Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning

Victor Zhong
Salesforce Research
Palo Alto, CA
vzhong@salesforce.com

Caiming Xiong
Salesforce Research
Palo Alto, CA
cmxiong@salesforce.com

Richard Socher
Salesforce Research
Palo Alto, CA
rsocher@salesforce.com

Abstract

A significant amount of the world's knowledge is stored in relational databases. However, the ability for users to retrieve facts from a database is limited due to a lack of understanding of query languages such as SQL. We propose Seq2SQL, a deep neural network for translating natural language questions to corresponding SQL queries. Our model leverages the structure of SQL queries to significantly reduce the output space of generated queries. Moreover, we use rewards from in-the-loop query execution over the database to learn a policy to generate unordered parts of the query, which we show are less suitable for optimization via cross entropy loss. In addition, we will publish WikiSQL, a dataset of 87726 hand-annotated examples of questions and SQL queries distributed across 26375 tables from Wikipedia. This dataset is required to train our model and is an order of magnitude larger than comparable datasets. By applying policy-based reinforcement learning with a query execution environment to WikiSQL, our model Seq2SQL outperforms attentional sequence to sequence models, improving execution accuracy from 35.9% to 60.3% and logical form accuracy from 23.4% to 49.2%.

1 Introduction

A vast amount of today's information is stored in relational databases, which provide the foundation of crucial applications such as medical records [Hillestad et al., 2005], financial markets [Beck et al., 2000], and customer relations management [Ngai et al., 2009]. However, accessing information in relational databases requires an understanding of query languages such as SQL, which, while powerful, is difficult to master. A prominent research area at the intersection of natural language processing and human-computer interactions is the study of natural language interfaces (NLI) [Androutsopoulos et al., 1995], which seek to provide means for humans to interact with computers through the use of natural language. We investigate one particular aspect of NLI applied to relational databases: translating natural language questions to SQL queries.

Our main contributions in this work are two-fold. First, we introduce Seq2SQL, a deep neural network for translating natural language questions to corresponding SQL queries. Seq2SQL, shown in Figure 1, consists of three components that leverage the structure of a SQL query to greatly reduce the output space of generated queries. Moreover, it uses policy-based reinforcement learning (RL) to generate the conditions of the query, which we show are unsuitable for optimization using cross entropy loss due to their unordered nature. Seq2SQL is trained using a mixed objective combining cross entropy losses and RL rewards from in-the-loop query execution on a database. These characteristics allow the model to achieve much improved results on query generation.

Second, we release WikiSQL, a corpus of 87726 hand-annotated instances of natural language questions, SQL queries, and SQL tables extracted from 26375 HTML tables from Wikipedia. WikiSQL is an order of magnitude larger than comparable datasets that also provide logical forms along with natural language utterances. We release the tables used in WikiSQL both in raw JSON format as well

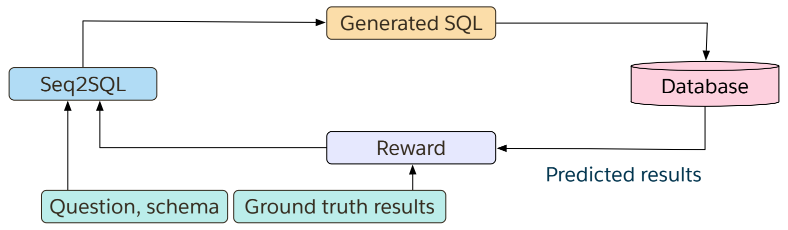

Figure 1: Seq2SQL takes as input a question and the columns of a table. It generates the corresponding SQL query, which, during training, is executed against a database. The result of the execution is utilized as the reward to train the reinforcement learning algorithm.

as in the form of a SQL database. Along with WikiSQL, we also release a query execution engine for the database, which we use for in-the-loop query execution to learn the policy. On WikiSQL, we show that Seq2SQL significantly outperforms an attentional sequence to sequence baseline, which obtains 35.9% execution accuracy, as well as a pointer network baseline, which obtains 52.8% execution accuracy. By leveraging the inherent structure of SQL queries and applying policy gradient methods using the reward signals from live query execution during training, Seq2SQL achieves a marked improvement on WikiSQL, obtaining 60.3% execution accuracy.

2 Related Work

Learning logical forms from natural language utterances. In semantic parsing for question answering (QA), natural language questions are parsed into logical forms, which are then executed on a knowledge graph [Zelle and Mooney, 1996, Wong and Mooney, 2007, Zettlemoyer and Collins, 2005, 2007]. Other work in semantic parsing has focused on learning parsers without relying on annotated logical forms by leveraging conversational logs [Artzi and Zettlemoyer, 2011], demonstrations [Artzi and Zettlemoyer, 2013], distant supervision [Cai and Yates, 2013, Reddy et al., 2014], and question-answer pairs [Liang et al., 2011]. Recently, Pasupat and Liang [2015] introduced a floating parser to address the large amount of variation in utterances and table schema for semantic parsing on web tables. Our approach is different from these semantic parsing techniques in that it does not rely on a high quality grammar and lexicon, and instead relies on representation learning.

Semantic parsing datasets. Previous semantic parsing systems were typically designed to answer complex and compositional questions over closed-domain, fixed-schema datasets such as GeoQuery [Tang and Mooney, 2001] and ATIS [Price, 1990]. Researchers have also investigated QA over subsets of large-scale knowledge graphs such as DBPedia [Starc and Mladenic, 2017] and Freebase [Cai and Yates, 2013, Berant et al., 2013]. The dataset “Overnight” [Wang et al., 2015] uses a similar crowd-sourcing process to build a dataset of (natural language question, logical form) pairs, but it is comprised of only 8 domains. WikiTableQuestions [Pasupat and Liang, 2015] is a collection of question and answers, also over a large quantity of tables extracted from Wikipedia. However, it does not provide logical forms whereas WikiSQL does. WikiTableQuestions is focused on the task of QA over potentially noisy web tables, whereas WikiSQL is focused on generating SQL queries for questions over relational database tables. Our intent is to build a natural language interface for databases.

Representation learning for sequence generation. Our baseline model is based on the attentional sequence to sequence model (Seq2Seq) proposed by [Bahdanau et al., 2015]. Seq2SQL is inspired by pointer networks [Vinyals et al., 2015, Merity et al., 2017], which, instead of generating words from a fixed vocabulary like Seq2Seq, generates by selecting words from the input sequence. This class of models has been successfully applied to tasks such as language modeling [Merity et al., 2017], summarization [Gu et al., 2016], combinatorial optimization [Bello et al., 2017], and question answering [Seo et al., 2017, Xiong et al., 2017]. Neural methods have also been applied to semantic parsing [Dong and Lapata, 2016, Neelakantan et al., 2017]. However, unlike [Dong and Lapata, 2016], we use pointer based generation instead of Seq2Seq generation and show that our approach achieves higher performance, especially for rare words and column names. Unlike [Neelakantan et al., 2017], our model does not require access to the table content during inference, which may be unavailable due to privacy concerns. In contrast to both methods, we train our model using policy-based RL, which again improves performance.

Natural language interface for databases. The practical applications of WikiSQL have much to do with Natural Language Interfaces (NLI). One of the prominent works in this area is PRE-

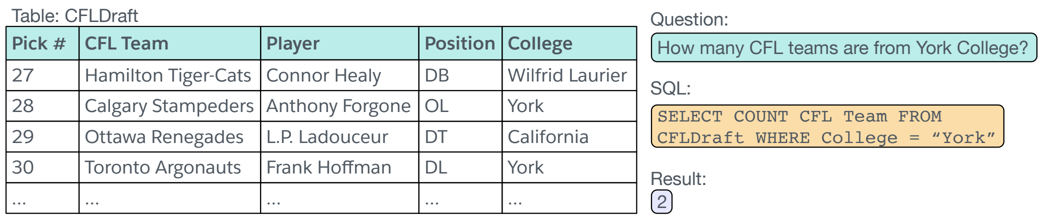

Table: CFLDraft

Pick #	CFL Team	Player	Position	College
27	Hamilton Tiger-Cats	Connor Healy	DB	Wilfrid Laurier
28	Calgary Stampeders	Anthony Forgone	OL	York
29	Ottawa Renegades	L.P. Ladouceur	DT	California
30	Toronto Argonauts	Frank Hoffman	DL	York
...	...	...	...	...

Question:
How many CFL teams are from York College?

SQL:
```
SELECT COUNT CFL Team FROM
CFLDraft WHERE College = “York”
```

Result:
2

Figure 2: An example in WikiSQL. The inputs consist of a table and question. The outputs consist of a ground truth SQL query and the corresponding result from execution.

CISE [Popescu et al., 2003], which translates questions to SQL queries and identifies questions that it is not confident about. Giordani and Moschitti [2012] proposed a system to translate questions to SQL by first generating candidate queries from a grammar then ranking them using tree kernels. Both of these approaches rely on a high quality grammar and are not suitable for tasks that require generalization to new schema. Iyer et al. [2017] also proposed a system to translate to SQL, but with a Seq2Seq model that is further improved with human feedback. Compared to this model, we show that Seq2SQL substantially outperforms Seq2Seq and uses reinforcement learning instead of human feedback during training.

3 Model

The Seq2SQL model generates a SQL query from a natural language question and table schema.[1] One powerful baseline model is the Seq2Seq model utilized for machine translation [Bahdanau et al., 2015]. However, the output space of the standard softmax in a Seq2Seq model is unnecessarily large for this task. In particular, we note that the problem can be dramatically simplified by limiting the output space of the generated sequence to the union of the table schema, the question utterance, and SQL key words. The resulting model is similar to a pointer network [Vinyals et al., 2015] with augmented inputs. We first show this augmented pointer network model, then address its limitations, particularly with respect to generating unordered query conditions, in our description of Seq2SQL.

3.1 Augmented Pointer Network

The augmented pointer network generates the SQL query token-by-token by selecting from an input sequence. In our case, the input sequence is the concatenation of the column names, required for the selection column and the condition columns of the query, the question, required for the conditions of the query, and the limited vocabulary of the SQL language such as `SELECT`, `COUNT` etc. In the example shown in Figure 2, the column name tokens consist of “Pick”, “#”, “CFL”, “Team” etc.; the question tokens consist of “How”, “many”, “CFL”, “teams” etc.; the SQL tokens consist of `SELECT`, `WHERE`, `COUNT`, `MIN`, `MAX` etc. With this augmented input sequence, the pointer network is able to fully produce the SQL query by selecting exclusively from the input.

Suppose we are given a list of N table columns and a question such as in Figure 2, and asked to produce the corresponding SQL query. Let $x^{\mathrm{c}}_j = [x^{\mathrm{c}}_{j,1}, x^{\mathrm{c}}_{j,2}, ...x^{\mathrm{c}}_{j,T_j}]$ denote the sequence of words in the name of the jth column, where $x^{\mathrm{c}}_{j,i}$ represents the ith word in the jth column and T_j represents the total number of words in the jth column. Similarly, let x^{q} and x^{s} respectively denote the sequence of words in the question and the set of all unique words in the SQL vocabulary.

We define the input sequence x as the concatenation of all the column names, the question, and the SQL vocabulary:

$$x = [\texttt{<col>}; x^{\mathrm{c}}_1; x^{\mathrm{c}}_2; ...; x^{\mathrm{c}}_N; \texttt{<sql>}; x^{\mathrm{s}}; \texttt{<question>}; x^{\mathrm{q}}] \tag{1}$$

where $[a; b]$ denotes the concatenation between the sequences a and b and we add special sentinel tokens between neighbouring sequences to demarcate the boundaries.

The network first encodes x using a two-layer, bidirectional Long Short-Term Memory network [Hochreiter and Schmidhuber, 1997]. The input to the encoder are the embeddings corresponding to words in the input sequence. We denote the output of the encoder by h, where h_t is the state of the encoder corresponding to the tth word in the input sequence. For brevity, we do not write

[1]For simplicity, we define table schema as the names of the columns in the table.

out the LSTM equations, which are described by Hochreiter and Schmidhuber [1997]. We then apply a pointer network similar to that proposed by Vinyals et al. [2015] to the input encodings h.

The decoder network uses a two layer, unidirectional LSTM. During each decoder step s, the decoder LSTM takes as input y_{s-1}, the query token generated during the previous decoding step, and outputs the state g_s. Next, the decoder produces a scalar attention score $\alpha^{\text{ptr}}_{s,t}$ for each position t of the input sequence:

$$\alpha^{\text{ptr}}_{s,t} = W^{\text{ptr}} \tanh\left(\mathrm{U}^{\text{ptr}} \mathrm{g_s} + \mathrm{V}^{\text{ptr}} \mathrm{h_t}\right) \tag{2}$$

The input token with the highest score is then chosen by the decoder to be the next token of the SQL query, $y_s = \operatorname{argmax}(\alpha^{\text{ptr}}_s)$.

3.2 Seq2SQL

While the augmented pointer model is able to solve the SQL generation problem, it does not leverage the structure inherent in SQL queries. Normally, a SQL query such as that shown in Figure 3 consists of three components. The first component is the aggregation operator, in this case `COUNT`, which produces a summary of the rows selected by the SQL query. Alternatively the query may request no summary statistics, in which case an aggregation operator is not provided. The second component is the `SELECT` column(s), in this case `Engine`, which identifies the column(s) that are to be included in the returned results. The third component is the `WHERE` clause of the query, in this cases `WHERE Driver = Val Musetti`, which contains conditions by which to filter the rows. Here, we want only rows in which the driver is "Val Musetti".

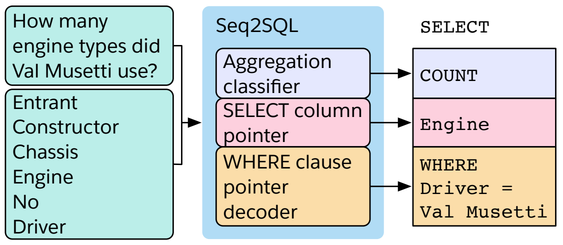

Figure 3: The Seq2SQL model has three components, corresponding to the three parts of a SQL query (right). The input to the model are the question (top left) and the table column names (bottom left).

To leverage the structure present in SQL queries, we introduce Seq2SQL. Seq2SQL, as shown in Figure 3, is composed of three parts that correspond to the aggregation operator, the `SELECT` column, and the `WHERE` clause. First, the network classifies an aggregation operation for the query, with the addition of a null operation that corresponds to no aggregation. Next, the network points to a column in the input table corresponding to the `SELECT` column. Finally, the network generates the conditions for the query using a pointer network. The first two components are supervised using cross entropy loss, whereas the third generation component is trained using policy gradient reinforcement learning in order to address the unordered nature of query conditions (we explain this in detail in Section 3.2). Utilizing the structure of a SQL query allows Seq2SQL to further reduce the output space of SQL queries, which we show leads to a higher performance compared to both the Seq2Seq model and the pointer model.

Aggregation Operation. The aggregation operation depends on the question. For the example shown in Figure 3, the correct operator is `COUNT` because the question asks for "How many".

To compute the aggregation operation, we first compute an input representation κ^{agg}. Similar to Equation 2, we first compute the scalar attention score, $\alpha^{\text{inp}}_t = W^{\text{inp}} h^{\text{enc}}_t$, for each tth token in the input sequence. The vector of all scores, $\alpha^{\text{inp}} = [\alpha^{\text{inp}}_1, \alpha^{\text{inp}}_2, ...]$, is then normalized to produce a distribution over all the tokens in the input sequence x, $\beta^{\text{inp}} = \operatorname{softmax}\left(\alpha^{\text{inp}}\right)$. The input representation, κ^{agg}, is the sum over the column encodings weighted by the corresponding normalized score:

$$\kappa^{\text{agg}} = \sum_t \beta^{\text{inp}}_t h^{\text{enc}}_t \tag{3}$$

Let α^{agg} denote the scores over the aggregation operations such as `COUNT`, `MIN`, `MAX`, and the no-aggregation operation `NULL`. We compute α^{agg} by applying a multi-layer perceptron to the input representation κ^{agg}:

$$\alpha^{\text{agg}} = W^{\text{agg}} \tanh\left(V^{\text{agg}} \kappa^{\text{agg}} + b^{\text{agg}}\right) + c^{\text{agg}} \tag{4}$$

The distribution over the set of possible aggregation operations, $\beta^{\text{agg}} = \operatorname{softmax}(\alpha^{\text{agg}})$, is obtained by applying the softmax function. We use cross entropy loss L^{agg} for the aggregation operation.

SELECT Column. The selection column depends on the table columns as well as the question. Namely, for the example in Figure 3, we see from the "How many engine types" part of the question that the query is asking for the "Engine" column. `SELECT` column prediction is then a matching problem, solvable using a pointer: given the list of column representations and a question representation, we select the column that best matches the question.

In order to produce the representations for the columns, we first encode each column name with a LSTM. The representation of a particular column j, e^{c}_j, is given by:

$$h^{\text{c}}_{j,t} = \operatorname{LSTM}\left(\operatorname{emb}\left(x^{\text{c}}_{j,t}\right), h^{\text{c}}_{j,t-1}\right) \qquad e^{\text{c}}_j = h^{\text{c}}_{j,T_j} \tag{5}$$

Here, $h^{\text{c}}_{j,t}$ denotes the tth encoder state of the jth column. e^{c}_j, column j's representation, is then taken to be the last encoder state.

To construct a representation for the question, we compute another input representation κ^{sel} using the same architecture as for κ^{agg} (Equation 3) but with untied weights. Finally, we apply a multi-layer perceptron over the column representations for the pointer estimation, conditioned on the input representation, to compute the a score for each column j:

$$\alpha^{\text{sel}}_j = W^{\text{sel}} \tanh\left(V^{\text{sel}}\kappa^{\text{sel}} + V^{\text{c}} e^{\text{c}}_j\right) \tag{6}$$

We normalize the scores with a softmax function to produce a distribution over the possible `SELECT` columns $\beta^{\text{sel}} = \operatorname{softmax}\left(\alpha^{\text{sel}}\right)$. For the example shown in Figure 3, the distribution would be over the columns "Entrant", "Constructor", "Chassis", "Engine", "No", and the ground truth `SELECT` column "Driver". We train the `SELECT` network using cross entropy loss L^{sel}.

WHERE Clause. The `WHERE` clause can be trained using a pointer decoder similar to that described in Section 3.1. However, there is a crucial limitation in using the cross entropy loss to optimize the network: the `WHERE` conditions of a query can be arbitrarily swapped and the query still yield the same result. Suppose we have the question "which men are older than 18" and the queries `SELECT name FROM insurance WHERE age > 18 AND gender = "male"` and `SELECT name FROM insurance WHERE gender = "male" AND age > 18`. Both queries obtain the correct execution result despite not having exact string match. If the former query is provided as the ground truth, using cross entropy loss to supervise the generation would then wrongly penalize the latter query. To address this problem, we apply reinforcement learning to learn a policy to directly optimize the expected "correctness" of the execution result (Equation 7).

Instead of teacher forcing at each step, we sample from the output distribution to obtain the next token. At the end of the generation procedure, we execute the generated SQL query against the database to obtain a reward. Let $q(y)$ denote the query generated by the model and q_g denote the ground truth query corresponding to the question. The reward $R(q(y), q_g)$ is defined as:

$$R(q(y), q_g) = \begin{cases} -2, & \text{if } q(y) \text{ is not a valid SQL query} \\ -1, & \text{if } q(y) \text{ is a valid SQL query and executes to an incorrect result} \\ +1, & \text{if } q(y) \text{ is a valid SQL query and executes to the correct result} \end{cases} \tag{7}$$

Let $y = [y^1, y^2, ..., y^T]$ denote the sequence of generated tokens in the `WHERE` clause of a SQL query $q(y)$ that resulted in an execution reward of $R(q(y), q_g)$. The loss, $L^{\text{whe}} = -\mathbb{E}_y[R(q(y), q_g)]$, is the negative expected reward over possible `WHERE` clauses.

As described by Schulman et al. [2015], the act of sampling during the forward pass of the network can be followed by the corresponding injection of a synthetic gradient, which is a function of the reward, during the backward pass in order to compute estimated parameter gradient. Specifically, if we let $P_t(y^t)$ denote the probability of choosing token y^t during time step t, the corresponding synthetic gradient is $R(q(y), q_g) \log P_t(y^t)$.

Mixed Objective Function. The model is trained using gradient descent to minimize the objective function $L = L^{\text{agg}} + L^{\text{sel}} + L^{\text{whe}}$. The total gradient is then the equally weighted sum of the gradients

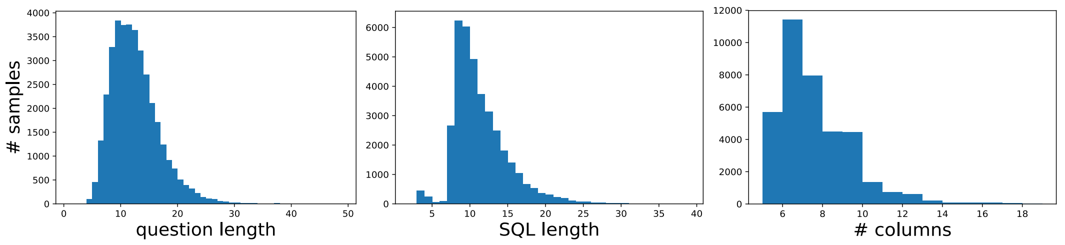

Figure 5: Distribution of numbers of columns, tokens per question, and tokens per query in WikiSQL.

from the cross entropy loss in predicting the SELECT column, the gradients from the cross entropy loss in predicting the aggregation operation, and the gradient estimate from the policy learning reward described above.

4 WikiSQL

WikiSQL is a collection of questions, corresponding SQL queries, and SQL tables. It is the largest available corpus of its kind to date. A single example in WikiSQL, shown in Figure 2, contains a table, a SQL query, and the natural language question corresponding to the SQL query.

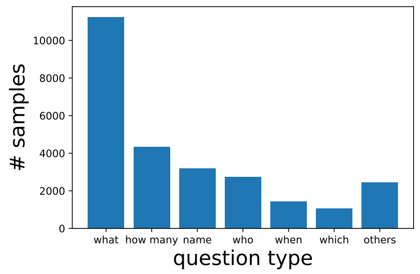

Figure 4: Distribution of question types in WikiSQL.

Table 1 shows how WikiSQL compares to related datasets. Namely, we note that WikiSQL is the largest hand-annotated semantic parsing dataset to date - it is an order of magnitude larger than other datasets that have logical forms, either in terms of the number of examples or the number of table schemas. Moreover, the queries in WikiSQL span over a large number of table schemas and cover a large diversity of data. The large quantity of schemas presents a challenge compared to other datasets: the model must be able to not only generalize to new queries, but to new schemas as well. Finally, WikiSQL contains challenging, realistic data extracted from the web. This is evident in the distributions of the number of columns, the lengths of questions, and the length of queries, respectively shown in Figure 5. Another indicator of the variety of questions in the dataset is the distribution of question types, shown in Figure 4.

WikiSQL is collected using crowd-sourcing on Amazon Mechanical Turk (AMT) in two phases. First, a worker is asked to paraphrase a generated question for a table. The generated question itself is formed using a template, filled using a randomly generated SQL query. We ensure the validity and complexity of the tables by keeping only those that are legitimate database tables and sufficiently large in the number of rows and columns. Next, two other workers are asked to verify that the paraphrase has the same meaning as the generated question. We discard paraphrases that do not show enough variation, as measured by the character edit distance from the generated question, as well as those that are deemed incorrect by both workers during verification. More details on how WikiSQL was collected is shown in Section 1 of the Appendix. We make available examples of the interface used during the paraphrase phase and during the verification phase in the supplementary materials. The dataset is available for download at https://github.com/MetaMind/wikisql.

Dataset	Size	LF	Schema
WikiSQL	**87726**	**yes**	**26375**
Geoquery	880	yes	8
ATIS	5871	yes*	141
Freebase917	917	yes	81*
Overnight	26098	yes	8
WebQuestions	5810	no	2420
WikiTableQuestions	22033	no	2108

Table 1: Comparison between WikiSQL and existing datasets. The datasets are GeoQuery880 [Tang and Mooney, 2001], ATIS [Price, 1990], Free917 [Cai and Yates, 2013], Overnight [Wang et al., 2015], WebQuestions [Berant et al., 2013], and WikiTableQuestions [Pasupat and Liang, 2015]. "Size" denotes the number of examples in the dataset. "LF" indicates whether it has annotated logical forms. "Schema" denotes the number of table schemas. ATIS is presented as a slot filling task. Each Freebase API page is counted as a separate domain.

The tables, their paraphrases, and SQL queries are randomly slotted into train, dev, and test splits, such that each table is present in exactly one split. In addition to the raw tables, queries, results, and natural utterances, we also release a corresponding SQL database and query execution engine.

4.1 Evaluation

Let N denote the total number of examples in the dataset, N_{ex} the number of queries that, when executed, result in the correct result, and N_{lf} the number of queries has exact string match with the ground truth query used to collect the paraphrase.

We evaluate using the execution accuracy metric $\text{Acc}_{\text{ex}} = \frac{N_{\text{ex}}}{N}$. One downside of Acc_{ex} is that it is possible to construct a SQL query that does not correspond to the question but nevertheless obtains the same result. For example, the two queries `SELECT COUNT(name) WHERE SSN = 123` and `SELECT COUNT(SSN) WHERE SSN = 123` produce the same result if no two people with different names share the SSN 123.

Hence, we also use the logical form accuracy $\text{Acc}_{\text{lf}} = \frac{N_{\text{lf}}}{N}$. However, as we showed in Section 3.2, Acc_{lf} incorrectly penalizes queries that achieve the correct result but do not have exact string match with the ground truth query. Due to these observations, we use both metrics to evaluate our models.

5 Experiments

The dataset is tokenized using Stanford CoreNLP [Manning et al., 2014]. In particular, we use the normalized tokens for training and reverse them into their original gloss before outputting the query. This way, the queries are composed of tokens from the original gloss and are hence executable in the database.

We use, and keep fixed, GloVe word embeddings [Pennington et al., 2014] and character n-gram embeddings [Hashimoto et al., 2016]. Let w_x^{g} denote the GloVe embedding and w_x^{c} the character embedding for word x. Here, w_x^{c} is the mean of the embeddings of all the character n-grams in x. For a given word x, we define its embedding $w_x = [w_x^{\text{g}}; w_x^{\text{c}}]$. For words that have neither word nor character embeddings, we assign the zero vector.

All networks are run for a maximum of 100 epochs with early stopping on dev split execution accuracy. We train using ADAM [Kingma and Ba, 2014] and regularize using dropout [Srivastava et al., 2014]. We implement all models using PyTorch [2].

To train Seq2SQL, we first pretrain a version in which the `WHERE` clause is supervised via teacher forcing (i.e. the sampling procedure is not trained from scratch) and then continue training using reinforcement learning. In order to obtain the rewards described in Section 3.2, we use the query execution engine described in Section 4.

5.1 Result

We compare results against an attentional sequence to sequence baseline used by Bahdanau et al. [2015] for machine translation. This model is described in detail in Section 2 of the Appendix. The performance of the three models are shown in Table 2.

Reducing the output space by utilizing the augmented pointer network significantly improves upon the attentional sequence to sequence model by 16.9%. Moreover, leveraging the structure of SQL queries leads to another significant improvement of 4.8%, as is shown by the performance of Seq2SQL without RL compared to the augmented pointer network. Finally, the full Seq2SQL model, trained using reinforcement learning based on rewards from in-the-loop query executions on a database, leads to yet another significant performance increase of 2.7%.

5.2 Analysis

Limiting the output space lead to more accurate conditions. Compared to Seq2Seq, the augmented pointer model generates much higher quality `WHERE` clause conditions. For example, for

[2]https://pytorch.org

Model	Dev Acc_{lf}	Dev Acc_{ex}	Test Acc_{lf}	Test Acc_{ex}
Attentional Seq2Seq	23.3%	37.0%	23.4%	35.9%
Aug. Pointer Network	44.1%	53.8%	42.8%	52.8%
Seq2SQL (no RL)	47.7%	57.9%	47.2%	57.6%
Seq2SQL	**49.8%**	**60.7%**	**49.2%**	**60.3%**

Table 2: Performance on WikiSQL. Both metrics are defined in Section 4.1. The Seq2SQL (no RL) model is identical in architecture when compared to Seq2SQL, except the `WHERE` clause is supervised via teacher forcing as opposed to reinforcement learning.

the question "in how many districts was a successor seated on march 4, 1850?", Seq2Seq generates `WHERE Date successor seated = seated march 4` whereas Seq2SQL generates `WHERE Date successor seated = seated march 4 1850`. Similarly, for the question "what's doug battaglia's pick number?", the Seq2Seq produces `WHERE Player = doug` whereas Seq2SQL produces `WHERE Player = doug battaglia`. A reason for this is that conditions tend to be comprised of rare words (e.g. the infrequent "1850"). The Seq2Seq model is then inclined to produce a condition that contains frequently occurring tokens in the training corpus, such as "march 4" for date, or "doug" for player name. This problem does not affect the the pointer network as much, since the output space is constructed from the input sequence and thus orders of magnitude smaller.

Incorporating structure reduces invalid queries. As expected, incorporating the simple selection and aggregation structure into the model reduces the percentage of invalid SQL queries generated from 39.8% to 7.1%. Generally, we find that a large quantity of invalid queries result from invalid column names. That is, the model generates a query that refers to columns that are not present in the table. This is particularly helpful when the names of columns contain many tokens, such as "Miles (km)", which is comprised of 4 tokens after tokenization. Introducing a specialized classifier for the aggregation also reduces the error rate due to having the incorrect aggregation operator. Table 3 shows that adding the aggregation classifier substantially improves the recall and F1 for predicting the `COUNT` operator. For more queries produced by the different models, please see Section 3 of the Appendix.

Model	Precision	Recall	F1
Aug. Pointer	87.0%	47.8%	61.7%
Seq2SQL	85.9%	79.7%	82.7%

Table 3: Performance on the `COUNT` operator.

Reinforcement learning generates higher quality `WHERE` clause that are ordered differently than ground truth. As expected, training with policy-based RL obtains correct results in which the order of the conditions is different than the order in the ground truth query. For example, for the question "in what district was the democratic candidate first elected in 1992?", the ground truth conditions are `WHERE First elected = 1992 AND Party = Democratic` whereas Seq2SQL generates `WHERE Party = Democratic AND First elected = 1992`. We also note that when Seq2SQL is correct and Seq2SQL without policy learning is not, the latter tends to produce an incorrect `WHERE` clause. For example, for the rather complex question "what is the race name of the 12th round trenton, new jersey race where a.j. foyt had the pole position?", Seq2SQL trained without RL generates `WHERE rnd = 12 and track = a.j. foyt AND pole position = a.j. foyt` whereas Seq2SQL trained with RL correctly generates `WHERE rnd = 12 AND pole position = a.j. foyt`.

6 Conclusion

We proposed Seq2SQL, a deep neural network for translating questions to SQL queries. Our model leverages the structure of SQL queries to reduce the output space of the model. As a part of Seq2SQL, we applied in-the-loop query execution to learn a policy for generating the conditions of the SQL query, which is unordered in nature and unsuitable for optimization via cross entropy loss. We also introduced WikiSQL, a dataset of questions and SQL queries that is an order of magnitude larger than comparable datasets. Finally, we showed that Seq2SQL outperforms attentional sequence to sequence models on WikiSQL, improving execution accuracy from 35.9% to 60.3% and logical form accuracy from 23.4% to 49.2%.

References

I. Androutsopoulos, G. Ritchie, and P. Thanisch. Natural language interfaces to databases - an introduction. 1995.

Y. Artzi and L. S. Zettlemoyer. Bootstrapping semantic parsers from conversations. In *EMNLP*, 2011.

Y. Artzi and L. S. Zettlemoyer. Weakly supervised learning of semantic parsers for mapping instructions to actions. *TACL*, 1:49–62, 2013.

D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *ICLR*, 2015.

T. Beck, A. Demirgüç-Kunt, and R. Levine. A new database on the structure and development of the financial sector. *The World Bank Economic Review*, 14(3):597–605, 2000.

I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural combinatorial optimization with reinforcement learning. *ICRL*, 2017.

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

Q. Cai and A. Yates. Large-scale semantic parsing via schema matching and lexicon extension. In *ACL*, 2013.

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

A. Giordani and A. Moschitti. Translating questions to SQL queries with generative parsers discriminatively reranked. In *COLING*, 2012.

J. Gu, Z. Lu, H. Li, and V. O. K. Li. Incorporating copying mechanism in sequence-to-sequence learning. *ACL*, 2016.

K. Hashimoto, C. Xiong, Y. Tsuruoka, and R. Socher. A Joint Many-Task Model: Growing a Neural Network for Multiple NLP Tasks. *arXiv*, cs.CL 1611.01587, 2016.

R. Hillestad, J. Bigelow, A. Bower, F. Girosi, R. Meili, R. Scoville, and R. Taylor. Can electronic medical record systems transform health care? potential health benefits, savings, and costs. *Health affairs*, 24(5):1103–1117, 2005.

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

S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer. Learning a neural semantic parser from user feedback. In *ACL*, 2017.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv*, abs/1412.6980, 2014.

P. Liang, M. I. Jordan, and D. Klein. Learning dependency-based compositional semantics. *Computational Linguistics*, 39:389–446, 2011.

C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

S. Merity, C. Xiong, J. Bradbury, and R. Socher. Pointer sentinel mixture models. *ICLR*, 2017.

A. Neelakantan, Q. V. Le, M. Abadi, A. McCallum, and D. Amodei. Learning a natural language interface with neural programmer. In *ICLR*, 2017.

E. W. Ngai, L. Xiu, and D. C. Chau. Application of data mining techniques in customer relationship management: A literature review and classification. *Expert systems with applications*, 36(2): 2592–2602, 2009.

P. Pasupat and P. Liang. Compositional semantic parsing on semi-structured tables. In *ACL*, 2015.

J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *EMNLP*, 2014.

A.-M. Popescu, O. Etzioni, and H. Kautz. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th International Conference on Intelligent User Interfaces*, pages 149–157. ACM, 2003.

P. J. Price. Evaluation of spoken language systems: The ATIS domain. 1990.

S. Reddy, M. Lapata, and M. Steedman. Large-scale semantic parsing without question-answer pairs. *TACL*, 2:377–392, 2014.

J. Schulman, N. Heess, T. Weber, and P. Abbeel. Gradient estimation using stochastic computation graphs. In *NIPS*, 2015.

M. J. Seo, A. Kembhavi, A. Farhadi, and H. Hajishirzi. Bidirectional attention flow for machine comprehension. *ICLR*, 2017.

N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15: 1929–1958, 2014.

J. Starc and D. Mladenic. Joint learning of ontology and semantic parser from text. *Intelligent Data Analysis*, 21:19–38, 2017.

L. R. Tang and R. J. Mooney. Using multiple clause constructors in inductive logic programming for semantic parsing. In *ECML*, 2001.

O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. In *NIPS*, 2015.

Y. Wang, J. Berant, and P. Liang. Building a semantic parser overnight. In *ACL*, 2015.

Y. W. Wong and R. J. Mooney. Learning synchronous grammars for semantic parsing with lambda calculus. In *ACL*, 2007.

C. Xiong, V. Zhong, and R. Socher. Dynamic coattention networks for question answering. *ICRL*, 2017.

J. M. Zelle and R. J. Mooney. Learning to parse database queries using inductive logic programming. In *AAAI/IAAI, Vol. 2*, 1996.

L. S. Zettlemoyer and M. Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Uncertainty in Artificial Intelligence*, 2005.

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