How to SEQ2SEQ for SQL

Anonymous submission

Abstract

Translating natural language to SQL queries for table-based question answering has recently attracted more research attention. Previous approaches develop models whose architecture is specifically tuned to the structure of the task, such as separately predicting the arguments of the SELECT clause. In this work, we show that a more general attention-based sequence-to-sequence model outperforms more specialized state-of-the-art approaches by only adapting the input and output layers. In particular, we extend it with on-the-fly embedding and output vectors as well as an input copying mechanism, which are used throughout the whole decoding process. We also investigate the potential *order-matters* problem that could arise due to having multiple correct decoding paths and investigate the use of a dynamic oracle in this context.

1 Introduction

Semantic parsing, the task of converting natural language utterances to their representation in a formal language, is a fundamental problem in NLP and has many applications, including Question Answering (QA) over structured data. In this work, we focus on QA over tabular data (NL2SQL), which recently attracted research efforts (Zhong et al., 2017; Xu et al., 2017; Neelakantan et al., 2017). As in many other semantic parsing tasks, in this task, we are confronted with tree-structured decoding targets and challenging output vocabularies. In NL2SQL, we must handle table columns, which may be different for every example and unseen during testing. We also need to handle other rare words that refer to values in the tables and thus are essential for building the correct query.

We develop a SEQ2SEQ-based approach that outperforms the state-of-the-art on the recently introduced WIKISQL (Zhong et al., 2017) dataset. With 80654 examples annotated with their logical forms, WIKISQL is, to the best of our knowledge, the largest fully annotated and manually verbalized dataset for semantic parsing and question answering with complex questions[1]. In contrast to previous works on WIKISQL (Zhong et al., 2017; Xu et al., 2017) that adapt the architecture of their models to the *structure* of the data, our approach is more general as it only adapts the input and output layers of a standard SEQ2SEQ model to better suit the task of generating SQL queries. In particular, we add on-the-fly embeddings (Bahdanau et al., 2017; Hill et al., 2016) and on-the-fly output vectors and incorporate a pointer-based (Vinyals et al., 2015) copying mechanism as part of the output layer. In contrast to previous works on WIKISQL, which used separate losses for SQL generation subtasks, our approach can be trained using normal sequence supervision.

The order of decoding of unordered sets using SEQ2SEQ models can have a significant impact on performance (Vinyals et al., 2016), as mentioned by previous works on WIKISQL (Zhong et al., 2017; Xu et al., 2017). We investigate the use of a dynamic oracle (Goldberg and Nivre, 2012) in an effort to improve the generation of logical forms with unordered children (in our case conditions in the `WHERE` clause).

To summarize, our contributions are as follows:

1. We outperform the state-of-the-art on the WIKISQL task using a SEQ2SEQ architecture with specialized input and output layers.

[1] Complex questions are those containing more than a single condition.

The devised model is more extensible to tasks with a different structure than previous work.

2. We investigate the potential *order-matters* problem (Vinyals et al., 2016) when having multiple correct decoding paths. In this context, we also investigate two types of dynamic oracle in training semantic parsers instead of teacher forcing. To the best of our knowledge, dynamic oracles have not been investigated for neural network-based semantic parsing before.

The paper is structured as follows: First, we illustrate the task with an example (Section 2). Then, we describe our model (Section 3), followed by a description of the training methods (Section 4). We then present our experimental setup and results in Section 5 and provide an extensive discussion in Section 6 that goes deeper into the most related work. We conclude with an overview of other related work (Section 7) and a conclusion (Section 8).

2 Queries, Trees and Linearizations

As an illustration, consider the question

> "*Which L1 Cache can we get with an FSB speed of 800MHz and a clock speed of 1.2GHz?*",

which should be mapped to the following query:

```
SELECT  L1_Cache
WHERE   FSB_Speed = 800 (Mhz)
  AND   Clock_Speed = 1.0 (GHz)
```

This query will be executed over a table listing processors, the sizes of their caches, their clocks speeds, etc.

The query can be represented as a tree (see Figure 1), 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). We refer to the linearization where the original order given in the data set is preserved as the *canonical linearization*. The two valid linearizations for the example are shown in Figure 2. See Supplement A for specifics on our implementation of tree linearization.

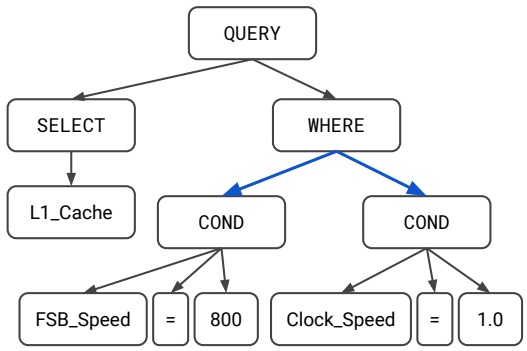

Figure 1: Example of a query tree. The blue arrows indicate unordered children.

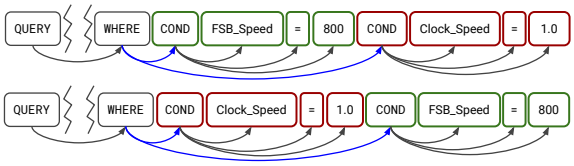

Figure 2: Two valid linearizations of the example query tree in Figure 1.

3 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 compute on-the-fly embeddings (Bahdanau et al., 2017) and output vectors for column tokens and implement a pointer-based (Vinyals et al., 2015) mechanism for copying tokens from the question. The SEQ2SEQ model, the embedding and output layers are described in Sections 3.1, 3.2 and 3.3, respectively.

3.1 The SEQ2SEQ Model

The general architecture of our model follows the standard sequence-to-sequence (SEQ2SEQ) attention-based architecture. The SEQ2SEQ model consists of an encoder, a decoder and an attention mechanism.

3.1.1 Encoding

We are given a question $Q = [q_0, q_1, \ldots, q_N]$ of natural language (NL) tokens q_i from a set $\mathcal{V}^E$ (i.e., the encoder vocabulary). We first pass the tokens through an embedding layer that maps every token q_i to its vector representation $\mathbf{q}_i \in \mathbb{R}^{d^{emb}}$, that is

$$\mathbf{q}_i = W^E \cdot \mathsf{one\_hot}(q_i) \ , \qquad (1)$$

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 two-layered LSTM (Hochreiter and Schmidhuber, 1997) (see Supplement B) encoder outputs encoding vectors $[\mathbf{h}_0, \mathbf{h}_1, \ldots, \mathbf{h}_N] = \mathsf{BiLSTM}([\mathbf{q}_0, \mathbf{q}_1, \ldots, \mathbf{q}_N])$, where $\mathbf{h}_i \in \mathbb{R}^{d^{enc}}$.

3.1.2 Decoding

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 unidirectional multi-layered LSTM. First, the previous output s_{t-1} is mapped to its vector representation:

$$\mathbf{s}_{t-1} = \mathrm{EMB}(s_{t-1}) \ . \tag{2}$$

The applied embedding function $\mathrm{EMB}(\cdot)$ is described in Section 3.2 in detail. Taking the embedding as input, the multi-layered LSTM updates the hidden states in every layer (see Supplement B for the update equations). Finally, the probabilities of the elements of $\mathcal{V}^D$ are calculated based on the output $\mathbf{y}_t$ of the final LSTM layer and a context vector $\hat{\mathbf{h}}_t$, which is defined in Section 3.1.3:

$$\mathbf{o}_t = \mathrm{OUT}(\mathbf{y}_t, \hat{\mathbf{h}}_t) \ , \tag{3}$$

$$p(s_t | s_{t-1}, \ldots, s_0, Q) = \mathsf{softmax}(\mathbf{o}_t) \ . \tag{4}$$

where the function $\mathrm{OUT}(\cdot)$ is the output layer that computes unnormalized scores over $\mathcal{V}^D$ as explained in Section 3.3.

3.1.3 Attention

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

$$a_i^{(t)} = \mathbf{h}_i \cdot \mathbf{y}_t \ , \tag{5}$$

$$\alpha_i^{(t)} = \mathsf{softmax}(a_0^{(t)}, \ldots, a_i^{(t)}, \ldots, a_N^{(t)})_i \ , \tag{6}$$

$$\hat{\mathbf{h}}_t = \sum_{i=0}^{N} \alpha_i^{(t)} \mathbf{h}_i \ , \tag{7}$$

where $\mathsf{softmax}(\dot{})_\mathsf{i}$ denotes the i-ith element of the output of the softmax function and $\mathbf{h}_1, \ldots, \mathbf{h}_N$ are the hidden vectors of the encoder described in Section 3.1.1.

3.2 Embedding Function of the Decoder

The whole output vocabulary $\mathcal{V}^D$ can be grouped in three parts: (1) structure tokens from $\mathcal{V}^{\mathrm{STR}}$, (2) column ids from $\mathcal{V}^{\mathrm{COL}}$, and (3) input words from the encoder vocabulary $\mathcal{V}^E$, i.e., $\mathcal{V}^D = \mathcal{V}^{\mathrm{STR}} \cup \mathcal{V}^{\mathrm{COL}} \cup \mathcal{V}^E$. Whereas structure tokens are shared among examples, column ids refer to columns that can have different names in every example depending on the corresponding table. In the following paragraphs, we describe how each of the three types of tokens is embedded in the decoder.

Structure tokens: These are tokens used to represent the structure of the query, such as SQL specific tokens like `SELECT` and `WHERE`. Since the available tokens $\mathcal{V}^{\mathrm{STR}}$ are the same for every example, and their meaning stays the same across examples, we use a vanilla embedding matrix W^{STR} and reuse it across all examples. See Supplement A for details on the used structure tokens.

Column tokens: These are column ids that are used to refer to a certain column in the table the question is being asked against. Since these ids themselves are largely meaningless, we map them to the corresponding column names to get a more meaningful representation. We encode the column names using a uni-directional single-layer LSTM to get the embedding vector representing the ids in $\mathcal{V}^{\mathrm{COL}}$.

Column names may consist of several words, which are first embedded using an embedding matrix W^{CT} containing an embedding vector for every word in $\mathcal{V}^{\mathrm{CT}}$ that occurs in any column name in any table. The word embedding are then fed into the LSTM and the final state of the LSTM is taken as the embedding vector for the column id token. This approach for computing column representations is similar in spirit 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 that is also used for encoding the question.

3.3 Output Layer of the Decoder

The output layer of the decoder takes a the context $\hat{\mathbf{h}}_t$ and the hidden state $\mathbf{y}_t$ of the decoder's LSTM and produces scores over the output vocabulary $\mathcal{V}^D$ (see $\mathsf{OUT}(\cdot)$ in Eq. 3). For each of the three subsets of $\mathcal{V}^D$ ($\mathcal{V}^{\mathrm{STR}}$, $\mathcal{V}^{\mathrm{COL}}$ and $\mathcal{V}^E$, as defined in Section 3.2), the scores are computed as

described in the following paragraphs. The whole output layer is visualized in Figure 3.

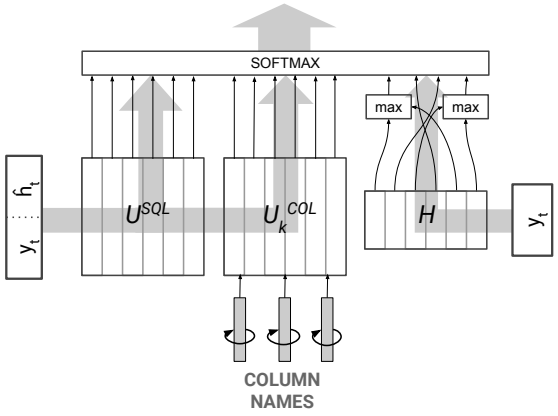

Figure 3: The output layer and its three components.

Structure tokens: For the structure tokens we apply a trainable matrix $U^{\text{STR}} \in \mathbb{R}^{|\mathcal{V}^{\text{STR}}| \times d^{out}}$ that is shared across examples. The output scores for tokens in $\mathcal{V}^{\text{STR}}$ are computed by the linear transformation: $U^{\text{STR}} \cdot [\mathbf{y}_t, \hat{\mathbf{h}}_t]$.

Column tokens: Similarly as in the embedding function, the output vectors for every column[2] from $\mathcal{V}^{\text{COL}}$ are computed by encoding the corresponding column names using an LSTM. Let the matrix U_k^{COL} contain the computed vectors for every column present in the table of the current example. The output scores for all column id token are then computed by $U_k^{\text{COL}} \cdot [\mathbf{y}_t, \hat{\mathbf{h}}_t]$.

Input words: To enable our model to copy tokens from the input question, we follow a pointer-based approach to compute scores over these words. That is, we compute scores b_i between $\mathbf{y}_t$ and the hidden state of the encoder LSTM at every input position i:

$$b_i = \mathbf{h}_i \cdot \mathbf{y}_t \ , \tag{8}$$

and then obtain the score for every token q that occurs in the question sequence Q by taking the maximum over all positions in $Q = [q_0, \ldots, q_i, \ldots, q_N]$ where q occurs, i.e. the score for q is given by:

$$\max_{i=0..N} \{b_i\}_{q_i=q} \ . \tag{9}$$

[2]Recall that each column is represent by a single column id token in the full output vocabulary

The scores for all input tokens $q \in \mathcal{V}^E$ that do not occur in the question Q are set to $-\infty$.

We also ran experiments without the pointer mechanism just described to test its influence (see "without pointer" experiment in Section 5.3). There, we replace the pointer mechanism by a simple linear transformation $U^{E_D} \cdot [\hat{\mathbf{h}}_t, \mathbf{y}_t]$ with a matrix U^{E_D} to yield scores over the set of tokens in the question. U^{E_D} is populated using pretrained word vectors which are kept fixed during training. The alternative, training randomly initialized U^{E_D} resulted in worse performance, and is not reported.

3.4 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.

4 Training

We train our models by minimizing the cross-entropy loss over the output probabilities:

$$\mathcal{L} = -\sum_{(R^{(k)}, Q^{(k)}) \in \mathcal{D}} \log(p(G^{(k)}|Q^{(k)})) \ , \tag{10}$$

$$p(G^{(k)}, Q^{(k)}) = \prod_{t=0}^{|G^{(k)}|} p(g_t^{(k)}|g_{<t}^{(k)}, Q^{(k)}) \ , \tag{11}$$

where $G^{(k)}$ is a sequence of tokens $g_t^{(k)} \in \mathcal{V}^D$ representing a valid linearization of the correct tree $R^{(k)}$ corresponding to question $Q^{(k)}$ from the training data $\mathcal{D}$. The probabilities of the tokens $g_t^{(k)}$ are computed as described in Section 3.3.

For constructing the target sequence $G^{(k)}$ from the tree $R^{(k)}$ during training we use teacher forcing (TF) and also experiment with a dynamic ora-

[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.

cle (Goldberg and Nivre, 2012). The dynamic oracle was proposed for dependency parsing, where different sequences of actions can lead to the same parse tree. We notice that in semantic parsing, when the target trees contain at least some unordered children, we are confronted with a similar setting where different sequences of tokens can describe the same query. To the best of our knowledge, dynamic oracles have not been applied before in training neural networks based semantic parsers.

4.1 Teacher Forcing

Teacher forcing takes the canonical linearizations of the query trees (as provided in the dataset) and uses it both for supervision and as input to the decoder. Thus, teacher forcing does not explore any deviations from the canonical path during training. In the presence of different correct sequences (as resulting from different correct linearizations in this scenario), 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).

4.2 Dynamic Oracle

Instead of forcing the model to follow the canonical decoding sequence, our dynamic oracle enables the exploration of alternative correct sequences (i.e. different valid linearizations of the same 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} (that will be used as input to the decoder in the next time step) from the set VNT_t of possible next tokens that would lead to a valid linearization of the given tree. 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. Note that, for decoding steps t where VNT_t is of size one, the oracle is equivalent to teacher forcing.

In the algorithm, p_t is the decoder's output distribution over $\mathcal{V}^D$ at time step t. The set of valid

Algorithm 1 Computing the supervision token and next token in dynamic oracle training.

1: **function** GETNEXTANDGOLD(p_t, t, $x_{\leq t}$)
2: $\quad\mathsf{VNT}_t \leftarrow \texttt{get\_valid\_next}(t, x_{\leq t})$
3: $\quad x_{t+1} \leftarrow \text{argmax}_{\mathcal{V}^D}\, p_t$
4: $\quad g_t \leftarrow \text{argmax}_{\mathsf{VNT}_t}\, p_t$
5: $\quad$ **if** $x_{t+1} \notin \mathsf{VNT}_t$ **then**
6: $\quad\quad x_{t+1} \leftarrow \texttt{random}(\mathsf{VNT}_t)$
7: $\quad$ **return** g_t, x_{t+1}

next tokens $\mathsf{VNT}_t \subset \mathcal{V}^D$, from which the correct tree can be reached, is returned by the function $\texttt{get\_valid\_next}(\cdot)$. In order to compute VNT_t, we keep track of where in the query tree we are and of what is left to decode, given the previously decoded tokens. 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 canonical sequence order. See Supplement C for more details on the construction of VNT_t.

Zhong et al. (2017) propose policy gradient (PG) based training for query generation. The presented oracle is similar to PG training in that it explores alternative paths to generate the same query. In contrast to the oracle, a PG-based method would sample the next token (x_{t+1}) according to the predictive distribution (p_t) and then use the sampled sequence to compute gradients for policy parameters:

$$\nabla J = \mathbb{E}[\nabla \log(p_t(x_{t+1}))A_t] \qquad (12)$$

Replacing $\text{argmax}_{\mathcal{V}^D}\, p_t$ in line 3 of Algorithm 1 with $\text{sample}_{\mathsf{VNT}_t} p_t$ (sample a token from VNT_t according to p_t) yields an algorithm equivalent to a specific case of REINFORCE (Williams, 1992): where A_t is the reward for the episode, and A_t is set to $+1$ if the sampled sequence produces the full correct query and 0 otherwise. With such rewards, the resulting vanilla REINFORCE does not update policy parameters when an incorrect query is sampled, ignoring the sampled episode. In contrast, Zhong et al. (2017) use a different reward scheme, assigning rewards of -2 for invalid queries and -1 if the query is valid but the execution result is incorrect and $+1$ otherwise.

5 Experiments

To evaluate our approach, we obtain the WIKISQL dataset by following the instructions on the WIKISQL website[4].

5.1 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. This allows to obtain a performance estimate for the developed approaches when confronted with questions over previously unseen tables.

Whereas many different tables occur in WIKISQL, a single query is limited to only a single table. Nevertheless, the data set poses a significant challenge due to the variation of table schemata across examples and the variable structure of expected SQL queries. In addition, the task is complicated by a high occurrence of rare words, which can not be omitted since they can be essential for constructing the correct SQL query. However, WIKISQL is constructed such that the values in the conditions of the query occur as substrings in the NL question.

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).

5.2 Experimental Setup

Evaluation: We report results with the same evaluation metrics as previous works (Zhong et al., 2017; Xu et al., 2017) on both the development and test sets:

- logical form accuracy $\text{Acc}_{\text{LF}} = K_{LF}/K$,
- query match accuracy $\text{Acc}_{\text{QM}} = K_{QM}/K$,
- query execution accuracy $\text{Acc}_{\text{EX}} = K_{EX}/K$

where K is the total number of examples, K_{LF} the number of examples for which the predicted sequence matched exactly the one provided by the dataset, K_{QM} the number of examples for which the predicted query matched exactly (i.e. the predicted sequence is one of the correct linearizations) and K_{EX} the number of examples for which the execution results matched exactly. Note that while Acc_{LF} accepts only the canonical ordering of conditions in the `WHERE` clause, Acc_{QM} and Acc_{EX} accept alternative orderings.

[4] http://github.com/salesforce/WikiSQL

Training details: After a hyperparameter search, we obtained the best results by using two layers both in the encoder and decoder LSTMs, using $d^{dec} = 600$, $d^{emb} = 300$ 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 (Kingma and Ba, 2014), with a learning rate of 0.001 and a batch size of 100, a maximum of 50 epochs, and early stopping and clipped gradient norms at 5.

Experimental settings: While training with an oracle does not depend on the chosen linearization of the query tree, training with TF does. Therefore, for experiments with TF we explored variations in the linearizations of the query tree by changing the order of conditions in the WHERE clause, that is the order of conditions in the canonical query linearization is (1) reversed or is (2) reassigned randomly.

We also run experiments with the oracle in two variants: (1) as described in Algorithm 1 and (2) replacing the argmax in line 3 of Algorithm 1 with sampling according to p_t, which is equivalent to REINFORCE with zero reward for incorrect queries (see Section 4.2).

For more details on the experimental setup, we refer to Supplement D. Our preprocessed data, code, and sample outputs will be made freely available after the double-blind review process is finished.

5.3 Results

We compare our obtained results under different settings with the performance reported for previous methods in Table 1. The proposed SEQ2SEQ model outperforms the current state-of-the-art by at least 4% Acc_{QM}. Using a single representation for rare words (indicated by "+ *rare*" in the table) provides a small improvement.

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 randomly reassigned order of conditions in the WHERE clause results in a 1.5% drop in accuracy. However, reversing the order of conditions does not seem to affect the results. The REINFORCE-like dynamic oracle (indicated by "*oracle - sample*" in the table) also does not seem to affect the results. However, the dynamic oracle as described

in Algorithm 1 seems to provide a small improvement.

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.

We also ran one setting (indicated by "*without pointer*" in the table) without the copying mechanism (described in Section 3.3), where we obtained significantly worse results, indicating the benefit brought by the pointer mechanism.

See Supplement E for separate accuracies for the `WHERE` and `SELECT` clauses.

6 Discussion

First, we discuss our model by comparison to Seq2SQL (Zhong et al., 2017) and SQLNet (Xu et al., 2017), followed by a discussion of the training methodology used by previous approaches and our approach.

6.1 Model Discussion

Seq2SQL: The baseline in Seq2SQL (Zhong et al., 2017) was based on the earlier SEQ2SEQ work by Dong and Lapata (2016) for semantic parsing with lambda expressions. Our model uses the same architecture (SEQ2SEQ) but in contrast to Seq2SQL's adaptation of Dong and Lapata (2016) to WIKISQL, we develop task-adapted embedding and output layers in the decoder. Given the poor performance of an unadapted SEQ2SEQ model, Zhong et al. (2017) propose an *augmented pointer network* that is better suited for copying words from the input.

Seq2SQL authors obtain further improvement by introducing separate predictors with dedicated parameters in order to predict the two arguments of the `SELECT` clause: (1) the `SELECT` column and (2) the aggregator. This also requires the addition of two losses in the final training loss. For the `WHERE` clause, Seq2SQL uses the augmented pointer network trained using policy gradient RL. The `SELECT` column prediction uses column names encoded using an LSTM, similar to our approach. However, our approach uses the same column name encodings throughout the whole decoding process .

SQLNet: Similarly to Seq2SQL, SQLNet uses dedicated networks for predicting the two parts of the `SELECT` clause, using column-conditioned question summaries and explicitly encoding task-specific prediction dependencies into the architecture. For the `WHERE` clause, SQLNet proposes a *sequence-to-set* approach that essentially makes a set inclusion prediction over the set of available columns in order to avoid the *order-matters* (Vinyals et al., 2016) problem. However, in addition to producing scores for the available columns, SQLNet needs a dedicated network to predict the number of columns that will ultimately be taken.

Compared to Seq2SQL and SQLNet, we believe our model is simpler and more general, while achieving better performance. Thus, we believe that our model can be applied to other tasks, such as machine translation, if they can benefit from the rare word handling extensions.

6.2 Training Discussion

The oracle we implement is similar to reinforcement learning (RL) in that it explores alternative paths to generate the same query and similar to the work of Vinyals et al. (2016) in that it's aimed at finding and using more effective linearizations during training. The *dynamic oracle with non-optimal transitions* from Goldberg and Nivre (2012) also explores incorrect paths but is able to provide more useful supervision at every time step on how to build the best possible tree after a mistake. Our oracle implements only the *dynamic oracle with spurious ambiguity* (Goldberg and Nivre, 2012) and thus explores only correct paths. We leave the oracle with non-optimal transitions for future work.

7 Other Related Work

Some examples of semantic parsers that do not rely on neural networks include Zettlemoyer and Collins (2007), Berant et al. (2013) and Reddy et al. (2014). Recently, many neural network-based approaches emerged

	Dev Accuracies (%)			Test Accuracies (%)		
	Acc_{LF}	Acc_{QM}	Acc_{EX}	Acc_{LF}	Acc_{QM}	Acc_{EX}
Seq2Seq baseline (Zhong et al., 2017)	23.3	–	37.0	23.4	–	35.9
Aug. Ptr. Net (Zhong et al., 2017)	44.1	–	53.8	43.3	–	53.3
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
*Seq2SQL (SQLNet) (Xu et al., 2017)	52.5	53.5	62.1	50.8	51.6	60.4
SQLNet (Xu et al., 2017)	–	63.2	69.8	–	61.3	68.0
Seq2Seq+DP+C (TF)	65.1	67.2	74.3	64.6	66.6	73.7
Seq2Seq+DP+C (TF - reversed)	65.1	67.3	74.6	64.5	66.2	74.0
Seq2Seq+DP+C (TF - arbitrary)	–	65.7	73.5	–	64.7	72.5
Seq2Seq+DP+C (TF) *without pointer*	60.9	62.7	69.9	60.1	61.7	69.0
Seq2Seq+DP+C (TF) + rare	**66.1**	68.2	75.4	**65.9**	67.7	75.0
Seq2Seq+DP+C (oracle)	52.8	67.9	75.2	51.8	67.2	74.6
Seq2Seq+DP+C (oracle - sample)	54.3	67.0	74.5	53.3	66.3	73.8
Seq2Seq+DP+C (oracle) + rare	53.2	**68.6**	**75.8**	52.3	**68.2**	**75.5**

Table 1: Evaluation results for our approach (bottom part) and comparison with previously reported results (top part). Our approach significantly outperforms the current state-of-the-art. 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.

focusing on semantic parsing and question answering. Works focusing on answering simple questions (Bordes et al., 2015; Yin et al., 2016; Lukovnikov et al., 2017; Dai et al., 2016) typically employ a ranking approach to rank logical forms (which always consist of 2 elements). Regarding more complex questions, some approaches (Yih et al., 2015; Yu et al., 2017) also rely on the ranking of (parts of) candidate queries and use rules to complete the query. Some other works on complex questions use reinforcement learning (Liang et al., 2016; Zhong et al., 2017).

Some interesting approaches for answering questions over tables include the Neural Programmer (Neelakantan et al., 2016) and Neural Enquirer (Yin et al., 2015), both of which use only the execution results to supervise query composition, relying on end-to-end decoding and execution instead of reinforcement learning. Neelakantan et al. (Neelakantan et al., 2017) further evaluate the Neural Programmer on the WIKITABLEQUESTIONS (Pasupat and Liang, 2015) dataset. The WIKITABLEQUESTIONS dataset is similar to the WIKISQL dataset in that it focuses on learning to query tables, however, WIKITABLEQUESTIONS does not provide logical forms.

8 Conclusion

In this work we present a SEQ2SEQ model adapted to the semantic parsing task of NL2SQL. It outperforms the state-of-the-art on the WIKISQL dataset, while being more general than previous works in the sense that we only adapt the input and output layers of a SEQ2SEQ model. We also investigated the order-matters problem, concluding that order of conditions in the WHERE clause matters to small degree. We also evaluated two dynamic oracles for training the neural network-based semantic parser, which in our experiments revealed that the RL-like oracle does not improve results and the oracle most similar to the work of Goldberg and Nivre (2012) provides a small improvement.

References

Dzmitry Bahdanau, Tom Bosc, Stanislaw Jastrzebski, Edward Grefenstette, Pascal Vincent, and Yoshua Bengio. 2017. Learning to compute word embeddings on the fly. *arXiv preprint arXiv:1706.00286*.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*.

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1544.

Antoine Bordes, Nicolas Usunier, Sumit Chopra, and Jason Weston. 2015. Large-scale simple question answering with memory networks. *arXiv preprint arXiv:1506.02075*.

Zihang Dai, Lei Li, and Wei Xu. 2016. Cfo: Conditional focused neural question answering with large-scale knowledge bases. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*.

Yoav Goldberg and Joakim Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. *Proceedings of COLING 2012*.

Felix Hill, KyungHyun Cho, Anna Korhonen, and Yoshua Bengio. 2016. Learning to understand phrases by embedding the dictionary. *Transactions of the Association of Computational Linguistics*.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Comput.*

Diederik Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization.

Chen Liang, Jonathan Berant, Quoc Le, Kenneth D Forbus, and Ni Lao. 2016. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. *arXiv preprint arXiv:1611.00020*.

Wang Ling, Tiago Luís, Luís Marujo, Ramón Fernandez Astudillo, Silvio Amir, Chris Dyer, Alan W Black, and Isabel Trancoso. 2015. Finding function in form: Compositional character models for open vocabulary word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*.

Denis Lukovnikov, Asja Fischer, Jens Lehmann, and Sören Auer. 2017. Neural network-based question answering over knowledge graphs on word and character level. In *Proceedings of the 26th international conference on World Wide Web*.

Arvind Neelakantan, Quoc V Le, Martin Abadi, Andrew McCallum, and Dario Amodei. 2017. Learning a natural language interface with neural programmer.

Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. 2016. Neural programmer: Inducing latent programs with gradient descent. In *International Conference on Learning Representations*.

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

Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

Siva Reddy, Mirella Lapata, and Mark Steedman. 2014. Large-scale semantic parsing without question-answer pairs. *Transactions of the Association of Computational Linguistics*.

Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. 2016. Order matters: Sequence to sequence for sets.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *Advances in Neural Information Processing Systems 28*.

Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*, pages 5–32. Springer.

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

Scott Wen-tau Yih, Ming-Wei Chang, Xiaodong He, and Jianfeng Gao. 2015. Semantic parsing via staged query graph generation: Question answering with knowledge base.

Pengcheng Yin, Zhengdong Lu, Hang Li, and Ben Kao. 2015. Neural enquirer: Learning to query tables with natural language. *arXiv preprint arXiv:1512.00965*.

Wenpeng Yin, Mo Yu, Bing Xiang, Bowen Zhou, and Hinrich Schütze. 2016. Simple question answering by attentive convolutional neural network. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*.

Mo Yu, Wenpeng Yin, Kazi Saidul Hasan, Cicero dos Santos, Bing Xiang, and Bowen Zhou. 2017. Improved neural relation detection for knowledge base question answering. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Luke Zettlemoyer and Michael Collins. 2007. Online learning of relaxed ccg grammars for parsing to logical form. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.