 Weakly Supervised Semantic Parsing by Learning from Mistakes

Jiaqi Guo†*, Jian-Guang Lou†, Ting Liu†, Dongmei Zhang‡
†Xi’an Jiaotong University, Xi’an, China
‡Microsoft Research, Beijing, China

jasperguo2013@stu.xjtu.edu.cn tingliu@mail.xjtu.edu.cn {jlou,dongmeiz}@microsoft.com

Abstract
Weakly supervised semantic parsing (WSP) aims at training a parser via utterance-denotation pairs. This task is challenging because it requires (1) searching consistent logical forms in a huge space; and (2) dealing with spurious logical forms. In this work, we propose Learning from Mistakes (LFM), a simple yet effective learning framework for WSP. LFM utilizes the mistakes made by a parser during searching, i.e., generating logical forms that do not execute to correct denotations, for tackling the two challenges. In a nutshell, LFM additionally trains a parser using utterance-logical form pairs created from mistakes, which can quickly bootstrap the parser to search consistent logical forms. Also, it can motivate the parser to learn the correct mapping between utterances and logical forms, thus dealing with the spuriousness of logical forms. We evaluate LFM on WikiTableQuestions, WikiSQL, and TabFact in the WSP setting. The parser trained with LFM outperforms the previous state-of-the-art semantic parsing approaches on the three datasets. Also, we find that LFM can substantially reduce the need for labeled data. Using only 10% of utterance-denotation pairs, the parser achieves 84.2 denotation accuracy on WikiSQL, which is competitive with the previous state-of-the-art approaches using 100% labeled data.

1 Introduction
Semantic parsing is the task of mapping a natural language utterance to a logical form that can be executed against a knowledge base to obtain a denotation. Much progress has been made in this area, thanks to the emergence of datasets that include a large number of utterance-logical form pairs. However, collecting such pairs at scale is generally expensive, because annotators must be skilled at programming. By contrast, collecting utterance-denotation pairs is much cheaper, because it can be performed by non-experts. Hence, it is tempting to train a semantic parser via utterance-denotation pairs, framing a weakly supervised semantic parsing problem (WSP) (Clarke et al., 2010; Liang et al., 2013; Zhang et al., 2017).

Training a parser from denotations rather than logical forms complicates training in two ways. First, training a parser requires exploring the huge space of logical forms to find those that execute to correct denotations, which we call “consistent” logical forms. This is a very difficult search problem due to the combinatorial nature of the search space. Figure 1 presents five logical forms for an utterance-denotation pair, among which the first three are consistent and the rest are mistake logical forms (they do not execute to the correct denotation). Second, consistent logical forms can be “spurious”. Spurious logical forms accidentally execute to correct denotations, but they do not reflect the meaning of utterances. For example, two of the three consistent logical forms in Figure 1 are spurious, and only the first one is “correct”, reflecting the utterance’s meaning. The presence of spurious

<table>
<thead>
<tr>
<th>Rank</th>
<th>Nation</th>
<th>Gold</th>
<th>Silver</th>
<th>Bronze</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>France</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Ukraine</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Turkey</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Sweden</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Iran</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Utterance: Who won the most silver medals?
Denotation: Iran

Figure 1: An illustrative example of weakly supervised semantic parsing.
logical forms severely hinders a parser from learning the correct mapping between utterances and logical forms.

Existing approaches for WSP can be categorized into static and dynamic, according to whether they perform searching at training time. Static approaches heuristically search consistent logical forms offline (Krishnamurthy et al., 2017; Wang et al., 2019a; Min et al., 2019). They assume that there are correct logical forms in the search results, and they do not perform searching at training time. However, this assumption may not hold when the spuriousness is severe. Considering a binary denotation (TRUE or FALSE), 50% of syntactically valid logical forms execute to the correct denotation, regardless of their semantics. Dynamic approaches do not make this assumption. They iteratively search consistent logical forms using a parser and train the parser with the search result in turn (Guu et al., 2017; Liang et al., 2017, 2018; Agarwal et al., 2019). But dynamic approaches generally suffer from a cold-start problem, because it is challenging for a randomly initialized parser to search consistent logical forms in an exponentially large space. Hence, most dynamic approaches require a set of pre-searched consistent logical forms to bootstrap the training.

In this work, we propose Learning from Mistakes (LFM for short), a simple yet effective dynamic learning framework for WSP. The core insight of LFM is that a parser will generate a huge number of mistake logical forms during searching. These mistake logical forms can be fully utilized to overcome the cold-start and spuriousness problems. In a nutshell, every time a parser makes a mistake, LFM synthesizes a faithful utterance for the mistake logical form. Then, LFM trains the parser with this utterance-logical form pair, so that the parser is taught the correct meaning of the mistake logical form. In addition, LFM also trains the parser like existing dynamic approaches, using consistent logical forms with learning objectives such as REINFORCE (Williams, 1992) and Maximum Marginal Likelihood (MML).

LFM has two major advantages over existing dynamic approaches. First, LFM does not need to pre-search consistent logical forms to warm start the training. Instead, it creates utterance-logical form pairs from mistakes on the fly to overcome the cold-start problem. Second, LFM can facilitate a parser learning the correct mapping between utterances and logical forms. Since the synthesized utterances are guaranteed to reflect the meaning of logical forms, a parser can learn the correct mapping from the synthesized utterance-logical form pairs. The idea of LFM is inspired by our human beings. Every time we make a mistake, we try to modify our knowledge to avoid suffering again in the future for the same reason (Giordana and Serra, 2001). Similarly, every time a parser makes a mistake, we try to teach the parser the correct meaning of the mistake logical form and help it avoid the mistake in the next round of searching.

To demonstrate the effectiveness of LFM, we conduct experiments on three challenging semantic parsing datasets in the WSP setting. The neural semantic parser trained with LFM achieves a denotation accuracy of 52.3 on WikiTableQuestions, 86.9 on WikiSQL, and 68.2 on TabFact, which all surpass previous state-of-the-art approaches in the same setting. Through a fine-grained analysis, we show that LFM is effective in addressing the cold-start and spuriousness problems, and LFM is more effective than prior data augmentation techniques for WSP. Also, we find that LFM can substantially reduce the need for labeled data to train a good parser. For example, the parser achieves an accuracy of 84.2 on WikiSQL using only 10% of utterance-denotation pairs, which already performs on par with previous state-of-the-art approaches.

2 Related Work

2.1 Weakly Supervised Semantic Parsing

As mentioned in the previous section, prior approaches for WSP can be categorized into static and dynamic. Static approaches, such as Krishnamurthy et al. (2017), heuristically search consistent logical forms offline and train a parser with the MML objective. When there are too many consistent logical forms for an utterance-denotation pair, they only consider top $K$ shortest logical forms (typically $K \leq 100$) and perform a beam search to approximate the sum in MML. Wang et al. (2019a) introduce an alignment model to distinguish between spurious and correct logical forms. The alignment model is jointly optimized with a parser via MML. Min et al. (2019) replace MML with a discrete hard EM objective and observe improvements on WikiSQL and some reading comprehension datasets.

Dynamic approaches iteratively search consistent logical forms using a parser and optimize the
parser via the search result in turn. For example, Liang et al. (2013) and Berant et al. (2013) perform a beam search on a parser at each training step to search consistent logical forms, and they optimize the parser with an approximated MML that sums over consistent logical forms in the beam. Guu et al. (2017) propose a randomized beam search and a $β$-meritocratic update strategy to improve the searching of consistent logical forms. Instead of using MML, Liang et al. (2017) optimize a parser with the REINFORCE algorithm (Williams, 1992). They sample logical forms at each training step to compute an unbiased estimate of the gradient. Liang et al. (2018) leverage a memory buffer of consistent logical forms to reduce the variance of policy gradient estimate. Agarwal et al. (2019) introduce an auxiliary reward function to provide fine-grained feedback for dealing with spurious logical forms. Our LFM framework also falls into this dynamic category. Unlike the approaches introduced above that primarily leverage consistent logical forms for optimization, LFM fully utilizes the mistakes made by a parser during searching to address the cold-start and spuriousness problems. Hence, LFM can be considered orthogonal to prior dynamic approaches.

There is another line of work that tackles WSP without logical forms (Neelakantan et al., 2017; Mou et al., 2017; Herzig et al., 2020). Neelakantan et al. (2017) propose a neural model that sequentially predicts symbolic operations over semi-structured tables, and the model can be trained end-to-end with utterance-denotation pairs. Herzig et al. (2020) and Eisenschlos et al. (2020) pre-train a language model for table understanding. They show that the pre-trained model can be used to address WSP with a simple cell selection module and a set of differentiable aggregation operators.

2.2 Data Augmentation for Semantic Parsing

Our work also closely relates to the area of data augmentation for semantic parsing, since LFM synthesizes utterance-logical form pairs. Jia and Liang (2016) induce a synchronous context-free grammar (SCFG) (Chiang, 2005) from manually labeled utterance-logical form pairs. They randomly sample new pairs from the SCFG and train a parser using both labeled and sampled data, leading to significant improvements on several fully supervised semantic parsing tasks. Goldman et al. (2018) manually induce an SCFG and pre-train a neural semantic parser using data sampled from the SCFG. The parser is then finetuned via utterance-denotation pairs using MML. Similar ideas have also been adopted to address the Text-to-SQL problem (Iyer et al., 2017; Yu et al., 2018, 2021). Instead of using SCFG, Guo et al. (2018), Zhong et al. (2020a), and Wang et al. (2021) train a SQL-to-question neural model via utterance-logical form pairs. They synthesize more training data by randomly sampling SQL queries and generating corresponding questions with the model. One shortcoming of the data augmentation work above is that they need to carefully design logical form sampling procedures and pre-define the amount of data to synthesize. It has been found that over-extensive data augmentation will cause a deep-learning model to overfit, leading to even worse performance than that without data augmentation (Shorten and Khoshgoftaar, 2019).

In LFM, the mistakes made by a parser serve as the source for data augmentation, and therefore, we do not need extra logical form sampling procedures. Also, we do not need to pre-define the amount of data to synthesize, because synthesis data are created at each training step. As we will show in Section 5.2, LFM is more effective than the other data augmentation techniques in WSP.

3 Learning Framework

In this section, we formally define the task of WSP and describe LFM in detail.

3.1 Preliminaries

**Task Formulation** Given a training set of $N$ examples $\{(x_i, \omega_i, y_i)\}_{i=1}^N$, where $x_i$ is an utterance,
\( \omega_i \) is the knowledge base that \( x_i \) is interested in (e.g., the semi-structured table in Figure 1), and \( y_i \) is the denotation of \( x_i \), the goal of WSP is to learn a parser (with parameter \( \theta \)) that can map an unseen utterance \( x \) to a logical form \( z \), such that \( z \) executes to the correct denotation \( y \) in the knowledge base \( \omega \), i.e., \( \|z\|^\omega = y \). The parser defines a distribution over logical forms conditioned on the given \( x \) and \( \omega \): \( P(z|x, \omega; \theta) \).

Text Generator Suppose that we have access to a text generator \( G(z, \omega) \), which generates a faithful utterance for a given logical form \( z \) and its knowledge base \( \omega \). The text generator can be implemented with either an SCFG or a neural network.

3.2 LFM: Learning from Mistakes

The learning objective \( J \) in LFM is made up of two sub-objectives: \( J_c \) and \( J_e \).

\[
J = J_c + \gamma J_e \tag{1}
\]

Like prior dynamic approaches (Guu et al., 2017; Liang et al., 2017), \( J_c \) primarily leverages consistent logical forms searched during training to optimize a parser. \( J_c \) can be instantiated as MML or REINFORCE. By contrast, \( J_e \) leverages the mistakes made by the parser to overcome the cold-start problem and facilitate the parser learning the correct mapping between utterances and logical forms. \( \gamma \) is a hyper-parameter to blend the two sub-objectives.

Figure 2 visualizes a training step in LFM. Given a training example \( (x, \omega, y) \), a set of \( K \) logical forms \( Z = \{ z_j \}_{j=1}^{K} \) are sampled from a parser via beam search or Monte Carlo sampling. Suppose that among the \( K \) logical forms, only a subset of them \( Z_c \) are consistent (\( \|z\|^\omega = y, z_1\cdots z_3 \) in Figure 2), and the remaining \( Z_e = Z - Z_c \) logical forms are mistakes (\( \|z\|^\omega \neq y, z_4\cdots z_5 \) in Figure 2).

If \( J_c \) is instantiated as MML, \( J_e \) is derived as follows:

\[
J_c(\theta) = log P(y|x, \omega) = log \sum_{\|z\|^\omega = y} P(z|x, \omega; \theta)
\]

\[
\approx log \sum_{z_j \in Z_c} P(z_j|x, \omega; \theta)
\]

\[
\nabla_{\theta} J_c \approx \sum_{z_j \in Z_c} q(z_j) \nabla_{\theta} log P(z_j|x, \omega; \theta), \tag{3}
\]

where \( q(z_j) = \frac{P(z_j|x, \omega; \theta)}{\sum_{z_i \in Z_c} P(z_i|x, \omega; \theta)} \).

If \( J_e \) is instantiated as REINFORCE, \( J_e \) is derived as follows:

\[
J_e(\theta) = E_{z \sim P(\cdot|X, \omega, \theta)} R(z) \tag{4}
\]

\[
\nabla_{\theta} J_e \approx \frac{1}{K} \sum_{z_j \in Z} R(z_j) \nabla_{\theta} log P(z_j|x, \omega; \theta), \tag{5}
\]

where \( R(z) \) is a reward function. Following prior REINFORCE-based WSP approaches (Liang et al., 2017, 2018; Agarwal et al., 2019), \( R(z) \) is set to 1 if \( \|z\|^\omega = y \); otherwise 0. To this end, equation 5 can be re-written as follows:

\[
\nabla_{\theta} J_e \approx \frac{1}{K} \sum_{z_j \in Z_c} \nabla_{\theta} log P(z_j|x, \omega; \theta) \tag{6}
\]

It is clear from Equation 3 and 6 that \( J_e \) primarily leverages consistent logical forms to optimize a parser. However, at the early stage of training, a randomly initialized parser hardly samples consistent logical forms in an exponentially large space, rendering a severe cold-start problem.

To overcome this problem, LFM fully utilizes the large number of mistake logical forms generated by the parser and introduces an extra training objective \( J_e \). Although a mistake logical form \( z_j \in Z_e \) fails to execute to \( y \) and does not reflect the meaning of \( x \), we can leverage a text generator \( G \) to generate an utterance \( \hat{x} = G(z_j, \omega) \), such that \( z_j \) reflects the meaning of \( \hat{x} \). By optimizing the parser’s likelihood of generating \( z_j \) given \( \hat{x} \) and \( \omega \), we can bootstrap the parser and overcome the cold-start problem. Also, we can motivate the parser to learn the correct mapping between utterances and logical forms. Formally, the objective \( J_e \) is defined as follows:

\[
J_e(\theta) = \sum_{z_j \in Z_e} log P(z_j|G(z_j, \omega), \omega; \theta) \tag{7}
\]

\[
\nabla_{\theta} J_e = \sum_{z_j \in Z_e} \nabla_{\theta} log P(z_j|G(z_j, \omega), \omega; \theta) \tag{8}
\]

Algorithm 1 summarizes the training procedure of LFM. In each training step, LFM first searches consistent logical forms for a given utterance-denotation pair (Line 4). Then, it optimizes the parser using consistent logical forms with objective \( J_c \) (Line 5). For the remaining mistake logical forms, LFM synthesizes their corresponding utterances and optimizes the parser with \( J_e \) (Line 6-10).
We perform this fixing and diversifying procedure for mistake logical forms before synthesizing the text generator cannot generate meaningful utterances (Line 7 in Algorithm 1).

### 3.3 Fixing and Diversifying Logical Forms

At the early stage of training, mistake logical forms are prone to violating semantic constraints. Consider the logical form $z_1$ in Table 1. The predicate `filter_num_larger` expects a number as its second argument, but a string “won” is given in $z_1$, thus violating the predicate’s semantic constraint. The text generator cannot generate meaningful utterances for such invalid logical forms. Hence, to improve the utilization of mistakes, LFM tries to pinpoint the source of violations and automatically fixes them. For example, $z_1$ is fixed by replacing “won” with a randomly generated number “2.0”.

In addition, LFM attempts to enrich the diversity of mistake logical forms by randomly replacing a logical form’s entities with proper ones in its associated knowledge base. For example, the logical form $\hat{z}_2$ in Table 1 is generated by (1) replacing the column `Gold` in $z_2$ with `Silver` which has the same data type with `Gold`; and (2) replacing the value “France” in $z_2$ with another cell value (“Turkey”) in the `Nation` column.

We perform this fixing and diversifying procedures for mistake logical forms before synthesizing their utterances (Line 7 in Algorithm 1).

---

**Table 1: Examples of fixing and diversifying logical forms.** $z_1$ is the original logical form, while $\hat{z}_i$ is the fixed or diversified logical form.

<table>
<thead>
<tr>
<th>Fixed Logical Form</th>
<th>Diversified Logical Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$z_1$ (hop Nation (filter_num_larger Gold “won” rows))</td>
<td>$\hat{z}_1$ (hop Nation (filter_num_larger Gold “2.0” rows))</td>
</tr>
<tr>
<td>$z_2$ (hop Gold (filter_in Nation “France” rows))</td>
<td>$\hat{z}_2$ (hop Silver (filter_in Nation “Turkey” rows))</td>
</tr>
</tbody>
</table>

---

**Algorithm 1: Learning from Mistakes**

**Input:** training data $\{(x_i, y_i, d_i)\}_{i=1}^N$

**Output:** final parameters $\theta$ of the parser

1. repeat
   2. Get a batch $B$ from training data;
   3. for $(x, y, d) \in B$ do
      4. Sample logical forms $Z = Z_+ \cup Z_-$ from $P(z|x, y, d; \theta)$;
      5. $d\theta \leftarrow d\theta + \nabla_{\theta} J_\theta$; // $J_\theta$
      6. for $z \in Z$ do
         7. $\hat{z} \leftarrow \text{FixAndDiversify}(z, \omega)$;
         8. $\hat{x} \leftarrow \text{GenerateUtterance}(\hat{z}, \omega)$;
         9. $d\theta \leftarrow d\theta + \gamma \nabla_{\theta} J_\theta$; // $J_\theta$
      end
   end
11. Update $\theta$ using $d\theta$;
12. until converge or early stop;

---

**Table 2: Dataset statistics.**

<table>
<thead>
<tr>
<th>WikiTQ</th>
<th>WikiSQL</th>
<th>TabFact</th>
</tr>
</thead>
<tbody>
<tr>
<td># Tables</td>
<td>2,108</td>
<td>24,241</td>
</tr>
<tr>
<td># Examples</td>
<td>18,496</td>
<td>56,355</td>
</tr>
<tr>
<td>Train</td>
<td>11,321</td>
<td>80,654</td>
</tr>
<tr>
<td>Dev</td>
<td>2,831</td>
<td>8,421</td>
</tr>
<tr>
<td>Test</td>
<td>4,344</td>
<td>15,878</td>
</tr>
</tbody>
</table>

---

**4 Experimental Setup**

In this section, we present the experimental setup for LFM, including datasets, implementations of the semantic parser and text generator.

### 4.1 Dataset and Metric

We evaluate LFM on the three challenging semantic parsing datasets in the WSP setting. Table 2 summarizes their statistics and characteristics. WikiTableQuestions (Pasupat and Liang, 2015) WikiTableQuestions (WikiTQ for short) contains semi-structured tables extracted from Wikipedia and crowdsourced question-answer (utterance-denotation) pairs about the tables. The questions involve a wide variety of operations such as comparisons, superlatives, and aggregations.

WikiSQL (Zhong et al., 2017) WikiSQL is to date the largest dataset for the Text-to-SQL problem. It consists of 24,241 tables extracted from Wikipedia and 80,654 question-SQL pairs. To experiment in the WSP setting, we obtain question-answer pairs by executing each SQL query.

TabFact (Chen et al., 2020) TabFact is a large-scale fact verification dataset with 118,275 examples. Each example consists of an utterance, a Wikipedia table, and a binary label indicating whether the facts described in the utterance are supported by the table. This verification problem can be formulated as a semantic parsing problem: an utterance is entailed if its corresponding logical form executes to `True` on the table. Unlike WikiTQ and WikiSQL, denotations in TabFact are binary, and thus, the spuriousness is much more severe.

**Metric** Following prior WSP work (Liang et al., 2013; Krishnamurthy et al., 2017), we evaluate the performance of a semantic parser via Denotation Accuracy: a predicted logical form is considered correct if it executes to the correct denotation.

### 4.2 Neural Semantic Parser

We develop a simple neural semantic parser for experiments. Given an utterance and a table, the
We implement the text generator using SCFG. An SCFG consists of a set of production rules: \( N \rightarrow (\alpha, \beta) \), where \( N \) is a non-terminal, and \( \alpha \) and \( \beta \) are sequence of terminals and non-terminals. Non-terminals in \( \alpha \) and \( \beta \) are aligned. Due to the absence of utterance-logical form pairs, we manually induced the SCFG by composing related utterances for each predicate in the query language and summarizing production rules accordingly. Since \( \alpha \) can follow the query language’s CFG, we only need to summarize \( \beta \). About 200 utterances were composed to induce the SCFG. Here is a subset of production rules in the SCFG, which are used to synthesize the canonical utterances for the mistake logical forms (\( z_4 \) and \( z_5 \)) shown in Figure 2.

\[
\begin{align*}
\text{Root} & \rightarrow (\text{Project}, \text{Project}) \\
\text{Project} & \rightarrow (\text{hop} \text{Col Target}, \text{"which Col Target"}) \\
\text{Target} & \rightarrow (\text{Arg}, \text{Arg}) \ (\text{Retrieve}, \text{Retrieve}) \\
\text{Arg} & \rightarrow (\text{argmin Col Filter}, \text{"has the least Col Filter"}) \\
\text{Retrieve} & \rightarrow (\text{first Filter}, \text{"is listed first Filter"}) \\
\text{Filter} & \rightarrow (\text{max}, \text{""}) \\
\text{Col} & \rightarrow (\text{nation}, \text{"nation"}) \ | (\text{silver}, \text{"silver"})
\end{align*}
\]

The production rules of \( \text{Col} \) are determined by the columns of a given table. Since most predicates are shared among three datasets, we can re-use their production rules in SCFG.

### 4.4 Implementation

We implement LFM and the semantic parser based on Pytorch (Paszke et al., 2019), AllenNLP (Gardner et al., 2018), and the Transformers library (Wolf et al., 2020). We instantiate \( J_c \) in LFM as REINFORCE.\(^1\) Sample size \( K \) is set to 5. \( \gamma \) is set to 1 initially and decays exponentially in each training step. We use the uncased base version of BERT in the parser. We use an AdamW (Loshchilov and Hutter, 2019) optimizer with learning rate 2e-5 and a linear decay scheduler to optimize the parameters in BERT. All remaining parameters are optimized with Adam (Kingma and Ba, 2015) using a constant learning rate 5e-4. At inference time, following prior WSP work (Liang et al., 2018), we apply a beam search of size 5, and we do not use ensemble. For all experiments, we report the averaged denotation accuracy of 5 independent runs. Section A.1 and A.3 in supplementary material provide more details about the implementation and hyper-parameters. Our source code is publicly available at https://github.com/JasperGuo/LFM.

### 5 Experimental Result

#### 5.1 Main Results

Table 3 and Table 4 compare the denotation accuracy of our parser (trained using LFM) with previous approaches on WikiTQ and WikiSQL. On

---

\(^1\) We have tried to instantiate \( J_c \) as MML, but we did not observe significant improvements over REINFORCE.
WikiTQ, our parser improves the state-of-the-art (SOTA) on five datasets, and it performs on par with the SOTA on the test set (52.3 vs. 52.7). On WikiSQL, our parser improves the SOTA from 51.9 to 53.6 on the development set, and from 84.7 to 86.9 on the test set. It is worth noting that previous approaches with pre-trained LM, such as STRUCTALIGN + GRAPPA (Yu et al., 2021) and MAPO + TaBERT (Yin et al., 2020), leveraged a wealth of external corpus to pre-train larger LMs for table understanding. Although our parser only uses the base version of BERT, it still rivals or even outperforms them on both datasets.

Table 5 compares our parser with previous approaches on TabFact. Due to the binary denotations of TabFact, previous approaches can be categorized into two groups: Semantic Parsing and Classification. While the former generates and executes a logical form to verify the facts described in an utterance, the latter sacrifices the interpretability and directly verifies the facts via a neural classification model. We can observe from the table that our parser significantly surpasses previous semantic parsing approaches, but there is still a large gap compared with the SOTA in classification.

5.2 Analysis

Effect of Learning from Mistakes To obtain an in-depth understanding of LFM, we train the parser without utterance-logical form pairs created from mistakes, which amounts to training with REINFORCE. Table 6 presents the experimental results on three datasets (w/o Mistake). We can observe that the parser’s performance drops significantly, and it lags behind SOTA approaches presented in Table 3 to Table 5 by a large margin. This result shows that jointly training with utterance-logical form pairs created from mistakes is crucial for LFM to achieve the SOTA. Figure 3 presents the denotation accuracy curves on the development set of WikiTQ. It is clear that the parser trained using LFM bootstraps and converges quickly, while the w/o Mistake variant converges much slower and ends up with lower accuracy.

To assess the effectiveness of LFM in dealing with the spuriousness of logical forms, we translate golden SQL queries in the development set of WikiSQL to corresponding logical forms in our query language, and we compare the parser’s predictions with golden logical forms. For the w/o Mistake variant, among the predictions that execute to correct denotations, 79.0% of them are semantically equivalent to golden logical forms. This number is improved from 79.0% to 87.3%, when the parser is trained using LFM, indicating that LFM can facili-
Figure 3: Dev accuracy curves of LFM and the w/o Mistake variant on WikiTQ.

tate a parser learning the correct mapping between utterances and logical forms.

Lastly, we ablate the fixing and diversifying mechanism described in Section 3.3 to understand its contribution in LFM. Experimental results are shown in Table 6 (w/o Fixing), from which we can observe that this mechanism consistently improves the parser’s performance on three datasets.

Comparison with Data Augmentation We compare LFM with other data augmentation techniques for WSP. Prior work explores data augmentation in two primary ways. (1) PRE-TRAIN. Goldman et al. (2018) first pre-train a parser with synthesized utterance-logical form pairs and then finetune the parser via utterance-denotation pairs. (2) JOINT-TRAIN. Guo et al. (2018) obtain extra utterance-denotation pairs from synthesized utterance-logical form pairs, and they jointly train the parser with original utterance-denotation pairs and the extra ones. Both PRE-TRAIN and JOINT-TRAIN synthesize utterance-logical form pairs once in offline, while LFM synthesizes pairs from mistakes in each training step.

In experiments, we synthesize a various number of utterance-logical form pairs using the SCFG (from \( \times 0.5 \) to \( \times 4 \) size of training data in WikiTQ).\(^2\) Figure 4 presents the experimental results, from which we can make two main observations. First, PRE-TRAIN improves the parser’s accuracy from 47.2 (\( \times 0.0 \)) to 50.5 (\( \times 1.5 \)) on WikiTQ, when \( \times 1.5 \) size of training data are synthesized for pre-training, but it is still lower than that achieved by LFM (53.6). Adding more synthesized data cannot further improve the accuracy, and it even hurts the performance, which is consistent with the findings in (Shorten and Khoshgoftaar, 2019). Second, JOINT-TRAIN cannot improve the parser’s performance, and we observe that training becomes very unstable when less than \( \times 2.0 \) size of training data are synthesized. These observations suggest that LFM is more effective than the prior data augmentation techniques for WSP.

We also compare a variant of LFM (LFM-RANDOM) that randomly synthesizes utterance-logical forms in each training step rather than synthesizing from mistakes. LFM-RANDOM achieves an accuracy of 53.0 on the development set of WikiTQ, which is slightly worse than LFM. This result suggests that the mistake logical forms generated by a parser during searching serve as a good source for data augmentation.

Data Efficiency Since LFM creates utterance-logical form pairs from mistakes to facilitate training, we study whether it can reduce the need for labeled data (i.e., utterance-denotation pairs).

\(^2\)We randomly sample logical forms in a top-down manner according to the query language’s CFG. Sampled logical forms must execute to non-empty denotations. Utterances are then synthesized for each logical form based on the SCFG.

---

### Table 6: Ablation study results on development sets.

<table>
<thead>
<tr>
<th></th>
<th>WikiTQ</th>
<th>WikiSQL</th>
<th>TabFact</th>
</tr>
</thead>
<tbody>
<tr>
<td>LFM</td>
<td>53.6</td>
<td>87.4</td>
<td>68.7</td>
</tr>
<tr>
<td>w/o Mistake</td>
<td>47.2 (−6.4)</td>
<td>79.8 (−7.6)</td>
<td>65.1 (−3.6)</td>
</tr>
<tr>
<td>w/o Fixing</td>
<td>53.2 (−0.4)</td>
<td>87.0 (−0.4)</td>
<td>67.5 (−1.2)</td>
</tr>
</tbody>
</table>

Figure 4: Dev accuracy on WikiTQ with a various number of synthesized utterance-logical form pairs.

Figure 5: Dev accuracy of parsers trained with a various number of utterance-denotation pairs.

---

2610
Figure 5 presents the parser’s accuracy achieved by using various proportions of labeled data. On WikiSQL, our parser achieves an accuracy of 84.5 using only 10% of labeled data, which already performs on par with the previous SOTA (85.9). With 70% of labeled data, the parser performs comparably with the one using all labeled data. Similar observations can also be made on TabFact. The parser trained with 30% of labeled data already achieves the SOTA. Adding more labeled data does not bring significant improvements. In terms of WikiTQ, our parser achieves an accuracy of 42.3 using 10% of labeled data, surpassing STRUCTAL-IGN+GRAPPA (Yu et al., 2021) in the same setting, which achieves 40.7 accuracy. The parser trained with 30% of labeled data performs on par with the w/o Mistake variant (47.7 vs. 47.2). Also, with 70% of labeled data, the parser performs comparably with the one using all labeled data. Hence, LFM can substantially reduce the need for labeled data to train a good semantic parser.

6 Conclusion & Future Work

In this work, we present LFM, a simple yet effective dynamic learning framework for WSP. LFM fully utilizes the mistake logical forms generated by a parser during searching to overcome the major challenges in WSP. Experimental results on three semantic parsing datasets show that LFM can effectively address the challenges, and LFM can substantially reduce the need for utterance-denotation pairs to train a good parser.

This work also opens up several avenues for future work. First, further improvements could be made by using a more advanced text generator in LFM. The generator is currently implemented using a hand-crafted SCFG, which often generates unnatural utterances. Second, LFM can be extended to other weakly supervised learning problems where synthesizing inputs (e.g., utterances) from latent variables (e.g., logical forms) is trivial. Consider the problem of learning to solve math word problems via utterance-answer pairs. It is trivial to synthesize an utterance from a math equation. Therefore, LFM could be applied to solve this learning problem.

Acknowledgments

We thank the anonymous reviewers for their helpful discussion and detailed comments. Ting Liu was partially supported by the National Natural Science Foundation of China (61632015, 61772408, 61833015), and the Fundamental Research Funds for the Central Universities.

Impact Statement

Semantic parsing has long been an important paradigm to solve knowledge base question answering problems. In this paper, we present a simple yet effective learning framework to address the major challenges in weakly supervised semantic parsing. This framework is not tied to any neural semantic parser or knowledge base question answering problem. Hence, we consider that there will be no certain societal consequences and ethical aspects caused by our framework.

References


2669–2680, Minneapolis, Minnesota. Association for Computational Linguistics.


Xi Victoria Lin, Richard Socher, and Caiming Xiong. 2020. Bridging textual and tabular data for cross-domain text-to-SQL semantic parsing. In Findings of the Association for Computational Linguistics:


A Supplementary Material

A.1 Implementation Details

Data Pre-processing Following Wang et al. (2019b), we identify mentions of entities (including table names, column names, and cell values) in an utterance via a string-match based method. We also identify numbers and dates in both utterances and tables using the CoreNLP toolkit. The identification results are converted to indicator features for the input encoder, as described in Section 4.2.

Sampling Logical Forms Since we instantiate \( \mathcal{J} \) as REINFORCE in experiments, we use the Monte Carlo sampling method to sample logical forms. During sampling, when the current action is SELVALUE, the sampled span is constrained to be mentions of cell values, numbers, or dates in an utterance. In this way, the parser is more likely to sample logical forms that meet the query language’s semantic constraints. Such a constraint has been widely used in prior WSP approaches to search consistent logical forms (Wang et al., 2019b; Min et al., 2019). Note that this constraint is not used during training and testing.

A.2 Context-Free Grammar

We present the context-free grammar (CFG) of our query language for each dataset. It is used in the grammar-based decoder to generate a logical form. All Col and Value rules in the three CFGs are determined by a given table and utterance.

### WikiSQL

\[
\begin{align*}
\text{Root} & \rightarrow \text{Project} \mid \text{Meta} \\
\text{Project} & \rightarrow (\text{hop Col Target}) \\
\text{Meta} & \rightarrow (\text{count Col Target}) \\
\text{Meta} & \rightarrow (\text{min Col Target}) \\
\text{Meta} & \rightarrow (\text{avg Col Target}) \\
\text{Target} & \rightarrow \text{rows} \\
\text{Target} & \rightarrow \text{Filter} \\
\text{Target} & \rightarrow (\text{intersect Filter Filter}) \\
\text{Target} & \rightarrow (\text{intersect Filter} (\text{intersect Filter Filter})) \\
\text{Filter} & \rightarrow (\text{filter in Col Value rows}) \\
\text{Filter} & \rightarrow (\text{filter number less Col Value rows}) \\
\text{Filter} & \rightarrow (\text{filter number greater Col Value rows}) \\
\text{Filter} & \rightarrow (\text{filter number equals Col Value rows}) \\
\text{Col} & \rightarrow \text{nation silver} \\
\text{Value} & \rightarrow \text{"france"} \mid \text{"turkey"}
\end{align*}
\]

### TabFact

\[
\begin{align*}
\text{Root} & \rightarrow \text{CmpDate} \mid \text{CmpNumber} \mid \text{CmpString} \\
& \mid \text{CmpPosition} \mid \text{BoolLogic} \\
\text{CmpDate} & \rightarrow (\text{date greater Date Date}) \\
\text{CmpDate} & \rightarrow (\text{date equals Date Date}) \\
\text{CmpDate} & \rightarrow (\text{date not equals Date Date}) \\
\text{CmpDate} & \rightarrow (\text{all date equals Col Value Target}) \\
\text{CmpDate} & \rightarrow (\text{all date not equals Col Value Target}) \\
\text{CmpDate} & \rightarrow (\text{all date greater equals Col Value Target}) \\
\text{CmpDate} & \rightarrow (\text{all date less equals Col Value Target}) \\
\text{Date} & \rightarrow \text{MinMax} \mid \text{Hop} \mid \text{Value} \\
\text{CmpNumber} & \rightarrow (\text{num greater Number Number}) \\
\text{CmpNumber} & \rightarrow (\text{num equals Number Number}) \\
\text{CmpNumber} & \rightarrow (\text{num not equals Number Number}) \\
\text{CmpNumber} & \rightarrow (\text{all num equals Col Value Target}) \\
\text{CmpNumber} & \rightarrow (\text{all num not equals Col Value Target}) \\
\text{CmpNumber} & \rightarrow (\text{all num greater equals Col Value Target}) \\
\text{CmpNumber} & \rightarrow (\text{all num less equals Col Value Target}) \\
\text{CmpNumber} & \rightarrow (\text{all num less equals Col Value Target}) \\
\text{Number} & \rightarrow \text{MinMax} \mid \text{Hop} \mid \text{Agg} \mid \text{CountRow} \mid \text{Value} \\
\text{CmpString} & \rightarrow (\text{is empty Col Target}) \\
\text{CmpString} & \rightarrow (\text{str equals Hop Value}) \\
\text{CmpString} & \rightarrow (\text{str not equals Hop Value}) \\
\text{CmpString} & \rightarrow (\text{mode equals Col Value Target}) \\
\text{CmpString} & \rightarrow (\text{mode not equals Col Value Target}) \\
\text{CmpString} & \rightarrow (\text{all str equals Col Value Target}) \\
\text{CmpString} & \rightarrow (\text{all str not equals Col Value Target}) \\
\text{Hop} & \rightarrow (\text{hop Col Target}) \\
\text{MinMax} & \rightarrow (\text{max Col Target}) \mid (\text{min Col Target}) \\
\text{CountRow} & \rightarrow (\text{count distinct Col Target}) \\
\text{CountRow} & \rightarrow (\text{count Target}) \mid (\text{half Target}) \\
\text{Agg} & \rightarrow (\text{sum Col Target}) \mid (\text{average Col Target}) \\
\text{Agg} & \rightarrow (\text{diff Col Target Target}) \\
\text{Target} & \rightarrow \text{Arg} \mid \text{Filter} \mid \text{rows} \\
\text{Filter} & \rightarrow (\text{union Filter Filter}) \\
\text{Filter} & \rightarrow (\text{intersect Filter Filter}) \\
\text{Filter} & \rightarrow (\text{filter in Col Value rows}) \\
\text{Filter} & \rightarrow (\text{filter not in Col Value rows}) \\
\text{Filter} & \rightarrow (\text{filter number less Col Value rows}) \\
\text{Filter} & \rightarrow (\text{filter number greater Col Value rows}) \\
\text{Filter} & \rightarrow (\text{filter number equals Col Value rows}) \\
\text{Filter} & \rightarrow (\text{filter date less Col Value rows}) \\
\text{Filter} & \rightarrow (\text{filter date greater Col Value rows}) \\
\text{Filter} & \rightarrow (\text{filter date equals Col Value rows}) \\
\text{Arg} & \rightarrow (\text{argmax Col Filter}) \\
\text{Arg} & \rightarrow (\text{argmin Col Filter}) \\
\text{Col} & \rightarrow \text{nation silver} \\
\text{Value} & \rightarrow \text{"france"} \mid \text{"turkey"}
\end{align*}
\]
A.3 Hyper-Parameters

Table 7 lists the hyper-parameters of our neural semantic parser on three datasets. Most hyper-parameters are shared among three datasets. Experimental results reported in Section 5 are averaged over 5 random runs using seeds {100, 200, 300, 400, 500}. Experiments are conducted on P40 GPUs with 24GB memory.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>WikiTQ</th>
<th>WikiSQL</th>
<th>TabFact</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input Encoder</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pre-train LM</td>
<td>BERT-base</td>
<td>BERT-base</td>
<td>BERT-base</td>
</tr>
<tr>
<td>Indicator Feature Embedding Size</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Input Encoder Output Size</td>
<td>256</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>Use Indicator Features for Synthesized Data</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Grammar-based Decoder</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decoder Layer</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Decoder RNN Cell</td>
<td>LSTM</td>
<td>LSTM</td>
<td>LSTM</td>
</tr>
<tr>
<td>Decoder Input Dropout</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>Decoder Hidden Size</td>
<td>256</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>Production Rule Embedding Size</td>
<td>256</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>Non-Terminal Embedding Size</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Decoder Attention Hidden Size</td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>APPLYRULE Classifier Dropout</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>SELCOLUMN Pointer Network Hidden Size</td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>SELCOLUMN Pointer Network Dropout</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>SELVALUE Pointer Network Hidden Size</td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>SELVALUE Pointer Network Dropout</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>Maximum Decode Step</td>
<td>25</td>
<td>25</td>
<td>30</td>
</tr>
<tr>
<td><strong>LM</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>$\gamma$ decay rate</td>
<td>5e-5</td>
<td>5e-5</td>
<td>2e-4</td>
</tr>
<tr>
<td><strong>Others</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Epsilon Greedy Rate for Sampling</td>
<td>0.15</td>
<td>0.15</td>
<td>0.0</td>
</tr>
<tr>
<td>Batch Size</td>
<td>16</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Learning Rate</td>
<td>5e-4</td>
<td>5e-4</td>
<td>5e-4</td>
</tr>
<tr>
<td>Pre-Train LM Learning Rate</td>
<td>2e-5</td>
<td>2e-5</td>
<td>2e-5</td>
</tr>
<tr>
<td>Pre-Train LM Learning Rate Freeze Step</td>
<td>3000</td>
<td>1500</td>
<td>0</td>
</tr>
<tr>
<td>Pre-Train LM Learning Rate Warmup Step</td>
<td>1500</td>
<td>1500</td>
<td>518</td>
</tr>
<tr>
<td>Entropy Regularization</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Entropy Regularization Weight</td>
<td>-</td>
<td>-</td>
<td>0.1</td>
</tr>
<tr>
<td>Gradient Clip</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Beam Size</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 7: Hyper-Parameters of our neural semantic parser on three datasets.