Sp21 CS388 NLP Final Report: Text to SQL

Haoqi Wang
University of Texas at Austin
EID: hw9335
haoqiwang@utexas.edu

Bingye Li
University of Texas at Austin
EID: bl26924
libingye@utexas.edu

Reproducibility Summary

Scope of Reproducibility

In this project, we want to test the hypothesis that hardcoding the format the output tokens according to SQL grammars can largely increase the performance of the model. We force the output to comply with the SQL grammers by two ways: First, we train the model to predict different types of tokens separately based on the position of the token. Second, we do run-time execution check to prune some predictions that are obviously wrong.

Methodology

Our model implements the two methods mentioned above. It borrows the ideas from two paper: PointSQL (Wang et al., 2017a) and Execution-Guided Decoding (Wang et al., 2018) with some tiny modifications. To compare with it, we also wrote a baseline model using the seq2seq method by reusing part of the code from project 2.

Results

The results shows the pointer network can significantly increase the development accuracy. Baseline seq2seq model with attention can achieve token-level accuracy of 41.6% but logical form accuracy 0. With pointer, the token-level accuracy reaches more than 60% and logical form accuracy around 25%. The result of pointer with execution guide is close to that of only pointer model.

What was easy

The data preprocessing is straightforward since the SQL query of our datasets is relatively well structured. The baseline model is easy to implement because we can directly reuse the code from project 2.

What was difficult

The new model is difficult to implement. We wrote our own version in PyTorch and met lots of bugs to fix. It is very time-consuming to figure out the reasons behind. In addition, we lack the hardware resources and annotated datasets to reach the same perfect results as the authors.

1 Introduction

As the development of IT, more and more large-scale data is stored in databases and spreadsheet tables, while querying such data requires understanding of specialized tools and programming languages like SQL. Our long term goal is to translate natural language questions into executable queries based on natural language processing algorithms.

2 Scope of reproducibility

Our baseline seq2seq model generates SQL sentence as text. However, SQL sentence has its own grammars to obey, which can be leveraged to generate correct predictions from the perspective of grammar. We guess that those predictions in correct grammars can show better accuracy than those generated randomly.

We force the output to comply with the SQL grammers in two ways: First, we train the model to predict different types of tokens separately based on the position of the token. Second, we do runtime execution check to prune some predictions that are obviously wrong.

In (Wang et al., 2017a) they claim that the model trained using only supervised learning that uses a simple type system of SQL expressions to structure the output prediction significantly outperforms the current state-of-the-art Seq2SQL model that uses reinforcement learning on the recently released WikiSQL dataset. In (Wang et al., 2018), it is shown that execution guidance universally improves model performance on various text-to-SQL datasets with different scales and query complexity including WikiSQL.

2.1 Model descriptions

We will partially combine those two approaches using our own implementation that is compatible to the seq2seq baseline model. The performance will be evaluated by the exact match and token level accuracy.

3 Methodology

In our project, we reimplemented the approaches from (Wang et al., 2017a) and (Wang et al., 2018) with some modifications. The model section will cover the architecture of our model inherited from those two papers. More details are described in Experimental setup and code section.

(Wang et al., 2017a) introduces a deep sequence to sequence model in which the decoder uses a simple type system of SQL expressions to structure the output prediction. Based on the type, the decoder either copies an output token from the input question using an attention-based copying mechanism or generates it from a fixed vocabulary. The models also has an encoder-decoder architecture. Both the encoder and decoders are composed of LSTM cells.

The idea is inspired by the fact that the subset of SQL necessary to answer WikiSQL Questions can be represented using the following grammar, in which t refers to the name of the table being queried, c refers to a column name in the table, and v refers to any open world string or number that may be used in the query.

Therefore, if we have to produce a column name or a constant, we enforce the use of a copying mechanism, otherwise we project the hidden state to a built-in vocabulary to obtain a built-in SQL operator. We found it sufficient to distinguish three different cases by types: τ_ν, τ_C, and τ_Q. τ_ν represents the built-in SQL operators like select, where, count, =, etc. τ_C represents the column name and table name. τ_Q represents the constant values used in where clauses. Given the general grammar of the SQL sentence, we can further express the type sequence as $\tau_\nu\tau_\nu\tau_C\tau_\nu\tau_C\tau_\nu(\tau_C\tau_\nu\tau_Q\tau_\nu)^*$. We can see that the type is determined given the index of output token in the SQL sentence.

The copy mechanism has three available loss functions provided by the paper. We selected the value-based one that achieved the best performance in paper's experiment, called Sum Transfer. This strategy calculates the probability of copying a token v in the input vocabulary as the sum of probabilities of pointers that point to token v. The benefit is that if the token v occurs multiple times in input sequence than it makes sense to treat them together because each occurrence contributes to predicting the token v as the output. Moreover, it guarantees that the output tokens are unique even though the input sequence has duplicated tokens.

(Wang et al., 2018) introduce a new mechanism, execution guidance, to leverage the semantics of SQL. It detects and excludes faulty programs during the decoding procedure by conditioning on the execution of partially generated program.

3.1 Datasets

In this project, we used WikiSql dataset. The original dataset can be found at https://github.com/

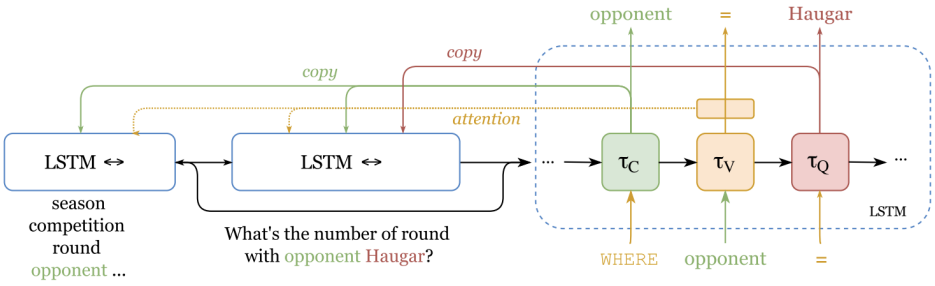

Figure 1: Model overview from (Wang et al., 2017a).

$$
\begin{array}{rcl}
Q & \rightarrow & s\ c\ \texttt{From}\ t\ \texttt{Where}\ p \\
s & \rightarrow & \texttt{Select} \mid \texttt{Max} \mid \texttt{Min} \mid \texttt{Count} \mid \texttt{Sum} \mid \texttt{Avg} \\
p & \rightarrow & c\ op\ v \mid p\ \texttt{And}\ p \\
op & \rightarrow & = \mid > \mid \geq \mid < \mid \leq
\end{array}
$$

Figure 2: SQL format from (Wang et al., 2017a).

Algorithm 1 Execution-guided decoding as an extension of a standard recurrent autoregressive decoder.

procedure EG-DECODING(encoded query O, encoded table columns C, beam size k)
  $\mathbf{h}_0 \leftarrow$ an initial hidden decoder state
  for all steps $1 \leq t \leq T$ of k-beam decoding **do**
    # Compute k new decoder states in the beam:
    $\mathbf{h}_t^1, \ldots, \mathbf{h}_t^k \leftarrow \text{DECODE}(O, C, \mathbf{h}_{t-1}^1, \ldots, \mathbf{h}_{t-1}^k)$
    $P_t^1, \ldots, P_t^k \leftarrow$ partial programs corresponding to the states $\mathbf{h}_t^1, \ldots, \mathbf{h}_t^k$
    if the current stage t is executable **then**
      *# Retain only the top k **executable** programs:*
      for $1 \leq i \leq k$ **do**
        if P_t^i has an error or empty output **then**
          Remove $\mathbf{h}_t^i$ from the beam
  return the top-scored program $\underset{1 \leq i \leq k}{\text{argmax}} \Pr(P_T^i)$

Figure 3: Pesudocode from (Wang et al., 2018).

salesforce/WikiSQL. We used a preprocessed dataset provided by (Huang et al., 2018); (Wang et al., 2018); (Wang et al., 2017b), which can be found here. This dataset contains 56324 training examples, 8419 development examples, 15873 testing examples. To efficiently develop our algorithm and debug, we only used a small portion, including randomly selected 500 training examples and 125 development examples.

We also did some additional preprocessing. The original data of groundtruth - y token is rather primitive, for example as shown below.

```
2-18226024-7 select date visiting^team =
    san^francisco^49ers year > 2007
```

We hardcode it to be a correct SQL query.

```
SELECT date FROM 2-18226024-7 WHERE
   visiting^team = san^francisco^49ers
   AND year > 2007
```

Especially for pointer-SQL model, we add $< GO >$ as a placeholder, which so that the input becomes

```
SELECT <GO> date FROM 2-18226024-7 WHERE
    visiting^team = san^francisco^49ers
    AND year > 2007
```

If there are operations, such as Max, Min, $< GO >$ will be replaced. An example is shown as figure 4. The columns names are annotated with the entity linking technique (Krishnamurthy et al., 2017).

3.2 Hyperparameters

For the baseline model, we used the hyperparameters as in project 2. Details are listed in the second column in table 1.

For the pointer model, we manually searched the hyperparameter. Details are listed in the third column in table 1. Learning rate is searched from 0.001 to 0.1. 0.001 can result in very slow gradient descent and 0.1 can cause gradient explosion. 0.01 is an appropriate learning rate. After learning rate

Question
what date after year 2007 had the san^francisco^49ers as the visiting^team ?

Table name and column names				
2-18226024-7	year	date	home^team	...
...	...	...	...	...

SQL query
SELECT date FROM 2-18226024-7 WHERE visiting^team = san^francisco^49ers AND year > 2007

Figure 4: An example of preprocessed WikiSQL dataset

was decided, we searched number of epochs and the range from 5 to 15 is proper. Larger number of epochs can lead to loss becoming 'nan'. The decoder length limit is set as 22. We determined this by looking at the maximum length output in training data. Other hyperparameter, including hidden dimension, embedding dimension, number of layers, etc., was the same as paper (Wang et al., 2018).

3.3 Experimental setup and code

The code of our project can be found here. The main branch contains the best model with execution check. The no_execution_check branch contains the pointSQL model. The no_pretrained branch contains the baseline model.

For baseline model, we replaced the dataset of project 2 with our WikiSQL and rerun the program again. For our new model, we maintained the interface of project 2 so the commands are the same while we further add more functionalities to the seq2seq model. In addition to input_indexer and output_indexer used for word embeddings, we added grammar_indexer and copy_indexers to index the output token individually for each kind of decoder. We changed the encoder model so that it not only outputs the last hidden state but also an intermediate hidden state right after the header inputs. We wrote another forward method in decoder to output the probability of attention mechanism for copying. Apart from that, we added a type system. In both training and decoding, the model need to call the correct forward method based on the type of the token to be predicted. Finally, we added an execution check in decoding. Since we don't use the annotated dataset, we wrote a simpler version here to guarantee that there is no syntax error. We also checked that the inequality operator must be followed by a numeric constant.

To evaluate the performance of our model, we compare the predicted tokens with the golden sequence and compute two metrics: the exact match accuracy and token-level accuracy. Special tokens like $< GO >$ and $< EOS >$ are omitted in evaluation.

3.4 Computational requirements

The baseline model was trained on MacBook Pro with CPU 2.3 GHz 8-Core Intel Core i9. Each epoch took about 45 - 50 seconds.

The pointer model was trained on Google Colaboratory with CPU Intel(R) Xeon(R) CPU @ 2.30GHz. Each epoch took about 30 - 35 seconds.

4 Results

Table 2 shows the results of our model with the best performance on development set. We report the accuracy computed with token-level match and the accuracy based on exact logical match. The results show pointer model outperforms the baseline model, but pointer with execution guide does not provide much improvement. More details are discussed below.

4.1 Baseline Result

Baseline seq2seq model with attention can learn some basic grammar like putting 'select' at the front as an example shown below, but it fails to predict the table name '2-15400315-1'. y_tok is the groundtruth and y_pred is the output from model. It can achieve token-level accuracy as 41.6%. However, it cannot produce any correct logical form as shown in table 2.

```
y_tok = "['select', 'nation', 'from',
    '2-15400315-1', 'where', 'total',
    '<', '27', 'and', 'gold', '<', '1', '
    and', 'bronze', '>', '1']"

y_pred = "['select', 'avg', 'silver', '
    from', '2-12377104-4', 'where', '
    rank', '>', '0', 'and', 'rank', '<',
     '9']"
```

	Baseline model	Pointer and Pointer + EG model
Loss function	Cross entropy	Cross entropy
Optimizer	Adam	Adagrad
Batch size	1	1
Number of epochs	20	15
Learning rate	0.001	0.01
Embedding dimension	100	100
Hidden dimension	100	100
Number of RNN layers	1	3

Table 1: Hyperparameter details

Model	$DevAcc_{tok}$	$DevAcc_{log}$
Baseline	515 / 1237 = 0.416	0 / 125 = 0.000
Pointer (3)	800 / 1237 = 0.647	33 / 125 = 0.264
Pointer (5)	751 / 1237 = 0.607	30 / 125 = 0.240
Pointer + EG (3)	784 / 1237 = 0.634	29 / 125 = 0.232
Pointer + EG (5)	783 / 1237 = 0.633	34 / 125 = 0.272

Table 2: Dev accuracy of the model, where Acc_{tok} refers to token-level accuracy and Acc_{log} refers to exact logical form matches. (k) indicates that model outputs are generated with beam size k.

4.2 Pointer Result

Based on seq2seq model, we added a pointer. This pointer is meant to learn to copy table name and column names from input. As shown in table 2, it can reach token-level accuracy of 64.7% and exact logical match of 26.4%. This is much better than the baseline model. The results from beam size 3 and 5 are similar.

As an example shown below, it can predict the correct table name and some column names are matched. However, this output is not executable because the grammar after 'where' is not logical.

```
y_tok = "['select', 'score', 'from',
    '2-11902503-8', 'where', 'game', '<',
     '69', 'and', 'march', '>', '2', '
    and', 'opponent', '=', 'new^york^
    islanders']"

y_pred = "['select', 'score', 'from',
    '2-11902503-8', 'where', 'game', '=',
     '69', 'and', '2', '<', 'march', '<',
     '2']"
```

4.3 Pointer with Execution Guide Result

Based on pointer model, we added an execution check in the decoding beam search. Here are two examples. The first is the same as in Pointer Result, compared with which the grammar after 'where' is correct. The second ends early. These examples have correct grammar but lacks in logic, which implies that the encoder may not be able to fully capture the meaning of natural question. The accuracy is similar to the pointer model. We did not observe a much better result as presented in paper (Wang et al., 2018).

```
y_tok = "['select', 'score', 'from',
    '2-11902503-8', 'where', 'game', '<',
     '69', 'and', 'march', '>', '2', '
    and', 'opponent', '=', 'new^york^
    islanders']"

y_pred = "['select', 'score', 'from',
    '2-11902503-8', 'where', 'march',
    '>', '2']"

y_tok = "['select', 'count', 'matches',
    'from', '2-11628153-8', 'where', '
    lost', '<', '5', 'and', 'season',
    '=', '2008/2009', 'and', 'draw', '>',
     '9']"

y_pred = "['select', 'avg', 'matches', '
    from', '2-11628153-8', 'where', '
    loss', '>', '5', 'and', '2008/2009',
     '<', '9']"
```

5 Discussion

5.1 Implementation

During implementing the model, we met various bugs and some of them are worth discussing.

The first one is backward error. The message says that one of the variables needed for gradient computation has been modified by an inplace operation. It turns out that we cannot reassign the value of elements in tensors after the forward method is

called. There are two places where inplace operations happen. One is that y_indexed is the index in output_indexer for embedding purpose. However, it should be the index in either grammar_indexer or copy_indexer. Initially, we do the transformation right before computing loss at each timestamp. This can be solved by preprocessing y_indexed. The other one is that copy mechanism requires summing up the probabilities of repeated tokens. Initially, we use a for loop to iterate through the probability tensor. We can avoid for loop by using mask tensor and tensor multiplication.

The second one is at decoding stage. We use beam search together with execution check to derive token sequences. However, it could be possible that all the top results in beach search fail in execution check. Therefore, we need another beam to store obviously wrong prediction. If the size of end_beam is zero, we turn to fetch predictions from error_beam which could be partially correct. Another problem is the probability in beam search. Because predicting next token always decreases the joint probability, it is possible that beam search select a prediction that mistakenly ends very early. To avoid this happen, we need to pay attention to the implementation. Even if the end token is predicted, the prediction cannot be immediately put into end_beam. It has to win in the topK selection.

5.2 Model performance

The baseline model totally failed at predicting any logical form (Baseline Result). One reason is that it fails to predict the correct table name. It is very likely that the development set contains tables different from the training set, in which case it will be labeled as unknown word. Without copying module, the model cannot predict the correct table name.

As we added pointer, the model is able to predict correct logical form (Pointer Result).This improvement shows pointer plays an important in predicting SQL queries. However, the execution guide (Pointer with Execution Guide Result) does not seem to improve much accuracy. Although it guarantees that the SQL query has the correct grammar, it is using the wrong column names or operators. It is possible that the model has limited or overtrained embeddings of vocabulary, in which case it cannot encode the natural question correctly so that the execution guide cannot assist much. To overcome this problem, we may use pretrained word vector and prevent overtraining. What is more, other encoder, like transformer, might be used to improve encoding.

References

Po-Sen Huang, Chenglong Wang, Rishabh Singh, Wen tau Yih, and Xiaodong He. 2018. Natural language to structured query generation via meta-learning. In *NAACL*.

Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. 2017. Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1516–1526, Copenhagen, Denmark. Association for Computational Linguistics.

Chenglong Wang, Marc Brockschmidt, and Rishabh Singh. 2017a. Pointing out sql queries from text. Technical Report MSR-TR-2017-45.

Chenglong Wang, Marc Brockschmidt, and Rishabh Singh. 2017b. Pointing out SQL queries from text. Technical Report MSR-TR-2017-45.

Chenglong Wang, Kedar Tatwawadi, Marc Brockschmidt, Po-Sen Huang, Yi Mao, Oleksandr Polozov, and Rishabh Singh. 2018. Robust text-to-sql generation with execution-guided decoding.