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

Denis Lukovnikov¹, Nilesh Chakraborty¹, Jens Lehmann¹², and Asja Fischer³
¹Smart Data Analytics Group, University of Bonn, Bonn, Germany
²Enterprise Information Systems, Fraunhofer IAIS, St. Augustin, Germany
³Ruhr 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 network and investigate how query decoding order matters in semantic parsing for SQL. Even though our model is a straightforward extension of a general-purpose pointer-generator, it outperforms early work for WikiSQL and remains competitive to concurrently introduced, more complex models. Moreover, we provide a deeper investigation of the potential “order-matters” problem due to having multiple correct decoding paths, and investigate the use of REINFORCE as well as a non-deterministic oracle in this context.¹

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 work 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 order to use sequence decoders for generating queries, the queries must first be linearized to sequences which can be used to train a sequence decoder. However, when translating NL questions to SQL queries, as in many semantic parsing tasks, target queries can contain unordered elements, which results in multiple valid decoding paths. The particular ordering of the unordered elements in the decoding path used for supervision can affect the performance of the trained 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 non-deterministic 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 non-deterministic 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 1.0GHz?”

This question should be mapped to the following SQL query

```
SELECT L1.Cache WHERE FSB.Speed = 800 (MHz)
AND Clock.Speed = 1.0 (GHz)
```

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:

```
SELECT L1.Cache AGG0 WHERE COND
FSB.Speed OP0 VAL 800 ENDVAL COND
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
We are given a question and an attention mechanism. The general architecture of our model follows the attention-based decoder and its output states are used to compute the output probabilities over $\mathcal{Y}^D$ using the output function $\text{OUT}(\cdot)$. EMB(·) and OUT(·) are described in the following sections.

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

\[
\begin{align*}
a^{(t)}_i &= h_i \cdot y_t , \\ a^{(t)}_{i} &= \text{softmax}(a^{(t)}_0, \ldots, a^{(t)}_i, \ldots, a^{(t)}_N)i , \\
\hat{h}_t &= \sum_{i=0}^{N} a^{(t)}_i h_i ,
\end{align*}
\]

where $\text{softmax}(\cdot)_i$ denotes the $i$-th element of the output of the softmax function, $y_t$ is the output state of the decoder and $h_1, \ldots, h_N$ are the embedding vectors returned by the encoder.

**Embedding Function of the Decoder**

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

**SQL tokens:** These tokens are which are used to represent the structure of the query, inherent to the formal target language of choice, such as SQL-specific tokens like SELECT and 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 $\mathcal{Y}^{\text{SQL}}$ are embedded based on a learnable, randomly initialized embedding matrix $W^{\text{SQL}}$ 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 work 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 $V^D$. 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 ($V^S$), the output scores are computed by the linear transformation: $o^S = U^S \cdot [y_t, \hat{h}_t]$, where $U^S \in \mathbb{R}^{|V^S| \times d_{out}}$ is a trainable matrix. For the column id tokens ($V^{COL}$), we compute the output scores based on a transformation matrix $U^{COL}$, holding dynamically computed encodings of all column ids 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 encoding of all column ids present in the table of the current example. By using skip connections we compute the average of the word embeddings of the tokens in the column name, $c_i$ for $i = 1, \ldots, K$, and add them to the preliminary column name encoding $u^*$ to obtain the final encoding for the column id:

$$u = u^* + \frac{1}{K} \sum_{i=1}^{K} c_i,$$

where we pad the word embeddings with zeros to match the dimensions of the encoding vector before adding.

The output scores for all column id tokens are then computed by the linear transformation:

$$o^{COL} = U^{COL} \cdot [y_t, \hat{h}_t].$$

**Pointer-based copying from the input**

To enable our model to copy tokens from the input question, we follow a pointer-based (Gu et al. 2016; See et al. 2017) approach to compute output scores over the words from the question. We explore two different copying mechanisms, a shared softmax approach inspired Gu et al. (2016) and a point-or-generate method similar to See et al. (2017). The two copying mechanisms are described in the following.

**Point-or-generate:** First, the concatenated output scores for SQL and column id tokens are turned into probabilities

$$p_{GEN}(S_t|s_{t-1}, \ldots, s_0, Q) = \text{softmax}(\langle o^S; o^{COL} \rangle).$$

Then we obtain the probabilities over the input vocabulary $V^E$ based on the attention probabilities $a_i^{\langle t \rangle}$ (Eq. 2) over the question sequence $Q = [q_0, \ldots, q_i, \ldots, q_N]$. To obtain the pointer probability for a token $q_i$ in the question sequence we sum over the attention probabilities corresponding to all the positions of $Q$ where $q$ occurs, that is,

$$p_{PTR}(q|s_{t-1}, \ldots, s_0, Q) = \sum_{i:q_i=q} a_i^{\langle t \rangle}.$$  

The pointer probabilities for all input tokens $q \in V^E$ 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:

$$p(S_t|s_{t-1}, \ldots, s_0, Q) = \gamma p_{PTR}(S_t|s_{t-1}, \ldots, s_0, Q) + (1 - \gamma) p_{GEN}(S_t|s_{t-1}, \ldots, s_0, Q),$$

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 $a_i^{\langle t \rangle}$ (Eq. 1) and obtain the output scores $o^E$ over the tokens $q \in V^E$ from the question as follows: for every token $q_i$ that occurs in the question sequence $Q$ the output score is given by the maximum attention score over all positions in $Q = [q_0, \ldots, q_i, \ldots, q_N]$ where $q$ occurs, i.e., it is given by:

$$\max_{i:q_i=q} a_i^{\langle t \rangle},$$

while the scores for all input tokens $q \in V^E$ 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:

$$p(S_t|s_{t-1}, \ldots, s_0, Q) = \text{softmax}(\langle o^S; o^{COL}; o^E \rangle).$$

**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.

\(^3\)The embedding matrix $W^E$ simultaneously used for question word embedding in the encoder and input word embedding in the embedding function of the decoder, the embedding matrix $W^{COL}$ 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 an 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. In our example discussed earlier (see Figure 1), after a COND token, only column id tokens (L1.Cache, 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 a correct logical form given the natural language question. We experiment with teacher forcing (TF) and a non-deterministic 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 multiple correct decoding paths, teacher forcing can suffer from suboptimal supervision order, as pointed out by previous work on WikiSQL (Zhong et al. 2017; Xu et al. 2017) and concurrently explored by (Shi et al. 2018).

Non-deterministic Oracle

Instead of forcing the model to follow the original decoding sequence, a non-deterministic oracle enables the exploitation of alternative linearizations of the query tree and is an adaptation of Goldberg and Nivre’s (2012) algorithm for a dynamic oracle with spurious ambiguity (developed in the context of dependency parsing). It is formally described in Algorithm 1, which is invoked at every decoding step 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.

Algorithm 1 Non-deterministic oracle

1: function GENEXTAPNGOLD($p_t, t, x_{\leq t}$)
2: $VNT_t \leftarrow$ get_valid_next($t, x_{\leq t}$)
3: $x_{t+1} \leftarrow \text{argmax}_{D} p_t$
4: $g_t \leftarrow \text{argmax}_{VNT_t} p_t$
5: if $x_{t+1} \notin UNT_t$ then
6: $x_{t+1} \leftarrow \text{random}(VNT_t)$
7: return $g_t, x_{t+1}$

In the algorithm, $p_t$ is the decoder’s output distribution over $Y^D$ at time step $t$. The set of valid next tokens $VNT_t \subset Y^D$, from which the correct tree can be reached, is returned by the function 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 $VNT_t$. In other cases, $VNT_t$ 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:

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

(10)

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

Algorithm 2 Our REINFORCE

1: function GENEXTAPNGOLD($p_t, t, x_{\leq t}$)
2: $VNT_t \leftarrow$ get_valid_next($t, x_{\leq t}$)
3: $x_{t+1} \sim p_t; x_{t+1} \in VNT_t$
4: $g_t \leftarrow x_{t+1}$
5: return $g_t, x_{t+1}$

Evaluation

To evaluate our approach, we obtain the WikiSQL (Zhong et al. 2017) dataset by following the instructions on the WikiSQL website. The dataset contains a total of 80654 examples. Each example provides a NL question, its SQL (Zhong et al. 2017) and concurrently explored by (Shi et al. 2018). We use these constraints during prediction only.

Algorithm 1 Non-deterministic oracle

1: function GENEXTAPNGOLD($p_t, t, x_{\leq t}$)
2: $VNT_t \leftarrow$ get_valid_next($t, x_{\leq t}$)
3: $x_{t+1} \leftarrow \text{argmax}_{D} p_t$
4: $g_t \leftarrow \text{argmax}_{VNT_t} p_t$
5: if $x_{t+1} \notin VNT_t$ then
6: $x_{t+1} \leftarrow \text{random}(VNT_t)$
7: return $g_t, x_{t+1}$

Evaluation: Similarly to previous work, we report (1) sequence match accuracy ($Acc_{LF}$), (2) query match accuracy

Experimental Setup

Evaluation: Similarly to previous work, we report (1) sequence match accuracy ($Acc_{LF}$), (2) query match accuracy
(AccQM) and (3) query execution accuracy (AccEX). Note that while AccLF accepts only the original linearizations of the trees, AccQM and AccEX 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 every layer of size 600, and embedding size of 300, 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 AccQM.

**Results**

We present our results, compared to previous and concurrent work in Table 1. Our method compares well against previous work, achieving performance similar to Coarse2Fine (Dong and Lapata 2018) and 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 concurrently developed oracle-based\(^6\) approach of Shi et al. (2018) improves upon our investigation of the oracle using the ANYCOL technique (see Related Work section).

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 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% AccQM on the test set. 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 versions of pointer-generators, it improves the shared softmax version by 2% test AccQM, as opposed to a slightly lower improvement of 1.4% for point-or-generate.

Incorporating skip connections into the encoder and decoder of our model improved performance by 0.5% AccQM on the test set.

**Effect of ordering in supervision** To investigate the influence of the order in the linearizations of queries, 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 certain degree. Training with a randomly reassigned order of conditions in the WHERE clause results in a 2.5% drop in query accuracy (AccQM) on the test set. However, reversing the order of conditions does not affect the results.

Furthermore, we trained our model with REINFORCE as well as with the non-deterministic 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 AccQM on the test set. The oracle as described in Alg. 1 results in an improvement of 0.6% query accuracy on the test set. We can also see that AccLF for the oracle is significantly lower compared to TF while AccQM is on par with TF. Given that AccLF is sensitive to the order of arbitrarily ordered clauses and AccQM 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 training with TF can suffer from supervision sequences that are not consistently ordered. 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 accuracies of different parts of the query over the development set of WikiSQL. 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 models trained with TF versus oracle reveals that oracle-trained models make fewer mistakes in the WHERE clause, showing a 1% improvement (84.4% from 83.4%) in WHERE clause accuracy, which is translated to the 0.8% (73.4% from 72.6%) improvement in full query accuracy (AccQM) on the validation set.

We find no difference between the accuracies for the SELECT clause between TF and oracle training settings. In both cases, 68.7% of examples with wrongly predicted SELECT clauses had an error in the predicted aggregator, and 36.5% had a wrongly selected column.

**Related Work**

Earlier work on semantic parsing relied on CCG and other grammars (Zettlemoyer and Collins 2007; Berant et al.
al. (2016) also investigates the effect of ordering in the lin-

mimize the likelihood of the execution results.

Iyyer et al. 2017; Guu et al. 2017) exist, which also maxi-

mize the likelihood of the execution results. Using policy gradient methods (such as REIN- 

FORCE) is a common strategy (Liang et al. 2016; Zhong et 


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 executable program (logical form). Training semantic parsers

datasets can be given either as execution results or as an ex-

cutable program (logical form). Training semantic parsers

datasets can be given either as execution results or as an ex-

cutable program (logical form). Training semantic parsers

datasets can be given either as execution results or as an ex-

cutable program (logical form). Training semantic parsers

datasets can be given either as execution results or as an ex-

cutable program (logical form). Training semantic parsers

datasets can be given either as execution results or as an ex-

cutable program (logical form). Training semantic parsers

datasets can be given either as execution results or as an ex-

cutable program (logical form). Training semantic parsers

with ordered elements.

earization of target structures with unordered elements. We

adapt the approach of Goldberg and Nivre (2012) that was
developed in the context of dependency parsing. Goldberg

and Nivre (2012) experiment with two versions of the dy-
namic oracle, one that handles spurious ambiguity, and one

that is also able to recover from incorrect actions after which

the gold tree can not be reached.

Similar to the WikiSQL dataset that we used in our ex-
periments are the ATIS (Dahl et al. 1994) and WikiTABLE-
QUESTIONS (Pasupat and Liang 2015) datasets, which also
focus on question answering over tables. In contrast to Wik-

iSQL however, both ATIS and WikiTABLEQUESTIONS

are significantly smaller and the latter does not provide log-


ical 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) is a dataset derived from WikiTABLEQUESTIONS

and focuses on question answering in a dialogue context.

Previous work on WikiSQL (Zhong et al. 2017; Xu et

2017). 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.

Related to the ordering issue, the work of Vinyals et al. (2016) also investigates the effect of ordering in the lin-

earization of target structures with unordered elements. We adapt the approach of Goldberg and Nivre (2012) that was developed in the context of dependency parsing. Goldberg and Nivre (2012) experiment with two versions of the dynamic oracle, one that handles spurious ambiguity, and one that is also able to recover from incorrect actions after which the gold tree can not be reached.

Similar to the WikiSQL dataset that we used in our experiments are the ATIS (Dahl et al. 1994) and WikiTABLEQUESTIONS (Pasupat and Liang 2015) datasets, which also focus on question answering over tables. In contrast to WikiSQL however, 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) is a dataset derived from WikiTABLEQUESTIONS and focuses on question answering in a dialogue context.

Previous work on WikiSQL (Zhong et al. 2017; Xu et

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). Entries marked by * are trained and evaluated using a slightly different version of the WikiSQL dataset. 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>
<thead>
<tr>
<th>Method</th>
<th>Dev Acc (%)</th>
<th>Test Acc (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq2SQL (no RL) (Zhong et al. 2017)</td>
<td>48.2</td>
<td>57.1</td>
</tr>
<tr>
<td>Seq2SQL (RL) (Zhong et al. 2017)</td>
<td>49.5</td>
<td>59.4</td>
</tr>
<tr>
<td>Pointer-SQL (Wang et al. 2018a)</td>
<td>59.6</td>
<td>65.1</td>
</tr>
<tr>
<td>SQLNet (Xu et al. 2017)</td>
<td>–</td>
<td>61.3</td>
</tr>
<tr>
<td>PT-MAML (Huang et al. 2018)*</td>
<td>63.1</td>
<td>68.0</td>
</tr>
<tr>
<td>TypeSQL (Yu et al. 2018)*</td>
<td>– 68.0</td>
<td>73.5</td>
</tr>
<tr>
<td>STAMP (Sun et al. 2018)*</td>
<td>61.7</td>
<td>74.6</td>
</tr>
<tr>
<td>Coarse2Fine (Dong and Lapata 2018)</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>MQAN (McCann et al. 2018)</td>
<td>–</td>
<td>80.4</td>
</tr>
<tr>
<td>(ours)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PtrGen-SQL (shared softmax)</td>
<td>70.2</td>
<td>78.4</td>
</tr>
<tr>
<td>PtrGen-SQL (point-or-generate)</td>
<td>70.0</td>
<td>78.0</td>
</tr>
<tr>
<td>PtrGen-SQL (shared softmax) + oracle</td>
<td>56.2</td>
<td>78.8</td>
</tr>
<tr>
<td>(EG-based or concurrent work)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pointer-SQL + EG(5) (Wang et al. 2018b)</td>
<td>67.5</td>
<td>78.3</td>
</tr>
<tr>
<td>Coarse2Fine + EG(5) (Wang et al. 2018b)</td>
<td>76.0</td>
<td>83.8</td>
</tr>
<tr>
<td>IncSQL + oracle + ANYCOL (Shi et al. 2018)</td>
<td>49.9</td>
<td>83.7</td>
</tr>
<tr>
<td>IncSQL + oracle + ANYCOL + EG(5) (Shi et al. 2018)</td>
<td>51.3</td>
<td>87.1</td>
</tr>
</tbody>
</table>

Table 2: Performance of different variations of our approach.

<table>
<thead>
<tr>
<th>Method</th>
<th>Dev Acc (%)</th>
<th>Test Acc (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PtrGen-SQL (shared softmax)</td>
<td>70.2</td>
<td>72.1</td>
</tr>
<tr>
<td>· no constraints</td>
<td>68.6</td>
<td>70.5</td>
</tr>
<tr>
<td>· using constraints during training</td>
<td>69.8</td>
<td>71.9</td>
</tr>
<tr>
<td>· no label smoothing</td>
<td>68.3</td>
<td>70.1</td>
</tr>
<tr>
<td>· no label smoothing (point-or-generate)</td>
<td>68.7</td>
<td>70.3</td>
</tr>
<tr>
<td>· no skip connections</td>
<td>69.6</td>
<td>71.6</td>
</tr>
</tbody>
</table>
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 order-matters problem, SQLNet (Xu et al. 2017) proposes a sequence-to-set model that makes a set inclusion prediction 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 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. STAMP (Sun et al. 2018) presents a “multi-channel” decoder that considers three types of tokens (SQL, column, cell), and mixes the distributions for each type using coefficients produced by a trainable subnetwork. Compared to STAMP, we do not encode cells (we assume no knowledge of table contents) and instead use a pointer to copy values of conditions from the input. Coarse2Fine (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.

Very recent and concurrent work on WIKI SQL explores execution-guided (EG) decoding (Wang et al. 2018b) and non-deterministic oracles (Shi et al. 2018). Execution-guided decoding keeps a beam of partially decoded queries, which are filtered based on the execution results, that is, a partially encoded query is not taken further into account if it can not be parsed, produces a runtime error, or returns no results after execution. This requires multiple queries to be executed against the database while decoding. In our work, we try to avoid parsing and semantics-related runtime errors more efficiently by using decoding constraints. We suspect that a significant part of the improvement due to EG in decoding relies on the assumption that execution results should not be empty. However, we believe this assumption does not hold in general, due to the existence of queries for which an empty set is the correct answer. IncSQL (Shi et al. 2018) also uses EG decoding, as well as a non-deterministic oracle extended with the ANYCOL token, 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. IncSQL’s model goes beyond ours by adding self- and cross-serial attention and a final inter-column BiLSTM encoder. 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 how the ordering of supervision sequences during training affects performance, concluding that the order of conditions in the linearization of the query tree matters to a certain degree for WIKI SQL. In this context, we also evaluated the use of REINFORCE and a non-deterministic 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 non-deterministic oracle is advisable if the original linearizations are inconsistently ordered.

### Acknowledgement

We acknowledge the support of the European Union H2020 framework ITN projects WD Aqua (grant no. 642795) and Cleopatra (grant no. 812997).

<table>
<thead>
<tr>
<th>Test Accs (%)</th>
<th>TF</th>
<th>oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whole Query</td>
<td>72.6</td>
<td>73.4</td>
</tr>
<tr>
<td>· SELECT</td>
<td>85.5</td>
<td>85.5</td>
</tr>
<tr>
<td>· Aggregator</td>
<td>90.0</td>
<td>90.0</td>
</tr>
<tr>
<td>· Column</td>
<td>94.7</td>
<td>94.7</td>
</tr>
<tr>
<td>· WHERE</td>
<td>83.4</td>
<td>84.4</td>
</tr>
</tbody>
</table>

Table 4: Error Analysis: Acc\textsubscript{QM} of different query parts on the development set for TF and oracle-trained shared softmax models.
References
Wang, C.; Brockschmidt, M.; and Singh, R. 2018a. Pointing out sql queries from text.