Span-Based Constituency Parsing with a Structure-Label System and Provably Optimal Dynamic Oracles

James Cross and **Liang Huang**
School of EECS, Oregon State University, Corvallis, OR, USA
{james.henry.cross.iii, liang.huang.sh}@gmail.com

Abstract

Parsing accuracy using efficient greedy transition systems has improved dramatically in recent years thanks to neural networks. Despite striking results in dependency parsing, however, neural models have not surpassed state-of-the-art approaches in constituency parsing. To remedy this, we introduce a new shift-reduce system whose stack contains merely sentence spans, represented by a bare minimum of LSTM features. We also design the first provably optimal dynamic oracle for constituency parsing, which runs in amortized $O(1)$ time, compared to $O(n^3)$ oracles for standard dependency parsing. Training with this oracle, we achieve the best F_1 scores on both English and French of any parser that does not use reranking or external data.

1 Introduction

Parsing is an important problem in natural language processing which has been studied extensively for decades. Between the two basic paradigms of parsing, constituency parsing, the subject of this paper, has in general proved to be the more difficult than dependency parsing, both in terms of accuracy and the run time of parsing algorithms.

There has recently been a huge surge of interest in using neural networks to make parsing decisions, and such models continue to dominate the state of the art in dependency parsing (Andor et al., 2016). In constituency parsing, however, neural approaches are still behind the state-of-the-art (Carreras et al., 2008; Shindo et al., 2012; Thang et al., 2015); see more details in Section 5.

To remedy this, we design a new parsing framework that is more suitable for constituency parsing, and that can be accurately modeled by neural networks. Observing that constituency parsing is primarily focused on sentence spans (rather than individual words, as is dependency parsing), we propose a novel adaptation of the shift-reduce system which reflects this focus. In this system, the stack consists of sentence spans rather than partial trees. It is also factored into two types of parser actions, structural and label actions, which alternate during a parse. The structural actions are a simplified analogue of shift-reduce actions, omitting the directionality of reduce actions, while the label actions directly assign nonterminal symbols to sentence spans.

Our neural model processes the sentence once for each parse with a recurrent network. We represent parser configurations with a very small number of span features (4 for structural actions and 3 for label actions). Extending Wang and Chang (2016), each span is represented as the difference of recurrent output from multiple layers in each direction. No pretrained embeddings are required.

We also extend the idea of dynamic oracles from dependency to constituency parsing. The latter is significantly more difficult than the former due to F_1 being a combination of precision and recall (Huang, 2008), and yet we propose a simple and extremely efficient oracle (amortized $O(1)$ time). This oracle is proved optimal for F_1 as well as both of its components, precision and recall. Trained with this oracle, our parser achieves what we believe to be the best results for any parser without reranking which was trained only on the Penn Treebank and the French Treebank, despite the fact that it is not only linear-time, but also strictly greedy.

We make the following main contributions:

- A novel factored transition parsing system where the stack elements are sentence spans rather than partial trees (Section 2).
- A neural model where sentence spans are represented as differences of output from a multi-layer bi-directional LSTM (Section 3).
- The first provably optimal dynamic oracle for

constituency parsing which is also extremely efficient (amortized $O(1)$ time) (Section 4).

- The best F_1 scores of any single-model, closed training set, parser for English and French.

We are also publicly releasing the source code for one implementation of our parser.[1]

2 Parsing System

We present a new transition-based system for constituency parsing whose fundamental unit of computation is the sentence span. It uses a stack in a similar manner to other transition systems, except that the stack contains sentence spans with no requirement that each one correspond to a partial tree structure during a parse.

The parser alternates between two types of actions, structural and label, where the structural actions follow a path to make the stack spans correspond to sentence phrases in a bottom-up manner, while the label actions optionally create tree brackets for the top span on the stack. There are only two structural actions: *shift* is the same as other transition systems, while *combine* merges the top two sentence spans. The latter is analogous to a reduce action, but it does not immediately create a tree structure and is non-directional. Label actions do create a partial tree on top of the stack by assigning one or more non-terminals to the topmost span.

Except for the use of spans, this factored approach is similar to the odd-even parser from Mi and Huang (2015). The fact that stack elements do not have to be tree-structured, however, means that we can create productions with arbitrary arity, and no binarization is required either for training or parsing. This also allows us to remove the directionality inherent in the shift-reduce system, which is at best an imperfect fit for constituency parsing. We do follow the practice in that system of labeling unary chains of non-terminals with a single action, which means our parser uses a fixed number of steps, $(4n-2)$ for a sentence of n words.

Figure 1 shows the formal deductive system for this parser. The stack σ is modeled as a list of strictly increasing integers whose first element is always

$$\begin{array}{lll}
\text{input:} & w_0 \ldots w_{n-1} & \\
\text{axiom:} & \langle 0,\ [0],\ \emptyset \rangle & \\
\text{goal:} & \langle 2(2n-1),\ [0,n],\ t \rangle & \\
\mathsf{sh} & \dfrac{\langle z,\ \sigma|j,\ t\rangle}{\langle z+1,\ \sigma|j|j{+}1,\ t\rangle} & j<n, \text{even } z \\
\mathsf{comb} & \dfrac{\langle z,\ \sigma|i|k|j,\ t\rangle}{\langle z+1,\ \sigma|i|j,\ t\rangle} & \text{even } z \\
\mathsf{label}\text{-}X & \dfrac{\langle z,\ \sigma|i|j,\ t\rangle}{\langle z+1,\ \sigma|i|j,\ t\cup\{{}_iX_j\}\rangle} & \text{odd } z \\
\mathsf{nolabel} & \dfrac{\langle z,\ \sigma|i|j,\ t\rangle}{\langle z+1,\ \sigma|i|j,\ t\rangle} & z<(4n{-}1), \text{odd } z
\end{array}$$

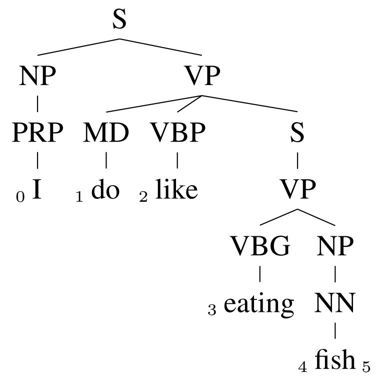
Figure 1: Deductive system for the Structure/Label transition parser. The stack σ is represented as a list of integers where the span defined by each consecutive pair of elements is a sentence segment on the stack. Each X is a nonterminal symbol or an ordered unary chain. The set t contains labeled spans of the form ${}_iX_j$, which at the end of a parse, fully define a parse tree.

zero. These numbers are word boundaries which define the spans on the stack. In a slight abuse of notation, however, we sometimes think of it as a list of pairs (i,j), which are the actual sentence spans, i.e., every consecutive pair of indices on the stack, initially empty. We represent stack spans by trapezoids $({}_i\triangle_j)$ in the figures to emphasize that they may or not have tree stucture.

The parser alternates between structural actions and label actions according to the parity of the parser step z. In even steps, it takes a structural action, either combining the top two stack spans, which requires at least two spans on the stack, or introducing a new span of unit length, as long as the entire sentence is not already represented on the stack

In odd steps, the parser takes a label action. One possibility is labeling the top span on the stack, (i,j) with either a nonterminal label or an ordered unary chain (since the parser has only one opportunity to label any given span). Taking no action, designated $\mathsf{nolabel}$, is also a possibility. This is essentially a null operation except that it returns the parser to an even step, and this action reflects the decision that (i,j) is not a (complete) labeled phrase in the tree. In the final step, $(4n-2)$, $\mathsf{nolabel}$ is not allowed

[1]code: `https://github.com/jhcross/span-parser`



(a) gold parse tree

steps	structural action	label action	stack after	bracket
1–2	sh(I/PRP)	label-NP	${}_0\triangle_1$	${}_0\mathrm{NP}_1$
3–4	sh(do/MD)	nolabel	${}_0\triangle_1\triangle_2$	
5–6	sh(like/VBP)	nolabel	${}_0\triangle_1\triangle_2\triangle_3$	
7–8	comb	nolabel	${}_0\triangle_1\triangle_3$	
9–10	sh(eating/VBG)	nolabel	${}_0\triangle_1\triangle_3\triangle_4$	
11–12	sh(fish/NN)	label-NP	${}_0\triangle_1\triangle_3\triangle_4\triangle_5$	${}_4\mathrm{NP}_5$
13–14	comb	label-S-VP	${}_0\triangle_1\triangle_3\triangle_5$	${}_3\mathrm{S}_5$, ${}_3\mathrm{VP}_5$
15–16	comb	label-VP	${}_0\triangle_1\triangle_5$	${}_1\mathrm{VP}_5$
17–18	comb	label-S	${}_0\triangle_5$	${}_0\mathrm{S}_5$

(b) static oracle actions

Figure 2: The running example. It contains one ternary branch and one unary chain (S-VP), and NP-PRP-I and NP-NN-fish are *not* unary chains in our system. Each stack is just a list of numbers but is visualized with spans here.

since the parser must produce a tree.

Figure 2 shows a complete example of applying this parsing system to a very short sentence (*"I do like eating fish"*) that we will use throughout this section and the next. The action in step 2 is label-NP because "I" is a one-word noun phrase (parts of speech are taken as input to our parser, though it could easily be adapted to include POS tagging in label actions). If a single word is not a complete phrase (e.g., "do"), then the action after a shift is nolabel.

The ternary branch in this tree (VP $\rightarrow$ MD VBP S) is produced by our parser in a straightforward manner: after the phrase "do like" is combined in step 7, no label is assigned in step 8, successfully delaying the creation of a bracket until the verb phrase is fully formed on the stack. Note also that the unary production in the tree is created with a single action, label-S-VP, in step 14.

The static oracle to train this parser simply consists of taking actions to generate the gold tree with a "short-stack" heuristic, meaning combine first whenever combine and shift are both possible.

3 LSTM Span Features

Long short-term memory networks (LSTM) are a type of recurrent neural network model proposed by Hochreiter and Schmidhuber (1997) which are very effective for modeling sequences. They are able to capture and generalize from interactions among their sequential inputs even when separated by a long distance, and thus are a natural fit for analyzing natural language. LSTM models have proved to be a powerful tool for many learning tasks in natural language, such as language modeling (Sundermeyer et al., 2012) and translation (Sutskever et al., 2014).

LSTMs have also been incorporated into parsing in a variety of ways, such as directly encoding an entire sentence (Vinyals et al., 2015), separately modeling the stack, buffer, and action history (Dyer et al., 2015), to encode words based on their character forms (Ballesteros et al., 2015), and as an element in a recursive structure to combine dependency subtrees with their left and right children (Kiperwasser and Goldberg, 2016a).

For our parsing system, however, we need a way to model arbitrary sentence spans in the context of the rest of the sentence. We do this by representing each sentence span as the elementwise difference of the vector outputs of the LSTM outputs at different time steps, which correspond to word boundaries. If the sequential output of the recurrent network for the sentence is $f_0, ..., f_n$ in the forward direction and $b_n, ..., b_0$ in the backward direction then the span (i, j) would be represented as the concatenation of the vector differences $(f_j - f_i)$ and $(b_i - b_j)$.

The spans are represented using output from both backward and forward LSTM components, as can be seen in Figure 3. This is essentially the LSTM-Minus feature representation described by Wang and Chang (2016) extended to the bi-directional case. In initial experiments, we found that there was essentially no difference in performance between using the difference features and concatenating all end-

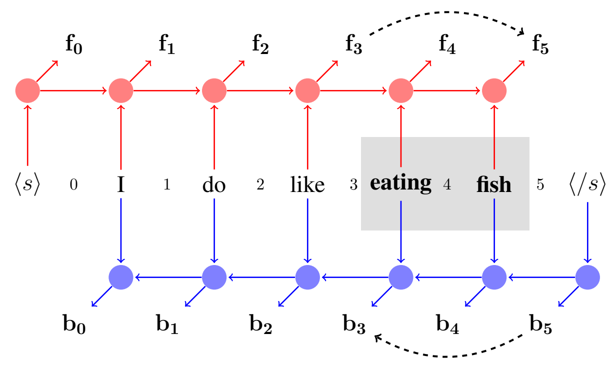


Figure 3: Word spans are modeled by differences in LSTM output. Here the span $_3$ *eating fish* $_5$ is represented by the vector differences ($\mathbf{f}_5 - \mathbf{f}_3$) and ($\mathbf{b}_3 - \mathbf{b}_5$). The forward difference corresponds to LSTM-Minus (Wang and Chang, 2016).

point vectors, but our approach is almost twice as fast.

This model allows a sentence to be processed once, and then the same recurrent outputs can be used to compute span features throughout the parse. Intuitively, this allows the span differences to learn to represent the sentence spans in the context of the rest of the sentence, not in isolation (especially true for LSTM given the extra hidden recurrent connection, typically described as a “memory cell”). In practice, we use a two-layer bi-directional LSTM, where the input to the second layer combines the forward and backward outputs from the first layer at that time step. For each direction, the components from the first and second layers are concatenated to form the vectors which go into the span features. See Cross and Huang (2016) for more details on this approach.

For the particular case of our transition constituency parser, we use only four span features to determine a structural action, and three to determine a label action, in each case partitioning the sentence exactly. The reason for this is straightforward: when considering a structural action, the top two spans on the stack must be considered to determine whether they should be combined, while for a label action, only the top span on the stack is important, since that is the candidate for labeling. In both cases the remaining sentence prefix and suffix are also included. These features are shown in Table 1.

The input to the recurrent network at each time step consists of vector embeddings for each word

Action	Stack	LSTM Span Features
Structural	$\sigma \mid i \mid k \mid j$	$_0$ ▭ $_i$ △ $_k$ △ $_j$ ▭ $_n$
Label	$\sigma \mid i \mid j$	$_0$ ▭ $_i$ △ $_j$ ▭ $_n$

Table 1: Features used for the parser. No label or tree-structure features are required.

and its part-of-speech tag. Parts of speech are predicted beforehand and taken as input to the parser, as in much recent work in parsing. In our experiments, the embeddings are randomly initialized and learned from scratch together with all other network weights, and we would expect further performance improvement from incorporating embeddings pretrained from a large external corpus.

The network structure after the the span features consists of a separate multilayer perceptron for each type of action (structural and label). For each action we use a single hidden layer with rectified linear (ReLU) activation. The model is trained on a per-action basis using a single correct action for each parser state, with a negative log softmax loss function, as in Chen and Manning (2014).

4 Dynamic Oracle

The baseline method of training our parser is what is known as a static oracle: we simply generate the sequence of actions to correctly parse each training sentence, using a short-stack heuristic (i.e., combine first whenever there is a choice of shift and combine). This method suffers from a well-documeted problem, however, namely that it only “prepares” the model for the situation where no mistakes have been made during parsing, an inevitably incorrect assumption in practice. To alleviate this problem, Goldberg and Nivre (2013) define a dynamic oracle to return the best possible action(s) at any arbitrary configuration.

In this section, we introduce an easy-to-compute optimal dynamic oracle for our constituency parser. We will first define some concepts upon which the dynamic oracle is built and then show how optimal actions can be very efficiently computed using this framework. In broad strokes, in any arbitrary parser configuration c there is a set of brackets $t^*(c)$ from the gold tree which it is still possible to reach. By following dynamic oracle actions, all of those brackets and only those brackets will be predicted.

Even though proving the optimality of our dynamic oracle (Sec. 4.3) is involved, computing the oracle actions is extremely simple (Secs. 4.2) and efficient (Sec. 4.4).

4.1 Preliminaries and Notations

Before describing the computation of our dynamic oracle, we first need to rigorously establish the desired optimality of dynamic oracle. The structure of this framework follows Goldberg et al. (2014).

Definition 1. We denote $c \vdash_\tau c'$ iff. c' is the result of action τ on configuration c, also denoted functionally as $c' = \tau(c)$. We denote $\vdash$ to be the union of $\vdash_\tau$ for all actions τ, and $\vdash^*$ to be the reflexive and transitive closure of $\vdash$.

Definition 2 (descendant/reachable trees)**.** We denote $\mathcal{D}(c)$ to be the set of final descendant trees derivable from c, i.e., $\mathcal{D}(c) = \{t \mid c \vdash^* \langle z,\ \sigma,\ t\rangle\}$. This set is also called "reachable trees" from c.

Definition 3 (F_1)**.** We define the standard F_1 metric of a tree t with respect to gold tree t_G as $\mathrm{F}_1(t) = \frac{2rp}{r+p}$, where $r = \frac{|t \cap t_\mathrm{G}|}{|t_\mathrm{G}|}, p = \frac{|t \cap t_\mathrm{G}|}{|t|}$.

The following two definitions are similar to those for dependency parsing by Goldberg et al. (2014).

Definition 4. We extend the F_1 function to configurations to define the maximum possible F_1 from a given configuration: $\mathrm{F}_1(c) = \max_{t_1 \in \mathcal{D}(c)} \mathrm{F}_1(t_1)$.

Definition 5 (oracle)**.** We can now define the desired dynamic oracle of a configuration c to be the set of actions that retrain the optimal F_1:

$$\mathit{oracle}(c) = \{\tau \mid \mathrm{F}_1(\tau(c)) = \mathrm{F}_1(c)\}.$$

This abstract oracle is implemented by $\mathit{dyna}(\cdot)$ in Sec. 4.2, which we prove to be correct in Sec. 4.3.

Definition 6 (span encompassing)**.** We say span (i, j) is encompassed by span (p, q), notated $(i, j) \preceq (p, q)$, iff. $p \le i < j \le q$.

Definition 7 (strict encompassing)**.** We say span (i, j) is strictly encompassed by span (p, q), notated $(i, j) \prec (p, q)$, iff. $(i, j) \preceq (p, q)$ and $(i, j) \ne (p, q)$. We then extend this relation from spans to brackets, and notate ${}_iX_j \prec {}_pY_q$ iff. $(i, j) \prec (p, q)$.

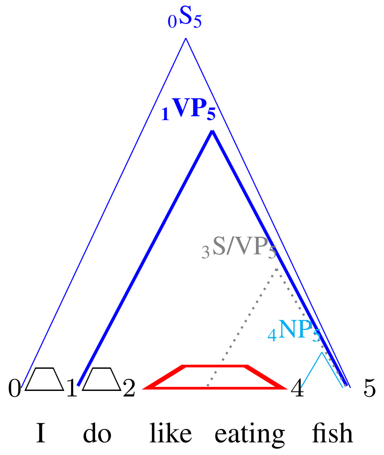


Figure 4: Reachable brackets (w.r.t. gold tree in Fig. 1) for $c = \langle 10,\ [0, 1, 2, 4],\ \{{}_0\mathrm{NP}_1\}\rangle$ which mistakenly combines "like eating". Trapezoids indicate stack spans (the top one in red), and solid triangles denote reachable brackets, with $\mathit{left}(c)$ in blue and $\mathit{right}(c)$ in cyan. The next reachable bracket, $\mathit{next}(c) = {}_1\mathbf{VP}_5$, is in bold. Brackets ${}_3\mathrm{VP}_5$ and ${}_3\mathrm{S}_5$ (in dotted triangle) cross the top span (thus unreachable), and ${}_0\mathrm{NP}_1$ is already recognized (thus not in $\mathit{reach}(c)$ either).

We next define a central concept, *reachable brackets*, which is made up of two parts, the left ones $\mathit{left}(c)$ which encompass (i, j) without crossing any stack spans, and the right ones $\mathit{right}(c)$ which are completely on the queue. See Fig. 4 for examples.

Definition 8 (reachable brackets)**.** For any configuration $c = \langle z,\ \sigma \,|\, i \,|\, j,\ t\rangle$, we define the set of reachable gold brackets (with respect to gold tree t_G) as

$$\mathit{reach}(c) = \mathit{left}(c) \cup \mathit{right}(c)$$

where the left- and right-reachable brackets are

$$\mathit{left}(c) = \{{}_pX_q \in t_\mathrm{G} \mid (i, j) \prec (p, q),\ p \in \sigma \,|\, i\}$$
$$\mathit{right}(c) = \{{}_pX_q \in t_\mathrm{G} \mid p \ge j\}$$

for even z, with the $\prec$ replaced by $\preceq$ for odd z.

Special case (initial): $\mathit{reach}(\langle 0,\ [0],\ \emptyset\rangle) = t_\mathrm{G}$.

The notation $p \in \sigma \,|\, i$ simply means (p, q) does not "cross" any bracket on the stack. Remember our stack is just a list of span boundaries, so if p coincides with one of them, (p, q)'s left boundary is not crossing and its right boundary q is not crossing either since $q \ge j$ due to $(i, j) \prec (p, q)$.

Also note that $\mathit{reach}(c)$ is strictly disjoint from t, i.e., $\mathit{reach}(c) \cap t = \emptyset$ and $\mathit{reach}(c) \subseteq t_\mathrm{G} - t$. See Figure 6 for an illustration.

Definition 9 (next bracket)**.** For any configuration $c = \langle z,\ \sigma \,|\, i \,|\, j,\ t \rangle$, the next reachable gold bracket (with respect to gold tree t_G) is the smallest reachable bracket (strictly) encompassing (i, j):

$$next(c) = \min_{\prec} left(c).$$

4.2 Structural and Label Oracles

For an even-step configuration $c = \langle z,\ \sigma \mid i \mid j,\ t \rangle$, we denote the next reachable gold bracket $next(c)$ to be ${}_pX_q$, and define the dynamic oracle to be:

$$dyna(c) = \begin{cases} \{\mathsf{sh}\} & \text{if } p = i \text{ and } q > j \\ \{\mathsf{comb}\} & \text{if } p < i \text{ and } q = j \\ \{\mathsf{sh}, \mathsf{comb}\} & \text{if } p < i \text{ and } q > j \end{cases} \quad (1)$$

As a special case $dyna(\langle 0,\ [0],\ \emptyset \rangle) = \{\mathsf{sh}\}$.

Figure 5 shows examples of this policy. The key insight is, if you follow this policy, you will *not* miss the next reachable bracket, but if you do not follow it, you certainly will. We formalize this fact below (with proof omitted due to space constraints) which will be used to prove the central results later.

Lemma 1. *For any configuration c, for any $\tau \in dyna(c)$, we have $reach(\tau(c)) = reach(c)$; for any $\tau' \notin dyna(c)$, we have $reach(\tau(c)) \subsetneq reach(c)$.*

The label oracles are much easier than structural ones. For an odd-step configuration $c = \langle z,\ \sigma \mid i \mid j,\ t \rangle$, we simply check if (i, j) is a valid span in the gold tree t_G and if so, label it accordingly, otherwise no label. More formally,

$$dyna(c) = \begin{cases} \{\mathsf{label}\text{-}X\} & \text{if some } {}_iX_j \in t_G \\ \{\mathsf{nolabel}\} & \text{otherwise} \end{cases} \quad (2)$$

4.3 Correctness

To show the optimality of our dynamic oracle, we begin by defining a special tree $t^*(c)$ and show that it is optimal among all trees reachable from configuration c. We then show that following our dynamic oracle (Eqs. 1–2) from c will lead to $t^*(c)$.

Definition 10 ($t^*(c)$)**.** For any configuration $c = \langle z,\ \sigma,\ t \rangle$, we define the optimal tree $t^*(c)$ to include all reachable gold brackets and nothing else. More formally, $t^*(c) = t \cup reach(c)$.

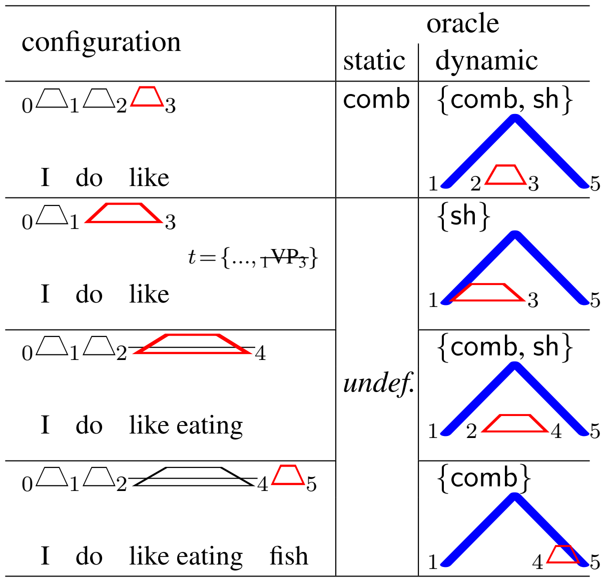


Figure 5: Dynamic oracle with respect to the gold parse in Fig. 2. The last three examples are off the gold path with strike out indicating structural or label mistakes. Trapezoids denote stack spans (top one in red) and the blue triangle denotes the next reachable bracket $next(c)$ which is ${}_1\mathrm{VP}_5$ in all cases.

We can show by induction that $t^*(c)$ is attainable:

Lemma 2. *For any configuration c, the optimal tree is a descendant of c, i.e., $t^*(c) \in \mathcal{D}(c)$.*

The following Theorem shows that $t^*(c)$ is indeed the best possible tree:

Theorem 1 (optimality of t^*)**.** *For any configuration c, $\mathrm{F}_1(t^*(c)) = \mathrm{F}_1(c)$.*

Proof. (SKETCH) Since $t^*(c)$ adds all possible additional gold brackets (the brackets in $reach(c)$), it is not possible to get higher recall. Since it adds no incorrect brackets, it is not possible to get higher pre-

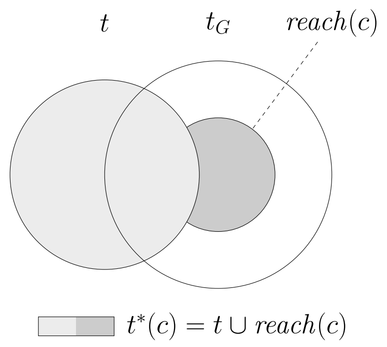


Figure 6: The optimal tree $t^*(c)$ adds all reachable brackets and nothing else. Note that $reach(c)$ and t are disjoint.

cision. Since F_1 is the harmonic mean of precision and recall, it also leads to the best possible F_1. $\square$

Corollary 1. *For any $c = \langle z,\ \sigma,\ t\rangle$, for any $t' \in \mathcal{D}(c)$ and $t' \neq t^*(c)$, we have $\mathrm{F}_1(t') < \mathrm{F}_1(c)$.*

We now need a final lemma about the policy $dyna(\cdot)$ (Eqs. 1–2) before proving the main result.

Lemma 3. *From any $c = \langle z,\ \sigma,\ t\rangle$, for any action $\tau \in dyna(c)$, we have $t^*(\tau(c)) = t^*(c)$. For any action $\tau' \notin dyna(c)$, we have $t^*(\tau'(c)) \neq t^*(c)$.*

Proof. (SKETCH) By case analysis on even/odd z. $\square$

We are now able to state and prove the main theoretical result of this paper (using Lemma 3, Theorem 1 and Corollary 1):

Theorem 2. *The function $dyna(\cdot)$ in Eqs. (1–2) satisfies the requirement of a dynamic oracle (Def. 5):*

$$dyna(c) = oracle(c) \text{ for any configuration } c.$$

4.4 Implementation and Complexity

For any configuration, our dynamic oracle can be computed in **amortized constant** time since there are only $O(n)$ gold brackets and thus bounding $|reach(c)|$ and the choice of $next(c)$. After each action, $next(c)$ either remains unchanged, or in the case of being crossed by a structural action or mislabeled by a label action, needs to be updated. This update is simply tracing the parent link to the next smallest gold bracket repeatedly until the new bracket encompasses span (i, j). Since there are at most $O(n)$ choices of $next(c)$ and there are $O(n)$ steps, the per-step cost is amortized constant time. Thus our dynamic oracle is much faster than the super-linear time oracle for arc-standard dependency parsing in Goldberg et al. (2014).

5 Related Work

Neural networks have been used for constituency parsing in a number of previous instances. For example, Socher et al. (2013) learn a recursive network that combines vectors representing partial trees, Vinyals et al. (2015) adapt a sequence-to-sequence model to produce parse trees, Watanabe and Sumita (2015) use a recursive model applying a shift-reduce system to constituency parsing with beam search, and Dyer et al. (2016) adapt the Stack-LSTM dependency parsing approach to this task. Durrett and Klein (2015) combine both neural and sparse features for a CKY parsing system. Our own previous work (Cross and Huang, 2016) use a recurrent sentence representation in a head-driven transition system which allows for greedy parsing but does not achieve state-of-the-art results.

The concept of "oracles" for constituency parsing (as the tree that is most similar to t_{G} among all possible trees) was first defined and solved by Huang (2008) in bottom-up parsing. In transition-based parsing, the dynamic oracle for shift-reduce dependency parsing costs worst-case $O(n^3)$ time (Goldberg et al., 2014). On the other hand, after the submission of our paper we became aware of a parallel work (Coavoux and Crabbé, 2016) that also proposed a dynamic oracle for their own incremental constituency parser. However, it is not optimal due to dummy non-terminals from binarization.

Network architecture	
Word embeddings	50
Tag embeddings	20
Morphological embeddings[†]	10
LSTM layers	2
LSTM units	200 / direction
ReLU hidden units	200 / action type
Training settings	
Embedding intialization	random
Training epochs	10
Minibatch size	10 sentences
Dropout (on LSTM output)	$p = 0.5$
ADADELTA parameters	$\rho = 0.99$, $\epsilon = 1 \times 10^{-7}$

Table 2: Hyperparameters. [†]French only.

6 Experiments

We present experiments on both the Penn English Treebank (Marcus et al., 1993) and the French Treebank (Abeillé et al., 2003). In both cases, all state-action training pairs for a given sentence are used at the same time, greatly increasing training speed since all examples for the same sentence share the same forward and backward pass through the recurrent part of the network. Updates are performed in minibatches of 10 sentences, and we shuffle the training sentences before each epoch. The results we report are trained for 10 epochs.

The only regularization which we employ during training is dropout (Hinton et al., 2012), which is applied with probability 0.5 to the recurrent outputs. It is applied separately to the input to the second LSTM layer for each sentence, and to the input to the ReLU hidden layer (span features) for each state-action pair. We use the ADADELTA method (Zeiler, 2012) to schedule learning rates for all weights. All of these design choices are summarized in Table 2.

In order to account for unknown words during training, we also adopt the strategy described by Kiperwasser and Goldberg (2016b), where words in the training set are replaced with the unknown-word symbol UNK with probability $p_{unk} = \frac{z}{z+f(w)}$ where $f(w)$ is the number of times the word appears in the training corpus. We choose the parameter z so that the training and validation corpora have approximately the same proportion of unknown words. For the Penn Treebank, for example, we used $z = 0.8375$ so that both the validation set and the (rest of the) training set contain approximately 2.76% unknown words. This approach was helpful but not critical, improving F_1 (on dev) by about 0.1 over training without any unknown words.

6.1 Training with Dynamic Oracle

The most straightforward use of dynamic oracles to train a neural network model, where we collect all action examples for a given sentence before updating, is "training with exploration" as proposed by Goldberg and Nivre (2013). This involves parsing each sentence according to the current model and using the oracle to determine correct actions for training. We saw very little improvement on the Penn treebank validation set using this method, however. Based on the parsing accuracy on the training sentences, this appears to be due to the model overfitting the training data early during training, thus negating the benefit of training on erroneous paths.

Accordingly, we also used a method recently proposed by Ballesteros et al. (2016), which specifically addresses this problem. This method introduces stochasticity into the training data parses by randomly taking actions according to the softmax distribution over action scores. This introduces realistic mistakes into the training parses, which we found was also very effective in our case, leading to higher F_1 scores, though it noticeably sacrifices recall in favor of precision.

This technique can also take a parameter α to flatten or sharpen the raw softmax distribution. The results on the Penn treebank development set for various values of α are presented in Table 3. We were surprised that flattening the distribution seemed to be the least effective, as training accuracy (taking into account sampled actions) lagged somewhat behind validation accuracy. Ultimately, the best results were for $\alpha = 1$, which we used for final testing.

Model	LR	LP	F_1
Static Oracle	91.34	91.43	91.38
Dynamic Oracle	91.14	91.61	91.38
+ Explore ($\alpha = 0.5$)	90.59	92.18	91.38
+ Explore ($\alpha = 1.0$)	91.07	92.22	**91.64**
+ Explore ($\alpha = 1.5$)	91.07	92.12	91.59

Table 3: Comparison of performance on PTB development set using different oracle training approaches.

6.2 Penn Treebank

Following the literature, we used the Wall Street Journal portion of the Penn Treebank, with standard splits for training (secs 2–21), development (sec 22), and test sets (sec 23). Because our parsing system seamlessly handles non-binary productions, minimal data preprocessing was required. For the part-of-speech tags which are a required input to our parser, we used the Stanford tagger with 10-way jackknifing.

Table 4 compares test our results on PTB to a range of other leading constituency parsers. Despite being a greedy parser, when trained using dynamic oracles with exploration, it achieves the best F_1 score of any closed-set single-model parser.

6.3 French Treebank

We also report results on the French treebank, with one small change to network structure. Specifically, we also included morphological features for each word as input to the recurrent network, using a small embedding for each such feature, to demonstrate that our parsing model is able to exploit such additional features.

We used the predicted morphological features, part-of-speech tags, and lemmas (used in place of word surface forms) supplied with the SPMRL 2014

Closed Training & Single Model	LR	LP	F_1
Sagae and Lavie (2006)	88.1	87.8	87.9
Petrov and Klein (2007)	90.1	90.3	90.2
Carreras et al. (2008)	90.7	91.4	91.1
Shindo et al. (2012)			91.1
†Socher et al. (2013)			90.4
Zhu et al. (2013)	90.2	90.7	90.4
Mi and Huang (2015)	90.7	90.9	90.8
†Watanabe and Sumita (2015)			90.7
Thang et al. (2015) (A*)	90.9	91.2	91.1
†*Dyer et al. (2016) (discrim.)			89.8
†*Cross and Huang (2016)			90.0
†***static oracle**	90.7	91.4	91.0
†***dynamic/exploration**	90.5	92.1	**91.3**
External/Reranking/Combo			
†Henderson (2004) (rerank)	89.8	90.4	90.1
McClosky et al. (2006)	92.2	92.6	92.4
Zhu et al. (2013) (semi)	91.1	91.5	91.3
Huang (2008) (forest)			91.7
†Vinyals et al. (2015) (WSJ)‡			90.5
†Vinyals et al. (2015) (semi)			92.8
†Durrett and Klein (2015)‡			91.1
†Dyer et al. (2016) (gen. rerank.)			92.4

Table 4: Comparison of performance of different parsers on PTB test set. †Neural. *Greedy. ‡External embeddings.

Parser	LR	LP	F_1
Björkelund et al. (2014)*,‡			82.53
Durrett and Klein (2015)‡			81.25
Coavoux and Crabbé (2016)			80.56
static oracle	83.50	82.87	83.18
dynamic/exploration	81.90	84.77	**83.31**

Table 5: Results on French Treebank. *reranking, ‡external.

data set (Seddah et al., 2014). It is thus possible that results could be improved further using an integrated or more accurate predictor for those features. Our parsing and evaluation also includes predicting POS tags for multi-word expressions as is the standard practice for the French treebank, though our results are similar whether or not this aspect is included.

We compare our parser with other recent work in Table 5. We achieve state-of-the-art results even in comparison to Björkelund et al. (2014), which utilized both external data and reranking in achieving the best results in the SPMRL 2014 shared task.

6.4 Notes on Experiments

For these experiments, we performed very little hyperparameter tuning, due to time and resource contraints. We have every reason to believe that performance could be improved still further with such techniques as random restarts, larger hidden layers, external embeddings, and hyperparameter grid search, as demonstrated by Weiss et al. (2015).

We also note that while our parser is very accurate even with greedy decoding, the model is easily adaptable for beam search, particularly since the parsing system already uses a fixed number of actions. Beam search could also be made considerably more efficient by caching post-hidden-layer feature components for sentence spans, essentially using the precomputation trick described by Chen and Manning (2014), but on a per-sentence basis.

7 Conclusion and Future Work

We have developed a new transition-based constituency parser which is built around sentence spans. It uses a factored system alternating between structural and label actions. We also describe a fast dynamic oracle for this parser which can determine the optimal set of actions with respect to a gold training tree in an arbitrary state. Using an LSTM model and only a few sentence spans as features, we achieve state-of-the-art accuracy on the Penn Treebank for all parsers without reranking, despite using strictly greedy inference.

In the future, we hope to achieve still better results using beam search, which is relatively straightforward given that the parsing system already uses a fixed number of actions. Dynamic programming (Huang and Sagae, 2010) could be especially powerful in this context given the very simple feature representation used by our parser, as noted also by Kiperwasser and Goldberg (2016b).

Acknowledgments

We thank the three anonymous reviewers for comments, Kai Zhao, Lemao Liu, Yoav Goldberg, and Slav Petrov for suggestions, Juneki Hong for proofreading, and Maximin Coavoux for sharing their manuscript. This project was supported in part by NSF IIS-1656051, DARPA FA8750-13-2-0041 (DEFT), and a Google Faculty Research Award.

References

Anne Abeillé, Lionel Clément, and François Toussenel. 2003. Building a treebank for french. In *Treebanks*, pages 165–187. Springer.

Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. 2016. Globally normalized transition-based neural networks. *Proceedings of ACL*.

Miguel Ballesteros, Chris Dyer, and Noah A Smith. 2015. Improved transition-based parsing by modeling characters instead of words with lstms. *arXiv preprint arXiv:1508.00657*.

Miguel Ballesteros, Yoav Goldberg, Chris Dyer, and Noah A Smith. 2016. Training with exploration improves a greedy stack-lstm parser. *arXiv preprint arXiv:1603.03793*.

Anders Björkelund, Ozlem Cetinoglu, Agnieszka Falenska, Richárd Farkas, Thomas Mueller, Wolfgang Seeker, and Zsolt Szántó. 2014. Introducing the ims-wrocław-szeged-cis entry at the spmrl 2014 shared task: Reranking and morpho-syntax meet unlabeled data. In *Proceedings of the First Joint Workshop on Statistical Parsing of Morphologically Rich Languages and Syntactic Analysis of Non-Canonical Languages*, pages 97–102.

Xavier Carreras, Michael Collins, and Terry Koo. 2008. Tag, dynamic programming, and the perceptron for efficient, feature-rich parsing. In *Proceedings of the Twelfth Conference on Computational Natural Language Learning*, pages 9–16. Association for Computational Linguistics.

Danqi Chen and Christopher D Manning. 2014. A fast and accurate dependency parser using neural networks. In *Empirical Methods in Natural Language Processing (EMNLP)*.

Maximin Coavoux and Benoît Crabbé. 2016. Neural greedy constituent parsing with dynamic oracles. *Proceedings of ACL*.

James Cross and Liang Huang. 2016. Incremental parsing with minimal features using bi-directional lstm. *Proceedings of ACL*.

Greg Durrett and Dan Klein. 2015. Neural crf parsing. *Proceedings of ACL*.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A Smith. 2015. Transition-based dependency parsing with stack long short-term memory. *Empirical Methods in Natural Language Processing (EMNLP)*.

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A Smith. 2016. Recurrent neural network grammars. *Proceedings of HLT-NAACL*.

Yoav Goldberg and Joakim Nivre. 2013. Training deterministic parsers with non-deterministic oracles. *Transactions of the association for Computational Linguistics*, 1:403–414.

Yoav Goldberg, Francesco Sartorio, and Giorgio Satta. 2014. A tabular method for dynamic oracles in transition-based parsing. *Trans. of ACL*.

James Henderson. 2004. Discriminative training of a neural network statistical parser. In *Proceedings of ACL*.

Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Liang Huang and Kenji Sagae. 2010. Dynamic programming for linear-time incremental parsing. In *Proceedings of ACL 2010*.

Liang Huang. 2008. Forest reranking: Discriminative parsing with non-local features. In *Proceedings of the ACL: HLT*, Columbus, OH, June.

Eliyahu Kiperwasser and Yoav Goldberg. 2016a. Easy-first dependency parsing with hierarchical tree lstms. *arXiv preprint arXiv:1603.00375*.

Eliyahu Kiperwasser and Yoav Goldberg. 2016b. Simple and accurate dependency parsing using bidirectional LSTM feature representations. *CoRR*, abs/1603.04351.

Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330.

David McClosky, Eugene Charniak, and Mark Johnson. 2006. Reranking and self-training for parser adaptation. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 337–344. Association for Computational Linguistics.

Haitao Mi and Liang Huang. 2015. Shift-reduce constituency parsing with dynamic programming and pos tag lattice. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

Slav Petrov and Dan Klein. 2007. Improved inference for unlexicalized parsing. In *Proceedings of HLT-NAACL*.

Kenji Sagae and Alon Lavie. 2006. A best-first probabilistic shift-reduce parser. In *Proceedings of the COLING/ACL on Main conference poster sessions*, pages 691–698. Association for Computational Linguistics.

Djamé Seddah, Sandra Kübler, and Reut Tsarfaty. 2014. Introducing the spmrl 2014 shared task on parsing morphologically-rich languages. In *Proceedings of the First Joint Workshop on Statistical Parsing of Morphologically Rich Languages and Syntactic Analysis of Non-Canonical Languages*, pages 103–109.

Hiroyuki Shindo, Yusuke Miyao, Akinori Fujino, and Masaaki Nagata. 2012. Bayesian symbol-refined tree substitution grammars for syntactic parsing. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers-Volume 1*, pages 440–448. Association for Computational Linguistics.

Richard Socher, John Bauer, Christopher D Manning, and Andrew Y Ng. 2013. Parsing with compositional vector grammars. In *ACL (1)*, pages 455–465.

Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. 2012. Lstm neural networks for language modeling. In *INTERSPEECH*, pages 194–197.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

Le Quang Thang, Hiroshi Noji, and Yusuke Miyao. 2015. Optimal shift-reduce constituent parsing with structured perceptron.

Oriol Vinyals, Łukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. 2015. Grammar as a foreign language. In *Advances in Neural Information Processing Systems*, pages 2755–2763.

Wenhui Wang and Baobao Chang. 2016. Graph-based dependency parsing with bidirectional lstm. In *Proceedings of ACL*.

Taro Watanabe and Eiichiro Sumita. 2015. Transition-based neural constituent parsing. *Proceedings of ACL-IJCNLP*.

David Weiss, Chris Alberti, Michael Collins, and Slav Petrov. 2015. Structured training for neural network transition-based parsing. In *Proceedings of ACL*.

Matthew D. Zeiler. 2012. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701.

Muhua Zhu, Yue Zhang, Wenliang Chen, Min Zhang, and Jingbo Zhu. 2013. Fast and accurate shift-reduce constituent parsing. In *ACL (1)*, pages 434–443.