Chapter 12
Graph Neural Networks: Graph Transformation

Xiaojie Guo, Shiyu Wang, Liang Zhao

Abstract Many problems regarding structured predictions are encountered in the process of "transforming" a graph in the source domain into another graph in target domain, which requires to learn a transformation mapping from the source to target domains. For example, it is important to study how structural connectivity influences functional connectivity in brain networks and traffic networks. It is also common to study how a protein (e.g., a network of atoms) folds, from its primary structure to tertiary structure. In this chapter, we focus on the transformation problem that involves graphs in the domain of deep graph neural networks. First, the problem of graph transformation in the domain of graph neural networks are formalized in Section 27.1. Considering the entities that are being transformed during the transformation process, the graph transformation problem is further divided into four categories, namely node-level transformation, edge-level transformation, node-edge co-transformation, as well as other graph-involved transformations (e.g., sequence-to-graph transformation and context-to-graph transformation), which are discussed in Section 24.2 to Section 20.5, respectively. In each subsection, the definition of each category and their unique challenges are provided. Then, several representative graph transformation models that address the challenges from different aspects for each category are introduced.

Xiaojie Guo
Department of Information Science and Technology, , George Mason University, e-mail: xguo7@gmu.edu

Shiyu Wang
Department of Computer Science, Emory University, e-mail: shiyu.wang@emory.edu

Liang Zhao
Department of Computer Science, Emory University, e-mail: liang.zhao@emory.edu

12.1 Problem Formulation of Graph Transformation

Many problems regarding structured predictions are encountered in the process of "translating" an input data (e.g., images, texts) into a corresponding output data, which is to learn a translation mapping from the input domain to the target domain. For example, many problems in computer vision can be seen as a "translation" from an input image into a corresponding output image. Similar applications can also be found in language translation, where sentences (sequences of words) in one language are translated into corresponding sentences in another language. Such a generic translation problem, which is important yet has been extremely difficult in nature, has attracted rapidly increasing attention in recent years. The conventional data transformation problem typically considers the data under special topology. For example, an image is a type of grid where each pixel is a node and each node has connections to its spatial neighbors. Texts are typically considered as sequences where each node is a word and an edge exists between two contextual words. Both grids and sequences are special types of graphs. In many practical applications, it is required to work on data with more flexible structures than grids and sequences, and hence more powerful translation techniques are required in order to handle more generic graph-structured data. Thus, there emerges a new problem named deep graph transformation, the goal of which is to learn the mapping from the graph in the input domain to the graph in the target domain. The mathematical problem formulation of the graph is provided in detail as below.

A graph is defined as $\mathscr{G}(\mathscr{V},\mathscr{E},F,E)$, where $\mathscr{V}$ is the set of N nodes, and $\mathscr{E} \subseteq \mathscr{V} \times \mathscr{V}$ is the set of M edges. $e_{i,j} \in \mathscr{E}$ is an edge connecting nodes v_i, $v_j \in \mathscr{V}$. A graph can be described in matrix or tensor using its (weighted) adjacency matrix A. If the graph has node attributes and edge attributes, there are node attribute matrix $F \in \mathbb{R}^{N\times L}$ where D is the number of node attributes, and edge attribute tensor $E \in \mathbb{R}^{N\times N\times K}$ where K is the number of edge attributes. L is the dimension of node attributes, and K is the dimension of edge attributes. Based on the definition of graph, we define the input graphs from the source domain as $\mathscr{G}_S$ and the output graphs from the target domain as $\mathscr{G}_S \rightarrow \mathscr{G}_T$ (Guo et al, 2019c).

Considering the entities that are being transformed during the transformation process, the graph transformation problem is further divided into three categories, namely (1) node-level transformation, where only nodes and nodes attributes can change during translation process; (2) edge-level transformation, where only topology or edge attributes can change during translation process; (3) node-edge co-transformation where both nodes and edges can change during translation process. There are also some other transformations involving graphs, including sequence-to-graph transformation, graph-to-sequence transformation and context-to-graph transformation. Although they can be absorbed into the above three types if regarding sequences as a special case of graphs, we want to separate them out because they may usually attract different research communities.

12.2 Node-level Transformation

12.2.1 Definition of Node-level Transformation

Node-level transformation aims to generate or predict the node attributes or node category of the target graph conditioning on the input graph. It can also be regarded as a node prediction problem with stochasticity. It requires the node set $\mathscr{V}$ or node attributes F to change while the graph edge set and edge attributes are fixed during the transformation namely $\mathscr{G}_S(\mathscr{V}_S, \mathscr{E}, F_S, E) \rightarrow \mathscr{G}_T(\mathscr{V}_T, \mathscr{E}, F_T, E)$. Node transformation has a wide range of real-world applications, such as predicting future states of a system in the physical domain based on the fixed relations (e.g. gravitational forces) (Battaglia et al, 2016) among nodes and the traffic speed forecasting on the road networks (Yu et al, 2018a; Li et al, 2018e). Existing works adopt different frameworks to model the transformation process.

Generally speaking, the straightforward way in dealing with the node translation problem is to regard it as the node prediction problem and utilize the conventional GNNs as encoder to learn the node embedding. Then, based on the node embedding, we can predict the node attributes of the target graphs. While solving the node transformation problem in specific domains, there come various unique requirements, such as considering the spatial and temporal patterns in the traffic flow prediction task. Thus, in this section, we focus on introducing three typical node transformation models in dealing with problems in different areas.

12.2.2 Interaction Networks

Battaglia et al (2016) proposed the interaction network in the task of reasoning about objects, relations, and physics, which is central to human intelligence, and a key goal of artificial intelligence. Many physical problems, such as predicting what will happen next in physical environments or inferring underlying properties of complex scenes, are challenging because their elements are composed and can influence each other as a whole system. It is impossible to solve such problems by considering each object and relation separately. Thus, the node transformation problem can help deal with this task via modeling the interactions and dynamics of elements in a complex system. To deal with the node transformation problem that is formalized in this scenario, an interaction network (IN) is proposed, which combines two main powerful approaches: structured models, simulation, and deep learning. Structured models are operated as the main component based on the GNNs to exploit the knowledge of relations among objects. The simulation part is an effective method for approximating dynamical systems, predicting how the elements in a complex system are influenced by interactions with one another, and by the dynamics of the system.

The overall complex system can be represented as an attributed, directed multigraph $\mathscr{G}$, where each node represents an object and the edge represents the rela-

tionship between two objects, e.g., a fixed object attached by a spring to a freely moving mass. To predict the dynamics of a single node (i.e., object), there is an object-centric function, $\mathbf{h}_i^{t+1} = f_O(\mathbf{h}_i^t)$ with the object's state $\mathbf{h}_t$ at time t of the object v_i as the inputs and a future state $\mathbf{h}_i^{t+1}$ at next time step as outputs. Assuming two objects have one directed relationship, the first object v_i influences the second object v_j via their interaction. The effect or influence of this interaction, $\mathbf{e}_{i,j}^{t+1}$ is predicted by a relation-centric function, f_R, with the object states as well as attributes of their relationship as inputs. The object updating process is then written as:

$$\mathbf{e}_{i,j}^{t+1} = f_R(\mathbf{h}_i^t, \mathbf{h}_j^t, \mathbf{r}_i); \quad \mathbf{h}_i^{t+1} = f_O(\mathbf{h}_i^t, \mathbf{e}_{i,j}^{t+1}), \tag{12.1}$$

where $\mathbf{r}_i$ refers to the interaction effects that node v_i receives.

It worth noting that the above operations are for an attributed, directed multi-graph because the edges/ relations can have attributes, and there can be multiple distinct relations between two objects (e.g., rigid and magnetic interactions). In summary, at each step, the interaction effects generated from each relationship is calculated and then an aggregation function is utilized to summarize all the interactions effects on the relevant objects and update the states of each object.

An IN applies the same f_R and f_O to every target nodes, respectively, which makes their relational and object reasoning able to handle variable numbers of arbitrarily ordered objects and relations (i.e., graphs with variables sizes). But one additional constraint must be satisfied to maintain this: the aggregation function must be commutative and associative over the objects and relations, for example summation as aggregation function satisfies this, but division would not.

The IN can be included in the framework of Message Passing Neural Network (MPNN), with the message passing process, aggregation process, and node updating process. However, different from MPNN models which focus on binary relations (i.e., there is one edge per pair of nodes), IN can also handle hyper-graph, where the edges can correspond to n-th order relations by combining n nodes ($n \geq 2$). The IN has shown a strong ability to learn accurate physical simulations and generalize their training to novel systems with different numbers and configurations of objects and relations. They could also learn to infer abstract properties of physical systems, such as potential energy. The IN implementation is the first learnable physics engine that can scale up to real-world problems, and is a promising template for new AI approaches to reasoning about other physical and mechanical systems, scene understanding, social perception, hierarchical planning, and analogical reasoning.

12.2.3 Spatio-Temporal Convolution Recurrent Neural Networks

Spatio-temporal forecasting is a crucial task for a learning system that operates in a dynamic environment. It has a wide range of applications from autonomous vehicles operations, to energy and smart grid optimization, to logistics and supply chain management. The traffic forecasting on road networks, the core component

of the intelligent transportation systems, can be formalized as a node transformation problem, the goal of which is to predict the future traffic speeds (i.e., node attributes) of a sensor network (i.e., graph) given historic traffic speeds (i.e., history node attributes). This type of node transformation is unique and challenging due to the complex spatio-temporal dependencies in a series of graphs and inherent difficulty in the long term forecasting. To deal with this, each pair-wise spatial correlation between traffic sensors is represented using a directed graph whose nodes are sensors and edge weights denote proximity between the sensor pairs measured by the road network distance. Then the dynamics of the traffic flow is modeled as a diffusion process and the diffusion convolution operation is utilized to capture the spatial dependency. The overall Diffusion Convolutional Recurrent Neural Network (DCRNN) integrates diffusion convolution, the sequence to sequence architecture and the scheduled sampling technique.

Denote the node information (e.g., traffic flow) observed on a graph $\mathscr{G}$ as a graph signal F and let F^t represent the graph signal observed at time t, the temporal node transformation problem aims to learn a mapping from T' historical graph signals to future T graph signals as: $[F^{t-T'+1},...,F^t;\mathscr{G}] \rightarrow [F^{t+1},...,F^{t+T};\mathscr{G}]$. The spatial dependency is modeled by relating node information to a diffusion process, which is characterized by a random walk on $\mathscr{G}$ with restart probability $\alpha \in [0,1]$ and a state transition matrix $D_O^{-1}W$. Here D_O is the out-degree diagonal matrix, and $\mathbf{1}$. After many time steps, such Markov process converges to a stationary distribution $P \in \mathbb{R}^{N\times N}$ whose i-th row represents the likelihood of diffusion from node v_i. Thus, a diffusion convolutional layer can be defined as

$$H_{:,q} = f(\sum_{p=1}^{P} F_{:,p} \star_{\mathscr{G}} f_{\Theta_{p,q,:,:}}), \quad q \in \{1,...,Q\} \tag{12.2}$$

where the diffusion convolution operation is defined as

$$F_{:,p} \star_{\mathscr{G}} f_\theta = \sum_{k=0}^{K-1} (\phi_{k,1}(D_O^{-1}W)^k + \phi_{k,2}(D_I^{-1}W^T)^k)F_{:,p}, \qquad p \in \{1,...,P\} \tag{12.3}$$

Here the D_O and D_I refer to the out-degree and in-degree diagonal matrix respectively. P and Q refer to the feature dimension of the input and output node features at each diffusion convolution layer. The diffusion convolution is defined on both directed and undirected graphs. When applied to undirected graphs, the existing graph convolution neural networks (GCN) can be considered as a special case of diffusion convolution network.

To deal with the temporal dependency during the node transformation process, the recurrent neural networks (RNN) or Gated Recurrent Unit (GRU) can be leveraged. For example, by replacing the matrix multiplications in GRU with the diffusion convolution, the Diffusion Convolutional Gated Recurrent Unit (DCGRU) is defined as

$$\begin{aligned}
\mathbf{r}^t &= \sigma(\Theta_r \star_{\mathscr{G}} [F^t, H^{t-1}] + \mathbf{b}_r^t) \\
\mathbf{u}^t &= \sigma(\Theta_u \star_{\mathscr{G}} [F^t, H^{t-1}] + \mathbf{b}_u^t) \\
C^t &= \mathbf{Tanh}(\sigma(\Theta_c \star_{\mathscr{G}} [F^t, (\mathbf{r}^t \odot H^{t-1})] + \mathbf{b}_c^t)) \\
H^{t-1} &= \mathbf{u}^t \odot H^{t-1} + (1 - \mathbf{u}^t) \odot C^t,
\end{aligned} \tag{12.4}$$

where X^t and H^t denote the input and output of all the nodes at time t, $\mathbf{r}^t$ and $\mathbf{u}^t$ are reset gate and update gate at time t, respectively. $\star_{\mathscr{G}}$ denotes the diffusion convolution defined in Equation equation 12.3. $\Theta_r, \Theta_u, \Theta_c$ are parameters for the corresponding filters in the diffusion network.

Another typical spatio-temporal graph convolution network for spatial-temporal node transformation is proposed by (Yu et al, 2018a). This model comprises several spatio-temporal convolutional blocks, which are a combination of graph convolutional layers and convolutional sequence learning layers, to model spatial and temporal dependencies. Specifically, the framework consists of two spatio-temporal convolutional blocks (ST-Conv blocks) and a fully-connected output layer in the end. Each ST-Conv block contains two temporal gated convolution layers and one spatial graph convolution layer in the middle. The residual connection and bottleneck strategy are applied inside each block. The input sequence of node information is uniformly processed by ST-Conv blocks to explore spatial and temporal dependencies coherently. Comprehensive features are integrated by an output layer to generate the final prediction. In contrast to the above mentioned DCGRU, this model is built completely from convolutional structures to capture both spatial and temporal patterns without any recurrent neural network; each block is specially designed to uniformly process structured data with residual connection and bottleneck strategy inside.

12.3 Edge-level Transformation

12.3.1 Definition of Edge-level Transformation

Edge-level transformation aims to generate the graph topology and edge attributes of the target graph conditioning on the input graph. It requires the edge set $\mathscr{E}$ and edge attributes E to change while the graph node set and node attributes are fixed during the transformation: $\mathscr{T}$: $\mathscr{G}_S(\mathscr{V}, \mathscr{E}_S, F, E_S) \rightarrow \mathscr{G}_T(\mathscr{V}, \mathscr{E}_T, F, E_T)$. Edge transformation has a wide range of real-world applications, such as modeling chemical reactions (You et al, 2018a), protein folding (Anand and Huang, 2018) and malware cybernetwork synthesis (Guo et al, 2018b). For example, in social networks where people are the nodes and their contacts are the edges, the contact graph among them vary dramatically across different situations. For example, when the people are organizing a riot, it is expected that the contact graph to become denser and several special "hubs" (e.g., key players) may appear. Hence, accurately predicting the contact net-

work in a target situation is highly beneficial to situational awareness and resource allocation.

Numerous efforts have been contributed to edge-level graph transformation. Here we introduce three typical methods in modelling the edge-level graph transformation problem, including graph transformation generative adversarial networks (GT-GAN), multi-scale graph transformation networks (Misc-GAN), and graph transformation policy networks (CTPN).

12.3.2 Graph Transformation Generative Adversarial Networks

Generative Adversarial Network (GANs) is a alternative method for generation problem. It is designed based on a game theory scenario called the min-max game, where a discriminator and a generator compete against each other. The generator generates data from stochastic noise, and the discriminator tries to tell whether it is real (coming from a training set) or fabricated (from the generator). The absolute difference between carefully calculated rewards from both networks is minimized so that both networks learn simultaneously as they try to outperform each other. GANs can be extended to a conditional model if both the generator and discriminator are conditioned on some extra auxiliary information, such as class labels or data from other modalities. Conditional GANs is realized by feeding the conditional information into the both the discriminator and generator as additional input layer. In this scenario, when the conditional information is a graph, the conditional GANs can be utilized to handle graph transformation problem to learning the mapping from the conditional graph (i.e., input graph) to the target graph (i.e., output graph). Here, we introduce two typical edge-level graph transformation technqiues that are based on the Conditional GANs.

A novel Graph-Translation-Generative Adversarial Networks (GT-GAN) proposed by (Guo et al, 2018b) can successfully implement and learn the mapping from the input to target graphs. GT-GAN consists of a graph translator $\mathscr{T}$ and a conditional graph discriminator $\mathscr{D}$. The graph translator $\mathscr{T}$ is trained to produce target graphs that cannot be distinguished from "real" ones by our conditional graph discriminator $\mathscr{D}$. Specifically, the generated target graph $\mathscr{G}_{T'} = \mathscr{T}(\mathscr{G}_S, U)$ cannot be distinguished from the real one, $\mathscr{G}_T$, based on the current input graph $\mathscr{G}_S$. U refers to the random noises. $\mathscr{T}$ and $\mathscr{D}$ undergo an adversarial training process based on input and target graphs by solving the following loss function:

$$\begin{aligned}\mathscr{L}(\mathscr{T},\mathscr{D}) = &\mathbb{E}_{\mathscr{G}_S,\mathscr{G}_T\sim\mathscr{S}}[\log \mathscr{D}(\mathscr{G}_T|\mathscr{G}_S)] \\ &+ \mathbb{E}_{\mathscr{G}_S\sim\mathscr{S}}[\log(1-\mathscr{D}(\mathscr{T}(\mathscr{G}_S,U)|\mathscr{G}_S))],\end{aligned} \tag{12.5}$$

where $\mathscr{S}$ refers to the dataset. $\mathscr{T}$ tries to minimize this objective while an adversarial $\mathscr{D}$ tries to maximize it, i.e. $\mathscr{T}^* = \arg\min_{\mathscr{T}} \max_{\mathscr{D}} \mathscr{L}(\mathscr{T},\mathscr{D})$. The graph translator includes two parts: graph encoder and graph decoder. A graph convolution neural net (Kawahara et al, 2017) is extended to serve as the graph encoder in order to embed

the input graph into node-level representations, while a new graph deconvolution net is designed as the decoder to generate the target graph. Specifically, the encoder consists of edge-to-edge and edge-to-node convolution layers, which first extract latent edge-level representations and then node-level representations $\{H_i\}_{i=1}^N$, where $H_i \in \mathbb{R}^L$ refers to the latent representation of node v_i. The decoder consists of node-to-edge and edge-to-edge deconvolution layers to first get each edge representation $\hat{E}_{i,j}$ based on H_i and H_j, and then finally get edge attribute tensor E based on $\hat{E}$. Based on the graph deconvolution above, it is possible to utilize skips to link the extracted edge latent representations of each layer in the graph encoder with those in the graph decoder.

Specifically, in the graph translator, the output of the l-th "edge deconvolution" layer in the decoder is concatenated with the output of the l-th "edge convolution" layer in the encoder to form joint two channels of feature maps, which are then input into the $(l+1)$-th deconvolution layer. It is worth noting that one key factor for effective translation is the design of a symmetrical encoder-decoder pair, where the graph deconvolution is a mirrored reversed way from graph convolution. This allows skip-connections to directly translate different level's edge information at each layer.

The graph discriminator is utilized to distinguish between the "translated" target graph and the "real" ones based on the input graphs, as this helps to train the generator in an adversarial way. Technically, this requires the discriminator to accept two graphs simultaneously as inputs (a real target graph and an input graph or a generated graph and an input graph) and classify the two graphs as either related or not. Thus, a conditional graph discriminator (CGD) that leverages the same graph convolution layers in the encoder is utilized for the graph classification. Specifically, the input and target graphs are both ingested by the CGD and stacked into a tensor, which can be considered a 2-channel input. After obtaining the node representations, the graph-level embedding is computed by summing these node embeddings. Finally, a softmax layer is implemented to distinguish the input graph-pair from the real graph or generated graph.

To further handle the situation when the pairing information of the input and the output is not available, Gao et al (2018b) proposes a Unpaired Graph Translation Generative Adversarial Nets (UGT-GAN) based on Cycle-GAN (Zhu et al, 2017) and incorporate the same encoder and deconder in GT-GAN to handle the unpaired graph transformation problems. The cycle consistency loss is utilized and generalized into graph cycle consistency loss for unpaired graph translation. Specifically, graph cycle consistency adds an opposite direction translator from target to source domain $\mathscr{T}_r : \mathscr{G}_T \to \mathscr{G}_S$ by training the mappings for both directions simultaneously, and adding a cycle consistency loss that encourages $\mathscr{T}_r(\mathscr{T}(\mathscr{G}_S)) \approx \mathscr{G}_S$ and $\mathscr{T}(\mathscr{T}_r(\mathscr{G}_T)) \approx \mathscr{G}_T$. Combining this loss with adversarial losses on domains $\mathscr{G}_T$ and $\mathscr{G}_S$ yields the full objective for unpaired graph translation.

12.3.3 Multi-scale Graph Transformation Networks

Many real-world networks typically exhibit hierarchical distribution over graph communities. For instance, given an author collaborative network, research groups of well-established and closely collaborated researchers could be identified by the existing graph clustering methods in the lower-level granularity. While, from a coarser level, we may find that these research groups constitute large-scale communities, which correspond to various research topics or subjects. Thus, it is necessary to capture the hierarchical community structures over the graphs for edge-level graph transformation problem. Here, we introduce a graph generation model for learning the distribution of the graphs, which, however, is formalized as a edge-level graph transformation problem.

Based on GANs, a multi-scale graph generative model, Misc-GAN, can be utilized to model the underlying distribution of graph structures at different levels of granularity. Inspired by the success of deep generative models in image translation, a cycle-consistent adversarial network (CycleGAN) (Zhu et al, 2017) is adopted to learn the graph structure distribution and and then generate a synthetic coarse graph at each granularity level. Thus, the graph generation task can be realized by "transferring" the hierarchical distribution from the graphs in the source domain to a unique graph in the target domain.

In this framework, the input graph is characterized as several coarse-grained graphs by aggregating the strongly coupled nodes with a small algebraic distance to form coarser nodes. Overall, the framework can be separated into three stages. First, the coarse-grained graphs at K levels of granularity are constructed from the input graph adjacent matrix A_S. The adjacent matrix of the coarse-grained graph $A_S^{(k)} \in \mathbb{R}^{N^{(k)} \times N^{(k)}}$ at the k-th layer is defined as follows:

$$A_S^{(k)} = {P^{(k-1)}}^\top ... {P^{(1)}}^\top A_S P^{(1)} ... P^{(k-1)}, \tag{12.6}$$

where $A_S^{(0)} = A_S$ and $P^{(k)} \in \mathbb{R}^{N^{(k)} \times N^{(k)}}$ is a coarse-grained operator for the kth level and $N^{(k)}$ refers to the number of nodes of the coarse-grained graph at level k. In the next stage, each coarse-grained graph at each level k will be reconstructed back into a fine graph adjacent matrix $A_T^{(k)} \in \mathbb{R}^{N^{(k)} \times N^{(k)}}$ as follows:

$$A_T^{(k)} = {R^{(1)}}^\top ... {R^{(k-1)}}^\top A_S^{(k)} R^{(k-1)} ... R^{(1)}, \tag{12.7}$$

where $R^{(k)} \in \mathbb{R}^{N^{(k)} \times N^{(k)}}$ is the reconstruction operator for the kth level. Thus all the reconstructed fine graphs at each layer are in the same scale. Finally, these graphs are aggregated into a unique one by a linear function to get the final adjacent matrix as follows: $A_T = \sum_{k=1}^{K} w^k A_T^{(k)} + b^k \mathbf{I}$, where $w^k \in \mathbb{R}$ and $b^k \in \mathbb{R}$ are weights and bias.

12.3.4 Graph Transformation Policy Networks

Beyond the general framework for edge-level transformation problem, it is necessary to deal with some domain-specific problems which may need to incorporate some domain knowledge or information into transformation process. For example, the chemical reaction product prediction problem is a typical edge-level transformation problem, where the input reactant and reagent molecules can be jointly represented as input graphs, and the process of generating product molecules (i.e., output graphs) from reactant molecules can be formulated as a set of edge-level graph transformations. Formalizing the chemical reaction product prediction problem as a edge-level transformation problem is beneficial due to two reasons: (1) it can capture and utilize the molecular graph structure patterns of the input reactants and reagents(i.e., atom pairs with changing connectivity); and (2) it can automatically chooses from these reactivity patterns a correct set of reaction triples to generate the desired products.

Do et al (2019) proposed a Graph Transformation Policy Network (GTPN), a novel generic method that combines the strengths of graph neural networks and reinforcement learning, to learn reactions directly from data with minimal chemical knowledge. The GTPN originally aims to generate the output graph by formalizing the graph transformation process as a Markov decision process and modifying the input source graph through several iterations. From the perspective of chemical reaction side, the process of reaction product prediction can be formulated as predicting a set of bond changes given the reactant and reagent molecules as input. A bond change is characterized by the atom pair that holds the bond (where is the change) and the new bond type (what is the change).

Mathematically, given a graph of reactant molecule as input graph, $\mathscr{G}_S$, they predict a set of reaction triples which transforms $\mathscr{G}_S$ into a graph of product molecule $\mathscr{G}_T$. This process is modeled as a sequence consisting of tuples like $(\zeta^t, v_i^t, v_j^t, b^t)$ where v_i^t and v_j^t are the selected nodes from node set at step t whose connection needs to be modified, b^t is the new edge type of (v_i^t, v_j^t) and ζ^t is a binary signal that indicates the end of the sequence. Generally, at every step of the forward pass, GTPN performs seven major steps: 1) computing the atom representation vectors through message passing neural network (MPNN); 2) computing the most possible K reaction atom pairs; 3) predicting the continuation signal ζ^t; 4) predicting the reaction atom pair (v_i^t, v_j^t); 5) predicting a new bond type b^t of this aton pair; 6) updating the atom representations; and 7) updating the recurrent state.

Specifically, the above iterative process of edge-level transformation is formulated as a Markov Decision Process (MDP) characterized by a tuple $(\mathscr{S}, \mathscr{A}, f_P, f_R, \Gamma)$, where $\mathscr{S}$ is a set of states, $\mathscr{A}$ is a set of actions, f_P is a state transition function, f_R is a reward function, and Γ is a discount factor. Thus, the overall model is optimized via the reinforcement learning. Specifically, a state $s^t \in \mathscr{S}$ is a immediate graph that is generated at the step t, and s^0 refers to the input graph. An action $a^t \in \mathscr{A}$ performed at step t is represented as a tuple $(\zeta^t, (v_i^t, v_j^t, b^t))$. The action is composed of three consecutive sub-actions: predicting ζ^t, (v_i^t, v_j^t) and b^t respectively. In the state

transition part, If $\zeta^t = 1$, the current graph $\mathscr{G}^t$ is modified based on the reaction triple (v_i^t, v_j^t, b^t) to generate a new intermediate graph $\mathscr{G}^{t+1}$. Regarding the reward, both immediate rewards and delayed rewards are utilized to encourage the model to learn the optimal policy faster. At every step t, if the model predicts $(\zeta^t, (v_i^t, v_j^t, b^t))$ correctly, it will receive a positive reward for each correct sub-action. Otherwise, a negative reward is given. After the prediction process has terminated, if the generated products are exactly the same as the ground-truth products, a positive delayed reward is also given, otherwise a negative reward.

Different from the encoder-decoder frameworks of GT-GAN, GTPN is a typical example of reinforcement learning-based graph transformation network, where the target graph is generated by making modifications on the input graphs in a iterative way. Reinforcement learning (RL) is a commonly used framework for learning controlling policies and generation process by a computer algorithm, the so-called agent, through interacting with its environment. The nature of reinforcement learning methods (i.e.,a sequential generation process) make it a suitable framework for graph transformation problems which sometime require the step-by-step edits on the input graphs to generate the final target output graphs.

12.4 Node-Edge Co-Transformation

12.4.1 Definition of Node-Edge Co-Transformation

Node-edge co-transformation (NECT) aims to generate node and edge attributes of the target graph conditioned on those of the input graph. It requires that both nodes and edges can vary during the transformation process between the source graph and the target graph as follows: $\mathscr{G}_S(\mathscr{V}_S, \mathscr{E}_S, F_S, E_S) \rightarrow \mathscr{G}_T(\mathscr{V}_T, \mathscr{E}_T, F_T, E_T)$. There are two categories of techniques used to assimilate the input graph to generate the target graph embedding-based and editing-based.

Embedding-based NECT usually encodes the source graph into latent representations using an encoder that contains higher-level information on the input graph which can then be decoded into the target graph by a decoder (Jin et al, 2020c, 2018c; Kaluza et al, 2018; Maziarka et al, 2020b; Sun and Li, 2019). These methods are usually based on either conditional VAEs (Sohn et al, 2015) or conditional GANs (Mirza and Osindero, 2014). Three main techniques will be introduced in this section, including junction-tree variational auto-encoder, molecule cycle-consistent adversarial networks and directed acyclic graph transformation networks.

12.4.1.1 Junction-tree Variational Auto-encoder Transformer

The goal of molecule optimization, which is one of the important molecule generation problems, is to optimize the properties of a given molecule by transforming it

into a novel output molecule with optimized properties. The molecule optimization problem is typically formalized as a NECT problem where the input graph refers to the initial molecule and the output graph refers to the optimized molecule. Both the node and edge attributes can change during the transformation process.

The Junction-tree Variational Auto-encoder (JT-VAE) is motivated by the key challenge of molecule optimization in the domain of drug design, which is to find target molecules with the desired chemical properties (Jin et al, 2018a). In terms of the model architecture, JT-VAE extends the VAE (Kingma and Welling, 2014) to molecular graphs by introducing a suitable encoder and a matching decoder. Under JT-VAE, each molecule is interpreted as being formalized from subgraphs chosen from a dictionary of valid components. These components serve as building blocks when encoding a molecule into a vector representation and decoding latent vectors back into optimized molecular graphs. The dictionary of components, such as rings, bonds and individual atoms, is large enough to ensure that a given molecule can be covered by overlapping clusters without forming cluster cycles. In general, JT-VAE generates molecular graphs in two phases, by first generating a tree-structured scaffold over chemical substructures and then combining them into a molecule with a graph message passing network.

The latent representation of the input graph $\mathscr{G}$ is encoded by a graph message passing network (Dai et al, 2016; Gilmer et al, 2017). Here, let $\mathbf{x}_v$ denote the feature vector of the vertex v, involving properties of the vertex such as the atom type and valence. Similarly, each edge $(u,v) \in \mathscr{E}$ has a feature vector $\mathbf{x}_{vu}$ indicating its bond type. Two hidden vectors ν_{uv} and ν_{vu} denote the message from u to v and vice versa. In the encoder, messages are exchanged via loopy belief propagation:

$$\nu_{uv}^{(t)} = \tau(W_1^g \mathbf{x}_u + W_2^g \mathbf{x}_{uv} + W_3^g \sum_{w \in N(u) \setminus v} \nu_{wu}^{(t-1)}), \tag{12.8}$$

where $\mathbf{v}_{uv}^t$ is the message computed in the t-th iteration, initialized with $\nu_{uv}^{(0)} = 0$, $\tau(\cdot)$ is the ReLU function, W_1^g, W_2^g and W_3^g are weights, and $N(u)$ denotes the neighbors of u. Then, after T iterations, the latent vector of each vertex is generated capturing its local graphical structure:

$$\mathbf{h}_u = \tau(U_1^g \mathbf{x}_u + \sum_{v \in N(u)} U_2^g \nu_{vu}^{(T)}), \tag{12.9}$$

where U_1^g and U_2^g are weights. The final graph representation is $\mathbf{h}_{\mathscr{G}} = \sum_i \mathbf{h}_i / |\mathscr{V}|$, where $|\mathscr{V}|$ is the number of nodes in the graph. The corresponding latent variable $\mathbf{z}_G$ can be sampled from $\mathscr{N}(\mathbf{z}_G; \mu_{\mathscr{G}}, \sigma_{\mathscr{G}}^2)$ and $\mu_{\mathscr{G}}$ and $\sigma_{\mathscr{G}}^2$ can be calculated from $h_{\mathscr{G}}$ via two separate affine layers.

A junction tree can be represented as $(\mathscr{V}, \mathscr{E}, \mathscr{X})$ whose node set is $\mathscr{V} = (C_1, ..., C_n)$ and edge set is $\mathscr{E} = (E_1, ..., E_n)$. This junction tree is labeled by the label dictionary $\mathscr{X}$. Similar to the graph representation, each cluster C_i is represented by a one-hot $\mathbf{x}_i$ and each edge (C_i, C_j) corresponds to two message vectors $\mathbf{v}_{ij}$ and $\mathbf{v}_{ji}$. An arbitrary leaf node is picked as the root and messages are propagated in two phases:

$$\begin{aligned}
\mathbf{s}_{ij} &= \sum_{k \in N(i) \setminus j} \mathbf{v}_{ki} \\
\mathbf{z}_{ij} &= \sigma(W^z \mathbf{x}_i + U^z \mathbf{s}_{ij} + \mathbf{b}^z) \\
\mathbf{r}_{ki} &= \sigma(W^r \mathbf{x}_i + U^r \mathbf{v}_{ki} + \mathbf{b}^r) \\
\tilde{\mathbf{v}}_{ij} &= tanh(W \mathbf{x}_i + U \sum_{k \in N(i) \setminus j} \mathbf{r}_{ki} \odot \mathbf{v}_{ki}) \\
\mathbf{v}_{ij} &= (1 - \mathbf{z}_{ij}) \odot \mathbf{s}_{ij} + \mathbf{z}_{ij} \odot \tilde{\mathbf{v}}_{ij}.
\end{aligned} \tag{12.10}$$

$\mathbf{h}_i$, the latent representation of node v_i can now be calculated:

$$\mathbf{h}_i = \tau(W^o \mathbf{x}_i + \sum_{k \in N(u)} U^o \mathbf{v}_{ki}) \tag{12.11}$$

The final tree representation is $\mathbf{h}_{\mathscr{T}_\mathscr{G}} = \mathbf{h}_{root}$. $\mathbf{z}_{\mathscr{T}_\mathscr{G}}$ is sampled in a similar way as in the encoding process.

Under the JT-VAE framework, the junction tree is decoded from $\mathbf{z}_{\mathscr{T}_\mathscr{G}}$ using a tree-structured decoder that traverses the tree from the root and generates nodes in their depth-first order. During this process, a node receives information from other nodes, and this information is propagated through message vectors $\mathbf{h}_{ij}$. Formally, let $\tilde{\mathscr{E}} = \{(i_1, j_1), ..., (i_m, j_m)\}$ be the set of edges traversed over the junction tree $(\mathscr{V}, \mathscr{E})$, where $m = 2|\mathscr{E}|$ because each edge is traversed in both directions. The model visits node i_t at time t. Let $\tilde{\mathscr{E}}_t$ be the first t edges in $\tilde{\mathscr{E}}$. The message is updated as $\mathbf{h}_{i_t, j_t} = GRU(\mathbf{x}_{i_t}, \{\mathbf{h}_{k,i_t}\}_{(k,i_t) \in \tilde{\mathscr{E}}_t, k \neq j_t})$, where $\mathbf{x}_{i_t}$ corresponds to the node features. The decoder first makes a prediction regarding whether the node i_t still has children to be generated, in which the probability is calculated as:

$$p_t = \sigma(\mathbf{u}^d \cdot \tau(W_1^d \mathbf{x}_{i_t} + W_2^d \mathbf{z}_{\mathscr{T}_\mathscr{G}} + W_3^d \sum_{(k,i_t) \in \tilde{\mathscr{E}}_t} \mathbf{h}_{k,i_t})), \tag{12.12}$$

where $\mathbf{u}^d$, W_1^d, W_2^d and W_3^d are weights. Then, when a child node j is generated from its parent i, its node label is predicted with:

$$\mathbf{q}_j = softmax(U^l \cdot \tau(W_1^l \mathbf{z}_{\mathscr{T}_\mathscr{G}} + W_2^l \mathbf{h}_{ij})), \tag{12.13}$$

where U^l, W_1^l and W_2^l are weights and $\mathbf{q}_j$ is a distribution over label dictionary $\mathscr{X}$.

The final step of the model is to reproduce a molecular graph $\mathscr{G}$ to represent the predicted junction tree $(\hat{\mathscr{V}}, \hat{\mathscr{E}})$ by assembling the subgraphs together into the final molecular graph. Let $\mathscr{G}(\mathscr{T}_\mathscr{G})$ be a set of graphs corresponding to the junction tree $\mathscr{T}_\mathscr{G}$. Decoding graph $\hat{\mathscr{G}}$ from the junction tree $\hat{\mathscr{T}}_\mathscr{G} = (\hat{\mathscr{V}}, \hat{\mathscr{E}})$ is a structured prediction:

$$\hat{\mathscr{G}} = \arg \max_{\mathscr{G}' = \mathscr{G}(\hat{\mathscr{T}}_\mathscr{G})} f^a(\mathscr{G}'), \tag{12.14}$$

where f^a is a scoring function over candidate graphs. The decoder starts by sampling the assembly of the root and its neighbors according to their scores, then proceeds to assemble the neighbors and associated clusters. In terms of scoring the realization of each neighborhood, let $\mathscr{G}_i$ be the subgraph resulting from a particular merging of cluster C_i in the tree with its neighbors $C_j, j \in N_{\hat{\mathscr{T}}_\mathscr{G}}(i)$. $\mathscr{G}_i$ is scored as a candidate subgraph by first deriving a vector representation $\mathbf{h}_{\mathscr{G}_i}$, and $f_i^a(\mathscr{G}_i) = \mathbf{h}_{\mathscr{G}_i} \cdot \mathbf{z}_\mathscr{G}$ is the

subgraph score. For atoms in $\mathcal{G}_i$, let $\alpha_v = i$ if $v \in C_i$ and $\alpha_v = j$ if $v \in C_j \backslash C_i$ to mark the position of atoms in the junction tree and retrieve messages $\hat{\mathbf{m}}_{i,j}$, summarizing the subtree under i along the edge (i, j) obtained by the tree encoder. Then the neural messages can be obtained and aggregated similarly to the encoding step with parameters:

$$\begin{aligned} \mu_{uv}^{(t)} &= \tau(W_1^a \mathbf{x}_u + W_2^a \mathbf{x}_{uv} + W_3^a \hat{\mu}_{uv}^{(t-1)}) \\ \tilde{\mu}_{uv}^{(t-1)} &= \begin{cases} \sum_{w \in N(u) \backslash v} \mu_{wu}^{(t-1)} & \alpha_u = \alpha_v \\ \hat{m}_{\alpha_u, \alpha_v} + \sum_{w \in N(u) \backslash v} \mu_{wu}^{(t-1)} & \alpha_u \neq \alpha_v, \end{cases} \end{aligned} \tag{12.15}$$

where W_1^a, W_2^a and W_3^a are weights.

12.4.1.2 Molecule Cycle-Consistent Adversarial Networks

Cycle-consistent adversarial networks, an alternative to achieve embedding-based NECT, were originally developed to achieve image-to-image transformations. The aim here is to learn to transform an image from a source domain to a target domain in the absence of paired examples by using an adversarial loss. To promote the chemical compound design process, this idea has been borrowed for graph transformation. For instance, Molecule Cycle-Consistent Adversarial Networks (Mol-CycleGAN) have been proposed to generate optimized compounds with high structural similarity to the originals (Maziarka et al, 2020b). Given a molecule $\mathcal{G}_X$ with the desired molecular properties, Mol-CycleGAN aims to train a model to perform the transformation $G : \mathcal{G}_X \to \mathcal{G}_Y$ and then use this model to optimize the molecules. Here $\mathcal{G}_Y$ is the set of molecules without the desired molecular properties. In order to represent the sets $\mathcal{G}_X$ and $\mathcal{G}_Y$, this model requires a reversible embedding that allows both the encoding and decoding of molecules. To achieve this, JT-VAE is employed to provide the latent space during the training process, during which the distance between molecules required to calculate the loss function can be defined directly. Each molecule is represented as a point in latent space, assigned based on the mean of the variational encoding distribution.

For the implementation, the sets $\mathcal{G}_X$ and $\mathcal{G}_Y$ must be defined (e.g., inactive/active molecules), after which the mapping functions $G : \mathcal{G}_X \to \mathcal{G}_Y$ and $F : \mathcal{G}_Y \to \mathcal{G}_X$ are introduced. The discriminators D_X and D_Y are proposed to force generators F and G to generate samples from a distribution close to the distributions of $\mathcal{G}_X$ and $\mathcal{G}_Y$. For this process, F, G, D_X and D_Y are modeled by neural networks. This approach to molecule optimization is designed to (1) take a prior molecule x with no specified features from set $\mathcal{G}_X$ and compute its latent space embedding; (2) use generative neural network G to obtain the embedding of molecule $G(x)$ that has this feature but is also similar to the original molecule x; and (3) decode the latent space coordinates given by $G(x)$ to obtain the optimized molecule.

The loss function to train Mol-CycleGAN is:

$$L(G,F,D_X,D_Y) = L_{GAN}(G,D_Y,\mathscr{G}_X,\mathscr{G}_Y) + L_{GAN}(F,D_X,\mathscr{G}_Y,\mathscr{G}_X) + \lambda_1 L_{cyc}(G,F) + \lambda_2 L_{identity}(G,F), \tag{12.16}$$

and $G^*,F^* = \arg\min_{G,F}\max_{D_X,D_Y} L(G,F,D_X,D_Y)$. The adversarial loss is utilized:

$$L_{GAN}(G,D_Y,\mathscr{G}_X,\mathscr{G}_Y) = \frac{1}{2}\mathbb{E}_{y\sim p_{data}^{\mathscr{G}_Y}}[(D_Y(y)-1)^2] + \frac{1}{2}\mathbb{E}_{x\sim p_{data}^{\mathscr{G}_X}}[D_Y(G(x))^2], \tag{12.17}$$

which ensures that the generator G (and F) generates samples from a distribution close to the distribution of $\mathscr{G}_Y$ (or $\mathscr{G}_X$), denoted by $p_{data}^{\mathscr{G}_Y}$ (or $p_{data}^{\mathscr{G}_X}$). The cycle consistency loss

$$L_{cyc}(G,F) = \mathbb{E}_{y\sim p_{data}^{\mathscr{G}_Y}}[\|G(F(y))-y\|_1] + \mathbb{E}_{x\sim p_{data}^{\mathscr{G}_X}}[\|F(G(x))-x\|_1], \tag{12.18}$$

reduces the space available to the possible mapping functions such that for a molecule x from set $\mathscr{G}_X$, the GAN cycle constrains the output to a molecule similar to x. The inclusion of the cyclic component acts as a regularization factor, making the model more robust. Finally, to ensure that the generated molecule is close to the original, identity mapping loss is employed:

$$L_{identity}(G,F) = \mathbb{E}_{y\sim p_{data}^{\mathscr{G}_Y}}[\|F(y)-y\|_1] + \mathbb{E}_{x\sim p_{data}^{\mathscr{G}_X}}[\|G(x)-x\|_1], \tag{12.19}$$

which further reduces the space available to the possible mapping functions and prevents the model from generating molecules that lay far away from the starting molecule in the latent space of JT-VAE.

12.4.1.3 Directed Acyclic Graph Transformation Networks

Another example of embedding-based NECT is a neural model for learning deep functions on the space of directed acyclic graphs (DAGs) (Kaluza et al, 2018). Mathematically, the neural methodologies developed to handle graph-structured data can be regarded as function approximation frameworks where both the domain and the range of the target function can be graph spaces. In the area of interest here, the embedding and synthesis methodologies are gathered into a single unified framework such that functions can be learned from one graph space onto another graph space without the need to impose strong assumption of independence between the embedding and generative process. Note that only functions in DAG space are considered here. A general encoder-decoder framework for learning functions from one DAG space onto another has been developed.

Here, RNN is employed to model the function F, denoted as D2DRNN. Specifically, the model consists of an encoder E_α with model parameters α that compute a fixed-size embedding of the input graph $\mathscr{G}_{in}$, and a decoder D_β with parameters β, using the embedding as input and producing the output graph $\hat{\mathscr{G}}_{out}$. Alternatively, the DAG-function can be defined as $F(\mathscr{G}_{in}) := D_\beta(E_\alpha(\mathscr{G}_{in}))$.

The encoder is borrowed from the deep-gated DAG recursive neural network (DG-DAGRNN) (Amizadeh et al, 2018), which generalizes the stacked recurrent neural networks (RNNs) on sequences to DAG structures. Each layer of DG-DAGRNN consists of gated recurrent units (GRUs) (Cho et al, 2014a), which are repeated for each node $v_i \in \mathscr{G}_{in}$. The GRU corresponding to node v contains an aggregated representation of the hidden states of the units regarding its predecessors $\pi(v)$. For an aggregation function A:

$$\mathbf{h}_v = GRU(\mathbf{x}_v, \mathbf{h}'_v), \text{ where } \mathbf{v}' = \mathbf{A}(\{\mathbf{h_u} | \mathbf{u} \in \pi(\mathbf{v})\}). \tag{12.20}$$

Since the ordering of the nodes is defined by the topological sort of $\mathscr{G}_{in}$, all the hidden states $\mathbf{h}_v$ can be computed with a single forward pass along a layer of DG-DAGRNN. The encoder contains multiple layers, each of which passes hidden states to the recurrent units in the subsequent layer corresponding to the same node.

The encoder outputs an embedding $H_{in} = E_\alpha(\mathscr{G}_{in})$, which serves as the input of the DAG decoder. The decoder follows the local-based node-sequential generation style. Specifically, first, the number of nodes of the target graph is predicted by a multilayer perceptron (MLP) with a Poisson regressor output layer, which takes the input graph embedding H_{in} and outputs the mean of a Poisson distribution describing the output graph. Whether it is necessary add an edge e_{u,v_n} for all the nodes $u \in \{v_1, ..., v_{n-1}\}$ already in the graph is determined by a module of MLP. Since the output nodes are generated in their topological order, the edges are directed from the nodes added earlier to the nodes added later. For each node v, the hidden state $\mathbf{h}_v$ is calculated using a similar mechanism to that used in the encoder, after which they are aggregated and fed to a GRU. The other input for the GRU consists of the aggregated states of all the sink nodes generated so far. For the first node, the hidden state is initialized based on the encoder's output. Then, the output node features are generated based on its hidden state using another module of MLP. Finally, once the last node has been generated, the edges are introduced with probability 1 for sinks in the graph to ensure a connected graph with only one sink node as an output.

12.4.2 Editing-based Node-Edge Co-Transformation

Unlike the encoder-decoder framework, modification-based NECT directly modifies the input graph iteratively to generate the target graphs (Guo et al, 2019c; You et al, 2018a; Zhou et al, 2019c). Two methods are generally used to edit the source graph. One employs a reinforcement-learning agent to sequentially modify the source graph based on a formulated Markov decision process (You et al, 2018a;

Zhou et al, 2019c). The modification at each step is selected from a defined action set that includes "add node", "add edge", "remove bonds" and so on. Another is to update the nodes and edges from the source graph synchronously in a one-shot manner through the MPNN using several iterations (Guo et al, 2019c).

12.4.2.1 Graph Convolutional Policy Networks

Motivated by the large size of chemical space, which can be an issue when designing molecular structures, graph convolutional policy networks (GCPNs) serve as useful general graph convolutional network-based models for goal-directed graph generation through reinforcement learning (RL) (You et al, 2018a). In this model, the generation process can be guided towards the specific desired objectives, while restricting the output space based on underlying chemical rules. To achieve goal-directed generation, three strategies, namely graph representation, reinforcement learning, and adversarial trainings are adopted. In GCPN, molecules are represented as molecular graphs, and partially generated molecular graphs can be interpreted as substructures. GCTN is designed as an RL agent which operates within a chemistry-aware graph generation environment. A molecule is successively constructed by either connecting a new substructure or atom to an existing molecular graph by adding a bond. GCPN is trained to optimize the domain-specific properties of the source molecule by applying a policy gradient to optimize it via a reward composed of molecular property objectives and adversarial loss; it acts in an environment which incorporates domain-specific rules. The adversarial loss is provided by a GCN-based discriminator trained jointly on a dataset of example molecules.

An iterative graph generation process is designed and formulated as a general decision process $M = (\mathscr{S}, \mathscr{A}, P, R, \gamma)$, where $\mathscr{S} = \{s_i\}$ is the set of states that comprises all possible intermediate and final graphs. $\mathscr{A} = (a_i)$ is the set of actions that describe the modifications made to the current graph during each iteration, P represents the transition dynamics that specify the possible outcomes of carrying out an action $p(s_{t+1}|s_t,...,s_0,a_t)$, $R(s_t) = r_t$ is a reward function that specifies the reward after reaching state s_t and γ is the discount factor. The graph generation process can now be formulated as $(s_0,a_0,r_0,...,s_n,a_n,r_n)$, and the modification of the graph at each time can be described as a state transition distribution: $p(s_{t+1}|s_t,...,s_0) = \sum_{a_t} p(a_t|s_t,...,s_0)p(s_{t+1}|s_t,...,s_0,a_t)$, where $p(a_t|s_t,...,s_0)$ is represented as a policy network π_θ. Note that in this process, the state transition dynamics are designed to satisfy the Markov property $p(s_{t+1}|s_t,...s_0) = p(s_{t+1}|s_t)$.

In this model, a distinct, fixed-dimension, homogeneous action space is defined and amenable to reinforcement learning, where an action is analogous to link prediction. Specifically, a set of scaffold subgraphs $\{C_1,...,C_s\}$ is first defined based on the source graph, thus serving as a subgraph vocabulary that contains the subgraphs to be added into the target graph during graph generation. Define $C = \cup_{i=1}^{s} C_i$. Given the modified graph $\mathscr{G}_t$ at step t, the corresponding extended graph can be defined as $\mathscr{G}_t \cup C$. Under this definition, an action can either correspond to connecting a new subgraph C_i to a node in $\mathscr{G}_t$ or connecting existing nodes within graph $\mathscr{G}_t$. GAN is

also employed to define the adversarial rewards to ensure that generated molecules do indeed resemble the originals.

Node embedding is achieved by message passing over each edge type for L layers through GCN. At the l-th layer of GCN, messages from different edge types are aggregated to calculate the node embedding $H^{(l+1)} \in \mathbb{R}^{(n+c)\times k}$ of the next layer, where n and c are the sizes of $\mathscr{G}_t$ and C, respectively, and k is the embedding dimension:

$$H^{(l+1)} = AGG(ReLU(\{\hat{D}_i^{-\frac{1}{2}}\hat{E}_i\hat{D}_i^{-\frac{1}{2}}H^{(l)}W_i^{(l)}\}, \forall i \in (1,...,b))). \tag{12.21}$$

E_i is the i^{th} slice of the edge-conditioned adjacency tensor E, and $\hat{E}_i = E_i + \mathbf{I}$; $\hat{D}_i = \sum_k \hat{E}_{ijk}$ and $W_i^{(l)}$ is the weight matrix for the i^{th} edge type. AGG denotes one of the aggregation functions from $\{MEAN, MAX, SUM, CONTACT\}$.

The link prediction-based action a_t ensures each component samples from a prediction distribution governed by the equations below:

$$a_t = CONCAT(a_{first}, a_{second}, a_{edge}, a_{stop}) \tag{12.22}$$

$$\begin{aligned}
&f_{first}(s_t) = softmax(m_f(X)), && a_{first} \sim f_{first}(s_t) \in \{0,1\}^n \\
&f_{second}(s_t) = softmax(m_s(X_{a_{first}}, X)), && a_{second} \sim f_{second}(s_t) \in \{0,1\}^{n+c} \\
&f_{edge}(s_t) = softmax(m_e(X_{a_{first}}, X)), && a_{edge} \sim f_{edge}(s_t) \in \{0,1\}^b \\
&f_{stop}(s_t) = softmax(m_t(AGG(X))), && a_{stop} \sim f_{stop}(s_t) \in \{0,1\}
\end{aligned} \tag{12.23}$$

Here m_f, m_s, m_e and m_f denote MLP modules.

12.4.2.2 Molecule Deep Q-networks Transformer

In addition to GCPN, molecule deep Q-networks (MolDQN) has also been developed for molecule optimization under the node-edge co-transformation problem utilizing an editing-based style. This combines domain knowledge of chemistry with state-of-the-art reinforcement learning techniques (double Q-learning and randomized value functions) (Zhou et al, 2019c). In this field, traditional methods usually employ policy gradients to generate graph representations of molecules, but these suffer from high variance when estimating the gradient (Gu et al, 2016). In comparison, MolDQN is based on value function learning, which is usually more stable and sample efficient. MolDQN also avoids the need for expert pretraining on some datasets, which may lead to lower variance but limits the search space considerably.

In the framework proposed here, modifications of molecules are directly defined to ensure 100% chemical validity. Modification or optimization is performed in a step-wise fashion, where each step belongs to one of the following three categories: (1) atom addition, (2) bond addition, and (3) bond removal. Because the molecule generated depends solely on the molecule being changed and the modification made, the optimization process can be formulated as a Markov decision process (MDP).

Specifically, when performing the action *atom addition*, an empty set of atoms $\mathcal{V}_T$ for the target molecule graph is first defined. Then, a valid action is defined as adding an atom in $\mathcal{V}_T$ and also a bond between the added atom and the original molecule wherever possible. When performing the action *bond addition*, a bond is added between two atoms in $\mathcal{V}_T$. If there is no existing bond between the two atoms, the actions between them can consist of adding a single, double or triple bond. If there is already a bond, this action changes the bond type by increasing the index of the bond type by one or two. When performing the action *bond removal*, the valid bond removal action set is defined as the actions that decrease the bond type index of an existing bond. Possible transitions include: (1) Triple bond $\rightarrow$ {Double, Single, No} bond, (2) Double bond $\rightarrow$ {Single, No} bond, and (3) Single bond $\rightarrow$ {No} bond.

Based on the molecule modification MDP defined above, RL aims to find a policy π that chooses an action for each state that maximizes future rewards. Then, the decision is made by finding the action a for a state s to maximize the Q function:

$$Q^{\pi}(s,a) = Q^{\pi}(m,t,a) = \mathbb{E}_{\pi}[\sum_{n=t}^{T} r_n], \tag{12.24}$$

where r_n is the reward at step n. The optimal policy can therefore be defined as $\pi^*(s) = \arg\max_a Q^{\pi^*}(s,a)$. A neural network is adopted to approximate $Q(s,a,\theta)$, and can be trained by minimizing the loss function:

$$l(\theta) = \mathbb{E}[f_l(y_t - Q(s_t,a_t;\theta))], \tag{12.25}$$

where $y_t = r_t + \max_a Q(s_{t+1},a;\theta)$ is the target value and f_l is the Huber loss:

$$f_l(x) = \begin{cases} \frac{1}{2}x^2 & \text{if } |x| < 1 \\ |x| - \frac{1}{2} & \text{otherwise} \end{cases} \tag{12.26}$$

In a real-world setting, it is usually desirable for several different properties to be optimized at the same time. Under the multi-objective RL setting, the environment will return a vector of rewards at each step t with one reward for each objective. A "scalar" reward framework is applied to achieve multi-objective optimization, with the introduction of a user defined weight vector $\mathbf{w} = [w_1, w_2, ..., w_k]^\top \in \mathbb{R}^k$. The reward is calculated as:

$$r_{s,t} = \mathbf{w}^\top \overrightarrow{\mathbf{r}_t} = \sum_{i=1}^{k} w_i r_{i,t}. \tag{12.27}$$

The objective of MDP is to maximize the cumulative scalarized reward.

The Q-learning model (Mnih et al, 2015) is implemented here, incorporating the improvements gained using double Q-learning (Van Hasselt et al, 2016), with a deep neural network being used to approximate the Q-function. The input molecule is converted to a vector, Taking the form of a Morgan fingerprint (Rogers and Hahn, 2010) a the radius of 3 and length of 2048. The number of steps remaining in the episode is concatenated to the vector and a four-layer fully-connected network with

hidden state size of [1024, 512, 128, 32] and ReLU activation is used as the architecture.

12.4.2.3 Node-Edge Co-evolving Deep Graph Translator

To overcome a number of challenges including, but not limited to, the mutually dependent translation of the node and edge attributes, asynchronous and iterative changes in the node and edge attributes during graph translation, and the difficulty of discovering and enforcing the correct consistency between node attributes and graph spectra, the Node-Edge Co-evolving Deep Graph Translator (NEC-DGT) has been developed to achieve so-called multi-attributed graph translation and proven to be a generalization of the existing topology translation models (Guo et al, 2019c). This is a node-edge co-evolving deep graph translator that edits the source graph iteratively through a generation process similar to the MPNN-based adjacency-based one-shot method for unconditional deep graph generation, with the main difference being that it takes the graph in the source domain as input rather than the initialized graph (Guo et al, 2019c).

NEC-DGT employs a multi-block translation architecture to learn the distribution of the graphs in the target domain, conditioning on the input graphs and contextual information. Specifically, the inputs are the node and graph attributes, and the model outputs are the generated graphs' node and edge attributes after several blocks. A skip-connection architecture is implemented across the different blocks to handle the asynchronous properties of different blocks, ensuring the final translated results fully utilize various combinations of blocks' information. The following loss function is minimized in the work:

$$\mathscr{L}_{\mathscr{T}} = \mathscr{L}(\mathscr{T}(\mathscr{G}(E_0, F_0), C), \mathscr{G}(E', F')), \tag{12.28}$$

where C corresponds to the contextual information vector, E_0, E' corresponds to the edge attribute tensors of the input and target graphs, respectively, and F_0, F' corresponds to the node attribute tensors of the input and target graphs, respectively.

To jointly handle the various interactions among the nodes and edges, the respective translation paths are considered for each block. For example, in the node translation path, *edges-to-nodes* and *nodes-to-nodes* interactions are considered in the generation of node attributes. Similarly, "node to edges" and "edges-to-edges" are considered in the generation of edge attributes.

The frequency domain properties of the graph are learned, by which the interactions between node and edge attributes are jointly regularized utilizing a non-parametric graph Laplacian. Also, shared patterns among the generated nodes and edges in different blocks are enforced through regularization. Then, the regularization term is

$$\mathscr{R}(\mathscr{G}(E, F)) = \sum_{s=0}^{S} \mathscr{R}_{\theta}(\mathscr{G}(E_S, F_S)) + \mathscr{R}_{\theta}, \tag{12.29}$$

where S corresponds to the number of blocks and θ refers to the overall parameters in the spectral graph regularization. $\mathscr{G}(E_S, F_S)$ is the generated target graph, where E_S is the generated edge attributes tensor and F_S is the node attributes matrix. Then the total loss function is

$$\tilde{\mathscr{L}} = \mathscr{L}(T(\mathscr{G}(E_0, F_0), C), \mathscr{G}(E', F')) + \beta \mathscr{R}(G(E, F)). \tag{12.30}$$

The model is trained by minimizing the MSE of E_S with E', F_S with F', enforced by the regularization. $T(\cdot)$ is the mapping from the input graph to the target graph learned from the multi-attributed graph translation.

The transformation process is modeled by several stages with each stage generating an immediate graph. Specifically, for each stage t, there are two options: node translation paths and edge translation paths. In the node translation path, an MLP-based influence-function is used to calculate the influence $I_i^{(t)}$ on each node v_i from its neighboring nodes. Another MLP-based updating-function is used to update the node attribute as $F_i^{(t)}$ with the input of influence $I_i^{(t)}$. The edge translation path is constructed in the same way as the node translation path, with each edge being generated by the influence from its adjacent edges.

12.5 Other Graph-based Transformations

12.5.1 Sequence-to-Graph Transformation

A deep sequence-to-graph transformation aims to generate a target graph G_T conditioned on an input sequence X. This problem is often seen in domains such as NLP (Chen et al, 2018a; Wang et al, 2018g) and time series mining (Liu et al, 2015; Yang et al, 2020c).

Existing methods (Chen et al, 2018a; Wang et al, 2018g) handle the semantic parsing task by transforming a sequence-to-graph problem into a sequence-to-sequence problem and utilizing the classical RNN-based encoder-decoder model to learn this mapping. A neural semantic parsing approach, named Sequence-to-Action, models semantic parsing as an end-to-end semantic graph generation process (Chen et al, 2018a). Given a sentence $X = \{x_1, ..., x_m\}$, the Sequence-to-Action model generates a sequence of actions $Y = \{y_1, .., y_m\}$ when constructing the correct semantic graph. A semantic graph consists of nodes (including variables, entities, and types) and edges (semantic relationships), with universal operations (e.g., argmax, argmin, count, sum, and not). To generate a semantic graph, six types of actions are defined: *Add Variable Node*, *Add Entity Node*, *Add Type Node*, *Add Edge*, *Operation Function* and *Argument Action*. In this way, the generated parse tree is represented as a sequence, and the sequence-to-graph problem is transformed into a sequence-to-sequence problem. The attention-based sequence-to-sequence RNN model with an encoder and decoder can be utilized, where the encoder converts the input sequence X to a sequence of context sensitive vectors $\{\mathbf{b}_1, ..., \mathbf{b}_m\}$ using a bidi-

rectional RNN and a classical attention-based decoder generates action sequence Y based on the context sensitive vectors (Bahdanau et al, 2015). The generation of a parse tree as a sequence of actions is represented (Wang et al, 2018g) and concepts from the Stack-LSTM neural parsing model are borrowed, producing two non-trivial improvements, Bi-LSTM subtraction and incremental tree-LSTM, that improve the process of learning a sequence-to-sequence mapping (Dyer et al, 2015).

Other methods have also been developed to handle the problem of Time Series Conditional Graph Generation (Liu et al, 2015; Yang et al, 2020c): given an input multivariate time series, the aim is to infer a target relation graph to model the underlying interrelationship between the time series and each node. A novel model of time series conditioned graph generation-generative adversarial networks (TSGG-GAN) for time series conditioned graph generation has been proposed that explores the use of GANs in a conditional setting (Yang et al, 2020c). Specifically, the generator in a TSGG-GAN adopts a variant of recurrent neural networks known as simple recurrent units (SRU) (Lei et al, 2017b) to extract essential information from the time series, and uses an MLP to generate the directed weighted graph.

12.5.2 Graph-to-Sequence Transformation

A number of graph-to-sequence encoder-decoder models have been proposed to handle rich and complex data structures, which are hard for sequence-to-sequence methods to handle (Gao et al, 2019c; Bastings et al, 2017; Beck et al, 2018; Song et al, 2018; Xu et al, 2018c). A graph-to-sequence model typically employs a graph-neural-network-based (GNN-based) encoder and an RNN/Transformer-based decoder, with most being designed to tackle tasks such as natural language generation (NLG), which is an important task in NLP (YILMAZ et al, 2020). Graph-to-sequence models have the ability to capture the rich structural information of the input and can also be applied to arbitrary graph-structured data.

Early graph-to-sequence methods and their follow-up works (Bastings et al, 2017; Damonte and Cohen, 2019; Guo et al, 2019e; Marcheggiani et al, 2018; Xu et al, 2020b,d; Zhang et al, 2020d,c) have mainly used a graph convolutional network (GCN) (Kipf and Welling, 2017b) as the graph encoder, probably because GCN was the first widely used GNN model that sparked this new wave of research on GNNs and their applications. Early GNN variants, such as GCN, were not originally designed to encode information on the edge type and so cannot be directly applied to the encoding of multi-relational graphs in NLP. Later on, more graph transformer models (Cai and Lam, 2020; Jin and Gildea, 2020; Koncel-Kedziorski et al, 2019) were introduced to the graph-to-sequence architecture to handle these multi-relational graphs. These graph transformer models generally function by either replacing the self-attention network in the original transformer with a masked self-attention network, or explicitly incorporate edge embeddings into the self-attention network.

Because edge direction in an NLP graph often encodes critical information regarding semantic meanings, capturing bidirectional information in the text is helpful and has been widely explored in works such as BiLSTM and BERT (Devlin et al, 2019). Some attention has also been devoted to extending the existing GNN models to handle directed graphs. For example, separate model parameters can be introduced for different edge directions (e.g., incoming/outgoing/self-loop edges) when conducting neighborhood aggregation (Guo et al, 2019e; Marcheggiani et al, 2018; Song et al, 2018). A BiLSTM-like strategy has also been proposed to learn the node embeddings of each direction independently using two separate GNN encoders and then concatenating the two embeddings for each node to obtain the final node embeddings (Xu et al, 2018b,c,d).

In the field of NLP, graphs are usually multi-relational, where the edge type information is vital for the prediction. Similar to the bidirectional graph encoder introduced above, separate model parameters for different edge types are considered when encoding edge type information with GNNs (Chen et al, 2018e; Ghosal et al, 2020; Schlichtkrull et al, 2018). However, usually the total number of edge types is large, leading to non-negligible scalability issues for the above strategies. This problem can be tackled by converting a multi-relational graph to a Levi graph (Levi, 1942), which is bipartite. To create a Levi graph, all the edges in the original graph are treated as new nodes and new edges are added to connect the original nodes and new nodes.

Apart from NLP, graph-to-sequence transformation has been employed in other fields, for example when modeling complex transitions of an individual user's activities among different healthcare subforums over time and learning how this is related to his various health conditions (Gao et al, 2019c). By formulating the transition of user activities as a dynamic graph with multi-attributed nodes, the health stage inference is formalized as a dynamic graph-to-sequence learning problem and, hence, a dynamic graph-to-sequence neural network architecture (DynGraph2Seq) has been proposed (Gao et al, 2019c). This model contains a dynamic graph encoder and an interpretable sequence decoder. In the same work, a dynamic graph hierarchical attention mechanism capable of capturing entire both time-level and node-level attention is also proposed, providing model transparency throughout the whole inference process.

12.5.3 Context-to-Graph Transformation

Deep graph generation conditioning on semantic context aims to generate the target graph $\mathcal{G}_T$ conditioning on an input semantic context that is usually represented in the form of additional meta-features. The semantic context can refer to the category, label, modality, or any additional information that can be intuitively represented as a vector C. The main issue here is to decide where to concatenate or embed the condition representation into the generation process. As a summary, the conditioning information can be added in terms of one or more of the following modules: (1)

the node state initialization module, (2) the message passing process for MPNN-based decoding, and (3) the conditional distribution parameterization for sequential generating.

A novel unified model of graph variational generative adversarial nets has been proposed, where the conditioning semantic context is input into the node state initialization module (Yang et al, 2019a). Specifically, the generation process begins by modeling the embedding Z_i of each node with the separate latent distributions, after which a conditional graph VAE (CGVAE) can be directly constructed by concatenating the condition vector C to each node's latent representation Z_i to obtain the updated node latent representation $\hat{Z}_i$. Thus, the distribution of the individual edge $\mathscr{E}_{i,j}$ is assumed to be a Bernoulli distribution, which is parameterized by the value $\hat{\mathscr{E}}_{i,j}$ and calculated as $\hat{\mathscr{E}}_{i,j} = \text{Sigmoid}(f(\hat{Z}_i)^\top f(\hat{Z}_j))$, where $f(\cdot)$ is constructed using a few fully connected layers. A conditional deep graph generative model that adds the semantic context information into the initialized latent representations Z_i at the beginning of the decoding process has also been proposed (Li et al, 2018d).

Other researchers have added the context information C into the message passing module as part of its MPNN-based decoding process (Li et al, 2018f). Specifically, the decoding process is parameterized as a Markov process and the graph generated by iteratively refining and updating the initialized graph. At each step t, an action is conducted based on the current node's hidden states $H^t = \{\mathbf{h}_1^t, ..., \mathbf{h}_N^t\}$. To calculate $\mathbf{h}_i^t \in \mathbb{R}^l$ (l denotes the length of the representation) for node v_i in the intermediate graph $\mathscr{G}_t$ after each updating of the graph, a message passing network is utilized with node message propagation. Thus, the context information $C \in \mathbb{R}^k$ is added to the operation of the MPNN layer as follows:

$$\mathbf{h}_i^t = W\mathbf{h}_i^{t-1} + \Phi \sum\nolimits_{v_j \in N(v_j)} \mathbf{h}_j^{t-1} + \Theta C, \tag{12.31}$$

where $W \in \mathbb{R}^{l \times l}$, $\Theta \in \mathbb{R}^{l \times l}$ and $\Phi \in \mathbb{R}^{k \times l}$ are all learnable weights vectors and k denotes the length of the semantic context vector.

Semantic context has also been considered as one of the inputs for calculating the conditional distribution parameter at each step during the sequential generating process (Jonas, 2019). The aim here is to solve the molecule inverse problem by inferring the chemical structure conditioning on the formula and spectra of a molecule, which provides a distinguishable fingerprint of its bond structure. The problem is framed as an MDP and molecules are constructed incrementally one bond at a time based on a deep neural network, where they learn to imitate a "subisomorphic oracle" that knows whether the generated bonds are correct. The context information (e.g., spectra) is applied in two places. The process begins with an empty edge set $\mathscr{E}_0$ that is sequentially updated to $\mathscr{E}_k$ at each step k by adding an edge sampled from $p(e_{i,j}|\mathscr{E}_{k-1}, \mathscr{V}, C)$. $\mathscr{V}$ denotes the node set that is defined in the given molecular formula. The edge set keeps updating until the existing edges satisfy all the valence constraints of a molecule. The resulting edge set $\mathscr{E}_K$ then serves as the candidate graph. For a given spectrum C, the process is repeated T times, generating T (potentially different) candidate structures, $\{\mathscr{E}_K^{(i)}\}_{i=1}^T$. Then based on a spectral prediction function $f(\cdot)$, the quality of these candidate structures are evaluated by

measuring how close their predicted spectra are to the condition spectrum C. Finally, the optimal generated graph is selected according to $\underset{i}{\operatorname{argmin}} \parallel f(\mathscr{E}_K^{(i)}) - C \parallel_2$.

12.6 Summary

In this chapter, we introduce the definitions and techniques for the transformation problem that involves graphs in the domain of deep graph neural networks. We provide a formal definition of the general deep graph transformation problem as well as its four sub-problems, namely node-level transformation, edge-level transformation,node-edge co-transformation, as well as other graph-involved transformations (e.g., sequence-to-graph transformation and context-to-graph transformation). For each sub-problem, its unique challenges and several representative methods are introduced. As an emerging research domain, there are still many open problems to be solved for future exploration, including but not limited to: (1) **Improved scalability**. Existing deep graph transformation models typically have super-linear time complexity to the number of nodes and cannot scale well to large networks. Consequentially, most existing works merely focus on small graphs, typically with dozens to thousands of nodes. It is difficult for them to handle many real-world networks with millions to billions of nodes, such as the internet of things, biological neuronal networks, and social networks. (2) **Applications in NLP**. As more and more GNN-based works have advanced the development of NLP, graph transformation is naturally a good fit for addressing some NLP tasks, such as information extraction and semantic parsing. For example, information extraction can be formalized into a graph-to-graph problem where the input graph is the dependency graph and the output graph is the information graph. (3) **Explainable graph transformation**. When we learn the underlying distribution of the generated target graphs, learning interpretable representations of graph that expose semantic meaning is very important. For example, it is highly beneficial if we could identify which latent variable(s) control(s) which specific properties (e.g., molecule mass) of the target graphs (e.g., molecules). Thus, investigations on the explainable graph transformation process are critical yet unexplored.

Editor's Notes: Graph transformation is deemed very relevant to graph generation (see Chapter 11) and can be considered as an extension of the latter. In many real-world applications, one is usually required to generate graphs with some condition or control from the users. For example, one may want to generate molecules under some targeted properties (see Chapters 24 and 25) or programs under some function (see Chapter 22). In addition, graph-to-graph transformation also has a connection to link prediction (Chapter 10) and node classification (Chapter 4), though the former could be more challenging since it typically requires simultaneous node-edge prediction, and possibly also comes with the consideration of stochasticity.