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

Denis Lukovnikov[1], Nilesh Chakraborty[1], Jens Lehmann[1,2], and Asja Fischer[3]
[1]Smart Data Analytics Group, University of Bonn, Bonn, Germany
[2]Enterprise Information Systems, Fraunhofer IAIS, St. Augustin, Germany
[3]Ruhr 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

[1]This is an updated version of our previous anonymous version (https://openreview.net/forum?id=HJMoLws2z) from May 2018.

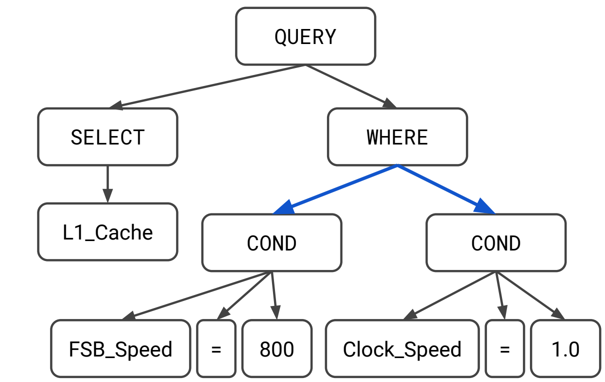

Figure 1: Example of a query tree. The blue arrows indicate unordered children.

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 illsustrated in Figure 2.). We refer to the linearization where the original order as given in the data set is preserved as the *original linearization*.

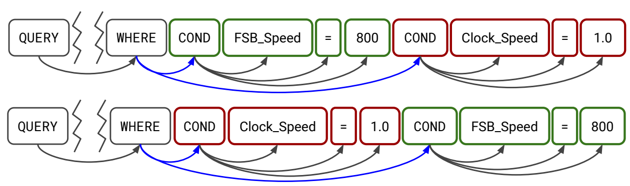

Figure 2: Two valid linearizations of the example query tree in Figure 1.

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) with on-the-fly representations for a subset of its 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 $\mathbf{q}_i = W^E \cdot \mathsf{one\_hot}(q_i)$ where $W^E \in \mathbb{R}^{|\mathcal{V}^E| \times d^{emb}}$ is a learnable weight matrix and $\mathsf{one\_hot}(\cdot)$ 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 $[\mathbf{h}^*{}_0, \mathbf{h}^*{}_1, \ldots, \mathbf{h}^*{}_N] = \mathsf{BiLSTM}([\mathbf{q}_0, \mathbf{q}_1, \ldots, \mathbf{q}_N])$.

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 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 OUT$(\cdot)$. EMB$(\cdot)$ and OUT$(\cdot)$ are described in the following sections.

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

$$a_i^{(t)} = \mathbf{h}_i \cdot \mathbf{y}_t \ , \tag{1}$$

$$\alpha_i^{(t)} = \mathsf{softmax}(a_0^{(t)}, \ldots, a_i^{(t)}, \ldots, a_N^{(t)})_i \ , \tag{2}$$

$$\hat{\mathbf{h}}_t = \sum_{i=0}^{N} \alpha_i^{(t)} \mathbf{h}_i \ , \tag{3}$$

where $\mathsf{softmax}(\cdot)_\mathsf{i}$ denotes the i-th element of the output of the softmax function, $\mathbf{y}_t$ is the output state of the decoder 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 `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 $\mathcal{V}^{\text{SQL}}$ are embedded based on a learnable, randomly initialized embedding matrix W^{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 $\hat{\mathbf{h}}_t$ and the hidden state $\mathbf{y}_t$ 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 ($\mathcal{V}^{\text{SQL}}$), the output scores are computed by the linear transformation: $\mathbf{o}^{\text{SQL}} = U^{\text{SQL}} \cdot [\mathbf{y}_t, \hat{\mathbf{h}}_t]$, where $U^{\text{SQL}} \in \mathbb{R}^{|\mathcal{V}^{\text{SQL}}| \times d^{out}}$ is a trainable matrix. For the *column id tokens* ($\mathcal{V}^{\text{COL}}$), 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 as a (preliminary) column name encoding $\mathbf{u}^*$, similarly to what is done in the embedding function. By using skip connections we compute the average of the word embeddings of the tokens in the column name, $\mathbf{c}_i$ for $i = 1, \ldots, K$, and add them to the preliminary column name encoding $\mathbf{u}^*$ to obtain the final encoding for the column id:

$$\mathbf{u} = \mathbf{u}^* + \begin{bmatrix} \mathbf{0} \\ \frac{1}{K}\sum_i^K \mathbf{c}_i \end{bmatrix} \quad , \tag{4}$$

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[2]: $\mathbf{o}^{\text{COL}} = U^{\text{COL}} \cdot [\mathbf{y}_t, \hat{\mathbf{h}}_t]$.

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 mechanism, 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 using a softmax

$$p^{\text{GEN}}(S_t | s_{t-1}, \ldots, s_0, Q) = \mathsf{softmax}([\mathbf{o}^{\text{SQL}}; \mathbf{o}^{\text{COL}}]) \quad . \tag{5}$$

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

$$p^{\text{PTR}}(q | s_{t-1}, \ldots, s_0, Q) = \sum_{i: q_i = q} \alpha_i^{(t)} \quad . \tag{6}$$

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

$$\begin{aligned} p(S_t | s_{t-1}, \ldots, s_0, Q) = \gamma p^{\text{PTR}}(S_t | s_{t-1}, \ldots, s_0, Q) \\ + (1-\gamma) p^{\text{GEN}}(S_t | s_{t-1}, \ldots, s_0, Q) \quad , \end{aligned} \tag{7}$$

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

Shared softmax: In this approach, we re-use the attention scores $a_i^{(t)}$ (Eq. 1) and obtain the output scores $\mathbf{o}^{\text{E}}$ 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 $Q = [q_0, \ldots, q_i, \ldots, q_N]$ where q occurs, i.e. it is given by:

$$\max_{i: q_i = q} a_i \quad , \tag{8}$$

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:

$$p(S_t | s_{t-1}, \ldots, s_0, Q) = \mathsf{softmax}([\mathbf{o}^{\text{SQL}}; \mathbf{o}^{\text{COL}}; \mathbf{o}^{\text{E}}]) \quad . \tag{9}$$

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.

[2]Note that the skip connections in both the question encoder and the column name encoder use padding such that the word embeddings are added to the same regions of $[\mathbf{y}_t, \hat{\mathbf{h}}_t]$ and $\mathbf{u}$, respectively, and thus are directly matched.

[3]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^{CT} 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[4].

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 exploration 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 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.

[4]We use these constraints during prediction only.

Algorithm 1 Non-deterministic oracle

function GETNEXTANDGOLD(p_t, t, $x_{\leq t}$)
    $\mathsf{VNT}_t \leftarrow \texttt{get\_valid\_next}(t, x_{\leq t})$
    $x_{t+1} \leftarrow \mathrm{argmax}_{\mathcal{V}^D}\, p_t$
    $g_t \leftarrow \mathrm{argmax}_{\mathsf{VNT}_t}\, p_t$
    if $x_{t+1} \notin \mathsf{VNT}_t$ **then**
        $x_{t+1} \leftarrow \texttt{random}(\mathsf{VNT}_t)$
    return g_t, x_{t+1}

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 $\mathsf{VNT}_t \subset \mathcal{V}^D$, from which the correct tree can be reached, is returned by the function $\texttt{get\_valid\_next}(\cdot)$. 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:

$$\nabla J = \mathbb{E}[\nabla \log(p_t(x_{t+1}))A_t] \tag{10}$$

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

function GETNEXTANDGOLD(p_t, t, $x_{\leq t}$)
    $\mathsf{VNT}_t \leftarrow \texttt{get\_valid\_next}(t, x_{\leq t})$
    $x_{t+1} \sim p_t;\;\; x_{t+1} \in \mathsf{VNT}_t$
    $g_t \leftarrow x_{t+1}$
    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[5]. The dataset contains a total of 80654 examples. Each example provides 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.

Experimental Setup

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

[5]`http://github.com/salesforce/WikiSQL`

(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 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 Acc_{QM}.

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% Acc_{QM} 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 Acc_{QM}, 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% Acc_{QM} on the test set.

Effect of ordering in supervision To investigate the influence of the ordering 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 (Acc_{QM}) 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 Acc_{QM} 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 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 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 (Acc_{QM}) 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.

[6]We also investigated non-deterministic oracles in the preprint of this work from May 2018 (`https://openreview.net/forum?id=HJMoLws2z`).

	Dev Accuracies (%)			Test Accuracies (%)		
	Acc_{LF}	Acc_{QM}	Acc_{EX}	Acc_{LF}	Acc_{QM}	Acc_{EX}
Seq2SQL (no RL) (Zhong et al. 2017)	48.2	–	58.1	47.4	–	57.1
Seq2SQL (RL) (Zhong et al. 2017)	49.5	–	60.8	48.3	–	59.4
Pointer-SQL (Wang et al. 2018a)	59.6	–	65.2	59.5	–	65.1
SQLNet (Xu et al. 2017)	–	63.2	69.8	–	61.3	68.0
PT-MAML (Huang et al. 2018)*	63.1	–	68.3	62.8	–	68.0
TypeSQL (Yu et al. 2018)*	–	68.0	74.5	–	66.7	73.5
STAMP (Sun et al. 2018)*	61.7	–	75.1	61.0	–	74.6
Coarse2Fine (Dong and Lapata 2018)	–	–	–	–	71.7	78.5
MQAN (McCann et al. 2018)	–	–	–	72.4	–	80.4
(ours)						
PtrGen-SQL (*shared softmax*)	70.2	72.6	79.0	69.9	72.1	78.4
PtrGen-SQL (*point-or-generate*)	70.0	72.4	78.5	69.7	71.7	78.0
PtrGen-SQL (*shared softmax*) + oracle	56.2	73.4	79.4	55.0	72.7	78.8
(EG-based or concurrent work)						
Pointer-SQL + EG(5) (Wang et al. 2018b)	67.5	–	78.4	67.9	–	78.3
Coarse2Fine + EG(5) (Wang et al. 2018b)	76.0	–	84.0	75.4	–	83.8
IncSQL + oracle + `ANYCOL` (Shi et al. 2018)	49.9	–	84.0	49.9	–	83.7
IncSQL + oracle + `ANYCOL` + EG(5) (Shi et al. 2018)	51.3	–	87.2	51.1	–	87.1

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.

	Dev Accs (%)		Test Accs (%)	
	Acc_{LF}	Acc_{QM}	Acc_{LF}	Acc_{QM}
PtrGen-SQL (*shared softmax*)	70.2	72.6	69.9	72.1
· no constraints	68.6	70.9	68.6	70.5
· using constraints during training	69.8	72.2	69.8	71.9
· no label smoothing	68.3	70.5	68.4	70.1
· no label smoothing (*point-or-generate*)	68.7	70.7	68.5	70.3
· no skip connections	69.6	72.0	69.4	71.6

Table 2: Performance of different variations of our approach.

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.

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

	Dev Accs (%)		Test Accs (%)	
	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 3: Results for different target tree linearizations.

	TF	oracle
Whole Query	72.6	73.4
· SELECT	85.5	85.5
· Aggregator	90.0	90.0
· Column	94.7	94.7
· WHERE	83.4	84.4

Table 4: Error Analysis: Acc_{QM} of different query parts on the development set for TF and oracle-trained *shared softmax* models.

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 WIKISQL. 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 WDAqua (grant no. 642795) and Cleopatra (grant no. 812997).

References

Alvarez-Melis, D., and Jaakkola, T. S. 2016. Tree-structured decoding with doubly-recurrent neural networks.

Bahdanau, D.; Cho, K.; and Bengio, Y. 2014. Neural machine translation by jointly learning to align and translate. In *ICLR 2014*.

Bahdanau, D.; Bosc, T.; Jastrzebski, S.; Grefenstette, E.; Vincent, P.; and Bengio, Y. 2017. Learning to compute word embeddings on the fly. *arXiv preprint arXiv:1706.00286*.

Berant, J.; Chou, A.; Frostig, R.; and Liang, P. 2013. Semantic parsing on freebase from question-answer pairs. In *Proceedings of EMNLP 2013*, 1533–1544.

Dahl, D. A.; Bates, M.; Brown, M.; Fisher, W.; Hunicke-Smith, K.; Pallett, D.; Pao, C.; Rudnicky, A.; and Shriberg, E. 1994. Expanding the Scope of the ATIS Task: The ATIS-3 Corpus. In *Proceedings of the Workshop on Human Language Technology*, HLT '94, 43–48. Stroudsburg, PA, USA: Association for Computational Linguistics.

Dong, L., and Lapata, M. 2016. Language to logical form with neural attention. In *Proceedings of ACL 2016*.

Dong, L., and Lapata, M. 2018. Coarse-to-fine decoding for neural semantic parsing. *arXiv preprint arXiv:1805.04793*.

Goldberg, Y., and Nivre, J. 2012. A dynamic oracle for arc-eager dependency parsing. *Proceedings of COLING 2012*.

Gu, J.; Lu, Z.; Li, H.; and Li, V. O. 2016. Incorporating copying mechanism in sequence-to-sequence learning. *arXiv preprint arXiv:1603.06393*.

Guu, K.; Pasupat, P.; Liu, E.; and Liang, P. 2017. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. In *Proceedings of ACL 2017*.

Haug, T.; Ganea, O.-E.; and Grnarova, P. 2018. Neural multi-step reasoning for question answering on semi-structured tables. 611–617.

Hill, F.; Cho, K.; Korhonen, A.; and Bengio, Y. 2016. Learning to understand phrases by embedding the dictionary. *Transactions of the Association of Computational Linguistics*.

Hochreiter, S., and Schmidhuber, J. 1997. Long short-term memory. *Neural Comput.*

Huang, P.-S.; Wang, C.; Singh, R.; Yih, W.-t.; and He, X. 2018. Natural language to structured query generation via meta-learning. *arXiv preprint arXiv:1803.02400*.

Iyyer, M.; Yih, W.-t.; and Chang, M.-W. 2017. Search-based neural structured learning for sequential question answering. In *Proceedings of ACL 2017*, volume 1, 1821–1831.

Krishnamurthy, J.; Dasigi, P.; and Gardner, M. 2017. Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of EMNLP 2017*.

Liang, C.; Berant, J.; Le, Q.; Forbus, K. D.; and Lao, N. 2016. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. *arXiv preprint arXiv:1611.00020*.

Ling, W.; Luís, T.; Marujo, L.; Astudillo, R. F.; Amir, S.; Dyer, C.; Black, A. W.; and Trancoso, I. 2015. Finding function in form: Compositional character models for open vocabulary word representation. In *Proceedings of EMNLP 2015*.

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*.

Neelakantan, A.; Le, Q. V.; and Sutskever, I. 2016. Neural programmer: Inducing latent programs with gradient descent. In *ICLR 2016*.

Pasupat, P., and Liang, P. 2015. Compositional semantic parsing on semi-structured tables. *arXiv preprint arXiv:1508.00305*.

Pennington, J.; Socher, R.; and Manning, C. 2014. Glove: Global vectors for word representation. In *Proceedings of EMNLP 2014*, 1532–1543.

Rabinovich, M.; Stern, M.; and Klein, D. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of ACL 2017*, volume 1, 1139–1149.

See, A.; Liu, P. J.; and Manning, C. D. 2017. Get to the point: Summarization with pointer-generator networks. *arXiv preprint arXiv:1704.04368*.

Shi, T.; Tatwawadi, K.; Chakrabarti, K.; Mao, Y.; Polozov, O.; and Chen, W. 2018. Incsql: Training incremental text-to-sql parsers with non-deterministic oracles. *arXiv preprint arXiv:1809.05054*.

Sun, Y.; Tang, D.; Duan, N.; Ji, J.; Cao, G.; Feng, X.; Qin, B.; Liu, T.; and Zhou, M. 2018. Semantic parsing with syntax- and table-aware sql generation. In *Proceedings of ACL 2018*.

Szegedy, C.; Vanhoucke, V.; Ioffe, S.; Shlens, J.; and Wojna, Z. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of CVPR 2016*, 2818–2826.

Vinyals, O.; Bengio, S.; and Kudlur, M. 2016. Order matters: Sequence to sequence for sets.

Wang, C.; Brockschmidt, M.; and Singh, R. 2018a. Pointing out sql queries from text.

Wang, C.; Huang, P.-S.; Polozov, A.; Brockschmidt, M.; and Singh, R. 2018b. Execution-guided neural program decoding. *arXiv preprint arXiv:1807.03100*.

Xu, X.; Liu, C.; and Song, D. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436*.

Yih, W.-t.; Richardson, M.; Meek, C.; Chang, M.-W.; and Suh, J. 2016. The value of semantic parse labeling for knowledge base question answering. In *Proceedings of ACL 2016*, volume 2, 201–206.

Yu, T.; Li, Z.; Zhang, Z.; Zhang, R.; and Radev, D. 2018. Typesql: Knowledge-based type-aware neural text-to-sql generation. *arXiv preprint arXiv:1804.09769*.

Zettlemoyer, L., and Collins, M. 2007. Online learning of relaxed ccg grammars for parsing to logical form. In *Proceedings of EMNLP-CoNLL 2007*.

Zhong, V.; Xiong, C.; and Socher, R. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.