AN ABSTRACT OF THE DISSERTATION OF

James Henry Cross III for the degree of Doctor of Philosophy in Computer Science
presented on December 8, 2016.

Title: Parsing with Recurrent Neural Networks

Abstract approved: 

Liang Huang

Machine learning models for natural language processing have traditionally relied on large numbers of discrete features, built up from atomic categories such as word forms and part-of-speech labels, which are considered completely distinct from each other. Recently however, the advent of dense feature representations coupled with deep learning techniques has led to powerful new models which can automatically learn to exploit various dimensions of implicit similarity between such discrete linguistic entities. This work extends that line of research as it applies to syntactic parsing, particularly by introducing recurrent network models which can encode the entirety of a sentence in context and by proposing novel parsing systems to take advantage of such models.

Syntactic parsing is an inherently difficult problem in natural language processing because of the ambiguous and highly compositional nature of language itself. Perfect agreement is not possible even among expert human annotators. Statistical and machine learning prediction of the syntactic structure of sentences has been the subject of
decades of study. Recent advances in applying deep neural models to language problems, however, have led to rapid strides in this domain, with models which are able to automatically exploit a whole new realm of hidden regularities in language. We continue this trend with feature-learning recurrent networks to model entire sentences, which allow the parser to incorporate information from the entire sentence context when making every decision. We also introduce new parsing paradigms designed explicitly to leverage this new representational power, including a state-of-the-art transition-based constituency parser, the first ever to achieve competitive results with greedy decoding.

We also introduce a straightforward dynamic oracle for the aforementioned constituency parsing system, and show that it is optimal in both label recall and precision. This is the first ever provably optimal dynamic oracle for a transition-based constituency parser. In addition to its optimality, our dynamic oracle is computable in amortized constant time per step, a dramatic improvement over its forerunners for arc-standard dependency parsing, which required worst-case cubic time per step.

Extending the optimality proof for that dynamic oracle, we show the surprising result that the entire space of possible parser states for a sentence of length $n$ can be reduced to $O(n^2)$ using a further simplified feature space. This simplification could have important future impact for search-based or globally-optimized training methods.

Finally, we extend our parsing model still further, by applying it to morphologically rich languages, using continuous embeddings over previously predicted morphological features. We find that we achieve very competitive results over a range of languages despite no language-specific architectural or hyper-parameter tuning, including achieving the best reported parsing results on the French Treebank.
Parsing with Recurrent Neural Networks

by

James Henry Cross III

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented December 8, 2016
Commencement June 2017

APPROVED:

__________________________________________________________________
Major Professor, representing Computer Science

__________________________________________________________________
Director of the School of Electrical Engineering and Computer Science

__________________________________________________________________
Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

__________________________________________________________________
James Henry Cross III, Author
ACKNOWLEDGEMENTS

First and foremost, I would like to thank my dissertation adviser Liang Huang, who has been an inspiration and from whom I have learned so much. I would also like to thank the other members of my Ph.D. committee, Prasad Tadepalli, Xiaoli Fern, Alan Fern, and Brett Tyler, for all of their valuable input, and for selflessly agreeing to serve on my committee on relatively short notice.

For incredible learning opportunities during industry internships, I would like to gratefully acknowledge my mentor Bing Xiang and manager Bowen Zhou at IBM, and my mentor Juan Miguel Pino and sponsor Joy Zhang at Facebook.

During my years working in Liang’s group, it has been a true pleasure to be colleagues and office mates with Kai Zhao, Mingbo Ma, Dezhong Deng, and Juneki Hong. I am also in eternal debt to my master’s degree advisers Irina Gladkova and Michael Grossberg.

Finally, I would like to thank my family, particularly my parents James and Gayle, for their unflagging support.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Background and Literature Review</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>2.1</td>
<td>Preliminaries</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>2.1.1</td>
<td>Parsing</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>2.1.2</td>
<td>Neural Networks</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>2.1.3</td>
<td>Word Embeddings</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>2.2</td>
<td>Recursive Neural Networks</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>2.2.1</td>
<td>Syntactic Parsing with Recursive Neural Networks</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>2.2.2</td>
<td>Compositional Vector Grammars</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>2.2.3</td>
<td>Recursive Neural Networks over Dependency Trees</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>2.3</td>
<td>Transition-Based Dependency Parsing</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>2.3.1</td>
<td>Neural Network Labeled Dependency Parsing</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>2.4</td>
<td>Recurrent Neural Networks</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>2.4.1</td>
<td>Long Short-Term Memory</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>2.4.2</td>
<td>Recurrent Networks for Parsing</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Incremental Parsing with Minimal Features Using Bi-Directional LSTM</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>3.1</td>
<td>Introduction</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>3.2</td>
<td>LSTM Position Features</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>3.3</td>
<td>Shift-Reduce Dependency Parsing</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>3.3.1</td>
<td>Hierarchical Classification</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>3.4</td>
<td>Shift-Promote-Adjoin Constituency Parsing</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>3.5</td>
<td>Experimental Results</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>3.5.1</td>
<td>Dependency Parsing: English and Chinese</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>3.5.2</td>
<td>Constituency Parsing: English &amp; Chinese</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>3.6</td>
<td>Related Work</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>3.7</td>
<td>Conclusions</td>
<td>50</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Span-Based Constituency Parsing with a Structure-Label System and Dynamic Oracles</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>4.1</td>
<td>Introduction</td>
<td>51</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>Parsing System</td>
<td>53</td>
</tr>
<tr>
<td>4.3</td>
<td>LSTM Span Features</td>
<td>57</td>
</tr>
<tr>
<td>4.4</td>
<td>Dynamic Oracle</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>4.4.1 Preliminaries and Notations</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>4.4.2 Structural and Label Oracles</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>4.4.3 Correctness</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>4.4.4 Implementation and Complexity</td>
<td>68</td>
</tr>
<tr>
<td>4.5</td>
<td>Related Work</td>
<td>68</td>
</tr>
<tr>
<td>4.6</td>
<td>Experiments</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>4.6.1 Training with Dynamic Oracle</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>4.6.2 Penn Treebank</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>4.6.3 French Treebank</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>4.6.4 Notes on Experiments</td>
<td>77</td>
</tr>
<tr>
<td>4.7</td>
<td>Conclusion and Future Work</td>
<td>77</td>
</tr>
<tr>
<td>5</td>
<td>Simplified Features for Span-Based Constituency Parsing</td>
<td>81</td>
</tr>
<tr>
<td>5.1</td>
<td>Feature Representation</td>
<td>82</td>
</tr>
<tr>
<td>5.2</td>
<td>Local Dynamic Oracle</td>
<td>84</td>
</tr>
<tr>
<td>5.3</td>
<td>Experimental Results</td>
<td>86</td>
</tr>
<tr>
<td>6</td>
<td>Search-Based and Extended Approaches</td>
<td>89</td>
</tr>
<tr>
<td>6.1</td>
<td>Beam-Search</td>
<td>89</td>
</tr>
<tr>
<td>6.2</td>
<td>Beam-Search with Dynamic Programming</td>
<td>92</td>
</tr>
<tr>
<td>6.3</td>
<td>Recursive Network Features</td>
<td>94</td>
</tr>
<tr>
<td>7</td>
<td>Summary</td>
<td>98</td>
</tr>
<tr>
<td>Bibliography</td>
<td></td>
<td>99</td>
</tr>
<tr>
<td>Appendices</td>
<td></td>
<td>107</td>
</tr>
<tr>
<td>A</td>
<td>Source Code for Span-Based Parser</td>
<td>108</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>An example of a labeled dependency tree. The arc labels indicate the type of syntactic relationship between each pair of head and dependent words.</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>An example constituency tree. Rules may have arbitrary arity, and this tree shows non-terminal phrase labels with one, two, and three children. Here the parts of speech are shown as parents of the words they describe, however they are sometimes treated as unified with the words at the leaf level.</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>A basic representation of a multilayer perceptron classifier with one hidden layer.</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>A recursive neural network for a small binary tree. The same weights $W$ are used at each internal node to combine two $n$-dimensional inputs and produce an $n$-dimensional output.</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>A labeled dependency tree for a short sentence.</td>
<td>16</td>
</tr>
<tr>
<td>2.4</td>
<td>An illustration of each of the possible actions for an unlabeled shift-reduce dependency parser (applied subsequently).</td>
<td>20</td>
</tr>
<tr>
<td>2.5</td>
<td>Atomic features used for neural network dependency parsing (blue, green, and red represent the three different sets of embeddings from which vector projections are drawn).</td>
<td>22</td>
</tr>
<tr>
<td>2.6</td>
<td>The basic architecture of a recurrent neural network layer. Shared weights produce $h_t$ from $h_{t-1}$ and $x_t$. These weights as well as $h_0$ are network parameters.</td>
<td>25</td>
</tr>
<tr>
<td>2.7</td>
<td>Architecture of an LSTM network for sequence-to-sequence translation</td>
<td>30</td>
</tr>
<tr>
<td>3.1</td>
<td>The sentence is modeled with an LSTM in each direction whose input vectors at each time step are word and part-of-speech tag embeddings.</td>
<td>35</td>
</tr>
<tr>
<td>3.2</td>
<td>In the 2-Layer architecture, the output of each LSTM layer is concatenated to create the positional feature vector.</td>
<td>37</td>
</tr>
</tbody>
</table>
**LIST OF FIGURES (Continued)**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3</td>
<td>The arc-standard dependency parsing system [42]. Stack $S$ is a list of heads, $j$ is the start index of the queue, and $s_0$ and $s_1$ are the top two head indices on $S$.</td>
</tr>
<tr>
<td>3.4</td>
<td>Our shift-promote-adjoin system for constituency parsing.</td>
</tr>
<tr>
<td>3.5</td>
<td>Shift-Promote-Adjoin parsing example. Upward and downward arrows indicate promote and (sister-)adjunction actions, respectively.</td>
</tr>
<tr>
<td>3.6</td>
<td>Recall on dependency arcs of various lengths in PTB dev set. The Bi-LSTM parser is particularly good at predicting longer arcs.</td>
</tr>
<tr>
<td>4.1</td>
<td>Deductive system for the Structure/Label transition parser. The stack $\sigma$ 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 $iXj$, which at the end of a parse, fully define a parse tree.</td>
</tr>
<tr>
<td>4.2</td>
<td>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.</td>
</tr>
<tr>
<td>4.3</td>
<td>Word spans are modeled by differences in LSTM output. Here the span $^3$ eating $^5$ fish is represented by the vector differences $(f_5 - f_3)$ and $(b_3 - b_5)$. The forward difference corresponds to LSTM-Minus [62].</td>
</tr>
<tr>
<td>4.4</td>
<td>Reachable brackets (w.r.t. gold tree in Fig. 4.1) for $c = \langle 10, [0, 1, 2, 4], {^0\text{NP}_1} \rangle$ which mistakenly combines “like eating”. Trapezoids indicate stack spans (the top one in red), and solid triangles denote reachable brackets, with $\text{left}(c)$ in blue and $\text{right}(c)$ in cyan. The next reachable bracket, $\text{next}(c) = ^1\text{VP}_5$, is in bold. Brackets $^3\text{VP}_5$ and $^3\text{S}_5$ (in dotted triangle) cross the top span (thus unreachable), and $^0\text{NP}_1$ is already recognized (thus not in $\text{reach}(c)$ either).</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES (Continued)

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5</td>
<td>Dynamic oracle with respect to the gold parse in Fig. 4.2. 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 $1_{VP_5}$ in all cases.</td>
<td>65</td>
</tr>
<tr>
<td>4.6</td>
<td>The optimal tree $t^*(c)$ adds all reachable brackets and nothing else. Note that $reach(c)$ and $t$ are disjoint.</td>
<td>67</td>
</tr>
<tr>
<td>5.1</td>
<td>An illustration of the spans used to make decisions in Cross and Huang (2016b) [15]. (a) Structural actions (here combine) are determined with 4 spans, while (b) label actions (here label-VP) require only 3 spans.</td>
<td>82</td>
</tr>
<tr>
<td>5.2</td>
<td>Simplified feature representations for the same states from the running example, used to make decisions for (a) structural actions, and (b) label actions.</td>
<td>84</td>
</tr>
<tr>
<td>6.1</td>
<td>Illustration of a combine action. Each stack span is represented by a vector of fixed dimensionality. For combine, the vectors for each of the top two stack spans are used to compute the vector for the new merged span.</td>
<td>95</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Feature templates. Note that, remarkably, even though we do labeled dependency parsing, we do not include arc label as features.</td>
<td>38</td>
</tr>
<tr>
<td>3.2 Development and test set results for shift-reduce dependency parser on Penn Treebank using only ((s_1, s_0, q_0)) positional features.</td>
<td>45</td>
</tr>
<tr>
<td>3.3 Ablation studies on PTB dev set (wsj 22). Forward and backward context, and part-of-speech input were all critical to strong performance.</td>
<td>46</td>
</tr>
<tr>
<td>3.4 Development and test set results for shift-reduce dependency parser on Penn Chinese Treebank (CTB-5) using only ((s_1, s_0, q_0)) position features (trained and tested with gold POS tags).</td>
<td>47</td>
</tr>
<tr>
<td>3.5 Test F-scores for constituency parsing on Penn Treebank and CTB-5.</td>
<td>48</td>
</tr>
<tr>
<td>3.6 Hyperparameters and training settings.</td>
<td>49</td>
</tr>
<tr>
<td>4.1 Features used for the parser. No label or tree-structure features are required.</td>
<td>60</td>
</tr>
<tr>
<td>4.2 Hyperparameters. °French only.</td>
<td>70</td>
</tr>
<tr>
<td>4.3 Comparison of performance on PTB development set using different oracle training approaches.</td>
<td>73</td>
</tr>
<tr>
<td>4.4 Comparison of performance of different parsers on PTB test set. °Neural. *Greedy. ‡External embeddings.</td>
<td>79</td>
</tr>
<tr>
<td>4.5 Results on French Treebank. °reranking, ‡external.</td>
<td>80</td>
</tr>
<tr>
<td>5.1 Comparison of performance on PTB development set with both the original and simplified feature sets.</td>
<td>87</td>
</tr>
<tr>
<td>5.2 Comparison of performance on PTB test set with both the original and simplified feature sets.</td>
<td>87</td>
</tr>
</tbody>
</table>
# LIST OF ALGORITHMS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Oracle with Exploration</td>
<td>72</td>
</tr>
<tr>
<td>Dynamic Oracle with Mixed Exploration</td>
<td>75</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

Parsing is a central problem in natural language processing. It refers to deducing the often quite complicated internal hierarchical structure of human language given only its surface form, typically a linear sequence of text. It is certainly not solvable in any rule-based form, and in fact there is not even perfect agreement among expert human annotators for certain complex language instances.

Statistical predictive parsing, the focus of this research, is a classic problem in supervised machine learning which has been the subject of a vast amount of research for decades. The recent advent of widely used deep learning models has led to impressive new results and has shown the way toward a more generalized framework where features representations are automatically learned and do not rely on linguistic intuitions. Such automatically learned features may be more robust, for instance to differences between languages or specific domains of application, and they may also be able to encode patterns which would be difficult or impossible for humans to discern.

We show that by exploiting the power of recurrent neural networks to encode essentially unbounded sentence context, this trend toward automatically learned feature representations can be extended still further to learn very accurate parsing models with extremely abstract representations of parser configuration. In this way, a neural model can learn all relevant aspects of language interaction directly from labeled data in a manner that can in principle be extended easily between languages and domains.
There are two main paradigms for syntactic sentence parsing, dependency parsing and constituency parsing, both of which are addressed in this work. A dependency tree is a graph where each word in the sentence is an internal or leaf node, and arcs indicate a syntactic relation between one word and another. These arcs may be further annotated with labels to indicate the nature of the syntactic relationship between the connected words. All words except one have exactly one parent. This word, typically the main verb of the sentence, is known as the “root”.

![Figure 1.1: An example of a labeled dependency tree. The arc labels indicate the type of syntactic relationship between each pair of head and dependent words.](image)

Constituency parsing on the other hand, often considered a more difficult problem, involves analyzing the sentence in a manner analogous to a formal grammar. Internal nodes are non-terminal labels which describe an entire (typically contiguous) phrase. Each such non-terminal label has one or more children, while the leaf nodes of the tree are the words of the sentence, annotated with their parts of speech.
Figure 1.2: An example constituency tree. Rules may have arbitrary arity, and this tree shows non-terminal phrase labels with one, two, and three children. Here the parts of speech are shown as parents of the words they describe, however they are sometimes treated as unified with the words at the leaf level.

We have demonstrated the power of such models with the development of multiple systems, achieving competitive, often state of the art results, across parsing paradigms and in parsing multiple languages. Chapter 3 describes the application of a bi-directional recurrent network to two different types of transition-based parsing systems: the well-known arc-standard approach to dependency parsing, and a novel system (“Shift-Promote-Adjoin”) which we developed for transition constituency parsing as part of this work. Chapter 4 describes yet another novel system for constituency parsing which was developed as part of this work, where stack elements are sentence spans rather than partial tree structures. Combined with a system of representing spans as the elementwise difference of recurrent network output vectors, this system achieves state-of-the-art results on both English and French treebanks, despite using strictly greedy decoding. Chapter 5 describes a modification of this parsing system where all parsing can be represented as a partition of the sentence into three spans. This simplification leads to equivalent results.
despite requiring significantly fewer network parameters, which we validate both theo-
retically and empirically. We have also conducted extensive experimentation attempting
to improve the results of the greedy span parsing with extensions such as search over
the space of action sequences and adding a recursive element, which we describe in
Chapter 6.

It should be noted that this dissertation is, in part, organized according to the “sta-
ple” approach, and that Chapter 3 and Chapter 4 are essentially updated and expanded
versions of previously published conference papers [14, 15].
Chapter 2: Background and Literature Review

2.1 Preliminaries

2.1.1 Parsing

In a linguistic context, parsing is the analysis of the relationships between parts of an utterance, typically the words in a sentence. Automatic parsing of natural language is an important task for many downstream NLP applications, and a great variety of machine learning approaches have been successfully applied to automating approximate solutions to this problem in its various forms. Prominent categories include applications of probabilistic grammars and sparse-feature learning models such as perceptron.

The exact methodology and output format of the syntactic parsing task can vary, but the two most common forms it takes are constituency parsing and dependency parsing. In constituency parsing (also known as phrase-structure parsing), the output is a tree where the leaves correspond to the words in the sentence, and each subtree represents a phrase, i.e., a continuous subsequence of words which can be considered atomically within the syntax of the sentence. Each internal node is typically also labeled with a symbol designating the grammatical function of the phrase it designates, and the tree conceptually describes how the sentence could be produced from a formal grammar for the language.

In dependency parsing, all of the nodes in the tree correspond to the words in the
sentence. Directed arcs link pairs of words and designate a head-modifier relationship between the words, with one word (the one without another head word in the sentence) designated the root of the sentence. In addition, the arcs may be labeled to further characterize the nature of the syntactic relationship between the words.

2.1.2 Neural Networks

Artificial neural networks (“NNs”) are a class of machine learning models originally inspired by the connections between neurons in the human brain. In general, they model potentially very complex functions (from vector to vector or vector to real value) by a series of “layers” each of which is a linear transformation followed by an elementwise non-linearity. This non-linearity is sometimes called the “activation function” by analogy to the activation of biological neurons under certain stimulation (input) conditions, where each term of the output vector would correspond to one neuron.

The linear transformation is defined by a weight matrix $W$ and bias term $b$ which are the parameters to be learned. The output of such a network layer with non-linear activation function $f$ and input vector $x$ is thus defined as:

$$y = f(Wx + b)$$ (2.1)

A typical use for such a network is in classification, where the input features are a real-valued vector, and the function modeled by the network is used to transform the input, which is then used as input to a standard classifier such as logistic regression. In such a case, the parameters of the (output layer) classifier are learned in conjunction
with the hidden layer weights, typically through back-propagated gradient descent to minimize a loss function.

In practice, typical choices for non-linear activation functions are the hyperbolic tangent or the logistic function. This is both because of their nice differentiability characteristics and because of their constrained behavior (mapping arbitrary real numbers into the space $[-1, 1]$ and the space $[0, 1]$ respectively). Recent work, however, has shown that better results may often be obtained using the much simpler linear rectifier function [22]:

$$f(x) = \begin{cases} 
  x & : x > 0 \\
  0 & : x \leq 0 
\end{cases} \quad (2.2)$$

This choice of function does have a non-differential point at 0, however, but this is not a problem for optimization, as any subgradient may be chosen when input values are at exactly this point.

The most basic form of such a network is the multilayer perceptron (“MLP”), which consists of one or more fully-connected hidden layers. An example of how such networks are often visualized can be seen in Figure 2.1. In this figure, each of the “cells” represents a single numerical value (the size of these vectors shown in figures is significantly reduced from the size actually used for illustrative purposes, a common practice). Here the input dimensionality is six and the size of the single hidden layer (i.e., the output dimensionality of Equation 2.1) is four. The arrows represent “connections” between cells realized by the weight matrix multiplication. For clarity of expression, not all of the connections are shown between the input and hidden layer here. Unless other-
Figure 2.1: A basic representation of a multilayer perceptron classifier with one hidden layer.

wise specified, all connections between layers should be assumed to be full connections, meaning the weight matrix is not restricted to be sparse and each input value affects each output value. In more complex diagrams, such full connections are often represented by a single arrow.

2.1.3 Word Embeddings

Problems in natural language processing are typically most easily conceived in terms of discrete features, such as words or n-grams of words. In a machine learning context, such features are often represented by binary-valued (or sometimes count-valued)
vectors of very high dimensionality. Features of this type are generally unsuitable as inputs to neural networks, however, because their sparsity makes learning intractable. Moreover, a well-chosen relatively low-dimensional continuous representation of such discrete units could have the additional advantage of encoding various dimensions of similarity between the discrete entities. These vectors are especially appropriate as neural network inputs, since the highly non-linear function represented by the network can learn to exploit the hidden relationships encoded by the vector space as it is optimized.

Because of these advantages, such representations for words have been explored for years and are known colloquially as “word embeddings”. Many different approaches have been taken to learn vector representations of words directly. Methods trained from large text examples by relying directly or indirectly on word co-occurrence counts date back to latent semantic analysis [16]. Recent examples include the window-based approach of Mikolov et al. [40] and the GloVe method [45]. There has also been work incorporating prior knowledge with bag-of-words context [67].

Because of these advantages, both practical and theoretical, words are generally represented as vectors when they are inputs to neural networks for NLP tasks. In some sense, they may be thought of as additional network parameters (in which case the features could be thought of as consisting of unique indices for words), where there is a projection layer “resolving” these indices to their respective word vectors. Seen this way, it is natural to think that the word vectors may be learned together with other network weights, through back-propagation, and this is in fact a common practice. However, it is also common to initialize these vectors with values learned from one of the well-known embedding methods described above, and researchers have frequently found that this
leads to faster learning and better results than random initialization.

It should also be noted that continuous vector representations can also be used to model other traditionally discrete features, such as parts of speech. This technique was successfully used for part-of-speech tags and dependency arc labels in Chen and Manning (2014) [7] and has become a common practice. The actual values are more likely to be randomly initialized and learned along with the network weights in such cases.

2.2 Recursive Neural Networks

Natural languages are well known to have compositional properties which can be represented by recursive structures. This is especially evident in the context of parsing, where the tree-structured output can be thought of as consisting of many instantiations of the same type of relationship patterns. In this way, each arc in a dependency tree can be thought of as an individual instance of the head-modifier relationship, and each interior node in a syntax tree can be thought of as representing a grammar rule combining several constituents into a single larger phrase. This led to the idea of using recursive neural networks not only to model but to predict this structure by replicating the same neural network architecture at each such (potential) point in a tree.

2.2.1 Syntactic Parsing with Recursive Neural Networks

Socher et al. first introduced the idea of using recursive neural networks for syntax parsing in 2010 [52]. The idea is that the neural network calculation flows from the
bottom of the tree upwards toward the root, with the same neural network architecture replicated at each internal node. This repeated neural network takes as input vector representations of each of the child nodes, and yields an output a vector representation of the parent node. Though this model could be applied to directed acyclic graphs generally, we will follow the convention of that paper and present the simplified example of binary trees.

![Recursive Neural Network](image)

Figure 2.2: A recursive neural network for a small binary tree. The same weights $W$ are used at each internal node to combine two $n$-dimensional inputs and produce an $n$-dimensional output.

A small example of such a network can be seen in Figure 2.2. Note the weights $W$ are shared throughout the tree and that every node has a representation of the same dimensionality $n$. In the most basic set-up, each of the child representations is concatenated together to form a vector $[x_1; x_2]$ of size $2n \times 1$. The weights $W$ are of size $n \times 2n$, and a hyperbolic tangent non-linearity is applied to the linear combination, giving the
following formula for the parent representation:

\[ p = \tanh(W[x_1; x_2] + b) \]  

(2.3)

Note that at the leaf level, the inputs to the network are vector representations of words. These can be learned for each word in the vocabulary, together with network weights, when training the network from a source of known trees. They can also be initialized using a vector training method as described in Section 2.1.3, as the authors did for their experiments.

Of course, in this form the network only describes how existing trees may be processed, not how to compare different potential parses, so some additional scoring mechanism is required. In the most basic model, (“Greedy RNN”), each potential parent representation is numerically evaluated for “validity” via inner product with a row vector \( W^{score} \in \mathbb{R}^{1 \times n} \), according to the following formula:

\[ s_{1,2} = W^{score} p \]  

(2.4)

At parse time, all adjacent pairs are evaluated, then the highest-scoring one is taken to be valid, and those elements are combined into a single phrase representation. This is repeated until there is one representation for the entire sentence, and those combinations (initially of words, later including phrases) define the tree structure.

One important addition to this model is the consideration of context (“Greedy Context-Sensitive RNN”), which adds the vector representation of adjacent words in the sentence as inputs to Equation 2.3 (thus also changing the dimensionality of the weight matrix).
This is important since sentence context obviously influences whether two words or phrases should be considered a unit in a given sentence.

Further improvement can be made by also adding a softmax classification layer independently on top of each interior-node instantiation of the network, i.e., each parent node representation ("Greedy Context-Sensitive RNN and Category Classifier"). This allows to the network to exploit nodes with discrete labels, such as non-terminal labels for the Penn Treebank, to improve network learning by backpropagating the cross-entropy error of the softmax layer throughout the entire tree.

Finally, rather than greedily collapsing the two nodes with the best independent score at each step, a model that considers also possible trees in proposed, where sentences are parsed using a CKY-style algorithm ("Global Context-Sensitive RNN and Category Classifier"). The global learning objective given a set of training (sentence, tree) pairs \((x_i, y_i)\) is to maximize:

\[
J = \sum_i s(x_i, y_i) - \max_{y \in A(x_i)} (s(x_i, y) + \Delta(y, y_i))
\]  

Here, \(A(x)\) is the set of all possible trees that can be constructed from sentence \(x\). \(s(x, y)\) is a tree-scoring function which amounts to the sum of all of the individual node scores corresponding to Equation 2.4. \(\Delta\) is a structure-loss function which amounts to adding a fixed penalty for each span in the first tree which is not in the second.

The objective \(J\) is maximized using the subgradient method since the objective is not strictly differentiable. This involves computing the current maximum-scoring tree, \(y_{\text{max}}\) for each sentence, which is done using CKY-style parsing. The subgradient over
an entire set of trees for any given parameter $W$ is given by:

$$\frac{\partial J}{\partial W} = \sum_i \frac{\partial s(x_i, y_i)}{\partial W} - \frac{\partial s(x_i, y_{\text{max}})}{\partial W}$$ (2.6)

It is also intriguing that it learns representations of every phrase in the tree, up to and including the full sentence, all of which contain syntactic, and possibly also semantic information. This notion gave rise to applying a recursive neural network architecture to other tasks such as sentiment classification [53]. Such classification over an existing natural-language tree structure was later further extended by applying a “deep” architectural element, in effect propagating network values along yet another feed-forward dimension (in some sense “within” each tree node) as well as up the tree structure, forming a deep recursive neural network [32].

This algorithm was later also generalized to “parse” natural scenes in images, where the tree structure represents breaking down elements of the scene in part-of-whole or adjacency relations [51]. It resulted in state-of-the-art performance when applied to established image processing tasks such as segmentation and scene classification.

### 2.2.2 Compositional Vector Grammars

This idea was subsequently combined with aspects of probabilistic context-free grammar (PCFG) parsing to create an approach known as Compositional Vector Grammars [50]. The approach is similar to above, but it also relies heavily on discrete syntactic categories, specifically parts of speech at the word level, and phrasal categories for internal nodes (such as “NP” for noun phrase, etc.). Not only does the recursive neural network
itself take into account these categories, but it also uses a full probabilistic grammar, which assigns a probability $P(A \rightarrow BC)$ to each rule, which is the probability of having the parent label $A$ given the child labels $B$ and $C$.

In addition to this, a different weight matrix (i.e., network instantiation) is applied at each internal node depending on the syntactic labels of the child nodes, so two vectors with labels $B$ and $C$ would be combined with weight matrix $W^{(B,C)}$. This is justifiable for both linguistic and practical reasons. Naturally, it stands to reason that a more nuanced model could be realized by conditioning the means of combining constituents based on the child labels. For one glaring example, in certain cases, such as a determiner and a noun phrase, or an independent clause and a punctuation mark, it is clear that one child should dominate in the determination of the parent representation, whereas the same is not true in other cases. On the practical side, this approach makes the network much easier to train since the same parameter is not replicated over and over again throughout the network.

The training algorithm used to exploit this architecture is two-stage. First, a full PCFG is determined, which assigns a probability $P(X \rightarrow YZ)$ to each valid rule $X \rightarrow YZ$ using statistical counts on the training trees. The neural network is then trained using back-propagation through structure in a manner similar to that described in the previous section, except that the score for each internal node (decision point) is given by:

$$s(p) = W^{(B,C)}_{score} p + \log P(A \rightarrow BC)$$

(2.7)
where $W_{score}^{(B,C)}$ is a row vector like $W_{score}$ from Equation 2.4 but, like the recursive weight matrices, is dependent on the labels of the child nodes and $P(A \rightarrow BC)$ is the probability from the PCFG.

### 2.2.3 Recursive Neural Networks over Dependency Trees

A recursive network architecture can also be applied to other types of structures, such as dependency trees, where each node in the tree represents a word, and the directed arcs represent various types of binary syntactic relations between words. This tractability of this approach was recently demonstrated by application to a factoid question-answering systems, where the questions each consisted of several sentences whose dependency trees were processed in this manner [33]. We present a basic outline of the network used for this purpose.

![Figure 2.3: A labeled dependency tree for a short sentence.](image)

Consider the simple dependency tree in Figure 2.3. Note that each word is the head
word of some continuous phrase in the sentence. The leaves “The” and “cold” are self-contained, but “wind” is the head of the noun phrase “The cold wind”, and the verb is the head of the entire sentence. The recursive neural network is designed to learn a hidden vector representation $h$ for the phrase headed by each word in the dependency tree.

As with the syntax trees previously discussed, values proceed through the network in a bottom-up fashion, beginning with the leaves. A single set of shared weights is used to process all word vectors, thus at the leaf this phrase representation consists only of applying these weights and a non-linear activation function. For example, the hidden representation of the leaf word “cold” in Figure 2.3 is:

$$h_{\text{cold}} = f(W_v \cdot x_{\text{cold}} + b)$$  \hspace{1cm} (2.8)

where $x_{\text{cold}}$ is the word representation for “cold”, initialized as we have seen before.

The important innovation is how multiple internal hidden representations are computed. There are two important issues: how to generalize over an arbitrary number of children, and how to leverage the information provided by the arc label (which is especially important when trying to establish a semantic representation, as here, given that syntax is crucial in determining the relative importance of different parts of the sentence). The authors’ successful solution is to apply a linear transformation to each child representation depending on the syntactic relation. Thus, for each possible arc label there is a different weight matrix $W_R$ shared by all instances of that relation in all trees.

---

1The “continuous” property is only true because this tree is projective (i.e., has no crossing arcs). The described algorithm would remain the same for non-projective trees, but a “phrase” might skip some words from the sentence as a whole.
The same weight matrix $W_v$ used for leaf words is applied to the parent at the internal node, and the vectors thus obtained for the parent and all children are combined additively (together with the universal bias vector $b$) inside the non-linear activation. Using Figure 2.3 as an example once more, the hidden representation for “wind” is:

$$h_{wind} = f(W_v \cdot x_{wind} + W_{DET} \cdot h_{the} + W_{AMOD} \cdot h_{cold} + b)$$

(2.9)

where $h_{the}$ and $h_{cold}$ are the representations recursively derived from Equation 2.8.

In the cited work, this network structure is trained on existing dependency trees, and used to generate semantic representations of question sentences (and the phrases they contain) which are then compared to representations in the same space for candidate answers (with word/entity embeddings trained concurrently) on the theory that different parts of the sentence may be the most important in different situations.

There is no reason in principle, however, that such an approach could not be extended to structure prediction, i.e., parsing. In practice, however, transition-based approaches are more common, because they have been shown to perform with very strong accuracy without the necessity of exploring the entire search space of possible trees. Recursive networks, in the form of a modified compositional vector framework, have in fact been combined with this type of parsing, though the results have so far lagged behind the state of the art [55]. In the next section, we will see that even a much simpler neural network architecture can yield impressive results on transition-based dependency parsing.
2.3 Transition-Based Dependency Parsing

Dependency parsing has proven over the years to be much more tractable than syntax parsing using local information only. As such, there many successful examples of using linear-time algorithms to achieve competitive results in this domain (see. e.g., [71]. (As a side note, this may be somewhat related to the way in which humans rapidly make sense of natural language, since of course a full grammatical deconstruction is not necessary or desirable to understand a sentence when having a conversation.)

This relatively efficient approach is exemplified by the so-called shift-reduce parsing algorithm for producing projective dependency trees. It utilizes two data structures: a queue of yet-to-be-processed words, and a stack of partially-constructed dependency trees. At the beginning, the stack is empty, and the queue consists of all of the words in the sentence to be parsed, beginning with the first. At each step, the parser takes one of three actions: \textit{shift}, \textit{left-reduce}, or \textit{right-reduce}.

A \textit{shift} action means popping the front element of the queue and pushing it onto the stack as a new single-element tree. A \textit{left-reduce} action requires popping the top two trees from the stack, connecting them via an arc from the right one (top element) to the left one (second element), then pushing the connected tree back onto the stack. The \textit{right-reduce} action is similar, but the new arc goes in the other direction (see Figure 2.4. Notice that sentence order is preserved among head words of the trees in the stack.\footnote{Internally, such parsers also preserve the order of children for each parent node, distinguishing between “left children” and “right children” so that the entire sentence order is preserved.}

A typical machine learning approach to this style of parser involves a feature representation of the entire current state of the parser (i.e., the current contents of the stack.
Figure 2.4: An illustration of each of the possible actions for an unlabeled shift-reduce dependency parser (applied subsequently).

and queue). It is local in the sense that only the top few elements of the stack and queue are considered (typically three each). The features used have traditionally consisted of the words and parts-of-speech for the sentence to be parsed.

Sparse linear models such as perceptron have been very successfully applied to this problem in the unlabeled context, but require hand-engineered concatenations of the words and POS tags that occur in certain positions in the current parser configuration. For example, in addition to the words and POS tags of the top three elements of the
queue and the heads of the top three elements of the stack, many bigram and trigram
concatenations are very important to consider, such as the head word of the top tree in
the stack, its part of speech, and the part of speech of the head word of the next tree in
the stack (see, e.g., [31].

Labeled dependency parsing imposes an additional layer of complexity in that the
type of syntactic relationship between the head word and its modifier is further classified
by a discrete label. In terms of the shift-reduce parsing model, this means that each left-reduce
or right-reduce parser action is further subdivided according to the identity of the
label assigned.

2.3.1 Neural Network Labeled Dependency Parsing

A relatively simple neural network architecture was successfully applied to the problem
of labeled dependency parsing by Chen et al. (2014) [7]. This approach relies on the
intuition that rather than hand-engineering may concatenations of discrete features, a
neural network classifier could be trained to combine them in the most relevant way for
each parser action. In this way, the network learns which combinations of features are
important at a given moment.

In particular, the “atomic features” of the labeled parser state fall into three cat-
egories: words in the natural language vocabulary, part-of-speech tags, and previously
assigned labels in the arcs near the top of the two top trees on the stack. To form a viable
neural network input, vector representations need to be learned for each of these sets of
discrete categories. For the words in the natural language vocabulary, they can be initial-
ized, as before, according to one of the well-known methods described in Section 2.1.3. The vector representations for POS tags and arc labels are learned together with the network weights during parser training using AdaGrad [17] and random dropout [54].

![Figure 2.5: Atomic features used for neural network dependency parsing (blue, green, and red represent the three different sets of embeddings from which vector projections are drawn).](image)

The discrete features used to represent each feature state include the top three words on the stack and their POS tags, as well as the same for the head words of the top tree trees on the stack. Special vectors `<s>` and `</s>` are learned in each embedding set to represent those instances when there are less than three elements on the stack or queue, respectively.

In addition, the top two trees on the stack are modeled much more extensively, given that these are the two trees that would be combined by a reduce action. Features extracted from those two trees include the words and parts of speech of the left-most and right-most children of each, as well as the “incoming” arc label for each of those child
nodes. All three of these features are also used for the second-left-most and second-right-most child of each of these trees. Finally, the right-most child of right-most child (and the same on the left side) is also extracted. A visualization of these atomic features can be seen in Figure 2.5.

Classification from state representation to parser action is done from a straightforward one-hidden-layer multilayer perceptron of the type depicted in Figure 2.1.2. In the referenced work, all three sets of vectors are represented in the same dimensionality $n$ (50 in the experiments), and all 48 vectors are concatenated together to form the input to the hidden layer (dimension 200 in the experiments).

The work also introduces a novel cubic activation function, so that the hidden layer is calculated as:

$$h = f(Wx + b)^3$$  \hspace{1cm} (2.10)

This is purported to be of particular importance in the parsing context given the importance of considering trigrams of atomic features, since the activation function essentially combines products of all individual feature-dimensions taken three at a time (including repeating such dimensions in the product).

Classification is done through a softmax classification layer with the important wrinkle that only possible parser actions are included in the normalization term. This is because a shift-reduce parser cannot perform a shift action (otherwise a very likely action) once the queue is exhausted.

Training this model involves parsing all of the known training trees using a canonical
sequence (short-stack preference, i.e., performing a reduce action as soon as possible to lead to the gold tree), while extracting the values for all of the atomic features at each step. This yields a large corpus of training examples containing parser states and correct actions. A long-period training strategy is employed, wherein at each iteration a large selection of such (state, action) pairs is selected, regardless of origin sentence (100,000 in the published experiments).

To speed up parsing at application time, the authors also introduce a pre-computation trick, wherein hidden-layer components for individual atomic feature selections (complete with cubic activation function) are computed for the most commonly-occurring atomic feature positions. This is effective because may such features are likely to occur in the same position repeatedly, and in that case hidden-layer computation only requires summing these values with the fully-computed components from those features which are not so cached.

The end result is a parser which is extremely fast and very accurate, scoring 92.2 in terms of unlabeled attachment score (“UAS”). It is also noteworthy that this algorithm learns continuous representations from scratch for part-of-speech tags and the grammatical relationships represented by dependency arcs. Low-dimension visualizations show that these vectors capture relationships between these labels as might be expected, such as that similar parts of speech are clustered together (e.g., nouns, plural nouns, proper nouns, etc.).

---

3Total percentage of words in the test corpus which are assigned their correct head by the dependency parser
2.4 Recurrent Neural Networks

Another powerful instance of a neural network architecture is the recurrent neural network. In this case, a shared network architecture is applied repeatedly along a number of time steps, with part of the input at each step being produced by the previous time step. Since the network weights are replicated along a single “time” dimension, recurrent networks can in some sense be considered a special case of recursive network, where the directed acyclic graph is a cascade with one recursive input and one new (leaf) input at each step. Nevertheless, the idea of “time steps” is a useful conceptual framework, which allows one to describe the recurrent connection as “remembering” selected aspects of the earlier inputs.

![Recurrent Neural Network Diagram](image)

Figure 2.6: The basic architecture of a recurrent neural network layer. Shared weights produce $h_t$ from $h_{t-1}$ and $x_t$. These weights as well as $h_0$ are network parameters.

The general structure of a recurrent neural network layer can be seen in Figure 2.6. The input to the layer is a vector sequence $(x_1, x_2, x_3, ...)$. At each step, a new hidden
vector value $h_t$ is produced from the previous hidden vector $h_{t-1}$ and the current input $x_t$. In its most essential, fully-connected and unconstrained form, this calculation would take the following form with non-linear activation function $f$:

$$h_t = f(W x_t + U h_{t-1} + b)$$

(2.11)

where the weight matrices $W$ and $U$ and the bias $b$ are shared across the entire layer. The “previous” hidden value supplied at the first step, $h_0$, is also a network parameter. Depending on the structure of the task for which the network is designed, the hidden value produced by each step may be used as input to another layer above, for example another recurrent layer of similar design, or a softmax classifier for a sequence labeling task. On the other hand, in some applications, only the final vector produced is taken as the output of the network, and thus it represents the entire sequence of input.

Whatever the exact use of the output, the network weights are learned using back-propagation through time (“BPTT”). This means that the gradient of the error resulting from the output $h_3$ in the network in Figure 2.6 will be accumulated as it affects the network weights at time steps 1 and 2, as well as the initial value $h_0$, which in theory allows the network to learn long-term dependencies over the sequence, thus extracting value from the recurrent connection.

Unfortunately, as has been well-documented empirically, and explained theoretically, training networks with this architecture suffers from two related problems: vanishing and exploding gradients [44]. In the former, gradients from error at one time step quickly dissipate as you step backwards through the network, so that there is essen-
tially no contribution from applications of network weights even a few time steps away, effectively preventing the network from learning to exploit information from previous time steps. In the latter case, gradients blow up exponentially as you step back through time. Though exploding gradients are rarer, they completely destroy any learning in the network.

A number of approaches have been adopted to successfully limit the extent of these problems. The exploding gradient problem, since it arises only infrequently in typical network training can be successfully addressed by simply clipping the gradients when they exceed some pre-determined threshold [44]. Vanishing gradients, or the problem of learning long-term dependencies, have often been tackled with more advanced architecture relying on memory and gating, especially long short-term memory, discussed in more detail below. Other recent advances in successfully dealing with this problem include artificially constraining network weights, to require some hidden units to change slowly by keeping certain recurrent connections close to the identity matrix [39].

2.4.1 Long Short-Term Memory

A powerful architectural solution to the problem of learning dependencies over large numbers of time steps in a recurrent network was introduced some time before the current wave of renewed interest in neural networks: long short-term memory (“LSTM”) networks [28]. In addition (but related) to the output at each step, an LSTM explicitly models a “memory” cell, $C_t$.

The crucial innovation is that the network also learns three “gate” functions: the
input gate, the forget gate, and the output gate. These are each dependent on the hidden output of the previous time step and the current input, just as in Equation 2.11, with each having its own set of weights. These control, respectively, how much the next memory state will be influenced by the new input, how much it will be influenced by the previous memory contents, and how much of the memory will be released as output to the next state.

Each of these gate functions is used to weight vector values elementwise, so element values in [0, 1] are desired. Because of this, the activation function for the gates is the logistic sigmoid function:

\[ \sigma(x) = \frac{e^x}{e^x + 1} \] (2.12)

Thus, the values for the three gate functions (input, forget, and output) are computed as follows:

\[ i_t = \sigma(W_ix_t + U_ih_{t-1} + b_i) \] (2.13)

\[ f_t = \sigma(W_fx_t + U_fh_{t-1} + b_f) \] (2.14)

\[ o_t = \sigma(W_ox_t + U_oh_{t-1} + b_o) \] (2.15)

Meanwhile a “candidate” new memory cell value is computed, using the more common hyperbolic tangent activation function:
\[ \tilde{C}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c) \] (2.16)

The new memory cell value is then computed by combining the candidate value weighted by the input gate (using element-wise multiplication \( \odot \)) and the previous value weighted by the forget gate:

\[ C_t = i_t \odot \tilde{C}_t + f_t \odot C_{t-1} \] (2.17)

Finally, another non-linearity is applied to new memory contents, and the output gate determines in a similar manner how much of the result is made “visible” as output (both for the next recurrent step and any feedforward application):

\[ h_t = o_t \odot \tanh(C_t) \] (2.18)

Though this machinery complicates the network architecture considerably, it has been shown repeatedly to yield very impressive results, and is able to effectively learn even very long-term dependencies in a sequence. Since the weights for the gates are learned together with the memory cell activation weights, the network can learn which aspects of past input are important to remember and which are not.

With renewed interest in this type of network, there has also been much recent work in developing somewhat simpler architectures that can achieve the same results. One particularly successful very recent effort in this direction that arose from work in machine translation is gated recurrent units (“GRU”) [9, 10]. It is somewhat similar in spirit, but only involves two gates (update and reset) and does not maintain explicit
memory between steps (other than the recurrent output $h_t$).

2.4.2 Recurrent Networks for Parsing

Recurrent neural networks have been very successfully applied to a number of NLP tasks, especially language modelling, which is a sequence prediction task which could potentially have a staggering number of long-term dependencies [41].

![Figure 2.7: Architecture of an LSTM network for sequence-to-sequence translation](image)

A recent breakthrough from a group of researchers at Google demonstrated that LSTM could be used for sequence-to-learning and actually produce good results for machine translation [57]. This is especially impressive since machine translation normally requires so many finely tuned components, such as training input alignment and explicit language modeling (though this result is for English-to-French, which has relatively little reordering).

The structure of the network can be seen in Figure 2.7. Note that though the figure
shows two LSTM layers, four were used in the actual experimentation, thus a somewhat “deep” architecture. Each layer produces output which is also part of the sequence input to the next higher layer. The authors used hidden layer cell and word embeddings of dimensionality 1000.

The network sees the entire input sequence before it is used to determine output words, which means the entire sentence is encoded in the vector at the time the input is read. At this point, after the end-of-sentence marker <EOS> is seen, it is used to produce output in the target language. At each step, the previously produced target language word is included as input (together with the recurrent connection). Reversing the order of the input sentence was seen to help performance because it introduces relatively short-term dependencies near the “border” between input and output. Output is produced in the target language until <EOS> is produced.

The exact same sequence-to-sequence LSTM approach was later applied to syntactic parsing, essentially framing it as a translation problem [61]. This was done by using a reversible linearized representation of syntax parse trees as the target language. This is essentially the parenthesized tree format containing phrase and part-of-speech labels as the elements.

The authors also offer an improvement on this basic model with a “stack” strategy employed during decoding. After the input sequence is consumed, a stack of the words in the input sentence is maintained, and at each step (during decoding) the network receives the current top word on the stack as an additional input. The network may produce an additional output symbol, ⊥, which results in a word on the stack being popped. It is trained to produce this symbol upon reaching the common ancestor of
the top two words on the stack (i.e., when it needs to start producing tree symbols corresponding to the next word).

Though it did not reach state-of-the-art results, that work was among the first to apply a multi-layer recurrent network to parsing. The remainder of this dissertation will show how we have improved upon this foundation by integrating a recurrent sentence representation into discriminative models for left-to-right transition-based parsing systems.
Chapter 3: Incremental Parsing with Minimal Features Using Bi-Directional LSTM

This chapter describes the work in Cross and Huang (2016) [14].

Recently, neural network approaches for parsing have largely automated the combination of individual features, but still rely on (often a larger number of) atomic features created from human linguistic intuition, and potentially omitting important global context. To further reduce feature engineering to the bare minimum, we use bi-directional LSTM sentence representations to model a parser state with only three sentence positions, which automatically identifies important aspects of the entire sentence. This model achieves state-of-the-art results among greedy dependency parsers for English. We also introduce a novel transition system for constituency parsing which does not require binarization, and together with the above architecture, achieves state-of-the-art results among greedy parsers for both English and Chinese.

3.1 Introduction

Recently, neural network-based parsers have become popular, with the promise of reducing the burden of manual feature engineering. For example, Chen and Manning (2014) [7] and subsequent work replace the huge amount of manual feature combinations in non-neural network efforts [43, 70] by vector embeddings of the atomic features.
However, this approach has two related limitations. First, it still depends on a large number of carefully designed atomic features. For example, Chen and Manning (2014) [7] and subsequent work such as Weiss et al. (2015) [65] use 48 atomic features from Zhang and Nivre (2011) [70], including select third-order dependencies.

More importantly, this approach inevitably leaves out some nonlocal information which could be useful. In particular, though such a model can exploit similarities between words and other embedded categories, and learn interactions among those atomic features, it cannot exploit any other details of the text.

We aim to reduce the need for manual induction of atomic features to the bare minimum by using bi-directional recurrent neural networks to automatically learn context-sensitive representations for each word in the sentence. This approach allows the model to learn arbitrary patterns from the entire sentence, effectively extending the generalization power of embedding individual words to longer sequences. Since such a feature representation is less dependent on earlier parser decisions, it is also more resilient to local mistakes.

With just three positional features, stack top, second stack top, and queue head, we can build a greedy shift-reduce dependency parser that is on par with the most accurate parser in the published literature for English Treebank. This effort is similar in motivation to the stack-LSTM of Dyer et al. (2015) [19], but uses a much simpler architecture.

We also extend this model to predict phrase-structure trees with a novel shift-promote-adjoin system tailored to greedy constituency parsing, and with just two more positional features (defining tree span) and nonterminal label embeddings we achieve the most accurate greedy constituency parser for both English and Chinese.
3.2 LSTM Position Features

Figure 3.1: The sentence is modeled with an LSTM in each direction whose input vectors at each time step are word and part-of-speech tag embeddings.

The central idea behind our approach is exploiting the power of recurrent neural networks to let the model decide what aspects of sentence context are important to making parsing decisions, rather than relying on fallible linguistic information (which moreover requires leaving out information which could be useful). In particular, we model an input sentence using Long Short-Term Memory networks (LSTM), which have made a recent resurgence after being initially formulated by Hochreiter and Schmidhuber (1997) [28].

The input at each time step is simply a vector representing the word, in this case an embedding for the word form and one for the part-of-speech tag. These embeddings are learned from random initialization together with other network parameters in this work. In our initial experiments, we used one LSTM layer in each direction (forward and backward), and then concatenated the output at each time step to represent that sentence
position: that word in the entire context of the sentence. This network is illustrated in Figure 3.1. Here, \( w_i \) and \( t_i \) are the word and part-of-speech embeddings respectively of the \( i \)th word in the sentence. Their concatenation forms the input to the recurrent networks in both the forward and backward directions. The position feature vector at step \( i \) is in turn the concatenation of the output at that time step from the network in each direction, \( f_i \) and \( b_i \).
Figure 3.2: In the 2-Layer architecture, the output of each LSTM layer is concatenated to create the positional feature vector.

It is also common to stack multiple such LSTM layers, where the output of the forward and backward networks at one layer are concatenated to form the input to the next. We found that parsing performance could be improved by using two bi-directional LSTM layers in this manner, and concatenating the output of both layers as the positional feature representation, which becomes the input to the fully-connected layer.
<table>
<thead>
<tr>
<th>positional</th>
<th>dependency</th>
<th>constituency</th>
</tr>
</thead>
<tbody>
<tr>
<td>labels</td>
<td>$s_1, s_0, q_0$</td>
<td>$s_1, s_0, q_0, s_1.left, s_0.left$</td>
</tr>
</tbody>
</table>

Table 3.1: Feature templates. Note that, remarkably, even though we do labeled dependency parsing, we do not include arc label as features.

This architecture is shown in Figure 3.2.

The input to the network at each time step is $w_i; t_i$, the concatenation of the word and part-of-speech embedding for the $i$th word. The input to the second LSTM layer (both forward and backward) at time step $i$ is the concatenation of the output vectors at that step from the first layer network in each direction, $f^1_i; b^1_i$. The final output for each step is the concatenation of the output vectors in each direction and at both levels, $h_i = f^1_i; b^1_i; f^2_i; b^2_i$.

Intuitively, this represents the sentence position by the word in the context of the sentence up to that point and the sentence after that point in the first layer, as well as modeling the “higher-order” interactions between parts of the sentence in the second layer. In Section 3.5 we report results using only one LSTM layer (“Bi-LSTM”) as well as with two layers where output from each layer is used as part of the positional feature (“2-Layer Bi-LSTM”).
input: \( w_0 \ldots w_{n-1} \)

axiom \( \langle \epsilon, 0 \rangle: \emptyset \)

sh \( \frac{\langle S, j \rangle: A}{\langle S|j, j+1 \rangle: A} \quad j < n \)

re_{\downarrow} \quad \frac{\langle S|s_1|s_0, j \rangle: A}{\langle S|s_0, j \rangle: A \cup \{s_1 \downarrow s_0\}}

re_{\uparrow} \quad \frac{\langle S|s_1|s_0, j \rangle: A}{\langle S|s_1, j \rangle: A \cup \{s_1 \uparrow s_0\}}

goal \quad \langle s_0, n \rangle: A

Figure 3.3: The arc-standard dependency parsing system [42]. Stack \( S \) is a list of heads, \( j \) is the start index of the queue, and \( s_0 \) and \( s_1 \) are the top two head indices on \( S \).

3.3 Shift-Reduce Dependency Parsing

We use the arc-standard system for dependency parsing (see Figure 3.4). By exploiting the LSTM architecture to encode context, we found that we were able to achieve competitive results using only three sentence-position features to model parser state: the head word of each of the top two trees on the stack (\( s_0 \) and \( s_1 \)), and the next word in the queue (\( q_0 \)); see Table 3.1.

The usefulness of the head words on the stack is clear enough, since those are the two words that are linked by a dependency when taking a reduce action. The next incoming word on the queue is also important because the top tree on the stack should not be reduced if it still has children which have not yet been shifted. That feature thus allows the model to learn to delay a right-reduce until the top tree on the stack is fully formed,
shifting instead.

3.3.1 Hierarchical Classification

The structure of our network model after computing positional features is fairly straightforward and similar to previous neural-network parsing approaches such as Chen and Manning (2014) [7] and Weiss et al. (2015) [65]. It consists of a multilayer perceptron using a single ReLU hidden layer followed by a linear classifier over the action space, with the training objective being negative log softmax.

We found that performance could be improved, however, by factoring out the decision over structural actions (i.e., shift, left-reduce, or right-reduce) and the decision of which arc label to assign upon a reduce. We therefore use separate classifiers for those decisions, each with its own fully-connected hidden and output layers but sharing the underlying recurrent architecture. This structure was used for the results reported in Section 3.5, and it is referred to as “Hierarchical Actions” when compared against a single action classifier in Table 3.3.

3.4 Shift-Promote-Adjoin Constituency Parsing

To further demonstrate the advantage of our idea of minimal features with bidirectional sentence representations, we extend our work from dependency parsing to constituency parsing. However, the latter is significantly more challenging than the former under the shift-reduce paradigm because:
Figure 3.4: Our shift-promote-adjoin system for constituency parsing.
• we also need to predict the nonterminal labels

• the tree is not binarized (with many unary rules and more than binary branching rules)

While most previous work binarizes the constituency tree in a preprocessing step [73, 63, 38], we propose a novel “Shift-Promote-Adjoin” paradigm which does not require any binarization or transformation of constituency trees (see Figure 3.5). Note in particular that, in our case only the Promote action produces a new tree node (with a non-terminal label), while the Adjoin action is the linguistically-motivated “sister-adjunction” operation, i.e., attachment [8, 25]. By comparison, in previous work, both Unary-X and Reduce-L/R-X actions produce new labeled nodes (some of which are auxiliary nodes due to binarization). Thus, our paradigm has two advantages:

• it dramatically reduces the number of possible actions, from $3X + 1$ or more in previous work to $3 + X$, where $X$ is the number of nonterminal labels, which we argue would simplify learning;

• it does not require binarization [73, 63] or compression of unary chains [38]

There is, however, a more closely-related “shift-project-attach” paradigm by Henderson (2003) [25]. For the example in Figure 3.5 he would use the following actions:

shift(I), project(NP), project(S), shift(like), project(VP), shift(sports), project(NP), attach, attach.

The differences are twofold: first, our Promote action is head-driven, which means we only promote the head child (e.g., VP to S) whereas his Project action promotes the
Figure 3.5: Shift-Promote-Adjoin parsing example. Upward and downward arrows indicate promote and (sister-)adjunction actions, respectively.

First child (e.g., NP to S); and secondly, as a result, his Attach action is always right-attach whereas our Adjoin action could be either left or right. The advantage of our method is its close resemblance to shift-reduce dependency parsing, which means that our constituency parser is jointly performing both tasks and can produce both kinds of trees. This also means that we use head rules to determine the correct order of gold actions.

We found that in this setting, we did need slightly more input features. As mentioned, node labels are necessary to distinguish whether a tree has been sufficiently promoted, and are helpful in any case. We used eight labels: the current and immediate predecessor label of each of the top two stacks on the tree, as well as the label of the left- and rightmost adjoined child for each tree. We also found it helped to add positional features for the leftmost word in the span for each of those trees, bringing the total number of positional features to five. See Table 3.1 for details.
3.5 Experimental Results

We report both dependency and constituency parsing results on both English and Chinese.

All experiments were conducted with minimal hyperparameter tuning. The settings used for the reported results are summarized in Table 3.6. Networks parameters were updated using gradient backpropagation, including backpropagation through time for the recurrent components, using ADADELTA for learning rate scheduling [68]. We also applied dropout [27] (with \( p = 0.5 \)) to the output of each LSTM layer (separately for each connection in the case of the two-layer network).

We tested both types of parser on the Penn Treebank (PTB) and Penn Chinese Treebank (CTB-5), with the standard splits for each of training, development, and test sets. Automatically predicted part of speech tags with 10-way jackknifing were used as inputs for all tasks except for Chinese dependency parsing, where we used gold tags, following the traditions in literature.

3.5.1 Dependency Parsing: English and Chinese

Table 3.2 shows results for English Penn Treebank using Stanford dependencies. Despite the minimally designed feature representation, relatively few training iterations, and lack of pre-computed embeddings, the parser performed on par with state-of-the-art incremental dependency parsers, and slightly outperformed the state-of-the-art greedy parser.

The ablation experiments shown in the Table 3.3 indicate that both forward and
backward contexts for each word are very important to obtain strong results. Using only word forms and no part-of-speech input similarly degraded performance.

<table>
<thead>
<tr>
<th>Parser</th>
<th>Dev</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UAS</td>
<td>LAS</td>
</tr>
<tr>
<td></td>
<td>UAS</td>
<td>LAS</td>
</tr>
<tr>
<td>C &amp; M 2014</td>
<td>92.0</td>
<td>89.7</td>
</tr>
<tr>
<td>Dyer et al. 2015</td>
<td>93.2</td>
<td>90.9</td>
</tr>
<tr>
<td>Weiss et al. 2015</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>+ Percept./Beam</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Bi-LSTM</td>
<td>93.31</td>
<td>91.01</td>
</tr>
<tr>
<td>2-Layer Bi-LSTM</td>
<td>93.67</td>
<td>91.48</td>
</tr>
</tbody>
</table>

Table 3.2: Development and test set results for shift-reduce dependency parser on Penn Treebank using only \((s_1, s_0, q_0)\) positional features.
Table 3.3: Ablation studies on PTB dev set (wsj 22). Forward and backward context, and part-of-speech input were all critical to strong performance.

<table>
<thead>
<tr>
<th>Parser</th>
<th>UAS</th>
<th>LAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bi-LSTM Hierarchical†</td>
<td>93.31</td>
<td>91.01</td>
</tr>
<tr>
<td>† - Hierarchical Actions</td>
<td>92.94</td>
<td>90.96</td>
</tr>
<tr>
<td>† - Backward-LSTM</td>
<td>91.12</td>
<td>88.72</td>
</tr>
<tr>
<td>† - Forward-LSTM</td>
<td>91.85</td>
<td>88.39</td>
</tr>
<tr>
<td>† - tag embeddings</td>
<td>92.46</td>
<td>89.81</td>
</tr>
</tbody>
</table>

Figure 3.6 compares our parser with that of Chen and Manning (2014) [7] in terms of arc recall for various arc lengths. While the two parsers perform similarly on short arcs, ours significantly outperforms theirs on longer arcs, and more interestingly, our accuracy does not degrade much after length 6. This confirms the benefit of having a global sentence representation in our model.

Table 3.4 summarizes the Chinese dependency parsing results. Again, our work is competitive with the state-of-the-art greedy parsers.
Figure 3.6: Recall on dependency arcs of various lengths in PTB dev set. The Bi-LSTM parser is particularly good at predicting longer arcs.

<table>
<thead>
<tr>
<th>Parser</th>
<th>Dev</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UAS</td>
<td>LAS</td>
</tr>
<tr>
<td>C &amp; M 2014</td>
<td>84.0</td>
<td>82.4</td>
</tr>
<tr>
<td>Dyer et al. 2015</td>
<td>87.2</td>
<td>85.9</td>
</tr>
<tr>
<td>Bi-LSTM</td>
<td>85.84</td>
<td>85.24</td>
</tr>
<tr>
<td>2-Layer Bi-LSTM</td>
<td>86.13</td>
<td>85.51</td>
</tr>
</tbody>
</table>

Table 3.4: Development and test set results for shift-reduce dependency parser on Penn Chinese Treebank (CTB-5) using only \((s_1, s_0, q_0)\) position features (trained and tested with gold POS tags).
3.5.2 Constituency Parsing: English & Chinese

Table 3.5 compares our constituency parsing results with state-of-the-art incremental parsers. Although our work definitely leads to less accurate results than those beam-search parsers, we achieve the highest accuracy among greedy parsers, for both English and Chinese.\(^1\)\(^2\)

<table>
<thead>
<tr>
<th>Parser</th>
<th>(b)</th>
<th>English greedy beam</th>
<th>Chinese greedy beam</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zhu et al. (2013) [73]</td>
<td>16</td>
<td>86.08 90.4</td>
<td>75.99 85.6</td>
</tr>
<tr>
<td>Mi &amp; Huang (05)</td>
<td>32</td>
<td>84.95 90.8</td>
<td>75.61 83.9</td>
</tr>
<tr>
<td>Vinyals et al. (05)</td>
<td>10</td>
<td>-      90.5</td>
<td>-</td>
</tr>
<tr>
<td>Bi-LSTM</td>
<td>-</td>
<td>89.75 -</td>
<td>79.44 -</td>
</tr>
<tr>
<td>2-Layer Bi-LSTM</td>
<td>-</td>
<td><strong>89.95</strong> -</td>
<td><strong>80.13</strong> -</td>
</tr>
</tbody>
</table>

Table 3.5: Test F-scores for constituency parsing on Penn Treebank and CTB-5.

---

\(^1\) The greedy accuracies for Mi and Huang (2015) [38] are from Haitao Mi, and greedy results for Zhu et al. (2013) [73] come from duplicating experiments with code provided by those authors.

\(^2\) The parser of Vinyals et al. (2015) [61] does not use an explicit transition system, but is similar in spirit since generating a right bracket can be viewed as a reduce action.
Because recurrent networks are such a natural fit for modeling languages (given the sequential nature of the latter), bi-directional LSTM networks are becoming increasingly

### Table 3.6: Hyperparameters and training settings.

<table>
<thead>
<tr>
<th></th>
<th>Dependency</th>
<th>Constituency</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Embeddings</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Word (dims)</td>
<td>50</td>
<td>100</td>
</tr>
<tr>
<td>Tags (dims)</td>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>Nonterminals (dims)</td>
<td>-</td>
<td>100</td>
</tr>
<tr>
<td>Pretrained</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Network details</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSTM units (each direction)</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>ReLU hidden units</td>
<td>200 / decision</td>
<td>1000</td>
</tr>
<tr>
<td><strong>Training</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Training epochs</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Minibatch size (sentences)</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Dropout (LSTM output only)</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>L2 penalty (all weights)</td>
<td>none</td>
<td>$1 \times 10^{-8}$</td>
</tr>
<tr>
<td>ADADELTA $\rho$</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>ADADELTA $\epsilon$</td>
<td>$1 \times 10^{-7}$</td>
<td>$1 \times 10^{-7}$</td>
</tr>
</tbody>
</table>

3.6 Related Work
common in all sorts of linguistic tasks, for example event detection in Ghaehini et al. (2016) [21]. In fact, we discovered after submission that Kiperwasser et al. (2016) [35] have concurrently developed an extremely similar approach to our dependency parser. Instead of extending it to constituency parsing, they also apply the same idea to graph-based dependency parsing.

3.7 Conclusions

We have presented a simple bi-directional LSTM sentence representation model for minimal features in both incremental dependency and incremental constituency parsing, the latter using a novel shift-promote-adjoin algorithm. Experiments show that our method are competitive with the state-of-the-art greedy parsers on both parsing tasks and on both English and Chinese.
Chapter 4: Span-Based Constituency Parsing with a Structure-Label System and Dynamic Oracles

This chapter describes the work in Cross and Huang (2016b) [15].

Parsing accuracy using extremely efficient greedy transition-based parsers has improved dramatically in recent years thanks to neural-network learning models. 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 parsing system which uses a stack of sentence spans, represented by a bare minimum of LSTM features. We also describe a dynamic oracle for this constituency parsing system. Training with this oracle, we achieve the best performance on Penn Treebank of any parser that does not use reranking or any external data.

4.1 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 chapter, 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 depen-
In constituency parsing, however, neural approaches are still behind the state-of-the-art [6, 49, 59]; see more details in Section 4.5.

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 (four for structural actions and three for label actions). Extending Wang and Chang (2016), [62], 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 [29], 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 4.2).

- A neural model where sentence spans are represented as differences of output from a multi-layer bi-directional LSTM (Section 4.3).

- The first provably optimal dynamic oracle for constituency parsing which is also extremely efficient (amortized $O(1)$ time) (Section 4.4).

- 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.\footnote{code: https://github.com/jhcross/span-parser}

### 4.2 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.

\footnote{code: https://github.com/jhcross/span-parser}
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), [38]. 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 4.1 shows the formal deductive system for this parser. The stack \(\sigma\) is modeled
as a list of strictly increasing integers whose first element is always 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 \((i, \square_j)\) in the figures to emphasize that they may or not have
tree structure.

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
input: \( w_0 \ldots w_{n-1} \)

axiom: \( \langle 0, [0], \emptyset \rangle \)

goal: \( \langle 2(2n - 1), [0, n], t \rangle \)

\[
\text{sh} \quad \frac{\langle z, \sigma \, j, t \rangle}{\langle z + 1, \sigma \, j + 1, t \rangle} \quad j < n, \text{even } z
\]

\[
\text{comb} \quad \frac{\langle z, \sigma \, i \, k, t \rangle}{\langle z + 1, \sigma \, i, t \rangle} \quad \text{even } z
\]

\[
\text{label-X} \quad \frac{\langle z, \sigma \, i, t \rangle}{\langle z + 1, \sigma \, i, t \cup \{iX_j\} \rangle} \quad \text{odd } z
\]

\[
\text{nolabel} \quad \frac{\langle z, \sigma \, i, t \rangle}{\langle z + 1, \sigma \, i, t \rangle} \quad z < (4n - 1), \text{odd } z
\]

Figure 4.1: Deductive system for the Structure/Label transition parser. The stack \( \sigma \) 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.

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
Figure 4.2: 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.

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, \((4n - 2)\), nolabel is not allowed since the parser must produce a tree.

Figure 4.2 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 → 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.

4.3 LSTM Span Features

Long short-term memory networks (LSTM) are a type of recurrent neural network model proposed by Hochreiter and Schmidhuber (1997) [28] 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 [56] and translation [57].

LSTMs have also been incorporated into parsing in a variety of ways, such as di-
rectly encoding an entire sentence [61], separately modeling the stack, buffer, and action history [19], to encode words based on their character forms [3], and as an element in a recursive structure to combine dependency subtrees with their left and right children [34].

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 $f_0, \ldots, f_n$ in the forward direction and $b_n, \ldots, b_0$ 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 4.3. This is essentially the LSTM-Minus feature representation described by Wang and Chang (2016) [62] 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 endpoint 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
Figure 4.3: Word spans are modeled by differences in LSTM output. Here the span \(3\) eating fish \(5\) is represented by the vector differences \((f_5 - f_3)\) and \((b_3 - b_5)\). The forward difference corresponds to LSTM-Minus [62].

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. The previous chapter (based on Cross and Huang (2016a) [14]) gives 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 4.1.

The input to the recurrent network at each time step consists of vector embeddings
Table 4.1: Features used for the parser. No label or tree-structure features are required.

for each word 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 pre-trained from a large external corpus.

The network structure after the span features consists of a separate multilayer perceptron for each type of action (structural and label). For each of these classifiers, 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) [7].

4.4 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-documented 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) [23] 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. 4.4.3) is involved, computing the oracle actions is extremely simple (Secs. 4.4.2) and efficient (Sec. 4.4.4).

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) [24].

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

**Definition 2** (descendant/reachable trees). We denote \( D(c) \) to be the set of final descendant trees derivable from \( c \), i.e., \( D(c) = \{ t \mid c \vdash^* \langle z, \sigma, t \rangle \} \). 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 $F_1(t) = \frac{2rp}{r+p}$, where $r = \frac{|r \cap t_G|}{|t_G|}$, $p = \frac{|r \cap t|}{|t|}$.

The following two definitions are similar to those for dependency parsing by Goldberg et al. (2014) [24].

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

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 $F_1$:

$$\text{oracle}(c) = \{ \tau \mid F_1(\tau(c)) = F_1(c) \}.$$ 

This abstract oracle is implemented by $\text{dyna}(\cdot)$ in Sec. 4.4.2, which we prove to be correct in Sec. 4.4.3.

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

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 $(i, j) \neq (p, q)$. We then extend this relation from spans to brackets, and notate $iX_j \prec pY_q$ iff. $(i, j) \prec (p, q)$.

We next define a central concept, reachable brackets, which is made up of two parts, the left ones $\text{left}(c)$ which encompass $(i, j)$ without crossing any stack spans, and the right ones $\text{right}(c)$ which are completely on the queue. See Fig. 4.4 for examples.
Figure 4.4: Reachable brackets (w.r.t. gold tree in Fig. 4.1) for \( c = \langle 10, [0, 1, 2, 4], \{NP_1\} \rangle \) which mistakenly combines “like eating”. Trapezoids indicate stack spans (the top one in red), and solid triangles denote reachable brackets, with \( \text{left}(c) \) in blue and \( \text{right}(c) \) in cyan. The next reachable bracket, \( \text{next}(c) = VP_5 \), is in bold. Brackets \( VP_5 \) and \( S_5 \) (in dotted triangle) cross the top span (thus unreachable), and \( NP_1 \) is already recognized (thus not in \( \text{reach}(c) \) either).

**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

\[
\text{reach}(c) = \text{left}(c) \cup \text{right}(c)
\]

where the left- and right-reachable brackets are

\[
\text{left}(c) = \{ pX_q \in t_G \mid (i, j) \prec (p, q), p \in \sigma | i \} \\
\text{right}(c) = \{ pX_q \in t_G \mid p \geq j \}
\]

for even \( z \), with the \( \prec \) replaced by \( \preceq \) for odd \( z \).
Special case (initial): $\text{reach}((0, [0], \emptyset)) = t_G$.

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 $q \geq j$ due to $(i, j) \prec (p, q)$.

Also note that $\text{reach}(c)$ is strictly disjoint from $t$, i.e., $\text{reach}(c) \cap t = \emptyset$ and $\text{reach}(c) \subseteq t_G - t$. See Figure 4.6 for an illustration.

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

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

### 4.4.2 Structural and Label Oracles

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

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

(4.1)

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

Figure 4.5 shows examples of this policy. The key insight is, if you follow this
<table>
<thead>
<tr>
<th>configuration</th>
<th>oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>dynamic</td>
</tr>
<tr>
<td>0△1△2△3</td>
<td>comb</td>
</tr>
<tr>
<td>I do like</td>
<td>{comb, sh}</td>
</tr>
<tr>
<td>0△1△3</td>
<td>sh</td>
</tr>
<tr>
<td>I do like</td>
<td>{comb, sh}</td>
</tr>
<tr>
<td>0△1△2△4</td>
<td>undefined</td>
</tr>
<tr>
<td>I do like eating</td>
<td>{comb, sh}</td>
</tr>
<tr>
<td>0△1△2△4△5</td>
<td>comb</td>
</tr>
<tr>
<td>I do like eating fish</td>
<td>{comb}</td>
</tr>
</tbody>
</table>

Figure 4.5: Dynamic oracle with respect to the gold parse in Fig. 4.2. 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 $1VP_3$ in all cases.

Policy, you will not miss the next reachable bracket, but if you do not follow it, you certainly will. We formalize this fact below, and it 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 $reach(\tau(c)) \subseteq reach(c)$.

The label oracles are much easier than structural ones. For an odd-step configuration $c = (z, \sigma \mid i \mid j, t)$, 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,

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

(4.2)

4.4.3 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. 4.1–4.2) 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, $t^*(c) = t \cup \text{reach}(c)$.

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 $c$, $F_1(t^*(c)) = F_1(c)$.

**Proof.** Since $t^*(c)$ adds all possible additional gold brackets (the brackets in $\text{reach}(c)$), it is not possible to get higher recall. Since it adds no incorrect brackets, it is not possible to get higher precision. Since $F_1$ is the harmonic mean of precision and recall, it also leads to the best possible $F_1$. 

\qed
Figure 4.6: The optimal tree $t^*(c)$ adds all reachable brackets and nothing else. Note that $\text{reach}(c)$ and $t$ are disjoint.

**Corollary 1.** For any $c = \langle z, \sigma, t \rangle$, for any $t' \in \mathcal{D}(c)$ and $t' \neq t^*(c)$, we have $F_1(t') < F_1(c)$.

We now need a final lemma about the policy $\text{dyna}(\cdot)$ (Eqs. 4.1–4.2) before proving the main result.

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

**Proof.** By case analysis on even/odd $z$. □

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 $\text{dyna}(\cdot)$ in Eqs. (4.1–4.2) satisfies the requirement of a dy-
4.4.4 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 $|\text{reach}(c)|$ and the choice of $\text{next}(c)$. After each action, $\text{next}(c)$ 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 $\text{next}(c)$ 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) [24].

4.5 Related Work

Neural networks have been used for constituency parsing in a number of previous instances. For example, Socher et al. (2013) [50] learn a recursive network that combines vectors representing partial trees; Vinyals et al. (2015) [61] adapt a sequence-to-sequence model to produce parse trees; Watanabe and Sumita (2015) [64] use a recursive model applying a shift-reduce system to constituency parsing with beam search; and Dyer et al. (2016) [20] adapt the Stack-LSTM dependency parsing approach to this
task. Durrett and Klein (2015) [18] combine both neural and sparse features for a CKY parsing system. Our own previous work [14] uses 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) [29] in bottom-up parsing. In transition-based parsing, the dynamic oracle for shift-reduce dependency parsing costs worst-case $O(n^3)$ time [24]. On the other hand, after the submission of our paper we became aware of a parallel work [11] that also proposed a dynamic oracle for their own incremental constituency parser. However, it is not optimal due to dummy non-terminals from binarization.

4.6 Experiments

We present experiments on both the Penn English Treebank [36] and the French Treebank [1]. 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 [27], 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 [68] to schedule learning rates for all weights. All of these design choices are summarized in Table 4.2.

In order to account for unknown words during training, we also adopt the strategy described by Kiperwasser and Goldberg (2016a) [35], 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.

<table>
<thead>
<tr>
<th><strong>Network architecture</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Word embeddings</td>
</tr>
<tr>
<td>Tag embeddings</td>
</tr>
<tr>
<td>Morphological embeddings†</td>
</tr>
<tr>
<td>LSTM layers</td>
</tr>
<tr>
<td>LSTM units</td>
</tr>
<tr>
<td>ReLU hidden units</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Training settings</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Embedding initialization</td>
</tr>
<tr>
<td>Training epochs</td>
</tr>
<tr>
<td>Minibatch size</td>
</tr>
<tr>
<td>Dropout (on LSTM output)</td>
</tr>
<tr>
<td>ADADELTA parameters</td>
</tr>
</tbody>
</table>

Table 4.2: Hyperparameters. †French only.
4.6.1 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) [23]. 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.
Algorithm 1 Dynamic Oracle with Exploration

\[ U \leftarrow \emptyset \] ▷ training updates

for sentence and gold tree \((s, t_G)\) in batch do

\[ c \leftarrow \text{initial}(s) \] ▷ initial configuration

\[ \rho \leftarrow \text{LSTM}(s, \theta) \] ▷ recurrent network forward propagation

while \(c\) is not final configuration do

\[ a' \leftarrow \text{oracle}(s, t_G, c) \]

\[ U \leftarrow U \cup (s, c, a') \]

\[ \text{scores} \leftarrow \text{MLP}(\rho, \theta, c) \] ▷ model action scores

\[ \mathbf{P} \leftarrow \text{softmax}(\text{scores}) \]

\[ a \leftarrow \text{draw}(\mathbf{P}) \] ▷ randomly sample action to follow

\[ c \leftarrow \text{take}(c, a) \]

end while

end for

for \((s, c, a)\) ∈ \(U\) do

\[ \theta \leftarrow \text{update}(\theta, s, c, a) \] ▷ back-propagate to all parameters

end for

Accordingly, we also used a method recently proposed by Ballesteros et al. (2016) [4], 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. Details of this training procedure can be seen
This technique can also take a parameter $\alpha$ to flatten or sharpen the raw softmax distribution. The results on the Penn treebank development set for various values of $\alpha$ are presented in Table 4.3. 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.

<table>
<thead>
<tr>
<th>Model</th>
<th>LR</th>
<th>LP</th>
<th>$F_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Oracle</td>
<td>91.34</td>
<td>91.43</td>
<td>91.38</td>
</tr>
<tr>
<td>Dynamic Oracle</td>
<td>91.14</td>
<td>91.61</td>
<td>91.38</td>
</tr>
<tr>
<td>+ Explore ($\alpha = 0.5$)</td>
<td>90.59</td>
<td>92.18</td>
<td>91.38</td>
</tr>
<tr>
<td>+ Explore ($\alpha = 1.0$)</td>
<td>91.07</td>
<td>92.22</td>
<td><strong>91.64</strong></td>
</tr>
<tr>
<td>+ Explore ($\alpha = 1.5$)</td>
<td>91.07</td>
<td>92.12</td>
<td>91.59</td>
</tr>
</tbody>
</table>

Table 4.3: Comparison of performance on PTB development set using different oracle training approaches.

For completeness, we also implemented a third dynamic-oracle training strategy, which we call “Mixed Exploration”, which represents an intermediate point between the quickly-overfitting naive dynamic oracle training, and the fully stochastic exploration strategy. In addition to $\alpha$, this strategy has another parameter $\beta$, which represents the probability that we take the correct action in any given training step, as defined by the dynamic oracle, rather than sampling from the distribution. This strategy, fully
presented as Algorithm 2, was motivated by the observation that the accuracy (i.e., $F_1$) of training parses, produced through the sampling method, lagged significantly behind the evaluation accuracy on the validation data throughout most of training. Although early experiments showed that this method (with $\beta = 0.5$) led to some gain over exploration alone, those results were ultimately not sufficiently consistent to introduce yet another hyperparameter for potential optimization.
Algorithm 2 Dynamic Oracle with Mixed Exploration

\[ U \leftarrow \emptyset \quad \triangleright \text{training updates} \]
\[ \beta \leftarrow 0.5 \quad \triangleright \text{gold probability (hyperparameter)} \]

for sentence and gold tree \((s, t_G)\) in batch do

\[ c \leftarrow \text{initial}(s) \quad \triangleright \text{initial configuration} \]
\[ \rho \leftarrow \text{LSTM}(s, \theta) \quad \triangleright \text{recurrent network forward propagation} \]

while \(c\) is not final configuration do

\[ a' \leftarrow \text{oracle}(s, t_G, c) \]
\[ U \leftarrow U \cup (s, c, a') \]
\[ r \leftarrow \text{draw(uniform}(0, 1)) \]

if \(r > \beta\) then

\[ c \leftarrow \text{take}(c, a') \]

else

\[ \text{scores} \leftarrow \text{MLP}(\rho, \theta, c) \quad \triangleright \text{model action scores} \]
\[ \text{P} \leftarrow \text{softmax}(\text{scores}) \]
\[ a \leftarrow \text{draw}(\text{P}) \quad \triangleright \text{randomly sample action to follow} \]
\[ c \leftarrow \text{take}(c, a) \]

end if

end while

end for

for \((s, c, a) \in U\) do

\[ \theta \leftarrow \text{update}(\theta, s, c, a) \quad \triangleright \text{back-propagate to all parameters} \]

end for
4.6.2 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 4.4 compares 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.

4.6.3 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 data set [48]. 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 4.5. We achieve state-of-
the-art results even in comparison to Björkelund et al. (2014) [5], which utilized both external data and reranking in achieving the best results in the SPMRL 2014 shared task.

4.6.4 Notes on Experiments

For these experiments, we performed very little hyperparameter tuning, due to time and resource constraints. 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) [65].

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) [7], but on a per-sentence basis.

4.7 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 [31] 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) [35].
<table>
<thead>
<tr>
<th>Closed Training &amp; Single Model</th>
<th>LR</th>
<th>LP</th>
<th>F₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sagae and Lavie (2006) [47]</td>
<td>88.1</td>
<td>87.8</td>
<td>87.9</td>
</tr>
<tr>
<td>Petrov and Klein (2007) [46]</td>
<td>90.1</td>
<td>90.3</td>
<td>90.2</td>
</tr>
<tr>
<td>Carreras et al. (2008) [6]</td>
<td>90.7</td>
<td>91.4</td>
<td>91.1</td>
</tr>
<tr>
<td>Shindo et al. (2012) [49]</td>
<td></td>
<td></td>
<td>91.1</td>
</tr>
<tr>
<td>†Socher et al. (2013) [50]</td>
<td></td>
<td></td>
<td>90.4</td>
</tr>
<tr>
<td>Zhu et al. (2013) [74]</td>
<td>90.2</td>
<td>90.7</td>
<td>90.4</td>
</tr>
<tr>
<td>Mi and Huang (2015) [38]</td>
<td>90.7</td>
<td>90.9</td>
<td>90.8</td>
</tr>
<tr>
<td>†Watanabe and Sumita (2015) [64]</td>
<td></td>
<td></td>
<td>90.7</td>
</tr>
<tr>
<td>Thang et al. (2015) [59] (A*)</td>
<td>90.9</td>
<td>91.2</td>
<td>91.1</td>
</tr>
<tr>
<td>†*Dyer et al. (2016) [20] (discrim.)</td>
<td></td>
<td></td>
<td>89.8</td>
</tr>
<tr>
<td>†*Cross and Huang (2016a) [14]</td>
<td></td>
<td></td>
<td>90.0</td>
</tr>
<tr>
<td>†*static oracle</td>
<td>90.7</td>
<td>91.4</td>
<td>91.0</td>
</tr>
<tr>
<td>†*dynamic/exploration</td>
<td>90.5</td>
<td>92.1</td>
<td>91.3</td>
</tr>
<tr>
<td>External/Reranking/Combo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>†Henderson (2014) [26] (rerank)</td>
<td>89.8</td>
<td>90.4</td>
<td>90.1</td>
</tr>
<tr>
<td>McClosky et al. (2016) [37]</td>
<td>92.2</td>
<td>92.6</td>
<td>92.4</td>
</tr>
<tr>
<td>Zhu et al. (2013) [74] (semi)</td>
<td>91.1</td>
<td>91.5</td>
<td>91.3</td>
</tr>
<tr>
<td>Huang (2008) [29] (forest)</td>
<td></td>
<td></td>
<td>91.7</td>
</tr>
<tr>
<td>†Vinyals et al. (2015) [61] (WSJ)‡</td>
<td></td>
<td></td>
<td>90.5</td>
</tr>
<tr>
<td>†Vinyals et al. (2015) [61] (semi)</td>
<td></td>
<td></td>
<td>92.8</td>
</tr>
<tr>
<td>†Durrett and Klein (2015) [18]‡</td>
<td></td>
<td></td>
<td>91.1</td>
</tr>
<tr>
<td>†Dyer et al. (2016) [20] (gen. rerank.)</td>
<td></td>
<td></td>
<td>92.4</td>
</tr>
</tbody>
</table>

Table 4.4: Comparison of performance of different parsers on PTB test set. †Neural. *Greedy. ‡External embeddings.
Table 4.5: Results on French Treebank. *reranking, ‡external.

<table>
<thead>
<tr>
<th>Parser</th>
<th>LR</th>
<th>LP</th>
<th>F₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>Björkelund et al. (2014) [5]‡</td>
<td></td>
<td></td>
<td>82.53</td>
</tr>
<tr>
<td>Durrett and Klein (2015) [18]§</td>
<td></td>
<td></td>
<td>81.25</td>
</tr>
<tr>
<td><strong>static oracle</strong></td>
<td>83.50</td>
<td>82.87</td>
<td>83.18</td>
</tr>
<tr>
<td><strong>dynamic/exploration</strong></td>
<td>81.90</td>
<td>84.77</td>
<td><strong>83.31</strong></td>
</tr>
</tbody>
</table>
Chapter 5: Simplified Features for Span-Based Constituency Parsing

The previous chapter introduces a novel transition system for constituency parsing, which, by leveraging the power of recurrent neural networks to represent an entire sentence, is able to achieve state-of-the-art parsing results despite using greedy inference. Given a recurrent representation of a sentence, the local features necessary to make a decision are quite minimalistic, with structural actions being based on four non-overlapping sentence spans, and label actions being based on three such sentence spans.

This chapter will show that this feature space can be simplified still further. In particular, structural actions can be predicted using the same three sentence spans as label actions. This is counterintuitive since it makes sense to individually represent the top two stack spans when considering whether to combine them, but the reasoning behind can be theoretically justified through an extension of the proof of correctness for the dynamic oracle described in Cross and Huang (2016b) [15]. The essential equivalence of the two feature representations is further borne out by experimental results.

This simplification is especially interesting given that the feature representation for every action decision now consists of a partition of the sentence into three non-overlapping spans. This means that for a sentence of length \(n\), there are only \(O(n^2)\) possible states. Such a compressed feature space could have many interesting applications, including local training on all possible states, exhaustive search of the action space, and the ability to cover an extremely large search space using dynamic program-
Figure 5.1: An illustration of the spans used to make decisions in Cross and Huang (2016b) [15]. (a) Structural actions (here combine) are determined with 4 spans, while (b) label actions (here label-VP) require only 3 spans.

5.1 Feature Representation

Recall that the parser model described in Cross and Huang (2016b) [15] relies on making local parsing decisions based on a representation of the sentence as a whole (using recurrent neural networks) and a representation of the parser state using only a few sentence spans. In particular, the sentence spans for each state comprise a partition of the sentence into contiguous spans: four in the case of structural actions, and three in the
case of the label actions. Examples of such partitions on a representative parser configuration can be seen in Figure 5.1.

It is very easy to see why the sentence is partitioned into the top stack span $s_0$ and the remainder of the sentence both before and after it when taking a label action: it is exactly that sentence span which may be assigned one or more phrase labels when taking an affirmative label action. In the structural action case, the intuition behind using both of the top two stack elements is also straightforward. Those are the sentence spans which will be merged if a comb action is taken.

From another point of view however, the fundamental question faced when predicting a structural action could be defined as whether top stack span $s_0$ is more likely to be the beginning or ending of a larger phrase. In some sense the determination that needs to be made is whether the top stack span “binds more closely” to the portion of the sentence preceding it or the portion of the sentence following it. Based on this alternative intuition, this chapter will demonstrate that the span-based parsing approach can perform equally well using only three sentence spans for both label and structural actions. This feature representation is illustrated in Figure 5.2.

We will place the simplified feature approach on firm theoretical ground by showing a formal equivalence between the optimal dynamic oracle set forth in Cross and Huang (2016b) [15] and a “local” dynamic oracle which depends only on the top stack span $s_0$. We will then present experimental results which demonstrate not only that the simplified feature representation is capable of producing equally good results as the original feature representation, but also that the local dynamic oracle actually works for dynamic oracle training.
5.2 Local Dynamic Oracle

In this section, we will prove the counterintuitive result that the bounds of the sentence span at the top of the stack are sufficient to determine the optimal structural action from any arbitrary state when using the span-based parsing approach. We will do so by defining a local dynamic oracle, $local(i, j, t_G)$, depending only on the top stack span $s_0$ (as defined by the rightmost integers in $\sigma$, $(i, j)$) and the gold tree $t_G$. We will then show that $local(i, j, t_G)$ is equivalent to the dynamic oracle $dyna(c)$, itself shown to be optimal in Cross and Huang (2016b) [15].

For label actions, $dyna(c)$ and $local(i, j, t_G)$ are identical, since the oracle does not
rely on any aspect of parser state except for the bounds \((i, j)\) of top stack span \(s_0\). The remainder of this section will therefore focus on showing the local oracle for structural actions.

The local oracle depends on a new notion, the \textit{enclosing bracket} of an arbitrary span with respect to some known tree structure (i.e., the gold tree in the case of the oracle). Intuitively, this is the next larger labeled bracket which exists in the tree, and which both includes and is strictly larger than the provided span.

**Definition 11** (enclosing bracket). \(\text{enclosing}(i, j, t_G)\) is defined as the bracket \(pX_q \in t_G\) such that \((i, j) \prec (p, q)\) and there does not exist any other bracket \(hZ_k \in t_G\) such that \((i, j) \prec (h, k)\) and \((h, k) \prec (p, q)\).

We are now in a position to define the local dynamic oracle based on the foregoing definition. It is completely analogous to the definition of \(\text{dyna}(c)\), except that it depends on \(\text{enclosing}(i, j, t_G)\) rather than \(\text{next}(c)\). Specifically, if \(\text{enclosing}(i, j, t_G)\) is \(pX_q\), then the oracle is defined as:

\[
\text{local}(i, j, t_G) = \begin{cases} 
\{\text{sh}\} & \text{if } p = i \text{ and } q > j \\
\{\text{comb}\} & \text{if } p < i \text{ and } q = j \\
\{\text{sh, comb}\} & \text{if } p < i \text{ and } q > j 
\end{cases} \tag{5.1}
\]

We will now show the equivalence of \(\text{local}(i, j, t_G)\) and \(\text{dyna}(c)\) in an arbitrary parser configuration \(c\) as follows. In essence, following the local dynamic oracle as described above will never remove the bracket \(\text{next}(c)\) from the set \(\text{reach}(c)\), and will therefore never change the set \(\text{reach}(c)\) at all, thus remaining optimal. It rests on the observation
that $\text{enclosing}(i, j, t_G) \preceq \text{next}(c)$. In the case of a sh action, it is evident $\text{next}(c)$ cannot be removed from $\text{reach}(c)$ since in those both of those cases $\text{enclosing}(i, j, t_G)$ (non-strictly) encompasses both the previous and new single-word top stack spans.

The reasoning for a comb action is more subtle, but it rests on the fact that even though the oracle does not have access to the exact identity of $\text{next}(c)$, we do know that its left-boundary is also a stack boundary, because otherwise that bracket would not be reachable. In particular, if the stack is $\sigma \| i \| k \| j$ and $\text{next}(c) = h Z_k$, we know that $h \leq i$ (if and only if comb $\in \text{local}(i, j, t_G)$ as defined by Equation 5.1). This means that performing a comb action also does not reduce $\text{reach}(c)$.

5.3 Experimental Results

To the reasonableness of our hypothesis that the simplified feature space is equivalent in expressivity to the larger feature space as originally set forth in in Cross and Huang (2016b) [15], we used the exact same training settings, keeping everything equal except for the feature representation. For training with exploration, we also used the local dynamic oracle formulation, as set forth in Section 5.2.
### Development Set

<table>
<thead>
<tr>
<th>Model</th>
<th>LR</th>
<th>LP</th>
<th>F₁</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Original Features</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static Oracle</td>
<td>91.34</td>
<td>91.43</td>
<td>91.38</td>
</tr>
<tr>
<td>Dynamic + Exploration</td>
<td>91.07</td>
<td>92.22</td>
<td>91.64</td>
</tr>
<tr>
<td><strong>Simplified Features</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static Oracle</td>
<td>91.12</td>
<td>91.24</td>
<td>91.18</td>
</tr>
<tr>
<td>Dynamic + Exploration</td>
<td>91.03</td>
<td>91.81</td>
<td>91.32</td>
</tr>
</tbody>
</table>

Table 5.1: Comparison of performance on PTB development set with both the original and simplified feature sets.

### Test Set

<table>
<thead>
<tr>
<th>Model</th>
<th>LR</th>
<th>LP</th>
<th>F₁</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Original Features</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static Oracle</td>
<td>90.68</td>
<td>91.39</td>
<td>91.03</td>
</tr>
<tr>
<td>Dynamic + Exploration</td>
<td>90.52</td>
<td>92.14</td>
<td>91.32</td>
</tr>
<tr>
<td><strong>Simplified Features</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static Oracle</td>
<td>90.83</td>
<td>91.30</td>
<td>91.06</td>
</tr>
<tr>
<td>Dynamic + Exploration</td>
<td>90.64</td>
<td>92.00</td>
<td>91.32</td>
</tr>
</tbody>
</table>

Table 5.2: Comparison of performance on PTB test set with both the original and simplified feature sets.
Table 5.1 compares the best-observed performance on the Penn Treebank validation set as between the original and simplified feature spaces, while Table 5.1 shows the same for the test set. We observed slight degradation in performance for the validation set when introducing the simplified features, but extremely close results for the test set. We speculate that this may be an overfitting issue, since validation performance was used to determine which model to save, combined with slightly more variance in performance when using an extra span for structural actions. In any case, it is plain that our experiments bear out our theoretical results, in that the simplified features are perfectly compatible with achieving state-of-the-art performance under the span-based parsing model.
Chapter 6: Search-Based and Extended Approaches

We have shown that extremely fast and accurate models for parsing can be learned by employing a recurrent network architecture to directly learn complicated patterns in the surface form of language. Such feature representations can be integrated directly into purely discriminative transition parsing systems, so that very high accuracy can be attained even with linear-time, and in fact greedy, decoding.

In much previous work in transition parsing, however, significant improvement has been seen by employing beam search over the space of possible action sequences [31, 64, 65, 2]. This chapter documents numerous attempts to apply various search-based training and inference approaches to the previously-detailed recurrent neural network training models for parsing. Unfortunately, this has been a primarily negative result, at least in terms of surpassing the performance of the greedy approaches. We will close with some analysis as to why search may be superfluous when using the recurrent models and very simple surface feature choices which we have described.

6.1 Beam-Search

A natural and popular method for improving parsing results is to directly search over the space of possible trees. In the particular case of transition-based parsing, this is typically accomplished by performing beam search over the space of possible action sequences
In general, linear models for beam-search transition parsing are trained using a procedure which tightly integrates the search itself with training updates. The model can be learned with the structured perceptron algorithm [12], coupled with such improved update methods as early update [13] or max-violation [30].

Though directly applying perceptron updates is not a very effective strategy in general for neural network models, Zhou et al. (2015) [72] developed a probabilistic variant where all of the action sequences in the beam at the update step, rather than just the highest-scored one, are used to normalize the training objective. The theory behind this approach was further developed by Andor et al. (2016) [2]. It is essentially an adaptation of conditional random fields (CRF) where the global probability mass is approximated using only the sequences in the beam.

Under this training method, the entire neural network produces a unitless score $h(c,a)$ for each action $a$ in a given parser configuration $c$, rather than the softmax output activation for each configuration which is used when training a greedy parser. We then denote a valid action sequence $y$ as consisting of a sequence of configuration action pairs $(c_0,a_0)...(c_m,a_m)$ constrained by the parser’s transition function $\tau(c_i,a_i) = c_{i+1}$, which means that taking action $a_i$ in configuration $c_i$ produces configuration $c_{i+1}$. During beam search, configurations are compared against each other using additive score, i.e. the score for an action sequence $y$ is given by

$$s(y) = \sum_{(c,a) \in y} h(c,a)$$  \hspace{1cm} (6.1)

Note that in this equation and the ones that follow, all scores are also conditioned...
on the input sentence (words and parts of speech), as well as the network parameters, though these variables are omitted for clarity. The probability of any given action sequence \( y' \) in the beam is defined in terms the exponential function of its total score, normalized over all action sequences in the beam:

\[
p(y') = \frac{\exp(s(y'))}{\sum_{y \in \text{BEAM}} \exp(s(y))}
\]  

(6.2)

The loss function to minimize in training this model is in turn the negative logarithm of the probability of the correct action sequence, as defined by the equation above. In practice, this is a prefix of the static oracle sequence of actions, often chosen using the early-update approach [12], and other beam sequences which are used in the normalization term of Equation 6.2 are all of the same length. If such an oracle prefix is defined as \( y^* \), then the loss function may be expressed as

\[
L(y^*) = -\log p(y^*)
\]  

(6.3)

As part of this dissertation work, we implemented this algorithm for both the dependency and constituency parsers described in Chapter 3, as well as the span-based constituency parser described in Chapter 4, without the simplified feature space described in Chapter 5. To accelerate training, we also adopted the technique described in Andor et al. (2016) [2], whereby network weights are pre-trained using a local objective (i.e., as with our greedy-parsing experiments) before the final output layer weights are set to zero and beam-search training begins. Our experiments included the early update approach described, as well as max-violation as proposed by Huang et al. (2012) [30].
We also implemented variants of the learning as search optimization (LaSO) framework as described by Wiseman and Rush (2016) [66].

Unfortunately, in all of these cases the performance after incorporating and training with beam search was equivalent to or slightly worse than when training on a local action-loss objective and using purely greedy inference. Our conclusion is that there is something fundamental about using a bi-directional recurrent representation of the entire input sentence which makes search extraneous. In particular, as demonstrated empirically by Andor et al. (2016) [2], search wanes in importance as lookahead increases. With a bi-directional recurrent representation, the entire sentence context may be taken into account when making every decision, essentially providing infinite lookahead.

6.2 Beam-Search with Dynamic Programming

Search may be made even more powerful with dynamic programming, in particular the state-merging technique pioneered by Huang and Sagae (2010) [31]. There are two primary reasons that suggest this could be an effective technique when combined with the parsing models described in this dissertation: the very high degree of state merging made possible by the simplicity of the surface features used, especially in the simplified span-based parser described in Chapter 5, which allows the exploration of a very large portion of the exponential search space, and the possibility of incorporating the dynamic programming directly into a CRF-like search objective.

Expanding on the second point, incorporating dynamic programming directly into training, using highly modular deep learning libraries it is possible to include all possi-
ble derivations for a single parse state (which may have an extremely large number of paths back through the action space) into the normalizing partition function of Equation 6.2. To account for multiple paths enabled by the graph-structured stack, adapted from Tomita (1991) [60], there is a separate “inside” score maintained for each parser configuration, the total score required to produce the top stack element, together with back pointers to the corresponding configurations before the first word of the top span was shifted (“left contexts”). In our experiments, we accounted for both the derivations of all left contexts and all derivations of the top stack span itself. For training (specifically, partition function) purposes, the combined score of two merged action sequences $y^{(a)}$ and $y^{(b)}$ can be expressed as follows:

$$s(y) = \log(\exp(s(y^{(a)})) + \exp(s(y^{(b)})))$$

(6.4)

The idea behind incorporating an exponential number of parsing derivations into the training objective in this way is to more closely approximate a global probability distribution, as defined by conditional random fields. We implemented beam-search parsers with dynamic programming for the constituency parsers described in Chapters 4 and 5, as well as training algorithms directly incorporating state merging as described above. Though these training methods were in most cases more reliable than vanilla beam search, they were still not able to match or exceed the performance obtainable by local training and greedy parsing. We speculate that this is due, as before, to the fact that the unbounded context available from the recurrent sentence representation obviates the need for search. It is possible however, that more directly encoding parsing decisions
(such as labels in the case of constituency parsing) into the feature representations of states, may make more suitable, and the direct incorporation of dynamic programming into a CRF-style training objective for structured prediction remains an attractive subject for future study.

6.3 Recursive Network Features

Another extension which we have explored in connection with above-described parsing systems is the incorporation of recursive network features directly representing partial tree structure. The greedy dependency parser described in Chapter 3 uses no arc-label information and minimal structure information (left- and right-most children of the top two stack elements), while the span-based parser described in Chapters 4 and 5 uses no tree-structured or label information whatsoever, and is still able to achieve state-of-the-art results despite using a discriminative approach. It seems only natural that performance may be able to be improved still further by also including features describing top stack elements, such as the subtrees, including labels that they contain.

As an example, because our span-based parser supports an arbitrary number of child nodes per non-terminal, and especially because we would like our feature representation to encompass the full structure of an arbitrary number of subtrees to arbitrary depth, stack spans may be represented as the output of a recursive neural network, inspired by works such as Socher et al. (2013) [50].

We implemented for the span-based network in the following way: for an unlabeled span of only one word (thus the top stack element after a shift), the span vector is the
output a simple transformation of the word and POS vectors, followed by a non-linearity. Producing stack vectors as \( s_p(z) \) for the \( p \)th stack element from the top in parser step \( z \), this step is represented by the following equation (where the previous top span was \([i, j]\) and thus \( w_{j+1} \) and \( t_{j+1} \) are the word and tag vectors representing the next word, and a colon indicates vector concatenation):

\[
s_0^{(z)} = f(W_{sh} \cdot [w_{j+1} : t_{j+1}] + b_{sh})
\]

Combining two spans is an operation on the previously calculated individual spans. This is illustrated by the diagram in Figure 6.1.

\[
s_0^{(z)} = f(W_{comb} \cdot [s_0^{(z-1)} : s_1^{(z-1)}] + b_{comb})
\]

Finally, each affirmative label action (i.e., not including no label, which leaves all
of the stack vectors unchanged) has its own weight matrix, and in this case the operation only changes the top stack vector:

$$s_0^{(z)} = f(W_{\text{label-X}} \cdot s_0^{(z-1)} + b_{\text{label-X}})$$

Thus, there is a weight vector for every separate parser action (except nolabel), and for each individual weight vector, the weights are tied throughout every application of the network. We anticipate including recursively computed span vectors for the top two or three stack spans as additional input to the hidden layer during discriminative parsing. Though it may seem that the depth of the recursive network could cause vanishing gradient problems, this is actually not the case since every such vector (and thus every matrix application) will appear near the “top” of the network for some action application. Nevertheless, we found that approach led to significantly worse performance for greedy parsing using both of the training techniques described in Chapter 4, even when the recursive features were used to strictly augment the difference features for spans.

While integrating the recursive approach into the search-based methods described in the preceding sections remains a subject for future work, we believe that this approach holds great promise, particularly since the recursive network directly exposes structured information (such as non-terminal labels) that is not available in the purely span-based approach. In particular, Watanabe and Sumita (2015) [64] showed that a recursive parsing approach could be trained with beam search and a max-violation objective, and we believe that the span-based parsing system which we have developed could be incorpo-
rated into such approaches to make them more performant in the future. Many other variants of recursive networks could also be combined with our parsing system in the future, such as the idea of Tree-LSTMs, proposed by Tai et al. (2015) [58], where the memory component controls the flow of information upwards from the leaf nodes to the root of the tree.
Chapter 7: Summary

This dissertation describes the application of memory-based recurrent neural networks to the fundamental and long-studied problem of predicting the grammatical structure of natural language sentences given only their surface form. We have proposed and described in detail both novel parsing systems and novel neural network learning elements, specifically in the form of multiple layer concatenation, which allows parsing decisions to be made by considering the sentence simultaneously at multiple levels of granularity.

We first described a way of incorporating a multi-layer bi-directional sentence representation into the widely used arc-standard transition system for dependency parsing. This parser was able to achieve on par with the state of the art for dependency parsers despite parameterizing each action over a strikingly minimal three word indices. We also introduced the first of our two novel transition systems for constituency parsing, which did not require tree binarization and also used a fairly minimal feature space. This was the first transition-based constituency to achieve very competitive accuracy despite using greedy inference. In conjunction with these two parsing systems, we also developed a new way of combining recurrent neural network architecture, specifically concatenating the output of multiple recurrent layers to reflect the hierarchical nature of language.

We also created a drastically new approach to transition-based constituency parsing with a system built up entirely from sentence spans rather than parser tree structures.
Such spans constituted the elements on this parser’s stack, and in terms of the statistical model used for parsing were represented as elementwise differences of vector outputs, which led to results equivalent to using all endpoints with fewer network parameters. We also developed an efficiently computable dynamic oracle for this system, the first dynamic oracle to be proved optimal for any transition-based constituency parser. Using this oracle and a technique of exploration through sampling, we were able to achieve state-of-the-art results among all single-model constituency parsers trained on a closed training set, despite performing strictly greedy inference. We also extended this approach to morphologically rich languages, where we achieved the best recorded results on the French Treebank.

We subsequently found that statistical models for this parsing system could be simplified still further by partitioning the sentence into only three sentence spans (i.e., with two split points) at each step of parsing for both action types. We showed why this counterintuitive result holds with reference to the aforementioned dynamic oracle, and demonstrated the equivalence of the simplified feature space with experiments.

We also conducted numerous experiments where parsing inference was conducted using beam search for the parsing systems and feature-learning models which we have created. We found uniformly that these did not improve over the much faster greedy approach. We believe that this is because our greedy models are able to learn to indiscriminately exploit the entire sentence context for every decision, and thus do not require search to excel. Nevertheless, we believe there is future scope to incorporate the parsing systems with search over recursively combined neural structures.
Bibliography


[53] Richard Socher, Alex Perelygin, Jean Y Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the conference on*


APPENDICES
Appendix A: Source Code for Span-Based Parser

The source code for the substantial software contribution from this dissertation work, the span-based constituency parser described at length in the text is included here (in the form at time of publication) for reference.

In the hope that future researchers will be able to build from the work, this software has also been published and made completely open source at:

https://github.com/jhcross/span-parser
A.1 features.py

```python
# features.py

Provides a class, FeatureMapper to translate between discrete features (words, POS tags, non-terminal labels) for use in constituency parsing.

from __future__ import print_function
from __future__ import division

import sys
import json
from collections import defaultdict, OrderedDict
import numpy as np
from phrase_tree import PhraseTree
from parser import Parser

class FeatureMapper(object):
    
    Maps words, tags, and label actions to indices.
    
    UNK = '<UNK>'
    START = '<s>'
    STOP = '</s>'

    @staticmethod
def vocab_init(fname, verbose=True):
        
        Learn vocabulary from file of trees in PTB format.
        
        word_freq = defaultdict(int)
        tag_freq = defaultdict(int)
        label_freq = defaultdict(int)

        trees = PhraseTree.load_treefile(fname)

        for i, tree in enumerate(trees):
            for {word, tag} in tree.sentence:
                word_freq[word] += 1
                tag_freq[tag] += 1

            for action in Parser.gold_actions(tree):
                if action.startswith('label-'):
                    label = action[6:]
                    label_freq[label] += 1

            if verbose:
                print('
Tree {}'.format(i), end=''
            sys.stdout.flush()
```

if verbose:
    print('\r', end='')

words = [
    FeatureMapper.UNK,
    FeatureMapper.START,
    FeatureMapper.STOP,
] + sorted(word_freq)
wdict = OrderedDict((w,i) for (i,w) in enumerate(words))
tags = [
    FeatureMapper.UNK,
    FeatureMapper.START,
    FeatureMapper.STOP,
] + sorted(tag_freq)
tdict = OrderedDict((t,i) for (i,t) in enumerate(tags))
labels = sorted(label_freq)
ldict = OrderedDict((l,i) for (i,l) in enumerate(labels))

if verbose:
    print('Loading features from {}'.format(fname))
    print('({} words, {} tags, {} nonterminal-chains)'.format(
        len(wdict),
        len(tdict),
        len(ldict),
    ))

return {
    'wdict': wdict,
    'word_freq': word_freq,
    'tdict': tdict,
    'ldict': ldict,
}

def __init__(self, vocabfile, verbose=True):
    
    Initialize FeatureMapper object from file of trees in PTB format.
    
    if vocabfile is not None:
        data = FeatureMapper.vocab_init(
            fname=vocabfile,
            verbose=verbose,
        )
        self.wdict = data['wdict']
        self.word_freq = data['word_freq']
        self.tdict = data['tdict']
        self.ldict = data['ldict']

        self.word_freq_list = []
        for word in self.wdict.keys():
            if word in self.word_freq:
                self.word_freq_list.append(self.word_freq[word])
            else:
def from_dict(data):
    """
    Create FeatureMapper object from attribute dictionary.
    """
    new = FeatureMapper(None)
    new.wdict = data['wdict']
    new.word_freq = data['word_freq']
    new.tdict = data['tdict']
    new.ldict = data['ldict']
    new.word_freq_list = data['word_freq_list']
    return new

def as_dict(self):
    """
    Representation of FeatureMapper object as attribute dictionary.
    """
    return {
        'wdict': self.wdict,
        'word_freq': self.word_freq,
        'tdict': self.tdict,
        'ldict': self.ldict,
        'word_freq_list': self.word_freq_list
    }

def save_json(self, filename):
    """
    Save FeatureMapper object as file in JSON format.
    """
    with open(filename, 'w') as fh:
        json.dump(self.as_dict(), fh)

@staticmethod
def load_json(filename):
    """
    Load FeatureMapper object from file in JSON format.
    """
    with open(filename) as fh:
        data = json.load(fh, object_pairs_hook=OrderedDict)
    return FeatureMapper.from_dict(data)

def total_words(self):
    """
    Total words in vocabulary.
    """
    return len(self.wdict)

def total_tags(self):
Total POS tags indexed.

```python
return len(self.tdict)
```
def sentence_sequences(self, sentence):
    """
    Array of indices for words and tags.
    """
    sentence = ([FeatureMapper.START, FeatureMapper.START] + sentence +
               [FeatureMapper.STOP, FeatureMapper.STOP])

    words = [self.wdict[w]
             if w in self.wdict else self.wdict[FeatureMapper.UNK]
             for (w, t) in sentence]
    tags = [self.tdict[t]
            if t in self.tdict else self.tdict[FeatureMapper.UNK]
            for (w, t) in sentence]

    w = np.array(words).astype('int32')
    t = np.array(tags).astype('int32')

    return w, t

def gold_data(self, reftree):
    """
    Static oracle for tree.
    """
    w, t = self.sentence_sequences(reftree.sentence)
    (s_features, l_features) = Parser.training_data(reftree)

    struct_data = {
                   for (features, action) in s_features:  
                   struct_data[features] = self.s_action_index(action)
                   }

    label_data = {
                   for (features, action) in l_features:  
                   label_data[features] = self.l_action_index(action)
                   }

    return {'tree': reftree,
            'w': w,
            't': t,
            'struct_data': struct_data,
            'label_data': label_data,
            }

def gold_data_from_file(self, fname):
    """
    Static oracle for file.
    """
trees = PhraseTree.load_treefile(fname)
result = []
for tree in trees:
    sentence_data = self.gold_data(tree)
    result.append(sentence_data)
return result
A.2  main.py

#!/usr/bin/python

"""
main.py
Command-line interface for Span-Based Constituency Parser.
"""

from __future__ import print_function
from __future__ import division

import sys
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser(prog='Span-Based Constituency Parser')
    parser.add_argument(
        '--model',
        dest='model',
        help='File to save or load model.',
    )
    parser.add_argument(
        '--train',
        dest='train',
        help='Training trees. PTB (parenthetical) format.',
    )
    parser.add_argument(
        '--test',
        dest='test',
        help=(
            'Evaluation trees. PTB (parenthetical) format.'
            ' Omit for training.'
        ),
    )
    parser.add_argument(
        '--dev',
        dest='dev',
        help=(
            'Validation trees. PTB (parenthetical) format.'
            ' Required for training'
        ),
    )
    parser.add_argument(
        '--vocab',
        dest='vocab',
        help='JSON file from which to load vocabulary.',
    )
    parser.add_argument(
        '--write-vocab',
        dest='vocab_output',
        help='Destination to save vocabulary from training data.'
    )
    parser.add_argument(
        '--word-dims',
        ...)
dest='word_dims',
type=int,
default=50,
help='Embedding dimensions for word forms. (DEFAULT=50)',
}
parser.add_argument(
    '--tag-dims',
dest='tag_dims',
type=int,
default=20,
help='Embedding dimensions for POS tags. (DEFAULT=20)',
}
parser.add_argument(
    '--lstm-units',
dest='lstm_units',
type=int,
default=200,
help='Number of LSTM units in each layer/direction. (DEFAULT=200)',
}
parser.add_argument(
    '--hidden-units',
dest='hidden_units',
type=int,
default=200,
help='Number of hidden units for each FC ReLU layer. (DEFAULT=200)',
}
parser.add_argument(
    '--epochs',
dest='epochs',
type=int,
default=10,
help='Number of training epochs. (DEFAULT=10)',
}
parser.add_argument(
    '--batch-size',
dest='batch_size',
type=int,
default=10,
help='Number of sentences per training update. (DEFAULT=10)',
}
parser.add_argument(
    '--droprate',
dest='droprate',
type=float,
default=0.5,
help='Dropout probability. (DEFAULT=0.5)',
}
parser.add_argument(
    '--unk-param',
dest='unk_param',
type=float,
default=0.8375,
help='Parameter z for random UNKing. (DEFAULT=0.8375)',
}
parser.add_argument(
    '--alpha',
dest='alpha',
parser.add_argument('('--beta',
    dest='beta',
    type=float,
    default=0,
    help='Probability of using oracle action in exploration. (DEFAULT=0)'
)
)
parser.add_argument('--cnn-seed', type=int, dest='cnn_seed')
parser.add_argument('--np-seed', type=int, dest='np_seed')

args = parser.parse_args()

if args.vocab is not None:
    from features import FeatureMapper
    fm = FeatureMapper.load_json(args.vocab)
elif args.train is not None:
    from features import FeatureMapper
    fm = FeatureMapper(args.train)
    if args.vocab_output is not None:
        fm.save_json(args.vocab_output)
        print('Wrote vocabulary file {}'.format(args.vocab_output))
    sys.exit()
else:
    print('Must specify either --vocab-file or --train-data.')
    print(' (Use -h or --help flag for full option list.)')
    sys.exit()

if args.model is None:
    print('Must specify --model or (or --write-vocab) parameter.')
    print(' (Use -h or --help flag for full option list.)')
    sys.exit()

if args.test is not None:
    from phrase_tree import PhraseTree
    from network import Network
    from parser import Parser

    test_trees = PhraseTree.load_treefile(args.test)
    print('Loaded test trees from {}'.format(args.test))
    network = Network.load(args.model)
    print('Loaded model from: {}'.format(args.model))
    accuracy = Parser.evaluate_corpus(test_trees, fm, network)
    print('Accuracy: {}'.format(accuracy))
elif args.train is not None:
    from network import Network
    if args.np_seed is not None:
        import numpy as np
        np.random.seed(args.np_seed)
    if args.cnn_seed is not None:
        import pycnn
Network.train(
    feature_mapper=fm,
    word_dims=arg.word_dims,
    tag_dims=arg.tag_dims,
    lstm_units=arg.lstm_units,
    hidden_units=arg.hidden_units,
    epochs=arg.epochs,
    batch_size=arg.batch_size,
    train_data_file=arg.train,
    dev_data_file=arg.dev,
    model_save_file=arg.model,
    droprate=arg.droprate,
    unk_param=arg.unk_param,
    alpha=arg.alpha,
    beta=arg.beta,
)

A.3 network.py

"""
network.py

Bi-LSTM network learning/decoding model for span-based constituency parsing.
"""

from __future__ import print_function
from __future__ import division

import time
import random
import sys
import pycnn
import numpy as np

from phrase_tree import PhraseTree, FScore
from features import FeatureMapper
from parser import Parser

class LSTM(object):
    """
    LSTM class with initial state as parameter, and all parameters
    initialized in [-0.01, 0.01].
    """

    number = 0

    def __init__(self, input_dims, output_dims, model):
        self.input_dims = input_dims
        self.output_dims = output_dims
        self.model = model
        LSTM.number += 1
        self.name = 'lstm_{}'.format(LSTM.number)

        self.model.add_parameters(
            '{}_W_i'.format(self.name),
            (output_dims, input_dims + output_dims),
        )
        self.model.add_parameters(
            '{}_b_i'.format(self.name),
            output_dims,
        )

        self.model.add_parameters(
            '{}_W_f'.format(self.name),
            (output_dims, input_dims + output_dims),
        )
        self.model.add_parameters(
            '{}_b_f'.format(self.name),
            output_dims,
        )
```python
self.model.add_parameters(
    '{}_W_c'.format(self.name),
    (output_dims, input_dims + output_dims),
)
self.model.add_parameters(
    '{}_b_c'.format(self.name),
    output_dims,
)
self.model.add_parameters(
    '{}_W_o'.format(self.name),
    (output_dims, input_dims + output_dims),
)
self.model.add_parameters(
    '{}_b_o'.format(self.name),
    output_dims,
)
self.model.add_parameters(
    '{}_c0'.format(self.name),
    output_dims,
)

self.W_i = self.model[ '{}_W_i'.format(self.name) ]
self.b_i = self.model[ '{}_b_i'.format(self.name) ]
self.W_f = self.model[ '{}_W_f'.format(self.name) ]
self.b_f = self.model[ '{}_b_f'.format(self.name) ]
self.W_c = self.model[ '{}_W_c'.format(self.name) ]
self.b_c = self.model[ '{}_b_c'.format(self.name) ]
self.W_o = self.model[ '{}_W_o'.format(self.name) ]
self.b_o = self.model[ '{}_b_o'.format(self.name) ]
self.c0 = self.model[ '{}_c0'.format(self.name) ]

self.W_i.load_array(np.random.uniform(-0.01, 0.01, self.W_i.shape()))
self.b_i.load_array(np.zeros(self.b_i.shape()))
self.W_f.load_array(np.random.uniform(-0.01, 0.01, self.W_f.shape()))
self.b_f.load_array(np.zeros(self.b_f.shape()))
self.W_c.load_array(np.random.uniform(-0.01, 0.01, self.W_c.shape()))
self.b_c.load_array(np.zeros(self.b_c.shape()))
self.W_o.load_array(np.random.uniform(-0.01, 0.01, self.W_o.shape()))
self.b_o.load_array(np.zeros(self.b_o.shape()))
self.c0.load_array(np.zeros(self.c0.shape()))

class State(object):
    def __init__(self, lstm):
        self.lstm = lstm
        self.outputs = []
        self.c = pycnn.parameter(self.lstm.c0)
        self.h = pycnn.tanh(self.c)
        self.W_i = pycnn.parameter(self.lstm.W_i)
```

self.b_i = pycnn.parameter(self.lstm.b_i)
self.W_f = pycnn.parameter(self.lstm.W_f)
self.b_f = pycnn.parameter(self.lstm.b_f)
self.W_c = pycnn.parameter(self.lstm.W_c)
self.b_c = pycnn.parameter(self.lstm.b_c)
self.W_o = pycnn.parameter(self.lstm.W_o)
self.b_o = pycnn.parameter(self.lstm.b_o)

def add_input(self, input_vec):
    """
    Note that this function updates the existing State object!
    """
    x = pycnn.concatenate([input_vec, self.h])
    i = pycnn.logistic(self.W_i * x + self.b_i)
    f = pycnn.logistic(self.W_f * x + self.b_f)
    g = pycnn.tanh(self.W_c * x + self.b_c)
    o = pycnn.logistic(self.W_o * x + self.b_o)
    c = pycnn.cwise_multiply(f, self.c) + pycnn.cwise_multiply(i, g)
    h = pycnn.cwise_multiply(o, pycnn.tanh(c))
    self.c = c
    self.h = h
    self.outputs.append(h)
    return self

def output(self):
    return self.outputs[-1]

def initial_state(self):
    return LSTM.State(self)

class Network(object):
    """
    This class holds architectural parameters as well as dropout
    (for learning).
    """
    def __init__(self,
        self,
        word_count,
        tag_count,
        word_dims,
        tag_dims,
        lstm_units,
        hidden_units,
        struct_out,
label_out,
  droprate=0,
  struct_spans=4,
  label_spans=3,
):  
  self.word_count = word_count
  self.tag_count = tag_count
  self.word_dims = word_dims
  self.tag_dims = tag_dims
  self.lstm_units = lstm_units
  self.hidden_units = hidden_units
  self.struct_out = struct_out
  self.label_out = label_out
  self.droprate = droprate
  self.model = pycnn.Model()
  self.trainer = pycnn.AdadeltaTrainer(self.model, lam=0, eps=1e-7, rho=0.99)
  random.seed(1)
  self.activation = pycnn.rectify
  self.model.add_lookup_parameters('word-embed', (word_count, word_dims))
  self.model.add_lookup_parameters('tag-embed', (tag_count, tag_dims))
  self.fwd_lstm1 = LSTM(word_dims + tag_dims, lstm_units, self.model)
  self.back_lstm1 = LSTM(word_dims + tag_dims, lstm_units, self.model)
  self.fwd_lstm2 = LSTM(2 * lstm_units, lstm_units, self.model)
  self.back_lstm2 = LSTM(2 * lstm_units, lstm_units, self.model)
  self.model.add_parameters(
    'struct-hidden-W',
    (hidden_units, 4 * struct_spans * lstm_units),
  )
  self.model.add_parameters('struct-hidden-b', hidden_units)
  self.model.add_parameters('struct-out-W', (struct_out, hidden_units))
  self.model.add_parameters('struct-out-b', struct_out)
  self.model.add_parameters(
    'label-hidden-W',
    (hidden_units, 4 * label_spans * lstm_units),
  )
  self.model.add_parameters('label-hidden-b', hidden_units)
  self.model.add_parameters('label-out-W', {label_out, hidden_units})
  self.model.add_parameters('label-out-b', label_out)

  def init_params(self):
    ""
    Called once to provide intital random values for network parameters.
    """
self.model[‘word-embed’].init_from_array(np.random.uniform(-0.01, 0.01, self.model[‘word-embed’].shape()),)
self.model[‘tag-embed’].init_from_array(np.random.uniform(-0.01, 0.01, self.model[‘tag-embed’].shape()),)

shape = self.model[‘struct-hidden-W’].shape()
r = np.sqrt(6./(shape[0] + shape[1]))
self.model[‘struct-hidden-W’].load_array(np.random.uniform(-r, r, shape),)
self.model[‘struct-hidden-b’].load_array(np.zeros(self.model[‘struct-hidden-b’].shape()),)

self.model[‘struct-out-W’].load_array(np.zeros(self.model[‘struct-out-W’].shape()),
self.model[‘struct-out-b’].load_array(np.zeros(self.model[‘struct-out-b’].shape()),

shape = self.model[‘label-hidden-W’].shape()
r = np.sqrt(6./(shape[0] + shape[1]))
self.model[‘label-hidden-W’].load_array(np.random.uniform(-r, r, shape),
self.model[‘label-hidden-b’].load_array(np.zeros(self.model[‘label-hidden-b’].shape()),

self.model[‘label-out-W’].load_array(np.zeros(self.model[‘label-out-W’].shape()),
self.model[‘label-out-b’].load_array(np.zeros(self.model[‘label-out-b’].shape()),

def prep_params(self):
    """
    Generates PyCNN expressions from model parameters
    for one instance of training or inference.
    """
    self.W1_struct = pycnn.parameter(self.model[‘struct-hidden-W’])
self.b1_struct = pycnn.parameter(self.model[‘struct-hidden-b’])
self.W2_struct = pycnn.parameter(self.model[‘struct-out-W’])
self.b2_struct = pycnn.parameter(self.model[‘struct-out-b’])
self.W1_label = pycnn.parameter(self.model[‘label-hidden-W’])
self.b1_label = pycnn.parameter(self.model['label-hidden-b'])
self.W2_label = pycnn.parameter(self.model['label-out-W'])
self.b2_label = pycnn.parameter(self.model['label-out-b'])

def evaluate_recurrent(self, word_inds, tag_inds, test=False):
    
    Generate expressions for recurrent outputs given the indices of words and tags making up a sentence.
    
    fwd1 = self.fwd_lstm1.initial_state()
    back1 = self.back_lstm1.initial_state()
    fwd2 = self.fwd_lstm2.initial_state()
    back2 = self.back_lstm2.initial_state()
    
    sentence = []
    for (w, t) in zip(word_inds, tag_inds):
        wordvec = pycnn.lookup(self.model['word-embed'], w)
        tagvec = pycnn.lookup(self.model['tag-embed'], t)
        vec = pycnn.concatenate([wordvec, tagvec])
        sentence.append(vec)
    
    fwd1_out = []
    for vec in sentence:
        fwd1 = fwd1.add_input(vec)
        fwd_vec = fwd1.output()
        fwd1_out.append(fwd_vec)
    
    back1_out = []
    for vec in reversed(sentence):
        back1 = back1.add_input(vec)
        back_vec = back1.output()
        back1_out.append(back_vec)
    
    lstm2_input = []
    for (f, b) in zip(fwd1_out, reversed(back1_out)):
        lstm2_input.append(pycnn.concatenate([f, b]))
    
    fwd2_out = []
    for vec in lstm2_input:
        if self.droprate > 0 and not test:
            vec = pycnn.dropout(vec, self.droprate)
        fwd2 = fwd2.add_input(vec)
        fwd_vec = fwd2.output()
        fwd2_out.append(fwd_vec)
    
    back2_out = []
    for vec in reversed(lstm2_input):
        if self.droprate > 0 and not test:
            vec = pycnn.dropout(vec, self.droprate)
        back2 = back2.add_input(vec)
        back_vec = back2.output()
        back2_out.append(back_vec)
fwd_out = [pycnn.concatenate([f1, f2]) for (f1, f2) in zip(fwd1_out, fwd2_out)]
back_out = [pycnn.concatenate([b1, b2]) for (b1, b2) in zip(back1_out, back2_out)]
return fwd_out, back_out[::-1]

def evaluate_struct(self, fwd_out, back_out, lefts, rights, test=False):
    
    Generate raw (linear) scores for each structural action given sentence
    recurrent representation and indices of span features.
    
    fwd_span_out = []
    for left_index, right_index in zip(lefts, rights):
        fwd_span_out.append(fwd_out[right_index] - fwd_out[left_index - 1])
    fwd_span_vec = pycnn.concatenate(fwd_span_out)
    
    back_span_out = []
    for left_index, right_index in zip(lefts, rights):
        back_span_out.append(back_out[left_index] - back_out[right_index + 1])
    back_span_vec = pycnn.concatenate(back_span_out)
    
    hidden_input = pycnn.concatenate([fwd_span_vec, back_span_vec])
    if self.droprate > 0 and not test:
        hidden_input = pycnn.dropout(hidden_input, self.droprate)
    hidden_output = self.activation(self.W1_struct * hidden_input + self.b1_struct)
    scores = (self.W2_struct * hidden_output + self.b2_struct)
    return scores

def evaluate_label(self, fwd_out, back_out, lefts, rights, test=False):
    
    Generate raw (label) scores for each structural action given sentence
    recurrent representation and indices of span features.
    
    fwd_span_out = []
    for left_index, right_index in zip(lefts, rights):
        fwd_span_out.append(fwd_out[right_index] - fwd_out[left_index - 1])
    fwd_span_vec = pycnn.concatenate(fwd_span_out)
    
    back_span_out = []
    for left_index, right_index in zip(lefts, rights):
        back_span_out.append(back_out[left_index] - back_out[right_index + 1])
    back_span_vec = pycnn.concatenate(back_span_out)
    
    hidden_input = pycnn.concatenate([fwd_span_vec, back_span_vec])
    if self.droprate > 0 and not test:
        hidden_input = pycnn.dropout(hidden_input, self.droprate)
    hidden_output = self.activation(self.W1_label * hidden_input + self.b1_label)
scores = (self.W2_label * hidden_output + self.b2_label)

return scores

def save(self, filename):
    """
    Appends architecture hyperparameters to end of PyCNN model file.
    """
    self.model.save(filename)

    with open(filename, 'a') as f:
        f.write('

        f.write('word_count = {}
        f.write('tag_count = {}
        f.write('word_dims = {}
        f.write('tag_dims = {}
        f.write('lstm_units = {}
        f.write('hidden_units = {}
        f.write('struct_out = {}
        f.write('label_out = {}

@staticmethod
def load(filename):
    """
    Loads file created by save() method.
    """
    with open(filename) as f:
        f.readline()
        f.readline()
        word_count = int(f.readline().split()[-1])
        tag_count = int(f.readline().split()[-1])
        word_dims = int(f.readline().split()[-1])
        tag_dims = int(f.readline().split()[-1])
        lstm_units = int(f.readline().split()[-1])
        hidden_units = int(f.readline().split()[-1])
        struct_out = int(f.readline().split()[-1])
        label_out = int(f.readline().split()[-1])

        network = Network(
            word_count=word_count,
            tag_count=tag_count,
            word_dims=word_dims,
            tag_dims=tag_dims,
            lstm_units=lstm_units,
            hidden_units=hidden_units,
            struct_out=struct_out,
            label_out=label_out,
        )

        network.model.load(filename)

        return network

@staticmethod
def train(
feature_mapper,
word_dims,
tag_dims,
lstm_units,
hidden_units,
ePOCHs,
batch_size,
train_data_file,
deV_data_file,
model_save_file,
droprate,
unk_param,
alpha=1.0,
beta=0.0,
):
    ""
    Train a model given FeatureMapper object, file names from which to load
    training and dev trees, and full set of hyperparameters.
    ""
    start_time = time.time()
    fm = feature_mapper
    word_count = fm.total_words()
    tag_count = fm.total_tags()

    network = Network(
        word_count=word_count,
        tag_count=tag_count,
        word_dims=word_dims,
        tag_dims=tag_dims,
        lstm_units=lstm_units,
        hidden_units=hidden_units,
        struct_out=2,
        label_out=fm.total_label_actions(),
        droprate=droprate,
    )
    network.init_params()

    print('Hidden units: {}, per-LSTM units: {}'.format(
        hidden_units,
        lstm_units,
    ))
    print('Embeddings: word={} tag={}'.format(
        word_count, word_dims,
        tag_count, tag_dims,
    ))
    print('Dropout rate: {}'.format(droprate))
    print('Parameters initialized in [-0.01, 0.01]')
    print('Random UNKing parameter z = {}'.format(unk_param))
    print('Exploration: alpha={} beta={}'.format(alpha, beta))

    training_data = fm.gold_data_from_file(train_data_file)
    num_batches = -(-len(training_data) // batch_size)
    print('Loaded {} training sentences ({} batches of size {})!'.format(
        len(training_data),
        num_batches,
        batch_size,
    )
parse_every = -(-num_batches // 4)

dev_trees = PhraseTree.load_treefile(dev_data_file)
print('Loaded {} validation trees!'.format(len(dev_trees)))
best_acc = FScore()

for epoch in xrange(1, epochs + 1):
    print('........... epoch {} ...........'.format(epoch))
    total_cost = 0.0
    total_states = 0
    training_acc = FScore()

    np.random.shuffle(training_data)

    for b in xrange(num_batches):
        batch = training_data[(b * batch_size) : ((b + 1) * batch_size)]

        explore = [
            Parser.exploration(
                example,
                fm,
                network,
                alpha=alpha,
                beta=beta,
            )
            for example in batch
        ]

        for (_, acc) in explore:
            training_acc += acc

        batch = [example for (example, _) in explore]

        pycnn.renew_cg()
        network.prep_params()

        errors = []

        for example in batch:

            ## random UNKing ##
            for (i, w) in enumerate(example['w']):
                if w <= 2:
                    continue

                freq = fm.word_freq_list[w]
                drop_prob = unk_param / (unk_param + freq)
                r = np.random.random()
                if r < drop_prob:
                    example['w'][i] = 0

            fwd, back = network.evaluate_recurrent(
                example['w'],
                example['t'],
            )
for (left, right), correct in example['struct_data'].items():
    scores = network.evaluate_struct(fwd, back, left, right)
    probs = pycnn.softmax(scores)
    loss = -pycnn.log(pycnn.pick(probs, correct))
    errors.append(loss)
    total_states += len(example['struct_data'])

for (left, right), correct in example['label_data'].items():
    scores = network.evaluate_label(fwd, back, left, right)
    probs = pycnn.softmax(scores)
    loss = -pycnn.log(pycnn.pick(probs, correct))
    errors.append(loss)
    total_states += len(example['label_data'])

batch_error = pycnn.esum(errors)
total_cost += batch_error.scalar_value()
network.trainer.update()

mean_cost = total_cost / total_states

print("
Batch {} Mean Cost {:.4f} [Train: {}]".format(b, mean_cost, training_acc),
end="",
)
sys.stdout.flush()

if ((b + 1) % parse_every) == 0 or b == (num_batches - 1):
    dev_acc = Parser.evaluate_corpus(dev_trees, fm, network,
)
    print(" [Val: {}]").format(dev_acc)
    if dev_acc > best_acc:
        best_acc = dev_acc
        network.save(model_save_file)
        print(" [saved model: {}]").format(model_save_file)

current_time = time.time()
runmins = (current_time - start_time)/60.
print(" Elapsed time: {:.2f}m").format(runmins)}
A.4 parser.py

from __future__ import print_function
from __future__ import division
import numpy as np
import pycnn
from collections import defaultdict
from phrase_tree import PhraseTree, FScore

class Parser(object):
    def __init__(self, n):
        self.n = n
        self.i = 0
        self.stack = []

    def can_shift(self):
        return (self.i < self.n)

    def can_combine(self):
        return (len(self.stack) > 1)

    def shift(self):
        j = self.i  # (index of shifted word)
        treelet = PhraseTree(leaf=j)
        self.stack.append((j, j, [treelet]))
        self.i += 1

    def combine(self):
        (_, right, treelist0) = self.stack.pop()
        (left, _, treelist1) = self.stack.pop()
        self.stack.append((left, right, treelist1 + treelist0))

    def label(self, nonterminals=[]):
        # Assigns label to current node given list of non-terminal symbols (bottom to top).
```python
for nt in nonterminals:
    (left, right, trees) = self.stack.pop()
    tree = PhraseTree(symbol=nt, children=trees)
    self.stack.append((left, right, [tree]))

def take_action(self, action):
    if action == 'sh':
        self.shift()
    elif action == 'comb':
        self.combine()
    elif action == 'none':
        return
    elif action.startswith('label-'):
        self.label(action[6:].split('-'))
    else:
        raise RuntimeError('Invalid Action: {}'.format(action))

def finished(self):
    return
    (self.i == self.n) and
    (len(self.stack) == 1) and
    (len(self.stack[0][2]) == 1)

def tree(self):
    if not self.finished():
        raise RuntimeError('Not finished.')
    return self.stack[0][2][0]

def s_features(self):
    """
    Features for predicting structural action (shift, combine):
      (pre-s1-span, s1-span, s0-span, post-s0-span)
    Note features use 1-based indexing:
      ... a span of (1, 1) means the first word of sentence
      ... (x, x-1) means no span
    """
    lefts = []
    rights = []

    # pre-s1-span
    lefts.append(1)
    if len(self.stack) < 2:
        rights.append(0)
    else:
        s1_left = self.stack[-2][0] + 1
        rights.append(s1_left - 1)

    # s1-span
    if len(self.stack) < 2:
        lefts.append(1)
```

rights.append(0)

else:
    s1_left = self.stack[-2][0] + 1
    lefts.append(s1_left)
    s1_right = self.stack[-2][1] + 1
    rights.append(s1_right)

# s0-span
if len(self.stack) < 1:
    lefts.append(1)
    rights.append(0)
else:
    s0_left = self.stack[-1][0] + 1
    lefts.append(s0_left)
    s0_right = self.stack[-1][1] + 1
    rights.append(s0_right)

# post-s0-span
lefts.append(self.i + 1)
rights.append(self.n)

return tuple(lefts), tuple(rights)

def l_features(self):
    
    """
    Features for predicting label action:
    (pre-s0-span, s0-span, post-s0-span)
    """
    lefts = []
    rights = []

    # pre-s0-span
    lefts.append(1)
    if len(self.stack) < 1:
        rights.append(0)
    else:
        s0_left = self.stack[-1][0] + 1
        rights.append(s0_left - 1)

    # s0-span
    if len(self.stack) < 1:
        lefts.append(1)
        rights.append(0)
    else:
        s0_left = self.stack[-1][0] + 1
        lefts.append(s0_left)
        s0_right = self.stack[-1][1] + 1
        rights.append(s0_right)

    # post-s0-span
    lefts.append(self.i + 1)
    rights.append(self.n)

    return tuple(lefts), tuple(rights)
def s_oracle(self, tree):
    ""
    Returns correct structural action in current (arbitrary) state,
    given gold tree.
    Deterministic (prefer combine).
    """
    if not self.can_shift():
        return 'comb'
    elif not self.can_combine():
        return 'sh'
    else:
        (left0, right0, _) = self.stack[-1]
        a, _ = tree.enclosing(left0, right0)
        if a == left0:
            return 'sh'
        else:
            return 'comb'

def l_oracle(self, tree):
    ""
    Returns correct label action in current (arbitrary) state,
    given gold tree.
    """
    (left0, right0, _) = self.stack[-1]
    labels = tree.span_labels(left0, right0)[::-1]
    if len(labels) == 0:
        return 'none'
    else:
        return 'label-' + '-'.join(labels)

@staticmethod
def gold_actions(tree):
    ""
    Returns the static oracle sequence of correct actions for a gold tree.
    ""
    n = len(tree.sentence)
    state = Parser(n)
    result = []
    for step in range(2 * n - 1):
        if state.can_combine():
            (left0, right0, _) = state.stack[-1]
            (left1, _, _) = state.stack[-2]
            a, b = tree.enclosing(left0, right0)
            if left1 >= a:
                result.append('comb')
                state.combine()
            else:
                result.append('sh')
                state.shift()
        else:
            result.append('sh')
state.shift()
(left0, right0, _) = state.stack[-1]
labels = tree.span_labels(left0, right0)[::-1]
if len(labels) == 0:
    result.append('none')
else:
    result.append('label-' + '-'.join(labels))
state.label(labels)

return result

@staticmethod
def training_data(tree):
    ""
    Using oracle (for gold sequence), omitting mandatory S-actions
    ""
    s_features = []
    l_features = []

    n = len(tree.sentence)
    state = Parser(n)
    result = []

    for step in range(2 * n - 1):
        if not state.can_combine():
            action = 'sh'
        elif not state.can_shift():
            action = 'comb'
        else:
            action = state.s_oracle(tree)
            features = state.s_features()
            s_features.append((features, action))
            state.take_action(action)

        action = state.l_oracle(tree)
        features = state.l_features()
        l_features.append((features, action))
        state.take_action(action)

    return (s_features, l_features)

@staticmethod
def exploration(data, fm, network, alpha=1.0, beta=0):
    ""
    Only data from this parse, including mandatory S-actions.
    Follow softmax distribution for structural data.
    data is as returned by training_data() method.
    fm is FeatureMapper object (features.py).
    network is Network object (network.py).
    alpha is flattening parameter for softmax exploration.
    beta is probability of taking oracle action
    ""
(rather than exploration by sampling).

```python
pycnn.renew_cg()
network.prep_params()

struct_data = {}
labeled_data = {}

tree = data['tree']
sentence = tree.sentence

n = len(sentence)
state = Parser(n)

w = data['w']
t = data['t']
fwd, back = network.evaluate_recurrent(w, t, test=True)

for step in xrange(2 * n - 1):
    features = state.s_features()
    if not state.can_combine():
        action = 'sh'
        correct_action = 'sh'
    elif not state.can_shift():
        action = 'comb'
        correct_action = 'comb'
    else:
        correct_action = state.s_oracle(tree)

    r = np.random.random()
    if r < beta:
        action = correct_action
    else:
        left, right = features
        scores = network.evaluate_struct(
            fwd,
            back,
            left,
            right,
            test=True,
        ).npvalue()

        # sample from distribution
        exp = np.exp(scores * alpha)
        softmax = exp / (exp.sum())
        r = np.random.random()

        if r <= softmax[0]:
            action = 'sh'
        else:
            action = 'comb'

    struct_data[features] = fm.s_action_index(correct_action)
    state.take_action(action)
```
features = state.l_features()
correct_action = state.l_oracle(tree)
label_data[features] = fm.l_action_index(correct_action)

r = np.random.random()
if r < beta:
    action = correct_action
else:
    left, right = features
    scores = network.evaluate_label(fwd, back, left, right, test=True).npvalue()
    if step < (2 * n - 2):
        action_index = np.argmax(scores)
    else:
        action_index = 1 + np.argmax(scores[1:])
    action = fm.l_action(action_index)
state.take_action(action)
predicted = state.stack[0][2][0].propagate_sentence(sentence)
accuracy = predicted.compare(tree)

example = {
    'w': w,
    't': t,
    'struct_data': struct_data,
    'label_data': label_data,
}

return example, accuracy

@staticmethod
def parse(sentence, fm, network):
    """
    Inference using a trained model.
    sentence is list of (word, POS) tuples, word and POS are strings.
    fm is FeatureMapper object (features.py).
    network is Network object (network.py).
    """
    pycnn.renew_cg()
    network.prep_params()

    n = len(sentence)
    state = Parser(n)

    w, t = fm.sentence_sequences(sentence)
fwd, back = network.evaluate_recurrent(w, t, test=True)

for step in xrange(2 * n - 1):
    if not state.can_combine():
        action = 'sh'
    elif not state.can_shift():
        action = 'comb'
    else:
        left, right = state.s_features()
        scores = network.evaluate_struct(
            fwd,
            back,
            left,
            right,
            test=True,
        ).npvalue()
        action_index = np.argmax(scores)
        action = fm.s_action(action_index)
        state.take_action(action)

        left, right = state.l_features()
        scores = network.evaluate_label(
            fwd,
            back,
            left,
            right,
            test=True,
        ).npvalue()
        if step < (2 * n - 2):
            action_index = np.argmax(scores)
        else:
            action_index = 1 + np.argmax(scores[1:]):
        action = fm.l_action(action_index)
        state.take_action(action)

    if not state.finished():
        raise RuntimeError('Bad ending state!')

    tree = state.stack[0][2][0]
    tree.propagate_sentence(sentence)
    return tree

@staticmethod
def evaluate_corpus(trees, fm, network):
    """
    Returns FScore object (phrase_tree.py).
    """
    accuracy = FScore()
    for tree in trees:
        predicted = Parser.parse(tree.sentence, fm, network)
        local_accuracy = predicted.compare(tree)
        accuracy += local_accuracy
    return accuracy
@staticmethod
def write_predicted(fname, test_trees, fm, network):
    
    Input trees being used only to carry sentences.
    
    f = open(fname, 'w')
    for tree in test_trees:
        predicted = Parser.parse(tree.sentence, fm, network)
        topped = PhraseTree(
            symbol='TOP',
            children=[predicted],
            sentence=predicted.sentence,
        )
        f.write(str(topped))
        f.write('
'  
    f.close()
A.5 phrase_tree.py

```python
from __future__ import print_function
from __future__ import division
from collections import defaultdict

class PhraseTree(object):
    ""
    Provides a recursive representation of a constituency tree
    for a natural-language sentence.
    ""
    puncs = [",", ".", ":", "\"", "\"", ",", ":", "\"", "\"", "PU"] # (COLLINS.prm)

def __init__(self, symbol=None, children=[], sentence=[], leaf=None):
    self.symbol = symbol # label at top node
    self.children = children # list of PhraseTree objects
    self.sentence = sentence # word at bottom level else None
    self.leaf = leaf
    self._str = None

def __str__(self):
    ""
    String representation of tree in parenthetical recursive format.
    ""
    if self._str is None:
        if len(self.children) != 0:
            childstr = ' '.join(str(c) for c in self.children)
            self._str = '( {} {})'.format(self.symbol, childstr)
        else:
            self._str = '( {} {})'.format(self.sentence[self.leaf][1],
                                           self.sentence[self.leaf][0],
                                       )
    return self._str

def propagate_sentence(self, sentence):
    ""
    Recursively assigns sentence (list of (word, POS) pairs)
    ```
to all nodes of a tree.

```python
def pretty(self, level=0, marker=' '):
    """
    String representation with indentation.
    """
    pad = marker + level
    if self.leaf is not None:
        leaf_string = '{(} {})'.format(
            self.symbol,
            self.sentence[self.leaf][0],
        )
        return pad + leaf_string
    else:
        result = pad + '(' + self.symbol
        for child in self.children:
            result += '

' + child.pretty(level + 1)
        result += ')
        return result
```

```python
@staticmethod
def parse(line):
    """
    Loads a tree from a sting in PTB parenthetical format.
    """
    line += " "
    sentence = []
    _, t = PhraseTree._parse(line, 0, sentence)
    if t.symbol == 'TOP' and len(t.children) == 1:
        t = t.children[0]
    return t
```

```python
@staticmethod
def _parse(line, index, sentence):
    """
    Helper function called recursively by parse()
    """
    assert line[index] == '(', "Invalid tree string () at {}".format(line, index)
    index += 1
    symbol = None
    children = []
    leaf = None
    while line[index] != '):
        index += 1
        symbol = line[index]
        if symbol == '(':
            index += 1
            children += _parse(line, index, sentence)
            index += 1
        elif symbol == ')':
            break
    return leaf, children
```
if line[index] == '(':  
    index, t = PhraseTree._parse(line, index, sentence)  
    children.append(t)
else:
    if symbol is None:
        # symbol is here!
        rpos = min(line.find(' ', index), line.find(')', index))  
        # see above N.B. (find could return -1)
        symbol = line[index:rpos]  
        # (word, tag) string pair
        index = rpos
    else:
        rpos = line.find(')', index)
        word = line[index:rpos]
        sentence.append((word, symbol))
        leaf = len(sentence) - 1
        index = rpos
    if line[index] == ' ':
        index += 1
assert line[index] == ')', "Invalid tree string %s at %d" % (line, index)

    t = PhraseTree(
        symbol=symbol,
        children=children,
        sentence=sentence,
        leaf=leaf,
    )
return (index + 1), t

def left_span(self):
    """
    Returns index of left-boundary or arbitrary subtree.
    """
    try:
        return self._left_span
    except AttributeError:
        if self.leaf is not None:
            self._left_span = self.leaf
        else:
            self._left_span = self.children[0].left_span()
        return self._left_span

def right_span(self):
    """
    Returns index of right-boundary or arbitrary subtree.
    """
    try:
        return self._right_span
    except AttributeError:
        if self.leaf is not None:
            ...
def _right_span(self):
    if self.leaf:
        self._right_span = self.leaf
    else:
        self._right_span = self.children[-1].right_span()
    return self._right_span

def brackets(self, advp_prt=True, counts=None):
    """Returns dictionary with counts for each bracket of the form
    (nonterminal, left_index, right_index)"
    if counts is None:
        counts = defaultdict(int)
    if self.leaf is not None:
        return {}
    nonterm = self.symbol
    if advp_prt and nonterm=="PRT":
        nonterm = 'ADVP'
    left = self.left_span()
    right = self.right_span()
    # ignore punctuation
    while (left < len(self.sentence) and
           self.sentence[left][1] in PhraseTree.puncs):
        left += 1
    while (right > 0 and self.sentence[right][1] in PhraseTree.puncs):
        right -= 1
    if left <= right and nonterm != 'TOP':
        counts[(nonterm, left, right)] += 1
    for child in self.children:
        child.brackets(advp_prt=advp_prt, counts=counts)
    return counts

def phrase(self):
    """Return part of sentence list covered by subtree."
    if self.leaf is not None:
        return [(self.leaf, self.symbol)]
    else:
        result = []
        for child in self.children:
            result.extend(child.phrase())
        return result
@staticmethod
def load_treefile(fname):
    ""
    Returns list of trees from file with trees in PTB format, one per line.
    ""
    trees = []
    for line in open(fname):
        t = PhraseTree.parse(line)
        trees.append(t)
    return trees

def compare(self, gold, advp_prt=True):
    ""
    Comparison of a hypothesis tree (self) with gold tree.
    Returns FScore evaluation object.
    ""
    predbracks = self.brackets(advp_prt)
goldbracks = gold.brackets(advp_prt)
correct = 0
    for gb in goldbracks:
        if gb in predbracks:
            correct += min(goldbracks[gb], predbracks[gb])
pred_total = sum(predbracks.values())
gold_total = sum(goldbracks.values())
    return FScore(correct, pred_total, gold_total)

def enclosing(self, i, j):
    ""
    Returns the left and right indices of the labeled span in the tree
    which is next-larger than (i, j)
    (whether or not (i, j) is itself a labeled span)
    ""
    for child in self.children:
        left = child.left_span()
        right = child.right_span()
        if (left <= i) and (right >= j):
            if (left == i) and (right == j):
                break
            return child.enclosing(i, j)
    return (self.left_span(), self.right_span())

def span_labels(self, i, j):
    ""
    Returns a list of span labels (if any) for (i, j)
    ""
    if self.leaf is not None:
        return []
    if (self.left_span() == i) and (self.right_span() == j):
result = [self.symbol]
else:
    result = []

for child in self.children:
    left = child.left_span()
    right = child.right_span()
    if (left <= i) and (right >= j):
        result.extend(child.span_labels(i, j))
        break

return result

class FScore(object):
    """
    Object for tabulating parsing evaluation (F-measure only).
    """
    def __init__(self, correct=0, predcount=0, goldcount=0):
        self.correct = correct  # correct brackets
        self.predcount = predcount  # total predicted brackets
        self.goldcount = goldcount  # total gold brackets

    def precision(self):
        if self.predcount > 0:
            return (100.0 * self.correct) / self.predcount
        else:
            return 0.0

    def recall(self):
        if self.goldcount > 0:
            return (100.0 * self.correct) / self.goldcount
        else:
            return 0.0

    def fscore(self):
        precision = self.precision()
        recall = self.recall()
        if (precision + recall) > 0:
            return (2 * precision * recall) / (precision + recall)
        else:
            return 0.0

    def __str__(self):
        precision = self.precision()
        recall = self.recall()
        fscore = self.fscore()
        return '{P= {:0.2f}, R= {:0.2f}, F= {:0.2f}}'.format(
            precision, recall, fscore)
def __iadd__(self, other):
    self.correct += other.correct
    self.predcount += other.predcount
    self.goldcount += other.goldcount
    return self

def __add__(self, other):
    return FScore(
        self.correct + other.correct,
        self.predcount + other.predcount,
        self.goldcount + other.goldcount,
    )

def __cmp__(self, other):
    return cmp(self.fscore(), other.fscore())

@staticmethod
def parseval(gold_file, test_file):
    gold_trees = PhraseTree.load_treefile(gold_file)
    test_trees = PhraseTree.load_treefile(test_file)
    cumulative = FScore()
    for gold, test in zip(gold_trees, test_trees):
        acc = test.compare(gold, advp_prt=True)
        cumulative += acc
    return cumulative