"Learning Structured Natural Language Representations for Semantic Parsing" 阅后总结

任务介绍

本文介绍的任务，语义解析（Sematic Parsing），是将自然语言的句子转换为机器可解析的语义表达。具体而言，在这篇文章中，研究的问题为：

给定一个知识库(knowledge base)$\mathcal{K}$，以及标注有对应的匹配知识库的语义表达（grounded meaning representation）G或者问题答案（denotation）y的自然语言句子（问题）x，要学习得到一个语义解析器，使之可以完成经由一个中间未匹配知识库的语义表达U，来将x映射到对应的G的功能。

相关工作

大多数现有工作将语义解析的处理方法分成两类，第一类是通过一个任务相关的文法将自然语言句子直接解析（parse）、匹配知识图谱（grounded to a knowledge base）转为语义表达。第二类方法是首先使用句法分析器将自然语言句子解析为任务无关的中间表达（intermediate representation），然后再将中间表达转换为匹配知识图谱的最终表达。

第一类方法

根据前人工作来看，目前使用编码器-解码器（encoder-decoder）模型处理语义解析的方法依然应当归为第一类方法。这种方法减少了领域相关假设、文法学习以及大量的特征工程工作的需要。但是这种模型无法去解释语义聚合（meaning composition）是如何进行的，这也是该类模型极大灵活性的代价。而**语义聚合的可解释性**在处理边界（modeling limitation）问题上有着很大的作用。另外，由于缺乏任务相关的先验知识的引入，学习任务关于可能推倒（derivation）的考虑上以及**目标输出的不规范性**（ill-formed，如解析结果少了或多了若干括号）问题上不能得到很好的限制。

第二类方法

第二类方法的一个优势在于产生了可重复利用的中间表达，这种中间表达由于其任务无关性可以一定程度上处理未出现的词（unseen words）以及不同领域的知识迁移（knowledge transfer）。

文章动机与主要贡献

为了解决目标输出的不规范性问题以及提供可解释的语义聚合过程，这篇文章提出了一个神经语义解析器（neural semantic parser）。与其他方法相比，该模型没有使用外部的解析器（如dependency parser）以及手工设计的CCG 文法，而是采用了基于状态转移（transition-based）的方法生成谓词-参数（predicate-argument）形式的中间表达，由此**避免了生成不规范形式**的语义表达。

另外与目前语义解析问题中大多数采用的CKY类似的自底向上（bottom-up）解析策略相比，本文提出的方法不需要对自然语言的句子结构的进行人为的特征分解（decomposition），可以利用大量的非局部特征。基于假设匹配知识图谱后的语义表达与未匹配的语义表达同构（isomorphic）的假设，本文提出的状态转移模型的输出最终可以匹配到某一个知识图谱。

模型细节与代码

整个网络是在标注了对应的逻辑形式或者问题答案的自然语言句子集上进行端到端训练的。这篇文章附上的开源代码只给出了适用于标注逻辑形式的数据集的版本，下面将结合具体训练样本的训练过程以及代码来进行这部分的介绍。

适用标注逻辑形式的数据集的版本

这里的逻辑形式为Funql（官网链接http://www.funql.org/），一个训练样本及其标注为

> how many states have a city called rochester
>
> $\mathbf{answer(count(state(loc(city(cityid(}\mathit{rochester}\mathbf{, \_)))))))}$ $(*)$

，下面结合对该样本的训练更新过程进行阐述。

训练过程

输入：

- 训练样本集（这里是用的语料库为GEOQUERY，包含880个关于美国地理信息的问题和对应的逻辑查询语句）
- 通用谓词表（这些谓词与领域无关，例如(*)中的answer, count, city, loc, cityid, _等，详细的关于通用谓词表见附录-通用谓词）

输出：

- 训练好的语义分析器，能够将自然语言问题映射到相应的逻辑表达（Funql）

过程：

1. 将问题的单词加入**单词词典**（这里和下文提到的词典都是为了之后神经网络的softmax输出对应输出字符串而建立的）

```
word_vocab.feed_all(sen)
```

2. 对逻辑表达进行**解析**得到**语法树**，可以从语法树中获得其中包含的**非终结符**和**终结符**以及能够更直接指导状态转移系统训练的**标注信息**U（ungrouned meaning representation）。

   其中$U=(a,u)$，

   - a为状态转移系统的动作（action）序列，包括$\mathbf{NT},\mathbf{TER}$和$\mathbf{ACT}$（对应原文的$\mathbf{RED}$），分别代表在栈中加入一个非终结符、在栈中加入一个终结符以及规约。
   - u为项（包括终结符和非终结符）序列，为$\mathbf{NT},\mathbf{TER}$动作的参数。

   在对逻辑表达解析以及构造语法树时，

   - 首先将逻辑表达转换为另一种方便处理形式：

     $$a(b(c))\rightarrow(a(b\ c))$$

     对应这里

     $$\begin{aligned}&\textbf{answer(count(state(loc(city(cityid(}\textit{rochester}\textbf{,\_))))))}\rightarrow\\&\textbf{(answer(count(state(loc(city(cityid(}\textit{rochester}\ \textbf{\_)))))))}\end{aligned}$$

- 交互调用

  ```
  def parse_list(tokens) #对表达式深入分解（进到括号内），直到不能分解（没有括号）
  ```

  与

  ```
  def parse_operands(tokens) #每个括号后紧跟的为运算符，也即动作，对这些动作进行记录
  ```

  最后得到了语法树的**嵌套list表示**，这里为

  ```
  <type 'list'>: ['answer', ['count', ['state', ['loc', ['city',
  ['cityid', 'rochester,', '_']]]]]]
  ```

(这两个函数的完整代码见附录-代码-parse_list和附录-代码-parse_operands)

- 得到所有的终结符和非终结符，加入对应的词典

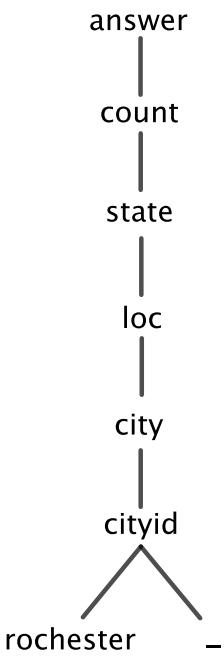
```
nt_vocab.feed_all(nt)
ter_vocab.feed_all(ter)
```

- 最后参照语法树的嵌套list表示，**构建语法树**。

```
def construct_from_list(self, content) #递归深度优先的自底向上构建树，将嵌套list表示里的list对应子树根节点；list中从第二个元素开始算起，若为字符串则将其对应为叶结点。
```

(完整代码见附录-代码-construct_from_list)

这里得到的语法树为



- 再根据语法树得到U。这部分可以自定向下（top-down）采用先序（pre-order）遍历（本文介绍的），也可以采用自底向上（bottom_up）遍历。

```
def top_down(self, identifier, general_predicates) #对语法树进行先序遍历，遍历到叶节点时在项序列u里加入对应字符串，在动作序列a里加入'NT'；遍历遇到子树根节点时在项序列u里加入对应字符串content，如果该字符串为通用谓词则在动作序列a里加入'NT(content)',否则加入'NT'；遍历完一个子树进行回溯时在动作序列a里加入'ACT'。
```

这里得到

  - 动作序列a

```
<type 'list'>: ['NT(answer)', 'NT(count)', 'NT', 'NT', 'NT',
'NT(cityid)', 'TER', 'TER', 'ACT', 'ACT', 'ACT', 'ACT', 'ACT',
'ACT']
```

- 项序列u

```
<type 'list'>: ['rochester,', '_', 'cityid', 'city', 'loc',
'state', 'count', 'answer']
```

注：这里的_或是可能出现的**all**也为通用谓词，但在状态转移系统里对应的动作都为$\mathbf{TER}$。

（完整代码见附录-代码-top_down）

3. 上一步得到的训练用的信息为

```
word_tokens[fname].append(sen) #自然语言问题句子对应的单词列表

tree_tokens[fname].append(tree_token) #上一步得到的u
tran_actions[fname].append(action) #上一步得到的a
```

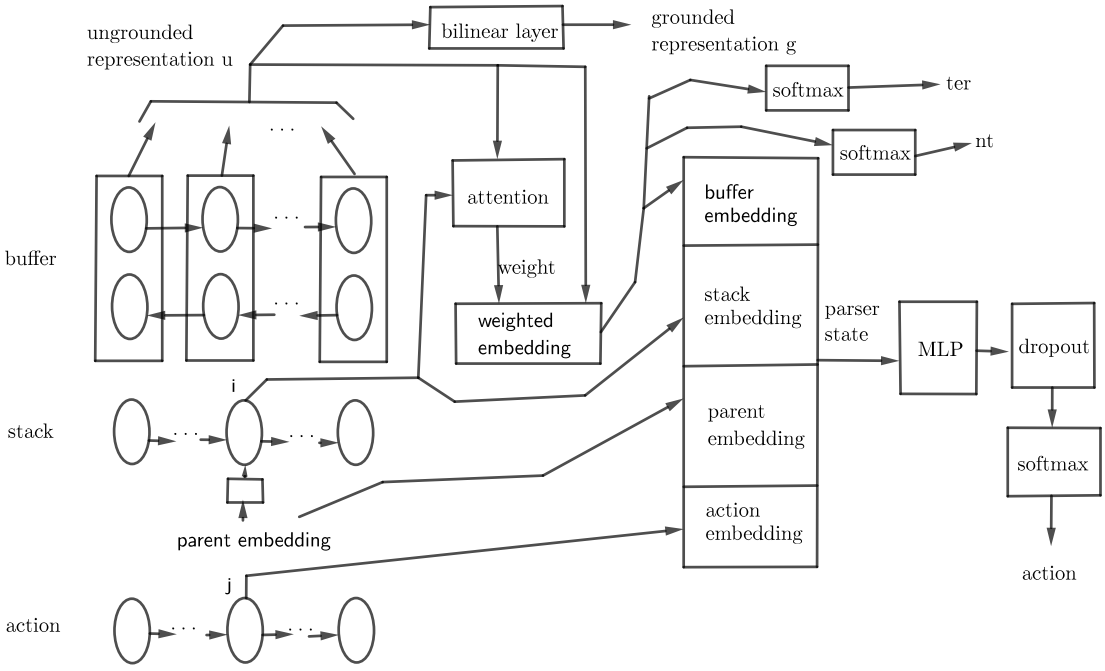
对应的

```
word_tokens, tree_tokens, tran_actions #键为{'train','valid','test'}值为
训练用信息的dict
```

，从这三个dict中按照序号各拿出一个同序号位置的元素即构成了一个训练用实例(sen, u, a)。

4. 下面介绍计算图（computing graph）的构建以及状态转移系统的转移过程。

这里训练的网络结构为：



对每个训练实例，首先将sen中的单词序列编码映射到一个双向LSTM的输出的隐层向量序列**buffer**

```
tok_embeddings =
[self.word_input_layer(self.word_lookup[self.word_vocab[word]]) for
word in words]

#get buffer representation with blstm
forward_sequence, backward_sequence =
self.blstm.predict(tok_embeddings)
buffer = [dy.concatenate([x, y]) for x, y in
          zip(forward_sequence,reversed(backward_sequence))]
#输出为两个方向相反的LSTM的输出的拼接（concatenate）
```

接下来自顶向下的生成语义解析树。

生的方式为根据状态转移系统的**buffer**，**stack**以及**action**信息来预测下一个动作，并和标注的动作比较得到相应的对数概率加入到loss里；然后根据**stack**的状态得到**可行动作**（valid action，来限制状态转移系统的动作），并根据当前标注动作分情况（$\mathbf{NT, NT\_general, TER, ACT}$）进行子操作，在每个子操作中根据**buffer**信息来预测动作的参数，并和标注的动作的参数来比较得到相应的对数概率加入到loss里。

下面分几部分阐述计算图构建以及状态转移过程

- 根据状态转移系统的**buffer**，**stack**以及**action**信息来预测下一个动作

  使用**stack**的当前隐层输出stack_embedding以及**buffer**隐层向量序列作为输入根据注意力机制来计算的权重，将**buffer**赋权累加（这里有不同的赋权累加方式）得到**buffer**的嵌入表示buffer_embedding。再在**stack**中从后往前找到最后入栈的非终结符（对应子树根节点），记录其对应的词向量经线性层（linear layer）输出的嵌入表示parent_embedding（还未经过隐层单元处理，下文称其为**原始表达**（raw representation））。最后将三个嵌入表示拼接得到解析器状态的嵌入表示parser_state:

  $$\mathbf{parser\_state} = \mathbf{buffer\_embedding} \oplus \mathbf{stack\_embedding} \oplus \mathbf{parent\_embedding}$$

  再将parser_state依次经由一层隐层的前馈神经网络、dropout处理、线性层处理、softmax处理得到可行动作的对数概率，并和标注的当前动作对比，将对数概率加入到loss中。

```
stack_embedding = stack[-1][0].output() # stack[-1] means the last
one in the list 'stack'.
action_summary = action_top.output()
word_weights = self.attention(stack_embedding, buffer) # Get the
attention weight list of the encoding words.
buffer_embedding, _ = attention_output(buffer, word_weights,
'soft_average')
# Get the attention-weighted word embedding vectors (a vector).

for i in range(len(stack)):
    if stack[len(stack)-1-i][1] == 'p':
        parent_embedding = stack[len(stack)-1-i][2]
        # parent_embedding is just the embeding vector computed
from
        # the word vector into a linear layer Wx+b.
        # Just the word vector without processing
        #  into the hidden state.
        break
parser_state = dy.concatenate([buffer_embedding, stack_embedding,
parent_embedding, action_summary])
h = self.mlp_layer(parser_state)

if options.dropout > 0:
    h = dy.dropout(h, options.dropout)

if len(valid_actions) > 0:
    log_probs = dy.log_softmax(self.act_proj_layer(h),
valid_actions)
    assert action in valid_actions, "action not in scope"
    losses.append(-dy.pick(log_probs, action))
    # Append the corresponding log probability of the action
    # annotated by the training examples.
```

- 根据**buffer**信息来预测动作的参数
  - 对于动作**NT, TER**，根据上一步预测动作中得到的注意力机制权重，对**buffer**进行加权累加，得到输出的特征，再经由线性层以及softmax处理得到对数概率，与相应标注的非终结符或终结符对比，将得到的对数概率加入loss中。

    对于状态转移处理，将标注的非终结符或终结符对应的词向量经由线性层得到嵌入表示nt_embedding（ter_embedding），作为LSTM的下一个隐层单元的输入，输入进去得到当前栈的状态stack_state，将三元组（对于动作**NT**）

    $$(\text{stack\_state}, \text{'p'}, \text{nt\_embedding})$$

    或（对于动作**TER**）

    $$(\text{stack\_state}, \text{'c'}, \text{ter\_embedding})$$

    入栈，其中'p'和'c'分别对应父节点和子节点。

- 对于动作**NT_general**，因为可以由action的内容得到具体的领域无关谓词的字符串，不作参数预测处理。

下面的代码只展示了关于动作**NT**的处理，其他类似。

```
# If the aciton is NT, then caculate the log probability of
# the corresponding annotated term choices.
#generate non-terminal
nt = self.nt_vocab[oracle_tokens.pop(0)]
#no need to predict the ROOT (assumed ROOT is fixed)
if word_weights is not None:
    train_selection = self.train_selection if np.random.randint(10)
> 5 else "soft_average"
    output_feature, output_logprob = attention_output(buffer,
word_weights, train_selection)
    log_probs_nt =
dy.log_softmax(self.nt_proj_layer(output_feature))
    losses.append(-dy.pick(log_probs_nt, nt))
    # Append the corresponding log probability of the NT
    # annotated by the training examples.

    if train_selection == "hard_sample":
        baseline_feature, _ = attention_output(buffer,
word_weights, "soft_average")
        log_probs_baseline =
dy.log_softmax(self.nt_proj_layer(baseline_feature))
        r = dy.nobackprop(dy.pick(log_probs_nt, nt) -
dy.pick(log_probs_baseline, nt))
        losses.append(-output_logprob * dy.rectify(r))

stack_state, label, _ = stack[-1] if stack else (stack_top, 'ROOT',
stack_top)
nt_embedding = self.nt_input_layer(self.nt_lookup[nt])
stack_state = stack_state.add_input(nt_embedding)
stack.append((stack_state, 'p', nt_embedding)) # 'p' for parent.
```

- 规约过程

规约时，先弹栈至刚将语义解析树的父节点对应的三元组弹出，记录每个三元组中的原始表达（也即词向量经一层线性层处理得到的向量）。将弹出的得到的叶节点对应的原始表达求平均，再和父节点的原始表达拼接得到组合表达composed_rep，再将组合表达作为**stack**的LSTM的下一个输入，构造新的隐层单元stack_state，最后将三元组

$$(\text{stack\_state}, \text{'c'}, \text{composed\_rep})$$

入栈。

```
#subtree completion
found_p = 0
path_input = []
#keep popping until the parent is found
while found_p != 1:
    top = stack.pop()
    top_raw_rep, top_label, top_rep = top[2], top[1], top[0]
    # Raw representation, label('c' or 'p') and representation
respectively.
    path_input.append(top_raw_rep)
    if top_label == 'p':
        found_p = 1
parent_rep = path_input.pop()
composed_rep =
self.subtree_input_layer(dy.concatenate([dy.average(path_input),
parent_rep]))

stack_state, _, _ = stack[-1] if stack else (stack_top, 'ROOT',
stack_top)
stack_state = stack_state.add_input(composed_rep)
stack.append((stack_state, 'c', composed_rep))
reduced = 1
```

这部分的完整代码见附录-代码-top_down_train

5. 最后将loss反向传播可更新所有的参数。这里先给出用于训练预测动作序列的目标函数：（训练逻辑表达预测的目标函数在下文给出）

$$\mathcal{L}_a = \sum_{x \in \mathcal{T}} \log p(a|x) = \sum_{x \in \mathcal{T}} \sum_{t=1}^{n} \log p(a_t|x),$$

其中$\mathcal{T}$表示训练样本集。

测试过程

测试过程与训练过程相比，每次状态转移用到的动作以及动作的参数都是经过类似过程取对数概率最大对应的来预测得到的，而非采用标注信息。

根据问题答案标注获得标注逻辑表示

给出了问题答案的标注，本文通过实体链接（entity linking），将自然语言问题中的实体与知识库的实体进行匹配，可以得到一些候选子图，这些子图可以生成候选的逻辑表示，最终选择可以搜索的到正确答案的逻辑表示作为标注逻辑表示指导训练。

对于没有经过实体链接的训练集，本文使用7个手工模版，配合知识库，将训练集的实体标注出来，然后用这些实体去训练结构化感知机（structrued perceptron）来进行实体消歧。

中间逻辑表示与最终逻辑表示

对于有逻辑表示标注的训练样本，最终逻辑表示（也即匹配了知识库的逻辑表示）和中间逻辑表示是一样的；而适用于仅仅标注了答案的训练样本的版本的代码没有完全给出（一些重要的部分没有再代码里实现）。这里结合作者引用的另外一文章

Tao Lei, Regina Barzilay, and Tommi Jaakkola. 2016. Rationalizing neural predictions. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas, pages 107–117.

来讨论可能的具体实现方式。

- 中间逻辑表示（ungrounded representation）和最终逻辑表示（grounded representation）之间的映射关系：作者给出了一个假设，假设两种同样语义的逻辑表示是同构的，也即两种表示的语法树相同，只是谓词或者终结符的字符串可能不同。所以在这种假设下，两者之间的转换仅仅变成了一个词汇映射的问题。
- 映射关系建模：使用双线性神经网络来刻画中间逻辑表示项u_t（ungrounded term）到最终逻辑表示项g_t（grounded term）的后验概率，

  $$p(g_t|u_t) \propto \exp \overrightarrow{u_t} \cdot W_{ug} \cdot \overrightarrow{g_t},$$

  其中，$\overrightarrow{u_t}$为上文提到的以自然语言问题输入为输入的双向LSTM的关于某一个中间逻辑表示项的输出；$\overrightarrow{g_t}$为最终逻辑表示项的嵌入表示，W_{ug}为权值矩阵。
- 特定领域（closed domain）vs. 开放领域（open domain）：对于特定领域，作者认为中间层的逻辑表示已经可为语义解析提供足够多的上下文信息，可以直接选择对应上述后验概率最大的最终表达作为输出（在特定领域内，一个中间表示项一般只匹配一个最终表示项）；而对于开放领域，如Freebase，这种表示得建模显得有些乏力。

  对于开放领域语义解析，本文设计训练了一个额外的排序模型，对可能的最终表示进行排序取序最靠前的解。这个排序模型使用最大熵分类器，以手工设计的关于两种逻辑表示之间的关系的信息表示为特征，优化以使得预测的最终逻辑表示在知识库上搜索的到的答案准确率最高。
- 目标函数：由于逻辑中间表示项是隐变量（latent variable），作者采用最大化逻辑最终表示项的期望对数似然$p(u|x) \log p(g|u,x)$的方式来构建优化目标：

  $$\begin{aligned}\mathcal{L}_g &= \sum_{x \in \mathcal{T}} \sum_u [p(u|x) \log p(g|u,x)] \\ &= \sum_{x \in \mathcal{T}} \sum_u \left[p(u|x) \sum_{t=1}^{k} \log p(g_t|u_t) \right]\end{aligned}$$

  结合上文提到的对动作预测的优化目标$\mathcal{L}_a$，可以得到最终的优化目标：

  $$\mathcal{L}_G = \mathcal{L}_a + \mathcal{L}_g.$$
- 目标函数优化：作者采用了另一篇文献优化有隐变量的模型的算法。在这篇引用文章中，处理的问题是文本分类，和本文的模型结构类似，有两层结构（涉及到一个生成器和一个编码器，生成器生成中间变量），中间连接的变量是隐变量。

  这片引用文章的求解任务是想要对文本提取一些原文的片段（对应$\mathbf{gen}(\cdot)$，generate）作为梗概信息（rationales），使得这些梗概信息尽量短的前提下尽可能与原始文本作为输入进行文本分类（对应$\mathbf{enc}(\cdot)$，encode）的效果一致。这就涉及到了生成梗概信息的问题，而引用文章的作者又试图想把生成器和编码器一起训练（训练$\mathbf{enc}(\mathbf{gen}(\mathbf{x}))$）。

  引用文章的最终的目标函数为：

$$\mathbf{cost}(\mathbf{z},\mathbf{x},\mathbf{y}) = \mathcal{L}(\mathbf{z},\mathbf{x},\mathbf{y}) + \Omega(\mathbf{z}),$$

其中，$\mathbf{z},\mathbf{x},\mathbf{y}$分别表示梗概信息，原始输入以及标注结果；$\mathcal{L}$惩罚编码器和标注结果的距离，$\Omega$惩罚中间生成的梗概的长度。由于中间项在训练过程中不能被提供，所以作者试图最小化期望损失（相当于把$\mathbf{z}$消掉）：

$$\min_{\theta_e,\theta_g} \sum_{(\mathbf{x},\mathbf{y})\in D} \mathbb{E}_{\mathbf{z}\sim\mathbf{gen}(\mathbf{x})}[\mathbf{cost}(\mathbf{z},\mathbf{x},\mathbf{y})],$$

其中$\boldsymbol{\theta_e}$，$\boldsymbol{\theta_g}$分别表示编码器和生成器的参数。作者假设用生成器采样中间项对于问题是可行的。借助恒等式

$$(\log f(\theta))' = f'(\theta)/f(\theta)$$

可以求得上述期望损失对两个参数的导数：

$$\frac{\partial E_{\mathbf{z}\sim\mathbf{gen}(\mathbf{x})}[\mathbf{cost}(\mathbf{z},\mathbf{x},\mathbf{y})]}{\partial\theta_g} = E_{\mathbf{z}\sim\mathbf{gen}(\mathbf{x})}[\mathbf{cost}(\mathbf{z},\mathbf{x},\mathbf{y})\frac{\partial\log p(\mathbf{z}|\mathbf{x})}{\partial\theta_g}]$$

以及

$$\frac{\partial E_{\mathbf{z}\sim\mathbf{gen}(\mathbf{x})}[\mathbf{cost}(\mathbf{z},\mathbf{x},\mathbf{y})]}{\partial\theta_e} = E_{\mathbf{z}\sim\mathbf{gen}(\mathbf{x})}[\frac{\mathbf{cost}(\mathbf{z},\mathbf{x},\mathbf{y})}{\partial\theta_e}]$$

这样对$\mathbf{z}$使用生成器进行采样，固定$\mathbf{z}$就可以求得上述两式里边的微分，进而可以求得对期望损失的微分，最后使用梯度优化的算法来更新参数。从两个微分等式可以看出，生成$\mathbf{z}$，可以利用编码器的预测结果指导更新生成器的参数；生成器被优化后，中间项的分布发生了改变，又会指导编码器的优化过程，类似于EM算法交互更新来学习隐变量的过程。

类比引用文章的求解方式，可以假定使用之前提到的双向LSTM来对逻辑中间表示项进行采样可以满足问题需求，则有

$$\frac{\partial\mathcal{L}_g}{\partial\theta_u} = \sum_{x\in\mathcal{T}} \mathbb{E}_{u\sim\mathbf{BiLSTM}(x)}[\log p(g|u,x)\frac{\partial\log p(u|x)}{\partial\theta_u}],$$

其中，$\mathbf{BiLSTM}(x)$为逻辑中间表示的生成器，θ_u为生成器的参数。在代码实现时，采用基于计算图的深度学习框架，可以直接采样构造目标函数然后反向传播即可。

实验

数据集

- GEOQUERY：关于美国地理的880个问题以及知识库，并标注有逻辑表示。这些问题句子虽然组合性（compositional）强，但语言过于简单，词汇量过于小，大多数问题涉及实体不超过一个。
- SPADES：包含$93,319$个从数据集CLUEWEB09通过随机去除实体获得的问题集合，仅仅标注有问题答案。虽然语料的组合性不强，但规模很大，并且每个句子包含2个或多个实体。
- WEBQUESTIONS：包含5810个问题-答案对。同样组合性不强，但问题均来源于真实的网络数据。
- GRAPHQUESTIONS：包含5166个问题-答案对，来源于将500个Freebase的图查询由亚马逊员工转换成的自然语言问题。

评价指标

本文使用准确率以及F1值作为评价指标，这里通过计算预测得到的语义表达与真实标注的一致比率来得到。

实验结果

Models	F1
Berant et al. (2013)	35.7
Yao and Van Durme (2014)	33.0
Berant and Liang (2014)	39.9
Bast and Haussmann (2015)	49.4
Berant and Liang (2015)	49.7
Reddy et al. (2016)	50.3
Bordes et al. (2014)	39.2
Dong et al. (2015)	40.8
Yih et al. (2015)	52.5
Xu et al. (2016)	53.3
Neural Baseline	48.3
SCANNER	49.4

Table 3: WEBQUESTIONS results.

Models	F1
SEMPRE (Berant et al., 2013)	10.80
PARASEMPRE (Berant and Liang, 2014)	12.79
JACANA (Yao and Van Durme, 2014)	5.08
Neural Baseline	16.24
SCANNER	17.02

Table 4: GRAPHQUESTIONS results. Numbers for comparison systems are from Su et al. (2016).

Models	Accuracy
Zettlemoyer and Collins (2005)	79.3
Zettlemoyer and Collins (2007)	86.1
Kwiatkowksi et al. (2010)	87.9
Kwiatkowski et al. (2011)	88.6
Kwiatkowski et al. (2013)	88.0
Zhao and Huang (2015)	88.9
Liang et al. (2011)	91.1
Dong and Lapata (2016)	84.6
Jia and Liang (2016)	85.0
Jia and Liang (2016) with extra data	89.1
SCANNER	86.7

Table 5: GEOQUERY results.

Models	F1
Unsupervised CCG (Bisk et al., 2016)	24.8
Semi-supervised CCG (Bisk et al., 2016)	28.4
Neural baseline	28.6
Supervised CCG (Bisk et al., 2016)	30.9
Rule-based system (Bisk et al., 2016)	31.4
SCANNER	31.5

Table 6: SPADES results.

- Table 5：下边隔开的一块内的模型均为神经模型。**Dong and Lapata (2016)** 将语义解析问题视为序列预测问题，使用LSTM直接将自然语言句子映射到逻辑形式。
- Table 6：这里之前的工作与本系统处理的总体思路类似，均是将自然语言句子映射到中间逻辑表示然后再匹配知识库。
- Table 3：隔开的第一块内的模型均为符号系统（symbolic systems）模型。其中 **Bast and Haussmann (2015)** 提出了一个相反的问答模型，不能产生语义表达。**Berant and Liang (2015)** 借鉴模仿学习（imitation learning）的思想，提出了一个复杂的基于智能体（agenda-based）的解析器。**Reddy et al. (2016)** 与本文概念上相似，也是通过语义中间表示来训练一个语义解析器，其中这个中间表示基于依存关系解析器的输出构建。

  隔开的第二块内的模型为一些神经系统。**Xu et al. (2016)**在这个数据集上到了目前最好的性能，他们的系统利用维基百科来过滤掉从Freebase抽取的错误候选答案。
- Table 4：比较的是三个符号系统和一个神经系统基准。

中间逻辑表示分析

Metrics	Accuracy
Exact match	79.3
Structure match	89.6
Token match	96.5

Table 7: GEOQUERY evaluation of ungrounded meaning representations. We report accuracy against a manually created gold standard.

Dataset	SCANNER	Baseline
SPADES	51.2	45.5
–*conj* (1422)	56.1	66.4
–*control* (132)	28.3	40.5
–*pp* (3489)	46.2	23.1
–*subord* (76)	37.9	52.9
WEBQUESTIONS	42.1	25.5
GRAPHQUESTIONS	11.9	15.3

Table 8: Evaluation of predicates induced by SCANNER against EASYCCG. We report F1(%) across datasets. For SPADES, we also provide a breakdown for various utterance types.

conj	the boeing_company was founded in 1916 and is headquartered in __ , illinois . nstar was founded in 1886 and is based in boston , __ . the __ is owned and operated by zuffa_,_llc , headquarted in las_vegas , nevada . hugh attended __ and then shifted to uppingham_school in england .	__ was incorporated in 1947 and is based in new_york_city . the ifbb was formed in 1946 by president ben_weider and his brother __ . wilhelm_maybach and his son __ started maybach in 1909 . __ was founded in 1996 and is headquartered in chicago .
control	__ threatened to kidnap russ . __ has also been confirmed to play captain_haddock . hoffenberg decided to leave __ . __ is reportedly trying to get impregnated by djimon now . for right now , __ are inclined to trust obama to do just that .	__ agreed to purchase wachovia_corp . ceo john_thain agreed to leave __ . so nick decided to create __ . salva later went on to make the non clown-based horror __ . eddie dumped debbie to marry __ when carrie was 2 .
pp	__ is the home of the university_of_tennessee . chu is currently a physics professor at __ . youtube is based in __ , near san_francisco , california . mathematica is a product of __ .	jobs will retire from __ . the nab is a strong advocacy group in __ . this one starred robert_reed , known mostly as __ . __ is positively frightening as detective bud_white .
subord	the__ is a national testing board that is based in toronto . __ is a corporation that is wholly owned by the city_of_edmonton . unborn is a scary movie that stars __ . __ 's third wife was actress melina_mercouri , who died in 1994 . sure , there were __ who liked the shah .	founded the __ , which is now also a designated terrorist group . __ is an online bank that ebay owns . zoya_akhtar is a director , who has directed the upcoming movie __ . imelda_staunton , who plays __ , is genius . __ is the important president that american ever had . plus mitt_romney is the worst governor that __ has had .

Table 9: Informative predicates identified by SCANNER in various types of utterances. Yellow predicates were identified by both SCANNER and EASYCCG, red predicates by SCANNER alone, and green predicates by EASYCCG alone.

这里进行了一些额外的实验来检验逻辑中间表示的生成质量。

- Table 7：作者手工标注了一些逻辑中间表示，以此来观察人的介入对逻辑中间表示的生成效果的影响。三行结果分别对应了不同的匹配标准。

- Table 8：为了在其他3个数据集上也能够进行类似的实验，作者将评价方式换为对比生成的逻辑中间表示与句法分析器分析结果。作者用EasyCCG将自然语言问题句子转换为事件-参数（event-argument）的结构。分析看出对于组合性强的数据集GRAPHQUESTIONS，本文的系统性能差一些。
- Table 9：展示了一些谓词预测结果的事例，并与EasyCCG的结果相对比。对于例子

  > *ceo john_thain agreed to leave _.*

  本文的系统会难以区分控制（control）动词和真正的谓词。

附录

通用谓词

适用于GEOQUERY的通用谓词表如下：（还需要再进行扩展，才能够使用于通用领域）

```
coutryid
cityid
stateid
riverid
placeid
answer
largest
smallest
highest
lowest
longest
shortest
count
fewest
intersect
union
exclude
all
_
```

代码

parse_list

```
def parse_list(tokens):
    # expect '(' always as first token for this function
    if len(tokens) == 0 or tokens[0] != '(':
        raise SyntaxError("(parse_list) Error: expected '(' token, found
<%s>" % str(tokens))
    first = tokens.pop(0) # consume the opening '('

    # consume the operator and all operands
    operator = tokens.pop(0) # operator always after opening ( syntatically
    operands, tokens = parse_operands(tokens)
    ast = [operator]
    ast.extend(operands)

    # consume the matching ')'
    if len(tokens) == 0 or tokens[0] != ')':
        raise SyntaxError("(parse_list) Error: expected ')' token, found
<%s>: " % str(tokens))
    first = tokens.pop(0)

    return ast, tokens
```

parseo_perands

```
def parse_operands(tokens):
    operands = []
    while len(tokens) > 0:
        # peek at next token, and if not an operand then stop
        if tokens[0] == ')':
            break

        # if next token is a '(', need to get sublist/subexpression
        if tokens[0] == '(':
            subast, tokens = parse_list(tokens)
            operands.append(subast)
            continue # need to continue trying to see if more operands
after the sublist

        # otherwise token must be some sort of an operand
        operand = tokens.pop(0) # consume the token and parse it

        operands.append(decode_operand(operand))

    return operands, tokens
```

construct_from_list

```
def construct_from_list(self, content):
    '''build the tree from hierarchical list'''
    if sexp.is_string(content):
        identifier = self.node_count
        new_node =  Node(identifier, content)
        self[identifier] = new_node
        self.node_count += 1
        return identifier
    else:
        identifier = self.node_count
        new_node =  Node(identifier, content[0])
        self[identifier] = new_node
        self.node_count += 1
        for child in content[1:]:
            child_identifier = self.construct_from_list(child)
            self[identifier].add_child(child_identifier)
            return identifier
```

top_down

```
    def top_down(self, identifier, general_predicates):
        '''
        pre-order traversal, return a list of node content and a list of
transition actions
        ACT: reduce top elements on the stack into a subtree
        NT: creates a subtree root node which is to be expanded, and push
it to the stack
        TER: generates terminal node under the current subtree
        '''
        data = []
        action = []

        def recurse(identifier):
            node = self[identifier]
            if len(node.children) == 0:
                data.append(node.content)
                action.append('TER')
            else:
                data.append(node.content)
                if node.content in general_predicates:
                    action.append("{}({})".format('NT', node.content))
                else:
                    action.append('NT')
                for child in node.children:
                    recurse(child)
                action.append('ACT')

        recurse(identifier)
        return data, action
```

top_down_train

```
def top_down_train(self, words, oracle_actions, oracle_tokens, options,
buffer, stack_top, action_top):
    stack = []
    losses = []

    reducable = 0
    reduced = 0

    #recursively generate the tree until training data is exhausted
    while not (len(stack) == 1 and reduced != 0):
        valid_actions = []
        if len(stack) == 0:
            valid_actions += [self._ROOT]
        if len(stack) >= 1:
            valid_actions += [self._TER, self._NT] + self._NT_general
        if len(stack) >= 2 and reducable != 0:
```

```
            valid_actions += [self._ACT]

        action = self.act_vocab[oracle_actions.pop(0)] # Get the action
identifier.

        word_weights = None

        #we make predictions when stack is not empty and _ACT is not the
only valid action
        if len(stack) > 0 and valid_actions[0] != self._ACT:
            stack_embedding = stack[-1][0].output() # stack[-1] means the
last one in the list 'stack'.
            action_summary = action_top.output()
            word_weights = self.attention(stack_embedding, buffer) # Get
the attention weight list of the encoding words.
            buffer_embedding, _ = attention_output(buffer, word_weights,
'soft_average')
            # Get the attention-weighted word embedding vectors (a vector).

            for i in range(len(stack)):
                if stack[len(stack)-1-i][1] == 'p':
                    parent_embedding = stack[len(stack)-1-i][2]
                    # parent_embedding is just the embeding vector computed
from
                    # the word vector into a linear layer Wx+b.
                    # Just the word vector without processing
                    #  into the hidden state.
                    break
            parser_state = dy.concatenate([buffer_embedding,
stack_embedding, parent_embedding, action_summary])
            h = self.mlp_layer(parser_state)

            if options.dropout > 0:
                h = dy.dropout(h, options.dropout)

            if len(valid_actions) > 0:
                log_probs = dy.log_softmax(self.act_proj_layer(h),
valid_actions)
                assert action in valid_actions, "action not in scope"
                losses.append(-dy.pick(log_probs, action))
                # Append the corresponding log probability of the action
                # annotated by the training examples.

        # Carrying on dealing with the action from the training example.
        if action == self._NT:
            # If the aciton is NT, then caculate the log probability of
            # the corresponding annotated term choices.
            #generate non-terminal
            nt = self.nt_vocab[oracle_tokens.pop(0)]
```

```
            #no need to predict the ROOT (assumed ROOT is fixed)
            if word_weights is not None:
                train_selection = self.train_selection if
np.random.randint(10) > 5 else "soft_average"
                output_feature, output_logprob = attention_output(buffer,
word_weights, train_selection)
                log_probs_nt =
dy.log_softmax(self.nt_proj_layer(output_feature))
                losses.append(-dy.pick(log_probs_nt, nt))
                # Append the corresponding log probability of the NT
                # annotated by the training examples.

                if train_selection == "hard_sample":
                    baseline_feature, _ = attention_output(buffer,
word_weights, "soft_average")
                    log_probs_baseline =
dy.log_softmax(self.nt_proj_layer(baseline_feature))
                    r = dy.nobackprop(dy.pick(log_probs_nt, nt) -
dy.pick(log_probs_baseline, nt))
                    losses.append(-output_logprob * dy.rectify(r))

            stack_state, label, _ = stack[-1] if stack else (stack_top,
'ROOT', stack_top)
            nt_embedding = self.nt_input_layer(self.nt_lookup[nt])
            stack_state = stack_state.add_input(nt_embedding)
            stack.append((stack_state, 'p', nt_embedding)) # 'p' for
parent.

        elif action in self._NT_general:
            nt = self.nt_vocab[oracle_tokens.pop(0)]
            stack_state, label, _ = stack[-1] if stack else (stack_top,
'ROOT', stack_top)
            nt_embedding = self.nt_input_layer(self.nt_lookup[nt])
            stack_state = stack_state.add_input(nt_embedding)
            stack.append((stack_state, 'p', nt_embedding))

        elif action == self._TER:
            #generate terminal
            ter = self.ter_vocab[oracle_tokens.pop(0)]
            output_feature, output_logprob = attention_output(buffer,
word_weights, self.train_selection)
            log_probs_ter =
dy.log_softmax(self.ter_proj_layer(output_feature))
            losses.append(-dy.pick(log_probs_ter, ter))

            if self.train_selection == "hard_sample":
                baseline_feature, _ = attention_output(buffer,
word_weights, "soft_average")
```

```
                #baseline_feature, _ = attention_output(buffer,
word_weights, self.train_selection, argmax=True)
                log_probs_baseline =
dy.log_softmax(self.ter_proj_layer(baseline_feature))
                r = dy.nobackprop(dy.pick(log_probs_ter, ter) -
dy.pick(log_probs_baseline, ter))
                losses.append(-output_logprob * dy.rectify(r))

            stack_state, label, _ = stack[-1] if stack else (stack_top,
'ROOT', stack_top)
            ter_embedding = self.ter_input_layer(self.ter_lookup[ter])
            stack_state = stack_state.add_input(ter_embedding)
            stack.append((stack_state, 'c', ter_embedding)) # 'c' for
child.

        else:
            # Into this branch means having met a reduce(ACT) action.
            #subtree completion
            found_p = 0
            path_input = []
            #keep popping until the parent is found
            while found_p != 1:
                top = stack.pop()
                top_raw_rep, top_label, top_rep = top[2], top[1], top[0]
                # Raw representation, label('c' or 'p') and representation
respectively.
                path_input.append(top_raw_rep)
                if top_label == 'p':
                    found_p = 1
            parent_rep = path_input.pop()
            composed_rep =
self.subtree_input_layer(dy.concatenate([dy.average(path_input),
parent_rep]))

            stack_state, _, _ = stack[-1] if stack else (stack_top, 'ROOT',
stack_top)
            stack_state = stack_state.add_input(composed_rep)
            stack.append((stack_state, 'c', composed_rep))
            reduced = 1

        action_embedding = self.act_input_layer(self.act_lookup[action])
        action_top = action_top.add_input(action_embedding)

        reducable = 1

        #cannot reduce after an NT
        if stack[-1][1] == 'p':
            reducable = 0
```

```
    return dy.esum(losses)
```