Supplementary Material: Neural Shift-Reduce CCG Semantic Parsing

Dipendra K. Misra and **Yoav Artzi**
Department of Computer Science and Cornell Tech
Cornell University
New York, NY 10011
{dkm,yoav}@cs.cornell.edu

1 Number of Operations in Shift-Reduce and CKY CCG Parsers

Let Λ be a CCG lexicon, $\mathcal{R}_b$ the set of binary CCG rules, and $\mathcal{R}_u$ the set of unary CCG rules. While in practice lexical entries may map phrases to categories, for simplification, we assume that each lexical entry contains only one token.[1] Let $|\lambda|$ be the number of lexical entries for a token in Λ. We assume an input sentence x with m tokens. We define an operation in shift-reduce parsing to be the application of a single action to a configuration. In CKY, an operation is an application of a unary rule to a cell in the chart, or a binary rule to a pair of adjacent cells.

CKY CCG Semantic Parser CKY parsing starts with populating the chart using the lexicon Λ. Under our single-token assumption, this requires at most $m|\lambda|$ operations. In practice, though, the number of categories maintained in each cell is capped by a beam of size k. We denote a cell that spans the token sequence $\langle x_i, \ldots, x_j \rangle$ as $[i, j]$. Given the cell $[i, j]$, $j > i$, CKY considers all possible splits $\{\langle [i, l], [l+1, j] \rangle \mid i \leq l \leq j\}$ of this cell and applies binary rules $b \in \mathcal{R}_b$ to the categories in the cells $[i, l]$ and $[l+1, j]$. This requires $mk^2|\mathcal{R}_b|$ operations due to $O(m)$ possible splits, k^2 possible categories from the beams of the two cells, and $|\mathcal{R}_b|$ binary rules. There is a total of m^2 cells. Therefore, the total number of operations for binary rules is $m^3k^2|\mathcal{R}_b|$. For every cell, we can also apply a unary rule from $\mathcal{R}_u$. The overall number of unary operations is $m^2k|\mathcal{R}_u|$. The total number of operations is $O(m|\lambda| + m^3k^2|\mathcal{R}_b| + m^2k|\mathcal{R}_u|)$.

Shift-Reduce CCG Semantic Parser The shift-reduce parser also uses a beam of size k. The beam maintains the k max-scoring configurations. At each step, it applies all possible actions to each configuration in the beam to generate a new configuration. The top-k new configurations are then retained in the beam. We can perform shift for each token on the buffer, which give m operations. Since binary reduce removes an element from the stack, we can do at most $m - 1$ such operations. We disallow two consecutive unary reduce actions. Therefore, unary reduce actions must follow a shift or a binary reduce, which translates to at most $m - 1 + m = 2m - 1$ operations. Therefore, the parser necessarily terminates after at most $4m - 2$ beam expansions. For a given configuration, we can apply $|\lambda| + |\mathcal{R}_b| + |\mathcal{R}_u|$ actions. In every step of the algorithm there are at most k configurations to process, giving a total of $O(4mk(|\lambda| + |\mathcal{R}_b| + |\mathcal{R}_u|)$ operations

Quantitative Comparison In our experiments, the lexicon Λ contains 1.7M entries for 11K words and phrases. If we define $|\lambda|$ to be the mean number of entries, we get $|\lambda| = 170$. The average sentence length m in the data is 25. Our CCG has 30 binary rules ($\mathcal{R}_b$) and 24 unary rules ($\mathcal{R}_u$). Artzi et al. (2015) use a beam size of 50 in their CKY parser, which gives roughly 10^9 operations per sentence of length 25. For our final results, we use a beam of 512, which gives roughly 10^7 operations for the same length, two orders of magnitude fewer.

[1] Generalization to multiple tokens is straightforward.

Feature Type	Dimension	Description
RULE-NAME	16	Action name
POS	12	POS tags of all tokens removed from the buffer in a SHIFT operation
TEMPLATE∧RPOS	32	Template and POS tag of the first token on the buffer following a SHIFT (not triggered for reduce operations)
TEMPLATE∧LPOS	32	Template and POS tag of the last token consumed before a SHIFT (not triggered for reduce operations)
NEXT-POS1	12	POS of the first token on the buffer after an action
NEXT-POS2	12	POS of the second token on the buffer after an action
PREV-POS1	12	POS of the recently consumed token before an action
PREV-POS2	12	POS of the second recently consumed token before an action
Features from Artzi et al. (2015)		
LEX-TEMPLATE	48	Triggers four features on SHIFT operations: Lexeme of the lexical entry Template of the lexical entry Conjunction of lexeme and template Conjunction of template and POS of the lexical entry tokens
TYPESHIFTSEM	32	Conjunction of a CCG type-shifting unary rule and the head predicate of the logical form
ATTRIBUTE∧POS	32	Conjunction of attributes used in the lexical entry and token POS tags
DYN	8	Using a lexical entry dynamically generated (e.g., NER)
DYNSKIP	8	Skipping a word
LOGEXP	8	Repeating conjuncts in the root logical form
SLOPPYLEX	16	Using a lexical entry dynamically created with sloppy heuristics
TYPESHIFT	16	Using a unary type-shifting rule
CROSS	16	Using a crossing composition binary rule
ATTACH	32	Entity-relation-entity logical form attachment features

Table 1: Sparse features used for action embedding

2 Action Features

Table 1 lists the features used to compute action embeddings $\phi(a, c)$. Each feature is mapped to its embedding representation via a lookup table. The embeddings are then concatenated to create the action representation. We use a factored lexicon representation (Kwiatkowski et al., 2011), where entries are dynamically generated by combining *lexemes* and *templates*. For example, the lexical entry: $remain \vdash S\backslash NP_{[pl]}/(N_{[pl]}/N_{[pl]}) : \lambda f.\lambda x.f(\lambda r.\text{remain-01}(r) \wedge \text{ARG1}(r, x))$ is generated from the lexeme $\langle remain, \{\text{remain-01}\}\rangle$ and the template $\lambda v_1.[S\backslash NP_{[pl]}/(N_{[pl]}/N_{[pl]}) : \lambda f.\lambda x.f(\lambda r.v_1(r) \wedge \text{ARG1}(r, x))]$. Feature type dimensionality was selected based on the possible number of features for the type. For example, there are many more lexemes than part-of-speech tags, requiring a relatively higher dimensionality for lexeme features. If more than one feature is active for a given feature type, we average the embeddings in the action representation. Additionally, we learn *inactive* embedding for every feature type, which is used when there are no active features of this type.

3 Embedding logical forms

Given a logical form z, its embedded representation is computed by the recursive function $\psi(z)$. We use simply-typed lambda calculus logical forms. A logical form is defined with four base cases:

- Constant c
- Variable v
- Literal $p(z_1, \dots, z_k)$, where the predicate p is a logical form and the arguments $z_1, \dots, z_k$ are logical forms
- Lambda term $\lambda v.z_1$, where v is a variable and the body z_1 is a logical form

Each logical form is typed. The function $\psi(z)$ follows these base cases to compute the embedding of z. Algorithm 1 describes $\psi(z)$. The recursive combination is achieved with a single-layer neural network parameterized by W_r, δ_r, and the $\tanh$ activation function. The embedding of a constant c is a combination of its name and type embeddings, each derived from a lookup table (line 2). Given a

Algorithm 1 ψ: Embeds a typed lambda calculus expression.

Input: A logical expression or a list of expressions e, embedding lookup tables U and V for logical constants and types.

Definitions: $[;]$ represents concatenation. $\mathbf{W}_r$ is a $M_r \times 2M_r$ matrix and $\delta_r \in \mathbb{R}^{M_r}$ is a bias term. We use c, v, and z for constant, variable, and generic logical form.

Output: Embedding $\nu \in R^{M_r}$

1: CASE e:
2: $c : \tanh(\mathbf{W}_r([U[c.name]; V[c.type]]) + \delta_r)$
3: $v : V[v.type]$
4: $p\,[z_1 \cdots z_k] : \tanh(\mathbf{W}_r[\psi(p); \psi([z_1 \cdots z_k])] + \delta_r)$
5: $[z_1 \cdots z_k] : \tanh(\mathbf{W}_r[\psi(z_1); \psi([z_2 \cdots z_k])] + \delta_r)$
6: $\lambda v.\, z : \tanh(\mathbf{W}_r[\psi(v); \psi(z)] + \delta_r)$

variable v, its embedding is given via a lookup table V indexed by variable types (line 3). Literals are embedded by recursively embedding their arguments and combining with the predicate embedding (lines 4-5). Finally, for lambda terms, the variable embedding is combined with the body embedding (line 6). The logical form embedding size M_r is 35. All parameters ($\mathbf{W}_r$, δ_r, and all lookup embeddings) are initialized using the Glorot and Bengio (2010) scheme, similar to the other parameters in the shift-reduce parser.

4 Word Skipping

Since word skipping is never selected during training, the model learns to discourage it. Therefore, we define the term $\epsilon(a)$, where $\epsilon(a) = \gamma$ if the action is a SHIFT that skips the next word, otherwise $\epsilon(a) = 0$. In practice, this is accomplished by adding special lexical entries to the lexicon that mark skipping. The probability of action a given configuration c then incorporates the term $\epsilon(a)$:

$$p(a \mid c) = \frac{\exp\{\phi(a,c)\mathbf{W}_b\mathcal{F}(\xi(c)) + \epsilon(a)\}}{\sum_{a' \in \mathcal{A}(c)} \exp\{\phi(a',c)\mathbf{W}_b\mathcal{F}(\xi(c)) + \epsilon(a')\}} \ .$$

We tune γ on a small subset of the development data and set it to $\gamma = 1.0$.

References

Artzi, Y., Lee, K., and Zettlemoyer, L. (2015). Broad-coverage CCG semantic parsing with AMR. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *International Conference on Artificial Intelligence and Statistics*.

Kwiatkowski, T., Zettlemoyer, L. S., Goldwater, S., and Steedman, M. (2011). Lexical generalization in CCG grammar induction for semantic parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.