like the GraphRNN model, we must also specify an ordering over nodes to compute the likelihood on training graphs, and Liao et al. [2019a] provides a detailed discussion on this challenge. Lastly, like the GraphRNN model, we can use GRAN as a generative model after training simply by running the stochastic generation process (e.g., from a fixed initial state), but this model could also be integrated into VAE or GAN-based frameworks.

The key benefit of the GRAN model—compared to GraphRNN—is that it does not need to maintain a long and complex history in a graph-level RNN. Instead, the GRAN model explicitly conditions on the already generated graph using a GNN at each generation step. Liao et al. [2019a] also provide a detailed discussion on how the GRAN model can be optimized to facilitate the generation of large graphs with hundreds of thousands of nodes. For example, one key performance improvement is the idea that multiple nodes can be added simultaneously in a single *block*, rather than adding nodes one at a time. This idea is illustrated in Figure 9.2.

9.4 Evaluating Graph Generation

The previous three sections introduced a series of increasingly sophisticated graph generation approaches, based on VAEs, GANs, and autoregressive models. As we introduced these approaches, we hinted at the superiority of some approaches over others. We also provided some examples of generated graphs in Figure 9.3, which hint at the varying capabilities of the different approaches. However, how do we actually quantitatively compare these different models? How can we say that one graph generation approach is better than another? Evaluating generative models is a challenging task, as there is no natural notion of accuracy or error. For example, we could compare reconstruction losses or model likelihoods on held out graphs, but this is complicated by the lack of a uniform likelihood definition across different generation approaches.

In the case of general graph generation, the current practice is to analyze different statistics of the generated graphs, and to compare the distribution of statistics for the generated graphs to a test set [Liao et al., 2019a]. Formally, assume we have set of graph statistics $\mathcal{S} = (s_1, s_2, ..., s_n)$, where each of these statistics $s_{i,\mathcal{G}} : \mathbb{R} \to [0, 1]$ is assumed to define a univariate distribution over $\mathbb{R}$

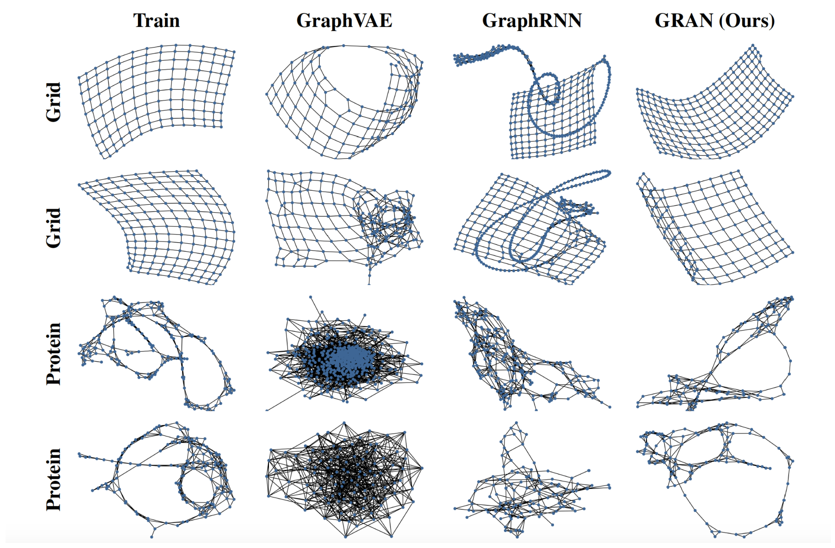

Figure 9.3: Examples of graphs generated by a basic graph-level VAE (Section 9.1), as well as the GraphRNN and GRAN models. Each row corresponds to a different dataset. The first column shows an example of a real graph from the dataset, while the other columns are randomly selected samples of graphs generated by the corresponding model [Liao et al., 2019a].

for a given graph $\mathcal{G}$. For example, for a given graph $\mathcal{G}$, we can compute the degree distribution, the distribution of clustering coefficients, and the distribution of different motifs or graphlets. Given a particular statistic s_i—computed on both a test graph $s_{i,\mathcal{G}_{\text{test}}}$ and a generated graph $s_{i,\mathcal{G}_{\text{gen}}}$—we can compute the distance between the statistic's distribution on the test graph and generated graph using a distributional measure, such as the total variation distance:

$$d(s_{i,\mathcal{G}_{\text{test}}}, s_{i,\mathcal{G}_{\text{gen}}}) = \sup_{x \in \mathbb{R}} |s_{i,\mathcal{G}_{\text{test}}}(x) - s_{i,\mathcal{G}_{\text{gen}}}(x)|. \tag{9.16}$$

To get measure of performance, we can compute the average pairwise distributional distance between a set of generated graphs and graphs in a test set.

Existing works have used this strategy with graph statistics such as degree distributions, graphlet counts, and spectral features, with distributional distances computed using variants of the total variation score and the first Wasserstein distance [Liao et al., 2019b, You et al., 2018].

9.5 Molecule Generation

All the graph generation approaches we introduced so far are useful for general graph generation. The previous sections did not assume a particular data domain, and our goal was simply to generate realistic graph structures (i.e., adjacency matrices) based on a given training set of graphs. It is worth noting, however, that many works within the general area of graph generation are focused specifically on the task of *molecule generation*.

The goal of molecule generation is to generate molecular graph structures that are both valid (e.g., chemically stable) and ideally have some desirable properties (e.g., medicinal properties or solubility). Unlike the general graph generation problem, research on molecule generation can benefit substantially from domain-specific knowledge for both model design and evaluation strategies. For example, Jin et al. [2018] propose an advanced variant of the graph-level VAE approach (Section 9.1) that leverages knowledge about known molecular motifs. Given the strong dependence on domain-specific knowledge and the unique challenges of molecule generation compared to general graphs, we will not review these approaches in detail here. Nonetheless, it is important to highlight this domain as one of the fastest growing subareas of graph generation.