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

Abstract

We present a shift-reduce CCG semantic parser. Our parser uses a neural network architecture that balances model capacity and computational cost. We train by transferring a model from a computationally expensive log-linear CKY parser. Our learner addresses two challenges: selecting the best parse for learning when the CKY parser generates multiple correct trees, and learning from partial derivations when the CKY parser fails to parse. We evaluate on AMR parsing. Our parser performs comparably to the CKY parser, while doing significantly fewer operations. We also present results for greedy semantic parsing with a relatively small drop in performance.

1 Introduction

Shift-reduce parsing is a class of parsing methods that guarantees a linear number of operations in sentence length. This is a desired property for practical applications that require processing large amounts of text or real-time response. Recently, such techniques were used to build state-of-the-art syntactic parsers, and have demonstrated the effectiveness of deep neural architectures for decision making in linear-time dependency parsing (Chen and Manning, 2014; Dyer et al., 2015; Andor et al., 2016; Kiperwasser and Goldberg, 2016). In contrast, semantic parsing often relies on algorithms with polynomial number of operations, which results in slow parsing times unsuitable for practical applications. In this paper, we apply shift-reduce parsing to semantic parsing. Specifically, we study transferring a learned Combinatory Categorial Grammar (CCG; Steedman, 1996, 2000) from a dynamic-programming CKY model to a shift-reduce neural network architecture.

We focus on the feed-forward architecture of Chen and Manning (2014), where each parsing step is a multi-class classification problem. The state of the parser is represented using simple feature embeddings that are passed through a multilayer perceptron to select the next action. While simple, the capacity of this model to capture interactions between primitive features, instead of relying on sparse complex features, has led to new state-of-the-art performance (Andor et al., 2016). However, applying this architecture to semantic parsing presents learning and inference challenges.

In contrast to dependency parsing, semantic parsing corpora include sentences labeled with the system response or the target formal representation, and omit derivation information. CCG induction from such data relies on latent-variable techniques and requires careful initialization (e.g., Zettlemoyer and Collins, 2005, 2007). Such feature initialization does not directly transfer to a neural network architecture with dense embeddings, and the use of hidden layers further complicates learning by adding a large number of latent variables. We focus on data that includes sentence-representation pairs, and learn from a previously induced log-linear CKY parser. This drastically simplifies learning, and can be viewed as bootstrapping a fast parser from a slow one. While this dramatically narrows down the number of parses per sentence, it does not eliminate ambiguity. In our experiments, we often get multiple correct parses, up to 49K in some cases. We also observe that the CKY parser generates no parses for

Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, pages 1775–1786,
Austin, Texas, November 1-5, 2016.

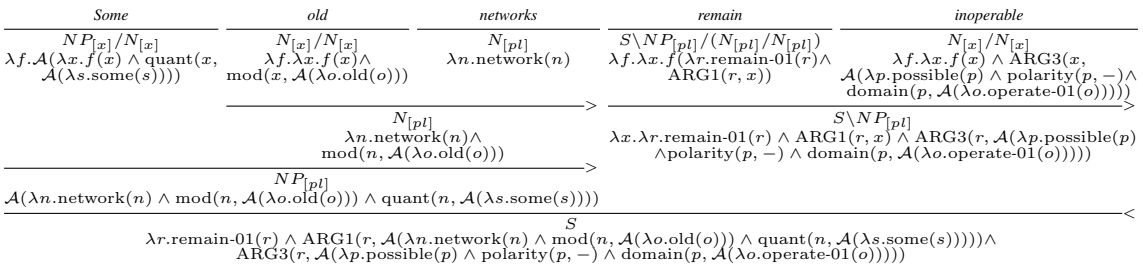

Figure 1: Example CCG tree with five lexical entries, three forward applications (>) and a backward application (<).

a significant number of training sentences. Therefore, we propose an iterative algorithm that automatically selects the best parses for training at each iteration, and identifies partial derivations for best-effort learning, if no parses are available.

CCG parsing largely relies on two types of actions: using a lexicon to map words to their categories, and combining categories to acquire the categories of larger phrases. In most semantic parsing approaches, the number of operations is dominated by the large number of categories available for each word in the lexicon. For example, the lexicon in our experiments includes 1.7M entries, resulting in an average of 146, and up to 2K, applicable actions. Additionally, both operations and parser state have complex structures, for example including both syntactic and semantic information. Therefore, unlike in dependency parsing (Chen and Manning, 2014), we can not treat action selection as multi-class classification, and must design an architecture that can accommodate a varying number of actions. We present a network architecture that considers a variable number of actions, and emphasizes low computational overhead per action, instead focusing computation on representing the parser state.

We evaluate on Abstract Meaning Representation (AMR; Banarescu et al., 2013) parsing. We demonstrate that our modeling and learning contributions are crucial to effectively commit to early decisions during parsing. Somewhat surprisingly, our shift-reduce parser provides equivalent performance to the CKY parser used to generate the training data, despite requiring significantly fewer operations, on average two orders of magnitude less. Similar to previous work, we use beam search, but also, for the first time, report greedy CCG semantic parsing results at a relatively modest 9% decrease in performance, while the source CKY parser with a beam of one demonstrates a 71% decrease. While we focus on semantic parsing, our learning approach makes no task-specific assumptions and has potential for learning efficient models for structured prediction from the output of more expensive ones.[1]

2 Task and Background

Our goal is to learn a function that, given a sentence x, maps it to a formal representation of its meaning z with a linear number of operations in the length of x. We assume access to a training set of N examples $\mathcal{D} = \{(x^{(i)}, z^{(i)})\}_{i=1}^{N}$, each containing a sentence $x^{(i)}$ and a logical form $z^{(i)}$. Since $\mathcal{D}$ does not contain complete derivations, we instead assume access to a CKY parser learned from the same data. We evaluate performance on a test set $\{(x^{(i)}, z^{(i)})\}_{i=1}^{M}$ of M sentences $x^{(i)}$ labeled with logical forms $z^{(i)}$. While we describe our approach in general terms, we apply our approach to AMR parsing and evaluate on a common benchmark (Section 6).

To map sentences to logical forms, we use CCG, a linguistically-motivated grammar formalism for modeling a wide-range of syntactic and semantic phenomena (Steedman, 1996, 2000). A CCG is defined by a lexicon Λ and sets of unary $\mathcal{R}_u$ and binary $\mathcal{R}_b$ rules. In CCG parse trees, each node is a category. Figure 1 shows a CCG tree for the sentence *Some old networks remain inoperable*. For example, $S\backslash NP_{[pl]}/(N_{[pl]}/N_{[pl]})$: $\lambda f.\lambda x.f(\lambda r.\text{remain-01}(r) \wedge \text{ARG1}(r, x))$ is the category of the verb *remain*. The syntactic type $S\backslash NP_{[pl]}/(N_{[pl]}/N_{[pl]})$ indicates that two arguments are expected: first an adjective $N_{[pl]}/N_{[pl]}$ and then a plural noun phrase $NP_{[pl]}$. The final syntactic type will be S. The forward slash / indicates the argument is expected on the right, and the backward slash \ indicates it is expected on the left. The syntactic attribute pl is used to express the plural-

[1]The source code and pre-trained models are available at `http://www.cs.cornell.edu/~dkm/ccgparser`.

ity constraint of the verb. The simply-typed lambda calculus logical form in the category represents semantic meaning. The typing system includes atomic types (e.g., entity e, truth value t) and functional types (e.g., $\langle e, t\rangle$ is the type of a function from e to t). In the example category above, the expression on the right of the colon is a $\langle\langle\langle e,t\rangle,\langle e,t\rangle\rangle,\langle e,\langle e,t\rangle\rangle\rangle$-typed function expecting first an adjectival modifier and then an ARG1 modifier. The conjunction $\wedge$ specifies the roles of remain-01. The lexicon Λ maps words to CCG categories. For example, the lexical entry $remain \vdash S\backslash NP_{[pl]}/(N_{[pl]}/N_{[pl]}) : \lambda f.\lambda x.f(\lambda r.\text{remain-01}(r) \wedge \text{ARG1}(r,x))$ pairs the example category with *remain*. The parse tree in the figure includes four binary operations: three forward applications ($>$) and a backward application ($<$).

3 Neural Shift Reduce Semantic Parsing

Given a sentence $x = \langle x_1, \ldots, x_m\rangle$ with m tokens x_i and a CCG lexicon Λ, let $\textsc{Gen}(x;\Lambda)$ be a function that generates CCG parse trees. We design GEN as a shift-reduce parser, and score decisions using embeddings of parser states and candidate actions.

3.1 Shift-Reduce Parsing for CCG

Shift-reduce parsers perform a single pass of the sentence from left to right to construct a parse tree. The parser configuration[2] is defined with a stack and a buffer. The stack contains partial parse trees, and the buffer the remainder of the sentence to be processed. Formally, a parser configuration c is a tuple $\langle\sigma,\beta\rangle$, where the stack σ is a list of CCG trees $[s_l \cdots s_1]$, and the buffer β is a list of tokens from x to be processed $[x_i \cdots x_m]$.[3] For example, the top-left of Figure 2 shows a parsing configuration with two partial trees on the stack and two words on the buffer (*remain* and *inoperable*).

Parsing starts with the configuration $\langle[], [x_1 \cdots x_m]\rangle$, where the stack is empty and the buffer is initialized with x. In each parsing step, the parser either consumes a word from the buffer and pushes a new tree to the stack, or applies a parsing rule to the trees at the top of the stack. For simplicity, we apply CCG rules to trees, where a rule is applied to the root categories of the argument trees to create a new tree with the arguments as children. We treat lexical entries as trees with a single node. There are three types of actions:[4]

$$\begin{aligned}
\textsc{Shift}(l, \langle\sigma, x_i|\cdots|x_j|\beta\rangle) &= \langle\sigma|g,\beta\rangle \\
\textsc{Binary}(b, \langle\sigma|s_2|s_1,\beta\rangle) &= \langle\sigma|b(s_2,s_1),\beta\rangle \\
\textsc{Unary}(u, \langle\sigma|s_1,\beta\rangle) &= \langle\sigma|u(s_1),\beta\rangle \quad .
\end{aligned}$$

Where $b \in \mathcal{R}_b$ is a binary rule, $u \in \mathcal{R}_u$ is a unary rule, and l is a lexical entry $x_i, \ldots x_j \vdash g$ for the tokens x_i,...,x_j and CCG category g. SHIFT creates a tree given a lexical entry for the words at the top of the buffer, BINARY applies a binary rule to the two trees at the head of the stack, and UNARY applies a unary rule to the tree at head of the stack. A configuration is terminal when no action is applicable.

Given a sentence x, a derivation is a sequence of action-configuration pairs $\langle\langle c_1, a_1\rangle, \ldots, \langle c_k, a_k\rangle\rangle$, where action a_i is applied to configuration c_i to generate configuration c_{i+1}. The result configuration c_{k+1} is of the form $\langle[s],[]\rangle$, where s represents a complete parse tree, and the logical form z at the root category represents the meaning of the complete sentence. Following previous work with CKY parsing (Zettlemoyer and Collins, 2005), we disallow consecutive unary actions. We denote the set of actions allowed from configuration c as $\mathcal{A}(c)$.

3.2 Model

Our goal is to balance computation and model capacity. To recover a rich representation of the configuration, we use a multilayer perceptron (MLP) to create expressive interactions between a small number of simple features. However, since we consider many possible actions in each step, computing activations for multiple hidden layers for each action is prohibitively expensive. Instead, we opt for a computationally-inexpensive action representation computed by concatenating feature embeddings. Figure 2 illustrates our architecture.

Given a configuration c, the probability of an action a is:

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

[2] We use the terms *parser configuration* and *parser state* interchangeably.

[3] The head of the stack σ is the right-most entry, and the head of the buffer β is the left-most entry.

[4] We follow the standard notation of $L|x$ indicating a list with all the entries from L and x as the right-most element.

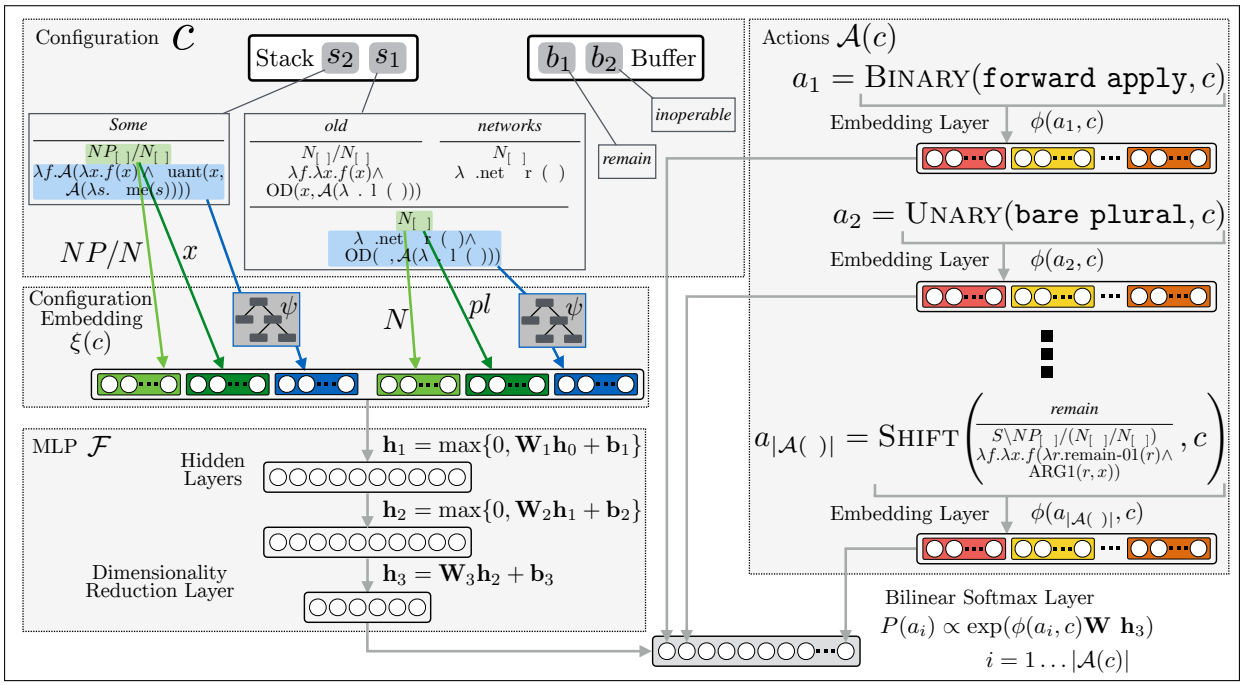

Figure 2: Illustration of scoring the next action given the configuration c when parsing the sentence *Some old networks remain inoperable*. Embeddings of the same feature type are colored the same. The configuration embedding $\xi(c)$ is a concatenation of syntax embeddings (green) and the logical form embedding (blue; computed by ψ) for the top entries in the stack. We then pass $\xi(c)$ through the MLP $\mathcal{F}$. Given the actions $\mathcal{A}(c)$, we compute the embeddings $\phi(a_i, c)$, $i = 1 \ldots |\mathcal{A}(c)|$. The actions and MLP representation are combined with a bilinear softmax layer. The number of concatenated vectors and stack elements used is for illustration. The details are described in Section 3.2.

where $\phi(a, c)$ is the action embedding, $\xi(c)$ is the configuration embedding, and $\mathcal{F}$ is an MLP. $\mathbf{W}_b$ is a bilinear transformation matrix. Given a sentence x and a sequence of action-configuration pairs $\langle\langle c_1, a_1\rangle, \ldots, \langle c_k, a_k\rangle\rangle$, the probability of a CCG tree y is

$$p(y \mid x) = \prod_{i=1\ldots k} p(a_i \mid c_i) \ .$$

The probability of a logical form z is then

$$p(z \mid x) = \sum_{y \in \mathcal{Y}(z)} p(y \mid x) \ ,$$

where $\mathcal{Y}(z)$ is the set of CCG trees with the logical form z at the root.

MLP Architecture $\mathcal{F}$ We use a MLP with two hidden layers parameterized by $\{\mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2\}$ with a ReLu non-linearity (Glorot et al., 2011). Since the output of $\mathcal{F}$ influences the dimensionality of $\mathbf{W}_b$, we add a linear layer parameterized by $\mathbf{W}_3$ and $\mathbf{b}_3$ to reduce the dimensionality of the configuration, thereby reducing the dimensionality of $\mathbf{W}_b$.

Configuration Embedding $\xi(c)$ Given a configuration $c = \langle[s_l \cdots s_1], [x_i \cdots x_m]\rangle$, the input to $\mathcal{F}$ is a concatenation of syntactic and semantic embeddings, as illustrated in Figure 2. We concatenate embeddings from the top three trees in the stack s_1, s_2, s_3.[5] When a feature is not present, for example when the stack or buffer are too small, we use a tunable null embedding.

Given a tree on the stack s_j, we define two syntactic features: *attribute set* and *stripped syntax*. The attribute feature is created by extracting all the syntactic attributes of the root category of s_j. The stripped syntax feature is the syntax of the root category without the syntactic attributes. For example, in Figure 2, we embed the stripped category N and attribute pl for s_1, and NP/N and x for s_2. The attributes are separated from the syntax to reduce sparsity, and the interaction between them is computed by $\mathcal{F}$. The sparse features are converted to dense embeddings using a lookup table and concatenated. In addition, we also embed the logical form at the root of s_j. Figure 3 illustrates the recursive embedding function ψ.[6] Using a recursive function to embed logical forms is computationally intensive. Due to strong correlation between sentence length and logical form complexity, this computation increases

[5] For simplicity, the figure shows only the top two trees.

[6] The algorithm is provided in the supplementary material.

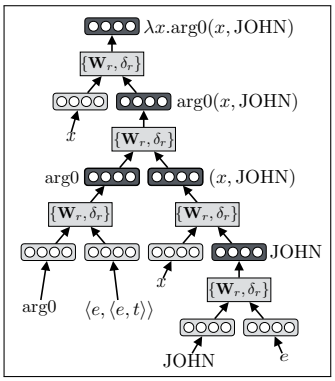

Figure 3: Illustration of embedding the logical form $\lambda x.\text{arg0}(x, \text{JOHN})$ with the recursive embedding function ψ. In each level in ψ, the children nodes are combined with a single-layer neural network parameterized by $\mathbf{W}_r$, δ_r, and the $\tanh$ activation function. Computed embeddings are in dark gray, and embeddings from lookup tables are in light gray. Constants are embedded by combining name and type embeddings, literals are unrolled to binary recursive structures, and lambda terms are combinations of variable type and body embeddings. For example, JOHN is embedded by combining the embeddings of its name and type, the literal $\text{arg0}(x, \text{JOHN})$ is recursively embedded by first embedding the arguments (x, JOHN) and then combining the predicate, and the lambda term is embedded to create the embedding of the entire logical form.

the cost of configuration embedding by a factor linear in sentence length. In Section 6, we experiment with including this option, balancing between potential expressivity and speed.

Action Embedding $\phi(a, c)$ Given an action $a \in \mathcal{A}(c)$, and the configuration c, we generate the action representation by computing sparse features, converting them to dense embeddings via table lookup, and concatenating. If more than one feature of the same type is triggered, we average their embeddings. When no features of a given type are triggered, we use a tunable placeholder embedding instead. The features include all the features used by Artzi et al. (2015), including all conjunctive features, as well as properties of the action and configuration, such as the POS tags of tokens on the buffer.[7]

Discussion Our use of an MLP is inspired by Chen and Manning (2014). However, their architecture is designed to handle only a fixed number of actions, while we observe varying number of actions. Therefore, we adopt a probabilistic model similar to Dyer et al. (2015) to effectively combine the benefits of the two approaches.[8] We factorize the exponent in our objective into action $\phi(a, c)$ and configuration $\mathcal{F}(\xi(c))$ embeddings. While every parse step involves a single configuration, the number of actions is significantly higher. With the goal of minimizing the amount of computation per action, we use simple concatenation only for action embedding. However, this requires retaining sparse conjunctive action features since they are never combined through hidden layers similar to configuration features.

3.3 Inference

To compute the set of parse trees $\textsc{Gen}(x; \Lambda)$, we perform beam search to recover the top-k parses. The beam contains configurations. At each step, we expand all configurations with all actions, and keep only the top-k new configurations. To promote diversity in the beam, given two configurations with the same *signature*, we keep only the highest scoring one. The signature includes the previous configuration in the derivation, the state of the buffer, and the root categories of all stack elements. Since all features are computed from these elements, this optimization does not affect the max-scoring tree. Additionally, since words are assigned structured categories, a key problem is unknown words or word uses. Following Zettlemoyer and Collins (2007), we use a two-pass parsing strategy, and allow skipping words controlled by the term γ in the second pass. The term γ is added to the exponent of the action probability when words are skipped. See the supplementary material for the exact form.

Complexity Analysis The shift-reduce parser processes the sentence from left to right with a linear number of operations in sentence length. We define an operation as applying an action to a configuration. Formally, the number of operations for a sentence of length m is bounded by $O(4mk(|\lambda| + |\mathcal{R}_b| + |\mathcal{R}_u|))$, where $|\lambda|$ is the number of lexical entries per token, k is the beam size, $\mathcal{R}_b$ is the set of binary rules, and $\mathcal{R}_u$ the set of unary rules. In comparison, the number of operations for the CKY parser, where an operation is applying a rule to a single cell or two adjacent cells in the chart, is bounded by $O(m|\lambda| + m^3k^2|\mathcal{R}_b| + m^2b|\mathcal{R}_u|)$. For sentence

[7] See the supplementary material for feature details.

[8] We experimented with an LSTM parser similar to Dyer et al. (2015). However, performance was not competitive. This direction remains an important avenue for future work.

length 25, the mean in our experiments, the shift-reduce parser performs 100 time fewer operations. See the supplementary material for the full analysis.

4 Learning

We assume access to a training set of N examples $\mathcal{D} = \{(x^{(i)}, z^{(i)})\}_{i=1}^{N}$, each containing a sentence $x^{(i)}$ and a logical form $z^{(i)}$. The data does not include information about the lexical entries and CCG parsing operations required to construct the correct derivations. We bootstrap this information from a learned parser. In our experiments we use a learned dynamic-programming CKY parser. We transfer the lexicon Λ directly from the input parser, and focus on estimating the parameters θ, which include feature embeddings, hidden layer matrices, and bias terms. The main challenge is learning from the noisy supervision provided by the input parser. In our experiments, the CKY parser fails to correctly parse 40% of the training data, and returns on average 147 max-scoring correct derivations for the rest. We propose an iterative algorithm that treats the choice between multiple parse trees as latent, and effectively learns from partial analysis when no correct derivation is available.

The learning algorithm (Algorithm 1) starts by processing the data using the CKY parser (lines 3 - 4). For each sentence $x^{(i)}$, we collect the max-scoring CCG trees with $z^{(i)}$ at the root. The CKY parser often contains many correct parses with identical scores, up to 49K parses per sentence. Therefore, we randomly sample and keep up to 1K trees. This process is done once, and the algorithm then runs for T iterations. At each iteration, given the sets of parses from the CKY parser $\mathcal{Y}$, we select the max-probability parse according to our current parameters θ (line 10) and add all the shift-reduce decisions from this parse to $\mathcal{D}_A$ (line 12), the action data set that we use to estimate the parameters. We approximate the $\arg\max$ with beam search using an oracle computed from the CKY parses.[9] CONFGEN aggregates the configuration-action pairs from the highest scoring derivation. Parse selection depends on θ and this choice will gradually converge as the parameters improve. The action data set is used to compute the ℓ_2-regularized negative log-likelihood objective

[9] Our oracle is non-deterministic and incomplete (Goldberg and Nivre, 2013).

Algorithm 1 The learning algorithm.

Input: Training set $\mathcal{D} = \{(x^{(i)}, z^{(i)})\}_{i=1}^{N}$, learning rate μ, regularization parameter ℓ_2, and number of iterations T.

Definitions: GENMAXCKY(x, z) returns the set of max-scoring CKY parses for x with z at the root. SCORE(y, θ) scores a tree y according to the parameters θ (Section 3.2). CONFGEN(x, y) is the sequence of action-configuration pairs that generates y given x (Section 3.1). BP$(\Delta_{\mathcal{J}})$ takes the objective $\mathcal{J}$ and back-propagates the error $\nabla\mathcal{J}$ through the computation graph for the sample used to compute the objective. ADAGRAD(Δ) applies a per-feature learning rate to the gradient Δ (Duchi et al., 2011).

Output: Model parameters θ.

```
1:  » Get trees from CKY parser.
2:  Y ← []
3:  for i = 1 to N do
4:      Y[i] = GENMAXCKY(x^(i), z^(i))
5:  for t = 1 to T do
6:      » Pick max-scoring trees and create action dataset.
7:      D_A = ∅
8:      for i = 1 to N do
9:          if Y[i] ≠ ∅ then
10:             A ← CONFGEN(x^(i),
11:                     arg max_{y∈Y[i]} SCORE(y, θ))
12:             for ⟨c, a⟩ ∈ A do
13:                 D_A ← D_A ∪ {⟨c, a⟩}
14:     » Back-propagate the loss through the network.
15:     for ⟨c, a⟩ ∈ D_A do
16:         J =def − log p(a | c) + (ℓ2/2) θ^T θ
17:         Δ ← BP(∇J)
18:         θ ← θ − μ ADAGRAD(Δ)
19: return θ
```

$\mathcal{J}$ (line 16) and back-propagate the error to compute the gradient (line 17). We use AdaGrad (Duchi et al., 2011) to update the parameters θ (line 18).

4.1 Learning from Partial Derivations

The input parser often fails to generate correct parses. In our experiments, this occurs for 40% of the training data. In such cases, we can obtain a forest of partial parse trees $\mathcal{Y}_p$. Each partial tree $y \in \mathcal{Y}_p$ corresponds to a span of tokens in the sentence and is scored by the input parser. In practice, the spans are often overlapping. Our goal is to generate high quality configuration-action pairs $\langle c, a \rangle$ from $\mathcal{Y}_p$. These pairs will be added to $\mathcal{D}_A$ for training. While extracting actions a is straightforward, generating configurations c requires reconstructing the stack σ from an incomplete forest of partial trees $\mathcal{Y}_p$. Figure 4 illustrates our proposed process. Let CKYSCORE(y) be the CKY score of the partial tree y. To reconstruct σ, we select non-overlapping par-

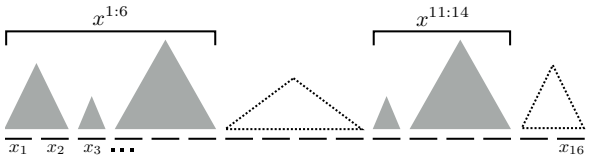

Figure 4: Partial derivation selection for learning (Section 4.1). The dotted triangles represent skipped spans in the sentence, where no high quality partial trees were found. Dark triangles represent the selected partial trees. We identify two contiguous spans, 1-6 and 11-14, and generate two synthetic sentences for training: the tokens are treated as complete sentences and actions and stack state are generated from the partial trees.

tial trees Y that correspond to the entire sentence by solving $\arg\max_{Y \subseteq \mathcal{Y}_p} \text{CKYSCORE}(y)$ under two constraints: (a) no two trees from Y correspond to overlapping tokens, and (b) for each token in x, there exists $y \in Y$ that corresponds to it. We solve the $\arg\max$ using dynamic programming. The generated set Y approximates an intermediate state of a shift-reduce derivation. However, $\mathcal{Y}_p$ often does not contain high quality partial derivation for all spans. To skip low quality partial trees and spans that have no trees, we generate empty trees y_e for every span, where $\text{CKYSCORE}(y_e) = 0$, and add them to $\mathcal{Y}_p$. If the set of selected partial trees Y includes empty trees, we divide the sentence to separate examples and ignore these parts. This results in partial and approximate stack reconstruction. Finally, since $\mathcal{Y}_P$ is noisy, we prune from it partial trees with a root that does not match the syntactic type for this span from an automatically generated CCGBank (Hockenmaier and Steedman, 2007) syntactic parse.

Our complete learning algorithm alternates between epochs of learning with complete parse trees and learning with partial derivations. In epochs where we use partial derivations, we use a modified version of Algorithm 1, where lines 9-10 are updated to use the above process.

5 Related work

Our approach is inspired by recent results in dependency parsing, specifically by the architecture of Chen and Manning (2014), which was further developed by Weiss et al. (2015) and Andor et al. (2016). Dyer et al. (2015) proposed to encode the parser state using an LSTM recurrent architecture, which has been shown generalize well between languages (Ballesteros et al., 2015; Ammar et al., 2016). Our network architecture combines ideas from the two threads: we use feature embeddings and a simple MLP to score actions, while our probability distribution is similar to the LSTM parser.

The majority of CCG approaches for semantic parsing rely on CKY parsing with beam search (e.g., Zettlemoyer and Collins, 2005, 2007; Kwiatkowski et al., 2010, 2011; Artzi and Zettlemoyer, 2011, 2013; Artzi et al., 2014; Matuszek et al., 2012; Kushman and Barzilay, 2013). Semantic parsing with other formalisms also often relied on CKY-style algorithms (e.g., Liang et al., 2009; Kim and Mooney, 2012). With a similar goal to ours, Berant and Liang (2015) designed an agenda-based parser. In contrast, we focus on a method with linear number of operations guarantee.

Following the work of Collins and Roark (2004) on learning for syntactic parsers, Artzi et al. (2015) proposed an early update procedure for inducing CCG grammars with a CKY parser. Our partial derivations learning method generalizes this method to parsers with global features.

6 Experimental Setup

Task and Data We evaluate on AMR parsing with CCG. AMR is a general-purpose meaning representation, which has been used in multiple tasks (Pan et al., 2015; Liu et al., 2015; Sawai et al., 2015; Garg et al., 2016), We use the newswire portion of AMR Bank 1.0 release (LDC2014T12), which displays some of the fundamental challenges in semantic parsing, including long newswire sentences with a broad array of syntactic and semantic phenomena. We follow the standard train/dev/test split of 6603/826/823 sentences. We evaluate with the SMATCH metric (Cai and Knight, 2013). Our parser is incorporated into the two-stage approach of Artzi et al. (2015). The approach includes a bi-directional and deterministic conversion between AMR and lambda calculus. Distant references, for example such as introduced by pronouns, are represented using Skolem IDs, globally-scoped existentially-quantified unique IDs. A derivation includes a CCG tree, which maps the sentence to an *underspecified logical form*, and a constant mapping, which maps underspecified elements to their fully specified form. The key to the approach is the underspecified logical forms, where distant references and most relations are not fully specified, but instead represented

AMR	Underspecified Logical Form	Logical Form
$(c/\text{conclude-02}$	$\mathcal{A}_1(\lambda c.\text{conclude-02}(c) \wedge$	$\mathcal{A}_1(\lambda c.\text{conclude-02}(c) \wedge$
:ARG0 (l/lawyer)	$\text{ARG0}(c, \mathcal{A}_2(\lambda l.\text{lawyer}(l))) \wedge$	$\text{ARG0}(c, \mathcal{A}_2(\lambda l.\text{lawyer}(l))) \wedge$
:ARG1 $(a/\text{argument}$	$\text{ARG1}(c, \mathcal{A}_3(\lambda a.\text{argument}(a) \wedge$	$\text{ARG1}(c, \mathcal{A}_3(\lambda a.\text{argument}(a) \wedge$
:poss $l)$	$\text{poss}(a, \mathcal{R}(\textbf{ID}))))) \wedge$	$\text{poss}(a, \mathcal{R}(2)))) \wedge$
:time $(l2/late))$	$\textbf{REL}(c, \mathcal{A}_4(\lambda l2.late(l2))))$	$\text{time}(c, \mathcal{A}_4(\lambda l2.late(l2))))$

Figure 5: AMR for the sentence *the lawyer concluded his arguments late*. In Artzi et al. (2015), The AMR (left) is deterministically converted to the logical form (right). The underspecified logical form is the result of the first stage, CCG parsing, and contains two placeholders (bolded): ID for a reference, and REL for a relation. To generate the final logical form, the second stage resolves ID to the identifier of the *lawyer* (2), and REL to the relation time. We focus on a model for the first stage and use an existing model for the second stage.

as placeholders. Figure 5 shows an example AMR, its lambda calculus conversion, and its underspecified logical form. (Artzi et al., 2015) use a CKY parser to identify the best CCG tree, and a factor graph for the second stage. We integrate our shift-reduce parser into the two-stage setup by replacing the CKY parser. We use the same CCG configuration and integrate our parser into the join probabilistic model. Formally, given a sentence x, the probability of an AMR logical form z is

$$p(z \mid x) = \sum_u p(z \mid u, x) \sum_{y \in \mathcal{Y}(u)} p(y \mid x) \ ,$$

where u is an underspecified logical form, $\mathcal{Y}(u)$ is the set of CCG trees with u at the root. We use our shift-reduce parser to compute $p(y \mid x)$ and use the pre-trained model from Artzi et al. (2015) for $p(z \mid u, x)$. Following Artzi et al. (2015), we disallow configurations that will not result in a valid AMR, and design a heuristic post-processing technique to recover a single logical form from terminal configurations that include multiple disconnected partial trees on the stack. We use the recovery technique when no complete parses are available.

Tools We evaluate with the SMATCH metric (Cai and Knight, 2013). We use EasyCCG (Lewis and Steedman, 2014) for CCGBank categories (Section 4.1). We implement our system using Cornell SPF (Artzi, 2016), and the deeplearning4j library.[10] The setup of Artzi et al. (2015) also includes the Illinois NER (Ratinov and Roth, 2009) and Stanford CoreNLP POS Tagger (Manning et al., 2014).

Parameters and Initialization We minimize our loss on a held-out 10% of the training data to tune our parameters, and train the final model on the full data. We set the number of epochs $T = 3$, regularization coefficient $\ell_2 = 10^{-6}$, learning rate $\mu = 0.05$, skipping term $\gamma = 1.0$. We set the dimensionality of feature embeddings based on the vocabulary size of the feature type. The exact dimensions are listed in the supplementary material. We use 65 ReLU units for $\mathbf{h}_1$ and $\mathbf{h}_2$, and 50 units for $\mathbf{h}_3$. We initialize θ with the initialization scheme of Glorot and Bengio (2010), except the bias term for ReLu layers, which we initialize to 0.1 to increase the number of active units on initialization. During test, we use the vector $\mathbf{0}$ as embedding for unseen features. We use a beam of 512 for testing and 2 for CONFGEN (Section 4).

[10] http://deeplearning4j.org/

Parser	P	R	F
CKY (Artzi et al., 2015)	67.2	65.1	66.1
Greedy CKY	64.1	11.29	19.19
SR (complete model)	67.0	63.4	65.3
w/o semantic embedding	67.1	63.3	65.1
w/o partial derivation learning	66.0	62.2	64.0
Ensemble SR (syntax)	68.2	64.1	66.0
Ensemble SR (syntax, semantics)	68.1	63.9	65.9
SR with CKY model	52.5	49.36	50.88

Table 1: Development SMATCH results.

Parser	P	R	F
JAMR[11]	67.8	59.2	63.2
CKY (Artzi et al., 2015)	66.8	65.7	66.3
Shift Reduce	68.1	64.2	66.1
Wang et al. (2015a)[13]	72.0	67.0	70.0

Table 2: Test SMATCH results.[12]

Model Ensemble For our final results, we marginalize the output over three models $\mathcal{M}$ using $p(z \mid x, \theta, \Lambda) = \frac{1}{|M|} \sum_{m \in \mathcal{M}} p(z \mid m, x, \theta, \Lambda)$.

7 Results

Table 1 shows development results. We trained each model three times and report the best performance. We observed a variance of roughly 0.5 in these runs. We experimented with different features for configuration embedding and with removing learning with partial derivations (Section 4.1). The com-

plete model gives the best single-model performance of 65.3 F1 SMATCH, and we observe the benefits for semantic embeddings and learning from partial derivations. Using partial derivations allowed us to learn 370K more features, 22% of observed embeddings. We also evaluate ensemble performance. We observe an overall improvement in performance. However, with multiple models, the benefit of using semantic embeddings vanishes. This result is encouraging since semantic embeddings can be expensive to compute if the logical form grows with sentence length. We also provide results for running a shift-reduce log-linear parser $p(a \mid c) \propto \exp\{\mathbf{w}^T \phi_{\text{CKY}}(a, c)\}$ using the input CKY model. We observe a significant drop in performance, which demonstrates the overall benefit of our architecture.

Figure 6 shows the development performance of our best performing ensemble model for different beam sizes. The performance decays slowly with decreasing beam size. Surprisingly, our greedy parser achieves 59.77 SMATCH F1, while the CKY parser with a beam of 1 achieves only 19.2 SMATCH F1 (Table 1). This allows our parser to trade-off a modest drop in accuracy for a significant improvement in runtime.

Table 2 shows the test results using our best performing model (ensemble with syntax features). We compare our approach to the CKY parser of Artzi et al. (2015) and JAMR (Flanigan et al., 2014).[11,12] We also list the results of Wang et al. (2015b), who demonstrated the benefit of auxiliary analyzers and is the current state of the art.[13] Our performance is comparable to the CKY parser of (Artzi et al., 2015), which we use to bootstrap our system. This demonstrates the ability of our parser to match the performance of a dynamic-programming parser, which executes significantly more operations per sentence.

Finally, Figure 7 shows our parser runtime relative to sentence length. In this analysis, we focus on runtime, and therefore use a single model.

[11] JAMR results are taken from Artzi et al. (2015).

[12] Pust et al. (2015), Flanigan et al. (2014), and Wang et al. (2015b) report results on different sections of the corpus. These results are not comparable to ours.

[13] Our goal is to study the effectiveness of our model transfer approach and architecture. Therefore, we avoid using any resources used in (Wang et al., 2015b) that are not used in the CKY parser we compare to.

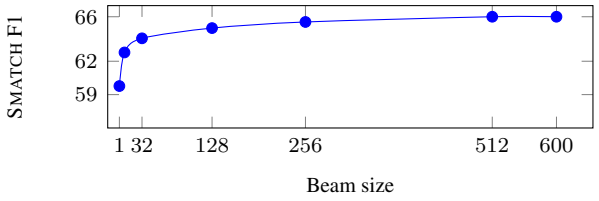

Figure 6: The effect of beam size on model performance.

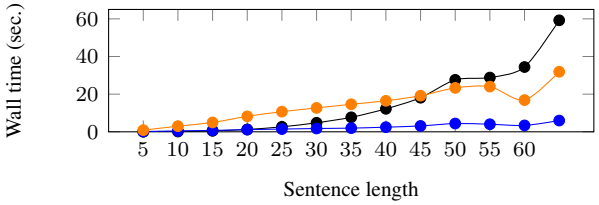

Figure 7: Wall-time performance of shift reduce parser with only syntax features (blue), with syntax and semantic features (orange) and the CKY parser of Artzi et al. (2015) (black).

We compare two versions of our system, including and excluding semantic embeddings, and the CKY parser of Artzi et al. (2015). We run both parsers with 16 cores and 122GB memory. The shift-reduce parser is three times faster on average, and up to ten times faster on long sentences. Since our parser is currently using CPUs, future work focused on GPU porting is likely to see further improvements.

8 Conclusion

Our parser design emphasizes a balance between model capacity and the ability to combine atomic features against the computational cost of scoring actions. We also design a learning algorithm to transfer learned models and learn neural network models from ambiguous and partial supervision. Our model shares many commonalities with transition-based dependency parsers. This makes it a good starting point to study the effectiveness of other dependency parsing techniques for semantic parsing, for example global normalization (Andor et al., 2016) and bidirectional LSTM feature representations (Kiperwasser and Goldberg, 2016).

Acknowledgments

This research was supported in part by gifts from Google and Amazon. The authors thank Kenton Lee for technical advice, and Adam Gibson and Alex Black of Skymind for help with Deeplearning4j. We also thank Tom Kwiatkowski, Arzoo Katiyar, Tianze Shi, Vlad Niculae, the Cornell NLP Group, and the reviewers for helpful advice.

References

Ammar, W., Mulcaire, G., Ballesteros, M., Dyer, C., and Smith, N. A. (2016). Many languages, one parser. *Transactions of the Association for Computational Linguistics*.

Andor, D., Alberti, C., Weiss, D., Severyn, A., Presta, A., Ganchev, K., Petrov, S., and Collins, M. (2016). Globally normalized transition-based neural networks. *CoRR*.

Artzi, Y. (2016). Cornell SPF: Cornell semantic parsing framework. *ArXiv e-prints*.

Artzi, Y., Das, D., and Petrov, S. (2014). Learning compact lexicons for CCG semantic parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

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*.

Artzi, Y. and Zettlemoyer, L. S. (2011). Bootstrapping semantic parsers from conversations. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Artzi, Y. and Zettlemoyer, L. S. (2013). Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics*, 1.

Ballesteros, M., Dyer, C., and Smith, N. A. (2015). Improved transition-based parsing by modeling characters instead of words with LSTMs.

Banarescu, L., Bonial, C., Cai, S., Georgescu, M., Griffitt, K., Hermjakob, U., Knight, K., Koehn, P., Palmer, M., and Schneider, N. (2013). Abstract meaning representation for sembanking. In *Proceedings of the Linguistic Annotation Workshop*.

Berant, J. and Liang, P. (2015). Imitation learning of agenda-based semantic parsers. *Transactions of the Association for Computational Linguistics*, 3.

Cai, S. and Knight, K. (2013). Smatch: an evaluation metric for semantic feature structures. In *Proceedings of the Conference of the Association of Computational Linguistics*.

Chen, D. and Manning, C. D. (2014). A fast and accurate dependency parser using neural networks. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Collins, M. and Roark, B. (2004). Incremental parsing with the perceptron algorithm. In *Proceedings of the Annual Meeting on Association for Computational Linguistics*.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*.

Dyer, C., Ballesteros, M., Ling, W., Matthews, A., and Smith, N. A. (2015). Transition-based dependency parsing with stack long short-term memory. In *Proceedings of the Annual Meeting on Association for Computational Linguistics*.

Flanigan, J., Thomson, S., Carbonell, J., Dyer, C., and Smith, N. A. (2014). A discriminative graph-based parser for the Abstract Meaning Representation. In *Proceedings of the Conference of the Association of Computational Linguistics*.

Garg, S., Galstyan, A., Hermjakob, U., and Marcu, D. (2016). Extracting biomolecular interactions using semantic parsing of biomedical text. In *Proceedings of the Conference on Artificial Intelligence*.

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

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics*.

Goldberg, Y. and Nivre, J. (2013). Training deterministic parsers with non-deterministic oracles. *Transactions of the Association for Computational Linguistics*, 1.

Hockenmaier, J. and Steedman, M. (2007). CCGBank: A corpus of CCG derivations and dependency structures extracted from the Penn Treebank. *Computational Linguistics*.

Kim, J. and Mooney, R. J. (2012). Unsupervised PCFG induction for grounded language learning

with highly ambiguous supervision. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Kiperwasser, E. and Goldberg, Y. (2016). Simple and accurate dependency parsing using bidirectional LSTM feature representations. *Transactions of the Association for Computational Linguistics*, 4.

Kushman, N. and Barzilay, R. (2013). Using semantic unification to generate regular expressions from natural language. In *Proceedings of the Human Language Technology Conference of the North American Association for Computational Linguistics*.

Kwiatkowski, T., Zettlemoyer, L. S., Goldwater, S., and Steedman, M. (2010). Inducing probabilistic CCG grammars from logical form with higher-order unification. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

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*.

Lewis, M. and Steedman, M. (2014). A* CCG parsing with a supertag-factored model. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Liang, P., Jordan, M., and Klein, D. (2009). Learning semantic correspondences with less supervision. In *Proceedings of the Joint Conference of the Association for Computational Linguistics the International Joint Conference on Natural Language Processing*.

Liu, F., Flanigan, J., Thomson, S., Sadeh, N., and Smith, N. A. (2015). Toward abstractive summarization using semantic representations. In *Proceedings of the North American Association for Computational Linguistics*.

Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S. J., and McClosky, D. (2014). The Stanford CoreNLP natural language processing toolkit. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

Matuszek, C., FitzGerald, N., Zettlemoyer, L. S., Bo, L., and Fox, D. (2012). A joint model of language and perception for grounded attribute learning. In *Proceedings of the International Conference on Machine Learning*.

Pan, X., Cassidy, T., Hermjakob, U., Ji, H., and Knight, K. (2015). Unsupervised entity linking with Abstract Meaning Representation. In *Proceedings of the North American Association for Computational Linguistics*.

Pust, M., Hermjakob, U., Knight, K., Marcu, D., and May, J. (2015). Parsing english into abstract meaning representation using syntax-based machine translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Ratinov, L. and Roth, D. (2009). Design challenges and misconceptions in named entity recognition. In *Proceedings of the Conference on Computational Natural Language Learning*.

Sawai, Y., Shindo, H., and Matsumoto, Y. (2015). Semantic structure analysis of noun phrases using abstract meaning representation. In *Proceedings of the annual meeting on Association for Computational Linguistics*.

Steedman, M. (1996). *Surface Structure and Interpretation*. The MIT Press.

Steedman, M. (2000). *The Syntactic Process*. The MIT Press.

Wang, C., Xue, N., and Pradhan, S. (2015a). Boosting transition-based amr parsing with refined actions and auxiliary analyzers. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

Wang, C., Xue, N., Pradhan, S., and Pradhan, S. (2015b). A transition-based algorithm for AMR parsing. In *Proceedings of the North American Association for Computational Linguistics*.

Weiss, D., Alberti, C., Collins, M., and Petrov, S. (2015). Structured training for neural network transition-based parsing. In *Proceedings of the annual meeting on Association for Computational Linguistics*.

Zettlemoyer, L. S. and Collins, M. (2005). Learning to map sentences to logical form: Structured clas-

sification with probabilistic categorial grammars. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*.

Zettlemoyer, L. S. and Collins, M. (2007). Online learning of relaxed CCG grammars for parsing to logical form. In *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*.