POINTING OUT SQL QUERIES FROM TEXT

Anonymous authors
Paper under double-blind review

ABSTRACT

The digitization of data has resulted in making datasets available to millions of users in the form of relational databases and spreadsheet tables. However, majority of these users come from diverse backgrounds and lack the programming expertise to query and analyze data in such tables. In this paper, we present a system that allows for querying the tabular data using natural language questions, where the system translates the question into an executable SQL query. The system uses a deep sequence to sequence model, where the decoder uses a simple type system of the SQL expressions to structure the output prediction. Based on the type, the decoder either copies the output from the input question using an attention-based copying mechanism or generates the output from a fixed vocabulary. We introduce a sum-transfer value-based loss function that transforms the distribution over pointer locations into distribution over input vocabularies pointed to by the pointers to efficiently train the architecture. We evaluate our model on the recently released WikiSQL dataset and show that our model trained using only supervised learning significantly outperforms current state-of-the-art Seq2SQL model that uses reinforcement learning.

1 INTRODUCTION

The IT revolution in past few decades has resulted in large-scale digitization of data and making it accessible to millions of users in the form of databases and spreadsheet tables. Despite advances in making this data more accessible to users by designing new high-level programming languages and user interfaces, querying and analyzing such tables still requires users to write complex programs in languages such as SQL or Excel, which is unfortunately beyond the programming expertise of majority of end-users who come from a diverse set of backgrounds (Gualtieri, 2009). Thus, building effective semantic parsers that can translate natural language questions into computer executable programs has been a long standing goal to improve end-user data accessibility (Poon, 2013; Zettlemoyer & Collins, 2005; Pasupat & Liang, 2015; Li et al., 2005; Gulwani & Marron, 2014).

Recent work has shown that recurrent neural networks with proper attention and copying mechanisms (Dong & Lapata, 2016; Neelakantan et al., 2016; Jia & Liang, 2016) can be used effectively to build successful semantic parsers. Notably, Zhong et al. (2017) recently introduced the state-of-the-art model (Seq2SQL model) for question to SQL translation in the supervised setting, where programs are explicitly provided with their corresponding questions. Seq2SQL model shows that using separate decoders for different parts of a query (i.e., aggregation operation, target column, and where predicates) increases prediction accuracy, and reinforcement learning further improves the model by allowing it to learn semantically equivalent queries beyond supervision.

In this paper, we present a new encoder-decoder model that is an extension of the attentional seq2seq model for natural language to SQL program translation and a training approach that is capable of learning the model in an effective and stable manner. Figure 1 shows an example table-question pair and how our system generates the answer by executing the synthesized SQL program.

First, we designed a simple static *type system* to control the decoding mode at each decoding step: based on the SQL grammar, a decoder cell is labeled to either (1) select a token from the SQL built-in vocabulary, (2) generate a pointer over the table header and the input question to copy a table column, or (3) generate a pointer to copy a constant from the user's question. The type system allows us to have a fine-grain control over the decoding process while retaining the simplicity and extensibility of

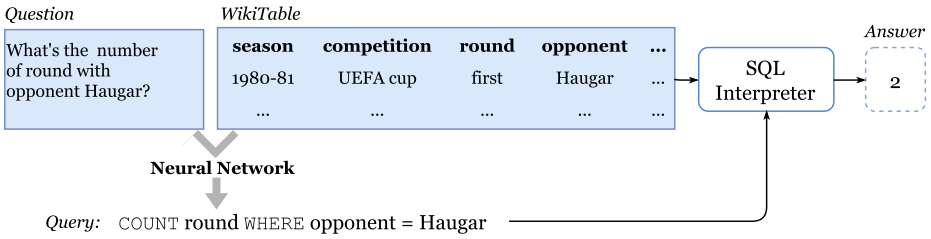


Figure 1: Our system answers a table question by synthesizing a query and then interpret it on the provided table.

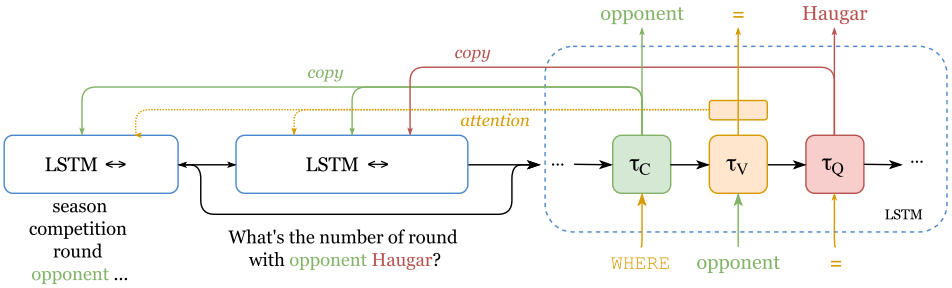


Figure 2: Model overview for the example in Figure 1. The model encodes table columns as well as the user question with a bidirectional LSTM and then decodes the hidden state with a typed LSTM, where the decoding action for each cell is statically determined.

the sequence structure, as opposed to designing multiple decoders for different language components or adding extra controllers for expansion of production rules (Krishnamurthy et al., 2017).

Second, we provide an effective approach to train our model with inferred pointers (for decoders of second and third types). The challenge for training pointer decoders comes from the fact the output value to be copied appears in multiple places in the input (i.e. both in the question and the table headers), and the "correct" pointer is not explicitly provided in question-program pairs regarding which occurrence or the token should the model copy from in the given context. Our solution to the problem is to use a new value-based loss function that transfers the distribution over the pointer locations in the input into distributions over input vocabularies pointed to by these pointers, by summing up the probabilities of the same vocabulary value appearing at different input indices. Our result shows that our training strategy performs the best compared to other alternatives (e.g., direct supervision on pointers). Especially, this approach allows us to consistently reproduce high accuracy models with random initializations.

We evaluate our approach though a set of experiments on the recently released WikiSQL dataset (Zhong et al., 2017), a corpus consisting of over 80,000 natural language question and pairs. Our result shows that our model that uses only supervised learning can significantly outperform the current state-of-the-art Seq2SQL model (Zhong et al., 2017) that uses specialized decoder and reinforcement learning (**59.5%** vs 48.3% for exact syntactic match and **65.1%** vs 59.4% for execution accuracy). Also, with a series of ablation experiments, we analyze the influence of different components of our model on the overall results.

2 MODEL

While generating the SQL query, we use the known structure of the DSL to modify a standard RNN-based encoder-decoder model with attention and copying mechanisms (Vinyals et al., 2015; Gu et al., 2016; Zhong et al., 2017). As shown in Figure 2, we *statically* determine the "type" of output of a decoding step based on the SQL grammar, e.g., we know that in our target language, the third token (after the aggregation function) is always a column name specifying the aggregation

target. Thus, when decoding, we first determine the type of the next token to generate, and then use a specialized decoder to generate the output: if we have to produce a column name or a constant, we enforce the use of a copying mechanism, whereas we use standard selection from a built-in vocabulary for the remaining tokens. This means that our decoder vocabulary only needs to contain built-in tokens and thus has size 18.

2.1 ENCODER

Our encoder is a bidirectional recurrent neural network (RNN) using Long Short-Term Memory (LSTM) cells, using a character-based embedding strategy for individual tokens. As input tokens, we use the concatenation of the table header (i.e., the column names) of the queried table and the user query, i.e., $X = [x_c^{(1)}, \ldots, x_c^{(C)}, x_q^{(1)}, \ldots, x_q^{(Q)}]$. This concatenation allows the model to learn how to compute a joint representation for both columns and the input query. We use $|X|$ to represent the input sequence length (equal to $C + Q$).

Token Embedding To handle the large number of different tokens in the input query, we construct a token embedding using a pre-trained character n-gram embedding along with a pre-trained global word embedding. For a token x in the input sequence X, we compute its embedding $emb_e(x)$ as the concatenation of its word embedding and the average embeddings of all n-gram features contained in x. Formally, if W_{word} is a pre-trained word model, $x[i, j]$ is the character sequence from i to j in x, $W_{n\text{-gram}}$ is a pre-trained n-gram for the n-gram feature set V, and N_x is the number of n-gram features contained in the word, then

$$emb_e(x) = \big(W_{\text{word}}(x),\ \frac{1}{N_x} \sum_{\substack{1 \le i < j \le |x| \\ x[i,j] \in V}} W_{n\text{-gram}}(x[i, j])\ \big).$$

We use the pre-trained n-gram model by Hashimoto et al. (2017) and GloVe embedding (Pennington et al., 2014) for words. Furthermore, we freeze the n-gram weights during training to avoid overfitting. This embedding method is also consistent with Zhong et al. (2017).

Bidirectional RNN We feed embedded tokens into a bidirectional RNN composed of LSTM cells $\mathcal{C}_{e,fw}, \mathcal{C}_{e,bw}$, computing

$$(o_{e,fw}^{(k+1)}, h_{e,fw}^{(k+1)}) = \mathcal{C}_{e,fw}(h_{e,fw}^{(k)}, emb_e(x^{(k)})) \qquad (o_{e,bw}^{(k)}, h_{e,bw}^{(k)}) = \mathcal{C}_{e,bw}(h_{e,bw}^{(k+1)}, emb_e(x^{(k)})),$$

and will use the sequence $O_e = [o_e^{(1)}, \ldots, o_e^{(|X|)}]$ for $o_e^{(k)} = (o_{e,fw}^{(k)}, o_{e,bw}^{(k)})$ as the learned representation of token $x^{(k)}$ for the attention and copying mechanisms of our decoder. We initialize the forward encoder with hidden states $h_{e,fw}^{(0)} = \mathbf{0}$ and initialize the backward encoder with $h_{e,bw}^{(|X|)}$, the last hidden state of encoder $h_{e,fw}^{(|X|)}$.

2.2 TYPED DECODER

Output Grammar Our model use types abstracted from the grammar of the target language to improve the decoding performance. Concretely, we know that the subset of SQL necessary to answer WikiSQL Questions can be represented using the following grammar, in which t refers to the name of the table being queried, c refers to a column name in the table, and v refers to any open world string or number that may be used in the query:

$$\begin{array}{rcl} Q & \rightarrow & s\ c\ \texttt{From}\ t\ \texttt{Where}\ p \\ s & \rightarrow & \texttt{Select} \mid \texttt{Max} \mid \texttt{Min} \mid \texttt{Count} \mid \texttt{Sum} \mid \texttt{Avg} \\ p & \rightarrow & c\ op\ v \mid p\ \texttt{And}\ p \\ op & \rightarrow & = \mid \neq \mid > \mid \geq \mid < \mid \leq \end{array}$$

A consequence of this observation is that we can, based on the tokens generated so far, determine the "type" of the next token to generate. For example, after generating the two tokens "`Select`

`Id`", we know that the following token must be one of the column names from the queried table. We found it sufficient to distinguish three different cases by types:

$\tau_{\mathcal{V}}$ The output is a token from the terminals $\mathcal{V} = \{$`Select`, `From`, `Where`, `Id`, `Max`, `Min`, `Count`, `Sum`, `Avg`, `And`, $=, \neq, >, \geq, <, \leq$, `<END>`, `<GO>`$\}$ of our grammar.

τ_C The output has to be a column name, which will be copied from either the table header or the question section of X. Note that the column required for the correct query may not be mentioned explicitly in the question.

τ_Q The output is a constant that would be copied from the question section of I.

The output types can be described using the regular expression $\tau_{\mathcal{V}}\tau_{\mathcal{V}}\tau_C\tau_{\mathcal{V}}\tau_C\tau_{\mathcal{V}}(\tau_C\tau_{\mathcal{V}}\tau_Q)^*$. We can then use the type of the output token we want to generate to specialize the decoder.

Decoder RNN We use a standard RNN, based on an LSTM cell with attention over O_e to generate the target program O. Notably, we initialize the decoder from both the final hidden states $h^{(0)}_{e,bw}, h^{(|X|)}_{e,fw}$ and the hidden states $h^{(C)}_{e,fw}, h^{(C)}_{e,bw}$ generated at index C, the index of the end of the table header in X. This state forwarding strategy allows the decoder to directly access the encoding of column names to improve decoding accuracy. Using $i^{(k)}_d$, $o^{(k)}_d$ and $h^{(k)}_d$ to denote the input (resp. output, hidden state) of the LSTM cell at decoding step k, we define three different output layers for our three output types:

$\tau_{\mathcal{V}}$ We define $u^{(k,\ell)} = v^T \tanh(W_h h^{(k)}_d + W_o o^{(\ell)}_e)$ using learnable parameters W_h, W_o, $b_{\mathcal{V}}$ and use it compute an attention mask $\alpha^{(k)} = \operatorname{softmax}([u^{(k,1)} \dots u^{(k,I)}])$. The chosen output token $o^{(k)}_d$ is then computed as $o^{(k)}_d = \operatorname{argmax}(W_{\mathcal{V}}(O_e\alpha^{(k)}) + b_{\mathcal{V}})$, where $W_{\mathcal{V}}, b_{\mathcal{V}}$ are trainable variables for $\tau_{\mathcal{V}}$ decoding, and $O_e\alpha^{(k)}$ is the attention vector.

Then, the input to the next decoder cell is $i^{(k+1)}_d = (emb_d(o^{(k)}_d), O_e\alpha^{(k)})$, the concatenation of the token embedding and the attention vector.

τ_C, τ_Q We use the same approach to compute the attention mask $\alpha^{(k)}$. However, instead of projecting $O_e\alpha^{(k)}$ to obtain the output, the model generates $o^{(k)}_d$ by copying a token v from the input sequence X. The index l of the token to copy is calculated by $l = \operatorname{argmax}([\alpha^{(k,1)} \dots \alpha^{(k,|X|)}])$, the one with the highest attention value, and the decoder output $o^{(k)}_d$ is set to x^l. (For τ_Q decoder, the model considers only the question part of X.) The input $i^{(k+1)}_d$ to the next decoder cell reuses the embedding of the copied token, and is computed as the concatenation $i^{(k+1)}_d = (emb_e(o^{(k)}_d), O_e\alpha^{(k)})$ of the token embedding and the attention vector.

As all different decoder types consume and produce similar values, they could easily be exchanged or extended if more types need to be supported. The advantage of this construction is that only a very small output vocabulary of SQL operators needs to be considered, whereas all other values are obtained through copying.

3 TRAINING

The model is trained from question-SQL program pairs (X, Y), where $Y = [y^{(1)}, \dots, y^{(|Y|)}]$ is a sequence representing the ground truth program for question X. Different typed decoder cells in our model are trained with different loss functions.

$\tau_{\mathcal{V}}$ This is the standard RNN case, where the loss for an output token with this type is computed as cross entropy of the one-hot encoding of the target symbol and the distribution over the decoder vocabulary $\mathcal{V}$, i.e.,

$$loss_{\mathcal{V}}(k) = -\operatorname{onehot}(y^{(k)}) \cdot \log(\operatorname{softmax}(W_{\mathcal{V}}(\alpha^{(k)}_{\mathcal{V}} O_e) + b_{\mathcal{V}})).$$

τ_C, τ_Q In this case, our objective is to copy a correct token from the input into the output. As the original input-output pair does not explicitly contain any pointers, we first need to find an

index $\lambda_k \in [1, \ldots, |X|]$ such that $y^{(k)} = x^{(\lambda_k)}$. In practice, there are often multiple such indices, i.e., the target token appears several times in the input query (e.g., both as a column name supplied from the table information and as part of the user question). We define two loss functions for this case and evaluate both.

- *(Pointer-based loss)* We pick the smallest λ_k with $y^{(k)} = x^{(\lambda_k)}$ and compute the loss as cross entropy between this index and the chosen index, i.e.,

$$loss_C^{\text{pntr}}(k) = -\operatorname{onehot}(\lambda_k) \cdot \log(\operatorname{softmax}(\alpha_C^{(k)}))$$

- *(Value-based loss)* While the first way to calculate the loss trains the network towards generating correct output sequence, it restricts the model to only point to first word occurrence in the input sequence. In contrast, we can allow the decoder to choose any one of the input tokens with the correct value. For that, we define value-based loss functions that transforms the distribution over locations pointed by the pointer header into distributions over input vocabularies that can be pointed by these headers. To transfer the distributions from pointer locations to values, we provide two transferring strategies: max-transfer and sum-transfer.

  * *(Max Transfer)* The max-transfer strategy calculates the probability of copying a token v in input vocabulary as the maximum probability of pointers that point to token v:

$$\phi_{\text{max}}^{(k)}(v) \quad = \max_{0 \le l \le |X|} \{\alpha^{(k,\ell)} \mid x^{(l)} = v\}$$

  * *(Sum Transfer)* The sum-transfer strategy calculates the probability of copying a token v in the input vocabulary as the sum of probability of pointers that point to token v:

$$\phi_{\text{sum}}^{(k)}(v) \quad = \sum_{0 \le l \le |X|} \{\alpha^{(k,\ell)} \mid x^{(l)} = v\}$$

  For both strategies, we calculate the loss function by:

$$loss_C^{\text{val}}(k) \quad = -\operatorname{onehot}(y^{(t)}) \cdot \log([\phi^{(k)}(v) \mid v \in I]).$$

  Also, for models trained with sum-transfer value-based loss function, the outputs of τ_Q, τ_C decoder cells are tokens with highest transfered probabilities, computed by $\operatorname{argmax}_{v \in X}(\phi_{\text{sum}}^{(k)}(v))$ instead of $x^{\operatorname{argmax}(\alpha^{(k)})}$, so that decoding results are consistent with the training goal.

The overall loss for a target output sequence O can then be computed as the sum of the appropriate loss functions for each individual output token $o^{(k)}$.

4 Evaluation

We evaluate our model on WikiSQL dataset (Zhong et al., 2017) by comparing it with prior work and our model with different sub-components to analyze their contributions.

4.1 Experiment Setup

We use the sequence version of the WikiSQL dataset with the default train/dev/test split. Besides question-query pairs, we also use the tables in the dataset to preprocess the dataset.

Preprocessing We first preprocess the dataset by running both tables and question-query pairs through Stanford Stanza (Manning et al., 2014) using the script included in WikiSQL dataset, which normalizes punctuation and cases of the dataset. Second, we normalize the dataset based on their corresponding table: for table entries and columns occurring in questions or queries, we normalize their format to be consistent with the table. This process aims to eliminate inconsistencies caused by different whitespace, e.g. for a column named "country (endonym)" in the table, we normalize its occurrences as "country (endonym)" in the question (or query) to "country (endonym)" so that they are consistent with the entity in table. Note that we restrict our normalization to only whitespace, comma (','), period ('.') and word permutation to avoid over-processing. Especially, we do not edit

tokens: e.g., a phrase "office depot" occurred from a question or a query will not be normalized into "the office depot" even if the latter occurs as a table entry. Similarly, "california district 10th" won't be normalized to "california 10th", and "citv" won't be normalized to "city". We also treat each occurrence of a column name or a table entry in questions as a single word for embedding and copying (instead of copying multiple times for multi-word names/constants).

Dataset After preprocessing, we filter the training set by removing pairs whose ground truth solution contains constants not mentioned in the question, as our model requires the constants to be copied from the question. We train and tune our model only on filtered training and filtered dev set, but we report our evaluation on the full dev and test sets. We obtain 59,845 (originally 61,297) training pairs, 8,928 (originally 9,145) dev pairs and 17,283 test pairs (the test set is not filtered).

Column Annotation We annotate table entry mentions in the question with their corresponding column name iff the table entry mention uniquely belongs to one column of the table. The purpose of this annotation is to bridge special column entries and their column information that cannot be learned elsewhere. For example, if an entity "rocco mediate" in the question uniquely belongs to the "player" column in the table, we annotate the question by concatenating the column name in front of the entity (resulting in "player rocco mediate"). This process resembles entity linking technique used by Krishnamurthy et al. (2017), but in a conservative and deterministic way.

Model Setup We use pre-trained n-gram embedding by Hashimoto et al. (2017) (100 dimensions) and GloVe word embedding (100 dimension) by Pennington et al. (2014) for encoder embedding; each token is embedded into a 200 dimension vector. Both the encoder and decoder are 3-layer dynamic bidirectional LSTM RNN with hidden states sized 100. The model is trained with question-query pairs with a batch size of 200 for 100 epochs. During training, we clip gradient at 10 and add gradient noise with $\eta = 0.3, \gamma = 0.55$ to stabilize training (Neelakantan et al., 2015). The model is implemented in Tensorflow and trained using the Adagrad optimizer (Duchi et al., 2011).

4.2 Overall Result

Table 1 shows the result of our model with the best performance on the dev set, compared against the augmented pointer model and Seq2SQL model (with RL), both by Zhong et al. (2017). We report both the accuracy computed with exact syntax match (Acc_{syn}) and the accuracy based on query execution result (Acc_{ex}). Since syntactically different queries can be equivalent on the table (e.g., queries with different predicate orders compared to the ground truth), the execution accuracy in all cases is higher than the corresponding syntax accuracy.

Our best model achieves 61.0% on the filtered dev set, and it is trained with value-based loss with sum-transfer strategy. Besides, our model syntax accuracy (Test Acc_{syn}) on the test set for problems whose ground truth contain [0, 1, 2, 3, 4] predicates are [54.2%, 65.0%, 50.9%, 37.6%, 23.6%], which indicates that our model retains the ability to correctly generate long queries.

Model	Filtered Dev Acc_{syn}	Dev Acc_{syn}	Dev Acc_{ex}	Test Acc_{syn}	Test Acc_{ex}
Pointer Model	-	44.1%	53.8%	43.3%	53.3%
Seq2SQL	-	49.5%	60.8%	48.3%	59.4%
Our Model	61.0%	**59.6%**	**65.2%**	**59.5%**	**65.1%**

Table 1: Dev and test accuracy of the model, where Acc_{syn} refers to syntax accuracy and Acc_{ex} refers to execution accuracy.

4.3 Ablation Tests

While the overall result shows significant improvement of our model over prior work, we now analyze different sub-components of our model individually to better understand their contribution to the overall performance. We ran four sets of abalation test on our model.

Set A compares our model with and without types, set B compares different loss functions for training pointers (Section 3), set C compares our result with and without column annotation in questions

for table entries, and set D compares different encoder embeddings (pre-trained v.s. trainable). For each experiment set, we ran every model 5 times and reports their results. All model variances are based on the model described in Section 4.1 with same hyper-parameters, and the results are plotted based on models' syntax accuracies on the filtered dev set.

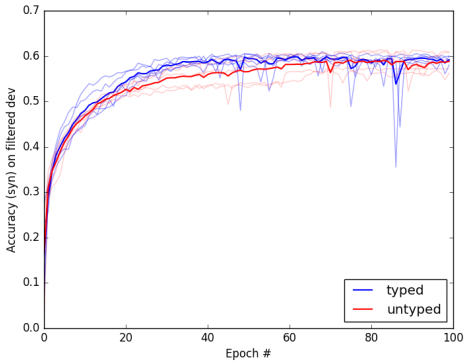
- *Set-A (Type-based decoding)* Figure 3a shows the model performance of with and without typed decoders. For the untyped model, we directly concatenate all SQL operators in the front of table header and set all decoder cells to copy mode. The result shows that while types do not significantly improve model performance (with an average improvement 1.4%), they allow the model to stabilize within fewer epochs. Additionally, we also observed that typed decoders increase the training speed per epoch by approximately $\sim$23%.
- *Set-B (Loss function)* Figure 3b shows dev accuracies for three models trained with different loss functions: value-based loss with sum-transfer strategy (sum-val-loss, blue line), value-based loss with max-transfer strategy (max-val-loss, red line), and pointer-based loss (pntr-loss, green line). The result shows that sum-transfer strategy significantly improves the model consistency and model accuracy compared to other strategies typically used in pointer models. Notably, while value-based loss with max-transfer strategy outperforms the pointer-based loss in its best runs (with an accuracy of 56.4%), its behavior among multiple runs is very unstable and significantly dependent on the initialization. Our investigation shows that the sum-transfer strategy benefits from the good initialization resulting from the fact that multiple mentions of column names in the question increases the probability of choosing the column in initial epochs. Furthermore, the result show that overly constraining the model by only allowing the model to only choose columns from the header and not from their mentions in questions (as in pointer loss) can be negative to the model performance.
- *Set-C (Column Annotation)* Figure 3c shows the result with and without column annotation used in preprocessing. In average, the model accuracy drops by 7.5% if trained and tested on questions without column annotation. The result suggests that deterministically linking entities with their column can benefit the model and incorporating entity linking can be important. On the other hand, the model can still stably reach $\sim$52.5% accuracy on unannotated questions, suggesting that the stabilization is coming from typed decoding and sum-loss function can still benefit under-processed data.
- *Set-D (Embedding Method)* Figure 3d shows the result for three models that are trained with different input embeddings: untrainable n-gram + GloVe embedding (untrainable in the plot), trainable embedding with n-gram + GloVe initialization (fixed-init) and trainable embedding with random initialization (random-init). Our results show that incorporating prior knowledge through untrainable embedding can effectively prevent over-fitting.

4.4 Error Analysis & Limitations

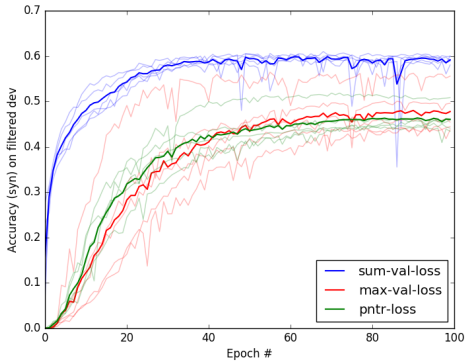
We classify errors made by our model described in Section 4.1 using the parts of a query (aggregation function, select column, or predicates) that was incorrectly predicted by the model. Among the 6,024 incorrectly predicted cases, 32.0% cases use wrong aggregation function, 47.1% cases copied the wrong column name, and 51.1% cases contain mistakes in predicates (27.6% cases made multiple mistakes). Notably, most cases with wrong predicates contain wrong column predictions. Such cases are typically caused by (1) the correct column name is not mentioned in the question (e.g., a column is named 'rank' was mentioned through 'best' in the question) or (2) there exist multiple columns with similar names (e.g., 'team 1', 'team 2'). These errors suggest that the model lacks the understanding of the knowledge presented in the table, and embedding the table content together with the question (Krishnamurthy et al., 2017; Yih et al., 2015) can potentially improve the model.

Also, the fact that our model does not support multiple pointer headers or external vocabulary for decoding constant results in 13.1% wrong predictions (e.g., our model cannot select 'intisar field' from 'field of intisar' in the question or generate 'score 4–4' from question 'which team wins by 4–4?'), which suggests that extending the model with multiple constant pointers per slot or introducing an extra decoding layer for constant rewriting can potentially improve the model.

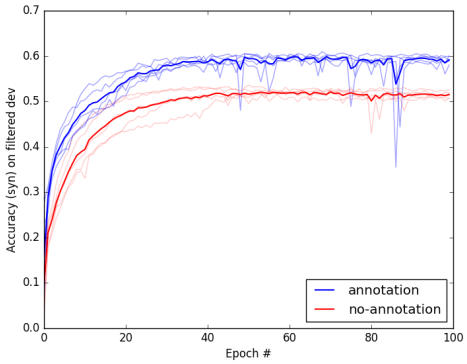
Finally, we do not directly train our model to learn syntactically different but semantically equivalent program and 62.2% among all wrong queries yield run-time error or return `None` during execution.



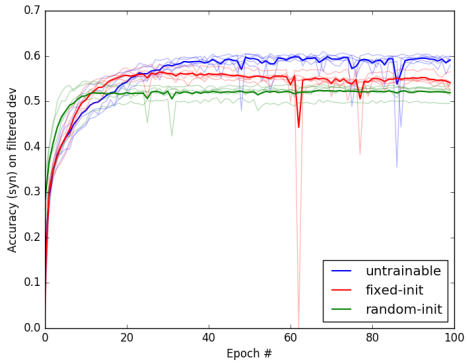
(a) Typed and untyped decoder.



(b) Different loss functions.



(c) With and without column annotation.



(d) Different encoder embedding.

Figure 3: Ablation test results showing the syntax accuracy (on filtered dev set) for each setting. For each setting, transparent lines show actual accuries for all 5 runs, and the none-transparent line highlights its average accuracy.

It suggests that our model can potentially be trained with a reinforcement loop to explicitly punish ill-formed queries and reward semantically equivalent ones (Zhong et al., 2017).

5 Related Work

Semantic Parsing Nearest to our work, mapping natural language to logic forms has been extensively studied in natural language processing research (Zettlemoyer & Collins, 2012; Artzi & Zettlemoyer, 2011; Berant et al., 2013; Wang et al., 2015; Iyer et al., 2017; Iyyer et al., 2017). Dong & Lapata (2016); Alvarez-Melis & Jaakkola (2016); Krishnamurthy et al. (2017); Yin & Neubig (2017); Rabinovich et al. (2017) are closely related neural semantic parsers adopting tree-based decoding that also utilize grammar production rules as decoding constraints. However, our model foregoes the complexity of generating a full parse tree and never produces non-terminal nodes, and instead retains the simplicity of a sequence decoder. This makes it substantially easier to implement and train, as the sequence model requires no explicit controller for production rule selection. To our knowledge, our model is also the first to use target token type information to specialize the decoder to a mode in which it copies from a type-compatible, restricted set of input tokens.

Pointer Networks Pointer networks and copy networks enhance RNNs with the ability to reuse input tokens, and they have been successfully used in interactive conversation (Gu et al., 2016), geometric problems (Vinyals et al., 2015) and program generation (Zhong et al., 2017). Our model differs from previous approaches as we use program types to explicitly guide pointer region; furthermore, we provide new training strategies for training pointer models when pointer aliases exist.

Program Induction / Synthesis Program induction (Reed & De Freitas, 2015; Neelakantan et al., 2016; Graves et al., 2014; Yin et al., 2015) aims to induce latent programs for question answering; on the other hand, program synthesis models (Zhong et al., 2017; Parisotto et al., 2016) aim to generate explicit programs and execute the program to obtain answer. Our model follows the line of neural program synthesis model and trains directly with question program pairs.

Orthogonal Approaches Entity linking (Calixto et al., 2017; Yih et al., 2015; Krishnamurthy et al., 2017) is a technique used to link knowledge between the encoding sequence and knowledge base (e.g., table, document) in semantic parsing that is orthogonal to the neural encoder decoder model. This technique can potentially be used to address our limitation in our deterministic column annotation process. Besides, reinforcement learning (Zhong et al., 2017) allows the model to freely learn semantically equivalent solutions to user questions, and can be composed with our model to improve the model accuracy.

6 Conclusion

We presented a new sequence to sequence based neural architecture to translate natural language questions over tables into executable SQL queries. Our approach uses a simple type system to guide the decoder to either copy a token from the input using a pointer-based copying mechanism or generate a token from a finite vocabulary. We presented a sum-transfer value based loss function that transforms the distribution over pointer locations into distribution over tokens in input vocabularies to efficiently train the architecture. Our evaluation on the WikiSQL dataset showed that our model significantly outperforms several advanced baseline architectures and the current state-of-the-art Seq2SQL model.

References

David Alvarez-Melis and Tommi S Jaakkola. Tree-structured decoding with doubly-recurrent neural networks. 2016.

Yoav Artzi and Luke Zettlemoyer. Bootstrapping semantic parsers from conversations. In *Proceedings of the conference on empirical methods in natural language processing*, pp. 421–432. Association for Computational Linguistics, 2011.

J. Berant, A. Chou, R. Frostig, and P. Liang. Semantic parsing on Freebase from question-answer pairs. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2013.

Iacer Calixto, Qun Liu, and Nick Campbell. Doubly-attentive decoder for multi-modal neural machine translation. *arXiv preprint arXiv:1702.01287*, 2017.

Li Dong and Mirella Lapata. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*, 2016. URL `http://aclweb.org/anthology/P/P16/P16-1004.pdf`.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

J. Gu, Z. Lu, H. Li, and V. O. K. Li. Incorporating Copying Mechanism in Sequence-to-Sequence Learning. *ArXiv e-prints*, March 2016.

Mike Gualtieri. Deputize end-user developers to deliver business agility and reduce costs. *Forrester Report for Application Development and Program Management Professionals*, 2009.

Sumit Gulwani and Mark Marron. Nlyze: interactive programming by natural language for spreadsheet data analysis and manipulation. In *SIGMOD*, pp. 803–814, 2014.

Kazuma Hashimoto, Caiming Xiong, Yoshimasa Tsuruoka, and Richard Socher. A joint many-task model: Growing a neural network for multiple NLP tasks. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*, pp. 446–456, 2017. URL http://aclanthology.info/papers/D17-1046/d17-1046.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. Learning a neural semantic parser from user feedback. *arXiv preprint arXiv:1704.08760*, 2017.

Mohit Iyyer, Wen tau Yih, and Ming-Wei Chang. Search-based neural structured learning for sequential question answering. In *Association for Computational Linguistics*, 2017.

Robin Jia and Percy Liang. Data recombination for neural semantic parsing. In *ACL*, 2016.

Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*, pp. 1517–1527, 2017. URL http://aclanthology.info/papers/D17-1160/d17-1160.

Yunyao Li, Huahai Yang, and H. V. Jagadish. Nalix: An interactive natural language interface for querying xml. In *SIGMOD*, pp. 900–902, 2005. ISBN 1-59593-060-4.

Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pp. 55–60, 2014. URL http://www.aclweb.org/anthology/P/P14/P14-5010.

Arvind Neelakantan, Luke Vilnis, Quoc V. Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding gradient noise improves learning for very deep networks. *CoRR*, abs/1511.06807, 2015. URL http://arxiv.org/abs/1511.06807.

Arvind Neelakantan, Quoc V Le, Martin Abadi, Andrew McCallum, and Dario Amodei. Learning a natural language interface with neural programmer. *arXiv preprint arXiv:1611.08945*, 2016.

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *CoRR*, abs/1611.01855, 2016. URL http://arxiv.org/abs/1611.01855.

Panupong Pasupat and Percy Liang. Compositional semantic parsing on semi-structured tables. In *ACL*, pp. 1470–1480, 2015.

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

Hoifung Poon. Grounded unsupervised semantic parsing. In *ACL*, pp. 933–943, 2013.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. *CoRR*, abs/1704.07535, 2017. URL http://arxiv.org/abs/1704.07535.

Scott Reed and Nando De Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

O. Vinyals, M. Fortunato, and N. Jaitly. Pointer Networks. *ArXiv e-prints*, June 2015.

Y. Wang, J. Berant, and P. Liang. Building a semantic parser overnight. In *Association for Computational Linguistics (ACL)*, 2015.

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

Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. *CoRR*, abs/1704.01696, 2017. URL http://arxiv.org/abs/1704.01696.

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

Luke S. Zettlemoyer and Michael Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *UAI*, pp. 658–666, 2005.

Luke S Zettlemoyer and Michael Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. *arXiv preprint arXiv:1207.1420*, 2012.

V. Zhong, C. Xiong, and R. Socher. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *ArXiv e-prints*, August 2017.

A APPENDIX

A.1 EXAMPLE OF GENERATED QUERIES

We show a few examples that our model correctly and wrongly predicted collected from the evaluation result from the test dataset.

Correct example (1)

- Table: 1-1014206-2 [#, shipyard, laid down, launched, commissioned, fleet, status]
- Question: *"List the # for ships commissioned on september 30, 1967"*
- Solution: `Select` # `From` 1-1014206-2 `Where` commissioned = september 30, 1967
- Prediction: `Select` # `From` 1-1014206-2 `Where` commissioned = september 30, 1967

Correct example (2)

- Table: 2-17829169-2 [rank, gold, silver, bronze, total]
- Question: *"what is the most possible bronze medals when rank is more than 11 and there are fewer than 0 gold medals ?"*
- Solution: `Max` bronze `From` 2-17829169-2 `Where` rank > 11 `And` gold < 0
- Prediction: `Max` bronze `From` 2-17829169-2 `Where` rank > 11 `And` gold < 0

Correct example (3)

- Table: 2-18421908-1 [year, competition, venue, position, event]
- Question: *"what is the position for event discus in year 2013?"*
- Solution: `Select` position `From` 2-18421908-1 `Where` event = discus `And` year = 2013
- Prediction: `Select` position `From` 2-18421908-1 `Where` year = 2013 `And` event = discus

(Remarks: the colum name 'event' before 'event discus' is annotated to the question during column annotation since 'discus' uniquely appear in the event column.)

Example with wrong target column

- Table: 1-11206916-1 [team, stadium, capacity, highest, lowest, average]
- Question: *"what was the lowest highest attendance for the dumbarton team ?"*
- Solution: `Min` highest `From` 1-11206916-1 `Where` team = dumbarton
- Prediction: `Min` lowest `From` 1-11206916-1 `Where` team = dumbarton

(Remarks: The model fails recognize the target should be highest instead of 'lowest' that is used to specify the aggregation function `Min`.)

Example with wrong aggregation function

- Table: 2-18569335-2 [rank, heat, name, nationality, time]

- Question: *"how many heat did runners from nationality guinea-bissau run , with rank higher than 33 ?"*
- Solution: `Sum` heat `From` 2-18569335-2 `Where` nationality = guinea-bissau `And` rank < 33
- Prediction: `Count` heat `From` 2-18569335-2 `Where` nationality = guinea-bissau `And` rank > 33

(Remarks: The model fails to recognize that the 'heat' column stores aggregated value which should be count using `Sum`. Besides, the model does not recognize 'higher rank' indicates smaller numerical value.)

Example with wrong predicate (1)

- Table: 2-18661293-4 [rank, nation, gold, silver, bronze, total]
- Question: *"what 's the gold medal count for total nation with a bronze count more than 0 and a total less than 54 ?"*
- Solution: `Max` gold `From` 2-18661293-4 `Where` bronze > 0 `And` nation = total `And` total < 54
- Prediction: `Sum` gold `From` 2-18661293-4 `Where` bronze < 0 `And` total < 54

(Remarks: The model fails to learn the existence of the 'total' row in the table and miss the choice for nation.)

Example with wrong predicate (2)

- Table: 2-1223181-1 [year, entrant, chassis, engine, points]
- Question: *"what is the most points for a vehicle with a lola thl1, chassis later than year 1986?"*
- Solution: `Max` points `From` 2-1223181-1 `Where` chassis = lola thl1 `And` year > 1986
- Prediction: `Max` points `From` 2-1223181-1 `Where` year = lola thl1 `And` chassis = lola thl1 `And` year > 1986

(Remarks: The model fails by predicting a wrong column and constant combination.)