Translating Natural Language to SQL using Pointer-Generator Networks and How Decoding Order Matters

Denis Lukovnikov¹, Nilesh Chakraborty¹, Jens Lehmann¹, and Asja Fischer²
¹University of Bonn, Bonn, Germany
²Ruhr University Bochum, Bochum, Germany

Abstract
Translating natural language to SQL queries for table-based question answering is a challenging problem and has received significant attention from the research community. In this work, we extend a pointer-generator and investigate the order-matters problem in semantic parsing for SQL. Even though our model is a straightforward extension of a general-purpose pointer-generator, it outperforms early works for WikiSQL and remains competitive to concurrently introduced, more complex models. Moreover, we provide a deeper investigation of the potential order-matters problem that could arise due to having multiple correct decoding paths, and investigate the use of REINFORCE as well as a dynamic oracle in this context.

Introduction
Semantic parsing, the task of converting Natural Language (NL) utterances to their representation in a formal language, is a fundamental problem in Natural Language Processing (NLP) and has important applications in Question Answering (QA) over structured data and robot instruction.

In this work, we focus on QA over tabular data, which attracted significant research efforts (Zhong et al. 2017; Xu et al. 2017; Yu et al. 2018; Huang et al. 2018; Haug et al. 2018; Wang et al. 2018a; 2018b; Shi et al. 2018; Krishnamurthy et al. 2017; Iyyer et al. 2017; Pasupat and Liang 2015). In this task, given a NL question and a table, the system must generate a query that will retrieve the correct answers for the question from the given table.

The model we use in this paper is a straightforward extension of pointer-generators, and yet outperforms early works and compares well against concurrently developed models. Concretely, we add simple LSTM-based column encoders, skip connections and constrained decoding, as elaborated later in the paper.

In translating NL questions to SQL queries, as in many semantic parsing tasks, target queries can contain unordered elements, resulting in multiple valid decoding paths. It has been shown by Vinyals et al. (2016) that in the case of multiple valid decoding paths, the order of sequences provided for supervision can affect the accuracy of SEQ2SEQ models. We provide a deeper investigation of the potential order-matters problem in translating NL to SQL that has been raised by previous work. In this context, we also investigate training with a dynamic oracle (Goldberg and Nivre 2012) as well as training with REINFORCE, both of which explore different possible linearizations of the target queries, and show that the use of a dynamic oracle can be beneficial when the original supervision sequences are ordered inconsistently.

In the following, we first introduce the problem, then describe our model and the training procedure, present an experimental analysis, and conclude with a comparison to related work.

Queries, Trees and Linearizations
As an illustration of table-based QA, consider the natural language question

“How much L1 Cache can we get with an FSB speed of 800MHz and a clock speed of 1.2GHz?”

This question should be mapped to the following SQL query

\[
\begin{align*}
\text{SELECT } & \text{L1Cache} \\
\text{WHERE } & \text{FSB Speed} = 800 \text{ (MHz)} \\
& \text{AND Clock Speed} = 1.0 \text{ (GHz)}
\end{align*}
\]

which will be executed over a table listing processors, the sizes of their caches, their clocks speeds etc. In the query representation format we use, the example SQL query will be represented as the following sequence of output tokens:

\[
\text{SELECT L1Cache WHERE FSB_Speed OP0 VAL 800 ENDVAL AND Clock_Speed OP0 VAL 1.0 ENDVAL },
\]

where AGG0 is a dummy “no aggregator” token that is used to indicate that no real aggregator should be applied and OP0 is the = (equality) operator. Other aggregators, like SUM and COUNT, and other operators, like < (less than) are also available.

As illustrated in Figure 1, the SQL query can also be represented as a tree where the root node has two children: SELECT and WHERE. Note that the order of the two conditions appearing in the WHERE clause is arbitrary and does not have any impact on the meaning of the query or the execution results. Trees containing such unordered nodes can be linearized into a sequence in different, equally valid, ways ("FSB Speed” first or "Clock Speed” first in the example, as...

¹This is an updated version of our previous anonymous version (https://openreview.net/forum?id=HJMoLws2z) from May 2018.
Figure 1: Example of a query tree. The blue arrows indicate unordered children.

Figure 2: Two valid linearizations of the example query tree in Figure 1.

Model

We start from a sequence-to-sequence model with attention and extend the embedding and output layers to better suit the task of QA over tabular data. In particular, we use on-the-fly (Bahdanau et al. 2017) embeddings and output vectors for column tokens and implement a pointer-based (Gu et al. 2016; See et al. 2017) mechanism for copying tokens from the question. The resulting model is a Pointer-Generator (Gu et al. 2016; See et al. 2017) mechanism for copying tokens from the output vocabulary.

The Seq2Seq Model

The general architecture of our model follows the attention-based sequence-to-sequence (SEQ2SEQ) architecture. The following formally introduces the major parts of our SEQ2SEQ model. Details about the embedding and the output layers are further elaborated in later sections.

The SEQ2SEQ model consists of an encoder, a decoder, and an attention mechanism.

Encoder We are given a question \( Q = [q_0, q_1, \ldots, q_N] \) consisting of NL tokens \( q_i \) from the set \( \mathcal{V}^E \) (i.e., the encoder vocabulary). The tokens are first passed through an embedding layer that maps every token \( q_i \) to its vector representation \( \mathbf{q}_i = \mathbf{W}^E \cdot \text{one_hot}(q_i) \) where \( \mathbf{W}^E \in \mathbb{R}^{1 \times d_E \times m_L} \) is a learnable weight matrix and \( \text{one_hot}(\cdot) \) maps a token to its one-hot vector.

Given the token embeddings, a bidirectional multi-layered Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber 1997) encoder produces the hidden state vectors \( [\mathbf{h}_0^*, \mathbf{h}_1^*, \ldots, \mathbf{h}_N^*] = \text{BiLSTM}([\mathbf{q}_0, \mathbf{q}_1, \ldots, \mathbf{q}_N]) \).

The encoder also contains skip connections that add word embeddings \( \mathbf{q}_i \) to the hidden states \( \mathbf{h}_i^* \).

Decoder The decoder produces a sequence of output tokens \( s_t \) from an output vocabulary \( \mathcal{V}^D \) conditioned on the input sequence \( Q \). It is realized by a uni-directional multi-layered LSTM. First, the previous output token \( s_{t-1} \) is mapped to its vector representation using the embedding function \( \mathbf{EMB}(\cdot) \). The embeddings are fed to a BiLSTM-based decoder and its output states are used to compute the output probabilities over \( \mathcal{V}^D \) using the output function \( \text{OUT}(\cdot) \). \( \mathbf{EMB}(\cdot) \) and \( \text{OUT}(\cdot) \) are described in the following sections.

Attention We use attention (Bahdanau et al. 2014) to compute the context vector \( \hat{\mathbf{h}}_i \), that is

\[
\begin{align*}
\alpha_{i}^{(t)} &= \mathbf{h}_i \cdot \mathbf{y}_t, \\
\alpha_{i}^{(t)} &= \text{softmax}(\alpha_{i}^{(t)}, \ldots, \alpha_{i}^{(t)}, \ldots, \alpha_{N}^{(t)})_i, \\
\hat{\mathbf{h}}_i &= \sum_{i=0}^{N} \alpha_{i}^{(t)} \mathbf{h}_i, 
\end{align*}
\]

where \( \text{softmax}(-) \) denotes the \( i \)-th element of the output of the softmax function and \( \mathbf{h}_1, \ldots, \mathbf{h}_N \) are the embedding vectors returned by the encoder.

Embedding Function of the Decoder

The whole output vocabulary \( \mathcal{V}^D \) can be grouped in three parts: (1) SQL tokens from \( \mathcal{V}^\text{SQL} \), (2) column ids from \( \mathcal{V}^\text{COL} \), and (3) input words from the encoder vocabulary \( \mathcal{V}^E \), that is, \( \mathcal{V}^D = \mathcal{V}^\text{SQL} \cup \mathcal{V}^\text{COL} \cup \mathcal{V}^E \). In the following paragraphs, we describe how each of the three types of tokens is embedded in the decoder.

SQL tokens: These are tokens which are used to represent the structure of the query, inherent to the formal target language of choice, such as SQL-specific tokens like \texttt{SELECT} and \texttt{WHERE}. Since these tokens have a fixed, example-independent meaning, they can be represented by their respective embedding vectors shared across all examples. Thus, the tokens from \( \mathcal{V}^\text{SQL} \) are embedded based on a learnable, randomly initialized embedding matrix \( \mathbf{W}^\text{SQL} \) which is reused for all examples.

Column id tokens: These tokens are used to refer to specific columns in the table that the question is being asked against.

Column names may consist of several words, which are first embedded and then fed into a single-layer LSTM. The final hidden state of the LSTM is taken as the embedding vector representing the column. This approach for computing column representations is similar to other works that encode external information to get better representations.
for rare words (Bahdanau et al. 2017; Ling et al. 2015; Hill et al. 2016).

Input words: To represent input words in the decoder we reuse the vectors from the embedding matrix \( W^E \), which is also used for encoding the question.

Output Layer of the Decoder
The output layer of the decoder takes the current context \( \hat{h}_t \) and the hidden state \( y_t \) of the decoder’s LSTM and produces probabilities over the output vocabulary \( \mathcal{V}^D \). Probabilities over SQL tokens and column id tokens are calculated based on a dedicated linear transformation, as opposed to the probabilities over input words which rely on a pointer mechanism that enables copying from the input question.

Generating scores for SQL tokens and column id tokens
For the SQL tokens (\( \mathcal{V}^{SQL} \)), the output scores are computed by the linear transformation: \( o^{SQL} = U^{SQL} \cdot [y_t, \hat{h}_t] \), where \( U^{SQL} \in \mathbb{R}^{\mathcal{V}^{SQL} \times d^{e^{out}}} \) is a trainable matrix. For the column id tokens (\( \mathcal{V}^{COL} \)), we compute the output scores based on a transformation matrix \( U^{COL} \), holding dynamically computed encodings of all column id tokens present in the table of the current example. For every column id token, we encode the corresponding column name using an LSTM, taking its final state by the linear transformation:

\[
\hat{h} = \text{linear transformation}(\text{column name})
\]

```
where the scalar mixture weight \( \gamma \) is given by the maximum attention score over all positions in \( \mathcal{V}^{COL} \) where \( \hat{y} \) and \( \hat{h} \) are trained together with the remaining model parameters.
```

Output of a two-layer feed-forward neural network, that gets the output scores for all column id tokens are then computed based on a single softmax function over the output vocabulary \( \mathcal{V}^D \) that do not occur in the question \( Q \) are set to 0.

Finally, the two distributions \( p^{GEN} \) and \( p^{PTR} \) are combined into a mixture distribution:

\[
p(S_t|s_t-1, \ldots, s_0, Q) = p^{GEN}(S_t|s_t-1, \ldots, s_0, Q) + (1 - \gamma)p^{PTR}(S_t|s_t-1, \ldots, s_0, Q),
\]

where the scalar mixture weight \( \gamma \in [0, 1] \) is given by the output of a two-layer feed-forward neural network, that gets \( [y_t, \hat{h}_t] \) as input.

Shared softmax: In this approach, we re-use the attention scores \( \alpha_i^{(t)} \) (Eq. 1) and obtain the output scores \( o^E \) over the tokens \( q \in \mathcal{V}^E \) from the question as follows: for every token \( q \) that occurs in the question sequence \( Q \) the output score is given by the maximum attention score over all positions in \( Q = [q_0, \ldots, q_t, \ldots, q_N] \) where \( q \) occurs, i.e. it is given by:

\[
\max_{i:q_i=q} \alpha_i^{(t)}
\]

while the scores for all input tokens \( q \in \mathcal{V}^E \) that do not occur in the question \( Q \) are set to \(-\infty\). The final output probabilities are then computed based on a single softmax function that takes the output scores of the whole output vocabulary as input:

\[
p(S_t|s_t-1, \ldots, s_0, Q) = \text{softmax}(o^{SQL} + o^{COL} + o^E).
\]

Pretrained Embeddings and Rare Words
We initialize all NL embedding matrices\(^3\) using Glove embeddings for words covered by Glove (Pennington et al. 2014) and use randomly initialized vectors for the remaining words. Whereas randomly initialized word embeddings are trained together with the remaining model parameters, we keep Glove embeddings fixed, since finetuning them led to worse results in our experiments.

We also replace rare words that do not occur in Glove with a rare word representation in all embedding matrices.

\[^3W^E \text{ simultaneously used for question word embedding in the encoder and input word embedding in the embedding function of the decoder, the embedding matrix } W^{CT} \text{ for words occurring in column names used in the embedding function of the decoder, and its analogue in the output function.}\]
Coherence of decoded logical forms

The output sequences produced by a unconstrained decoder can be syntactically incorrect and result in execution errors or they can make mistakes against table semantics. We avoid such mistakes by implementing a constrained decoder that exploits task-specific syntactic and semantic rules.

The grammar behind the produced sequences is simple and the constraints can be implemented easily by keeping track of the previous token and whether we are in the `SELECT` or `WHERE` clause. For example, after a `COND` token (see also Figure 1 and Section), only column id tokens (`ColumnID`, `FSB_Speed`, ...) can follow, and after a column id token, only an operator token (`OP1`, `OP2`, ...) is allowed if we are currently decoding the `WHERE` clause.

In addition to such syntactic rules, we take into account the types of columns to restrict the set of aggregators and operators that can follow a column id. In the case of WikiSQL, there are two column types: `text` and `float`. Aggregators like `average` and operators like `greater-than` only apply on `float`-typed columns and thus are not allowed after `text` columns. We also enforce span consistency when copying tokens, leaving only the choice of copying the next token from the input or terminating copying, if the previous action was a copy action.

Training

We train our models by maximizing the likelihood of correct logical forms given the natural language question. We experiment with teacher forcing (TF) and a dynamic oracle (Goldberg and Nivre 2012).

Teacher forcing takes the original linearizations of the query trees (as provided in the dataset) and uses it both for supervision and as input to the decoder. However, in the presence of different correct sequences (as resulting from different correct linearizations of a query tree), teacher forcing can suffer from suboptimal supervision order (Vinyals et al. 2016). This might also be the case for semantic parsing, as pointed out by previous works on WikiSQL (Zhong et al. 2017; Xu et al. 2017) and concurrently explored by (Shi et al. 2018).

Dynamic Oracle

Instead of forcing the model to follow the original decoding sequence, the dynamic oracle enables the exploration of alternative linearizations of the query tree and is an adaptation of Goldberg and Nivre’s (2012) dynamic oracle with spurious ambiguity. It is formally described in Algorithm 1, which is invoked at every decoding step \( t \) to get a token \( g_t \) (used for supervision) and a token \( x_{t+1} \) (used as input to the decoder in the next time step). Essentially, the algorithm always picks the best-scored correct token for supervision and uniformly samples one of the correct tokens to be used as decoder input in the next time step, if the overall best-scored token (over the whole output vocabulary) does not belong to the correct ones. Thus, the oracle explores alternative paths if the decoder would make a mistake in free-running mode.

\[
\begin{align*}
\text{Algorithm 1 Dynamic oracle} \\
1: \text{function} & \text{getNextAndGold}(p_t, t, x_{\leq t}) \\
2: & \quad \text{VNT}_t \leftarrow \text{get\_valid\_next}(t, x_{\leq t}) \\
3: & \quad x_{t+1} \leftarrow \text{argmax}_{p_t} g_t \\
4: & \quad g_t \leftarrow \text{argmax}_{\text{VNT}_t} p_t \\
5: & \quad \text{if } x_{t+1} \notin \text{VNT}_t \text{ then} \\
6: & \quad \quad x_{t+1} \leftarrow \text{random}(\text{VNT}_t) \\
7: & \quad \text{return } g_t, x_{t+1} \\
\end{align*}
\]

In the algorithm, \( p_t \) is the decoder’s output distribution over \( \mathcal{V}^D \) at time step \( t \). The set of valid next tokens \( \text{VNT}_t \subset \mathcal{V}^D \), from which the correct tree can be reached, is returned by the function \( \text{get\_valid\_next}() \). The query tree can have nodes with either ordered or unordered children (for example, children of the `WHERE` clause are unordered). If we are currently decoding the children of a node with unordered children, all the children that have not been decoded yet are returned as \( \text{VNT}_t \). In other cases, \( \text{VNT}_t \) contains the next token according to the original sequence order.

REINFORCE

The presented oracle is similar to REINFORCE in that it explores alternative paths to generate the same query. In contrast to the oracle, REINFORCE samples the next token \( x_{t+1} \) according to the predictive distribution \( p_t \) and then uses the sampled sequence to compute gradients for policy parameters:

\[
\nabla J = \mathbb{E}[\nabla \log(p_t(x_{t+1})) A_t]
\]

In Alg. 2, we adapt the oracle into a REINFORCE algorithm with episode reward \( A_t \) set to +1 if the sampled sequence produces a correct query and 0 otherwise.

\[
\begin{align*}
\text{Algorithm 2 REINFORCE-like oracle} \\
1: \text{function} & \text{getNextAndGold}(p_t, t, x_{\leq t}) \\
2: & \quad \text{VNT}_t \leftarrow \text{get\_valid\_next}(t, x_{\leq t}) \\
3: & \quad x_{t+1} \sim p_t; \ x_{t+1} \in \text{VNT}_t \\
4: & \quad g_t \leftarrow x_{t+1} \\
5: & \quad \text{return } g_t, x_{t+1} \\
\end{align*}
\]

Evaluation

To evaluate our approach, we obtain the WikiSQL dataset by following the instructions on the WikiSQL website.

Dataset

Each example in the dataset contains a NL question, its SQL equivalent and the table against which the SQL query should be executed. The original training/dev/test splits of WikiSQL use disjoint sets of tables with different schemas.

\[4^\text{We use these constraints during prediction only.}\]

\[5^\text{Teacher forcing can be seen as a static oracle.}\]

\[6^\text{http://github.com/salesforce/WikiSQL}\]
For details on the construction of the dataset and how it compares to existing datasets, we refer the reader to the WIKISQL paper (Zhong et al. 2017).

Experimental Setup

Evaluation: Similarly to previous works, we report (1) sequence match accuracy (Acc$_{LF}$), (2) query match accuracy (Acc$_{QM}$) and (3) query execution accuracy (Acc$_{EX}$). Note that while Acc$_{LF}$ accepts only the original linearizations of the trees, Acc$_{QM}$ and Acc$_{EX}$ accept all orderings leading to the same query.

Training details: After a hyperparameter search, we obtained the best results by using two layers both in the encoder and decoder LSTMs, with $d^{dec} = 600$ in every layer, and $d^{emb} = 300$ hidden neurons, respectively, and applying time-shared dropouts on the inputs of the recurrent layers (dropout rate 0.2) and recurrent connections (dropout rate 0.1). We trained using Adam, with a learning rate of 0.001 and a batch size of 100, a maximum of 50 epochs and early stopping. We also use label smoothing with a mixture weight $\epsilon = 0.2$, as described in Szegedy et al. (2016). We ran all reported experiments at least three times and report the average of the computed metrics. While the variance of the metrics varies between settings, it generally stays between 0.1 and 0.25 percent for Acc$_{QM}$.

Results

We present our results, compared to previous and concurrent work in Table 1. Our method compares well against previous works, achieving performance similar to Coarse2Fine (Dong and Lapata 2018) and close to MQAN (McCann et al. 2018), works, achieving performance similar to Coarse2Fine (Dong and Lapata 2018) and close to MQAN (McCann et al. 2018), as well as being close to MQAN (McCann et al. 2018), which have more complicated architectures. Approaches using execution-guided decoding (EG) show better performance at the expense of access to table content and repeated querying during decoding, and relies on the assumption that the query should not return empty result sets. The currently developed oracle-based approach of Shi et al. (2018) improves upon our investigation of the oracle using the ANYCOL technique (see Related Work section). However, the ANYCOL technique assumes knowledge of table content during training and therefore might require retraining when table contents change.

In the following sections, we provide an ablation study, an in-depth analysis of the influence of the linearization order of query trees, as well as an error analysis. The analysis reveals that the overall improvement in accuracy obtained from using the dynamic oracle can be attributed to improved prediction accuracy of WHERE clauses, which contain unordered elements.

Ablation study Starting from the best variant of our model (i.e. the shared softmax pointer-generator) and standard TF based training, we want to investigate the role of different model components and the different training approaches.

Table 2 presents the results of this ablation study. Without constraints enforcing the coherence of the decoded logical rule at test time, the results drop by 1.6%. While also using the constraints during training doesn’t deteriorate results much, it results in slower training.

Label smoothing (Szegedy et al. 2016) has a significant impact on performance. Label smoothing relaxes the target distribution and thus helps to reduce overfitting. While label smoothing improves the performance of both types of pointer-generators, it brings more benefit for the shared softmax version (2% vs 1.4% for point-or-generate).

Incorporating skip connections into the encoder and decoder of our model improved performance by 0.5%. This improvement is achieved because skip connections allow to bypass more complicated RNN encoding functions and make predictions based on the more low-level features provided by word embeddings possible.

Influence of order in supervision To investigate the influence of the order of the target sequences on the results, we trained our model with teacher forcing and experimented with (1) reversing the original order of conditions in the WHERE clause and (2) training with target sequences where we assigned a different random order to the conditions in every trial. The results indicate that the order of conditions in the linearization matters for the performance of TF based training to a small degree. Training with a randomly reassigned order of conditions in the WHERE clause results in a 2.5% drop in test accuracy. However, reversing the order of conditions does not affect the results.

Furthermore, we trained our model with REINFORCE as well as with the dynamic oracle. In both methods, the originally provided order of the target sequence does not matter. Using REINFORCE (indicated by “RL” in Table 3) results in a 1.5% drop in performance.

The dynamic oracle as described in Alg. 1 provides an improvement of 0.5%. We can also see that Acc$_{LF}$ for the oracle is significantly lower compared to TF while Acc$_{QM}$ is on par with TF. Given that Acc$_{LF}$ is sensitive to the order of arbitrarily ordered clauses and Acc$_{QM}$ is not, this means that the oracle-trained models effectively learned to use alternative paths.

Comparing the oracle to TF with arbitrarily reordered conditions in the WHERE clause shows that when the supervision sequences are not consistently ordered, training with TF can suffer. When training with the oracle, the order of unordered nodes as provided in supervision sequences does not matter. Thus, it can be beneficial (in this case by 3% query accuracy) to use the oracle if the original linearization is arbitrary and can not be made consistent.

Error analysis Table 4 shows a breakdown of the causes of errors over the development set of WIKIQL. The numbers shown are percentages of error cases relative to the parent category. For example, in 36.5% of cases where the SELECT clause had an error, the column predicted in the SELECT clause was wrong. From the presented numbers,

---

1We also investigated dynamic oracles in the previous unpublished version of this work from May 2018 (https://openreview.net/forum?id=BjMoLws2z).

2Note that the percentages do not sum up to one since all components of a predicted clause can be wrong (which is indicated by
Table 1: Evaluation results for our approach (middle section) and comparison with previously reported results (top part) and concurrent work or EG-based systems (bottom part). Note that *Seq2SQL is the reimplementation of Seq2SQL (Zhong et al. 2017) by SQLNet authors (Xu et al. 2017). Some values in the table, indicated by “–”, could not be filled because the authors did not report the metric or the metric was not applicable.

Table 2: Performance of different variations of our approach.

we can conclude that the main cause of a wrongly predicted SELECT clause is an error in the predicted aggregator, while the main cause of error overall is the prediction of the WHERE clause. Comparison of errors of a models trained with TF versus Oracle reveals that oracle-trained models makes fewer mistakes in the WHERE clause, showing a 1% improvement in WHERE clause accuracy, which is translated to the 0.5% improvement in overall accuracy as observed in Table 1.

Related Work

Earlier works on semantic parsing relied on CCG and other grammars (Zettlemoyer and Collins 2007; Berant et al. 2013). With the recent advances in recurrent neural networks and attention (Bahdanau et al. 2014; See et al. 2017), neural translation based approaches for semantic parsing have been developed (Dong and Lapata 2016; Liang et al. 2016; Rabinovich et al. 2017).

Labels provided for supervision in semantic parsing datasets can be given either as execution results or as an executable program (logical form). Training semantic parsers on logical forms yields better results than having only the execution results (Yih et al. 2016) but requires a more elaborate data collection scheme. Significant research effort has been dedicated to train semantic parsers only with execution results. Using policy gradient methods (such as REINFORCE) is a common strategy (Liang et al. 2016; Zhong et al. 2017). Alternative methods (Krishnamurthy et al. 2017; Iyyer et al. 2017; Guu et al. 2017) exist, which also maximize the likelihood of the execution results.

Similar to the WikiSQL dataset that we used in our experiments are the ATIS and WIKITABLEQUESTIONS (Pasupat and Liang 2015) datasets, which also focus on question answering over tables. In contrast to WikiSQL how-
ever, both ATIS and WikiTableQuestions are significantly smaller and the latter does not provide logical forms for supervision and thus requires training with execution results as supervision (Neelakantan et al. 2016; Haug et al. 2018; Krishnamurthy et al. 2017). SQA (Iyyer et al. 2017) focuses on question answering in a dialogue context.

Previous works on WikiSQL (Zhong et al. 2017; Xu et al. 2017; Huang et al. 2018; Yu et al. 2018; Wang et al. 2018a) generally incorporate both slot-filling and sequence decoding, predicting the SELECT clause arguments with separate slot-filling networks, and also include some form of a pointing mechanism. Seq2SQL (Zhong et al. 2017) proposes an augmented pointer network that also uses a pointer but encodes the question, column names and SQL tokens together, and completely relies on a pointer to generate the target sequence. To avoid the potential order-matters problem, SQLNet (Xu et al. 2017) proposes a sequence-to-set model that makes a set inclusion prediction for the clauses in order to avoid decoding the conditions in any particular order. Both predict the SELECT clause arguments using separate specialized predictors. Zhong et al. (2017) also use Dong and Lapata (2016)’s SEQ2SEQ model as a baseline, however, get poor performance due to the lack of a pointer and column encoders. Yu et al. (2018) build upon SQLNet (Xu et al. 2017)’s slot filling approach, proposing several improvements such as weight sharing between SQLNet’s subnetworks, and incorporate precomputed type information for question tokens in order to obtain a better question encoding. Wang et al. (2018a) develop a model similar to ours; they propose a SEQ2SEQ model with copy actions. Similarly to Zhong et al. (2017), they encode the concatenation of column names and the question. Similarly to our work, they use a constrained decoder to generate SQL tokens or copy column names or question words from the encoded input sequence. In contrast to Wang et al. (2018a), we encode column names separately, and independently from the question. Huang et al. (2018) experiment with meta-learning (MAML), using Wang et al. (2018a)’s model. Course2Fine (Dong and Lapata 2018) explores a middle ground between purely sequence and tree decoding models (Alvarez-Melis and Jaakkola 2016; Dong and Lapata 2016) and proposes a two-stage decoding process, where first a template (sketch) of the query is decoded and subsequently filled in.

Table 3: Results under different ways of linearizing the target query trees.

|                | Dev Acc (%) | Test Acc (%) |
|----------------|-------------|--------------|
|                | Acc_{LF}    | Acc_{QM}     | Acc_{LF} | Acc_{QM} |
| Original order (TF) | 70.2 | 72.6 | 69.9 | 72.1 |
| · Reversed (TF) | – | 72.6 | – | 72.1 |
| · Arbitrary (TF) | – | 70.4 | – | 69.6 |
| · RL | 59.9 | 71.4 | 59.1 | 70.6 |
| · Oracle | 56.2 | 73.4 | 55.0 | 72.7 |

Table 4: Errors for the TF-trained and oracle-trained models with shared softmax. The reported numbers indicate percentages relative to the parent category (see text for details). Percentages are derived from error counts averaged across different runs of the same setting.

Very recent and concurrent work on WikiSQL (Wang et al. 2018b; Shi et al. 2018) explores execution-guided (EG) decoding (Wang et al. 2018b) and dynamic oracles (Shi et al. 2018). Execution-guided decoding keeps a beam of partially decoded queries, which are filtered based on the execution, that is, if the partially decoded query can not be parsed, produces a runtime error or returns an empty result set (if we expect a non-empty output). However, this technique requires multiple queries to be executed against the database while decoding and thus is sensitive to the current state of the database. A significant part of improvement obtained by EG decoding relies on the assumption that result sets should be non-empty. IncSQL (Shi et al. 2018) also uses EG decoding, as well as a dynamic oracle extended with the ANYCOL technique, which adds the option to produce a wildcard column token that matches any column. During training, the wildcard column token is provided as an alternative to the true column token in the supervision sequence if it can be unambiguously resolved to the true column using the condition value. This makes the training process dependent on table contents and thus might result in a need to retrain when the table contents change. IncSQL’s model goes beyond ours by adding self- and cross-serial attention and a final inter-column BiLSTM encoder. In addition, they also feed column attention and question attention summaries as an input to the decoder.

Conclusion

In this work we present a SEQ2SEQ model adapted to the semantic parsing task of translating natural language questions to queries over tabular data. We investigated the order-matters problem, concluding that the order of conditions in the linearization of the query tree matters to small but significant degree for WikiSQL. In this context, we also evaluated the use of REINFORCE and a dynamic oracle for training the neural network-based semantic parser. Our experiments revealed that REINFORCE does not improve results and the oracle provides a small improvement, which can be attributed to improved decoding of the WHERE clause. Furthermore, from the results we can conclude that training with a dynamic oracle is advisable if the original linearizations are inconsistently ordered.
The natural language decathlon: Multitask learning as question answering. arXiv preprint arXiv:1806.08730.

McCann, B.; Keskar, N. S.; Xiong, C.; and Socher, R. 2018. The natural language decathlon: Multitask learning as question answering. arXiv preprint arXiv:1806.08730.