Graph to Graph: a Topology Aware Approach for Graph Structures Learning and Generation

Mingming Sun
Cognitive Computing Lab (CCL)
Baidu Research, Haidian Bejing, China
sunmingming01@baidu.com

Ping Li
Cognitive Computing Lab (CCL)
Baidu Research, Bellevue WA, USA
liping11@baidu.com

Abstract

This paper is concerned with the problem of learning the mapping from one graph to another graph. Primarily, we focus on the issue of how to effectively learn the topology of the source graph and then decode it to form the topology of the target graph. We embed the topology of the graph into the states of nodes by exerting a topology constraint, which results in our Topology-Flow encoder. To decoder the encoded topology, we design a conditioned graph generation model with two edge generation options, which result in the Edge-Bernoulli decoder and the Edge-Connect decoder. Experimental results on the 10-nodes simple graph dataset illustrate the substantial progress of the proposed method. The MNIST digits skeleton mapping experiment also reveals the ability of our approach to discover different typologies.

1 Introduction

Learning from complex structures and generating complex structures has long been a "hot" topic in machine learning. Sequence encoders, sequence decoders and the sequence to sequence learning have become standard methodologies for many applications in natural language processing (NLP). Tree structures have also attracted substantial attentions, including the tree encoders (Tai et al., 2015) and tree decoders (Zhou et al., 2018). As a more general task, learning from graphs is a recently emerged field (Battaglia et al., 2018), because graph structures are the natural representations for complex relationships among elements.

Proceedings of the 22[nd] International Conference on Artificial Intelligence and Statistics (AISTATS) 2019, Naha, Okinawa, Japan. PMLR: Volume 89.

Recently, several graph encoders and decoders have already been proposed and applied in some structure mapping frameworks, including the probabilistic graph generation models (You et al., 2018; Li et al., 2018), the sequence-to-graph frameworks (Gildea et al., 2018; Wang et al., 2018, 2016; Peng et al., 2018), and the graph-to-sequence frameworks (Li et al., 2016; Xu et al., 2018; Song et al., 2018). The problem of learning to map a general graph to another graph, however, has not been thoroughly studied. One may think that building a graph-to-graph system is a routine task since one can might be able to build one by assembling a graph encoder and a graph decoder. However, the algorithm does not work as a casual combination. The problem of graph-to-graph learning has its distinct nature, and the encoders and decoders proposed for other tasks may be incompetent for graph-to-graph learning.

In this paper, we propose a general learning framework for mapping one graph to another graph. Primarily, we focus on the problem of how to effectively learn the topology of the source graph and then decode it to form the topology of the target graph. We embed the topology of the graph into the states of nodes by exerting a topology constraint, which results in our Topology-Flow encoder. To decoder the encoded topology, we design a conditioned graph generation model with two edge generation options, which result in the Edge-Bernoulli decoder and the Edge-Connect decoder. Experimental results on the 10-nodes simple graph dataset illustrate the substantial progress of the proposed method. The MNIST digits skeleton mapping experiment also reveals the ability of our approach to discover different typologies.

In the rest of the paper, we first discuss the related work in Section 2 and propose the problem statement of the graph-to-graph learning in Section 3. The proposed encoders and decoder are explained in Section 4 and Section 5, respectively. Section 6 presents the learning algorithm and Section 7 presents the experimental results. We conclude our work in Section 8.

2 Related Work

2.1 Learning from Graphs

Just like learning from sequences, there are two major classes of methodologies for the task of learning from graphs. The first one is the neighborhood/convolution based approach, including Graph Neural Network (GNN) (Scarselli et al., 2009), Graph Convolution Network (GCN) (Johnson, 2017b), Gated Graph Neural Network (GGNN) (Li et al., 2016), Graph-LSTM (Liang et al., 2016; Agrawal et al., 2017), and other graph/network embedding algorithms (Cai et al., 2018). Convolution-based methods study the structures of the neighborhoods of nodes on the graph and generate an annotation for each node as the representation of the neighborhood centered at the node. The convolution operator can be stacked so the neighborhood information can be spread away. Due to the neighborhood sensitive nature, the convolution-based graph learning strategy might be suitable for many neighborhood sensitive tasks, such as semi-supervised learning, link prediction, node classification, and so on.

Another class of algorithms adopt the recurrent based approach, including the DAG-LSTM (Zhu et al., 2016; Chen et al., 2017) and Document Graph-LSTM (Peng et al., 2017). Instead of focusing on using neighborhood information, recurrent based approach pays more attention to the long-term dependencies between nodes, which are important in many NLP applications. This approach has been successfully applied in tasks such as word segmentation (Chen et al., 2017), relation extraction (Peng et al., 2017) and sentiment composition (Zhu et al., 2016). However, these methods are mostly proposed for specific NLP tasks with problem dependent designs. How to design a general-purpose recurrent learning approach is still an open problem.

The topology of a graph represents the long-term dependency between nodes. For our purpose, we believe it is suitable to follow the recurrent based approach. Following this intuition, in this paper, we propose a recurrent learning method - Energy-Flow for generalized graph learning problem, and a variation - Topology-Flow that focuses more on the topology of the graph.

2.2 Generating Graphs

Graph structure generations have been studied in several application fields. For example, transition based graph generation is a common technique to generate a dependency graph (Gildea et al., 2018; Wang et al., 2018, 2016) or AMR structure (Peng et al., 2018) for an input sentence in NLP; MolGAN (Cao and Kipf, 2018) builds a generative model on molecular graphs. These works implement task-specific structure for graphs, but these structures cannot be applied in general graph generation tasks.

For the general graph generation tasks, (You et al., 2018; Li et al., 2018) learn the probabilistic generation models for a set of graphs, that is, unconditioned generation. Although it is mentioned in (Li et al., 2018) that the proposed generation procedure can be extended into conditioned cases, conditioned general graph generation still needs detailed investigation.

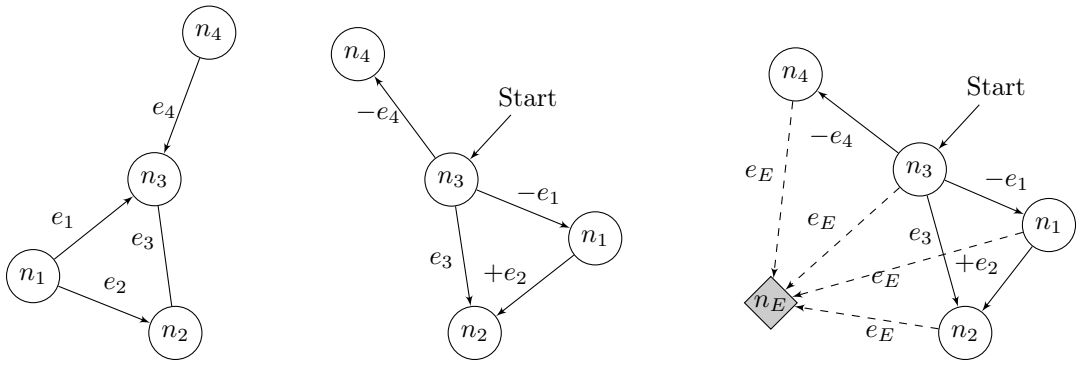
The generation procedure of a target graph differs from task to task. For example, NetGAN (Bojchevski et al., 2018) generates graphs by random-walk and to fine-tune the generation procedure using GAN techniques. But interestingly, we observe that a large number of algorithms follow the same sequence generation procedure (Gildea et al., 2018; Wang et al., 2018, 2016; You et al., 2018; Li et al., 2018), where the process creates one node, and then generates the edges connecting this node with the previously generated nodes. The differences between these algorithms lie in the details of node generation and edge generation.

In this paper, we design a graph decoder conditioned on input graphs. The decoder follows the same sequence generation procedure mentioned above. It learns the attention on input graph annotations, sequentially generates the nodes of target graph, and employs two optional strategies to generate edges: the dependent Bernoulli procedure (You et al., 2018) (results in the Edge-Bernoulli decoder), and the pair of nodes decision (results in the Edge-Connect decoder).

2.3 From Graph to Graph

Given the graph encoder and decoders mentioned above, a variety of sequence-to-graph and graph-to-sequence algorithms have been proposed. Li et al. (2016); Xu et al. (2018); Song et al. (2018) proposed several graph-to-sequence networks to transform graphs into sequences. The applications of graph-to-sequence algorithms include text generation (Beck et al., 2018; Xu et al., 2018; Song et al., 2018), graph algorithms (Li et al., 2016; Song et al., 2018), bAbI tasks (Li et al., 2016; Song et al., 2018), etc. The sequence-to-graph algorithms are generally proposed in the NLP field, including dependency graph generation (Gildea et al., 2018; Wang et al., 2018, 2016) and AMR structure generation (Peng et al., 2018).

The graph-to-graph problem, however, has not been thoroughly exploited in existing literature. You et al. (2018) and Li et al. (2018) built probabilistic generation models from a set of graphs, which can be viewed as a self-to-self map. Johnson (2017a) extended the graph-to-sequence network of Li et al. (2016) to generate a sequence of graph operations to manipulate



(a) A general graph $\mathcal{G}$ (b) DAG $\mathcal{G}_T$ (c) After adding end node $\mathcal{G}_D$

Figure 1: An example to illustrate how to transform arbitrary graphs into DAGs (directed acyclic graphs).

the input graph, so that it can potentially be used to generate another graph. Nevertheless, it has not been tested on graph-to-graph tasks. On the surface, one may think that building a graph-to-graph system is an easy task since there are many graph encoders and decoders available. However, the cooperation between encoder and decoder is a complicated problem and needs further investigation.

In this paper, we study the graph to graph task to learn the topology of source graphs and recover the same or generate another topology to form the target graphs. We select a set of graph encoders and accompany them with our proposed conditioned graph decoder with different edge generation strategy to build several graph to graph algorithms. These algorithms are tested on two topology related tasks, and the experimental results verify the reasonableness of our approach.

3 A DAG Formulation of Graph to Graph Learning

Recall that our goal is to learn a mapping from a source graph to a target graph: $\mathcal{S} \to \mathcal{T}$. We assume that both $\mathcal{S}$ and $\mathcal{T}$ are connected graphs without self-loops or multiple edges. They can be directed or undirected or mixed, with node attributes or edge attributes.

In this paper, we adopt the sequence generation method to generate $\mathcal{T}$, in a one-node-per-step manner. This means it requires a partial order among the nodes in $\mathcal{T}$. DAG (directed acyclic graph) is a natural way to represent such order information. Moreover, to fully exploit the long-term topology relationship among nodes and edges in $\mathcal{S}$, we propose to use recurrent based models to encoder the $\mathcal{S}$, which again needs a partial order on the nodes, i.e., a DAG. Overall speaking, DAG is a suitable structure for both encoder and decoder and is adopted in our proposed algorithm.

As illustrated in Figure 1, we adopt the following procedure to convert a graph $\mathcal{G}$ into a DAG without loss of information:

1. Let $\mathcal{G} =< \mathbb{N}, \mathbb{E} >$, where $\mathbb{N}$ is the set of nodes, and $\mathbb{E}$ is the set of edges. We select a node n_1 and start a breadth-first-traversal over the $\mathcal{G}$ from n_1. The travel history builds a partial order relation $\prec$ between nodes of $\mathcal{G}$, where $n_i \prec n_j$ means n_i is visited earlier than n_j in the travel.

2. We define the travel graph of $\mathcal{G}$ as a DAG $\mathcal{G}_T =< \mathbb{N}, \mathbb{E}_T >$, where $\mathbb{E}_T = \{< n_i, n_j, \widetilde{e_{ij}} > | n_i \prec n_j, (n_i, n_j, e_{ij}) \in \mathbb{E}\}$, where $\widetilde{e_{ij}}$ is an edge connecting n_i to n_j with transformed attributes, which add extra information about the traversal order on the original e_{ij} so that we can recover the $\mathcal{G}$ from $\mathcal{G}_T$. For example, if e_{ij} is a directed edge from n_j to n_i with type t, then we introduce a new type $-t$ and assign it to $\widetilde{e_{ij}}$.

3. We add an end node n_{END} and connect all nodes in N to n_{END} with an edge e_{END} with a special type T_{END}. The resulting graph is denoted by $\mathcal{G}_D =< \mathbb{N}_D, \mathbb{E}_D >$, where $\mathbb{N}_D = \mathbb{N} \cup \{n_{\text{END}}\}$, $\mathbb{E}_D = \mathbb{E}_T \cup \{\cup_{i=1}^{N} < n_i, n_{\text{END}}, e_{\text{END}} > | n_i \in \mathbb{N}\}$.

It is easy to check that with suitable transformation on edge attributes, there is no information loss during the procedure and we can recover $\mathcal{G}$ from $\mathcal{G}_D$. For the decoder, given a target graph $\mathcal{T}$, we build the corresponding DAG $\mathcal{T}_D$ as our learning target. For the encoder, given a source graph $\mathcal{S}$, by selecting different starting node multiple times, we obtain several DAGs $\mathcal{S}_D^1, \cdots, \mathcal{S}_D^K$ as multiple views of the graph $\mathcal{S}$. In this way, our formulation of the graph-to-graph learning boils down to finding the mapping from several source DAGs to a target DAG:

$$\mathfrak{M} : \mathcal{S}_D^1, \cdots, \mathcal{S}_D^K \to \mathcal{T}_D.$$

Algorithm 1 Energy-Flow Encoder for DAG

Require:
A DAG $\mathcal{G}$;
An initial energy vector s_0;
An energy flow function $\mathfrak{E}$;
An representation function $\mathfrak{R}$ to generate representation for nodes and edges;

Start

1: Compute the energy vector for starting node n_1: $s_1 = \mathfrak{E}(s_0, [\mathfrak{R}(n_1)])$;
2: **for** each node n_j in $\mathcal{G}$ **do**
3: Compute the energy vector for node n_j:

$$s_j = \frac{1}{|\mathbb{P}(n_j)|} \sum_{n_i \in \mathbb{P}(n_j)} \mathfrak{E}(\frac{s_i}{|\mathbb{C}(n_i)|}, [\mathfrak{R}(e_{ij}), \mathfrak{R}(n_j)]);$$

4: **end for**
5: **return** $s_1, \cdots, s_{\text{END}}$

Algorithm 2 Topology-Flow Encoder for DAG

Require:
A DAG $\mathcal{G}$;
An initial energy vector s_0;
An energy flow function $\mathfrak{E}$;
An representation function $\mathfrak{R}$ to generate representation for nodes and edges;

Start

1: Compute the energy vector for starting node n_1: $s_1 = \mathfrak{E}(s_0, [\mathfrak{R}(n_1)])$;
2: **for** each node n_j in $\mathcal{G}$ **do**
3: Compute the energy vector for node n_j:

$$\widetilde{s_j} = \frac{1}{|\mathbb{P}(n_j)|} \sum_{n_i \in \mathbb{P}(n_j)} \mathfrak{E}(\frac{s_i}{|\mathbb{C}(n_i)|}, [\mathfrak{R}(e_{ij}), \mathfrak{R}(n_j)]);$$

$$s_j = \text{Transform}(\widetilde{s_j} - \text{Project}(\widetilde{s_j}, [s_k, n_k \in \mathbb{D}(n_j)]));$$

4: **end for**
5: **return** $s_1, \cdots, s_{\text{END}}$

4 Recurrent Encoder for DAG

The encoder for multiple DAGs can be implemented by applying the encoder to every single DAG and then concatenate the annotations of each node. It is hence sufficient that we focus on the encoder for a single DAG. To help express the algorithm, we introduce three operators on graphs:

1. $\mathbb{P}(n_j) = \{n_i | < n_i, n_j, e_{ij} > \in \mathbb{E}_D, n_i \prec n_j\}$: the set of parent nodes of node n_j;
2. $\mathbb{C}(n_i) = \{n_j | < n_i, n_j, e_{ij} > \in \mathbb{E}_D, n_i \prec n_j\}$: the set of child nodes of node n_i;
3. $\mathbb{D}(n_j) = \{n_i | n_i \notin \mathbb{P}(n_j), n_i \prec n_j\}$: the set of nodes which are not the parents of node n_j.

4.1 Energy-Flow Encoder

We first describe our proposed Energy-Flow encoder, which is summarized in Algorithm 1:

1. We initialize the energy of each node in the DAG as zero.
2. We place a high-energy point at the starting point n_1. Then the energy would flow to the place with lower energy, that is, other nodes.
3. During the flow procedure, each node gathers the energy from its parents, and then distributes the energy equally to its children.
4. Lastly, the end node n_{END} gathers all the flowed energy and terminates the procedure.

4.2 Topology-Flow Encoder

The above Energy-Flow encoder algorithm exploits the information about the direct connection between nodes (via the parent relationship). It is thus aware of any part of the topology of the graph. However, another essential topology information – the non-connection relationship, is not emphasized during the procedure. We propose a Topology Aware Energy Flow encoder (simplified as the Topology-Flow encoder) to encode such information.

Algorithm 2 summarizes the procedure, where $\text{Project}(v, X)$ is the projected vector of the vector v on the space spanned by the set of vectors X, and the $\text{Transform}(s)$ is a transformed vector of s, which could be linear or nonlinear.

The Topology-Flow encoder differs from the Energy-Flow encoder with an additional operation:

$$s_j = \text{Transform}(\widetilde{s_j} - \text{Project}(\widetilde{s_j}, [s_k, n_k \in \mathbb{D}(n_j)])).$$

In this operation, let $v_j = \widetilde{s_j} - \text{Project}(\widetilde{s_j}, [s_k, n_k \in \mathbb{D}(n_j)])$, then v_j will reside in the orthogonal complement space spanned by the states of nodes in $\mathbb{D}(n_j)$, that is, $v_j \cdot s_k = 0, \forall n_k \in \mathbb{D}(n_j)$. It is a strong signal to force the encoder to memorize the topology information. The v_j can be computed as follows:

$$v_j = (I - S_j S_j^+)\widetilde{s_j}$$

where $S_j = [s_k, n_k \in \mathbb{D}(n_j)]$ is the matrix with columns of states of nodes in $\mathbb{D}(n_j)$, and the $S_j^+$is the Moore–Penrose inverse of S_j.

4.3 Implementation

There are four customizable components in the proposed encoders: the representation function $\mathfrak{R}$, the energy flow function $\mathfrak{E}$, the initial states s_0 and the Transform function. We will elaborate on them all.

4.3.1 Representation Function $\mathfrak{R}$

The representation function $\mathfrak{R}$ generates representation vectors for nodes and edges. A simple implementation is to combine the features of a node or edge to form a vector (for categorical feature, embedding or one-hot vector may be used). For nodes, it can also be a graph encoder that generates node annotations, e.g., graph convolution network or Energy/Topology Flow network. The overall encoder in this case becomes a multi-layer stacked neural network.

4.3.2 Energy-Flow Function $\mathfrak{E}$

The energy flow function $\mathfrak{E}$ compute the contribution of energy from a parent node n_i (with energy s_i) to the node n_j through an edge e_{ij}. We use a gated RNN function (GRU or LSTM) to implement $\mathfrak{E}$. It takes s_i as the initial states, and process the input sequence of $[\mathfrak{R}(e_{ij}), \mathfrak{R}(n_j)]$. The output state vector will be taken as the energy contribution from node n_i to n_j.

4.3.3 Initial States

For the initial states s_0, one straightforward option is to set it to a zero vector. It is also possible to use a representation vector to represent the global characteristic of the input graph, which could be generated by a global graph convolutional network.

4.3.4 Transform Function

The Transform function introduced in the Topology-Flow encoder can be either linear or nonlinear mapping. A linear mapping is used in this study.

4.4 Encode the Starting Point Information

Since our DAG formulation is dependent on the selection of the starting node, it would be better to encode the information about which node is selected as the starting node. Although the encoder algorithm may be able to learn such information, when the starting node of the source graph is explicitly related to the starting node of the target graph, it would be better directly tell the decode which node is the starting node. The straightforward approach is to append a binary variable to the obtained representations of each node, while for the starting node, the binary variable is set to 1, and for other nodes, it is set to 0.

Algorithm 3 Topology-Flow Decoder for DAG

Require:
A set of annotation vectors $A = [s_1^{\mathcal{S}}, \cdots, s_{\text{END}}^{\mathcal{S}}]$.
A start node state model StartStateModel;
A node state model NodeStateModel;
A node generation model NodeGenModel;
An edge generation model EdgeGenModel;

Start

Compute the state of starting node s_1 and the starting node n_1:

$$s_1 = \text{StartStateModel}(A);$$
$$n_1 = \arg\max p(n_1|A) = \arg\max \text{NodeGenModel}(s_1, A)$$

Build the graph $\mathcal{T}_1 =< n_1, \{\} >$;
do
  Compute the state for new node :

$$s_k = \text{NodeStateModel}(\mathcal{T}_{k-1}, A);$$

  Compute the new node n_k:

$$n_k = \arg\max p(n_k|\mathcal{T}_{k-1}, A) = \arg\max \text{NodeGenModel}(s_k, A);$$

  Add n_k into $\mathcal{T}_{k-1}$ and obtain $\mathcal{T}_k$
  for each node n_j in $\mathcal{T}_{k-1}$ **do**:
    Compute the new edge e_{jk}:

$$e_{jk} = \arg\max p(e_{jk}| < n_j, n_k >) = \arg\max \text{EdgeGenModel}(s_j, s_k);$$

    if e_{jk} is not empty edge **then**
      Add e_{jk} into $\mathcal{T}_k$ to connect n_j and n_k;
    end if
  end for
while $n_k \neq n_{\text{END}}$
return $\mathcal{T}_k$

5 Recurrent Decoder for Graphs

Our decoder follows the sequence generation procedure for graphs. The procedure tarts with an initial empty graph $\mathcal{T}_0$, and then at each step, add a node n_k and corresponding edges $\{e_{kj}\}$ into the graph $\mathcal{T}_{k-1}$ to form a new graph $\mathcal{T}_k$, until n_{END} is added to the graph, or the number of nodes exceeds a predefined threshold. Following the procedure, we develop a family of recurrent decoders, which is presented in Algorithm 3.

5.1 StartStateModel

The StartStateModel is used to generate suitable states for Node Generation Model to choose the correct starting point. The information about starting

point is either explicitly encoded in the nodes annotations (as stated in Section 4.4) or could be determined by the global information of graph, that is, the statistic character of the set of nodes annotations. In either case, the state of the starting nodes can be computed as an aggravated transformation of the node annotations. Generally, we use the following transformation:

$$s_0 = \text{StartStateModel}(A) = \tanh(W^T \cdot \frac{1}{|A|} \sum s_i^{\mathcal{S}}).$$

5.2 NodeStateModel

The Node State Model is to compute the state of the new node to be added to the graph to guide the Node Generation Model to generate the correct node. Intuitively, the state of new node s_i is related to the generated graph $\mathcal{T}_{i-1}$ and the annotations from the source graph A:

$$s_i = \text{NodeStateModel}(\mathcal{T}_{i-1}, A).$$

In this paper, we assume the state of the graph $\mathcal{T}_{i-1}$ can be represented in the state s_{i-1} of node n_{i-1}, so $s_i = \text{NodeStateModel}(s_{i-1}, A)$. This can be done by the attention-based state update scheme:

1. Compute the context on source annotations: $c_i = \text{AttentionContext}(s_{i-1}, A)$;
2. Compute the state of the node n_i by $s_i = \text{ContextRecurrent}(s_{i-1}, \mathfrak{R}(n_{i-1}), c_i)$, where the recurrent unit can be LSTM or GRU.

The details of the above computations can be found in (Bahdanau et al., 2014).

5.3 NodeGenerationModel

In our DAG formulation as described in Section 3, the source graph and the target graph both contain at least one type of node n_{END}. Other nodes in the source graph could be of totally different types from the nodes in the target graph, although it is also possible that they share same types. In this situation, the copy mechanism may be helpful to reduce the difficulty of the learning problem (Gu et al., 2016), since a target node s_i can be either selected from the vocabulary V for target graphs, or copied from the source graph $\mathcal{S}$. So the Node Generation Model is:

$$\begin{aligned} p(n_i|\mathcal{T}_{i-1}, A) &= \text{NodeGenModel}(s_{i-1}, A), \\ &= p_{\mathcal{S}}(n_i|s_i) + p_V(n_i|s_i) \end{aligned}$$

where $p_{\mathcal{S}}$ is the probability of being copied from the source graph $\mathcal{S}$, and p_V is the probability of being selected from the vocabulary V. The details of the copy mechanism can be find in (Gu et al., 2016).

5.4 EdgeGenerationModel

People may have different view point for edge generation process. In GraphRNN algorithm, the generation of edges e_{ij} for node n_j is modeled as a Dependent Bernoulli process and implemented as a gated recurrent sequence, with initial state as the state of node n_j and input as the previous generated edge $e_{i-1,j}$:

$$\begin{aligned} p(e_{ij}| < n_j, n_i >) &= p(e_{ij}|n_i) \\ &= \text{DependentBernoulli}(s_i) \end{aligned}$$

Note that the procedure does not directly depend on the state of node n_i. We name the graph decoder using this edge generation model as Edge-Bernoulli Decoder.

Another viewpoint is that the edge is decided by the states of nodes whom it connects. So the probability of an edge e_{ij} connecting the newly generated node n_i and any other node n_j in $\mathcal{T}_{i-1}$ is determined by the states s_i and s_j:

$$\begin{aligned} p(e_{ij}| < n_j, n_i >) &= \text{EdgeGenModel}(s_i, s_j) \\ &= \text{Softmax}(f(s_k \| s_j)), \end{aligned}$$

where f is a transform function. In this paper, f is a fully-connected three-layer RELU network. We name the graph decoder using this edge generation model as Edge-Connect Decoder.

6 Learning

Given a source graph $\mathcal{S}$, the encoder-decoder procedure essentially builds a probability distribution over the target graph space. We need to optimize the encoder and decoder to maximize the conditional probability of generating the target graph $\mathcal{T}$. In the generation procedure of the Topology-Flow decoder, the probability of $\mathcal{T}$ given $\mathcal{S}$ can be written as:

$$\begin{aligned} &p(\mathcal{T}|\mathcal{S}) \\ &= \prod_i p(n_i|\mathcal{T}_{i-1}, \mathcal{S}) \prod_{n_j \in \mathcal{T}_{i-1}} p(e_{ji}| < n_j, n_i >, \mathcal{T}_{i-1}, \mathcal{S}) \end{aligned}$$

The learning objective of our graph-to-graph learning is the negative likelihood:

$$\text{NegLikelyHood}(\mathcal{T}|\mathcal{S}) = -\log p(\mathcal{T}|\mathcal{S}).$$

7 Experimental Results

We use two topology learning tasks to test the performance of the proposed method. The first one is the graph copy task, which examines the ability of algorithms to copy a random graph, that is, to honestly memorize the topology. The second one is the task

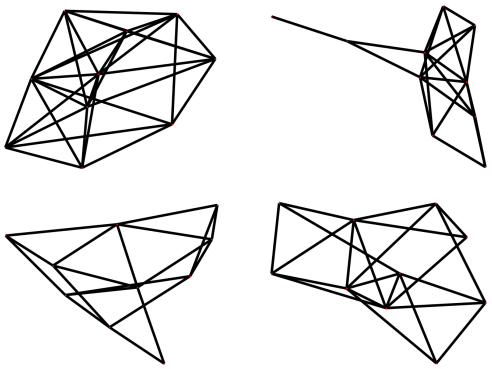


Figure 2: Examples of simple graph dataset.

of learning the mapping from a skeleton graph of a digit to that of another digit, which tests the ability of the decoder to recover the topology, on the basis of honestly memorizing the source topology.

7.1 Graph Copy Task

The Graph Copy task is to train a machine that can honestly copy a graph. In this experiment, we use a dataset containing simple graphs with ten nodes and all possible topology, provided by `https://users.cecs.anu.edu.au/~bdm/data/graphs.html`. This dataset includes 11,716,571 samples, i.e., there is the same number of types of topology for simple graphs with ten nodes. Several examples of these sparse graphs are shown in Figure 2. Using this dataset allows us to thoroughly examine the capability of different algorithms to learn the large variety of topology.

7.1.1 Experimental Setting

We randomly select 50,000 samples as the training set, 1,000 sample as the validating set and 1,000 samples as the testing set. For each graph, we select the node with maximum degree as the start node to build the DAGs. We test the performances of eight combinations of four encoders (GGNN, GCN, Energy-Flow, and Topology-Flow) and two decoders (Edge-Bernoulli and Edge-Connect). In the experiments, we use one layer network for Topology-Flow and Energy-Flow encoders, two layer network for GCN and GGNN, and one-layer network for all decoders. In all experiments, we set the hidden dimension to be 256.

7.1.2 Evaluation Criterion

We test the performances of these algorithms by checking whether the generated graph is isomorphic with the target graph and report the graph-level accuracy (denoted by the "G-ACC"). To investigate the difference between the graph-level accuracy, we also compute the precisions and recalls on the set of edges and report the F1 scores (denoted by the "Edge-F1").

Table 1: Performance results on the graph-copy task.

Encoder	Decoder	G-Acc	Edge-F1
GCN	Edge-Bernoulli	$< 1\%$	76.8%
GCN	Edge-Connect	$< 1\%$	79.2%
GGNN	Edge-Bernoulli	$< 1\%$	54.9%
GGNN	Edge-Connect	$< 1\%$	88.9%
Energy-Flow	Edge-Bernoulli	$< 1\%$	89.6%
Energy-Flow	Edge-Connect	$< 1\%$	90.4%
Topology-Flow	Edge-Bernoulli	87.1%	99.5%
Topology-Flow	Edge-Connect	98.3%	99.9%

7.1.3 Experimental Results

The experimental results are shown in Table 1. We can see that the algorithm with the Topology-Flow encoder and Edge-Connect decoder achieves substantial higher graph-level accuracy than any other algorithm. The reason is revealed by the F1 scores on edges: the algorithms with encoders other than Topology-Flow have a lot of error on edge prediction, which confirm the effectiveness of the explicit topology constraint in the Topology-Flow encoder. We also observe that the performance of the Edge-Connect decoder is better than that of the Edge-Bernoulli decoder, which is expected. Since the Topology-Flow encoder encodes the topology information into pair of nodes, the decoder should consider the information about pairs of nodes to generate edges just as in Edge-Connect, while the Edge-Bernoulli decoder does not.

7.2 MNIST Digit Skeleton Mapping

In this task, we learn the skeleton mapping between different digits. We use the MNIST sequence data (`https://edwin-de-jong.github.io/blog/mnist-sequence-data/`) and transform the sequences into skeleton graphs by connecting neighborhood points and down sampling. Example skeleton graphs are shown in Figure 3.

Then we build pairs of different digits, and random sampling several skeleton graphs for each digit to build a graph mapping dataset. In this paper, we choose the pairs from neighboring digits, that is, <0, 1>, <1, 2>, <2, 3>, <3, 4> $\cdots$, <9, 0> . The generated dataset includes 34,084 pairs of skeleton graphs for training, 8,528 pairs for validating, and 7,192 pairs for testing.

Using this dataset, we would like to evaluate the ability of various algorithms to learn from a graph and to generate another different graph.

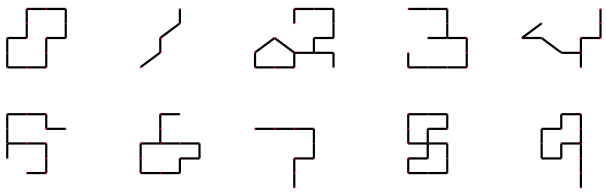


Figure 3: Examples of skeleton graphs of digits from MNIST dataset, from 0-9.

7.2.1 Experimental Setting

The coordinates of nodes of the skeleton graphs are normalized to $[0, 7)$ in both directions. For each graph, we select the node with maximum degree as the start node to build the DAGs. We make the coordinates of nodes as the attribute of the nodes, and obtain the representations of the nodes by a randomly initialized embedding method with the capacity of 50 (all type of positions and one extra n_{END}). The representations of edges are obtained by a randomly initialized embedding method with the capacity of 2 (a normal edge and e_{END}).

Again, we test the performances of eight combinations of four encoders (GGNN, GCN, Energy-Flow, and Topology-Flow) and two decoders (Edge-Bernoulli and Edge-Connect). In the experiment, decoders of all models are with one-layer network. Topology-Flow and Energy-Flow encoders uses one-layer network, while GCN and GGNN uses three-layer nested network. In all experiments, we set the hidden dimension to be 256. The maximum number of training epochs is set to be 500.

7.2.2 Evaluation Criterion

We evaluate the quality of the generated skeletons by examining whether they still are the skeletons of the target digits. It means that we need a skeleton graph classifier to classify a skeleton graph to a digit. One approach is to generate an image from the skeleton graph and use the MNIST digit classifier. However, we can see from Figure 3 that the skeleton graph is significantly different from the normal digit image, which means that the MNIST digit classifiers may not work well. Furthermore, since this work is about the graph learning, we would instead develop a skeleton graph classifier using the graph learning techniques.

Our skeleton graph classifier first employs a graph encoder to generate the annotation for each node, and averages these annotations to obtain a global representation of the graph. Then the representation vector is sent to a three-layer fully connected RELU network to conduct a 10-class classification task. The hidden dim is set as 256. Because the convolution-based graph learning algorithm works well in extracting global graph features, we use the 2-layer GGNN encoder in the classifier. The dataset used to train the classifier contains 39,560 graphs for training, 9,890 graphs for validation, and 8,325 graphs for testing. The performance of the trained classifier on the testing set is 95.6%.

7.2.3 Experimental Results

The performance of all the combinations of encoders and decoders is summarized in Table 2. We can see observe that the Topology-Flow encoder works well with the Edge-Bernoulli decoder, which learns the map between skeletons of digits with high accuracy. The Energy-Flow encoder and the GGNN encoder also perform well, while the GCN encoder fails completely. We also observe that the Edge-Bernoulli decoder generally works better than Edge-Connect decoder for this task (expect when working with the Energy-Flow encoder). Our explanation is that for the simple skeleton topologies, the Edge-Bernoulli decoder has enough capacity, and its simplified connection to nodes helps for better generalization.

Table 2: Performance results on the MNIST digit skeleton mapping task.

Encoder	Decoder	Accuracy
GCN	Edge-Bernoulli	64.09%
GCN	Edge-Connect	54.85%
GGNN	Edge-Bernoulli	81.27%
GGNN	Edge-Connect	77.15%
Energy-Flow	Edge-Bernoulli	54.71%
Energy-Flow	Edge-Connect	81.24%
Topology-Flow	Edge-Bernoulli	82.31%
Topology-Flow	Edge-Connect	79.28%

8 Conclusion

In this paper, we study the problem of learning the mapping from one graph to another graph. More specifically, we focus on learning the topology of the source graph and transforming it to the topology of the target graph. Our preliminary experimental results on two common topology learning tasks verify the ability of the proposed approach.

This work can be extended in several directions. Firstly, we would like to investigate algorithms to make the algorithm more efficient for large scale graphs. Secondly, we would like to apply the proposed graph-to-graph algorithm to piratical tasks, such as the graph-structured Open Information Extraction (Sun et al., 2018b,a) when the relations between facts are considered.

References

Rakshit Agrawal, Luca de Alfaro, and Vassilis Polychronopoulos. Learning from graph neighborhoods using lstms. In *The Workshops of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI Workshop)*, San Francisco, California, 2017.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014. URL `http://arxiv.org/abs/1409.0473`.

Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinícius Flores Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Çaglar Gülçehre, Francis Song, Andrew J. Ballard, Justin Gilmer, George E. Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matthew Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. *CoRR*, abs/1806.01261, 2018. URL `http://arxiv.org/abs/1806.01261`.

Daniel Beck, Gholamreza Haffari, and Trevor Cohn. Graph-to-sequence learning using gated graph neural networks. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 273–283, Melbourne, Australia, 2018.

Aleksandar Bojchevski, Oleksandr Shchur, Daniel Zügner, and Stephan Günnemann. Netgan: Generating graphs via random walks. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, pages 609–618, Stockholmsmässan, Stockholm, Sweden, 2018.

HongYun Cai, Vincent W. Zheng, and Kevin Chen-Chuan Chang. A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE Trans. Knowl. Data Eng.*, 30(9):1616–1637, 2018.

Nicola De Cao and Thomas Kipf. Molgan: An implicit generative model for small molecular graphs. *CoRR*, abs/1805.11973, 2018. URL `http://arxiv.org/abs/1805.11973`.

Xinchi Chen, Zhan Shi, Xipeng Qiu, and Xuanjing Huang. Dag-based long short-term memory for neural word segmentation. *CoRR*, abs/1707.00248, 2017.

Daniel Gildea, Giorgio Satta, and Xiaochang Peng. Cache transition systems for graph parsing. *Computational Linguistics*, 44(1), 2018.

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O. K. Li. Incorporating copying mechanism in sequence-to-sequence learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, page 1631–1640, Berlin, Germany, 2016.

Daniel D Johnson. Learning Graphical State Transitions. In *the International Conference on Learning Representations (ICLR)*, Neptune, Toulon, France, 2017a.

Daniel D Johnson. Semi-supervised classification with graph convolutional networks. In *the International Conference on Learning Representations (ICLR)*, Neptune, Toulon, France, 2017b.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated Graph Sequence Neural Networks. In *International Conference on Learning Representations (ICLR)*, San Juan, Puerto Rico, 2016.

Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. Learning deep generative models of graphs. *CoRR*, abs/1803.03324, 2018. URL `http://arxiv.org/abs/1803.03324`.

Xiaodan Liang, Xiaohui Shen, Jiashi Feng, Liang Lin, and Shuicheng Yan. Semantic object parsing with graph LSTM. In *Proceedings of the 14th European Conference on Computer Vision (ECCV)*, pages 125–143, Amsterdam, The Netherlands, 2016.

Nanyun Peng, Hoifung Poon, Chris Quirk, Kristina Toutanova, and Wen-tau Yih. Cross-sentence n-ary relation extraction with graph lstms. *Transactions of the Association for Computational Linguistics*, 5: 101–115, 2017.

Xiaochang Peng, Daniel Gildea, and Giorgio Satta. AMR parsing with cache transition systems. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI)*, pages 4897–4904, New Orleans, Louisiana, 2018.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Trans. Neural Networks*, 20(1):61–80, 2009.

Linfeng Song, Yue Zhang, Zhiguo Wang, and Daniel Gildea. A graph-to-sequence model for amr-to-text generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1616–1626, Melbourne, Australia, 2018.

Mingming Sun, Xu Li, and Ping Li. Logician and orator: Learning from the duality between language and knowledge in open domain. In *Proceedings of the 2018 Conference on Empirical Methods in*

Natura (EMNLP), pages 2119–2130, Brussels, Belgium, 2018a.

Mingming Sun, Xu Li, Xin Wang, Miao Fan, Yue Feng, and Ping Li. Logician: A Unified End-to-End Neural Approach for Open-Domain Information Extraction. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining (WSDM)*, pages 556–564, Los Angeles, CA, 2018b.

Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1556–1566, Beijing, China, 2015.

Yuxuan Wang, Jiang Guo, Wanxiang Che, and Ting Liu. Transition-based chinese semantic dependency graph parsing. In *International Symposium on Natural Language Processing Based on Naturally Annotated Big Data (NLP NABD)*, pages 12–24, Yantai, China, 2016.

Yuxuan Wang, Wanxiang Che, Jiang Guo, and Ting Liu. A neural transition-based approach for semantic dependency graph parsing. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI)*, pages 5561–5568, New Orleans, Louisiana, 2018.

Kun Xu, Lingfei Wu, Zhiguo Wang, Yansong Feng, and Vadim Sheinin. Graph2seq: Graph to sequence learning with attention-based neural networks. *CoRR*, abs/1804.00823, 2018. URL `http://arxiv.org/abs/1804.00823`.

Jiaxuan You, Rex Ying, Xiang Ren, William L. Hamilton, and Jure Leskovec. Graphrnn: Generating realistic graphs with deep auto-regressive models. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, pages 5694–5703, Stockholmsmässan, Stockholm, Sweden, 2018.

Ganbin Zhou, Ping Luo, Rongyu Cao, Yijun Xiao, Fen Lin, Bo Chen, and Qing He. Tree-structured neural machine for linguistics-aware sentence generation. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI)*, pages 5722–5729, New Orleans, Louisiana, 2018.

Xiao-Dan Zhu, Parinaz Sobhani, and Hongyu Guo. Dag-structured long short-term memory for semantic compositionality. In *Proceedings of the 15th Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL HLT)*, pages 917–926, San Diego, California, 2016.