SQL-to-Text Generation with Graph-to-Sequence Model

Kun Xu[1]*, **Lingfei Wu**[2], **Zhiguo Wang**[2], **Yansong Feng**[3], **Vadim Sheinin**[2]
[1]Tencent AI Lab
[2]IBM Research
[3]Peking University, Beijing, China
{syxu828,zgw.tomorrow}@gmail.com, lwu@email.wm.edu
fengyansong@pku.edu.cn, vadims@us.ibm.com

Abstract

Previous work approaches the SQL-to-text generation task using vanilla Seq2Seq models, which may not fully capture the inherent graph-structured information in SQL query. In this paper, we first introduce a strategy to represent the SQL query as a directed graph and then employ a graph-to-sequence model to encode the global structure information into node embeddings. This model can effectively learn the correlation between the SQL query pattern and its interpretation. Experimental results on the WikiSQL dataset and Stackoverflow dataset show that our model significantly outperforms the Seq2Seq and Tree2Seq baselines, achieving the state-of-the-art performance.

1 Introduction

The goal of the SQL-to-text task is to automatically generate human-like descriptions interpreting the meaning of a given structured query language (SQL) query (Figure 1 gives an example). This task is critical to the natural language interface to a database since it helps non-expert users to understand the esoteric SQL queries that are used to retrieve the answers through the question-answering process (Simitsis and Ioannidis, 2009) using varous text embeddings techniques (Kim, 2014; Arora et al., 2017; Wu et al., 2018a).

Earlier attempts for SQL-to-text task are rule-based and template-based (Koutrika et al., 2010; Ngonga Ngomo et al., 2013). Despite requiring intensive human efforts to design temples or rules, these approaches still tend to generate rigid and stylized language that lacks the natural text of the human language. To address this, Iyer et al. (2016) proposes a sequence-to-sequence (Seq2Seq) network to model the SQL query and natural language jointly. However, since the SQL is designed

(SQL): **SELECT** company **WHERE** assets > val_0 **AND** sales > val_0 **AND** industry_rank <= val_2 **AND** revenue = val_3

Interpretation:
which company has both the market value and assets higher than val_0, ranking in top val_2 and revenue of val_3

Figure 1: An example of SQL query and its interpretation.

to express graph-structured query intent, the sequence encoder may need an elaborate design to fully capture the global structure information. Intuitively, varous graph encoding techniques base on deep neural network (Kipf and Welling, 2016; Hamilton et al., 2017; Song et al., 2018) or based on Graph Kernels (Vishwanathan et al., 2010; Wu et al., 2018b), whose goal is to learn the node-level or graph-level representations for a given graph, are more proper to tackle this problem.

In this paper, we first introduce a strategy to represent the SQL query as a directed graph (see §2) and further make full use of a novel graph-to-sequence (Graph2Seq) model (Xu et al., 2018) that encodes this graph-structured SQL query, and then decodes its interpretation (see §3). On the encoder side, we extend the graph encoding work of Hamilton et al. (2017) by encoding the edge direction information into the node embedding. Our encoder learns the representation of each node by aggregating information from its K-hop neighbors. Different from Hamilton et al. (2017) which neglects the edge direction, we classify the neighbors of a node according to the edge direction, say v, into two classes, i.e., forward nodes (v directs to) and backward nodes (direct to v). We apply two distinct aggregators to aggregate the information of these two types of nodes, resulting two representations. The node embedding of v is the concatenation of these two representations. Given the learned node embeddings, we further introduce a pooling-based and an aggregation-based method to generate the graph embedding.

* Work done when the author was at IBM Research.

Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, pages 931–936
Brussels, Belgium, October 31 - November 4, 2018.

On the decoder side, we develop an RNN-based decoder which takes the graph vector representation as the initial hidden state to generate the sequences while employing an attention mechanism over all node embeddings. Experimental results show that our model achieves the state-of-the-art performance on the WikiSQL dataset and Stackoverflow dataset. Our code and data is available at https://github.com/IBM/SQL-to-Text.

2 Graph Representation of SQL Query

Representing the SQL query as a graph instead of a sequence could better preserve the inherent structure information in the query. An example is illustrated in the blue dashed frame in Figure 2. One can see that representing them as a graph instead of a sequence could help the model to better learn the correlation between this graph pattern and the interpretation "*...both X and Y higher than Z...*". This observation motivates us to represent the SQL query as a graph. In particular, we use the following method to transform the SQL query to a graph:[1]

SELECT Clause. For the SELECT clause such as "SELECT company", we first create a node assigned with text attribute *select*. This SELECT node connects with column nodes whose text attributes are the selected column names such as *company*. For SQL queries that contain aggregation functions such as `count` or `max`, we add one aggregation node which is connected with column nodes. Similarly, their text attributes are the aggregation function names.

WHERE Clause. The WHERE clause usually contains more than one condition. For each condition, we use the same process as for the SELECT clause to create nodes. For example, in Figure 2, we create node *assets* and $>val_0$ for the first condition, the node *sales* and $>val_0$ for the second condition. We then integrate the constraint nodes that have the same text attribute (e.g., $>val_0$ in Figure 2). For a logical operator such as AND, OR and NOT, we create a node that connects with all column nodes that the operator works on. Finally, these logical operator nodes connect with the SELECT node.

[1]This method could be simply extended to cope with more general SQL queries that have complex syntaxes such as JOIN and ORDER BY.

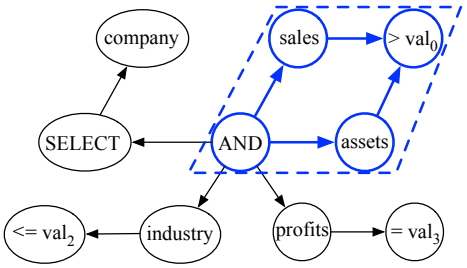

Figure 2: The graph representation of the SQL query in Figure 1.

3 Graph-to-sequence Model

Based on the constructed graphs for the SQL queries, we make full use of a novel graph-to-sequence model (Xu et al., 2018), which consists of a graph encoder to learn the embedding for the graph-structured SQL query, and a sequence decoder with attention mechanism to generate sentences. Conceptually, the graph encoder generates the node embedding for each node by accumulating information from its K-hop neighbors, and produces a graph embedding for the entire graph by abstracting all node embeddings. Our decoder takes the graph embedding as the initial hidden state and calculates the attention over all node embeddings on the encoder side to generate natural language interpretations.

Node Embedding. Given the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, since the text attribute of a node may include a list of words, we first use a Long Short Term Memory (LSTM) to generate the feature vector $\mathbf{a}_v$ for all nodes $\forall v \in \mathcal{V}$ from v's text attribute. We use these feature vectors as initial node embeddings. Then, our model incorporates information from a node's neighbors within K hop into its representation by repeating the following process K times:

$$\mathbf{h}^0_{v\vdash} = \mathbf{a}_v, \mathbf{h}^0_{\vdash v} = \mathbf{a}_v, \forall v \in \mathcal{V} \quad (1)$$

$$\mathbf{h}^k_{\mathcal{N}_\vdash(v)} = \mathbf{M}^k_\vdash(\{\mathbf{h}^{k-1}_{u\vdash}, \forall u \in \mathcal{N}_\vdash(v)\}) \quad (2)$$

$$\mathbf{h}^k_{v\vdash} = \sigma(\mathbf{W}^k \cdot \texttt{CONCAT}(\mathbf{h}^{k-1}_{v\vdash}, \mathbf{h}^k_{\mathcal{N}_\vdash(v)})) \quad (3)$$

$$\mathbf{h}^k_{\mathcal{N}_\dashv(v)} = \mathbf{M}^k_\dashv(\{\mathbf{h}^{k-1}_{u\dashv}, \forall u \in \mathcal{N}_\dashv(v)\}) \quad (4)$$

$$\mathbf{h}^k_{v\dashv} = \sigma(\mathbf{W}^k \cdot \texttt{CONCAT}(\mathbf{h}^{k-1}_{v\dashv}, \mathbf{h}^k_{\mathcal{N}_\dashv(v)})) \quad (5)$$

where $k \in \{1, ..., K\}$ is the iteration index, $\mathcal{N}$ is the neighborhood function[2], $\mathbf{h}^k_{v\vdash}$ ($\mathbf{h}^k_{v\dashv}$) is node v's forward (backward) representation which aggregates the information of nodes in $\mathcal{N}_\vdash(v)$ ($\mathcal{N}_\dashv(v)$),

[2]$\mathcal{N}_\vdash(v)$ returns the nodes that v directs to and $\mathcal{N}_\dashv(v)$ returns the nodes that direct to v.

$\mathbf{M}_{\vdash}^{k}$ and $\mathbf{M}_{\dashv}^{k}$ are the forward and backward aggregator functions, $\mathbf{W}^{k}$ denotes weight matrices, σ is a non-linearity function.

For example, for node $v \in \mathcal{V}$, we first aggregate the forward representations of its immediate neighbors $\{\mathbf{h}_{u\vdash}^{k-1}, \forall u \in \mathcal{N}_{\vdash}(v)\}$ into a single vector $\mathbf{h}_{\mathcal{N}_{\vdash}(v)}^{k}$ (equation 2). Note that this aggregation step only uses the representations generated at previous iteration and its initial representation is $\mathbf{a}_v$. Then we concatenate v's current forward representation $\mathbf{h}_{v\vdash}^{k-1}$ with the newly generated neighborhood vector $\mathbf{h}_{\mathcal{N}_{\vdash}(v)}^{k}$. This concatenated vector is fed into a fully connected layer with nonlinear activation function σ, which updates the forward representation of v to be used at the next iteration (equation 3). Next, we update the backward representation of v in the similar fashion (equation 4∼5). Finally, the concatenation of the forward and backward representation at last iteration K, is used as the resulting representation of v. Since the neighbor information from different hops may have the different impact on the node embedding, we learn a distinct aggregator function at each step. This aggregator feeds each neighbor's vector to a fully-connected neural network and an element-wise max-pooling operation is applied to capture different aspects of the neighbor set.

Graph Embedding. Most existing works of graph convolution neural networks focus more on node embeddings rather than graph embeddings (GE) since their focus is on the node-wise classification task. However, graph embeddings that convey the entire graph information are essential to the downstream decoder, which is crucial to our task. For this purpose, we propose two ways to generate graph embeddings, namely, the Pooling-based and Node-based methods.

Pooling-based GE. This method feeds the obtained node embeddings into a fully-connected neural network and applies the element-wise *max*-pooling operation on all node embeddings. In experiments, we did not observe significant performance improvement using min-pooling and average-pooling.

Node-based GE. Following (Scarselli et al., 2009), this method adds a **super** node v_s that is connected to all other nodes by a special type of edge. The embedding of v_s, which is treated as graph embedding, is produced using node embedding generation algorithm mentioned above.

Sequence Decoding. The decoder is an RNN which predicts the next token y_i given all the previous words $y_{<i} = y_1, ..., y_{i-1}$, the RNN hidden state s_i for time-step i and the context vector c_i that captures the attention of the encoder side. In particular, the context vector c_i depends on a set of node representations ($\mathbf{h}_1$,...,$\mathbf{h}_{\mathcal{V}}$) to which the encoder maps the input graph. The context vector c_i is dynamically computed using attention mechanism over the node representations. Our model is jointly trained to maximize the conditional log-probability of the correct description given a source graph with respect to the parameters θ of the model:

$$\theta^* = \arg\max_{\theta} \sum_{n=1}^{N} \sum_{t=1}^{T_n} \log p(y_t^n | y_{<t}^n, x^n)$$

where (x^n, y^n) is the n-th SQL-interpretation pair in the training set, and T_n is the length of the n-th target sentence y^n. In the inference phase, we use the beam search algorithm with beam size = 5.

4 Experiments

We evaluate our model on two datasets, WikiSQL (Zhong et al., 2017) and Stackoverflow (Iyer et al., 2016). WikiSQL consists of a corpus of 87,726 hand-annotated SQL query and natural language question pairs. These SQL queries are further split into training (61,297 examples), development (9,145 examples) and test sets (17,284 examples). StackOverflow consists of 32,337 SQL query and natural language question pairs, and we use the same train/development/test split as (Iyer et al., 2016). We use the BLEU-4 score (Papineni et al., 2002) as our automatic evaluation metric and also perform a human study. For human evaluation, we randomly sampled 1,000 predicted results and asked three native English speakers to rate each interpretation against both the correctness conforming to the input SQL and grammaticality on a scale between 1 and 5. We compare some variants of our model against the template, Seq2Seq, and Tree2Seq baselines.

***Graph2Seq*-PGE.** This method uses the **P**ooling method for generating **G**raph **E**mbedding.

***Graph2Seq*-NGE.** This method uses the **N**ode based **G**raph **E**mbedding.

Template. We implement a template-based method which first maps each element of a SQL query to an utterance and then uses simple rules to assemble these utterances. For example, we

	BLEU-4	Grammar.	Correct.
Template	15.71	1.50	-
Seq2Seq	20.91	2.54	62.1%
Seq2Seq + Copy	24.12	2.65	64.5%
Tree2Seq	26.67	2.70	66.8%
Graph2Seq-PGE	**38.97**	**3.81**	**79.2%**
Graph2Seq-NGE	34.28	3.26	75.3%
(Iyer et al., 2016)	18.4	3.16	64.2%
Graph2Seq-PGE	**23.3**	**3.23**	**70.2%**
Graph2Seq-NGE	21.9	2.97	65.1%

Table 1: Results on the WikiSQL (above) and Stackoverflow (below).

SQL Query & Interpretations
1. COUNT Player WHERE starter = val_0 AND touchdowns = val_1 AND position = val_2
S: How many players played in position val_2
G: number of players with starter val_0 and get touchdowns val_1 for val_2
2. SELECT Tires WHERE engine = val_0 AND chassis = val_1 AND team = val_2
S: which tire has engine val_0 and chassis val_1 and val_2
G: which tire does val_2 run with val_0 engine and val_1 chassis

Table 2: Example of SQL queries and predicted interpretations where S and G denotes Seq2Seq and Graph2Seq models, respectively.

map SELECT to *which*, WHERE to *where*, $>$ to *more than*. This method translates the SQL query of Figure 1 to *which company where assets more than* val_0 *and sales more than* val_0 *and industry less than or equal to* val_1 *and profits equals* val_2.

Seq2Seq. We choose two Seq2Seq models as our baselines. The first one is the attention-based Seq2Seq model proposed by Bahdanau et al. (2014), and the second one additionally introduces the copy mechanism in the decoder side (Gu et al., 2016). To evaluate these models, we employ a template to convert the SQL query into a sequence: "SELECT + <*aggregation function*> + <Split Symbol> + <*selected column*> + WHERE + <*condition*$_0$> + <Split Symbol> + <*condition*$_1$> + ... ".

Tree2Seq. We also choose a tree-to-sequence model proposed by (Eriguchi et al., 2016) as our baseline. We use the SQL Parser tool[3] to convert a SQL query into the tree structure[4] which is fed to the Tree2Seq model.

Our proposed models are trained using the Adam optimizer (Kingma and Ba, 2014), with mini-batch size 30. Our hyper-parameters are set based on performance on the validation set. The learning rate is set to 0.001. We apply the dropout strategy (Srivastava et al., 2014) with the ratio of 0.5 at the decoder layer to avoid overfitting. Gradients are clipped when their norm is bigger than 20. We initialize word embeddings using GloVe word vectors from Pennington et al. (2014), and the word embedding dimension is 300. For the graph encoder, the hop size K is set to 6, the non-linearity function σ is implemented as ReLU (Glorot et al., 2011), the parameters of weight matrices $\mathbf{W}^k$ are randomly initialized. The decoder has one layer, and its hidden state size is 300.

[3] http://www.sqlparser.com

[4] See Appendix for details.

Results and Discussion Table 1 summarizes the results of our models and baselines. Although the template-based method achieves decent BLEU scores, its grammaticality score is substantially worse than other baselines. We can see that on both two datasets, our Graph2Seq models perform significantly better than the Seq2Seq and Tree2Seq baselines. One possible reason is that in our graph encoder, the node embedding retains the information of neighbor nodes within K hops. However, in the tree encoder, the node embedding only aggregates the information of descendants while losing the knowledge of ancestors. The pooling-based graph embedding is found to be more useful than the node-based graph embedding because *Graph2Seq*-NGE adds a nonexistent node into the graph, which introduces the noisy information in calculating the embeddings of other nodes. We also conducted an experiment that treats the SQL query graph as an undirected graph and found the performance degrades.

By manually analyzing the cases in which the Graph2Seq model performs better than Seq2Seq, we find the Graph2Seq model is better at interpreting two classes of queries: (1) the complicated queries that have more than two conditions (Query 1); (2) the queries whose columns have implicit relationships (Query 2). Table 2 lists some such SQL queries and their interpretations. One possible reason is that the Graph2Seq model can better learn the correlation between the graph pattern and natural language by utilizing the global structure information.

We find the hop size has a significant impact on our model since it determines how many neighbor nodes to be considered during the node embedding generation. As the hop size increasing, the performance is found to be significantly improved. However, after the hop size reaches 6,

increasing the hop size can not boost the performance on WikiSQL anymore. By analyzing the most complicated queries (around 6.2%) in WikiSQL, we find there are average six hops between a node and its most distant neighbor. This result indicates that the selected hop size should guarantee each node can receive the information from others nodes in the graph.

5 Conclusions

Previous work approaches the SQL-to-text task using an Seq2Seq model which does not fully capture the global structure information of the SQL query. To address this, we proposed a Graph2Seq model which includes a graph encoder, an attention based sequence decoder. Experimental results show that our model significantly outperforms the Seq2Seq and Tree2Seq models on the WikiSQL and Stackoverflow datasets.

Appendix

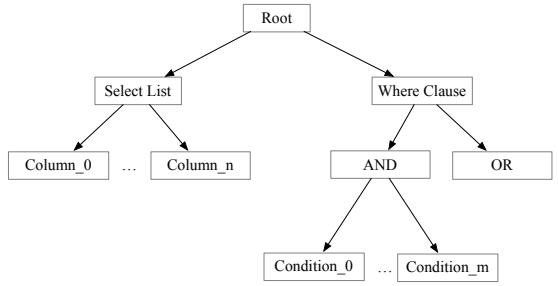

Figure 3: Tree representation of the SQL query.

We apply the SQL Parser tool[5] to convert an SQL query to a tree whose structure is illustrated in Figure 3. More specifically, the root has two child nodes, namely *Select List* and *Where Clause*. The child nodes of *Select List* represent the selected columns in the SQL query. The *Where Clause* has the logical operators occurred in the SQL query as its children. The children of a logical operator node are the conditions on which this operator works.

References

Sanjeev Arora, Yingyu Liang, and Tengyu Ma. 2017. A simple but tough-to-beat baseline for sentence embeddings. In *ICLR*.

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

Akiko Eriguchi, Kazuma Hashimoto, and Yoshimasa Tsuruoka. 2016. Tree-to-sequence attentional neural machine translation. *arXiv preprint arXiv:1603.06075*.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011*, pages 315–323.

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor OK Li. 2016. Incorporating copying mechanism in sequence-to-sequence learning. *arXiv preprint arXiv:1603.06393*.

William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 1025–1035.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 2073–2083.

Yoon Kim. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*.

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.

Georgia Koutrika, Alkis Simitsis, and Yannis E Ioannidis. 2010. Explaining structured queries in natural language. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 333–344. IEEE.

Axel-Cyrille Ngonga Ngomo, Lorenz Bühmann, Christina Unger, Jens Lehmann, and Daniel Gerber. 2013. Sorry, i don't speak sparql: translating sparql queries into natural language. In *Proceedings of the 22nd international conference on World Wide Web*, pages 977–988. ACM.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA.*, pages 311–318.

[5] http://www.sqlparser.com

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543.

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

Alkis Simitsis and Yannis Ioannidis. 2009. Dbmss should talk back too. *arXiv preprint arXiv:0909.1786*.

Linfeng Song, Yue Zhang, Zhiguo Wang, and Daniel Gildea. 2018. A graph-to-sequence model for amr-to-text generation. *arXiv preprint arXiv:1805.02473*.

Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958.

S Vichy N Vishwanathan, Nicol N Schraudolph, Risi Kondor, and Karsten M Borgwardt. 2010. Graph kernels. *Journal of Machine Learning Research*, 11(Apr):1201–1242.

Lingfei Wu, Ian E.H. Yen, Kun Xu, Fangli Xu, Avinash Balakrishnan, Pin-Yu Chen, Pradeep Ravikumar, and Michael J. Witbrock. 2018a. Word mover's embedding: From word2vec to document embedding. In *EMNLP*.

Lingfei Wu, Ian En-Hsu Yen, Fangli Xu, Pradeep Ravikuma, and Michael Witbrock. 2018b. D2ke: From distance to kernel and embedding. *arXiv preprint arXiv:1802.04956*.

Kun Xu, Lingfei Wu, Zhiguo Wang, and Vadim Sheinin. 2018. Graph2seq: Graph to sequence learning with attention-based neural networks. *arXiv preprint arXiv:1804.00823*.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103.