Coarse-to-Fine Decoding for Neural Semantic Parsing

Li Dong and **Mirella Lapata**
Institute for Language, Cognition and Computation
School of Informatics, University of Edinburgh
10 Crichton Street, Edinburgh EH8 9AB
li.dong@ed.ac.uk mlap@inf.ed.ac.uk

Abstract

Semantic parsing aims at mapping natural language utterances into structured meaning representations. In this work, we propose a structure-aware neural architecture which decomposes the semantic parsing process into two stages. Given an input utterance, we first generate a rough sketch of its meaning, where low-level information (such as variable names and arguments) is glossed over. Then, we fill in missing details by taking into account the natural language input and the sketch itself. Experimental results on four datasets characteristic of different domains and meaning representations show that our approach consistently improves performance, achieving competitive results despite the use of relatively simple decoders.

1 Introduction

Semantic parsing maps natural language utterances onto machine interpretable meaning representations (e.g., executable queries or logical forms). The successful application of recurrent neural networks to a variety of NLP tasks (Bahdanau et al., 2015; Vinyals et al., 2015) has provided strong impetus to treat semantic parsing as a sequence-to-sequence problem (Jia and Liang, 2016; Dong and Lapata, 2016; Ling et al., 2016). The fact that meaning representations are typically structured objects has prompted efforts to develop neural architectures which explicitly account for their structure. Examples include tree decoders (Dong and Lapata, 2016; Alvarez-Melis and Jaakkola, 2017), decoders constrained by a grammar model (Xiao et al., 2016; Yin and Neubig, 2017; Krishnamurthy et al., 2017), or modular decoders which use syntax to dynamically compose various submodels (Rabinovich et al., 2017).

In this work, we propose to decompose the decoding process into two stages. The first decoder focuses on predicting a rough *sketch* of the meaning representation, which omits low-level details, such as arguments and variable names. Example sketches for various meaning representations are shown in Table 1. Then, a second decoder fills in missing details by conditioning on the natural language input and the sketch itself. Specifically, the sketch constrains the generation process and is encoded into vectors to guide decoding.

We argue that there are at least three advantages to the proposed approach. Firstly, the decomposition disentangles high-level from low-level semantic information, which enables the decoders to model meaning at different levels of granularity. As shown in Table 1, sketches are more compact and as a result easier to generate compared to decoding the entire meaning structure in one go. Secondly, the model can explicitly share knowledge of coarse structures for the examples that have the same sketch (i.e., basic meaning), even though their actual meaning representations are different (e.g., due to different details). Thirdly, after generating the sketch, the decoder knows what the basic meaning of the utterance looks like, and the model can use it as global context to improve the prediction of the final details.

Our framework is flexible and not restricted to specific tasks or any particular model. We conduct experiments on four datasets representative of various semantic parsing tasks ranging from logical form parsing, to code generation, and SQL query generation. We adapt our architecture to these tasks and present several ways to obtain sketches from their respective meaning representations. Experimental results show that our framework achieves competitive performance compared

Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Long Papers), pages 731–742
Melbourne, Australia, July 15 - 20, 2018.

Dataset	Length	Example
GEO	7.6	x : *which state has the most rivers running through it?*
	13.7	y : (argmax $0 (state:t $0) (count $1 (and (river:t $1) (loc:t $1 $0))))
	6.9	a : (argmax#1 state:t@1 (count#1 (and river:t@1 loc:t@2)))
ATIS	11.1	x : *all flights from dallas before 10am*
	21.1	y : (lambda $0 e (and (flight $0) (from $0 dallas:ci) (< (departure_time $0) 1000:ti)))
	9.2	a : (lambda#2 (and flight@1 from@2 (< departure_time@1 ?)))
DJANGO	14.4	x : *if length of bits is lesser than integer 3 or second element of bits is not equal to string 'as' ,*
	8.7	y : if len(bits) < 3 or bits[1] != 'as':
	8.0	a : if len (NAME) < NUMBER or NAME [NUMBER] != STRING :
WIKISQL	17.9	Table schema: ‖*Pianist*‖*Conductor*‖*Record Company*‖*Year of Recording*‖*Format*‖
	13.3	x : *What record company did conductor Mikhail Snitko record for after 1996?*
	13.0	y : SELECT *Record Company* WHERE (*Year of Recording* > *1996*) AND (*Conductor* = *Mikhail Snitko*)
	2.7	a : WHERE > AND =

Table 1: Examples of natural language expressions x, their meaning representations y, and meaning sketches a. The average number of tokens is shown in the second column.

with previous systems, despite employing relatively simple sequence decoders.

2 Related Work

Various models have been proposed over the years to learn semantic parsers from natural language expressions paired with their meaning representations (Tang and Mooney, 2000; Ge and Mooney, 2005; Zettlemoyer and Collins, 2007; Wong and Mooney, 2007; Lu et al., 2008; Kwiatkowski et al., 2011; Andreas et al., 2013; Zhao and Huang, 2015). These systems typically learn lexicalized mapping rules and scoring models to construct a meaning representation for a given input.

More recently, neural sequence-to-sequence models have been applied to semantic parsing with promising results (Dong and Lapata, 2016; Jia and Liang, 2016; Ling et al., 2016), eschewing the need for extensive feature engineering. Several ideas have been explored to enhance the performance of these models such as data augmentation (Kočiský et al., 2016; Jia and Liang, 2016), transfer learning (Fan et al., 2017), sharing parameters for multiple languages or meaning representations (Susanto and Lu, 2017; Herzig and Berant, 2017), and utilizing user feedback signals (Iyer et al., 2017). There are also efforts to develop structured decoders that make use of the syntax of meaning representations. Dong and Lapata (2016) and Alvarez-Melis and Jaakkola (2017) develop models which generate tree structures in a top-down fashion. Xiao et al. (2016) and Krishnamurthy et al. (2017) employ the grammar to constrain the decoding process. Cheng et al. (2017) use a transition system to generate variable-free queries. Yin and Neubig (2017) design a grammar model for the generation of abstract syntax trees (Aho et al., 2007) in depth-first, left-to-right order. Rabinovich et al. (2017) propose a modular decoder whose submodels are dynamically composed according to the generated tree structure.

Our own work also aims to model the structure of meaning representations more faithfully. The flexibility of our approach enables us to easily apply sketches to different types of meaning representations, e.g., trees or other structured objects. Coarse-to-fine methods have been popular in the NLP literature, and are perhaps best known for syntactic parsing (Charniak et al., 2006; Petrov, 2011). Artzi and Zettlemoyer (2013) and Zhang et al. (2017) use coarse lexical entries or macro grammars to reduce the search space of semantic parsers. Compared with coarse-to-fine inference for lexical induction, sketches in our case are abstractions of the final meaning representation.

The idea of using sketches as intermediate representations has also been explored in the field of program synthesis (Solar-Lezama, 2008; Zhang and Sun, 2013; Feng et al., 2017). Yaghmazadeh et al. (2017) use SEMPRE (Berant et al., 2013) to map a sentence into SQL sketches which are completed using program synthesis techniques and iteratively repaired if they are faulty.

3 Problem Formulation

Our goal is to learn semantic parsers from instances of natural language expressions paired with their structured meaning representations.

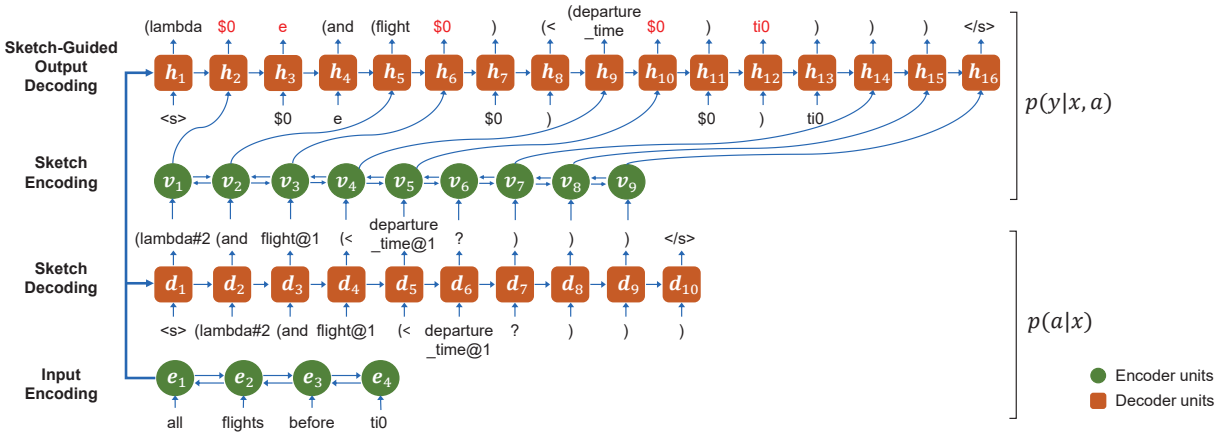

Figure 1: We first generate the meaning sketch a for natural language input x. Then, a fine meaning decoder fills in the missing details (shown in red) of meaning representation y. The coarse structure a is used to guide and constrain the output decoding.

Let $x = x_1 \cdots x_{|x|}$ denote a natural language expression, and $y = y_1 \cdots y_{|y|}$ its meaning representation. We wish to estimate $p(y|x)$, the conditional probability of meaning representation y given input x. We decompose $p(y|x)$ into a two-stage generation process:

$$p(y|x) = p(y|x, a)\, p(a|x) \tag{1}$$

where $a = a_1 \cdots a_{|a|}$ is an abstract sketch representing the meaning of y. We defer detailed description of how sketches are extracted to Section 4. Suffice it to say that the extraction amounts to stripping off arguments and variable names in logical forms, schema specific information in SQL queries, and substituting tokens with types in source code (see Table 1).

As shown in Figure 1, we first predict sketch a for input x, and then fill in missing details to generate the final meaning representation y by conditioning on both x and a. The sketch is encoded into vectors which in turn guide and constrain the decoding of y. We view the input expression x, the meaning representation y, and its sketch a as sequences. The generation probabilities are factorized as:

$$p(a|x) = \prod_{t=1}^{|a|} p(a_t|a_{<t}, x) \tag{2}$$

$$p(y|x, a) = \prod_{t=1}^{|y|} p(y_t|y_{<t}, x, a) \tag{3}$$

where $a_{<t} = a_1 \cdots a_{t-1}$, and $y_{<t} = y_1 \cdots y_{t-1}$. In the following, we will explain how $p(a|x)$ and $p(y|x, a)$ are estimated.

3.1 Sketch Generation

An *encoder* is used to encode the natural language input x into vector representations. Then, a *decoder* learns to compute $p(a|x)$ and generate the sketch a conditioned on the encoding vectors.

Input Encoder Every input word is mapped to a vector via $\mathbf{x}_t = \mathbf{W}_x \mathbf{o}(x_t)$, where $\mathbf{W}_x \in \mathbb{R}^{n \times |\mathcal{V}_x|}$ is an embedding matrix, $|\mathcal{V}_x|$ is the vocabulary size, and $\mathbf{o}(x_t)$ a one-hot vector. We use a bi-directional recurrent neural network with long short-term memory units (LSTM, Hochreiter and Schmidhuber 1997) as the input encoder. The encoder recursively computes the hidden vectors at the t-th time step via:

$$\overrightarrow{\mathbf{e}}_t = \mathrm{f}_{\mathrm{LSTM}}\left(\overrightarrow{\mathbf{e}}_{t-1}, \mathbf{x}_t\right), t = 1, \cdots, |x| \tag{4}$$

$$\overleftarrow{\mathbf{e}}_t = \mathrm{f}_{\mathrm{LSTM}}\left(\overleftarrow{\mathbf{e}}_{t+1}, \mathbf{x}_t\right), t = |x|, \cdots, 1 \tag{5}$$

$$\mathbf{e}_t = [\overrightarrow{\mathbf{e}}_t, \overleftarrow{\mathbf{e}}_t] \tag{6}$$

where $[\cdot, \cdot]$ denotes vector concatenation, $\mathbf{e}_t \in \mathbb{R}^n$, and $\mathrm{f}_{\mathrm{LSTM}}$ is the LSTM function.

Coarse Meaning Decoder The decoder's hidden vector at the t-th time step is computed by $\mathbf{d}_t = \mathrm{f}_{\mathrm{LSTM}}(\mathbf{d}_{t-1}, \mathbf{a}_{t-1})$, where $\mathbf{a}_{t-1} \in \mathbb{R}^n$ is the embedding of the previously predicted token. The hidden states of the first time step in the decoder are initialized by the concatenated encoding vectors $\mathbf{d}_0 = [\overrightarrow{\mathbf{e}}_{|x|}, \overleftarrow{\mathbf{e}}_1]$. Additionally, we use an attention mechanism (Luong et al., 2015) to learn soft alignments. We compute the attention score for the current time step t of the decoder, with the k-th hidden state in the encoder as:

$$s_{t,k} = \exp\{\mathbf{d}_t \cdot \mathbf{e}_k\} / Z_t \tag{7}$$

where $Z_t = \sum_{j=1}^{|x|} \exp\{\mathbf{d}_t \cdot \mathbf{e}_j\}$ is a normalization term. Then we compute $p(a_t|a_{<t}, x)$ via:

$$\mathbf{e}_t^d = \sum_{k=1}^{|x|} s_{t,k}\mathbf{e}_k \quad (8)$$

$$\mathbf{d}_t^{att} = \tanh\left(\mathbf{W}_1\mathbf{d}_t + \mathbf{W}_2\mathbf{e}_t^d\right) \quad (9)$$

$$p(a_t|a_{<t}, x) = \text{softmax}_{a_t}\left(\mathbf{W}_o\mathbf{d}_t^{att} + \mathbf{b}_o\right) \quad (10)$$

where $\mathbf{W}_1, \mathbf{W}_2 \in \mathbb{R}^{n\times n}$, $\mathbf{W}_o \in \mathbb{R}^{|\mathcal{V}_a|\times n}$, and $\mathbf{b}_o \in \mathbb{R}^{|\mathcal{V}_a|}$ are parameters. Generation terminates once an end-of-sequence token "*</s>*" is emitted.

3.2 Meaning Representation Generation

Meaning representations are predicted by conditioning on the input x and the generated sketch a. The model uses the encoder-decoder architecture to compute $p(y|x, a)$, and decorates the sketch a with details to generate the final output.

Sketch Encoder As shown in Figure 1, a bidirectional LSTM encoder maps the sketch sequence a into vectors $\{\mathbf{v}_k\}_{k=1}^{|a|}$ as in Equation (6), where $\mathbf{v}_k$ denotes the vector of the k-th time step.

Fine Meaning Decoder The final decoder is based on recurrent neural networks with an attention mechanism, and shares the input encoder described in Section 3.1. The decoder's hidden states $\{\mathbf{h}_t\}_{t=1}^{|y|}$ are computed via:

$$\mathbf{i}_t = \begin{cases} \mathbf{v}_k & y_{t-1} \text{ is determined by } a_k \\ \mathbf{y}_{t-1} & \text{otherwise} \end{cases} \quad (11)$$

$$\mathbf{h}_t = \text{f}_{\text{LSTM}}(\mathbf{h}_{t-1}, \mathbf{i}_t)$$

where $\mathbf{h}_0 = [\overrightarrow{\mathbf{e}}_{|x|}, \overleftarrow{\mathbf{e}}_1]$, and $\mathbf{y}_{t-1}$ is the embedding of the previously predicted token. Apart from using the embeddings of previous tokens, the decoder is also fed with $\{\mathbf{v}_k\}_{k=1}^{|a|}$. If y_{t-1} is determined by a_k in the sketch (i.e., there is a one-to-one alignment between y_{t-1} and a_k), we use the corresponding token's vector $\mathbf{v}_k$ as input to the next time step.

The sketch constrains the decoding output. If the output token y_t is already in the sketch, we force y_t to conform to the sketch. In some cases, sketch tokens will indicate what information is missing (e.g., in Figure 1, token "*flight@1*" indicates that an argument is missing for the predicate "*flight*"). In other cases, sketch tokens will not reveal the number of missing tokens (e.g., "STRING" in DJANGO) but the decoder's output will indicate whether missing details have been generated (e.g., if the decoder emits a closing quote token for "STRING"). Moreover, type information in sketches can be used to constrain generation. In Table 1, sketch token "NUMBER" specifies that a numeric token should be emitted.

For the missing details, we use the hidden vector $\mathbf{h}_t$ to compute $p(y_t|y_{<t}, x, a)$, analogously to Equations (7)–(10).

3.3 Training and Inference

The model's training objective is to maximize the log likelihood of the generated meaning representations given natural language expressions:

$$\max \sum_{(x,a,y)\in\mathcal{D}} \log p(y|x, a) + \log p(a|x)$$

where $\mathcal{D}$ represents training pairs.

At test time, the prediction for input x is obtained via $\hat{a} = \arg\max_{a'} p(a'|x)$ and $\hat{y} = \arg\max_{y'} p(y'|x, \hat{a})$, where a' and y' represent coarse- and fine-grained meaning candidates. Because probabilities $p(a|x)$ and $p(y|x, a)$ are factorized as shown in Equations (2)–(3), we can obtain best results approximately by using greedy search to generate tokens one by one, rather than iterating over all candidates.

4 Semantic Parsing Tasks

In order to show that our framework applies across domains and meaning representations, we developed models for three tasks, namely parsing natural language to logical form, to Python source code, and to SQL query. For each of these tasks we describe the datasets we used, how sketches were extracted, and specify model details over and above the architecture presented in Section 3.

4.1 Natural Language to Logical Form

For our first task we used two benchmark datasets, namely GEO (880 language queries to a database of U.S. geography) and ATIS (5,410 queries to a flight booking system). Examples are shown in Table 1 (see the first and second block). We used standard splits for both datasets: 600 training and 280 test instances for GEO (Zettlemoyer and Collins, 2005); 4,480 training, 480 development, and 450 test examples for ATIS. Meaning representations in these datasets are based on λ-calculus (Kwiatkowski et al., 2011). We use brackets to linearize the hierarchical structure.

Algorithm 1 Sketch for GEO and ATIS

Input: t: Tree-structure λ-calculus expression
$t.pred$: Predicate name, or operator name
Output: a: Meaning sketch
▷ *(count $0 (< (fare $0) 50:do))→(count#1 (< fare@1 ?))*
function SKETCH(t)
  if t is leaf **then** ▷ *No nonterminal in arguments*
    return "%s@%d" % ($t.pred$, len($t.args$))
  if $t.pred$ is λ operator, or quantifier **then** ▷ *e.g., count*
    Omit variable information defined by $t.pred$
    $t.pred \leftarrow$ "%s#%d" % ($t.pred$, len($variable$))
  for $c \leftarrow$ argument in $t.args$ **do**
    if c is nonterminal **then**
      c $\leftarrow$ SKETCH(c)
    else
      c $\leftarrow$ "?" ▷ *Placeholder for terminal*
  return t

The first element between a pair of brackets is an operator or predicate name, and any remaining elements are its arguments.

Algorithm 1 shows the pseudocode used to extract sketches from λ-calculus-based meaning representations. We strip off arguments and variable names in logical forms, while keeping predicates, operators, and composition information. We use the symbol "@" to denote the number of missing arguments in a predicate. For example, we extract "*from@2*" from the expression "*(from $0 dallas:ci)*" which indicates that the predicate "*from*" has two arguments. We use "*?*" as a placeholder in cases where only partial argument information can be omitted. We also omit variable information defined by the lambda operator and quantifiers (e.g., *exists*, *count*, and *argmax*). We use the symbol "#" to denote the number of omitted tokens. For the example in Figure 1, "*lambda $0 e*" is reduced to "*lambda#2*".

The meaning representations of these two datasets are highly compositional, which motivates us to utilize the hierarchical structure of λ-calculus. A similar idea is also explored in the tree decoders proposed in Dong and Lapata (2016) and Yin and Neubig (2017) where parent hidden states are fed to the input gate of the LSTM units. On the contrary, parent hidden states serve as input to the softmax classifiers of both fine and coarse meaning decoders.

Parent Feeding Taking the meaning sketch "*(and flight@1 from@2)*" as an example, the parent of "*from@2*" is "*(and*". Let p_t denote the parent of the t-th time step in the decoder. Compared with Equation (10), we use the vector $\mathbf{d}_t^{att}$ and the hidden state of its parent $\mathbf{d}_{p_t}$ to compute the probability $p(a_t|a_{<t}, x)$ via:

$$p(a_t|a_{<t}, x) = \text{softmax}_{a_t}\left(\mathbf{W}_o[\mathbf{d}_t^{att}, \mathbf{d}_{p_t}] + \mathbf{b}_o\right)$$

where $[\cdot, \cdot]$ denotes vector concatenation. The parent feeding is used for both decoding stages.

4.2 Natural Language to Source Code

Our second semantic parsing task used DJANGO (Oda et al., 2015), a dataset built upon the Python code of the Django library. The dataset contains lines of code paired with natural language expressions (see the third block in Table 1) and exhibits a variety of use cases, such as iteration, exception handling, and string manipulation. The original split has 16,000 training, 1,000 development, and 1,805 test instances.

We used the built-in lexical scanner of Python[1] to tokenize the code and obtain token types. Sketches were extracted by substituting the original tokens with their token types, except delimiters (e.g., "[", and ":"), operators (e.g., "+", and "*"), and built-in keywords (e.g., "True", and "while"). For instance, the expression "if s[:4].lower() == 'http':" becomes "if NAME [: NUMBER] . NAME ( ) == STRING :", with details about names, values, and strings being omitted.

DJANGO is a diverse dataset, spanning various real-world use cases and as a result models are often faced with out-of-vocabulary (OOV) tokens (e.g., variable names, and numbers) that are unseen during training. We handle OOV tokens with a copying mechanism (Gu et al., 2016; Gulcehre et al., 2016; Jia and Liang, 2016), which allows the fine meaning decoder (Section 3.2) to directly copy tokens from the natural language input.

Copying Mechanism Recall that we use a softmax classifier to predict the probability distribution $p(y_t|y_{<t}, x, a)$ over the pre-defined vocabulary. We also learn a copying gate $g_t \in [0, 1]$ to decide whether y_t should be copied from the input or generated from the vocabulary. We compute the modified output distribution via:

$$g_t = \text{sigmoid}(\mathbf{w}_g \cdot \mathbf{h}_t + b_g)$$
$$\tilde{p}(y_t|y_{<t}, x, a) = (1 - g_t)p(y_t|y_{<t}, x, a) + \mathbb{1}_{[y_t \notin \mathcal{V}_y]} g_t \sum_{k:x_k=y_t} s_{t,k}$$

[1] https://docs.python.org/3/library/tokenize

where $\mathbf{w}_g \in \mathbb{R}^n$ and $b_g \in \mathbb{R}$ are parameters, and the indicator function $\mathbb{1}_{[y_t \notin \mathcal{V}_y]}$ is 1 only if y_t is not in the target vocabulary $\mathcal{V}_y$; the attention score $s_{t,k}$ (see Equation (7)) measures how likely it is to copy y_t from the input word x_k.

4.3 Natural Language to SQL

The WIKISQL (Zhong et al., 2017) dataset contains 80,654 examples of questions and SQL queries distributed across 24,241 tables from Wikipedia. The goal is to generate the correct SQL query for a natural language question and table schema (i.e., table column names), without using the content values of tables (see the last block in Table 1 for an example). The dataset is partitioned into a training set (70%), a development set (10%), and a test set (20%). Each table is present in one split to ensure generalization to unseen tables.

WIKISQL queries follow the format "`SELECT agg_op agg_col WHERE (cond_col cond_op cond) AND ...`", which is a subset of the SQL syntax. `SELECT` identifies the column that is to be included in the results after applying the aggregation operator `agg_op`[2] to column `agg_col`. `WHERE` can have zero or multiple conditions, which means that column `cond_col` must satisfy the constraints expressed by the operator `cond_op`[3] and the condition value `cond`. Sketches for SQL queries are simply the (sorted) sequences of condition operators `cond_op` in `WHERE` clauses. For example, in Table 1, sketch "`WHERE > AND =`" has two condition operators, namely ">" and "=".

The generation of SQL queries differs from our previous semantic parsing tasks, in that the table schema serves as input in addition to natural language. We therefore modify our input encoder in order to render it table-aware, so to speak. Furthermore, due to the formulaic nature of the SQL query, we only use our decoder to generate the `WHERE` clause (with the help of sketches). The `SELECT` clause has a fixed number of slots (i.e., aggregation operator `agg_op` and column `agg_col`), which we straightforwardly predict with softmax classifiers (conditioned on the input). We briefly explain how these components are modeled below.

Table-Aware Input Encoder Given a table schema with M columns, we employ the special token "$\|$" to concatenate its header names as "$\|c_{1,1} \cdots c_{1,|c_1|}\| \cdots \|c_{M,1} \cdots c_{M,|c_M|}\|$", where the k-th column ("$c_{k,1} \cdots c_{k,|c_k|}$") has $|c_k|$ words. As shown in Figure 2, we use bi-directional LSTMs to encode the whole sequence. Next, for column c_k, the LSTM hidden states at positions $c_{k,1}$ and $c_{k,|c_k|}$ are concatenated. Finally, the concatenated vectors are used as the encoding vectors $\{\mathbf{c}_k\}_{k=1}^M$ for table columns.

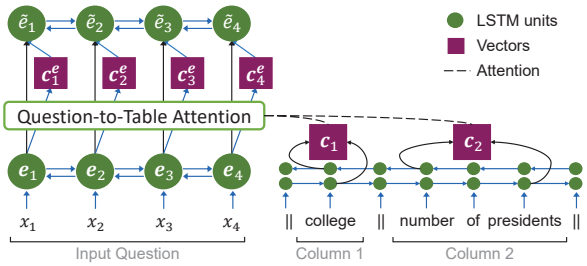

Figure 2: Table-aware input encoder (left) and table column encoder (right) used for WIKISQL.

As mentioned earlier, the meaning representations of questions are dependent on the tables. As shown in Figure 2, we encode the input question x into $\{\mathbf{e}_t\}_{t=1}^{|x|}$ using LSTM units. At each time step t, we use an attention mechanism towards table column vectors $\{\mathbf{c}_k\}_{k=1}^M$ to obtain the most relevant columns for $\mathbf{e}_t$. The attention score from $\mathbf{e}_t$ to $\mathbf{c}_k$ is computed via $u_{t,k} \propto \exp\{\alpha(\mathbf{e}_t) \cdot \alpha(\mathbf{c}_k)\}$, where $\alpha(\cdot)$ is a one-layer neural network, and $\sum_{k=1}^M u_{t,k} = 1$. Then we compute the context vector $\mathbf{c}_t^e = \sum_{k=1}^M u_{t,k}\mathbf{c}_k$ to summarize the relevant columns for $\mathbf{e}_t$. We feed the concatenated vectors $\{[\mathbf{e}_t, \mathbf{c}_t^e]\}_{t=1}^{|x|}$ into a bi-directional LSTM encoder, and use the new encoding vectors $\{\tilde{\mathbf{e}}_t\}_{t=1}^{|x|}$ to replace $\{\mathbf{e}_t\}_{t=1}^{|x|}$ in other model components. We define the vector representation of input x as:

$$\tilde{\mathbf{e}} = [\overrightarrow{\tilde{\mathbf{e}}}_{|x|}, \overleftarrow{\tilde{\mathbf{e}}}_1] \quad (12)$$

analogously to Equations (4)–(6).

`SELECT` Clause We feed the question vector $\tilde{\mathbf{e}}$ into a softmax classifier to obtain the aggregation operator `agg_op`. If `agg_col` is the k-th table column, its probability is computed via:

$$\sigma(\mathbf{x}) = \mathbf{w}_3 \cdot \tanh(\mathbf{W}_4\mathbf{x} + \mathbf{b}_4) \quad (13)$$

$$p(\texttt{agg\_col} = k|x) \propto \exp\{\sigma([\tilde{\mathbf{e}}, \mathbf{c}_k])\} \quad (14)$$

where $\sum_{j=1}^M p(\texttt{agg\_col} = j|x) = 1$, $\sigma(\cdot)$ is a scoring network, and $\mathbf{W}_4 \in \mathbb{R}^{2n \times m}$, $\mathbf{w}_3, \mathbf{b}_4 \in \mathbb{R}^m$ are parameters.

[2] `agg_op` $\in \{$*empty*, `COUNT`, `MIN`, `MAX`, `SUM`, `AVG`$\}$.
[3] `cond_op` $\in \{=, <, >\}$.

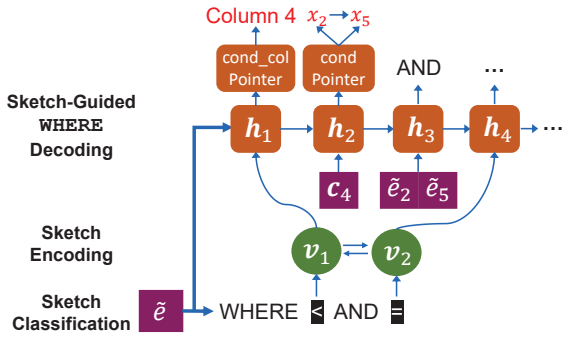

Figure 3: Fine meaning decoder of the WHERE clause used for WIKISQL.

WHERE Clause We first generate sketches whose details are subsequently decorated by the fine meaning decoder described in Section 3.2. As the number of sketches in the training set is small (35 in total), we model sketch generation as a classification problem. We treat each sketch a as a category, and use a softmax classifier to compute $p(a|x)$:

$$p(a|x) = \text{softmax}_a(\mathbf{W}_a \tilde{\mathbf{e}} + \mathbf{b}_a)$$

where $\mathbf{W}_a \in \mathbb{R}^{|\mathcal{V}_a| \times n}, \mathbf{b}_a \in \mathbb{R}^{|\mathcal{V}_a|}$ are parameters, and $\tilde{\mathbf{e}}$ is the table-aware input representation defined in Equation (12).

Once the sketch is predicted, we know the condition operators and number of conditions in the WHERE clause which follows the format "WHERE (cond_op cond_col cond) AND ...". As shown in Figure 3, our generation task now amounts to populating the sketch with condition columns cond_col and their values cond.

Let $\{\mathbf{h}_t\}_{t=1}^{|y|}$ denote the LSTM hidden states of the fine meaning decoder, and $\{\mathbf{h}_t^{att}\}_{t=1}^{|y|}$ the vectors obtained by the attention mechanism as in Equation (9). The condition column $\texttt{cond\_col}_{y_t}$ is selected from the table's headers. For the k-th column in the table, we compute $p(\texttt{cond\_col}_{y_t} = k|y_{<t}, x, a)$ as in Equation (14), but use different parameters and compute the score via $\sigma([\mathbf{h}_t^{att}, \mathbf{c}_k])$. If the k-th table column is selected, we use $\mathbf{c}_k$ for the input of the next LSTM unit in the decoder.

Condition values are typically mentioned in the input questions. These values are often phrases with multiple tokens (e.g., *Mikhail Snitko* in Table 1). We therefore propose to select a *text span* from input x for each condition value $\texttt{cond}_{y_t}$ rather than copying tokens one by one. Let $x_l \cdots x_r$ denote the text span from which $\texttt{cond}_{y_t}$ is copied. We factorize its probability as:

$$\begin{aligned}
&p(\texttt{cond}_{y_t} = x_l \cdots x_r | y_{<t}, x, a) \\
&\quad = p\left([\![l]\!]_{y_t}^L | y_{<t}, x, a\right) p\left([\![r]\!]_{y_t}^R | y_{<t}, x, a, [\![l]\!]_{y_t}^L\right) \\
&p\left([\![l]\!]_{y_t}^L | y_{<t}, x, a\right) \propto \exp\{\sigma([\mathbf{h}_t^{att}, \tilde{\mathbf{e}}_l])\} \\
&p\left([\![r]\!]_{y_t}^R | y_{<t}, x, a, [\![l]\!]_{y_t}^L\right) \propto \exp\{\sigma([\mathbf{h}_t^{att}, \tilde{\mathbf{e}}_l, \tilde{\mathbf{e}}_r])\}
\end{aligned}$$

where $[\![l]\!]_{y_t}^L / [\![r]\!]_{y_t}^R$ represents the first/last copying index of $\texttt{cond}_{y_t}$ is l/r, the probabilities are normalized to 1, and $\sigma(\cdot)$ is the scoring network defined in Equation (13). Notice that we use different parameters for the scoring networks $\sigma(\cdot)$. The copied span is represented by the concatenated vector $[\tilde{\mathbf{e}}_l, \tilde{\mathbf{e}}_r]$, which is fed into a one-layer neural network and then used as the input to the next LSTM unit in the decoder.

5 Experiments

We present results on the three semantic parsing tasks discussed in Section 4. Our implementation and pretrained models are available at https://github.com/donglixp/coarse2fine.

5.1 Experimental Setup

Preprocessing For GEO and ATIS, we used the preprocessed versions provided by Dong and Lapata (2016), where natural language expressions are lowercased and stemmed with NLTK (Bird et al., 2009), and entity mentions are replaced by numbered markers. We combined predicates and left brackets that indicate hierarchical structures to make meaning representations compact. We employed the preprocessed DJANGO data provided by Yin and Neubig (2017), where input expressions are tokenized by NLTK, and quoted strings in the input are replaced with place holders. WIKISQL was preprocessed by the script provided by Zhong et al. (2017), where inputs were lowercased and tokenized by Stanford CoreNLP (Manning et al., 2014).

Configuration Model hyperparameters were cross-validated on the training set for GEO, and were validated on the development split for the other datasets. Dimensions of hidden vectors and word embeddings were selected from $\{250, 300\}$ and $\{150, 200, 250, 300\}$, respectively. The dropout rate was selected from $\{0.3, 0.5\}$. Label smoothing (Szegedy et al., 2016) was employed for GEO and ATIS. The smoothing parameter was set to 0.1. For WIKISQL, the hidden size of $\sigma(\cdot)$

Method	GEO	ATIS
ZC07 (Zettlemoyer and Collins, 2007)	86.1	84.6
UBL (Kwiatkowksi et al., 2010)	87.9	71.4
FUBL (Kwiatkowski et al., 2011)	88.6	82.8
GUSP++ (Poon, 2013)	—	83.5
KCAZ13 (Kwiatkowski et al., 2013)	89.0	—
DCS+L (Liang et al., 2013)	87.9	—
TISP (Zhao and Huang, 2015)	88.9	84.2
SEQ2SEQ (Dong and Lapata, 2016)	84.6	84.2
SEQ2TREE (Dong and Lapata, 2016)	87.1	84.6
ASN (Rabinovich et al., 2017)	85.7	85.3
ASN+SUPATT (Rabinovich et al., 2017)	87.1	85.9
ONESTAGE	85.0	85.3
COARSE2FINE	88.2	87.7
− sketch encoder	87.1	86.9
+ oracle sketch	93.9	95.1

Table 2: Accuracies on GEO and ATIS.

Method	Accuracy
Retrieval System	14.7
Phrasal SMT	31.5
Hierarchical SMT	9.5
SEQ2SEQ+UNK replacement	45.1
SEQ2TREE+UNK replacement	39.4
LPN+COPY (Ling et al., 2016)	62.3
SNM+COPY (Yin and Neubig, 2017)	71.6
ONESTAGE	69.5
COARSE2FINE	74.1
− sketch encoder	72.1
+ oracle sketch	83.0

Table 3: DJANGO results. Accuracies in the first and second block are taken from Ling et al. (2016) and Yin and Neubig (2017).

and $\alpha(\cdot)$ in Equation (13) was set to 64. Word embeddings were initialized by GloVe (Pennington et al., 2014), and were shared by table encoder and input encoder in Section 4.3. We appended 10-dimensional part-of-speech tag vectors to embeddings of the question words in WIKISQL. The part-of-speech tags were obtained by the spaCy toolkit. We used the RMSProp optimizer (Tieleman and Hinton, 2012) to train the models. The learning rate was selected from $\{0.002, 0.005\}$. The batch size was 200 for WIKISQL, and was 64 for other datasets. Early stopping was used to determine the number of epochs.

Evaluation We use accuracy as the evaluation metric, i.e., the percentage of the examples that are correctly parsed to their gold standard meaning representations. For WIKISQL, we also execute generated SQL queries on their corresponding tables, and report the execution accuracy which is defined as the proportion of correct answers.

5.2 Results and Analysis

We compare our model (COARSE2FINE) against several previously published systems as well as various baselines. Specifically, we report results with a model which decodes meaning representations in one stage (ONESTAGE) without leveraging sketches. We also report the results of several ablation models, i.e., without a sketch encoder and without a table-aware input encoder.

Table 2 presents our results on GEO and ATIS. Overall, we observe that COARSE2FINE outperforms ONESTAGE, which suggests that disentangling high-level from low-level information during decoding is beneficial. The results also show that removing the sketch encoder harms performance since the decoder loses access to additional contextual information. Compared with previous neural models that utilize syntax or grammatical information (SEQ2TREE, ASN; the second block in Table 2), our method performs competitively despite the use of relatively simple decoders. As an upper bound, we report model accuracy when gold meaning sketches are given to the fine meaning decoder (+oracle sketch). As can be seen, predicting the sketch correctly boosts performance. The oracle results also indicate the accuracy of the fine meaning decoder.

Table 3 reports results on DJANGO where we observe similar tendencies. COARSE2FINE outperforms ONESTAGE by a wide margin. It is also superior to the best reported result in the literature (SNM+COPY; see the second block in the table). Again we observe that the sketch encoder is beneficial and that there is an 8.9 point difference in accuracy between COARSE2FINE and the oracle.

Results on WIKISQL are shown in Table 4. Our model is superior to ONESTAGE as well as to previous best performing systems. COARSE2FINE's accuracies on aggregation `agg_op` and `agg_col` are 90.2% and 92.0%, respectively, which is comparable to SQLNET (Xu et al., 2017). So the most gain is obtained by the improved decoder of the `WHERE` clause. We also find that a table-aware input encoder is critical for doing well on this task, since the same question might lead to different SQL queries depending on the table schemas. Consider the question "*how many presidents are graduated from A*". The SQL query over table "‖*President*‖*College*‖" is "`SELECT`

Method	Accuracy	Execution Accuracy
SEQ2SEQ	23.4	35.9
Aug Ptr Network	43.3	53.3
SEQ2SQL (Zhong et al., 2017)	48.3	59.4
SQLNET (Xu et al., 2017)	61.3	68.0
ONESTAGE	68.8	75.9
COARSE2FINE	71.7	78.5
− sketch encoder	70.8	77.7
− table-aware input encoder	68.6	75.6
+ oracle sketch	73.0	79.6

Table 4: Evaluation results on WIKISQL. Accuracies in the first block are taken from Zhong et al. (2017) and Xu et al. (2017).

Method	GEO	ATIS	DJANGO	WIKISQL
ONESTAGE	85.4	85.9	73.2	95.4
COARSE2FINE	89.3	88.0	77.4	95.9

Table 5: Sketch accuracy. For ONESTAGE, sketches are extracted from the meaning representations it generates.

COUNT(*President*) WHERE (*College* = *A*)", but the query over table "||*College*||*Number of Presidents*||" would be "SELECT *Number of Presidents* WHERE (*College* = *A*)".

We also examine the predicted sketches themselves in Table 5. We compare sketches generated by COARSE2FINE against ONESTAGE. The latter model generates meaning representations without an intermediate sketch generation stage. Nevertheless, we can extract sketches from the output of ONESTAGE following the procedures described in Section 4. Sketches produced by COARSE2FINE are more accurate across the board. This is not surprising because our model is trained explicitly to generate compact meaning sketches. Taken together (Tables 2–4), our results show that better sketches bring accuracy gains on GEO, ATIS, and DJANGO. On WIKISQL, the sketches predicted by COARSE2FINE are marginally better compared with ONESTAGE. Performance improvements on this task are mainly due to the fine meaning decoder. We conjecture that by decomposing decoding into two stages, COARSE2FINE can better match table columns and extract condition values without interference from the prediction of condition operators. Moreover, the sketch provides a canonical order of condition operators, which is beneficial for the decoding process (Vinyals et al., 2016; Xu et al., 2017).

6 Conclusions

In this paper we presented a coarse-to-fine decoding framework for neural semantic parsing. We first generate meaning sketches which abstract away from low-level information such as arguments and variable names and then predict missing details in order to obtain full meaning representations. The proposed framework can be easily adapted to different domains and meaning representations. Experimental results show that coarse-to-fine decoding improves performance across tasks. In the future, we would like to apply the framework in a weakly supervised setting, i.e., to learn semantic parsers from question-answer pairs and to explore alternative ways of defining meaning sketches.

Acknowledgments We would like to thank Pengcheng Yin for sharing with us the preprocessed version of the DJANGO dataset. We gratefully acknowledge the financial support of the European Research Council (award number 681760; Dong, Lapata) and the AdeptMind Scholar Fellowship program (Dong).

References

Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 2007. *Compilers: principles, techniques, and tools*, volume 2. Addison-wesley Reading.

David Alvarez-Melis and Tommi S Jaakkola. 2017. Tree-structured decoding with doubly-recurrent neural networks. In *Proceedings of the 5th International Conference on Learning Representations*, Toulon, France.

Jacob Andreas, Andreas Vlachos, and Stephen Clark. 2013. Semantic parsing as machine translation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, pages 47–52, Sofia, Bulgaria.

Yoav Artzi and Luke Zettlemoyer. 2013. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association of Computational Linguistics*, 1:49–62.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *Proceedings of the 3rd International Conference on Learning Representations*, San Diego, California.

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, Seattle, Washington. Association for Computational Linguistics.

Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural Language Processing with Python*. O'Reilly Media.

Eugene Charniak, Mark Johnson, Micha Elsner, Joseph Austerweil, David Ellis, Isaac Haxton, Catherine Hill, R. Shrivaths, Jeremy Moore, Michael Pozar, and Theresa Vu. 2006. Multilevel coarse-to-fine PCFG parsing. In *Proceedings of the Human Language Technology Conference of the NAACL*, pages 168–175, New York, NY.

Jianpeng Cheng, Siva Reddy, Vijay Saraswat, and Mirella Lapata. 2017. Learning structured natural language representations for semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pages 44–55. Association for Computational Linguistics.

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*, pages 33–43, Berlin, Germany.

Xing Fan, Emilio Monti, Lambert Mathias, and Markus Dreyer. 2017. Transfer learning for neural semantic parsing. In *Proceedings of the 2nd Workshop on Representation Learning for NLP*, pages 48–56, Vancouver, Canada.

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas Reps. 2017. Component-based synthesis for complex apis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 599–612, New York, NY.

Ruifang Ge and Raymond J. Mooney. 2005. A statistical semantic parser that integrates syntax and semantics. In *Proceedings of the Ninth Conference on Computational Natural Language Learning*, pages 9–16, Ann Arbor, Michigan.

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. 2016. Incorporating copying mechanism in sequence-to-sequence learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 1631–1640, Berlin, Germany.

Caglar Gulcehre, Sungjin Ahn, Ramesh Nallapati, Bowen Zhou, and Yoshua Bengio. 2016. Pointing the unknown words. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 140–149, Berlin, Germany. Association for Computational Linguistics.

Jonathan Herzig and Jonathan Berant. 2017. Neural semantic parsing over multiple knowledge-bases. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pages 623–628, Vancouver, Canada.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation*, 9:1735–1780.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a neural semantic parser from user feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pages 963–973, Vancouver, Canada.

Robin Jia and Percy Liang. 2016. Data recombination for neural semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 12–22, Berlin, Germany.

Tomáš Kočiský, Gábor Melis, Edward Grefenstette, Chris Dyer, Wang Ling, Phil Blunsom, and Karl Moritz Hermann. 2016. Semantic parsing with semi-supervised sequential autoencoders. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1078–1087, Austin, Texas.

Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. 2017. Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1517–1527, Copenhagen, Denmark.

Tom Kwiatkowksi, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2010. Inducing probabilistic CCG grammars from logical form with higher-order unification. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 1223–1233, Cambridge, MA.

Tom Kwiatkowski, Eunsol Choi, Yoav Artzi, and Luke Zettlemoyer. 2013. Scaling semantic parsers with on-the-fly ontology matching. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1545–1556, Seattle, Washington.

Tom Kwiatkowski, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2011. Lexical generalization in CCG grammar induction for semantic parsing. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 1512–1523, Edinburgh, Scotland.

Percy Liang, Michael I. Jordan, and Dan Klein. 2013. Learning dependency-based compositional semantics. *Computational Linguistics*, 39(2).

Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Fumin Wang, and Andrew Senior. 2016. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 599–609, Berlin, Germany.

Wei Lu, Hwee Tou Ng, Wee Sun Lee, and Luke Zettlemoyer. 2008. A generative model for parsing natural language to meaning representations. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 783–792, Honolulu, Hawaii.

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal.

Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics System Demonstrations*, pages 55–60, Baltimore, Maryland.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering*, pages 574–584, Washington, DC.

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*, pages 1532–1543, Doha, Qatar.

Slav Petrov. 2011. *Coarse-to-fine natural language processing*. Springer Science & Business Media.

Hoifung Poon. 2013. Grounded unsupervised semantic parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, pages 933–943, Sofia, Bulgaria.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pages 1139–1149, Vancouver, Canada.

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. thesis, University of California at Berkeley, Berkeley, CA.

Raymond Hendy Susanto and Wei Lu. 2017. Neural architectures for multilingual semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pages 38–44, Vancouver, Canada.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826.

Lappoon R. Tang and Raymond J. Mooney. 2000. Automated construction of database interfaces: Intergrating statistical and relational learning for semantic parsing. In *2000 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora*, pages 133–141, Hong Kong, China.

T. Tieleman and G. Hinton. 2012. Lecture 6.5—RMSProp: Divide the gradient by a running average of its recent magnitude. Technical report.

Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. 2016. Order matters: Sequence to sequence for sets. In *Proceedings of the 4th International Conference on Learning Representations*, San Juan, Puerto Rico.

Oriol Vinyals, Lukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. 2015. Grammar as a foreign language. In *Proceedings of the 28th International Conference on Neural Information Processing Systems*, pages 2773–2781, Montreal, Canada.

Yuk Wah Wong and Raymond Mooney. 2007. Learning synchronous grammars for semantic parsing with lambda calculus. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 960–967, Prague, Czech Republic.

Chunyang Xiao, Marc Dymetman, and Claire Gardent. 2016. Sequence-based structured prediction for semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 1341–1350, Berlin, Germany.

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

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: Query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1:63:1–63:26.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pages 440–450, Vancouver, Canada.

Luke Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence*, pages 658–666, Edinburgh, Scotland.

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*, pages 678–687, Prague, Czech Republic.

Sai Zhang and Yuyin Sun. 2013. Automatically synthesizing SQL queries from input-output examples. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 224–234, Piscataway, NJ.

Yuchen Zhang, Panupong Pasupat, and Percy Liang. 2017. Macro grammars and holistic triggering for efficient semantic parsing. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1214–1223. Association for Computational Linguistics.

Kai Zhao and Liang Huang. 2015. Type-driven incremental semantic parsing with polymorphism. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1416–1421, Denver, Colorado.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.