Text to SQL with Recursive Decoding

By

Akash - ABS3211
Akbar - AIE6401
Anupam Tripathi - AAT1666
Jiuqi Xian - JXY9577
Nikita Raut - NBR1932

Contents Page number

***Abstract* - Machine Translation is the task of converting sequences from one language to another. SQL generation can be thought of as a sub-task of machine translations, where questions in natural language are translated to the equivalent SQL query. Some of the problems of the existing methods are poor generalizability to unseen data and learning to match the results without leveraging the structure of SQL language. We have implemented a model that exploits the SQL structure to construct a SQL syntax tree and parses it by recursive decoder calls.**

1. Introduction

The problem of Machine Translation has been around for many years. It refers to the process of converting text from one language to another. The origins of Machine Translation can be traced back to the 1930s where translation was done using an automatic bilingual dictionary. However, it has received much more attention in recent years due to the development of Natural Language Processing. The demand for powerful machine translation techniques has been on a rise in recent years, due to its ubiquitous usage in industries, making it a hot topic amongst researchers today. This technique is often used to convert one natural language to another, however, nowadays is being employed for much more complex tasks like code generation. One such code generation task that enthralls us is generating SQL queries from natural language questions, for easier retrieval of data from databases. Creating a SQL query from a natural language is very helpful for people who have limited technical background. By using simple phrases(text) the model should be able to retrieve the information from databases. Most existing models have achieved decent accuracies at this task by making use of encoder-decoder models with an attention mechanism. However, the evaluation scheme used by most of these models has a number of limitations [1]. One of the major limitations is poor generalizability to new programs and databases. Most datasets have queries in the test set which also appear in the training set, making the evaluation of many models flawed. We have tried to overcome this limitation using a recursive decoding approach [2]. The dataset used for our project is WikiSQL [3], which contains simple queries and uses different databases for training and testing. Due to the limited time and resources, we only focus on simple queries as opposed to [2] which focuses on complex nested queries as well. We have studied and implemented our model in three steps, thus adding granularity to each model 1) Seq to Seq without schema, 2) Seq to Seq with schema 3) Recursive Decoding

2. Related work

With the emergence of popular datasets like WikiSQL [3] and Spider [4], many algorithms have been proposed in the past, to address this problem. In addition to releasing the WikiSQL database, Seq2Sql [3] proposes a deep neural network model for translating questions to SQL queries which make use of policy-based reinforcement learning to optimize the ordering of conditions by rewarding the decoder. A sketch-based approach has been used by SQLNet [5] which generates SQL from the syntactic structure of the query by filling slots in the sketch using neural networks. The state-of-the-art method SyntaxSQLNet [2], exploits the structured nature of SQL queries to develop a SQL specific syntax tree. The decoding process is broken down into various modules, where each module is a decoder. The recursive decoding process works by each decoder calling another decoder which is next in line. We have employed this recursive decoding technique in our model.

3. Problem Definition

Given a natural language question posed to retrieve information from a large database, the task is to translate this question to an equivalent SQL query, such that the execution of that query provides the user with the desired results. We have focussed our work on implementing a model that generalizes well for unseen data by limiting the problem to simple queries.

4. Our Methods

In this section we present the models that we have implemented for our application. The first model that we implemented was a naive sequence to sequence encoder decoder model which took the question as input and generated the SQL query. We improved on this model by taking an additional input, the schema. Lastly, we made changes to the decoding step in the above model to incorporate for recursive decoding. The best results were obtained by the third model i.e using recursive decoding.

4.1 Seq to seq without schema

Since our task can be viewed as a translation task (from natural language to query language), we decide to use a naive encoder-decoder with attention architecture to serve as a baseline model. For the encoder we use a single layer bidirectional GRU unit whereas the decoder is a regular GRU unit. The first hidden state of the decoder is the concatenation of the last states of forward and backward encoder.

4.1.1 Attention

The output from the encoder contains encodings for the entire text input. For predicting a word in the SQL, certain words in the text input are more important than others. For example, for predicting the '>' sign in the where part of the SQL query, the words 'greater' and 'than' are more important and so must be weighted higher. For doing this, we use the attention mechanism [6]. Fig 1 shows the attention approach used with our encoder decoder. For each decoder, weights are calculated using encoder outputs, the current hidden state for that decoder unit, h_n and the inputs to the decoder.

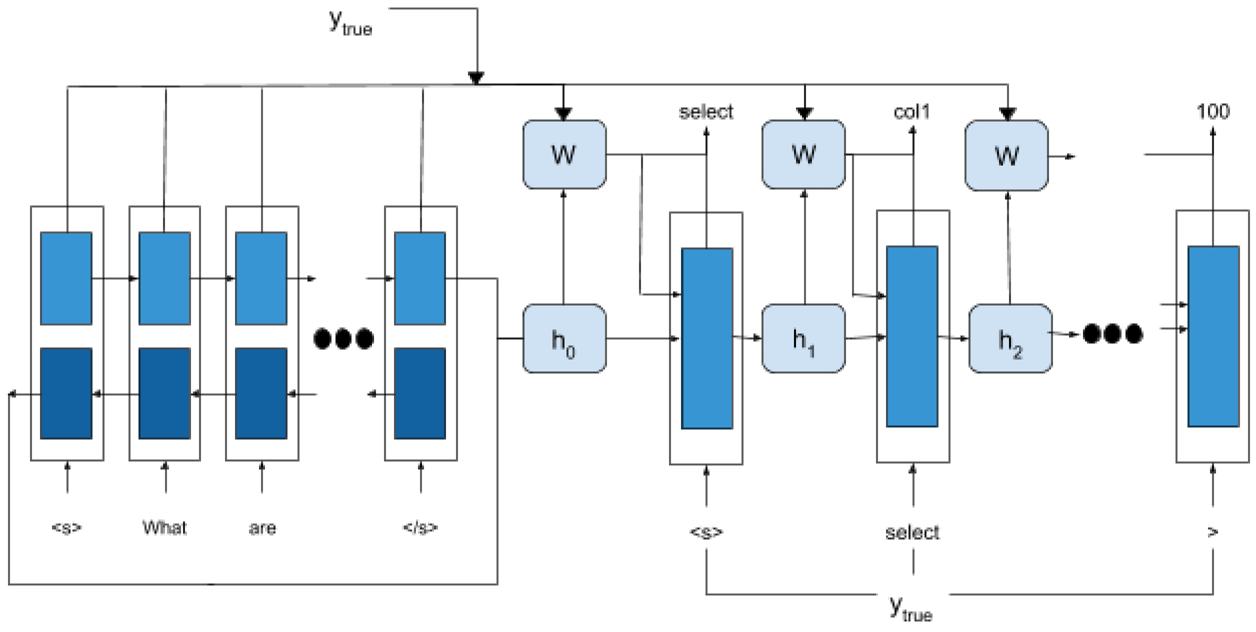

Fig 1. Encoder - Decoder Architecture for Seq to Seq translation

4.2 Seq to seq with schema

This model is a slight modification from the previous model. We use another encoder of the same structure to encode the schema, and concatenate the hidden states to those of the text encoder. Then we use attention on both encoders.

4.3 Recursive Decoding

This method, like our previous approaches, uses an encoder decoder architecture (Figure 3) for the given task. There are multiple encoders and decoders used, which have the same model, but differ only because the input they get and the order in which they are called.

4.3.1 Encoders

The encoders encode all the input to the network and pass the encodings to the decoders.
The encoding part is done by two encoders:

- **Text Encoder**: This encoder encodes the word vectors for the english text.
- **Schema Encoder**: This encoder is responsible for encoding the schema information of the database. We have used just the table names and column names from the schema information given in the WikiSQL dataset.

4.3.2 Decoders

The decoders receive all the encoding from the encoders. The decoding part is recursive, i.e. the decoder contains a call to itself. In terms of machine translation done in our previous approach, each of the RNN cell is a decoder in this method and is responsible for predicting a specific part of the SQL query., The decoding part is done by eight decoders (Figure 4):

- **Root Decode**r: This decoder is the one which is called first. It is not responsible for predicting any part of the sql query, but only calling the keyword decoder.

- **Keyword Decoder**: This decoder is responsible for predicting the keyword in the SQL languages. Since the WikiSQL dataset has limited keywords, this decoder predicts one of SELECT, FROM and WHERE. For a bigger dataset, this decoder should also be responsible for predicting other keywords like GROUP BY and HAVING. It can call the aggregator decoder, column decoder or the Table Decoder.

- **Column Decode**r: This decoder predicts the column name in the SQL query. The vocab of this decoder is all the column names present in the schema. It can call the Operator Decoder.

- **Table Decoder**: Similar to the previous decoder, this one predicts the table name and has the vocab of all the table names from the schema. It does not call any decoder.

- **Aggregator Decoder**: This decoder predicts the aggregator for columns. It can be one of COUNT, SUM, MAX, MIN and AVG. It does not call other decoders.

- **Operator Decoder**: It predicts the operator in the 'where' section of the query. It can predict one of =, !=, >, >=, <, <=, like etc. It calls the Constant Decoder after it.

- **Constant Decoder**: The where part of the query can have constants with whom the column values are compared to. These constants can be numbers or strings. The output vocab of this decoder is the entire input vocab size of our text encoder because the constant can literally be anything. It can call the And Or Decoder

- **And Or Decoder**: This decoder predicts the keywords 'AND' and 'OR' in the where part of the SQL query. It calls the Column Decoder after it.

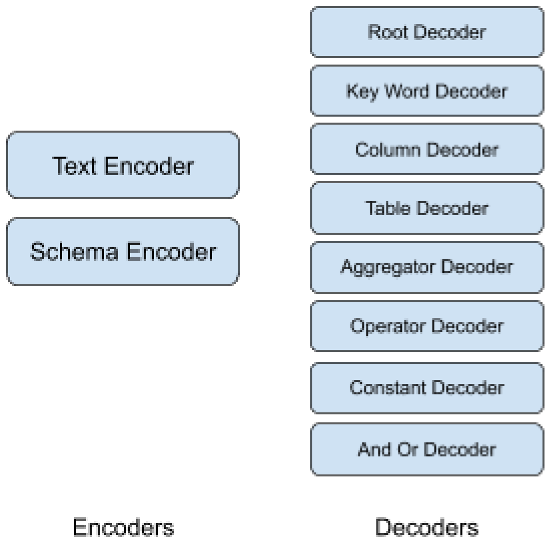

Fig 1. Encoders and Decoders

Fig 3 shows the decoding procedure. Consider the example query given in it. First the root decoder will be called. It will intern call the keyword decoder and this procedure goes on until the terminating condition is encountered. The training done is teacher forced, so the next decoder will have the right answer from the previous decoder.

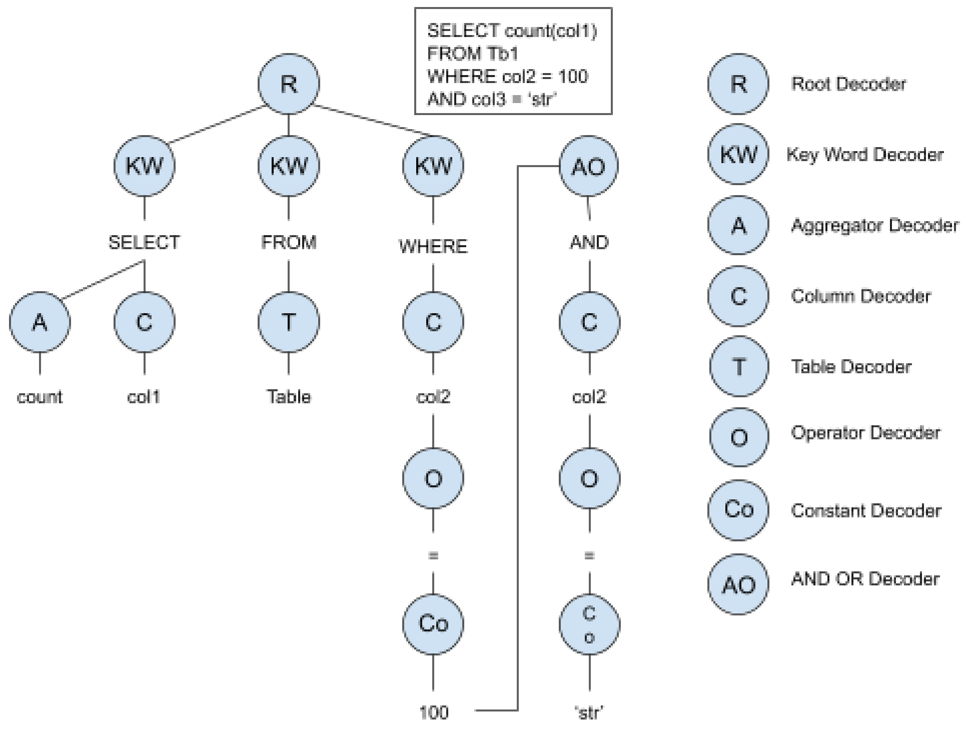

Fig 2. Decoding process

5. Experiments

5.1 Dataset

We have used the WikiSQL [link] dataset for our project. This dataset was released by salesforce as a part of their work Seq2Sql [1]. It is a large dataset that consists of 60,000 question-SQL query pairs and 18,000 tables in the train set. The test set comprises 16,000 question-SQL query pairs and about 5,000 tables in the test set which are different from those in the train set. The test set provided by WikiSQL has been divided by us into test and validation sets at 0.5 splits.

The reasons for using this dataset over the other available datasets are 1) The tables and databases used in the train set are not present in the test set. This ensures that the evaluation of our model on unseen data is accurate. 2) WikiSQL is a large dataset which does not include complex nested queries. Hence, it is perfect for our application which considers only simple queries.

WikiSQL provides files in json and db format. It consists of 3 json files - train test and dev, each file containing the questions and sql queries and another set of 3 json files for the tables information which contains the name of the table, columns in the table and the row values. Before using this data for our application, we had to convert the json data into sequence to sequence format, thus some amount of preprocessing was involved. The structure of *.json file was:

```
{
   "phase":1,
   "question":"who is the manufacturer for the order year 1998?",
   "sql":{
      "conds":[
        [
            0,
            0,
            "1998"
         ]
      ],
      "sel":1,
      "agg":0
   },
   "table_id":"1-10007452-3"
}
```

In order to convert sql json format to a sequence, the indexes provided in front of each element in the above example were used to find the references from the table json file eg: “sel” : 1 (the first element in “header” was referenced for the select clause) . The basic structure of table json file is provided below

```
{
   "id":,
   "header":,
   "types":,
   "rows":
}
```

5.2 Evaluation

The evaluation metrics used are component matching and exact matching. For component matching we have computed the scores for each module i.e Keyword, Column, Table, Aggreagtor, Operator, Constant, And-or. The reason for selecting this metric is that the baseline model uses this metric and hence better comparison can be provided. We compute the F1 scores for each component separately. F1 score combines precision and recall and can be calculated as

$$F1 = 2 \times (Precision \times Recall) / (Precision + Recall)$$

where

$$Precision = True\ Positive / (True\ Positive + False\ Positive)$$

$$Recall = True\ Positive / (True\ Positive + False\ Negative)$$

Precision is a good metric to use when the cost of false positives is high. It tells how many are actually positive from the predicted positive. On the other hand Recall is a good metric to be used when the cost of false negatives is high. F1 score tries to balance the two and hence is a better metric to be used when the classes are imbalanced. F1 score ranges from 0 to 1, with 1 being the score for a model with perfect accuracy. One of the disadvantages of F1 Scores is that the scores for each component are computed independently. It does not consider the mutual information between these components. Thus, we use a second evaluation i.e exact matching score which compares the entire translation with the label. If all the components match then then score is 1, otherwise the score is 0. An alternative to using exact matching would have been BLEU score. The BLEU score measures how many words overlap in the translated query when compared to the actual query. However, exact matching suits our application more since the translated query is of use only when the entire query is correct.

5.3 Experimental Settings

Our model was trained on GPU using the torch toolbox. We have used a GRU based encoder-decoder model. The embedding dimension used is 50. The dimension of the encoder hidden layer is 40 and that of the decoder hidden layer is 30. Adam optimizer has been used with default hyperparameters for optimization on a batch size of 1. Loss function used is Negative Log Likelihood.

6. Results and Analysis

6.1 Comparison with Previous Methods

We only include the results for our best model i.e recursive decoding model, since the results obtained from the other models were poor.

Method	Exact Match Score (on WikiSQL)
Seq2SQL	53.5 %
SQLNet	61.3 %
Our Model (Recursive Decoding)	23 %

Table 1 . Exact Matching Accuracies on different models

Alternative metric
BLEU Score = 30.13 %

6.2 Analysis

As seen from table 1, the exact match score of our model was on the lower side as compared to the state of the art methods, so we analysed the f1 scores for each decoder, results of which are shown in table 2.

Our Model	Keyword	Column	Table	Aggregator	Operator	Constant	And-Or
F1 Score	100 %	68.2 %	32.3 %	94.7 %	100 %	74.93 %	100 %

Table 2 .Component wise F1 scores for our model

It can be seen from the table that our model has failed miserably for table decoder. This can be due to poor labelling of tables in the WikiSQL dataset. WikiSQL uses id to identify tables uniquely. Eg - 1-2001-2. The model could have learnt the table names better, had they provided the names of actual tables in the database, since the natural language questions will never contain ids of tables. Another component which got lower accuracy was the Column module, which is understood given the number of epochs we trained for. Due to limited resources and time constraints we could only train for a few epoch, but the model has to learn the correct columns from millions of column names as opposed to other components that have to learn only from limited data (operator can only be one of >, <, >=, <=, =)

We didn't include the result in the table for our naive baseline methods because the exact matching accuracies are zero. This means that our naive methods are not suitable for this task because the generation is too unconstrained for the target which is a structured language with strict grammar.

7. Conclusion and Future work

In this paper, we have documented our implementation for sequence to SQL using three models, the best results of which were obtained by exploiting the SQL structure. We constructed a SQL Syntax tree and each module was an independent decoder. The decoding process can be viewed as recursive calls to these modules. The dataset used was WikiSQL.

Many adaptations and further experiments can be performed to improve the performance of this model. A pre-trained word embedding such as GloVe or BERT can be used instead of training the embedding from scratch. Future work concerns constructing syntax trees for nested complex queries. The model must also be tested on complex datasets like Spider.

References

[1] Finegan-Dollak, Catherine, et al. "Improving text-to-sql evaluation methodology." *arXiv preprint arXiv:1806.09029* (2018).

[2] Yu, Tao, et al. "Syntaxsqlnet: Syntax tree networks for complex and cross-domaintext-to-sql task." *arXiv preprint arXiv:1810.05237* (2018).

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

[4] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. arXiv preprint arXiv:1809.08887, 2018

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

[6] Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems. 2017.