SyntaxSQLNet: Syntax Tree Networks for Complex and Cross-Domain Text-to-SQL Task

Tao Yu   Michihiro Yasunaga   Kai Yang   Rui Zhang
Dongxu Wang   Zifan Li   Dragomir R. Radev
Department of Computer Science, Yale University
{tao.yu, michihiro.yasunaga, k.yang, r.zhang, dragomir.radev}@yale.edu

Abstract

Most existing studies in text-to-SQL tasks do not require generating complex SQL queries with multiple clauses or sub-queries, and generalizing to new, unseen databases. In this paper we propose SyntaxSQLNet, a syntax tree network to address the *complex* and *cross-domain* text-to-SQL generation task. SyntaxSQLNet employs a SQL specific syntax tree-based decoder with SQL generation path history and table-aware column attention encoders. We evaluate SyntaxSQLNet on a new large-scale text-to-SQL corpus containing databases with multiple tables and complex SQL queries containing multiple SQL clauses and nested queries. We use a database split setting where databases in the test set are unseen during training. Experimental results show that SyntaxSQLNet can handle a significantly greater number of complex SQL examples than prior work, outperforming the previous state-of-the-art model by 9.5% in exact matching accuracy. To our knowledge, we are the first to study this complex text-to-SQL task. Our task and models with the latest updates are available at https://yale-lily.github.io/seq2sql/spider.

1 Introduction

Text-to-SQL task is one of the most important sub-task of semantic parsing in natural language processing (NLP). It maps natural language sentences to corresponding SQL queries.

In recent years, some state-of-the-art methods with Seq2Seq encoder-decoder architectures are able to obtain more than 80% exact matching accuracy on some complex text-to-SQL benchmarks such as ATIS and GeoQuery. These models seem to have already solved most problems in this area.

However, as (Finegan-Dollak et al., 2018) show, because of the problematic task definition in the traditional datasets, most of these models just
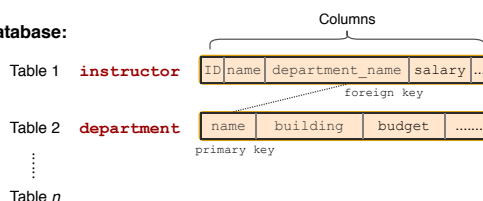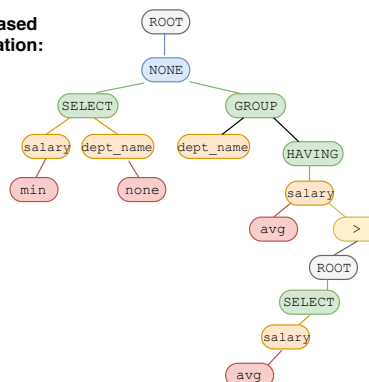

Figure 1: To address the complex text-to-SQL generation task, SyntaxSQLNet employs a tree-based SQL generator. For example, our model can systematically generate a nested query as illustrated above.

learn to match semantic parsing results, rather than truly learn to understand the meanings of inputs and generalize to new programs and databases. More specifically, most existing complex text-to-SQL datasets have less than 500 SQL labels. They are expanded by paraphrasing 4-10 questions for each SQL query. Under the standard train and test split (Zettlemoyer and Collins, 2005), most queries in the test set also appear in the train set. The WikiSQL dataset recently developed by (Zhong et al., 2017) is much larger and does use different databases for training and testing, but it only contains very simple SQL queries and

database schemas.

To address those issues in the current semantic parsing datasets, Yu et al. (2018b) have developed a large-scale human labeled text-to-SQL dataset consisting of about 6,000 complex SQL queries and 200 databases with multiple tables. This dataset defines a new *complex* and *cross-domain* text-to-SQL task that requires models to generalize well to both new SQL queries and databases. The task cannot be solved easily without truly understanding the semantic meanings of the input questions.

In this paper, we propose SyntaxSQLNet, a SQL specific syntax tree network to address the aforementioned task. Specifically, to generate complex SQL queries with multiple clauses, selections and sub-queries, we develop a syntax tree-based decoder with SQL generation path history. To make our model learn to generalize to new databases with new tables and columns, we also develop a table-aware column encoder. Our contributions are as follows:

- We propose SQL specific syntax tree networks for the complex and cross-domain text-to-SQL task, which is even able to solve nested queries on new, unseen databases. We are the first to develop a methodology for this challenging semantic parsing task.

- We introduce a SQL specific syntax tree-based decoder with SQL path history and table-aware column attention encoders. Even with no hyper-parameter tuning, our model can significantly outperform the previous best models, with 4.8% boost in exact matching accuracy. Error analysis shows that our model is able to generalize, and solve much more complex (e.g., nested) queries in unseen databases than prior work.

- We also develop a cross-domain data augmentation method to generate more diverse training examples across databases, which further improves the exact matching accuracy by 7.5%. As a result, our model achieves 22.0% accuracy, a 12.3% total improvement compared with the previous best models.

2 Related Work

Semantic parsing maps natural language to formal meaning representations. There are a range of representations, such as logic forms and executable programs (Zelle and Mooney, 1996; Zettlemoyer and Collins, 2005; Wong and Mooney, 2007; Das

et al., 2010; Liang et al., 2011; Banarescu et al., 2013; Artzi and Zettlemoyer, 2013; Reddy et al., 2014; Berant and Liang, 2014; Pasupat and Liang, 2015).

As a sub-task of semantic parsing, the text-to-SQL problem has been studied for decades (Warren and Pereira, 1982; Popescu et al., 2003a, 2004; Li et al., 2006; Giordani and Moschitti, 2012; Wang et al., 2017b). The methods proposed in the database community (Li and Jagadish, 2014; Yaghmazadeh et al., 2017) tend to involve hand feature engineering and user interactions with the systems. In this work, we focus on recent neural network-based approaches (Yin et al., 2016; Zhong et al., 2017; Xu et al., 2017; Wang et al., 2017a; Iyer et al., 2017). Dong and Lapata (2016) introduce a sequence-to-sequence (seq2seq) approach to converting texts to logical forms. Most previous work focuses on a specific table schema. Zhong et al. (2017) publish the WikiSQL dataset and propose a seq2seq model with reinforcement learning to generate SQL queries. Xu et al. (2017) further improve the results on the WikiSQL task by using a SQL-sketch based approach employing a sequence-to-set model. Dong and Lapata (2018) propose a coarse-to-fine model which achieves the new state-of-the-art performances on several datasets including WikiSQL. Their model first generate a sketch of the target program. Then the model fills in missing details in the sketch.

Our syntax tree-based decoder is related to recent work that exploits syntax information for code generation tasks (Yin and Neubig, 2017; Rabinovich et al., 2017). Yin and Neubig (2017) introduce a neural model that transduces a natural language statement into an abstract syntax tree (AST). While they format the generation process as a seq2seq decoding of rules and tokens, our model uses a module for each grammar component, and calls them recursively to generate a SQL syntax tree. Similarly, Rabinovich et al. (2017) propose abstract syntax networks that use a collection of recursive modules for decoding. Our model differs from theirs in the following points. First, we exploit a SQL specific grammar instead of AST. AST-based models have to predict many non-terminal rules before predicting the terminal tokens, involving more steps. Whereas, our SQL-specific grammar enables direct prediction of SQL tokens. Second, our model uses different sequence-to-set modules to avoid the "or-

dering issue" (Xu et al., 2017) in many code generation tasks. Third, different from (Rabinovich et al., 2017), we pass a pre-order traverse of SQL decoding history to each module. This provides each module with important dependence information: e.g., if a SQL query has `GROUP BY`, it is very likely that the grouped column have appeared in `SELECT` too.

In addition to the distinction in model design, our work differs from theirs in the data and task definition. They aim to develop general syntax model for code generation via abstract syntax trees. Instead, we are interested in solving the complex and cross-domain SQL query generation problem; this motivates us to take advantage of SQL specific syntax for decoding, which guides systematic generation of complex SQL queries.

3 Problem Formulation

This work aims to tackle the complex text-to-SQL task that involves multiple tables, SQL clauses and nested queries. Further, we use separate databases for training and testing, aiming to develop models that generalize to new databases.

Dataset. We use *Spider* (Yu et al., 2018b) as the main dataset, which contains 10,181 questions, 5,693 unique complex SQL queries, and 200 databases with multiple tables.

Task and Challenges.

- The dataset contains a large number of complex SQL labels, which involve more tables, SQL clauses, and nested queries than prior datasets such as WikiSQL. Existing models developed for the WikiSQL task cannot handle those complex SQL queries in the Spider dataset.

- The dataset contains 200 databases (∼138 domains), and different databases are used for training and testing. Unlike most previous semantic parsing tasks (e.g., ATIS), this task requires models to generalize to new, unseen databases.

In sum, we train and test models on different complex SQL queries from different databases in this task. This aims to ensure that models can make the correct prediction only when they truly understand the meaning of the questions under the given database, rather than by mere memorization.
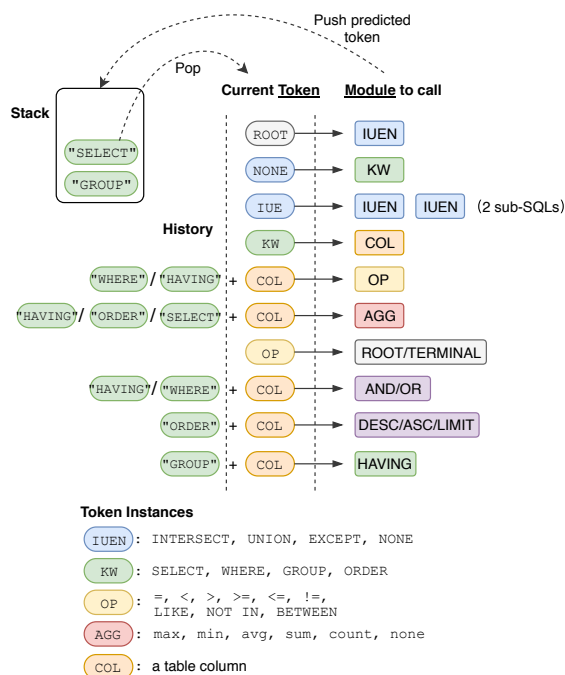

Figure 2: Our modules and SQL grammar used in decoding process. A round symbol represents a SQL tokens, a table column, etc. A square symbol indicates a module that predicts the next SQL token from its corresponding token instances with the same color.

4 Methodology

Similar to (Rabinovich et al., 2017), our model structures the decoder as a collection of recursive modules. However, as we discussed in the related work section, we make use of a SQL specific grammar to guide the decoding process, which allows us to take advantage of SQL queries' well-defined structure. [1]

4.1 Module Overview

Our model decomposes the SQL decoding process into 9 modules to handle the prediction of different SQL components such as keywords, operators, and columns. We provide the overview in this section and more details in later sections.

Figure 2 illustrates our modules and SQL grammar used in decoding process. A round symbol represents a SQL token, such as `SELECT`, `WHERE`, a table column, etc. A square symbol indicates a module that predicts the next SQL token from its corresponding token instances with the same color. Specifically, we have the following modules.

- **IUEN Module**, predicting `INTERCEPT`,

---

[1] Please check out our website for the latest updates on the model at https://yale-lily.github.io/seq2sql/spider

UNION, EXCEPT, and NONE, which determines if we need to call itself again to generate nested queries.

- **KW Module**, predicting keywords from WHERE, GROUP BY, and ORDER BY. All queries in our dataset have SELECT.

- **COL Module**, predicting table columns.

- **OP Module**, for =, >, <, >=, <=, !=, LIKE, NOT IN, IN, BETWEEN.

- **AGG Module**, predicting aggregators from MAX, MIN, SUM, COUNT, AVG, and NONE.

- **Root/Terminal Module**, predicting the ROOT of a new subquery or terminal value. It also enables our model to generate nested queries.

- **AND/OR Module**, predicting the presence of AND or OR operator between two conditions.

- **DESC/ASC/LIMIT Module**, predicting the keywords associated with ORDER BY. It is invoked only when ORDER BY is predicted before.

- **HAVING Module**, predicting the presence of HAVING for GROUP BY clause. It is invoked only when GROUP BY is predicted earlier.

4.2 SQL Grammar

In order to structure our decoder to generate complex queries, we consider a SQL grammar. It determines which module to be invoked at each recursive decoding step. Figure 2 illustrates our SQL grammar. During decoding process, given the current SQL token and the SQL history (the tokens we have gone over to reach the current token), we determine which module to invoke, and predict the next SQL token to generate.

To invoke some modules such as HAVING and OP during decoding, we not only check the type of current token instance but also see whether the type of the previously decoded SQL token is GROUP for HAVING module, and WHERE or HAVING for OP module.

In the grammar, IUEN and Root/Terminal modules are able to generate ROOT, which can activate IUEN module again. In this way, our model can recursively generate nested subqueries, and can also predict two or more subqueries in queries that have EXCEPT, INTERSECT, and UNION.

4.3 Input Encoder

Our inputs of each module consist of three types of information: question, table schema, and current

SQL decoding history path. We encode a question sentence by a bi-directional LSTM, BiLSTM^{Q}. We encode table schema and history path in the manners described below.

4.3.1 Table-Aware Column Representation

In order to generalize to new databases in testing, it is important to make our model learn to obtain necessary information from a database schema. While SQLNet (Xu et al., 2017) only needed the column names as WikiSQL dataset only contained one table per question-SQL pair, *Spider*'s databases contain multiple tables. To address this setting, we propose to use both table and column names to construct column embeddings.

Specifically, given a database, we first obtain embedding for each table by taking the average embedding of the words constituting the table name (e.g., for table name student_id, we average the word embeddings for student and id). Next, for each column, we obtain the initial column name embedding in the same manner, and then concatenate the corresponding table embedding, and the type information of the column (string, or number, primary/foreign key) in a way similar to (Yu et al., 2018a) to produce a column embedding. On top, we apply another level of BiLSTM ($\text{BiLSTM}^{\text{COL}}$) that connects all columns in the database, to obtain high-level column embeddings. This way, our encoding scheme can effectively capture both the global (table names) and local (column names and types) information in the database schema to understand the question in the context of the given database.

Similarly to (Yu et al., 2018a), while the the order of column names or table names does not matter in practice, the use of BiLSTM performed better than the direct use of input column embeddings.

4.3.2 SQL Decoding History

In addition to question and column information, we also pass the SQL query's current decoding history as an input to each module. This enables us to use the information of previous decoding states to predict the next SQL token. For example, in Figure 1, the COL module would be more likely to predict salary in the subquery by considering the path history which contains salary for HAVING, and SELECT in the main query.

In contract, each module in SQLNet does not consider the previous decoded SQL history.

Hence, if directly applied to our recursive SQL decoding steps, each module would just predict the same output every time it is invoked. By passing the SQL history, each module is able to predict a different output according to the history every time it is called during the recursive SQL generation process. Also, the SQL history can improve the performance of each module on long and complex queries because the history helps the model capture the relations between clauses.

Predicted SQL history is used during test decoding. For training, we first traverse each node in the the gold query tree in pre-order to generate gold SQL path history for each training example used in different modules.

4.3.3 Attention for Input Encoding

For each module, like SQLNet, we apply the attention mechanism to encode question representation. We also employs this technique on SQL path history encoding. The specific formulas used are described in the next section.

4.4 Module Details

Similarly to SQLNet (Xu et al., 2017), we employ a sketch-based approach for each module. We apply a seq2set prediction framework introduced by (Xu et al., 2017), to avoid the order issue that happens in seq2seq based models for SQL generation. For example, in Figure 1, SELECT salary, dept_name is the same as SELECT dept_name, salary. The traditional seq2seq decoder generates each of them one by one in order; hence the model could get penalized even if the prediction and gold label are the same as sets. To avoid this problem, SQLNet predicts them together in one step so that their order does not affect the model's training process. For instance, in Figure 1, our model invokes the COL module to predict salary and dept_name and push to stack at the same time. However, SQLNet trains each modules independently, so no information passes through each component.

We first describe how to compute the conditional embedding $\mathbf{H}_{1/2}$ of an embedding $\mathbf{H}_1$ given another embedding $\mathbf{H}_2$:

$$\mathbf{H}_{1/2} = \mathbf{softmax}(\mathbf{H}_1\mathbf{W}\mathbf{H}_2^\top)\mathbf{H}_1.$$

Here $\mathbf{W}$ is a trainable parameter. Moreover, we get a probability distribution from a given score matrix $\mathbf{U}$ by

$$\mathcal{P}(\mathbf{U}) = \mathbf{softmax}\left(\mathbf{V}\mathbf{tanh}(\mathbf{U})\right),$$

where $\mathbf{V}$ is a trainable parameter.

We denote the hidden states of LSTM on question embeddings, path history, and columns embeddings as $\mathbf{H}_Q$, $\mathbf{H}_{HS}$, and $\mathbf{H}_{COL}$ respectively. In addition, we denote the hidden states of LSTM on multiple keywords embeddings and keywords embeddings as $\mathbf{H}_{MKW}$ and $\mathbf{H}_{KW}$ respectively. Finally, we use $\mathbf{W}$ to denote trainable parameters that are not shared between modules. The output of each module is computed as follows:

IUEN Module In the IUEN module, since only one of the multiple keywords from {INTERCEPT, UNION, EXCEPT, NONE} will be used, we compute the probabilities by

$$P_{\text{IUEN}} = \mathcal{P}\left(\mathbf{W}_1\mathbf{H}_{Q/MKW}^\top + \mathbf{W}_2\mathbf{H}_{HS/MKW}^\top + \mathbf{W}_3\mathbf{H}_{MKW}^\top\right)$$

KW Module In the KW module, we first predict the number of keywords in the SQL query and then predict the keywords from {SELECT, WHERE, GROUP BY, ORDER BY}.

$$P_{\text{KW}}^{\text{num}} = \mathcal{P}\left(\mathbf{W}_1^{\text{num}}\mathbf{H}_{Q/KW}^{\text{num}}{}^\top + \mathbf{W}_2^{\text{num}}\mathbf{H}_{HS/KW}^{\text{num}}{}^\top\right)$$

$$P_{\text{KW}}^{\text{val}} = \mathcal{P}\left(\mathbf{W}_1^{\text{val}}\mathbf{H}_{Q/KW}^{\text{val}}{}^\top + \mathbf{W}_2^{\text{val}}\mathbf{H}_{HS/KW}^{\text{val}}{}^\top + \mathbf{W}_3^{\text{val}}\mathbf{H}_{KW}^\top\right)$$

COL Module Similarly, in the COL module, we first predict the number of columns in the SQL query and then predict which ones to use.

$$P_{\text{COL}}^{\text{num}} = \mathcal{P}\left(\mathbf{W}_1^{\text{num}}\mathbf{H}_{Q/COL}^{\text{num}}{}^\top + \mathbf{W}_2^{\text{num}}\mathbf{H}_{HS/COL}^{\text{num}}{}^\top\right)$$

$$P_{\text{COL}}^{\text{val}} = \mathcal{P}\left(\mathbf{W}_1^{\text{val}}\mathbf{H}_{Q/COL}^{\text{val}}{}^\top + \mathbf{W}_2^{\text{val}}\mathbf{H}_{HS/COL}^{\text{val}}{}^\top + \mathbf{W}_3^{\text{val}}\mathbf{H}_{COL}^\top\right)$$

OP Module In the OP module, for each predicted column from the COL module that is in the WHERE clause, we first predict the number of operators on it then predict which operators to use from $\{=, >, <, >=, <=, !=$, LIKE, NOT IN, IN, BETWEEN$\}$. We use $\mathbf{H}_{CS}$ to denote the embedding of one of the predicted columns from the COL module.

$$P_{\text{OP}}^{\text{num}} = \mathcal{P}\left(\mathbf{W}_1^{\text{num}}\mathbf{H}_{Q/CS}^{\text{num}}{}^\top + \mathbf{W}_2^{\text{num}}\mathbf{H}_{HS/CS}^{\text{num}}{}^\top + \mathbf{W}_3^{\text{num}}\mathbf{H}_{CS}^\top\right)$$

$$P_{\text{OP}}^{\text{val}} = \mathcal{P}\left(\mathbf{W}_1^{\text{val}}\mathbf{H}_{Q/CS}^{\text{val}}{}^\top + \mathbf{W}_2^{\text{val}}\mathbf{H}_{HS/CS}^{\text{val}}{}^\top + \mathbf{W}_3^{\text{val}}\mathbf{H}_{CS}^\top\right)$$

AGG Module In the AGG module, for each predicted column from the COL module that is in the SELECT clause, we first predict the number of aggregators on it then predict which aggregators to use from {MAX, MIN, SUM, COUNT, AVG, NONE}

$$P_{\text{AGG}}^{\text{num}} = \mathcal{P}\left(\mathbf{W}_1^{\text{num}}\mathbf{H}_{Q/CS}^{\text{num}}{}^\top + \mathbf{W}_2^{\text{num}}\mathbf{H}_{HS/CS}^{\text{num}}{}^\top + \mathbf{W}_3^{\text{num}}\mathbf{H}_{CS}^\top\right)$$

$$P_{\text{AGG}}^{\text{val}} = \mathcal{P}\left(\mathbf{W}_1^{\text{val}}\mathbf{H}_{Q/CS}^{\text{val}}{}^\top + \mathbf{W}_2^{\text{val}}\mathbf{H}_{HS/CS}^{\text{val}}{}^\top + \mathbf{W}_3^{\text{val}}\mathbf{H}_{CS}^\top\right)$$

Root/Terminal Module To predict nested subqueries, we add a module to predict if there is a new "ROOT" after an operator, which allows the

model to decode queries recursively. For each predicted column from the COL module that is in the `WHERE` clause, we first call OP module, and then predict whether the next decoding step is a "ROOT" node or a value terminal node by

$$P_{\text{RT}} = \mathcal{P}\left(\mathbf{W}_1\mathbf{H}_{\text{Q/CS}}^\top + \mathbf{W}_2\mathbf{H}_{\text{HS/CS}}^\top + \mathbf{W}_3\mathbf{H}_{\text{CS}}^\top\right)$$

AND/OR Module For each condition column predicted from the COL module with number bigger than 1, we predict from {`AND`, `OR`} by

$$P_{\text{AO}} = \mathcal{P}\left(\mathbf{W}_1\mathbf{H}_{\text{Q}}^\top + \mathbf{W}_2\mathbf{H}_{\text{HS}}^\top\right)$$

DESC/ASC/LIMIT Module In the DAL module, for each predicted column from the COL module that is in the `ORDER BY` clause, we predict from {`DESC`, `ASC`, `DESC LIMIT`, `ASC LIMIT`} by

$$P_{\text{DAL}} = \mathcal{P}\left(\mathbf{W}_1\mathbf{H}_{\text{Q/CS}}^\top + \mathbf{W}_2\mathbf{H}_{\text{HS/CS}}^\top + \mathbf{W}_3\mathbf{H}_{\text{CS}}^\top\right)$$

HAVING Module In the HAVING module, for each predicted column from the COL module that is in the `GROUP BY` clause, we predict whether it is in the `HAVING` clause by

$$P_{\text{HAVING}} = \mathcal{P}\left(\mathbf{W}_1\mathbf{H}_{\text{Q/CS}}^\top + \mathbf{W}_2\mathbf{H}_{\text{HS/CS}}^\top + \mathbf{W}_3\mathbf{H}_{\text{CS}}^\top\right)$$

4.5 Recursive SQL Generation

The SQL generation process is a process of activating different modules recursively. As illustrated in Figure 2, we employ a stack to organize our decoding process. At each decoding step, we pop one SQL token instance from the stack, and invoke a module based on the grammar to predict the next token instance, and then push the predicted instance into the stack. The decoding process continues until the stack is empty.

More specifically, we initialize a stack with only `ROOT` at the first decoding step. At the next step, the stack pops `ROOT`. As illustrated in Figure 2, `ROOT` actives the `IUEN` module to predict if there is `EXCEPT`, `INTERSECT` or `UNION`. If so, there are two subqueries to be generated in the next step. If the model predicts `NONE` instead, it will be pushed into the stack. The stack pops `NONE` at next step. For example, in Figure 2, the current popped token is `SELECT`, which is a instance of keyword (KW) type. It calls the COL module to predict a column name, which will be pushed to the stack.

4.6 Data Augmentation

Even though *Spider* already has a significantly larger number of complex queries than existing datasets, the number of training examples for some complex SQL components is still limited. A widely used way is to conduct data augmentation to generate more training examples automatically. Many studies (Berant and Liang, 2014; Iyer et al., 2017; Su and Yan, 2017) have shown that data augmentation can bring significant improvement in performance. In prior work, data augmentation was typically performed within a single domain dataset. We propose a cross-domain data augmentation method to expand our training data for complex queries. Cross-domain data augmentation is more difficult than the in-domain setting because question-program pairs tend to have domain specific words and phrases. To tackle this issue, we first create a list of universal patterns for question-SQL pairs, based on the human labeled pairs from all the different training databases in *Spider*. To do so, we use a script to remove (and later fill in) all the table/column names and value tokens in the labeled question-SQL pairs, and then group together the same SQL query patterns. Consequently, each SQL query pattern has a list of about 5-20 corresponding questions. In our task, we want to generate more complex training examples. Thus, we filter out simple SQL query patterns by measuring the length and the number of SQL keywords used. We obtain about 280 different complex SQL query patterns from the 4,000+ SQL labels in the train set of our corpus. We then select the 50 most frequent complex SQL patterns that contain multiple SQL components and nested subqueries. After this, we manually edit the selected SQL patterns and their corresponding list of questions to make sure that the table/column/value slots in the questions have one-to-one correspondence to the slots in the corresponding SQL query. For each slot, we also add column type or table information. Thus, for example, columns with string type do not appear in the column slot with integer type during data augmentation (i.e., slot refilling) process. In this way, our question-SQL patterns are generated based on existing human labeled examples, which ensures that the generated training examples are natural.

Once we have the one-to-one slot mapping between questions and SQL queries, we apply a script that takes a new database schema with type information and generates new question-SQL examples by filling empty slots. Specifically, for each database in WikiSQL, we first randomly sam-

ple 10 question-SQL patterns. We randomly sample columns from the database schema based on its type: for example, if the slot type in the pattern is "number", and then we only sample from columns with "real" type in the current database. We then refill the slots in both the question and SQL query with the selected column names. Similarly, we also refill table/value slots.

By this data augmentation method, we finally obtain about 98,000 question and SQL pairs using about 18,000 WikiSQL databases with one single table. Additionally, we also use the original WikiSQL training dataset in experiments.

5 Experiments

Dataset In our experiments, we use Spider (Yu et al., 2018b), the new large-scale human annotated text-to-SQL dataset with complex SQL queries and cross-domain databases. In addition to their originally annotated data, their training split includes 752 queries and 1659 questions from six existing datasets: Restaurants (Tang and Mooney, 2001; Popescu et al., 2003b), GeoQuery (Zelle and Mooney, 1996), Scholar (Iyer et al., 2017), Academic (Li and Jagadish, 2014), Yelp and IMDB (Yaghmazadeh et al., 2017). In total, this dataset consists of 11,840 questions, 6,445 unique complex SQL queries, and 206 databases with multiple tables. We follow (Yu et al., 2018b), and use 130, 36, 40 databases for train, development, test, respectively (randomly split).

5.1 Evaluation

We use evaluation metrics including SQL Component matching and Exact matching. To compute the component matching scores, we first decompose predicted queries on SQL clauses including SELECT, WHERE, GROUP BY, ORDER BY, and KEYWORDS separately. After that, we evaluate each predicted clause and the ground truth as bags of several sub-components, and check whether or not these two sets of components match exactly. F1 scores are reported for each of the above 5 clauses. Exact matching score is 1 if the model predicts all clauses correctly for a given example.

To better understand model performance on different queries, (Yu et al., 2018b) divide SQL queries into 4 levels: easy, medium, hard, extra hard. The definition of difficulty is based on the number of SQL components, selections, and conditions. Queries that contain more SQL keywords are considered harder.

5.2 Experimental Settings

Our model is implemented in PyTorch (Paszke et al., 2017). We build each module based on the TypeSQL (Yu et al., 2018a) implementation. We use pre-trained GloVe (Pennington et al., 2014) embeddings for all question, SQL history, and schema tokens. All word embeddings are fixed. For each experiment, the dimension and dropout rate of all hidden layers is set to 120 and 0.3 respectively. We use Adam (Kingma and Ba, 2015) with the default hyperparameters for optimization, with a batch size of 64. The same loss functions in (Xu et al., 2017) are used.

- **Training data:** *Spider* (plus examples from 6 existing datasets) + WikiSQL + data augmentation

- **Model architecture:** history path + table-aware column encoding

We will conduct ablation studies to analyze the effect of each of the proposed techniques.

5.3 Baseline Models

To demonstrate the efficacy of our model in addressing the complex, cross-domain text-to-SQL task, we compare the performance of our model with several previous state-of-the-art models in the text-to-SQL task. As the dataset and task definition used in this work are fundamentally different from prior work using datasets such as GeoQuery, WikiSQL, we adapted these models to our task in the same way as (Yu et al., 2018b). Specifically:

Seq2Seq with Attention or Copying In order to make the models aware of the table schema information, Yu et al. (2018b) pass the models with a vocabulary that contains SQL keywords and column names of the given database.

(Iyer et al., 2017) Iyer et al. (2017) apply an attention based seq2seq model similar to (Luong et al., 2015) to text-to-SQL tasks. Yu et al. (2018b) adapt their model without user interaction to the task.

SQLNet & TypeSQL Xu et al. (2017) introduce SQLNet, which employs a column attention mechanism and a sketch-based method to generates SQL queries as a slot-filling task. Yu et al. (2018a) improves SQLNet by utilizing word types extracted from a knowledge graph or table content

Method	Easy	Medium	Hard	Extra Hard	All
Seq2Seq	17.9%	2.7%	1.3%	0.6%	5.4%
Seq2Seq+Attention (Dong and Lapata, 2016)	17.9%	2.9%	1.8%	1.3%	5.7%
Seq2Seq+Copying	15.1%	3.4%	1.0%	1.3%	5.2%
Iyer et al. (2017)	7.9%	2.1%	1.3%	1.1%	3.1%
SQLNet (Xu et al., 2017)	23.7%	5.9%	2.3%	0.3%	8.3%
TypeSQL (Yu et al., 2018a)	29.6%	6.1%	2.3%	0.3%	9.7%
SyntaxSQLNet	**43.1%**	**19.2%**	**17.8%**	**4.8%**	**22.0%**
-augment	36.9%	16.7%	9.5%	1.6%	17.6%
-wikiSQL -augment	34.1%	11.2%	9.0%	2.2%	14.5%
-table -wikiSQL -augment	30.1%	7.3%	4.5%	1.9%	10.9%
-history -table -wikiSQL -augment	18.8%	4.5%	0.0%	0.0%	6.1%

Table 1: Exact match accuracy on SQL queries with different hardness levels.

Method	SELECT	WHERE	GROUP BY	ORDER BY	KEYWORDS
Seq2Seq	13.7%	3.7%	3.2%	4.9%	8.9%
Seq2Seq+Attention	14.0%	5.0%	3.2%	6.1%	9.4%
Seq2Seq+Copying	12.0%	2.7%	5.2%	6.9%	6.7%
Iyer et al. (2017)	6.3%	1.9%	3.0%	3.6%	3.5%
SQLNet	24.0%	18.0%	11.8%	47.1%	61.9%
TypeSQL	36.2%	14.7%	6.4%	49.5%	59.4%
SyntaxSQLNet	**48.2%**	**31.6%**	**28.9%**	**58.4%**	**68.9%**
-augment	45.3%	29.2%	22.0%	47.3%	67.5%
-wikiSQL -augment	39.0%	20.6%	17.1%	50.2%	65.8%
-table -wikiSQL -augment	29.0%	15.0%	13.1%	45.5%	67.0%
-history -table -wikiSQL -augment	18.6%	10.5%	4.1%	30.0%	49.1%

Table 2: F1 scores of Component Matching on all SQL queries.

to help the model better understand entities and numbers in natural language inputs. As they are originally designed for WikiSQL, to conduct experiments on Spider, Yu et al. (2018b) extend their SELECT and WHERE modules to other SQL components.

6 Results and Discussion

Table 1 presents our test results on the *Spider* dataset with database splitting. Our model with full history and data augmentation achieves 22.0% exact matching on all SQL queries, which is a 12.3% absolute increase compared to the previous best model, TypeSQL.

6.1 Comparison with Baselines

Even though our individual modules are similar to SQLNet and TypeSQL, our syntax-aware decoder allows the modules to incorporate the SQL decoding history, which helps to achieve a significant gain in exact matching for queries of all hardness levels. Specifically, even without our data augmentation technique, SyntaxSQLNet outperforms the previous best, TypeSQL, by 5%. This result

suggests that the syntax and history information is beneficial for this complex text-to-SQL task.

Moreover, the tree-based decoder enables SyntaxSQLNet to systematically generate nested queries, boosting the performance for Hard/Extra Hard. As Table 1 shows, SyntaxSQLNet achieves particularly high scores 17.8% and 4.8% for Hard and Extra Hard, which contain nested queries. The Seq2Seq models suffer from generating ungrammatical queries, yielding very low exact matching accuracy on Hard and Extra Hard SQL queries. In contrast, our model generates valid SQL queries by enforcing the syntax.

For the detailed component matching results in Table 2, our model consistently outperforms other previous work by significant margins. Specifically, our model improve F1 score for most of the SQL components by more than 10%.

6.2 Ablation Study

In order to understand the techniques that are responsible for the performance of our model, we perform an ablation study where we remove one of the proposed techniques from our model at a time. The exact match scores are shown in the same ta-

bles as other baselines.

6.2.1 Data Augmentation

Our model's exact matching performance on all queries drops 4.4% by excluding data augmentation technique. This drop is particularly large for `GROUP BY` and `ORDER BY` components (Table 2), for which the original Spider dataset has a relatively small number of training examples. Thus, our cross-domain data augmentation technique significantly benefits the model performance by extending to a much larger training dataset.

6.2.2 SQL Decoding History

In order to gain more insight into how our SQL decoding history addresses complex SQL, we report our model's performance without SQL path history. As shown in the Table 1, the model's performance drops about 4.8% on exacting matching metric without the SQL history input. More importantly, its performance on hard and extra hard SQL queries decreases to 0%. This indicates that our model is able to predict nested queries thanks to the SQL decoding history.

6.2.3 Column Encoding

To see how our table-aware column encoding affects performance of our model, we also report the model's result without using table information for our column encoding. After excluding the table embedding from column embeddings, the test performance goes down by about 4%. This drop is especially large for Medium/Hard SQL queries and `SELECT` component, where the correct column prediction is a key. This result shows that the table-aware encoding is very important to predict the correct columns in unseen, complex dasebases (with many foreign keys).

7 Conclusion

In this paper, we presented a syntax tree-based model to address complex and cross-domain text-to-SQL task. Utilizing a SQL specific syntax decoder, as well as SQL path history and table-aware column attention encoders, our model outperforms previous work by a significant margin. The ablation study demonstrates that our proposed techniques are able to predict nested, complex SQL queries correctly even for unseen databases.

References

Yoav Artzi and Luke Zettlemoyer. 2013. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association forComputational Linguistics*.

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*.

Jonathan Berant and Percy Liang. 2014. Semantic parsing via paraphrasing. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1415–1425, Baltimore, Maryland. Association for Computational Linguistics.

Dipanjan Das, Nathan Schneider, Desai Chen, and Noah A. Smith. 2010. Probabilistic frame-semantic parsing. In *NAACL*.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*.

Li Dong and Mirella Lapata. 2018. Coarse-to-fine decoding for neural semantic parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 731–742. Association for Computational Linguistics.

Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan Dhanalakshmi Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. 2018. Improving text-to-sql evaluation methodology. In *ACL 2018*. Association for Computational Linguistics.

Alessandra Giordani and Alessandro Moschitti. 2012. Translating questions to sql queries with generative parsers discriminatively reranked. In *COLING (Posters)*, pages 401–410.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a neural semantic parser from user feedback. *CoRR*, abs/1704.08760.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. *The 3rd International Conference for Learning Representations, San Diego*.

Fei Li and HV Jagadish. 2014. Constructing an interactive natural language interface for relational databases. *VLDB*.

Yunyao Li, Huahai Yang, and HV Jagadish. 2006. Constructing a generic natural language interface for an xml database. In *EDBT*, volume 3896, pages 737–754. Springer.

P. Liang, M. I. Jordan, and D. Klein. 2011. Learning dependency-based compositional semantics. In *Association for Computational Linguistics (ACL)*, pages 590–599.

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*.

Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*, pages 1470–1480.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. *NIPS 2017 Workshop*.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *EMNLP*, pages 1532–1543. ACL.

Ana-Maria Popescu, Alex Armanasu, Oren Etzioni, David Ko, and Alexander Yates. 2004. Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *Proceedings of the 20th international conference on Computational Linguistics*, page 141. Association for Computational Linguistics.

Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 2003a. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th international conference on Intelligent user interfaces*, pages 149–157. ACM.

Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 2003b. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th International Conference on Intelligent User Interfaces*.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *ACL (1)*, pages 1139–1149. Association for Computational Linguistics.

Siva Reddy, Mirella Lapata, and Mark Steedman. 2014. Large-scale semantic parsing without question-answer pairs. *Transactions of the Association for Computational Linguistics*, 2:377–392.

Yu Su and Xifeng Yan. 2017. Cross-domain semantic parsing via paraphrasing. *CoRR*, abs/1704.05974.

Lappoon R Tang and Raymond J Mooney. 2001. Using multiple clause constructors in inductive logic programming for semantic parsing. In *ECML*, volume 1, pages 466–477. Springer.

Chenglong Wang, Marc Brockschmidt, and Rishabh Singh. 2017a. Pointing out sql queries from text. *Technical Report*.

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017b. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 452–466. ACM.

David HD Warren and Fernando CN Pereira. 1982. An efficient easily adaptable system for interpreting natural language queries. *Computational Linguistics*, 8(3-4):110–122.

Yuk Wah Wong and Raymond J. Mooney. 2007. Learning synchronous grammars for semantic parsing with lambda calculus. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics (ACL-2007)*, Prague, Czech Republic.

Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436*.

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. Sqlizer: Query synthesis from natural language. *Proc. ACM Program. Lang.*, 1(OOPSLA):63:1–63:26.

Pengcheng Yin, Zhengdong Lu, Hang Li, and Ben Kao. 2016. Neural enquirer: Learning to query tables in natural language. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 2308–2314.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *ACL (1)*, pages 440–450. Association for Computational Linguistics.

Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir Radev. 2018a. Typesql: Knowledge-based type-aware neural text-to-sql generation. In

Proceedings of NAACL. Association for Computational Linguistics.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018b. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *EMNLP*.

John M. Zelle and Raymond J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *AAAI/IAAI*, pages 1050–1055, Portland, OR. AAAI Press/MIT Press.

Luke S. Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. *UAI*.

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