# STRUCTURED LEARNING WITH INEXACT SEARCH: ADVANCES IN SHIFT-REDUCE CCG PARSING

Wenduan Xu

This dissertation is submitted to the Computer Laboratory
of the University of Cambridge in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
in
Computer Science
2017

# STRUCTURED LEARNING WITH INEXACT SEARCH: ADVANCES IN SHIFT-REDUCE CCG PARSING

by

Wenduan Xu

Submitted to the Computer Laboratory of the University of Cambridge
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

## Abstract

Statistical shift-reduce parsing involves the interplay of representation learning, structured learning, and inexact search. This dissertation considers approaches that tightly integrate these three elements and explores three novel models for shift-reduce CCG parsing. First, I develop a dependency model, in which the selection of shift-reduce action sequences producing a dependency structure is treated as a hidden variable; the key components of the model are a dependency oracle and a learning algorithm that integrates the dependency oracle, the structured perceptron, and beam search. Second, I present expected F-measure training and show how to derive a globally normalized RNN model, in which beam search is naturally incorporated and used in conjunction with the objective to learn shift-reduce action sequences optimized for the final evaluation metric. Finally, I describe an LSTM model that is able to construct parser state representations incrementally by following the shift-reduce syntactic derivation process; I show expected F-measure training, which is agnostic to the underlying neural network, can be applied in this setting to obtain globally normalized greedy and beam-search LSTM shift-reduce parsers.

## Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text. It is not substantially the same as any that I have submitted, or, is being concurrently submitted for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. I further state that no substantial part of my dissertation has already been submitted, or, is being concurrently submitted for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. It does not exceed $60,000$ words.

# Preface

This dissertation contains the following prior publications; I declare the main contributions contained in these works are my own.

1. Wenduan Xu, Stephen Clark, and Yue Zhang. 2014. Shift-Reduce CCG Parsing with a Dependency Model. In *Proc. of ACL*.

2. Wenduan Xu, Michael Auli, and Stephen Clark. 2015. CCG Supertagging with a Recurrent Neural Network. In *Proc. of ACL (Vol. 2)*.

3. Wenduan Xu, Michael Auli, and Stephen Clark. 2016. Expected F-measure Training for Shift-Reduce Parsing with Recurrent Neural Networks. In *Proc. of NAACL*.

4. Wenduan Xu. 2016. LSTM Shift-Reduce CCG Parsing. In *Proc. of EMNLP*.

## Acknowledgments

A series of random events have led to this thesis, which all started with the projects I did with the extraordinary hacker Philipp Koehn whose unmatched passion has definitely inspired me, and it was the Hiero paper (Chiang, 2007), which I read while working on syntax-based translation, propelled me into parsing.

To this end, I am grateful to Steve Clark (sc609) for allowing me to focus on parsing, although it was not in the best interests of the other projects. Nevertheless, Steve his given sound advice, valuable criticism and unique perspectives whenever I needed, and has been ever tolerant of my frequent, irrelevant and divergent explorations. I also owe a great deal to him for reading the often last-minute drafts, and for suggesting the topic of the very first project—the dependency model, which set my foot on CCG.

Among all the people I have interacted with, I have benefited from the many Skype discussions with Michael Auli, who suggested me to look into RNNs, and more importantly, helped me debug my thoughts from countless failed experiments, which were largely responsible for my so-very-late revelation about how XF1 training could be done for the parser while walking across a car park near ISI.

The summer stint at ISI was also memorable, and the whole natural language group is acknowledged for involving me in their research activities. In particular, I thank the heroes Daniel Marcu and Kevin Knight for showing unreserved openness to an external visitor, and many thanks to Sahil Garg, Ulf Hermjakob, Jon May, Ashish Vaswani and Barret Zoph for various discussions. Thanks also to Aliya Deri and Qing Dou for organizing all the fun outings, and to the amazing Peter Zamar—who would beat any future AI agents—for demonstrating efficiency in action and proving the great value of zero-bureaucracy no-nonsense administration. And special thanks to Juliane Kruck.

In addition, I want to thank Giorgio Satta—who virtually knows everything about parsing algorithms—for the teaching of CCG parsing over Skype (which sometimes required very patient and repeated explanations), and for showing me “An algorithm must be seen to be believed”.

Of course, many thanks to my examiners Ted Briscoe (ejb1) and Chris Dyer, for their enlightenment.

In the lab, I had the good fortune of meeting Laurent Simon (lmrs2), without whom my grad school would be incomplete, and whose vast knowledge about security and privacy constantly fueled the lunchtime brainstorming sessions. The inspiring random walks and debates also taught me about research and life, and perhaps even more.

At the base camp, many thanks to Kris Cao (kc391), Jimme Jardine (jgj29), Alex Kuhnle (aok25), Jean Maillard (jm864), Yiannos Stathopoulos (yas23) and all other members of GS06, past and present, for the edifying company.

Eternally, I am indebted to my undergrad DoS, Murray Cole, for always being a strong source of support when I was in Edinburgh.

Financially, this work was supported in full and made possible by the Carnegie Trust for the Universities of Scotland, which asked nothing in return for an unrestricted grant. My debt to them is unbounded and I will always be grateful for their openness, kindness, fairness, and generosity. Additionally, I am thankful for the Cambridge Trust for furnishing me with extra funding.

Finally, I would like to thank WM, who knows why.

I dedicate this thesis to PJXKY and my grandparents.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Overview

Structured prediction is characterized by involving an exponentially-sized output space that renders exhaustive search prohibitive. To address this, dynamic programming algorithms are often invoked to construct polynomially-sized representations of the search space. A representative example of this approach is the CKY (Cocke-Kasami-Younger) algorithm applied for syntactic parsing with a variety of formalisms, where the derivation space for a given input is compactly packed into a parse forest, over which parameter estimation and inference can be conducted in polynomial time (Eisner, 1996b; Collins, 1999; Hockenmaier, 2003). For some tasks, dynamic programming also makes exact search tractable, and in such situations, dynamic programming coupled with structured learning models that assume exact search has been proven to be an effective combination, repeatedly demonstrating impressive empirical utility (McCallum et al., 2000; Johnson, 2001; Lafferty et al., 2001; Geman and Johnson, 2002; Miyao and Tsujii, 2002; Taskar et al., 2003; Clark and Curran, 2004b; Taskar et al., 2004; Tsochantaridis et al., 2005; Carreras et al., 2008).

Unfortunately, making a structured prediction problem amenable to dynamic programming often leads to one notable issue: model expressiveness in terms of feature scope is often sacrificed, because features are strictly required to factor locally into substructures of a dynamic program. As a result, methods for incorporating features with arbitrary scope while preserving or abandoning dynamic programming have received much attention, and their benefits are clearly evident (Briscoe and Carroll, 1993; Henderson, 2003; Yamada and Matsumoto, 2003; Nivre and Scholz, 2004; Charniak and Johnson, 2005; Nivre et al., 2006; Carreras, 2007; Huang, 2008; Zhang and Clark, 2011a; Zhang and McDonald, 2012; Socher et al., 2013; Zhu et al., 2013; Chen and Manning, 2014; Sutskever et al., 2014; Durrett and Klein, 2015). Not surprisingly, such methods almost always seek a compromise between exact search and feature scope, as expanding the latter tends to make the former computationally inefficient or even intractable.

Broadly speaking, recent work in parsing that has considered such approaches fall into a few threads. In early attempts, efforts mainly focused on discriminative $k$-best reranking (Collins, 2000; Charniak and Johnson, 2005), in which the key motivation is to sidestep feature locality restrictions enforced by dynamic programming using a reranker that can incorporate arbitrary features. The main drawback of this approach is that both the size and the diversity of the $k$-best lists are usually limited, which directly dictate obtainable improvements (Huang, 2008).

More recently, a line of research has investigated incorporating non-local features into parsing models while striving to maintain exact search (McDonald and Pereira, 2006; Riedel and Clarke, 2006; Carreras, 2007; Martins et al., 2009; Koo and Collins, 2010; Martins et al., 2010). Typically, such solutions use specialized algorithms that decompose non-local features into elementary structures compatible with the model formulations. In doing so, the model inevitably becomes tied to specific feature sets, which often decreases the freedom in feature definitions while increasing computational overhead.

Embracing inexact search, another strand of research chooses to strike a different balance between exactness of search and rich non-local features. One prominent instance of such approaches is forest reranking (Huang, 2008), which scales reranking from finite $k$-best lists to packed parse forests encoding exponentially many alternatives. More specifically, it employs the structured perceptron (Collins, 2002) and approximate chart-based inference to free feature definitions from the underlying dynamic program, allowing arbitrary non-local features to be incorporated, and to be scored incrementally in an efficient bottom-up fashion. Forest reranking primarily stems from cube pruning (Chiang, 2007), which is a generic heuristic search procedure that enables on-demand exploration of the search space, and in conjunction with a backbone inference algorithm (usually based on dynamic programming), it alleviates the potential combinatorial explosion resulting from adding non-local features. The idea of cube pruning originated from constituency parsing (Huang and Chiang, 2005) and has been applied to syntax-based translation with chart-based inference (Chiang, 2007; Huang and Chiang, 2007), to phrase-based translation with beam search (Huang and Chiang, 2007), and to graph-based dependency parsing with higher-order dependency features (Zhang and McDonald, 2012; Zhang et al., 2013; Zhang and McDonald, 2014). In the translation cases, cube pruning greatly improves decoding efficiency when language models are integrated, and it has produced translation systems as accurate as those based on exact decoding (Chang and Collins, 2011) (when using BLEU as the accuracy metric). In the dependency parsing case, it has been shown that the resulting parsers outperform those that allow for exact inference using optimization techniques such as integer linear programming (Riedel and Clarke, 2006; Martins et al., 2009) or dual decomposition (Koo et al., 2010; Martins et al., 2011). In essence, it can be argued that the cube-pruned models all use a hybrid of dynamic programming and more principled inexact search.

At another extreme, exemplified by deterministic shift-reduce parsers without any reliance on search for both parameter estimation and inference (Yamada and Matsumoto, 2003; Nivre and Scholz, 2004), dynamic programming and exact search are completely abandoned. In comparison with chart parsers, the primary advantage of such parsers is their speed, as the number of shift-reduce actions needed is linear in the sentence length. In terms of accuracy, perhaps surprisingly, deterministic shift-reduce models have even matched the overall performance of some global, exhaustive, chart-based models (McDonald and Nivre, 2007; Nivre and McDonald, 2008). However, not only are they more prone to search errors, they also make strong independence assumptions, ignoring sequence-level structural relationships of shift-reduce actions.

Consequently, a natural question that arises is: Can we further improve upon deterministic models while maintaining their speed advantage?

A central issue surrounding this question resolves around improving shift-reduce parser state representations, which are crucial for making accurate shift-reduce parsing decisions. Fortunately, unlike chart parsers that rely on dynamic programming or exact search, incorporating arbitrary features into shift-reduce parsers is trivial, and carefully crafted feature sets often lead to direct parsing accuracy improvements (Cer et al., 2010; Zhang and Nivre, 2011), with only minor penalties on the runtime, even with beam search (Zhang and Clark, 2008; Goldberg et al., 2013). As a more recent development, neural network models have been used to learn such representations, and they are able to dispense with feature engineering either partially through learning distributed representations for higher-order feature conjunctions (Chen and Manning, 2014), or completely through learning parser state representations exhibiting "global" sensitivity to the complete parsing history (Dyer et al., 2015; Watanabe and Sumita, 2015). This accumulated empirical evidence suggests better parser state representations can, to some extent, counter the ill effects of inexact search.

But even with improved representations, search errors inherent with inexact search can still exert a negative impact on learning, especially when inexact search is coupled with a structured learning algorithm that assumes exact search (Collins, 2002; Huang et al., 2012). For the structured perceptron, the early update technique (Collins and Roark, 2004) is widely adopted for this reason. In particular, it replaces the standard perceptron update rule and clearly shows the benefits of accounting for search errors during parameter estimation. Zhang and Clark (2008) adapted this technique for dependency parsing, and Huang et al. (2012) formalized it into the violation-fixing perceptron, providing theoretical justification for early update and formal guarantees for its convergence. Since then, the violation-fixing framework has been further generalized and applied to models utilizing the structured perceptron in a range of tasks including chart-based and shift-reduce parsing (Zhang and McDonald, 2012; Zhang et al., 2013; Xu et al., 2014; Zhang and McDonald, 2014). For neural network models, although the effects of inexact search and search errors have not been formalized, recent work has shown the importance of incorporating structured learning into models that abandon it completely in favour of the representational power provided by neural networks. By either bringing to bear techniques developed for the structured perceptron (Weiss et al., 2015; Watanabe and Sumita, 2015; Lee et al., 2016), or by formulating sequence-level training (Andor et al., 2016; Ranzato et al., 2016; Wiseman and Rush, 2016; Xu et al., 2016; Xu, 2016), such models show the orthogonality of learning richer representations and learning better structured prediction models that take into account the structural properties of the output under inexact search, and they indicate improving both components results in additive gains.

In this thesis, I study shift-reduce CCG parsing, with the unifying theme of structured *learning* with inexact *search*. I advocate inexact search, in particular the kind that is free from dynamic programming (Huang and Sagae, 2010; Kuhlmann et al., 2011), which allows complete freedom in feature representations. I show shift-reduce is a competitive paradigm for CCG parsing and demonstrate its simplicity, accuracy, and speed.

## 1.2 Three Advances for Shift-Reduce CCG Parsing

Combinatory Categorial Grammar (CCG; Steedman, 2000) parsing is challenging due to so-called "spurious" ambiguity that permits a large number of non-standard derivations (Vijay-Shanker and Weir, 1993; Kuhlmann and Satta, 2014). To address this, the *de facto* models resort to chart-based CKY (Hockenmaier, 2003; Clark and Curran, 2007), despite CCG being naturally compatible with shift-reduce parsing (Ades and Steedman, 1982). More recently, Zhang and Clark (2011a) introduced the first shift-reduce model for CCG, which also showed substantial improvements over the long-established state of the art (Clark and Curran, 2007).

The success of the shift-reduce model (Zhang and Clark, 2011a) can be tied to two main contributing factors. First, without any feature locality restrictions, it is able to use a much richer feature set; while intensive feature engineering is inevitable, it has nevertheless delivered an effective and conceptually simpler alternative for both parameter estimation and inference. Second, it couples beam search with global structured learning (Collins, 2002; Collins and Roark, 2004; Zhang and Clark, 2008; Huang et al., 2012), which enables it to model shift-reduce action sequences while making it less prone to search errors than deterministic models. In this thesis, I capitalize on the strengths of the Zhang and Clark (2011a) model and introduce three novel shift-reduce CCG parsing models.

I begin by filling a gap in the literature and developing the first dependency model for shift-reduce CCG parsing (§3). In order to do this, I first introduce a dependency oracle, in which all derivations are hidden. A challenge arises from the potentially exponential number of derivations leading to a gold standard dependency structure, which the oracle needs to keep track of during the shift-reduce process. The solution I propose is an integration of a packed parse *forest*, which efficiently stores all the derivations, with the beam-search decoder *at training time*. The derivations are not explicitly part of the data, since the forest is built from the gold standard dependencies. By adapting the violation-fixing perceptron (Huang et al., 2012), I also show how the dependency oracle can be integrated with the structured perceptron and beam search, which is essential for learning a global structured model.

Departing from the linear perceptron model, next I shift the focus onto shift-reduce parsing with Elman recurrent neural networks (RNNs; Elman, 1990). Recent work has shown that by combining distributed representations and neural network models (Chen and Manning, 2014), accurate and efficient shift-reduce parsing models can be obtained with little feature engineering, largely alleviating the feature sparsity problem of linear models. In practice, the most common objective for optimizing neural network shift-reduce parsing models is maximum likelihood. In the greedy search setting, the log-likelihood of each target action is maximized during training, and the most likely action is committed to at each step of the parsing process during inference (Chen and Manning, 2014; Dyer et al., 2015). In the beam-search setting, Zhou et al. (2015) and Andor et al. (2016) show that sequence-level likelihood and a conditional random field (Lafferty et al., 2001) objective can be used to derive globally normalized models which incorporate beam search at both training and inference time (Zhang and Clark, 2008), giving significant accuracy gains over locally normalized models. However, despite the efficacy of optimizing likelihood, it is often desirable to directly optimize for task-specific metrics, which often leads to higher accuracies for a variety of models and applications (Goodman, 1996; Och, 2003; Smith and Eisner, 2006; Rosti et al., 2010; Auli and Lopez, 2011b; He and Deng, 2012; Auli et al., 2014; Auli and Gao, 2014; Gao et al., 2014).

Here, I present the first neural network parsing model optimized for a task-specific loss based on expected F-measure (§4). The model is globally normalized, and naturally incorporates beam search during training to learn shift-reduce action *sequences* that lead to parses with high expected F-scores. In contrast to Auli and Lopez (2011b), who optimize a log-linear parser for F-measure via softmax-margin (Gimpel and Smith, 2010), I directly optimize an expected F-measure objective, derivable from only a set of shift-reduce action sequences and sentence-level F-scores. More generally, the method can be seen as an approach for training a neural beam-search parsing model (Watanabe and Sumita, 2015; Weiss et al., 2015; Zhou et al., 2015; Andor et al., 2016), combining the benefits of a *global* model and *task-specific* optimization.

Finally, I describe a neural architecture for learning parser state representations for shift-reduce CCG parsing based on long short-term memories (LSTMs; Hochreiter and Schmidhuber, 1997) (§5). The model is inspired by Dyer et al. (2015), in which I explicitly *linearize* the complete history of parser states in an *incremental* fashion by requiring no feature engineering (Zhang and Clark, 2011a; Xu et al., 2014), and no atomic feature sets (Chen and Manning, 2014). However, a key difference is that this linearization is achieved without relying on any additional control operations or compositional tree structures (Socher et al., 2010; Socher et al., 2011; Socher et al., 2013), both of which are vital in the architecture of Dyer et al. (2015). Crucially, unlike the sequence-to-sequence transduction model of Vinyals et al. (2015), which drops the parser completely, I construct parser state representations, which are also sensitive to all aspects of the parsing history and the complete input, by following the shift-reduce syntactic derivation process. To do structured learning, I show expected F-measure training, which is agnostic to the underlying neural network, can be applied in this setting to obtain globally normalized greedy and beam-search LSTM parsers, which give state-of-the-art results for shift-reduce CCG parsing (§5.4.2).

In all the shift-reduce models I introduce, structured learning and inexact search are integrated such that inexact search, specifically, beam search, is used to define the criteria that dictate parameter updates, in addition to being used for approximate inference. In the dependency model, beam search locates model violations for online structured perceptron learning, and updates are performed along the corresponding shift-reduce action prefixes contained in the beam. In the neural network models, beam search provides a set of shift-reduce action sequences from which the learning objective and gradients are derived. In both cases, learning is guided by search and search is informed by learning.

For CCG, the first competitive dependency model was developed by Clark and Curran (2007), in which chart-based exact search together with a log-linear framework are used to calculate feature expectations over a parked forest of all derivations, including those "correct" derivations leading to a given CCG dependency structure. Once such derivations are identified, model estimation techniques used for the derivation-only normal-form model are applied with minimal modifications, and there is no need to reconcile learning with search (§3.2.1). In contrast, this reconciliation problem cannot be avoided when developing the shift-reduce dependency model, since latent "correct" derivations need to be dealt with without potentially violating the convergence properties of the structured perceptron with beam-search inference (Collins and Roark, 2004; Huang et al., 2012). Fortunately, the development of the violation-fixing perceptron with convergence guarantees (Huang et al., 2012) provides an almost tailor-made technique that can be adapted, and it is precisely the dependency oracle, the violation-fixing perceptron, and their seamless integration with beam search that yield the dependency model.

In many recent neural network-based models, the use of inexact search is evidently typical, and a standard framework adopted is based on locally normalized maximum likelihood estimation and greedy or beam-search inference (Chen and Manning, 2014; Vinyals et al., 2015; Dyer et al., 2015; Lewis and Steedman, 2014b; Xu et al., 2015; Lewis et al., 2016). In such models, search does not interact with or influence parameter estimation, and it is only introduced during inference to maintain tractability. As such, they are susceptible to the label bias problem (Bottou, 1991; Lafferty et al., 2001; Andor et al., 2016), as well as to exposure bias, loss-evaluation mismatch (Ranzato et al., 2016), and search errors inherent with inexact search. To tackle this, a number of works (Ranzato et al., 2016; Watanabe and Sumita, 2015; Weiss et al., 2015; Zhou et al., 2015; Andor et al., 2016), including the global RNN and LSTM shift-reduce models in this thesis, have emerged, and they all represent a move towards utilizing neural network representations without sacrificing the proven strengths of structured learning. In addition to improving accuracy, such models also largely retain the efficiency of inexact search. A prime example of this kind is the expected F-measure trained greedy LSTM shift-reduce parser (§5.4.2), which displays the accuracy of the global model while maintaining fully greedy inference.

## 1.3 Contributions

The primary contributions of this thesis are the three shift-reduce parsing models described above, in summary:

- I develop a dependency oracle for shift-reduce CCG parsing.
- I develop a learning algorithm that integrates the dependency oracle with the structured perceptron and beam search by generalizing early update under the violation-fixing perceptron framework.
- I develop expected F-measure training for shift-reduce CCG parsing.
- I develop a shift-reduce CCG parsing model based on Elman RNNs.
- I develop a shift-reduce CCG parsing model based on LSTMs.
- I apply the expected F-measure training framework to both the RNN- and LSTM-based parsing models.

As the secondary contribution, I introduce three recurrent neural network models for CCG supertagging, which form an integral part of and are indispensable for the respective parsing models. Recent work on supertagging using a feed-forward neural network achieved significant improvements for CCG supertagging and parsing (Lewis and Steedman, 2014b); however, their architecture is limited to considering local contexts and does not naturally model sequences of arbitrary length. I show how directly capturing sequence information using RNNs (§4.2 and §4.5) and LSTMs (§5.3) leads to further accuracy improvements for both supertagging and parsing. I also show the improvements in supertagging accuracy translates into state-of-the-art parsing accuracies for the C&C parsing models on three different domains (§4.6).

Last, but not least, the implementation of all the parsers presents some considerable challenges given the various CCG- and parsing model-related elements involved. To enable reproducibility, I describe the models in detail and release the code. In tandem with the experimental findings, this thesis serves as a reference and benchmark for future work on shift-reduce CCG parsing.

# Chapter 2

# Background

## 2.1 Combinatory Categorial Grammar

A *lexicon*, together with a set of CCG combinatory *rules*, formally constitute a CCG. The former defines a mapping from words to sets of lexical categories representing syntactic types, and the latter gives schemata which dictate whether two categories can be combined. Given the lexicon and the rules, the syntactic types of complete constituents can be obtained by recursive combination of categories using the rules.

More generally, both lexical and non-lexical CCG categories can be either *atomic* or *complex*: atomic categories are categories without any slashes (e.g., $NP$ and $PP$), and complex categories are constructed recursively from atomic ones using forward ($/$) and backward slashes ($\backslash$) as two binary operators (e.g., $(S\backslash NP)/NP$). As such, all categories can be represented as follows (Vijay-Shanker and Weir, 1993; Kuhlmann and Satta, 2014):

$$X \coloneqq \alpha|_1 Z_1|_2 Z_2 \ldots |_m Z_m, \tag{2.1}$$

where $m \geq 0$ is referred to as the *arity*, $\alpha$ is an atomic category, $|_i \in \{\backslash, /\}$, and $Z_i$ are metavariables for categories.

CCG rules are either *binary* or *unary*. Binary rules have the following two schematic forms, where each is a generalized version of *functional composition* (Vijay-

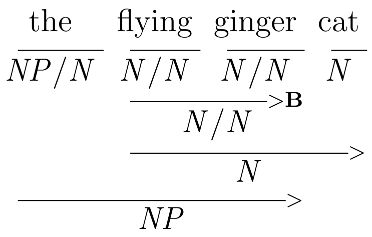


Figure 2-1: Example CCG derivation. By convention, CCG derivations are drawn upside down.

Shanker and Weir, 1993; Kuhlmann and Satta, 2014):

$$\begin{aligned} &X/Y \quad Y|_1Z_1 \ldots |_dZ_d \Rightarrow X|_1Z_1 \ldots |_dZ_d, \\ &Y|_1Z_1 \ldots |_dZ_d \quad X\backslash Y \Rightarrow X|_1Z_1 \ldots |_dZ_d. \end{aligned} \tag{2.2}$$

The first schematic form above instantiates into a forward application ($>$) rule for $d = 0$ and forward composition rule ($> \mathbf{B}$) for $d > 0$. Similarly, the second schematic form, which is symmetric to the first, instantiates into backward application ($<$) and composition ($< \mathbf{B}$) rules. In both schemata, $d$ is referred to as the rule *degree*; the left category $X/Y$ in the forward case is the called the *primary* category, while the one on the right is *secondary*; and the primary category in the backward case is the category $X\backslash Y$ on the right.

Fig.2-1 shows an example CCG derivation. All the *rule instances* in this derivation are instantiated from forward rules, which in turn take the schematic form of generalized forward composition; for example, $N/N \ N \rightarrow N$ is an instance of forward application (with $d = 0$), and $N/N \ N/N \rightarrow N/N$ is an instance of forward composition (with $d = 1$).

Unary rules are referred to as type-raising rules, and they have the following two schemata (Steedman, 2000):

$$\begin{aligned} X \quad &\Rightarrow_{\mathbf{T}} \quad T/(T\backslash X) \quad (> \mathbf{T}), \\ X \quad &\Rightarrow_{\mathbf{T}} \quad T\backslash(T/X) \quad (< \mathbf{T}). \end{aligned}$$

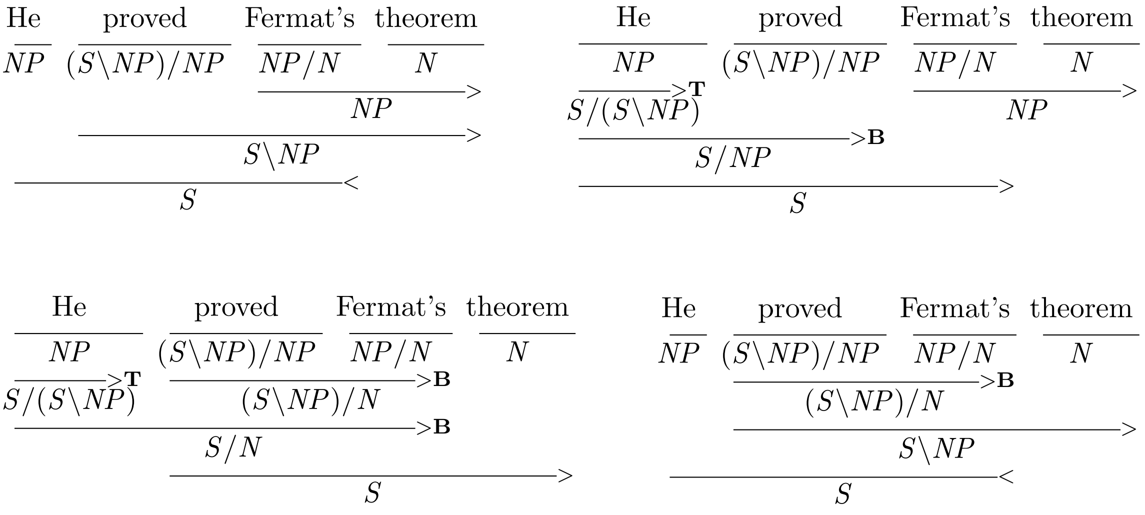


Figure 2-2: A normal-form derivation, with three possible spurious alternatives.

Two examples of forward type-raising ($> \mathbf{T}$) are shown in Fig.2-2.

### 2.1.1 Normal-Form CCG Derivations

CCG exhibits so-called "spurious" ambiguity, which mainly arises from composition and type-raising. To alleviate this, the Eisner constraint (Eisner, 1996a) can be applied to obtain normal-form CCG derivations.

Informally, this constraint ensures that composition and type-raising are not used unless necessary. This is achieved by preventing the result of a forward composition from serving as the primary (left) category in another forward application or composition; similarly, any constituent which is the result of a backward composition is kept from serving as the primary (right) category in another backward application or composition. As an example, Fig.2-2 shows a normal-form derivation, along with three possible spurious alternatives.[1]

In practice, when type-raising is available in a CCG grammar (which is almost always the case for modern data-driven CCG parsing), Eisner's constraint is not guaranteed to produce normal-form derivations (not complete); moreover, when the degree (Eq.2.2) of composition is bounded, it may not preserve all interpretations

[1] In this example, composition and type-raising are not present in the normal-form derivation; however, this is not the general situation, because composition and type-raising are often necessary to analyze a sentence, and Eisner's constraint is designed to not completely eliminate such analyses.

(not safe) (Hockenmaier and Bisk, 2010). Nevertheless, CCG parsers often implement it, because efficiency improvement is usually considerable even when spurious ambiguity is partially alleviated. This is the choice in the first wide-coverage CCG parser by Hockenmaier (2003), and it was later also implemented in the normal-form and hybrid dependency models of Clark and Curran (2007), as part of the C&C toolkit (Curran et al., 2007).

### 2.1.2 CCG Recognition Complexity

CCG is binary-branching and directly compatible with the CKY algorithm. However, a naive application of CKY results in worst case exponential runtime and space with respect to the input length. To see this, recall that standard CKY for an unlexicalized binarized grammar has a worst case runtime of $\mathcal{O}(|G|n^3)$ (often abbreviated as $\mathcal{O}(n^3)$, because $|G|$ is the grammar size usually independent of $n$). In CCG, however, no restrictions are enforced on the arity of its categories (Eq. 2.1), and the availability of generalized composition allows the arity of primary categories (Eq. 2.2) to grow linearly in $n$, resulting in an exponentially-sized grammar with an exponential number of categories in the worst case. This exponential growth is also what gives CCG its mild context-sensitivity (Vijay-Shanker and Weir, 1993; Kuhlmann et al., 2010), which contrasts with the context-free Categorial Grammar (Bar-Hillel, 1953).

To alleviate this, Vijay-Shanker and Weir (1993) proposed a polynomial-time recognition algorithm for CCG, which runs in $\mathcal{O}(n^6)$ time disregarding the grammar constant. To date, however, this algorithm has been proven difficult to understand and implement, hindering its adoption in practical parsers. As an alternative, Kuhlmann and Satta (2014) introduced an algorithm that is in principle more accessible and much easier to implement. But like the Vijay-Shanker and Weir (1993) algorithm, the new algorithm only deals with a specific variant of CCG, and is incompatible with some CCG rules (most notably type-raising) implemented in current CCG parsers, which are necessary for wide-coverage parsing.

More recently, as a somewhat surprising result, it has been proved that any recognition algorithm for CCG in the formalism of Vijay-Shanker and Weir (1994) would indeed have an exponential complexity if the grammar size is taken into account (Kuhlmann et al., 2017). Although the ramifications of this result on modern CCGBank parsing is still unknown, it is likely, however, that the pursuit of more complex recognition algorithms would be impeded, at least to some extent.

In practice, the pragmatic solution widely adopted to manage the exponential factor in CCG parsers is to use a hard arity restriction tuned on the parser's training data (Curran et al., 2007); although crude, this method has been empirically shown to be quite effective with minimal impact on parsing accuracy and coverage.[2]

As a less desirable solution, which I discuss below, a context-free *cover* grammar derived from the parser's training data can be used to avoid this exponential growth.

### 2.1.3 CCGBank

CCGBank (Hockenmaier, 2003; Hockenmaier and Steedman, 2007) is a treebank of normal-form CCG derivations, created from the Penn Treebank (Marcus et al., 1993) with a semi-automatic conversion process, incorporating CCG specific analyses that are not originally present in the Penn Treebank derivations. Given CCGBank, there are two approaches to extract a grammar for data-driven parsing. The first is to extract all binary and unary CCG rule instances from CCGBank derivations (Fowler and Penn, 2010; Zhang and Clark, 2011a). The main drawback of this approach is that it results in a strictly context-free cover grammar, which only allows rule instances seen in the treebank to be applied and can potentially compromise the coverage by suppressing many derivations. On the other hand, a convenient, but rather inelegant way of using this grammar is to apply Penn Treebank constituency parsing models directly to CCG (Fowler and Penn, 2010). However, this deviates away from CCG's original design, in which a key element is a small set of language-independent, universal combinatory rules (Steedman, 2000).

Adhering to this design, Hockenmaier (2003) and Clark and Curran (2007) extract only the lexicon from CCGBank and define CCG rules without explicitly enumerating any rule instances (Hockenmaier, 2003). Here I follow this approach by using exactly

[2] For example in the C&C parser, the maximum arity is set to 9.

the same CCG rules as in Clark and Curran (2007, Appendix A), including all the non-standard rules that are necessary for handling CCGBank derivations.

**The Sentence Category.** Another noteworthy property of CCGBank relevant to the current work is that sentence categories $S$ in derivations are furnished with features designating sentence types. For example, $S[dcl]$ is for declarative, $S[q]$ for yes-no questions, and $S[inv]$ for elliptical inversion. In particular, sentence categories in verb phrases may also carry features; for example, $S[b]$ is for bare infinitives, subjunctives and imperatives, as in $S[b]\backslash NP$, and $S[to]$ is for to-infinitive, as in $S[to]\backslash NP$. Hockenmaier (2003, §3.4) contains a complete list for both sentential and verb phrase features for the sentence category.

**The Standard Splits.** For all experiments in this thesis, the standard splits of CCGBank are used: Sections 2-21 for training (39,604 sentences), Section 00 for development (1,913 sentences), and Section 23 (2,407 sentences) for testing.

### 2.1.4 CCG Dependency Structures

CCG has a completely transparent interface between surface syntax and semantics (Steedman, 2000), and each syntactic constituent in a CCG derivation can be associated with a semantically interpretable expression. Specifically, the slash operators / and \ in complex CCG categories which dictate how categories combine can be interpreted as $\lambda$ operators in lambda calculus. For example, *join* as a transitive verb has a syntactic category $(S[dcl]\backslash NP)/NP$, and can be paired with a semantic interpretation based on the lambda expression $\lambda x \lambda y.join(y, x)$. More importantly, the semantic arguments in the lambda expression of a category are consumed in the same order as the syntactic arguments of the same category. That is, there is a one-to-one correspondence between the slash and $\lambda$ operators. For example, Fig.2-3 shows how the semantic interpretation *joined(he, NASA)* is built "incrementally" (Ambati et al., 2015) from the lambda expressions associated with the individual words in the sentence *He joined NASA*, as the derivation is built.

In the theory of CCG, this close coupling of syntactic and semantic interpretations is primarily linked to its *Principle of Combinatory Type Transparency* (Steedman,

$$\begin{array}{cccc}
\text{He} & \text{joined} & \text{NASA} & \\
\hline
NP : he & (S[dcl]\backslash NP)/NP : \lambda x \lambda y.joined(y, x) & NP : NASA & \\
\cline{2-3}
 & \multicolumn{2}{c}{(S[dcl]\backslash NP) : \lambda y.joined(y, NASA)} & > \\
\cline{1-3}
\multicolumn{3}{c}{S[dcl] : joined(he, NASA)} & < \\
\end{array}$$

Figure 2-3: Example semantic representations expressed in lambda expressions built as part of a syntactic derivation.

2000, p. 37), which states that:

> All syntactic combinatory rules are type-transparent versions of one of a small number of simple semantic operations over functions.

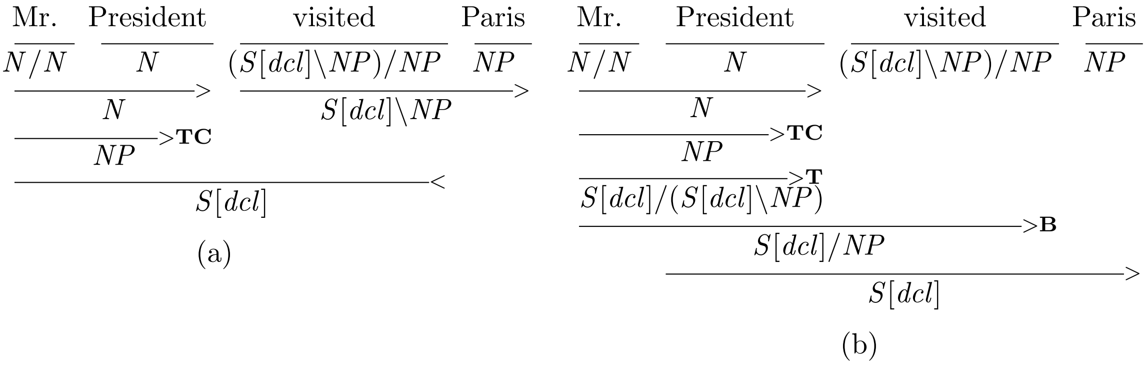
As a result, the semantic interpretation (as represented by a lambda expression) associated with the category resulting from a combinatory rule is uniquely determined by the slashes of the primary and secondary categories (Eq. 2.2). Steedman (1996) further argues that this principle is the most important single property of CCG combinatory rules. Indeed, it is also closely related to how CCG predicate-argument dependencies—which loosely represent the semantic interpretations—are obtained during the syntactic derivation process (Hockenmaier, 2003).

I define CCG dependencies following Clark and Curran (2007), where each complex category in the lexicon defines one or more predicate-argument relations. For example, the transitive verb category $(S[dcl]\backslash NP_1)/NP_2$ defines two relations: one for the subject $NP_1$ and one for the object $NP_2$. During a derivation, the predicate-argument relations are realized as predicate-argument dependencies as categories combine, and all the dependencies resulting from a derivation form a set, which is referred to as a CCG *dependency structure*.

The goal of parsing with a CCG is to recover the dependency structures, and any parsing model for CCG is evaluated on their dependency recovery accuracy.

**Definition 1.** A CCG **predicate-argument dependency** is a 4-tuple: $\langle h_f, f, s, h_a \rangle$ where $h_f$ is the lexical item of the lexical category expressing the dependency, $f$ is the lexical category, $s$ is the argument slot, and $h_a$ is the head word of the argument.

**Definition 2.** A CCG **dependency structure** is a *set* of CCG predicate-argument



$$\langle \textit{Mr.}, N/N_{\mathit{1}}, 1, \textit{President}\rangle$$
$$\langle \textit{visited}, (S[dcl]\backslash NP_{\mathit{1}})/NP_{\mathit{2}}, 2, \textit{Paris}\rangle$$
$$\langle \textit{visited}, (S[dcl]\backslash NP_{\mathit{1}})/NP_{\mathit{2}}, 1, \textit{President}\rangle$$

Figure 2-4: Two derivations leading to the same dependency structure. **TC** denotes type-changing (which is only defined in CCGBank but not in the original formalism).

dependencies, in which all the lexical items are indexed by sentence position.

Fig. 2-4 shows an example demonstrating a CCG dependency structure in relation to spurious ambiguity. In both derivations, the first two lexical categories are combined using forward application (>) and the dependency

$$\langle \textit{Mr.}, N/N_{\mathit{1}}, 1, \textit{President}\rangle$$

is realized, with the category expressing the dependency being $N/N_{\mathit{1}}$, which has one argument slot. The head word of the argument is *President*, which becomes the head of the resulting category $N$.[3]

In Fig. 2-4a, a normal-form derivation is shown. In this example, the dependency $\langle \textit{visited}, (S[dcl]\backslash NP_{\mathit{1}})/NP_{\mathit{2}}, 2, \textit{Paris}\rangle$ is realized by combining the transitive verb category with the object *NP* using forward application, which fills the object $NP_{\mathit{2}}$ slot in the transtive verb category. The $NP_{\mathit{1}}$ slot is then filled when the dependency $\langle \textit{visited}, (S[dcl]\backslash NP_{\mathit{1}})/NP_{\mathit{2}}, 1, \textit{President}\rangle$ is realized at the root node $S[dcl]$ through backward application (<).

Fig. 2-4b shows a non-normal-form derivation, which uses type-raising (**T**) and

[3] Head identification which uses additional category annotations is discussed in detail in Hockenmaier (2003) and Clark and Curran (2007).

composition ($\mathbf{B}$) (that are not required to derive the correct dependency structure). In this alternative derivation, the dependency $\langle \textit{visited}, (S[dcl]\backslash NP_1)/NP_2, 1, \textit{President}\rangle$ is realized using forward composition ($> \mathbf{B}$), and $\langle \textit{visited}, (S[dcl]\backslash NP_1)/NP_2, 2, \textit{Paris}\rangle$ is realized when the $S[dcl]$ root is produced.

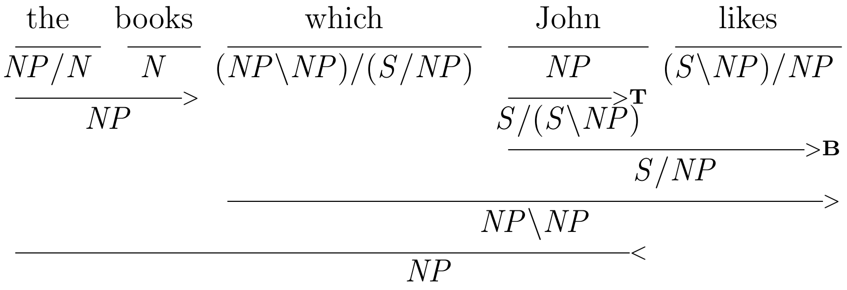
As can be seen in both examples above, all the dependencies realized are local, that is, they conjoin only two neighbouring words. In addition, using a unification mechanism (Hockenmaier, 2003; Clark and Curran, 2007), CCG is also able to handle long-range dependencies naturally; I show an example in Fig. 2-5.

In the bottom left column, the rules applied to obtain each constituent are depicted. In the right column, I depict the set of categories that are expressing dependencies, their filled (shaded) and unfilled argument slots, and each superscripted index indicates the corresponding dependency that is realized. Categories with unfilled argument slots are also passed over during the whole derivation process.

The first dependency $\langle$*the*, $NP/N_1$, 1, *books*,$\rangle$ is realized through the rule application on the first row; it is a local dependency which fills the only argument slot in $NP/N_1$. Similarly, the second dependency $\langle$*likes*, $(S\backslash NP_1)/NP_2$, 1, *John*$\rangle$ is also local, and it is realized when the last two words are combined; however, this leaves the second argument slot in the category for *likes* unfilled, which is passed over. On the third row, the first non-local dependency $\langle$*which*, $(NP/NP_1)/(S/NP)_2$, 2, *likes*$\rangle$ is realized. Eventually at the root, the second argument slot in the category of *likes* is filled, realizing the non-local dependency $\langle$*likes*, $(S\backslash NP_1)/NP_2$, 2, *books*$\rangle$. Also at the root, the first argument slot in the category of *which* is filled, realizing the local dependency $\langle$*which*, $(NP/NP_1)/(S/NP)_2$, 1, *books*$\rangle$.

### 2.1.5 Comparison with Dependency Grammars

In a dependency grammar, such as the one defined in McDonald (2006), a set of constraints are imposed such that the resulting dependency graph forms a valid tree. In CCG, no external constraints are placed on the well-formedness of the dependency graphs, and this gives great flexibility in how CCG dependencies are realized, which is only dictated by the predicate-argument relations defined by the complex categories.



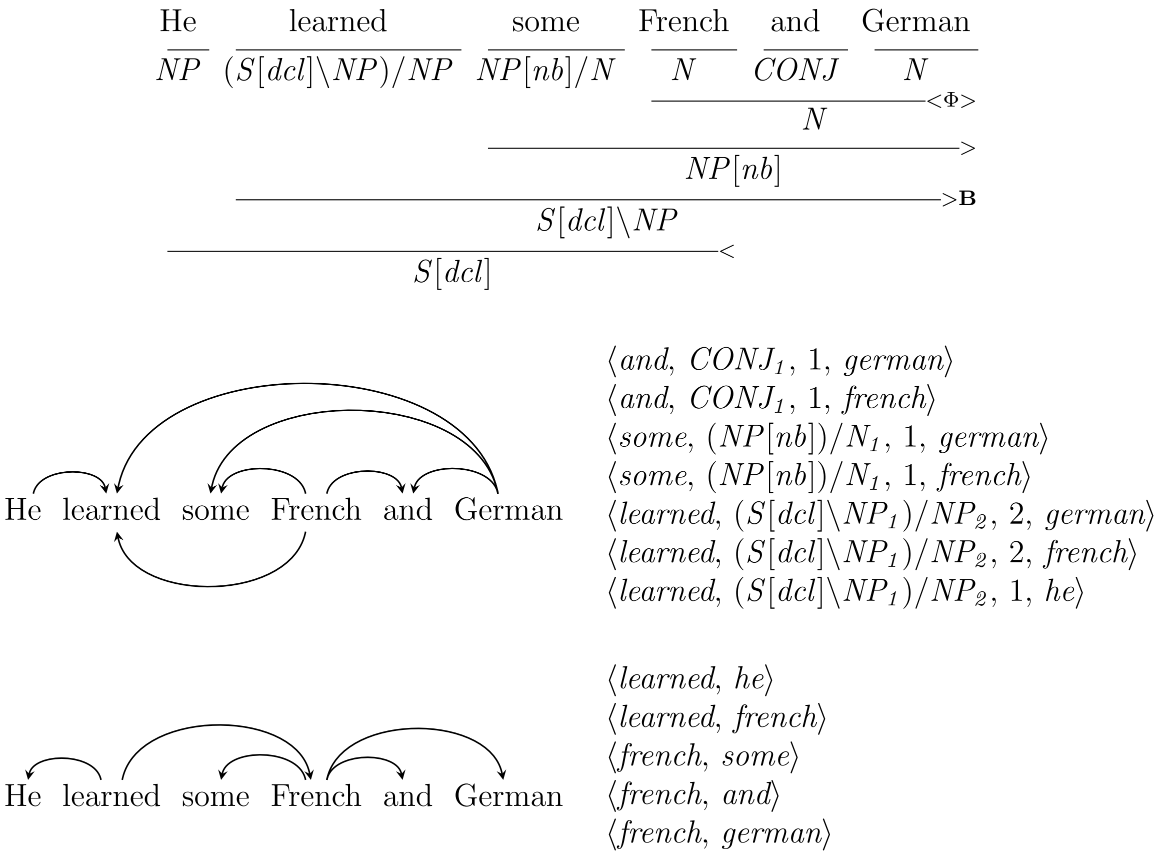
$$\begin{gathered}
\langle \textit{the},\ NP/N_{\mathit{1}},\ 1,\ \textit{books}\rangle^{\text{①}}\\
\langle \textit{likes},\ (S\backslash NP_{\mathit{1}})/NP_{\mathit{2}},\ 1,\ \textit{John}\rangle^{\text{②}}\\
\langle \textit{which},\ (NP/NP_{\mathit{1}})/(S/NP)_{\mathit{2}},\ 2,\ \textit{likes}\rangle^{\text{③}}\\
\langle \textit{likes},\ (S\backslash NP_{\mathit{1}})/NP_{\mathit{2}},\ 2,\ \textit{books}\rangle^{\text{④}}\\
\langle \textit{which},\ (NP/NP_{\mathit{1}})/(S/NP)_{\mathit{2}},\ 1,\ \textit{books}\rangle^{\text{⑤}}
\end{gathered}$$

$$\begin{array}{rcl|l}
NP/N_{\mathit{1}} \quad N & \rightarrow & NP & {}^{\text{①}}NP/N_1{}^{\textit{the}}\\[1em]
S/(S\backslash NP) \quad (S\backslash NP_{\mathit{1}})/NP_{\mathit{2}} & \rightarrow & S/NP & {}^{\text{②}}(S\backslash NP_1)/NP_{\mathit{2}}{}^{\textit{likes}}\\[1em]
(NP\backslash NP_{\mathit{1}})/(S/NP)_{\mathit{2}} \quad S/NP & \rightarrow & NP\backslash NP & (S\backslash NP_1)/NP_{\mathit{2}}{}^{\textit{likes}}\\
 & & & {}^{\text{③}}(NP\backslash NP_{\mathit{1}})/(S/NP)_2{}^{\textit{which}}\\[1em]
NP \quad NP\backslash NP & \rightarrow & NP & {}^{\text{④}}(S\backslash NP_1)/NP_2{}^{\textit{likes}}\\
 & & & {}^{\text{⑤}}(NP\backslash NP_1)/(S/NP)_2{}^{\textit{which}}
\end{array}$$

Figure 2-5: Long-range dependency realization example. Top is the derivation; middle is the dependency structure; bottom is unification in action. The right column at the bottom shows filled (shaded) and unfilled dependencies in the lexical categories, where categories with unfilled dependencies are passed over through the derivation as part of the unification process.

For instance, in McDonald (2006), it is strictly required that "Each word has exactly one incoming edge in the graph (except the root, which has no incoming edge)." and "If there are $n$ words in the sentence (including root), then the graph has exactly $n-1$ edges." In a CCG dependency structure, both of these two constraints can be violated.

For example, in the CCG dependency tree shown in Fig. 2-6, there are multiple words with more than one incoming edges, and the total number of dependencies



⟨*and*, *CONJ*$_1$, 1, *german*⟩
⟨*and*, *CONJ*$_1$, 1, *french*⟩
⟨*some*, $(NP[nb])/N_1$, 1, *german*⟩
⟨*some*, $(NP[nb])/N_1$, 1, *french*⟩
⟨*learned*, $(S[dcl]\backslash NP_1)/NP_2$, 2, *german*⟩
⟨*learned*, $(S[dcl]\backslash NP_1)/NP_2$, 2, *french*⟩
⟨*learned*, $(S[dcl]\backslash NP_1)/NP_2$, 1, *he*⟩

⟨*learned*, *he*⟩
⟨*learned*, *french*⟩
⟨*french*, *some*⟩
⟨*french*, *and*⟩
⟨*french*, *german*⟩

Figure 2-6: Comparison of a CCG and a projective dependency tree.

is greater than the number of words in the sentence.[4] Additionally, there is a non-projective edge from *French* to *learned*, which would be disallowed in a projective dependency tree, as in Fig. 2-6.[5,6]

In comparison with typical constituency and dependency grammars, the flexibility in dependency realizations and the ability to recover many types of linguistically sophisticated dependencies are two distinguishing features of CCG (Rimell et al., 2009; Nivre et al., 2010).

[4] In CCG, there is no convention for drawing the dependency arcs, here the arcs always leave from the arguments.

[5] The CCG dependencies are produced with the locally normalized LSTM parser described in §5. The projective dependency tree is obtained with the Stanford dependency parser demo: `http://nlp.stanford.edu:8080/parser/index.jsp`, and Google Cloud API Demo: `https://cloud.google.com/natural-language`.

[6] *NP*[*nb*] is treated to be equal to *NP* in this derivation by the parser. The handling of this is implementation specific.

## 2.2 CCG Supertagging

CCG is strongly lexicalized by definition. A CCG grammar extracted from CCGBank contains over 1,000 lexical types (Clark and Curran, 2007), compared to about 50 POS tags for typical CFG parsers. This makes accurate disambiguation of lexical types much more challenging. However, the assignment of lexical categories can still be solved reasonably well by treating it as a sequence tagging problem, often referred to as supertagging (Bangalore and Joshi, 1999). Clark and Curran (2004a) show that high tagging accuracy can be achieved by leaving some ambiguity to the parser to resolve, but with enough of a reduction in the number of tags assigned to each word so that parsing efficiency is greatly increased.

In addition to improving parsing efficiency, supertagging also has a large impact on parsing accuracy (Curran et al., 2006; Kummerfeld et al., 2010), since the derivation space of the parser is determined by the supertagger, at both training and test time. Clark and Curran (2007) enhanced supertagging using a so-called adaptive strategy, such that additional categories are supplied to the parser only if a spanning analysis cannot be found. This strategy is used in the *de facto* C&C parser (Curran et al., 2007), and the two-stage CCG parsing pipeline (supertagging and parsing) continues to be the choice for most recent CCG parsers (Zhang and Clark, 2011a; Auli and Lopez, 2011a), including all the CCG parsing models in this thesis.

**Supertagger Pruning.** For all current CCG parsers, including both chart and shift-reduce parsers, the supertagger front end uses a probability cutoff parameter $\beta$ to determine the average number of supertags assigned to each word (ambiguity), pruning categories whose probabilities are not within $\beta$ times the probability of the 1-best category.

For all shift-reduce parsers in this thesis, the multi-tagging output obtained with a *fixed* $\beta$ is used for training and testing, and adpative supertagging is not applied. In general, smaller $\beta$ values can be used by a shift-reduce parser than by a dynamic programming-based chart parser, since $\beta$ values which are too small may explode the dynamic program of the latter, while search can still remain tractable with shift-reduce parsing and inexact search.

## 2.3 Shift-Reduce CCG Parsing

Not only is supertagging challenging for CCG, the lexicalized nature of CCG and the number of possible lexical categories makes lexical disambiguation more challenging for the parser. This lexical disambiguation problem remains even when a context-free cover grammar extracted from CCGBank is used. Moreover, there are more than 1,500 non-terminals in a standard CCG grammar (Hockenmaier, 2003; Clark and Curran, 2007), which is an order of magnitude more than those of a typical CFG parser. Indeed, as noted by by Auli and Lopez (2011a), the search problem for CCG parsing is equivalent to finding an optimal derivation in the weighted intersection of a regular language (generated by the supertagger) and a mildly context-sensitive language (generated by the parser), which can quickly become expensive.

The shift-reduce algorithm (Aho and Ullman, 1972; Yamada and Matsumoto, 2003; Nivre, 2004; Nivre and Scholz, 2004) applied to CCG presents a more elegant solution to this lexical disambiguation problem by allowing the parser to conduct lexical assignment "incrementally" (Ambati et al., 2015) as a complete parse is being built by the decoder (Zhang and Clark, 2011a). This is not possible with a chart parser based on dynamic programming, in which complete derivations must be built first. Therefore, a shift-reduce parser is able to consider a larger set of categories per word for a given input, achieving higher lexical assignment accuracy than the C&C parser (Clark and Curran, 2007), even with the same supertagging model (Zhang and Clark, 2011a; Xu et al., 2014).

In all the shift-reduce CCG parsers below, I follow this strategy and adopt the Zhang and Clark (2011a) style shift-reduce transition system, which assumes a set of lexical categories has been assigned to each word using a supertagger. Parsing then proceeds by applying a sequence of actions to transform the input maintained on a *queue*, into partially constructed derivations, kept on a *stack*, until the queue and available actions on the stack are both exhausted.

Figure 2-7 shows a shift-reduce parsing example for the sentence *Mr. President*

| step | stack $(\delta_n, ..., \delta_1, \delta_0)$ | queue $(\beta_0, \beta_1, ..., \beta_m)$ | action |
|---|---:|---|---|
| 0 | | Mr. President visited Paris | |
| 1 | $N/N$ | President visited Paris | SHIFT |
| 2 | $N/N\ N$ | visited Paris | SHIFT |
| 3 | $N$ | visited Paris | REDUCE |
| 4 | $NP$ | visited Paris | UNARY |
| 5 | $NP\ (S[dcl]\backslash NP)/NP$ | Paris | SHIFT |
| 6 | $NP\ (S[dcl]\backslash NP)/NP\ N$ | | SHIFT |
| 7 | $NP\ (S[dcl]\backslash NP)/NP\ NP$ | | UNARY |
| 8 | $NP\ S[dcl]\backslash NP$ | | REDUCE |
| 9 | $S[dcl]$ | | REDUCE |

Figure 2-7: Deterministic example of shift-reduce CCG parsing. $\beta$ is a queue of remaining input, consisting of words and the gold standard lexical category for each word (with $\beta_0$ being the front word), and $\delta$ is the stack that holds subtrees (with $\delta_0$ at the top); lexical categories omitted on queue.

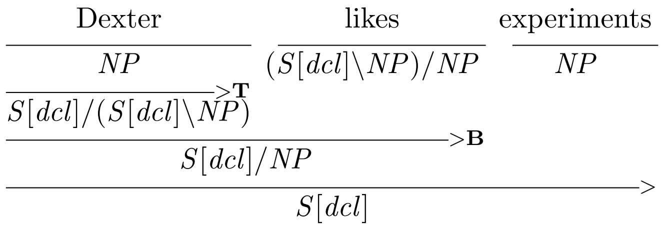


Figure 2-8: A CCG derivation, in which each point corresponds to the result of a shift-reduce action. In this example, composition (**B**) and application (>) are REDUCE actions, and type-raising (**T**) is a UNARY action.

*visited Paris*, giving a single, deterministic sequence of shift-reduce actions which produces a correct derivation (i.e., one producing the correct set of dependencies). Starting with the initial parser state (row 0), which has an empty stack and a full queue, a total of nine actions are applied to produce the complete derivation. Another deterministic example is shown in Fig. 2-8, where the sequence of shift-reduce actions that builds the derivation is: SHIFT $\Rightarrow NP$, UNARY $\Rightarrow S[dcl]/(S[dcl]\backslash NP)$, SHIFT $\Rightarrow (S[dcl]\backslash NP)/NP$, REDUCE $\Rightarrow S[dcl]/NP$, SHIFT $\Rightarrow NP$ and REDUCE $\Rightarrow S[dcl]$, where $\Rightarrow$ is used to indicate the CCG category produced by an action.

input : $w_0 \ldots w_{n-1}$
axiom : $0 : (0, \epsilon, \beta, \phi)$
goal : $2n - 1 + \mu : (n, \delta, \epsilon, \Delta)$

$$\frac{\omega : (j, \delta, x_{w_j}|\beta, \Delta)}{\omega + 1 : (j+1, \delta|x_{w_j}, \beta, \Delta)} \qquad (\textsc{shift}; 0 \leq j < n)$$

$$\frac{\omega : (j, \delta|s_1|s_0, \beta, \Delta)}{\omega + 1 : (j, \delta|x, \beta, \Delta \cup \langle x \rangle))} \qquad (\textsc{reduce}; s_1 s_0 \to x)$$

$$\frac{\omega : (j, \delta|s_0, \beta, \Delta)}{\omega + 1 : (j, \delta|x, \beta, \Delta)} \qquad (\textsc{unary}; s_0 \to x)$$

Figure 2-9: The shift-reduce deduction system.

### 2.3.1 The Transition and Deduction Systems

More formally, I denote parser states as $(j, \delta, \beta, \Delta)$, where $\delta$ is the stack (with top element $\delta|s_0$), $\beta$ is the queue (with top element $x_{w_j}|\beta$), $j$ is the positional index of the word at the front of the queue, and $\Delta$ is the set of CCG dependencies realized for the input consumed so far.[7] I also assume a set of lexical categories has been assigned to each word using a supertagger. The transition system is then specified using three action types:

- SHIFT removes one of the lexical categories $x_{w_j}$ of the front word $w_j$ in the queue, pushes it onto the stack, and removes $w_j$ from the queue.
- REDUCE combines the top two subtrees $s_0$ and $s_1$ on the stack using a CCG rule ($s_1 s_0 \to x$) and replaces them with a subtree rooted in $x$. It also appends the set of newly realized dependencies on $x$, denoted as $\langle x \rangle$, to $\Delta$.
- UNARY applies either a type-raising or type-changing rule ($s_0 \to x$) to the stack-top element and replaces it with a unary subtree rooted in $x$.

[7] Standard notations from dependency parsing (Nivre, 2008) are partly adopted.

The deduction system (Fig. 2-9) follows from the transition system.[8] Each parser state is associated with a step indicator $\omega$, which denotes the number of actions used to build it. Given a sentence of length $n$, a full derivation requires $2n-1+\mu$ steps to terminate, where $\mu$ is the total number of UNARY actions applied. In Zhang and Clark (2011a), a *finish* action is used to indicate termination, which is not used here—a state is finished when no further action can be taken. Another difference between the transition systems is that Zhang and Clark (2011a) omit the $\Delta$ field in each state, due to their use of a context-free, phrase-structure cover, and dependencies are recovered at a post-processing step; in my parsers, dependencies are built on-the-fly.

### 2.3.2 Statistical Shift-Reduce Parsing

From here on, I assume a statistical shift-reduce parser is made up of three components, namely a *deduction system* (which subsumes the transition system), a *model* and a *search strategy.* As can be seen above, the deduction system defines the symbolic rules of a parser, and given an input sentence $x$, it can be used to find the set of all possible valid shift-reduce action sequences. The goal of a statistical parser is to produce the most likely sequence $y^*$ from this set, with a model and a search strategy. More formally, the inference problem can be written as

$$y^* = \arg\max_{y \in \mathcal{Y}} \sum_{1 \leq t \leq |y|} \gamma\left(y_t, \langle \alpha, \beta \rangle_y^{t-1}; \theta\right),$$

where $\mathcal{Y}$ is the set of all possible shift-reduce action sequences given $x$, with each $y$ having length $|y|$; $y_t$ is the $t$th action in $y$ and $\langle \alpha, \beta \rangle_y^{t-1}$ is the $(t-1)$th parser state in $y$; $\gamma$ is a *scoring function* obtained under the supervision of an *oracle*, with a *learning algorithm* that is associated with the model parametrized by $\theta$.

Because $\mathcal{Y}$ is exponentially-sized with respect to the length of a given $x$ in general, solving this inference problem exactly is often intractable. Instead, greedy search or beam search is often used to approximately find $y^*$. For all the shift-reduce models that I consider later, both of these two strategies can be used.

[8] Notation is slightly abused for the SHIFT deduction, where $x_{w_j}|\beta$ is used to denote the lexical category $x_{w_j}$ available for the front word on the queue.

Under greedy search, a locally optimal action ${y_t}^*$ is chosen at each step $t$ until the goal is reached, and the inference rule can be summarized as

$${y_t}^* = \underset{y_t \in \mathcal{T}\left(\langle \alpha, \beta \rangle_y^{t-1}\right)}{\arg\max} \; \gamma\left(y_t, \langle \alpha, \beta \rangle_y^{t-1}; \theta\right),$$

where $\mathcal{T}\left(\langle \alpha, \beta \rangle_y^{t-1}\right)$ is the set of all feasible actions for the parser state $\langle \alpha, \beta \rangle_y^{t-1}$.

With beam search, a beam is used to store the top-$k$ highest-scoring items at each step resulting from expanding all items in the previous beam, and the parser keeps track of the highest scored candidate output which is returned as the final output.

Compared with greedy search, the use of beam search allows the parser to explore a larger search space. More importantly, beam search can be naturally integrated with the learning algorithms to derive global shift-reduce parsing models (Zhang and Clark, 2008; Zhang and Clark, 2011a; Xu et al., 2014; Xu et al., 2016; Xu, 2016), as in the following chapters.

# Chapter 3

# Shift-Reduce CCG Parsing with a Dependency Model

Typed dependency structures recovered by CCG (Hockenmaier, 2003; Clark and Curran, 2007) (§2.1.4) provide a useful approximation to the underlying predicate-argument relations of "who did what to whom". To date, CCG remains the most competitive formalism for recovering "deep" dependencies arising from many linguistic phenomena such as raising, control, extraction and coordination (Rimell et al., 2009; Nivre et al., 2010).

To achieve its expressiveness, CCG exhibits so-called "spurious" ambiguity, permitting many non-standard surface derivations which ease the recovery of certain dependencies, especially those arising from type-raising and composition (§2.1.1). But this raises the question of what is the most suitable model for CCG: Should we model the derivations, the dependencies, or both? The choice for some existing parsers (Hockenmaier, 2003; Clark and Curran, 2007) is to model derivations directly, restricting the gold standard to be the normal-form derivations (Eisner, 1996a) from CCGBank (Hockenmaier, 2003; Hockenmaier and Steedman, 2007).

Modelling dependencies, as a proxy for the semantic interpretation, fits well with the theory of CCG, in which Steedman (2000) argues that the derivation is merely a "trace" of the underlying syntactic process, and that the structure which is built, and predicated over when applying constraints on grammaticality, is the semantic interpretation. The early dependency model of Clark et al. (2002), in which model features were defined over *only* dependency structures, was partly motivated by these theoretical observations.

More generally, dependency models are desirable for a number of reasons. First, modelling dependencies provides an elegant solution to the spurious ambiguity problem (Clark and Curran, 2007). Second, obtaining training data for dependencies is likely to be easier than for syntactic derivations, especially for incomplete data (Schneider et al., 2013). Clark and Curran (2006) show how the dependency model from Clark and Curran (2007) extends naturally to the partial-training case, and also how to obtain dependency data cheaply from gold standard lexical category sequences alone. And third, it has been argued that dependencies are an ideal representation for parser evaluation, especially for CCG (Briscoe and Carroll, 2006; Clark and Hockenmaier, 2002), and so optimizing for dependency recovery makes sense from an evaluation perspective.

The first shift-reduce model for CCG is due to Zhang and Clark (2011a), and it is a normal-form model, where the oracle for each sentence specifies a unique sequence of gold standard actions that produces the corresponding normal-form derivation (§2.3); no dependency structures are involved at training and test time, except for evaluation. Here I fill a gap in the literature by developing the first dependency model for a shift-reduce CCG parser, which considers all sequences of actions producing a gold standard dependency structure to be correct. Each training instance in this model is an input sentence paired with its CCG dependency structure (§2.1.4), and the selection of shift-reduce action sequences producing the dependency structure is treated as a hidden variable during training.

The same as the normal-form model, the dependency model preserves the left-to-right, incremental nature of shift-reduce parsing, which fits with CCG's cognitive claims (Ambati et al., 2015). In particular, it has the same transition and deduction systems as the normal-form model (Zhang and Clark, 2011a); it is also discriminative and global (Zhang and Clark, 2008) and uses beam search (Collins and Roark, 2004) with the advantage of linear-time inference (Goldberg et al., 2013).

| | SR | $\Delta$ | $\mathcal{D}_f$ | $\Delta_f$ |
|---|---|---|---|---|
| Clark et al. (2002) | ✗ | ✓ | ✗ | ✓ |
| Clark and Curran (2007) | ✗ | ✓ | ✓ | ✓ |
| Zhang and Clark (2011a) | ✓ | ✗ | ✓ | ✗ |
| **present work** | ✓ | ✓ | ✓ | ✓ |

Table 3.1: Comparison of the dependency model in this chapter with the chart-based dependency models in Clark et al. (2002) and Clark and Curran (2007). Zhang and Clark (2011a) is a shift-reduce normal-form model. SR (Shift-Reduce); $\Delta$ (Dependency model); $\mathcal{D}_f$ (Derivation features); $\Delta_f$ (Dependency features).

As the first contribution of this chapter, I develop a dependency oracle (§3.1). I then show how it can be integrated with online structured perceptron learning and beam search (Collins and Roark, 2004) by generalizing early update (Collins and Roark, 2004) under the violation-fixing perceptron framework (Huang et al., 2012) (§3.2). By also taking advantage of a rich feature set incorporating both features from the normal-form shift-reduce model (Zhang and Clark, 2011a) and the chart-based dependency model (Clark and Curran, 2007) (see Table 3.1), the final shift-reduce parser outperforms the chart-based dependency models of Clark and Curran (2007) as well as the shift-reduce normal-form model of Zhang and Clark (2011a), showing up to 1.84 labeled F1 improvements.

One possible way to view the present dependency model is through the lens of a dynamic oracle (Goldberg and Nivre, 2012), but it is worth pointing out that this view should only be restricted to the high-level definition of the dependency oracle function (Eq. 5, §3.1.2). Other than this, the dependency model has different motivations from Goldberg and Nivre (2012), and it has been formulated independently. In particular, the dynamic oracle of Goldberg and Nivre (2012) is tailored for the specific class of dependency grammar they consider, whereas the CCG parser I consider allows great flexibility in dependency realization with a unification mechanism (§2.1.4). This flexibility also calls for a novel oracle and training method to handle the resulting algorithmic and structured learning challenges, which are addressed below.

## 3.1 The Dependency Oracle

The chart-based dependency model of Clark and Curran (2007) treats all derivations as hidden, and defines a probabilistic model for a dependency structure by summing probabilities of all derivations leading to a particular structure. Features are defined over both derivations and CCG predicate-argument dependencies. I follow a similar approach, but rather than define a probabilistic model (which requires summing), I define a linear model over sequences of shift-reduce actions, as for the normal-form shift-reduce model. However, the difference compared to the normal-form model is that I do not assume a single gold standard sequence of actions for each input.

More specifically, I define an *oracle* which determines, for a gold standard dependency structure, $G$, what the valid transition sequences are (i.e., those sequences producing derivations leading to $G$). As such, given $G$ and a parser state $\langle \delta, \beta \rangle$, the oracle can determine what the valid actions are for that state (i.e., what actions can potentially lead to $G$, starting with $\langle \delta, \beta \rangle$ and the dependencies already built on $\delta$). Because there can be exponentially many valid action sequences for $G$, I opt to represent them using a packed parse forest, and show how the forest can be used, during beam search at training time, to determine the set of valid actions for a given state.

### 3.1.1 The Oracle Forest

A CCG parse forest efficiently represents an exponential number of derivations. Following Clark and Curran (2007) (which builds on Miyao and Tsujii (2002)), and using the same notation, I define a CCG parse forest $\Phi$ as a tuple $\langle C, D, R, \gamma, \delta \rangle$, where $C$ is a set of conjunctive nodes and $D$ is a set of disjunctive nodes.[1] More specifically, conjunctive nodes are individual CCG categories in $\Phi$, and are either obtained from the lexicon, or by combining two disjunctive nodes using a CCG rule, or by applying a unary rule to a disjunctive node. Disjunctive nodes are equivalence classes of conjunctive nodes. Two conjunctive nodes are equivalent iff they have the same category, head and unfilled dependencies (i.e., they will lead to the same derivation,

[1] Under the hypergraph framework (Gallo et al., 1993; Huang and Chiang, 2005), a conjunctive node corresponds to a hyperedge and a disjunctive node corresponds to the head of a hyperedge or hyperedge bundle.

and produce the same dependencies, in any future parsing). Moreover, let $R \subseteq D$ be the set of root disjunctive nodes; let $\gamma : D \rightarrow 2^C$ be the conjunctive child function, which returns the set of all conjunctive child nodes of a disjunctive node; and let $\delta : C \rightarrow 2^D$ be the disjunctive child function, which returns the disjunctive child nodes of a conjunctive node.

The dependency model requires all the conjunctive and disjunctive nodes of $\Phi$ that are part of the derivations leading to a gold standard dependency structure $G$. I refer to such derivations as *correct* derivations and the packed forest containing all these derivations as the *oracle forest*, denoted as $\Phi_G$, which is a subset of $\Phi$. It is prohibitive to enumerate all correct derivations, but it is possible to identify, from $\Phi$, all the conjunctive and disjunctive nodes that are part of $\Phi_G$. Clark and Curran (2007) give an algorithm for doing so, which I use here. The main intuition behind the algorithm is that a gold standard dependency structure decomposes over derivations, thus gold standard dependencies realized at conjunctive nodes can be counted when $\Phi$ is built, and all nodes that are part of $\Phi_G$ can then be marked out of $\Phi$ by traversing it top-down. A key idea in understanding the algorithm is that dependencies are created when disjunctive nodes are combined, and hence are associated with, or "live on", conjunctive nodes in the forest.

Following Clark and Curran (2007), I also define the following three values, where the first decomposes only over local rule productions, while the other two decompose over derivations:

$$cdeps(c) = \begin{cases} * \text{ if } \exists\, \tau \in deps(c), \tau \notin G \\ |deps(c)| \text{ otherwise;} \end{cases}$$

$$dmax(c) = \begin{cases} * \text{ if } cdeps(c) == * \\ * \text{ if } dmax(d) == * \text{ for some } d \in \delta(c) \\ \sum_{d \in \delta(c)} dmax(d) + cdeps(c) \text{ otherwise;} \end{cases}$$

$$dmax(d) = \max\{dmax(c) \mid c \in \gamma(d)\},$$

```
function MARK-ORACLE($\langle C, D, R, \gamma, \delta \rangle$)
  for each $d_r \in R$ s.t. $dmax(d_r) = |G|$ do        ▷ each root disjunctive node $d_r$
    MARK($d_r$)
procedure MARK($d$)
  mark $d$ as a correct node
  for each $c \in \gamma(d)$ do                          ▷ each child conjunctive node of $d$
    if $dmax(c) == dmax(d)$ then
      mark $c$ as a correct node
      for each $d' \in \delta(c)$ do                     ▷ each child disjunctive node of $c$
        if $d'$ has not been visited then
          MARK($d'$)
```

Figure 3-1: The forest marking algorithm of Clark and Curran (2007); the input is a parse forest $\langle C, D, R, \gamma, \delta \rangle$ with $dmax(c)$ and $dmax(d)$ already computed.

where $deps(c)$ is the set of all dependencies on conjunctive node $c$, and $cdeps(c)$ counts the number of *correct* dependencies on $c$; $dmax(c)$ is the maximum number of correct dependencies over any sub-derivation headed by $c$ and is calculated recursively; $dmax(d)$ returns the same value for a disjunctive node. In all cases, a special value $*$ indicates the presence of incorrect dependencies. To obtain the oracle forest, I first precompute $dmax(c)$ and $dmax(d)$ for all $d$ and $c$ in $\Phi$ when $\Phi$ is built using CKY, then I use the algorithm given in Fig. 3-1 to identify all the conjunctive and disjunctive nodes in $\Phi_G$.

### 3.1.2 The Dependency Oracle Algorithm

Observe that the canonical shift-reduce algorithm applied to a single parse tree exactly resembles bottom-up post-order traversal of that tree. As an example, consider the derivation in Fig. 3-2a, where the corresponding sequence of actions is: SHIFT ⇒ $N/N$, SHIFT ⇒ $N$, REDUCE ⇒ $N$, UNARY ⇒ $NP$, SHIFT ⇒ $(S[dcl]\backslash NP)/NP$, SHIFT ⇒ $NP$, REDUCE ⇒ $S[dcl]\backslash NP$, REDUCE ⇒ $S[dcl]$.

The order of traversal is left-child, right-child and parent. For a single parse, the corresponding shift-reduce action sequence is unique, and for a given parser state this canonical order restricts the possible derivations that can be formed using further actions. I now extend this observation to the more general case of an oracle forest,

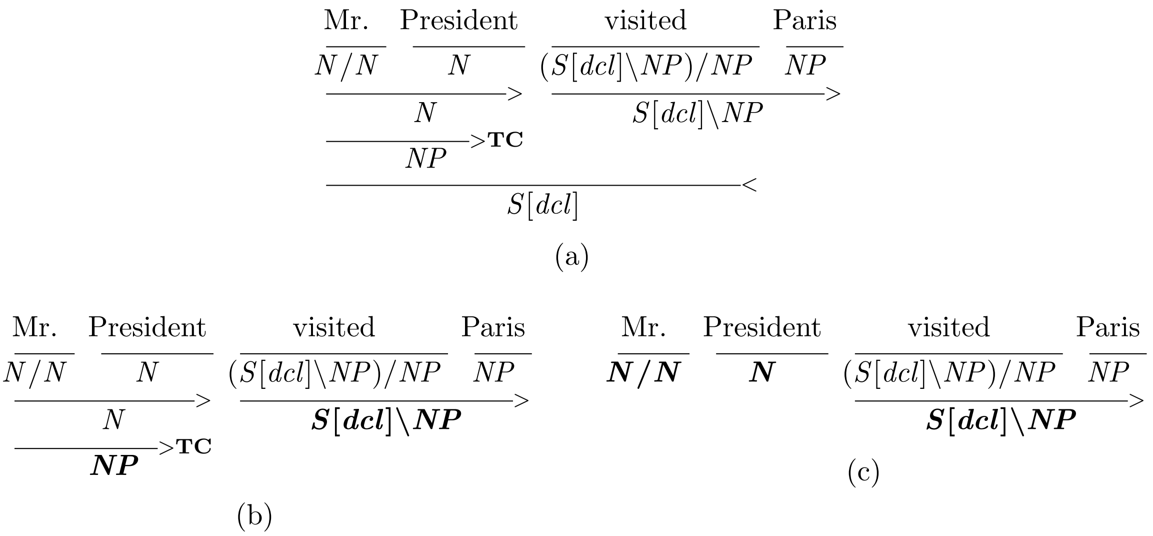


Figure 3-2: Example subtrees on two stacks assuming $\Phi_G$ contains only the derivation in (a); two subtrees in (b) and three in (c), and roots of subtrees are in bold.

where there may be more than one valid action available for a given state.

**Definition 3.** Given a gold standard dependency structure $G$, an oracle forest $\Phi_G$, and a parser state $\langle \delta, \beta \rangle$, $\delta$ is a said to be a **realization** of $G$, denoted $\delta \simeq G$, if $|\delta| = 1$, $\beta$ is empty, and the single derivation on $\delta$ is correct. If $|\delta| > 0$ and the subtrees on $\delta$ can lead to a correct derivation in $\Phi_G$ using further actions, $\delta$ is said to be a **partial-realization** of $G$, denoted as $\delta \sim G$. If $|\delta| = 0$, $\delta \sim G$.

As an example, assume that $\Phi_G$ contains only the derivation in Fig. 3-2a, then a stack containing the two subtrees in Fig. 3-2b is a partial-realization, while a stack containing the three subtrees in Fig. 3-2c is not. Note that each of the three subtrees in Fig. 3-2c is present in $\Phi_G$ (Fig. 3-2a); however, these subtrees cannot be combined into the correct derivation, since the correct action sequence must first combine the lexical categories for *Mr.* and *President* before shifting the lexical category for *visited.*

Let $(x, c)$ denote an action pair, where $x \in \{\textsc{shift}, \textsc{reduce}, \textsc{unary}\}$ and $c$ is the root of the subtree resulting from that action, which corresponds to a unique conjunctive node in the *complete* forest $\Phi$. Let $c_{s_i}$ denote the conjunctive node in $\Phi$ corresponding to subtree $s_i$ on a stack; let $\langle \delta', \beta' \rangle = \langle \delta, \beta \rangle \circ (x, c)$ be the resulting parser state from applying the action $(x, c)$ to $\langle \delta, \beta \rangle$; let the set of all possible actions for $\langle \delta, \beta \rangle$ be $\mathcal{T}(\langle \delta, \beta \rangle) = \{(x, c) \mid (x, c) \text{ is applicable to } \langle \delta, \beta \rangle\}$.

**Definition 4.** Given $\Phi_G$ and a parser state $\langle \delta, \beta \rangle$ s.t. $\delta \sim G$, an applicable action $(x, c)$ for the state is said to be **valid** iff $\delta' \sim G$ or $\delta' \simeq G$, where $\langle \delta', \beta' \rangle = \langle \delta, \beta \rangle \circ (x, c)$.

**Definition 5.** Given $\Phi_G$, the dependency oracle function $f_d$ is defined as:

$$f_d(\langle \delta, \beta \rangle, (x, c), \Phi_G) = \begin{cases} \textit{true} & \text{if } \delta' \sim G \text{ or } \delta' \simeq G \\ \textit{false} & \text{otherwise,} \end{cases}$$

where $(x, c) \in \mathcal{T}(\langle \delta, \beta \rangle)$ and $\langle \delta', \beta' \rangle = \langle \delta, \beta \rangle \circ (x, c)$.

The pseudocode in Fig. 3-3 implements $f_d$. It determines, for a given parser state, whether an applicable action is valid in $\Phi_G$.

It is trivial to determine the validity of a SHIFT action for the initial state, $\langle \delta, \beta \rangle_0$, since the SHIFT action is valid iff its category matches the gold standard lexical category of the first word in the sentence. For any subsequent SHIFT action $(\text{SHIFT}, c)$ to be valid, the *necessary* condition is $c \equiv c_{lex_0}$, where $c_{lex_0}$ denotes the gold standard lexical category of the front word in the queue, $\delta_0$ (line 3). However, this condition is not sufficient; a counterexample is where all the gold standard lexical categories for the sentence in Fig. 3-2a are shifted in succession. Hence, in general, the conditions under which an action is valid are more complex than the trivial case above.

First, for ease of exposition, suppose again that there is only one correct derivation in $\Phi_G$. A SHIFT action $(\text{SHIFT}, c_{lex_0})$ is valid whenever $c_{\delta_0}$ (the conjunctive node in $\Phi_G$ corresponding to the subtree $\delta_0$ on the stack) and $c_{lex_0}$ (the conjunctive node in $\Phi_G$ corresponding to the next gold standard lexical category from the queue) are both dominated by the conjunctive node parent $p$ of $c_{\delta_0}$ in $\Phi_G$.[2] A REDUCE action $(\text{REDUCE}, c)$ is valid if $c$ matches the category of the conjunctive node parent of $c_{\delta_0}$ and $c_{\delta_1}$ is in $\Phi_G$. A UNARY action $(\text{UNARY}, c)$ is valid if $c$ matches the conjunctive node parent of $c_{\delta_0}$ in $\Phi_G$. I now generalize the case where $\Phi_G$ contains a single correct derivation to the case of an oracle forest, where each parent $p$ is replaced by a set of conjunctive nodes in $\Phi_G$.

[2] Strictly speaking, the conjunctive node parent is a parent of the disjunctive node containing the conjunctive node $c_{\delta_0}$, but I will continue to use this shorthand for parents of conjunctive nodes throughout this chapter.

**Definition 6.** The **left parent set** $\mathcal{P}_L(c)$ of a conjunctive node $c \in \Phi_G$ is the set of all parent conjunctive nodes of $c$ in $\Phi_G$, which have the disjunctive node $d$ containing $c$ (i.e., $c \in \gamma(d)$) as a left child.

**Definition 7.** The **ancestor set** $\mathcal{A}(c)$ of conjunctive node $c \in \Phi_G$ is the set of all reachable ancestor conjunctive nodes of $c$ in $\Phi_G$.

**Definition 8.** Given a parser state $\langle \delta, \beta \rangle$, $\delta$ is said to be a **frontier stack** if $|\delta| = 1$.

A key to defining the dependency oracle function is the notion of a **shared ancestor set**. Intuitively, shared ancestor sets are built up through SHIFT actions, and they contain sets of nodes which can potentially become the results of REDUCE or UNARY actions. A further intuition is that shared ancestor sets define the space of possible correct derivations, and nodes in these sets are "ticked off" when REDUCE and UNARY actions are applied, as a single correct derivation is built through the shift-reduce process (corresponding to a bottom-up post-order traversal of the derivation). The following definition shows how the dependency oracle function builds shared ancestor sets for each action type.

**Definition 9.** Let $\langle \delta, \beta \rangle$ be a parser state and let $\langle \delta', \beta' \rangle = \langle \delta, \beta \rangle \circ (x, c)$. The **shared ancestor set** $\mathcal{R}(c_{\delta'_1}, \mathbf{c}_{\delta'_0})$ of $\mathbf{c}_{\delta'_0}$, obtained after applying action $(x, c)$, is defined as:

- $\{c' \mid c' \in \mathcal{P}_L(c_{\delta_0}) \cap \mathcal{A}(c)\}$, if $\delta$ is frontier and $x =$ SHIFT;
- $\{c' \mid c' \in \mathcal{P}_L(c_{\delta_0}) \cap \mathcal{A}(c)$ and $\mathcal{R}(c_{\delta_1}, \mathbf{c}_{\delta_0}) \cap \mathcal{A}(c') \neq \emptyset\}$, if $\delta$ is non-frontier and $x =$ SHIFT;
- $\{c' \mid c' \in \mathcal{R}(c_{\delta_2}, \mathbf{c}_{\delta_1}) \cap \mathcal{A}(c)\}$, if $x =$ REDUCE;
- $\{c' \mid c' \in \mathcal{R}(c_{\delta_1}, \mathbf{c}_{\delta_0}) \cap \mathcal{A}(c)\}$, if $\delta$ is non-frontier and $x =$ UNARY;
- $\mathcal{R}(\epsilon, \mathbf{c}^0_{\delta_0}) = \emptyset$, where $\mathbf{c}^0_{\delta_0}$ is the conjunctive node corresponding to the gold standard lexical category of the first word in the sentence.[3]

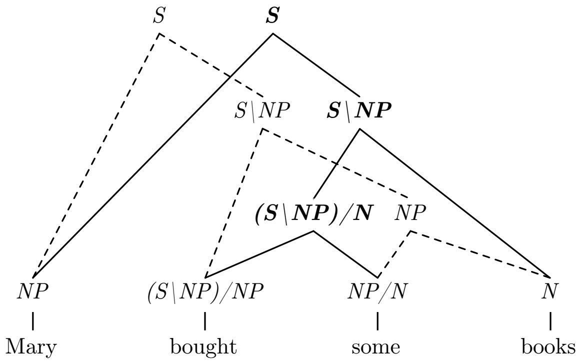
[3] With a slight abuse of notation, I use $\epsilon$ to indicate the bottom of stack.

```
**function** DEP-ORACLE($\langle\delta,\beta\rangle$, $(x,c)$, $\Phi_G$)
  **if** $x$ is SHIFT **then**
    **if** $c \not\equiv c_{lex_0}$ **then**                        ▷ c not gold lexical category
      **return false**
    **else if** $c \equiv c_{lex_0}$ and $|\delta| = 0$ **then**        ▷ the initial state
      **return true**
    **else if** $c \equiv c_{lex_0}$ and $|\delta| \neq 0$ **then**
      compute $\mathcal{R}(c_{\delta'_1}, \mathbf{c}_{\delta'_0})$
      **return** $\mathcal{R}(c_{\delta'_1}, \mathbf{c}_{\delta'_0}) \neq \emptyset$

  **if** $x$ is REDUCE **then**                                  ▷ $\delta$ is non-frontier
    **if** $c \in \mathcal{R}(c_{\delta_1}, \mathbf{c}_{\delta_0})$ **then**
      compute $\mathcal{R}(c_{\delta'_1}, \mathbf{c}_{\delta'_0})$
      **return true**
    **else return false**

  **if** $x$ is UNARY **then**
    **if** $|\delta| = 1$ **then**                                 ▷ $\delta$ is frontier
      **return** $c \in \Phi_G$
    **if** $|\delta| \neq 1$ **and** $c \in \Phi_G$ **then**                ▷ $\delta$ is non-frontier
      compute $\mathcal{R}(c_{\delta'_1}, \mathbf{c}_{\delta'_0})$
      **return** $\mathcal{R}(c_{\delta'_1}, \mathbf{c}_{\delta'_0}) \neq \emptyset$
```

Figure 3-3: The dependency oracle algorithm that implements $f_d$ (Definition 5). The input consists of an oracle parse forest $\Phi_G$, a parser state $\langle\delta,\beta\rangle$ s.t. $\delta \sim G$, and an applicable action $(x,c) \in \mathcal{T}(\langle\delta,\beta\rangle)$.

The base case for Definition 9 is when the gold standard lexical category of the first word in the sentence has been shifted, which creates an empty shared ancestor set. Furthermore, the shared ancestor set is always empty for frontier stacks.

The dependency oracle algorithm checks the validity of applicable actions (Fig. 3-3). A SHIFT action is valid if $\mathcal{R}(c_{\delta'_1}, \mathbf{c}_{\delta'_0}) \neq \emptyset$ for the resulting stack $\delta'$. A valid REDUCE action consumes $\delta_1$ and $\delta_0$. For the newly produced node, its shared ancestor set is the subset of the conjunctive nodes in $\mathcal{R}(c_{\delta_2}, \mathbf{c}_{\delta_1})$ which dominate the resulting conjunctive node of a valid REDUCE action. The UNARY case for a frontier stack is trivial: an UNARY action is valid if it is in $\Phi_G$. For a non-frontier stack, the UNARY case is similar to REDUCE, except the resulting shared ancestor set is a subset of $\mathcal{R}(c_{\delta_1}, \mathbf{c}_{\delta_0})$.



| | stack $(s_n, ..., s_1, s_0)$ | $\mathcal{R}(c_{\delta'_1}, \mathbf{c}_{\delta'_0})$ | computation |
|---|---|---|---|
| SHIFT | $NP$ | $()$ | |
| SHIFT | $NP\ (S\backslash NP)/NP$ | $(S, \boldsymbol{S})$ | $\mathcal{P}_L(NP) \cap \mathcal{A}((S\backslash NP)/NP)$ |
| SHIFT | $NP\ (S\backslash NP)/NP\ NP/N$ | $(S\backslash NP, \boldsymbol{(S\backslash NP)/N})$ | $\mathcal{P}_L((S\backslash NP)/NP) \cap \mathcal{A}(NP/N)$ |
| SHIFT | $NP\ (S\backslash NP)/NP\ NP/N\ N$ | $(NP)$ | $\mathcal{P}_L(NP/N) \cap \mathcal{A}(N)$ |
| REDUCE | $NP\ (S\backslash NP)/NP\ NP$ | $(S\backslash NP)$ | $\mathcal{R}((S\backslash NP)/NP, NP/N) \cap \mathcal{A}(NP)$ |
| REDUCE | $NP\ S\backslash NP$ | $(S)$ | $\mathcal{R}(NP, (S\backslash NP)/NP) \cap \mathcal{A}(S\backslash NP)$ |
| REDUCE | $S$ | $()$ | |

Figure 3-4: Example of the dependency oracle algorithm in action.

Fig. 3-4 shows an example in action. The parse forest is shown above the table, where both derivations are correct. From left to right, the table shows the trace of the action sequence producing the dashed correct tree, the trace of the stack, and the shared ancestor sets along with the computations invoked to obtain them. For the first SHIFT, lines $5-6$ of Fig. 3-3 are executed, and the base case in Definition 9 applies, where $\mathcal{R}(NP, (S\backslash NP)/NP)$ is empty. For the second SHIFT, lines $7-9$ are executed, and the frontier SHIFT case in Definition 9 applies. The third SHIFT is a non-frontier SHIFT case, and according to Definition 9, the extra checks needed (which are omitted in Fig. 3-4) are $\mathcal{R}(NP, (S\backslash NP)/NP) \cap \mathcal{A}(S\backslash NP) \neq \emptyset$ and $\mathcal{R}(NP, (S\backslash NP)/NP) \cap \mathcal{A}(\boldsymbol{(S\backslash NP)/N}) \neq \emptyset$; the fourth SHIFT is similar to the third. The first REDUCE produces $NP$, and is valid because $NP \in \mathcal{R}(NP/N, N)$ (line 11 in Fig. 3-3), and the resulting shared ancestor set is $\mathcal{R}((S\backslash NP)/NP, NP)$. The remaining REDUCE cases are treated similarly.

I now turn to the problem of finding the shared ancestor sets. In practice, I do not traverse $\Phi_G$ top-down from the conjunctive nodes in $\mathcal{P}_L(c_{\delta_0})$ on-the-fly to find each member of $\mathcal{R}$. Instead, when $\Phi_G$ is built (using CKY), I precompute reachable disjunctive nodes of each conjunctive node $c$ in $\Phi_G$ as

$$\mathcal{D}(c) = \delta(c) \cup (\cup_{c' \in \gamma(d), d \in \delta(c)}(\mathcal{D}(c'))),$$

where each such $\mathcal{D}$ is implemented as a hash map to allow the membership of one potential conjunctive node to be tested in $\mathcal{O}(1)$ time: a conjunctive node $c \in \mathcal{P}_L(c_{\delta_0})$ is reachable from $c_{lex_0}$ if there is a disjunctive node $d \in \mathcal{D}(c)$ s.t. $c_{lex_0} \in \gamma(d)$. With this implementation, the complexity of checking each SHIFT action is $\mathcal{O}(|\mathcal{P}_L(c_{\delta_0})|)$.

## 3.2 Training

Clark and Curran (2007) introduced three probabilistic log-linear CCG parsing models, including a normal-form model, a dependency model and a hybrid dependency model (which is the same as the dependency model except in addition it uses the Eisner constraint), and all three models are based on exact search using CKY. For each model, training involves constructing complete parse forests for the input sentences, and estimating the parameters over them—there is no interaction between learning and search, and no pruning of the search space.[4] More specifically, the same as shift-reduce parsing, in the normal-form model, only one gold standard derivation appears in a parse forest; in the dependency models, all derivations consistent with a dependency structure are considered to be correct. In both cases, parameters are estimated in an identical manner (modulo minor modifications of the objective function). In this section, I first review parameter estimation of the chart-based models (which draws heavily on Miyao and Tsujii (2002)) to motivate the problem of reconciling learning and search in the shift-reduce dependency model.

### 3.2.1 Training Chart-Based Models

In all three chart-based models, the probability of a parse is defined in the same parametric form; the distinction lies in how an actual parse is instantiated.

[4] There is pruning at the supertagging stage, but no pruning whatsoever at the parsing stage (Clark and Curran, 2007).

**Definition 10.** A parse $y$ in the normal-form model is a head-lexicalized CCG derivation $d$.

**Definition 11.** A parse $y$ in the dependency model is a CCG derivation $d$ coupled with its dependency structure $\Delta_d$, denoted as $\langle d, \Delta_d \rangle$.

**Definition 12.** The conditional probability of a parse $y$ given a sentence $x$ is defined as

$$p(y|x) = \frac{1}{Z_x} \exp\left\{\mathbf{w} \cdot \phi(x, y)\right\},$$

where $Z_x = \sum_{y'} \exp\left\{\mathbf{w} \cdot \phi(x, y')\right\}$ is a global normalization term over the set of all possible parses for $x$; $\mathbf{w}$ is the parameter vector of the model; and $\phi$ is a function that returns the feature frequency vector.

Note that in the normal-form model, the packed chart for each input almost always contains non-normal-form derivations, because it is not possible to completely eliminate all non-normal-form derivations even with the Eisner constraint (§2.1.1), but the gold standard derivations are normal-form CCGBank derivations.

**Chart-based Normal-Form Model.** This is the chart-based counterpart of the shift-reduce normal-form model (Zhang and Clark, 2011a), and the training data consists of sentence and normal-form derivation pairs, denoted as $\{(x_i, d_i)\}_{i=1}^{n}$. The conditional log-likelihood objective function is defined as

$$\begin{aligned} J'(\mathbf{w}) &= J(\mathbf{w}) - G(\mathbf{w}) \\ &= \log \prod_{i=1}^{n} p_{\mathbf{w}}(d_i|x_i) - \sum_{w_j \in \mathbf{w}} \frac{w_j^2}{2\sigma^2}, \end{aligned}$$

where $w_j$ is the $j$th component of $\mathbf{w}$; $G(\mathbf{w})$ is a Gaussian prior for smoothing that is parametrized by $\sigma$ for all values of $j$.

The gradients of the objective are obtained as

$$\frac{\partial J'(\mathbf{w})}{\partial w_j} = \sum_{i=1}^{n} \phi_j(x_i, d_i) - \sum_{i=1}^{n} \sum_{d \in \mathcal{D}_{x_i}} \frac{\exp\{\mathbf{w} \cdot \phi(x_i, d)\} \phi_j(x_i, d)}{\sum_{d' \in \mathcal{D}_{x_i}} \exp\{\mathbf{w} \cdot \phi(x_i, d')\}} - \frac{w_j}{\sigma^2},$$

where $\mathcal{D}_{x_i}$ is the set of all possible parses (i.e., normal-form derivations) for $x_i$ and $\phi_j$ is the $j$th component of $\phi$.

The first term in the gradient computation takes into account only the gold standard derivation (the empirical expectations), while the second term deals with the expectations over all derivations contained in the chart (the model expectations).

**Chart-based Dependency Model.** The same as the shift-reduce dependency model, the training data for this model consists of sentence and dependency structure pairs, denoted as $\{(x_i, \Delta_{x_i})\}_{i=1}^n$ and the objective function is

$$\begin{aligned}
J'(\mathbf{w}) &= J(\mathbf{w}) - G(\mathbf{w}) \\
&= \log \prod_{i=1}^{n} p_{\mathbf{w}}(\Delta_{x_i}|x_i) \ - \sum_{w_j \in \mathbf{w}} \frac{w_j^2}{2\sigma^2} \\
&= \sum_{i=1}^{n} \log \frac{\sum_{d \in \Phi(\Delta_{x_i})} \exp\{\mathbf{w} \cdot \phi(x_i, \langle d, \Delta_{x_i} \rangle)\}}{\sum_{\langle d', \Delta_{d'} \rangle \in \Lambda(x_i)} \exp\{\mathbf{w} \cdot \phi(x_i, \langle d', \Delta_{d'} \rangle)\}} \ - \sum_{w_j \in \mathbf{w}} \frac{w_j^2}{2\sigma^2} \\
&= \sum_{i=1}^{n} \log \sum_{d \in \Phi(\Delta_{x_i})} \exp\{\mathbf{w} \cdot \phi(x_i, \langle d, \Delta_{x_i} \rangle)\} \\
&- \sum_{i=1}^{n} \log \sum_{\langle d', \Delta_{d'} \rangle \in \Lambda(x_i)} \exp\{\mathbf{w} \cdot \phi(x_i, \langle d', \Delta_{d'} \rangle)\} \ - \sum_{w_j \in \mathbf{w}} \frac{w_j^2}{2\sigma^2},
\end{aligned}$$

where $\Phi(\Delta_{x_i})$ is the oracle forest for the dependency structure $\Delta_{x_i}$, and $\Lambda(x_i)$ is the set of all possible parses (i.e., derivation and dependency structure pairs).

The gradients are obtained as

$$\begin{aligned}
\frac{\partial J'(\mathbf{w})}{\partial w_j} &= \sum_{i=1}^{n} \sum_{d \in \Phi(\Delta_{x_i})} \frac{\exp\{\mathbf{w} \cdot \phi(x_i, \langle d, \Delta_{x_i} \rangle)\} \phi_j(x_i, \langle d, \Delta_{x_i} \rangle)}{\sum_{d \in \Phi(\Delta_{x_i})} \exp\{\mathbf{w} \cdot \phi(x_i, \langle d, \Delta_{x_i} \rangle)\}} \\
&- \sum_{i=1}^{n} \sum_{\langle d', \Delta_{d'} \rangle \in \Lambda(x_i)} \frac{\exp\{\mathbf{w} \cdot \phi(x_i, \langle d', \Delta_{d'} \rangle)\} \phi_j(x_i, \langle d', \Delta_{d'} \rangle)}{\sum_{\langle d', \Delta_{d'} \rangle \in \Lambda(x_i)} \exp\{\mathbf{w} \cdot \phi(x_i, \langle d', \Delta_{d'} \rangle)\}} - \frac{w_j}{\sigma^2}
\end{aligned}$$

For both the normal-form and dependency models, obtaining feature expectations requires summing over all possible derivations for an input, and this is made possible by CKY, in combination with the supertagger so that packed charts can be built with reasonable amount of space. The calculations are identical in both cases (modulo the definition of a parse $y$), in which the feature expectation of the $j$th feature is

$$\mathbb{E}(\phi_j) = \sum_{i=1}^{n} \frac{1}{Z_{x_i}} \sum_{y} \exp\{\mathbf{w} \cdot \phi(x_i, y)\} \phi_j(x_i, y).$$

To do this efficiently, the inside ($\alpha$) and outside ($\beta$) scores of the conjunctive nodes in the parse forest for $x_i$, denoted as $C(x_i)$, can be used, where

$$\mathbb{E}(\phi_j) = \sum_{i=1}^{n} \frac{1}{Z_{x_i}} \sum_{c \in C(x_i)} \phi_j(x_i, c) \alpha_c \beta_c.$$

**Viterbi and Minimal-Risk Inference.** It is straightforward to use the Viterbi algorithm to return the 1-best derivation for the normal-form model. However, for the dependency model, it seeks to return the highest-scoring dependency structure. Clark and Curran (2007) achieve this by adapting the labeled recall algorithm of Goodman (1996), which is a form of minimal-risk inference that goes through all dependency structures and returns the one that maximizes the expected recall rate (or equivalently unnormalized expected recall):

$$\Delta^* = \arg\max_{\Delta} \sum_{\Delta'} p(\Delta'|x_i)|\Delta' \cap \Delta|,$$

which can be written more explicitly as

$$\begin{aligned}
\Delta^* &= \arg\max_{\Delta} \sum_{\Delta'} p(\Delta'|x_i) \sum_{\tau \in \Delta} \mathbb{1}_{\tau \in \Delta'} \\
&= \arg\max_{\Delta} \sum_{\tau \in \Delta} \sum_{\Delta' | \tau \in \Delta'} p(\Delta'|x_i) \\
&= \arg\max_{\Delta} \sum_{\tau \in \Delta} \sum_{d \in \Phi(\Delta') | \tau \in \Delta'} p(d|x_i),
\end{aligned}$$

where $\Delta'$ ranges over all possible dependency structures for $x_i$; $\tau$ is an individual dependency in a dependency structure; $\mathbb{1}$ is an indicator function with the condition $\tau \in \Delta'$; and $\Phi(\Delta')$ is the parse forest corresponding to the dependency structure $\Delta'$.

Through the above derivation, each dependency structure $\Delta$ is decomposed into individual dependencies, and the expected recall score for each $\Delta$ is obtained as the sum of the scores of individual dependencies $\tau \in \Delta$. In turn, the score for each dependency $\tau$ can be calculated by summing all the derivations containing it, as represented by the inner sum. Again, an efficient implementation is possible with the inside and outside scores of conjunctives nodes in a parse forest, where the final expression becomes:

$$\Delta^* = \arg\max_{\Delta} \sum_{\tau \in \Delta} \frac{1}{Z_{x_i}} \sum_{c \in C(x_i)} \alpha_c \beta_c \text{ if } \tau \in \mathit{deps}(c),$$

which can also be obtained with the Viterbi algorithm.

As seen above, both the chart-based normal-form and the dependency models model derivations. But the dependency model is more flexible in allowing all derivations consistent with a dependency structure to be considered correct. This is the same in the shift-reduce dependency model, with the difference being the shift-reduce model has to do so with inexact search. At inference time, however, the shift-reduce dependency model stays identical with the shift-reduce normal-form model, with some potential overhead in feature computations, because additional dependency features not found in the normal-form model are incorporated (§3.3.1).

### 3.2.2 The Structured Perceptron with Inexact Search

The learning algorithm I use for the shift-reduce dependency model is based on the structured perceptron (Collins, 2002), which is a weighted linear model that extends the classic perceptron (Rosenblatt, 1958) and its voted/averaged versions (Freund and Schapire, 1999) to structured learning. In this section, I briefly review this model, and discuss the incremental variant of the standard structured perceptron (Collins and Roark, 2004), the associated early update mechanism (Collins and Roark, 2004), and how they are extended into the violation-fixing perceptron framework (Huang et al., 2012), which accommodates the inexactness of beam-search inference with formal guarantees for convergence. In the next section, the violation-fixing perceptron is seamlessly fused with the dependency oracle and beam search to learn the shift-reduce dependency model.

Given a set $\mathcal{X}$ representing the set of all possible inputs and a set $\mathcal{Y}$ of all possible outputs, the structured perceptron consists of the following four components (Collins, 2002):

- A training set $\{(x_i, y_i)\}_{i=1}^n$, where $x_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$.
- A generation function GEN($x$) that enumerates the set of all possible outputs for the input $x$.
- A feature frequency extractor function $\phi$ that defines a mapping from $\mathcal{X} \times \mathcal{Y}$ to $\mathbb{R}^d$, s.t. $\phi(x, y) \in \mathbb{R}^d$, where $d$ can either be preset or learned in an online fashion.
- A parameter vector $\mathbf{w} \in \mathbb{R}^d$ that the model is trying to estimate.

For each training instance $(x_i, y_i)$, the model assumes an exact solution can be found for the inference problem:

$$y^* \leftarrow \arg\max_{y \in \mathrm{GEN}(x_i)} \mathbf{w} \cdot \phi(x_i, y), \tag{3.1}$$

where the dot product $\mathbf{w} \cdot \phi(x_i, y)$ scores the output $y$ with the current weight vector. To estimate $\mathbf{w}$, the algorithm first initializes $\mathbf{w}$ to either $\mathbf{0}$ or random values, then it starts iterating the training set in multiple epochs. When a new training instance $(x_i, y_i)$ is encountered during an epoch, the above inference problem is solved, and the result $y^*$ is compared with $y_i$. If $y^*$ is correct, the model moves onto the next training instance; otherwise, a weight update occurs by adding $\phi(x_i, y_i)$ to and subtracting $\phi(x_i, y^*)$ from $\mathbf{w}$. This iteration process is continued for a preset number of epochs or until the model has reached convergence. The pseudocode for this process is shown in Fig. 3-5.

**Incremental Perceptron and Early Update.** The convergence of the standard perceptron is dependent on obtaining an exact solution for Eq. 3.1. In other words, it assumes exact search. For shift-reduce parsing, this is intractable, as invoking

```
$\mathbf{w} \leftarrow \mathbf{0}$                                ▷ the input is the training set $\{(x_i, y_i)\}_{i=1}^n$
while not converged do
  for $i \leftarrow 1, \ldots, n$ do
    $y^* \leftarrow \arg\max_{y \in \mathrm{GEN}(x_i)} \mathbf{w} \cdot \phi(x_i, y)$          ▷ obtain model prediction
    if $y^* \neq y_i$ then                                       ▷ $y^*$ not correct
      $\mathbf{w} \leftarrow \mathbf{w} + \phi(x_i, y_i) - \phi(x_i, y^*)$            ▷ online update
```

Figure 3-5: The structured perceptron (Collins, 2002).

GEN($x_i$) is prohibitive. However, Collins and Roark (2004) showed that the structured perceptron algorithm can be adapted to an incremental parser with beam search despite the violation of the search exactness assumption. In this incremental setting, the algorithm remains unchanged except beam search is used to replace GEN($x_i$) and to approximately find $y^*$ in Eq. 3.1.

Collins and Roark (2004) have also found that by stopping inference and updating the weights as soon as a search error has occurred increased both the accuracy of the resulting model and training efficiency. This strategy is referred to as early update, which locates a search error at one particular step of beam search. This is possible because the $j$th beam (for $j \geq 0$; $j = 0$ indicates the first beam containing only the initial parser state) contains only shift-reduce action sequences with length $j$. If the $j$th gold standard action has fallen outside of the $j$th beam, it can be guaranteed a search error has occurred.

**The Violation-Fixing Perceptron.** Early update first arose as an empirical technique to make weight updates less noisy (Collins, 2002). Huang et al. (2012) formalized it by observing the connection between convergence of the structured perceptron and *violations* of the model. Under exact search, if $y^* \neq y_i$, then it is true that $\mathbf{w} \cdot \phi(x_i, y^*) > \mathbf{w} \cdot \phi(x_i, y_i)$, that is, the 1-best output scores higher than the correct output. This condition is referred to as a violation, and it has been shown that if it holds for any $y_i$, the convergence of the structured perceptron is guaranteed (Huang et al., 2012). In other words, it can be said that exact search ensures *global* violations. Under inexact search, Huang et al. (2012) link search errors and *local* violations and show that as long as a violation is guaranteed at the site of search error for each

```
1:  $\mathbf{w} \leftarrow \mathbf{0}$; $\mathcal{B}_0 \leftarrow \emptyset$; $j \leftarrow 0$          ▷ the input is $(x_i, y_i)$
2:  $\mathcal{B}_0$.push($\langle \delta, \beta \rangle_0$)          ▷ the initial state
3:  $cand \leftarrow \emptyset$          ▷ candidate output priority queue
4:  **while** $\mathcal{B}_j \neq \emptyset$ **do**
5:    **for each** $\langle \delta, \beta \rangle \in \mathcal{B}_j$ **do**
6:      **if** $|\beta| = 0$ **then**          ▷ candidate output
7:        $cand$.push($\langle \delta, \beta \rangle$)
8:      expand $\langle \delta, \beta \rangle$ into $\mathcal{B}_{j+1}$
9:    $\mathcal{B}_{j+1} \leftarrow \mathcal{B}_{j+1}[1:k]$          ▷ apply beam of size $k$
10:   **if** $y_{ij+1} \notin \mathcal{B}_{j+1}$ **and** $cand[0] \neq y_i$ **then**
11:     $\mathbf{w} \leftarrow \mathbf{w} + \phi(x_i, y_{ij+1}) - \phi(x_i, \mathcal{B}_{j+1}[0])$          ▷ early update
12:     **return**
13:   $j \leftarrow j + 1$          ▷ continue to next step
14: **if** $cand[0] \neq y_i$ **then**          ▷ final update
15:   $\mathbf{w} \leftarrow \mathbf{w} + \phi(x_i, y_i) - \phi(x_i, cand[0])$
```

Figure 3-6: Normal-Form Model Training. The input $(x_i, y_i)$ consists of the sentence $x_i$ and the gold standard shift-reduce action sequence $y_i$; $y_{ij+1}$ denotes the $(j+1)$th action in $y_i$ (notation is abused on line 11, where $y_{ij+1}$ denotes the shift-reduce action sequence up to and including the $(j+1)$th action).

update, convergence of the model will not be invalidated.

As a result of this observation, multiple update strategies become valid for inexact search, including early update, which is subsumed in the violation-fixing framework. Alternatively, a *max-violation* update can be made to correct the worst search error (Huang et al., 2012; Watanabe and Sumita, 2015). Most recently, Lee et al. (2016) generalized max-violation to *all-violation* which corrects all search errors found for one input in aggregation using a single update, and they showed the efficacy of the new method in a neural chart-based CCG parser.

### 3.2.3 Training the Shift-Reduce Dependency Model

The normal-form shift-reduce model (Zhang and Clark, 2011a) uses early update (line 10, Fig. 3-6). For the dependency model, there can be multiple correct states (states resulting from valid actions) in the beam at each step. One option would be to apply early update whenever at least one of these states falls outside the beam. However, this may not be a true violation of the model. Thus, I use a relaxed version of early update, in which *all* correct states must fall outside the beam before an

```
$\mathbf{w} \leftarrow \mathbf{0}$; $\mathcal{B}_0 \leftarrow \emptyset$; $j \leftarrow 0$          ▷ input is $(x_i, \Delta_{x_i})$
$\mathcal{B}_0$.push($\langle \delta, \beta \rangle_0$)          ▷ the initial state
$cand \leftarrow \emptyset$          ▷ candidate output priority queue
$gold \leftarrow \emptyset$          ▷ gold output priority queue
while $\mathcal{B}_j \neq \emptyset$ do
  for each $\langle \delta, \beta \rangle \in \mathcal{B}_j$ do
    if $|\beta| = 0$ then          ▷ candidate output
      $cand$.push($\langle \delta, \beta \rangle$)
    if $\langle \delta, \beta \rangle \simeq \Delta_{x_i}$ then          ▷ a correct state
      $gold$.push($\langle \delta, \beta \rangle$)
    expand $\langle \delta, \beta \rangle$ into $\mathcal{B}_{j+1}$
  $\mathcal{B}_{j+1} \leftarrow \mathcal{B}_{j+1}[1 : k]$          ▷ apply beam of size $k$
  if $\Pi_G \neq \emptyset$, $\Pi_G \cap \mathcal{B}_{j+1} = \emptyset$ and $cand[0] \not\simeq \Delta_{x_i}$ then
    $\mathbf{w} \leftarrow \mathbf{w} + \phi(x_i, \Pi_G[0]) - \phi(x_i, \mathcal{B}_{j+1}[0])$          ▷ early update
    return
  $j \leftarrow j + 1$          ▷ continue to next step
if $cand[0] \not\simeq \Delta_{x_i}$ then          ▷ final update
  $\mathbf{w} \leftarrow \mathbf{w} + \phi(x_i, gold[0]) - \phi(x_i, cand[0])$
```

Figure 3-7: Dependency Model Training.

update is performed. This update mechanism generalizes early update, guarantees a local violation of the model (at the beam where search error has occurred), and is consistent with the violation-fixing framework (Huang et al., 2012).

Let $(x_i, \Delta_{x_i})$ be a training sentence paired with its gold standard dependency structure; let $\langle \delta, \beta \rangle$ be a parser state in the $j$th beam ($j \geq 0$) s.t. $\langle \delta, \beta \rangle \sim \Delta_{x_i}$, and let $\Pi_{\langle \delta, \beta \rangle}$ be the set

$$\left\{ \langle \delta, \beta \rangle \circ (x, c) \mid f_d(\langle \delta, \beta \rangle, (x, c), \Phi_{\Delta_{x_i}}) = true \right\},$$

which contains all *correct* states at the $(j+1)$th beam obtained by expanding $\langle \delta, \beta \rangle$. Further, let the set of *all* correct states at the $(j+1)$th beam be:[5]

$$\Pi_G = \bigcup_{\langle \delta, \beta \rangle \in \mathcal{B}_j} \Pi_{\langle \delta, \beta \rangle}.$$

Fig. 3-7 shows the pseudocode for training the dependency model with early

[5] In Fig. 3-7, I use $\Pi_G[0]$ to denote the highest scoring correct state in the set.

update for one input $(x_i, \Delta_{x_i})$. The score of a parser state $\langle \delta, \beta \rangle$ is calculated as $\mathbf{w} \cdot \phi(\langle \delta, \beta \rangle)$ with respect to the current model $\mathbf{w}$, where $\phi(\langle \delta, \beta \rangle)$ is the feature vector for the state. At the $j$th beam, all states are expanded and added on to the next beam $\mathcal{B}_{j+1}$, and the top-$k$ retained. Early update is applied when all correct states first fall outside the beam, and any candidate output is incorrect (line 13). Since there are potentially many correct states, and only one is required for the perceptron update, a decision needs to be made regarding which state to update against. Here, I choose to reward the highest scoring correct state, and penalize the highest scoring incorrect state. Finally, when no more expansions are possible but the final output is incorrect, an additional update is performed using the highest scoring correct and incorrect outputs (line 18).

## 3.3 Experiments

**Baselines and Setup.** Baselines are the normal-form shift-reduce model (Zhang and Clark, 2011a), the C&C normal-form and hybrid models (retrained using the SVN version of the toolkit), and a faithful reimplementation of the normal-form model used as an additional reference.

I used 10-fold jackknifing for POS tagging and supertagging the training data, and automatically assigned POS tags for all experiments. A probability cut-off value of $0.01 \times 10^{-2}$ for the $\beta$ parameter in the supertagger is used for both training and testing, which is a relatively small value that allows a large number of categories, compared to the default values used in Clark and Curran (2007). For training only, if the gold standard lexical category is not supplied by the supertagger for a particular word, it is added to the list of categories.

### 3.3.1 Features

The feature templates include all the derivation-based templates in Zhang and Clark (2011a) (reproduced in Table 3.2), and also all the CCG predicate-argument dependency templates from Clark and Curran (2007) (reproduced in Table 3.3). However, unlike Zhang and Clark (2011a), which uses CFG-like rule-based head selection, I use

| | | | |
|---|---|---|---|
| $s_0$.wp | $s_1$.wp | $s_2$.pc | $s_3$.pc |
| $s_0$.wc | $s_1$.wc | $s_2$.wc | $s_3$.wc |
| $s_0$.c$s_0$.pc | $s_1$.c$s_1$.pc | | |
| $q_0$.wp | $q_1$.wp | $q_2$.wp | $q_3$.wp |
| $s_0$.l.pc | $s_0$.r.wc | $s_1$.l.pc | $s_1$.r.wc |
| $s_0$.l.wc | $s_0$.u.pc | $s_1$.l.wc | $s_1$.u.pc |
| $s_0$.r.pc | $s_0$.u.wc | $s_1$.r.pc | $s_1$.u.wc |
| $s_0$.wc $\circ$ $s_1$.wc | $s_0$.c $\circ$ $s_1$.w | $s_1$.wc $\circ$ $q_0$.wp | |
| $s_0$.wc $\circ$ $q_0$.wp | $s_0$.c $\circ$ $s_1$.c | $s_1$.wc $\circ$ $q_0$.p | |
| $s_0$.wc $\circ$ $q_0$.p | $s_0$.c $\circ$ $q_0$.wp | $s_1$.c $\circ$ $q_0$.wp | |
| $s_0$.w $\circ$ $s_1$.c | $s_0$.c $\circ$ $q_0$.p | $s_1$.c $\circ$ $q_0$.p | |
| $s_0$.wc $\circ$ $s_1$.c $\circ$ $q_0$.p | $s_0$.c $\circ$ $q_0$.wp $\circ$ $q_1$.p | $s_0$.c $\circ$ $s_1$.c $\circ$ $s_2$.wc | |
| $s_0$.wc $\circ$ $s_1$.c $\circ$ $s_2$.c | $s_0$.c $\circ$ $q_0$.p $\circ$ $q_1$.wp | $s_0$.c $\circ$ $s_1$.c $\circ$ $s_2$.c | |
| $s_0$.wc $\circ$ $q_0$.p $\circ$ $q_1$.p | $s_0$.c $\circ$ $q_0$.p $\circ$ $q_1$.p | $s_0$.p $\circ$ $s_1$.p $\circ$ $s_2$.p | |
| $s_0$.c $\circ$ $s_1$.wc $\circ$ $q_0$.p | $s_0$.c $\circ$ $s_1$.c $\circ$ $q_0$.wp | $s_0$.p $\circ$ $s_1$.p $\circ$ $q_0$.p | |
| $s_0$.c $\circ$ $s_1$.wc $\circ$ $s_2$.c | $s_0$.c $\circ$ $s_1$.c $\circ$ $q_0$.p | $s_0$.p $\circ$ $q_0$.p $\circ$ $q_1$.p | |
| $s_0$.w $\circ$ $s_1$.c $\circ$ $s_1$.r.c | $s_0$.c $\circ$ $s_0$.r.c $\circ$ $q_0$.w | $s_0$.c $\circ$ $s_0$.h.c $\circ$ $s_0$.l.c | |
| $s_0$.c $\circ$ $s_0$.l.c $\circ$ $s_1$.w | $s_0$.c $\circ$ $s_0$.r.c $\circ$ $q_0$.p | $s_0$.c $\circ$ $s_1$.c $\circ$ $s_1$.r.c | |
| $s_0$.c $\circ$ $s_0$.l.c $\circ$ $s_1$.c | $s_0$.c $\circ$ $s_0$.h.c $\circ$ $s_0$.r.c | $s_1$.c $\circ$ $s_1$.h.c $\circ$ $s_1$.r.c | |

Table 3.2: Feature templates of the normal-form shift-reduce model reproduced from Zhang and Clark (2011a). $\circ$ denotes feature conjunction.

a CCG-style head passing mechanism (Clark and Curran, 2007). Concretely, each template captures various aspects of the stack and queue context, and is defined as a pair $(f\langle\delta, \beta\rangle, \mathit{action})$, consisting of a template $f$, to be instantiated from a parser state $\langle\delta, \beta\rangle$ and a shift-reduce *action*. Applying the definitions from Zhang and Clark (2011a) to Table 3.2, l points to the left child of a binary node $s$ (i.e., a node with two children), if the head is from the right node, and symmetrically for r; u points to the left child of a unary node $s$ (i.e., a node with a single child); and h points to the child node of a binary node that has the head. It is also always the case that the head selection rules used by Zhang and Clark (2011a) will return a single head from either the left or the right child of a binary node $s$, and no two symmetric templates will activate together for the same $s$. In my model, this is not the case; for example, when a node contains more than one head (e.g., through coordination rules), each of these heads (from either the left or the right child node) is instantiated as a feature. In any

| Feature type | Example |
|---|---|
| Word–Word | $\langle \textit{bought}, (S\backslash NP_1)/NP_2, 2, \textit{company}, (NP\backslash NP)/(S[dcl]/NP)\rangle$ |
| Word–POS | $\langle \textit{bought}, (S\backslash NP_1)/NP_2, 2, \text{NN}, (NP\backslash NP)/(S[dcl]/NP)\rangle$ |
| POS–Word | $\langle \text{VBD}, (S\backslash NP_1)/NP_2, 2, \textit{company}, (NP\backslash NP)/(S[dcl]/NP)\rangle$ |
| POS–POS | $\langle \text{VBD}, (S\backslash NP_1)/NP_2, 2, \text{NN}, (NP\backslash NP)/(S[dcl]/NP)\rangle$ |
| Word + Dist. (words) | $\langle \textit{bought}, (S\backslash NP_1)/NP_2, 2, (NP\backslash NP)/(S[dcl]/NP)\rangle + 2$ |
| Word + Dist. (punct) | $\langle \textit{bought}, (S\backslash NP_1)/NP_2, 2, (NP\backslash NP)/(S[dcl]/NP)\rangle + 0$ |
| Word + Dist. (verbs) | $\langle \textit{bought}, (S\backslash NP_1)/NP_2, 2, (NP\backslash NP)/(S[dcl]/NP)\rangle + 0$ |
| POS + Dist. (words) | $\langle \text{VBD}, (S\backslash NP_1)/NP_2, 2, (NP\backslash NP)/(S[dcl]/NP)\rangle + 2$ |
| POS + Dist. (punct) | $\langle \text{VBD}, (S\backslash NP_1)/NP_2, 2, (NP\backslash NP)/(S[dcl]/NP)\rangle + 0$ |
| POS + Dist. (verbs) | $\langle \text{VBD}, (S\backslash NP_1)/NP_2, 2, (NP\backslash NP)/(S[dcl]/NP)\rangle + 0$ |

Table 3.3: Feature templates of the chart-based dependency model reproduced from Clark and Curran (2007). The last field in each dependency 5-tuple indicates the category that mediates a long-range dependency (Clark and Curran, 2007, p. 507), which is omitted in the definition in §2.1.4.

case, w, c and p represent the head word, CCG category and POS tag, respectively.

In addition, the CCG dependency features contribute to the score of a REDUCE action if one or more dependencies are realized by that action. As shown in Table 3.3, the dependency features are defined over CCG predicate-argument dependencies and represent a system of back-off features from words to POS tags. All these features are also conjoined with three types of distance measure which count the number of intervening words, the number of intervening punctuation marks, and the number of intervening verbs (as determined by POS tags). The first two types of distance measure have four possible values 0, 1, 2, or more and the third one has three possible values 0, 1, or more.

### 3.3.2 Results

To estimate the parameters of the model, I tuned on the dev set, and found the accuracy converged after the 25th epoch, with the resulting model containing 16.50M features with a non-zero weight. For training, a beam size of 16 was used; for all other experiments, a beam size of 128, determined on the dev set, was used.

Table 3.4 shows dev set results. Compared with C&C, the dependency model (SHIFT-REDUCE-DEP) shows significant gains across all metrics, improving F1 over the normal-form and hybrid models by 1.35% and 0.79%, and it also achieves a more

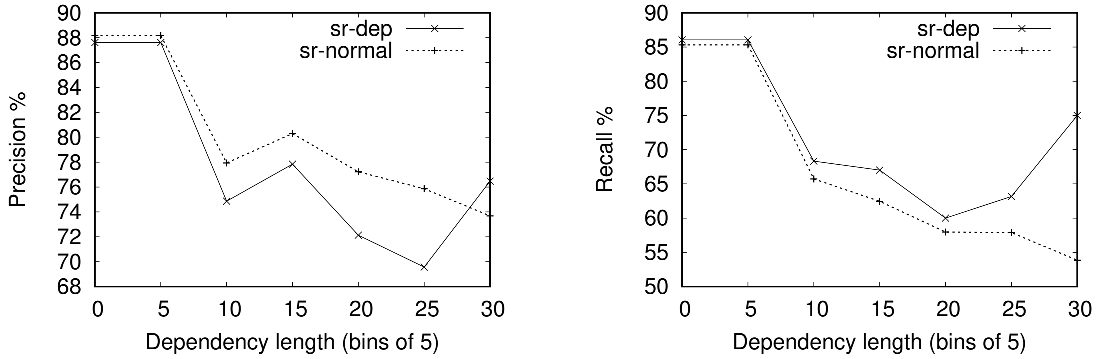


Figure 3-8: Labeled precision and recall relative to dependency length on the dev set.

| | beam | LP | LR | LF | SENT | CAT |
|---|---|---|---|---|---|---|
| C&C (normal) | - | 85.18 | 82.53 | 83.83 | 31.42 | 92.39 |
| C&C (hybrid) | - | 86.07 | 82.77 | 84.39 | 32.62 | 92.57 |
| Zhang and Clark (2011a) | 16 | **87.15** | 82.95 | 85.00 | 33.82 | **92.77** |
| Zhang and Clark (2011a)* | 16 | 86.76 | 83.15 | 84.92 | 33.72 | 92.64 |
| Shift-Reduce-Dep | 128 | 86.29 | **84.09** | **85.18** | **34.40** | 92.75 |
| Shift-Reduce-Dep$-\Delta_f$ | 128 | 85.67 | 83.36 | 84.50 | 33.30 | 92.40 |

Table 3.4: Parsing results on Section 00 (100% coverage and auto POS). $-\Delta_f$ (without dependency-based features); $*$ = reimplementation.

balanced precision and recall over the normal-form shift-reduce model.

Table 3.4 also shows that the dependency model improved recall over the normal-form shift-reduce model at some expense of precision. To probe this further, I evaluated the models over labeled precision and recall relative to different dependency lengths, as measured by the distance between the two words in a dependency, grouped into bins of 5 values (Fig. 3-8). The results show that the normal-form model favors precision over recall, giving higher precision scores for almost all dependency lengths. In terms of recall, the dependency model outperforms the normal-form model over all dependency lengths, especially for longer dependencies ($x \geq 20$).

As an ablation experiment, I experimented with ablating the dependency features, and found they contributed +0.68 F1 to the final model (Shift-Reduce-Dep$-\Delta_f$, Table 3.4). One motivation behind the dependency features is that all shift-reduce action sequences producing the same CCG dependencies would produce the same set

|  | beam | LP | LR | LF | SENT | CAT | Speed |
|---|---|---|---|---|---|---|---|
| C&C (normal) | - | 85.58 | 82.85 | 84.20 | 32.90 | 92.84 | 97.90 |
| C&C (hybrid) | - | 86.24 | 84.17 | 85.19 | 33.24 | 93.00 | 95.25 |
| Zhang and Clark (2011a) | 16 | **87.43** | 83.61 | 85.48 | 35.19 | **93.12** | - |
| Zhang and Clark (2011a)* | 16 | 87.04 | 84.14 | 85.56 | 34.98 | 92.95 | 49.54 |
| SHIFT-REDUCE-DEP | 128 | 87.03 | **85.08** | **86.04** | **35.69** | 93.10 | 12.85 |

Table 3.5: Parsing results on Section 23 (100% coverage and auto POS). $*$ = reimplementation.

of dependency features. Thus, such features are in a sense more applicable to spurious derivations, in comparison with the derivation-based ones. As also shown in Clark and Curran (2007), this result demonstrates that the combination of derivation and dependency features are essential for the dependency model.

As the final development experiment, dependency recovery accuracy for the most frequent dependency relations were evaluated. As shown in Table 3.6, the dependency model gives higher recall for all but one of the relations, and higher F1 for over half of them, showing that it is more balanced.

Table 3.5 presents the final results on Section 23. Again, the dependency model achieves the highest scores across all metrics, except for precision and lexical category assignment accuracy.

Finally, in terms of speed, the dependency model underperforms the normal-form model.[6] However, a further optimized implementation, especially in regards to feature look-up (Bohnet, 2010), should provide additional speed improvements, due to the rich feature set used. Nevertheless, experiments suggest that parsing efficiency is mainly related to the beam size that scales the runtime by a constant factor. Therefore, incorporating further optimizations (Goldberg et al., 2013), in addition to adopting Tree Structure Stack (§4.3.5), is likely to provide some additional benefits.

## 3.4 Summary

The first shift-reduce dependency model is presented for CCG, motivated by the advantages in modelling dependencies for it. One key strength of the model, over the

[6] Speed measured with an Intel i7-4790K CPU.

| category | slot | LP (d) | LP (z) | LP (c) | LR (d) | LR (z) | LR (c) | LF (d) | LF (z) | LF (c) | freq. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $N/N$ | 1 | 95.53 | **95.77** | 95.28 | **95.83** | 95.79 | 95.62 | 95.68 | **95.78** | 95.45 | 7288 |
| $NP/N$ | 1 | 96.53 | **96.70** | 96.57 | **97.12** | 96.59 | 96.03 | **96.83** | 96.65 | 96.30 | 4101 |
| $(NP\backslash NP)/NP$ | 2 | 81.64 | **83.19** | 82.17 | **90.63** | 89.24 | 88.90 | 85.90 | **86.11** | 85.40 | 2379 |
| $(NP\backslash NP)/NP$ | 1 | 81.70 | **82.53** | 81.58 | **88.91** | 87.99 | 85.74 | 85.15 | **85.17** | 83.61 | 2174 |
| $((S\backslash NP)\backslash(S\backslash NP))/NP$ | 3 | **77.64** | 77.60 | 71.94 | 72.97 | 71.58 | **73.32** | **75.24** | 74.47 | 72.63 | 1147 |
| $((S\backslash NP)\backslash(S\backslash NP))/NP$ | 2 | 75.78 | **76.30** | 70.92 | 71.27 | 70.60 | **71.93** | **73.45** | 73.34 | 71.42 | 1058 |
| $((S[dcl]\backslash NP)/NP$ | 2 | 83.94 | **85.60** | 81.57 | 86.04 | 84.30 | **86.37** | **84.98** | 84.95 | 83.90 | 917 |
| $PP/N$ | 1 | **77.06** | 73.76 | 75.06 | **73.63** | 72.83 | 70.09 | **75.31** | 73.29 | 72.49 | 876 |
| $((S[dcl]\backslash NP)/NP$ | 1 | 82.03 | **85.32** | 81.62 | 83.26 | 82.00 | **85.55** | 82.64 | **83.63** | 83.54 | 872 |
| $((S\backslash NP)\backslash(S\backslash NP))$ | 2 | 86.42 | 84.44 | **86.85** | 86.19 | 86.60 | **86.73** | 86.31 | 85.51 | **86.79** | 746 |

Table 3.6: Accuracy comparison on most frequent dependency types, for the dependency model (d), the Zhang and Clark (2011a) normal-form shift-reduce model (z), and the C&C hybrid model (c).

dependency model of Clark and Curran (2007), is it fully aligns with the left-to-right, incremental nature of shift-reduce parsing.

In a broader context, a closely related work is Yu et al. (2013), which adapted the violation-fixing perceptron to deal with latent derivations in an MT model. Like the present work, they also show that more principled integration of structured learning and inexact search is a simple and powerful mechanism for structured prediction with the structured perceptron.

# Chapter 4

# Expected F-measure Training for Shift-Reduce Parsing with Recurrent Neural Networks

In this chapter, I first take a detour from shift-reduce parsing and introduce a supertagging model based on a simple Elman recurrent neural network (RNN; Elman, 1990). This is the first neural network model in this thesis, which also serves as the basis for all supertagging models introduced later. As a proof-of-concept use of this model, I show it leads to direct parsing accuracy gains for all the CCG parsing models of Clark and Curran (2007).

Supertagging was first introduced for CCG by Clark (2002) and the model in Clark and Curran (2007) became the most widely adopted, along with the C&C parser (Curran et al., 2007). However, despite its proven efficacy, the MaxEnt supertagging model of Clark and Curran (2007) has a number of drawbacks. First, it relies too heavily on POS tags, which leads to lower accuracy on out-of-domain data (Rimell and Clark, 2008). Second, due to the sparse, indicator feature sets mainly based on raw words and POS tags, it shows pronounced performance degradation in the presence of rare and unseen words (Rimell and Clark, 2008; Lewis and Steedman, 2014b). And third, in order to reduce computational requirements and feature sparsity, each tagging decision is made without considering any potentially useful contextual information beyond a local context window.

Lewis and Steedman (2014b) introduced a feed-forward neural network to supertagging, and addressed the first two problems mentioned above. However, their attempt to tackle the third problem by pairing a conditional random field (CRF; Lafferty et al., 2001) with their feed-forward tagger provided little accuracy improvement and vastly increased computational complexity, incurring a large efficiency penalty.

Here I present an RNN model for supertagging to tackle all the above problems, with an emphasis on the third one. RNNs are powerful models for sequential data, which can potentially capture long-term dependencies, based on an *unbounded* history of previous words (§4.1.1). I show that as a standalone model, the RNN supertagger outperforms the feed-forward setup, and by integrating it with the C&C parser as its adaptive supertagger, at both training and inference time, I obtain substantial accuracy improvements on both supertagging and parsing.

Next, as the main contribution of this chapter, I present expected F-measure training for shift-reduce parsing with RNNs. Specifically, I show how to derive a globally normalized model optimized for the final evaluation metric, in which beam search is naturally incorporated during training and used in conjunction with the objective to learn shift-reduce action sequences that lead to parses with high expected F-scores. Recently, RNNs have been used to learn explicit representations for parser states as well as actions performed on the stack and queue in shift-reduce parsers (Dyer et al., 2015; Watanabe and Sumita, 2015). In comparison, both my locally normalized baseline and the global model are based on a natural extension of the Chen and Manning (2014) feed-forward model.

I apply the models to CCG, and by combining the global RNN parsing model with a bidirectional RNN CCG supertagger (§4.5) extending the basic RNN supertagging model, the final parser achieves accuracies higher than both the normal-form (Zhang and Clark, 2011a) and the dependency (§3) shift-reduce models.

As stated in §2.3.2, it is convenient to assume a shift-reduce parser is made up of three interrelated components, namely, a deduction system, a model and a search strategy. In a neural network-based shift-reduce parser, the model component is typically factored as one or more neural networks, which are often used to learn representations for various aspects of parser states. In Chen and Manning (2014), who demonstrate the first successful application of neural networks to transition-based dependency parsing, they target learning representations for higher-order features using a small set of dense atomic feature templates and a feed-forward neural network. They show the resulting model lends itself well to alleviating the drawbacks of one-hot indicator features, which are sparsity, incompleteness and expensive feature computation (Chen and Manning, 2014, §2). However, despite achieving then state-of-the-art greedy dependency parsing results, the Chen and Manning (2014) model only models isolated shift-reduce actions and is susceptible of the label bias problem (Bottou, 1991; Lafferty et al., 2001; Andor et al., 2016). In contrast, my global RNN model is not only able to take advantage of the RNN by relying on it for the automatic discovery of higher-order features, it also models shift-reduce actions as structured sequences rather than in isolation, overcoming the label bias problem.

Although the experiments below are focused on CCG, it is worth noting that the expected F-measure training framework proposed is general enough to be adapted to other tasks, in particular, to parsing with formalisms other than CCG. Moreover, because the framework is agnostic of the underlying neural network, as demonstrated in the next chapter, other kinds of neural networks with varying architectural configurations can be used.

## 4.1 Background

### 4.1.1 The Elman Recurrent Neural Network

An Elman RNN consists of an input layer $\mathbf{x}_t$, a hidden state (layer) $\mathbf{h}_t$ with a recurrent connection to the previous hidden state $\mathbf{h}_{t-1}$ and an output layer $\mathbf{y}_t$. In the supertagging models, the input layer is a vector representing the surrounding context of the current word at position $t$, whose supertag is being predicted (§4.2); in the parsing models, the input is a concatenation of atomic feature embeddings (Chen and

Manning, 2014) (§4.3).[1] The hidden state $\mathbf{h}_{t-1}$ keeps a representation of all context history, and the current hidden state $\mathbf{h}_t$ is computed using the current input $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$. The output layer represents probability scores of all possible supertags (§4.2) or shift-reduce actions (§4.3).

The parameterization of the network consists of three matrices which are learned during supervised training. Matrix $\mathbf{U}$ contains weights between the input and hidden layers, $\mathbf{V}$ contains weights between the hidden and output layers, and $\mathbf{W}$ contains weights between the previous hidden state and the current hidden state. To compute the hidden state activations at time step $t$, the following recurrence is used:[2]

$$\mathbf{h}_t = f(\mathbf{U}\mathbf{x}_t + \mathbf{W}\mathbf{h}_{t-1}),$$

where f is the sigmoid function $f(h_i) = \frac{1}{1+e^{-h_i}}$.[3] To calculate the output activations $\mathbf{y}_t$, $\mathbf{h}_t$ is linearly transformed using $\mathbf{V}$, and finally the softmax activation function $g(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$ is applied s.t.

$$\mathbf{z}_t = g(\mathbf{y}_t).$$

### 4.1.2 Backpropagation

All the RNN supertagging models and the baseline RNN shift-reduce parsing model are trained using the cross-entropy loss defined, at each time step, as

$$J(\theta) = -\sum_{i}^{C} t_i \log z_i,$$

where $C$ is the total number of classes; $1 \leq i \leq C$; $t_i$ are the gold standard class probabilities, and $z_i$ are the softmax activations. Given this loss, the derivative of the loss w.r.t. an output $y_i$ can be derived analytically without assuming a specific

[1] This is different from some RNN models (e.g., Mikolov et al. (2010)) where the input is a one-hot vector.

[2] All vectors are assumed to be column vectors unless otherwise stated.

[3] Other non-linear functions such as tanh can be used instead, but negligible improvements were observed in my experiments.

neural network architecture.

First, I compute

$$\frac{\partial J(\theta)}{\partial z_i} = -\frac{t_i}{z_i}, \tag{4.1}$$

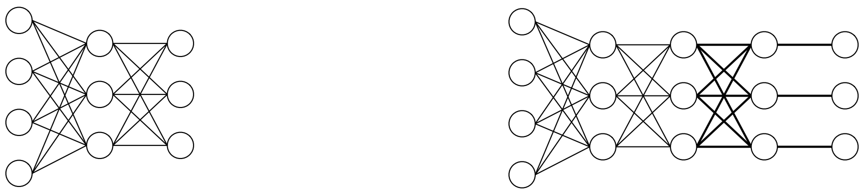
which is the derivative of the loss w.r.t to the $i$th output of the softmax layer. Next, I derive $\frac{\partial z_k}{\partial y_i}$, which is the derivative of the $k$th softmax output ($1 \leq k \leq C$) w.r.t. the $i$th output $y_i$, computed as

$$\begin{cases} \frac{\partial z_k}{\partial y_i} = \frac{\partial \frac{e^{y_k}}{\sum_{j=1}^{C} e^{y_j}}}{\partial y_i} = \frac{e^{y_k}\sum_{j=1}^{C} e^{y_j} - e^{y_k}e^{y_i}}{(\sum_{j=1}^{C} e^{y_j})^2} = z_k - z_k z_i = z_k(1 - z_i) & \text{if } i = k \\ \frac{\partial z_k}{\partial y_i} = \frac{\partial \frac{e^{y_k}}{\sum_{j=1}^{C} e^{y_j}}}{\partial y_i} = \frac{0 - e^{y_k}e^{y_i}}{(\sum_{j=1}^{C} e^{y_j})^2} = -\frac{e^{y_k}}{\sum_{j=1}^{C} e^{y_j}}\frac{e^{y_i}}{\sum_{j=1}^{C} e^{y_j}} = -z_k z_i & \text{if } i \neq k. \end{cases} \tag{4.2}$$

The required derivative is then:

$$\begin{aligned} \frac{\partial J(\theta)}{\partial y_i} &= -\sum_{k=1}^{C} \frac{\partial t_k \log z_k}{\partial y_i} \\ &= -\sum_{k=1}^{C} \frac{t_k \partial \log z_k}{\partial y_i} \\ &= -\sum_{k=1}^{C} t_k \frac{1}{z_k}\frac{\partial z_k}{\partial y_i} \\ &= \frac{-t_i}{z_i}\underbrace{z_i(1 - z_i)}_{\text{Eq. 4.2}} - \sum_{k \neq i}^{C} t_k \frac{1}{z_k}\underbrace{(-z_i z_k)}_{\text{Eq. 4.2}} \\ &= z_i(\sum_{k \neq i}^{C} t_k + t_i) - t_i \\ &= z_i - t_i. \end{aligned}$$

Following through, if a specific neural network was given, I could continue to derive all other derivatives analytically down to the hidden and input layers, and this process is referred to as backpropagation (Werbos, 1974; Rumelhart et al., 1988). However, it is more illuminating to see backpropagation (BP) in a vectorized form using only a series of matrix multiplications. It can be shown this form of BP gives identical gradients to those derived analytically. Below, I first present vectorized BP as a mechanical



(a) (b)

Figure 4-1: A three-layered feed-forward network (a), extended with a softmax layer and a cross-entropy loss layer with unity weight connections (b).

procedure using a simple three-layered feed-forward network, then I discuss back-propagation through structure (BPTS; Goller and Kuchler, 1996), which generalizes BP from fixed network topologies to arbitrary directed acyclic graphs (DAGs); BP applied to feed-forward, recursive, and recurrent networks are all specific instances of BPTS. BPTS can also be realized in a vectorized fashion, as shown in §4.1.4, §4.3.6, and again in §5.1.2.

Assuming a three-layered feed-forward network, with an input layer size $m$, and hidden and output layer size $n$ (e.g., in Fig. 4-1a, $m = 4$ and $n = 3$). For ease of exposition, I first assume that the derivatives w.r.t. the loss at the output layer, denoted as $\boldsymbol{\delta}_y$, are given, s.t.

$$\boldsymbol{\delta}_y = \begin{pmatrix} z_1 - t_1 \\ z_2 - t_2 \\ \vdots \\ z_n - t_n \end{pmatrix};$$

I also assume that sigmoid activation functions are used at the hidden layer, and let $\mathbf{h}$ be the vector of activations and $\mathbf{h}'$ be its derivative, denoted as

$$\mathbf{h}' = \begin{pmatrix} h_1(1 - h_1) \\ h_2(1 - h_2) \\ \vdots \\ h_n(1 - h_n) \end{pmatrix}.$$

In addition, notate the weight matrix between the input layer and the hidden layers as $\mathbf{M}_1$ and the weight matrix between the hidden and output layers as $\mathbf{M}_2$. As the first step, vectorized BP entails computing the *errors* backpropagated to the hidden layer:

$$\boldsymbol{\delta}_h = \mathbf{h}' \circ (\mathbf{M}_2 \boldsymbol{\delta}_y),$$

where $\circ$ denotes element-wise product. Then, the gradients of $\mathbf{M}_2$ can be obtained as

$$\nabla_{\mathbf{M}_2} = \mathbf{h}\boldsymbol{\delta}_y^\top;$$

similarly, the gradients of $\mathbf{M}_1$ can be obtained as

$$\nabla_{\mathbf{M}_1} = \mathbf{x}\boldsymbol{\delta}_h^\top,$$

where $\mathbf{x}$ is the input vector. Finally, gradient descent steps can then be taken as $-\gamma\nabla_{\mathbf{M}_1}$ and $-\gamma\nabla_{\mathbf{M}_2}$, where $\gamma$ is the learning rate.

In general, this procedure can be used for a feed-forward network with an arbitrary number of layers. As above, the first step involves the recursive computation:

$$\boldsymbol{\delta}_l = \mathbf{d}_l' \circ (\mathbf{M}_l \boldsymbol{\delta}_{l+1}), \quad \text{for} \;\; l = 1 \dots L-1, \tag{4.3}$$

where $\mathbf{d}_l'$ is the activation derivative vector at the $l$th layer; $\mathbf{M}_l$ is the weight matrix between the $l$th and $(l+1)$th layer; $\boldsymbol{\delta}_{l+1}$ is the error vector at the $(l+1)$th layer; and $L$ is the total number of network layers up to and including the output layer ($L = 3$ in Fig. 4-1a). Next, once all the error vectors are obtained, the gradients can be computed as

$$\nabla_{\mathbf{M}_l} = \mathbf{a}_l \boldsymbol{\delta}_{l+1}^\top,$$

where $\mathbf{a}_l$ is the input or activation vector at the $l$th layer.

### 4.1.3 Backpropagation Through Structure

In actuality, the above vectorized form of BP is BPTS applied to a feed-forward network. The key insight is that the input $\pi_i$ to a given weight $m_{ij}$ within a weight matrix $\mathbf{M}$ is constant in one feed-forward pass, and therefore the derivative of this weight w.r.t. the loss $J$ is

$$\frac{\partial J}{\partial m_{ij}} = \pi_i \frac{\partial J}{\partial \pi_i m_{ij}}.$$

By using the chain rule, it can be shown that $\frac{\partial J}{\partial \pi_i m_{ij}}$ is the cumulative backpropagated error up to the node to the right of weight $m_{ij}$. Denote this cumulative error as $\boldsymbol{\delta}_j$, the derivative can then be expressed as

$$\frac{\partial J}{\partial m_{ij}} = \pi_i \boldsymbol{\delta}_j.$$

In essence, BP and BPTS are procedures for computing cumulative backpropagated errors, such as $\boldsymbol{\delta}_j$, for all the nodes in a network. Equivalently, BP entails running a network backwards with the errors obtained from the feed-forward step as input.

In fact, BPTS can also be used to compute the derivatives w.r.t. the loss at the output layer, whereas in the above these derivatives are assumed to be given. To do this, a network can be extended with a softmax layer that is connected to a cross-entropy loss layer, and BPTS can be applied to compute the derivatives using Eq. 4.1 and Eq. 4.2, by assuming all the weights between the output, softmax, and the cross-entropy layers are constant and unity. The extended network for Fig. 4-1a is shown in Fig. 4-1b.

### 4.1.4 Backpropagation Through Time

The most common BP algorithm used to train an RNN is backpropagation through time (BPTT; Werbos, 1990), which is BPTS applied to an unrolled RNN, and it is used for training all models using the cross-entropy loss in this chapter.

Assuming that the derivatives at the output layers were given and the RNN is unrolled for three time steps (Fig. 4-2). First, I compute the errors at the hidden state $\mathbf{h}_t$ as (by Eq. 4.3):

$$\boldsymbol{\delta}_{h_t} = \mathbf{h}'_t \circ (\mathbf{V}\boldsymbol{\delta}_{y_t}),$$

where $\boldsymbol{\delta}_{y_t}$ is the error vector at the output layer of step $t$ and $\mathbf{h}'_t$ is the vector of hidden

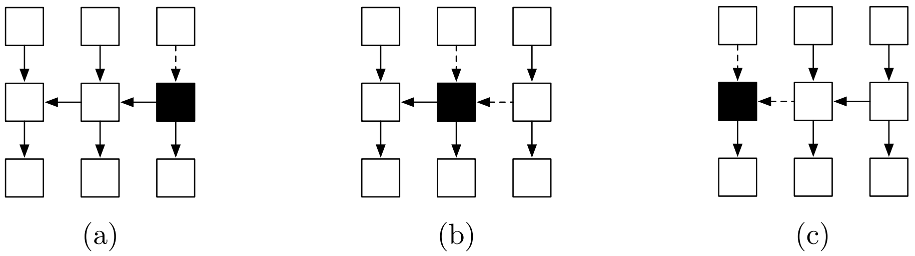


Figure 4-2: RNN BPTT. Errors flow back to $\mathbf{h}_t$ (a), $\mathbf{h}_{t-1}$ (b), and $\mathbf{h}_{t-2}$ (c).

state activation derivatives. I then compute the errors at hidden state $\mathbf{h}_{t-1}$:

$$\boldsymbol{\delta}_{h_{t-1}} = \mathbf{h}'_{t-1} \circ (\mathbf{W}\boldsymbol{\delta}_{h_t} + \mathbf{V}\boldsymbol{\delta}_{y_{t-1}});$$

similarly, the errors at hidden state $\mathbf{h}_{t-2}$ can be computed as

$$\boldsymbol{\delta}_{h_{t-2}} = \mathbf{h}'_{t-2} \circ (\mathbf{W}\boldsymbol{\delta}_{h_{t-1}} + \mathbf{V}\boldsymbol{\delta}_{y_{t-2}}).$$

After all the errors at the hidden states are found, the errors at the input layers can be computed in any order.

Because manual implementation of backpropagation is prone to error, even in the vectorized form. It is common practice to numerically verify the correctness of an implementation using two-sided approximations of the gradients. This verification is implemented in all models of this chapter implemented in plain C++ using the Armadillo matrix library (Sanderson, 2010).

## 4.2 RNN Supertagging

The RNN supertagger only uses continuous vector representations for features and each feature type has an associated look-up table, which maps a feature to its distributed representation. In total, three feature types are used. The first type is word embeddings: given a sentence of $N$ words, $w_1, w_2, \ldots, w_N$, the embedding feature of $w_t$ (for $1 \leq t \leq N$) is obtained by projecting it onto a $n$-dimensional vector space through the look-up table $\mathcal{L}_w \in \mathbb{R}^{n \times |w|}$, where $|w|$ is vocabulary size. Algebraically, the projection operation is a simple matrix-vector product where a one-hot vector

$b_j \in \mathbb{R}^{|w| \times 1}$ is multiplied with $\mathcal{L}_w$ s.t.

$$e_{w_t} = \mathcal{L}_w b_j \in \mathbb{R}^{1 \times n},$$

where $j$ is the look-up index for $w_t$.

In addition, as in Lewis and Steedman (2014b), for every word I also include its 2-character suffix and capitalization as features. Two more look-up tables are used for these features. $\mathcal{L}_s \in \mathbb{R}^{m \times |s|}$ is the look-up table for suffix embeddings, where $|s|$ is the suffix vocabulary size; $\mathcal{L}_c \in \mathbb{R}^{m \times 2}$ is the look-up table for the capitalization embeddings; and $\mathcal{L}_c$ contains only two embeddings, representing whether or not a given word is capitalized.

I extract features from a context window surrounding the target word to make a tagging decision. Concretely, with a context window of size $k$, $\lfloor k/2 \rfloor$ words either side of the target word are included, giving a continuous feature representation

$$f_{w_t} = [e_{w_t}; s_{w_t}; c_{w_t}],$$

where $e_{w_t}$, $s_{w_t}$ and $c_{w_t}$ are the output vectors from the three different look-up tables, and ";" denotes vertical concatenation s.t. $f_{w_t} \in \mathbb{R}^{(n+2m) \times 1}$. At word position $t$, the input layer of the network $x_t$ is

$$x_t = [f_{w_{t-\lfloor k/2 \rfloor}}; \ldots f_{w_t}; \ldots; f_{w_{t+\lfloor k/2 \rfloor}}],$$

where $x_t \in \mathbb{R}^{k(n+2m) \times 1}$ and the right-hand side is the concatenation of all feature representations in a size $k$ context window.

## 4.3 RNN Shift-Reduce Parsing

In this section, I start by describing the baseline model, which is also taken as the pretrained model to train the global model (§4.3.3). I abstract away from the details of CCG and present the models in a canonical shift-reduce framework (Aho and Ullman, 1972), which is henceforth assumed: partially constructed derivations are maintained on a *stack*, and a *queue* stores remaining words from the input string; the initial parser state has an empty stack and no input has been consumed on the queue. Parsing proceeds by applying a sequence of shift-reduce actions to transform the input until the queue has been exhausted and no more actions can be applied.

### 4.3.1 Model and Feature Embeddings

Similar to Chen and Manning (2014), the input layer $x_t$ encodes stack and queue contexts of a parser state through concatenation of feature embeddings. The output layer $y_t$ represents a probability distribution over the feasible shift-reduce actions for the current state.

Given a parser state, I first extract *atomic* features using a set of predefined feature templates (Table 4.1, §4.4). The features are atomic in the sense that they are a set of first-order features from which all higher-order features are automatically induced. The simplicity of these atomic features reduces feature engineering efforts in comparison with the structured perceptron model (Table 3.2, §3), and it is one of the main motivations of the Chen and Manning (2014) model.

Similar to the RNN supertagging model, each feature template in the parsing model belongs to a feature type $f$ (such as word or POS tag), which has an associated look-up table, denoted as $L_f$, to project a feature to its distributed representation. The embedding for a concrete feature is obtained by retrieving the corresponding column from $L_f$. At time step $t$, the input layer $x_t$ is:

$$x_t = [e_{f_{1,1}}; \dots ; e_{f_{1,|f_1|}}; \dots ; e_{f_{k,1}}; \dots ; e_{f_{k,|f_k|}}],$$

where ";" denotes vertical concatenation, $f_{1,i}$ denotes the $i$th template of the $k$th feature type, and $|f_k|$ is the total number of feature templates for the $k$th feature type s.t. $1 \leq i \leq |f_k|$. In addition, for each feature type, a special embedding is used for unknown features.

### 4.3.2 The Label Bias Problem: Local vs. Global Normalization

The baseline parser in this chapter is a locally-normalized RNN model. That is, softmax normalization is computed locally at each step and the log-likelihood of the gold standard shift-reduce action is maximized with a cross-entropy loss.

Specifically, let $y$ be a shift-reduce action sequence, and $y_t$ be the $t$th action in $y$, for $1 \leq t \leq |y|$. The probability of $y_t$ conditioned on the parser state $\langle \alpha, \beta \rangle_y^{t-1}$ is computed as

$$p(y_t | \langle \alpha, \beta \rangle_y^{t-1}; \theta) = \frac{\exp\{\gamma(y_t, \langle \alpha, \beta \rangle_y^{t-1}; \theta)\}}{Z_\theta\left(\langle \alpha, \beta \rangle_y^{t-1}\right)},$$

where $\gamma$ is a scoring function that computes the score of an action, conditioned on the parser state $\langle \alpha, \beta \rangle_y^{t-1}$ given the model parameters $\theta$, and

$$Z_\theta(\langle \alpha, \beta \rangle_y^{t-1}) = \sum_{y_t{}' \in \mathcal{T}\left(\langle \alpha, \beta \rangle_y^{t-1}\right)} \exp\{\gamma(y_t{}', \langle \alpha, \beta \rangle_y^{t-1}; \theta)\}$$

is a local normalization term for the parser state $\langle \alpha, \beta \rangle_y^{t-1}$ over the set of feasible actions $\mathcal{T}\left(\langle \alpha, \beta \rangle_y^{t-1}\right)$, at step $t$. During inference, the probability of a sequence of actions can be obtained as

$$\begin{aligned} p(y|\theta) &= \prod_{t=1}^{|y|} p(y_t | (\langle \alpha, \beta \rangle_y^{t-1}); \theta) \\ &= \frac{\exp\{\sum_{t=1}^{|y|} \gamma(y_t, \langle \alpha, \beta \rangle_y^{t-1}; \theta)\}}{\prod_{t=1}^{|y|} Z_\theta(\langle \alpha, \beta \rangle_y^{t-1})}, \end{aligned}$$

and either greedy or beam search can be used.

The critical drawback of locally normalized models is that they are susceptible of the label bias problem (Bottou, 1991; Lafferty et al., 2001), which is a well-understood phenomenon of local normalization in the context of maximum-entropy Markov models (MEMMs; McCallum et al., 2000). MEMMs model conditional probabilities of next states given the current state using local normalization, which results in biases being given to states with fewer outgoing states. CRFs (Lafferty et al., 2001) are designed to tackle this, in which the per-state local normalization is replaced with global normalization of the joint probability of label sequences.

The label bias problem in the context of neural networks for transition-based modelling, including shift-reduce parsing, is further elaborated by Andor et al. (2016), who showed that a locally normalized model is strictly less expressive than its global counterpart with a CRF loss by extending the proof in Smith and Johnson (2007). In particular, they proved that for models defining conditional distributions of output decision sequences given input observation sequences, the set of all possible distributions $\mathcal{P}_L$ in a local model is a strict subset of the set of all possible distributions $\mathcal{P}_G$ in a global model.[4]

For shift-reduce parsing, a CRF loss defines the probability of a sequence of shift-reduce actions as

$$\hat{p}(y|\theta) = \frac{\exp\{\sum_{t=1}^{|y|} \gamma(y_t, \langle \alpha, \beta \rangle_y^{t-1}; \theta)\}}{\hat{Z}_\theta},$$

where

$$\hat{Z}_\theta = \sum_{y' \in \mathcal{S}_{|y|}} \exp \sum_{t=1}^{|y|} \gamma(y'_t, \langle \alpha, \beta \rangle_{y'}^{t-1}; \theta)$$

is a global normalization term over all action possible action sequences $\mathcal{S}$ of length $|y|$. Under this model, the inference problem is

$$y^* = \underset{y' \in \mathcal{S}_{|y|}}{\arg\max} \sum_{t=1}^{|y|} \gamma(y'_t, \langle \alpha, \beta \rangle_{y'}^{t-1}; \theta).$$

Again, either greedy or beam search can be used to find an inexact solution.

Unfortunately, although I use an RNN, which keeps a representation of previous parser states in its hidden state and has the potential to capture long-term dependencies, the locally normalized model still suffers from the label bias problem. This is also clear from the proof in Andor et al. (2016) which holds irrespective of the scoring function used to obtain the raw scores before local normalization is applied.[4] This indicates the importance of global normalization in addition to improving the

[4] In the proof, a tagging problem is considered with the restriction that at each time step $t$, the prediction depends only on the first $t$ input symbols; hence the model does not have access to any future contexts. Although it is argued any amount of future contexts still makes a locally normalized model less expressive than its global counterpart.

representation learning component.

For both locally and globally normalized neural shift-reduce parsing models, inexact search also introduces search errors, as it does for both local and global structured perceptron-based shift-reduce parsing models (Huang et al., 2012). In locally normalized models, such errors may potentially be further exacerbated by the label bias problem, which hinders the model's ability to reward or penalize earlier decisions based on future evidence (Andor et al., 2016).

To enlarge the search space, beam search is often the preferred first solution. By allowing the highest scored action sequence to be taken as the output, it also provides a weak solution to the label bias problem for a locally normalized model, leading to some accuracy improvements (Table 4.10). However, as expected, such improvements diminish quickly after a certain beam size due to the underlying local normalization. Instead, I show that by using the weights of a locally normalized model as the starting point, a globally normalized model optimized for the evaluation metric can be obtained, which gives further significant accuracy improvements (§4.7).

Finally, note that as pointed out by Ranzato et al. (2016), locally normalized neural network models—such as the baseline RNN shift-reduce model above—can exhibit two other problems in addition to label bias. First, because the model is only exposed to the gold standard at training time, it can suffer from *exposure bias* whereby at test time only the model predictions are available. Second, there exists a *loss-evaluation mismatch*, where the loss function does not directly optimize towards the final evaluation metric.

As an added benefit, the global model address loss-evaluation mismatch using an expected F1 objective, it is also exposed to non-gold standard parser states and shift-reduce action sequences during training, thus mitigating exposure bias to some degree, without being explicitly designed to do so.

### 4.3.3 Expected F1 Training

The RNN used to train the global model has the same Elman architecture as the cross-entropy model. Given the cross-entropy model, its weights can be summarized as $\theta = \{\mathbf{U}, \mathbf{V}, \mathbf{W}\}$, which is used initialize the weights of the global model, and training proceeds as follows:

1. Parse a sentence $x_n$ in the training data using a beam-search decoder, and generate a $k$-best list of output parses using the *current* $\theta$, denoted as $\Lambda_{x_n}$.[5] Similar to other structured training approaches that use beam search (Zhang and Clark, 2008; Weiss et al., 2015; Watanabe and Sumita, 2015; Zhou et al., 2015), $\Lambda_{x_n}$ is as an approximation to the set of all possible parses of an input.

2. Let $y_i$ be the shift-reduce action sequence of a parse in the $k$-best list $\Lambda_{x_n}$, and let $|y_i|$ be its total number of actions and $y_{ij}$ be the $j$th action in $y_i$, for $1 \leq j \leq |y_i|$. Compute the log-linear action sequence score of $y_i$, $\rho(y_i)$, as a sum of individual action scores in that sequence: $\rho(y_i) = \sum_{j=1}^{|y_i|} \log s_\theta(y_{ij})$, where $s_\theta(y_{ij})$ is the softmax action score of $y_{ij}$ given by the RNN model. For each $y_i$, also compute its sentence-level F1 using the set of labeled, directed dependencies, denoted as $\Delta$, associated with its parser state. (F1 over labeled, directed dependencies is also the parser evaluation metric.)

3. Compute the negative expected F1 objective (-XF1, defined below) for $x_n$ using the scores obtained in the above step and minimize this objective using stochastic gradient descent (SGD) (maximizing the expected F1 for $x_n$). Repeat these three steps for other sentences in the training data, updating $\theta$ after processing each sentence, and iterate training in epochs until convergence.

Note that the above process is different from parse reranking (Collins, 2000; Charniak and Johnson, 2005), in which $\Lambda_{x_n}$ would stay the same for each $x_n$ in the training data across all epochs, and a reranker is trained on all fixed $\Lambda_{x_n}$; whereas the XF1 training procedure is on-line learning with parameters updated after processing each sentence and each $\Lambda_{x_n}$ is generated with a new $\theta$.

More formally, I define the loss $J(\theta)$, which incorporates all action scores in each

[5] No limit was put on $k$, and whenever a finished parser state was generated, it was appended to the $k$-best list. The $k$-best lists were found to be twice the size of a given beam size on average.

action sequence, and all action sequences in $\Lambda_{x_n}$, for each $x_n$ as

$$\begin{aligned} J(\theta) &= -\mathrm{XF1}(\theta) \\ &= -\sum_{y_i \in \Lambda_{x_n}} p(y_i|\theta)\mathrm{F1}(\Delta_{y_i}, \Delta_{x_n}), \end{aligned} \tag{4.4}$$

where $\mathrm{F1}(\Delta_{y_i}, \Delta_{x_n})$ is the sentence level F1 of the parse derived by $y_i$, with respect to the gold standard dependency structure $\Delta_{x_n}$ of $x_n$; $p(y_i|\theta)$ is the normalized probability score of the action sequence $y_i$, computed as

$$p(y_i|\theta) = \frac{\exp\{\gamma\rho(y_i)\}}{\sum_{y \in \Lambda_{x_n}} \exp\{\gamma\rho(y)\}}, \tag{4.5}$$

where $\gamma$ sharpens or flattens the distribution (Tromble et al., 2008).[6]

### 4.3.4 BPTS Derivatives for Training the XF1 Model

I now find the derivatives of the XF1 objective. Practically, they are used in Eq. 4.8 (§4.3.6) for training the model, but more importantly, they convey the intuitive insights of how each local softmax normalization contributes to the globally normalized objective.

First, by applying the chain rule to $J(\theta)$, I obtain

$$\begin{aligned} \frac{\partial J(\theta)}{\partial \theta} &= -\sum_{y_i \in \Lambda(x_n)} \sum_{y_{ij} \in y_i} \frac{\partial J(\theta)}{\partial s_\theta(y_{ij})} \frac{\partial s_\theta(y_{ij})}{\partial \theta} \\ &= -\sum_{y_i \in \Lambda(x_n)} \sum_{y_{ij} \in y_i} \delta_{y_{ij}} \frac{\partial s_\theta(y_{ij})}{\partial \theta}, \end{aligned}$$

where $\frac{\partial s_\theta(y_{ij})}{\partial \theta}$ is the standard softmax derivatives. Next, to compute $\delta_{y_{ij}}$, which is the derivative of the loss at the softmax layer, rewrite the loss in (4.4) as

$$\begin{aligned} J(\theta) = -\mathrm{XF1} &= -\frac{G(\theta)}{Z(\theta)} \\ &= -\frac{\sum_{y_i \in \Lambda_{x_n}} \exp\{\rho(y_i)\}\mathrm{F1}(\Delta_{y_i}, \Delta_{x_n})}{\sum_{y_i \in \Lambda_{x_n}} \exp\{\rho(y_i)\}}, \end{aligned} \tag{4.6}$$

[6] $\gamma = 1$ is used in all experiments.

and by simplifying:

$$\frac{\partial G(\theta)}{\partial s_\theta(y_{ij})} = \frac{1}{s_\theta(y_{ij})}\exp\{\rho(y_i)\}\mathrm{F1}(\Delta_{y_i}, \Delta_{x_n}),$$

$$\frac{\partial Z(\theta)}{\partial s_\theta(y_{ij})} = \frac{1}{s_\theta(y_{ij})}\exp\{\rho(y_i)\},$$

since

$$\frac{\partial \rho(y_i)}{\partial s_\theta(y_{ij})} = \frac{1}{s_\theta(y_{ij})}.$$

Finally, using Eq. 4.5 and Eq. 4.6 plus the above simplifications, the error term $\delta_{y_{ij}}$ can be derived using the quotient rule:

$$\begin{aligned}\delta_{y_{ij}} &= -\frac{\partial \mathrm{XF1}(\theta)}{\partial s_\theta(y_{ij})} \\ &= \frac{G(\theta)Z'(\theta) - G'(\theta)Z(\theta)}{Z^2(\theta)} \\ &= \frac{\exp\{\rho(y_i)\}}{Z(\theta)}(\mathrm{XF1}(\theta) - \mathrm{F1}(\Delta_{y_i}, \Delta_{x_n}))\frac{1}{s_\theta(y_{ij})} \\ &= p(y_i|\theta)(\mathrm{XF1}(\theta) - \mathrm{F1}(\Delta_{y_i}, \Delta_{x_n}))\frac{1}{s_\theta(y_{ij})}, \qquad (4.7)\end{aligned}$$

which has a simple closed form. It is also illuminating to see that the product consisting of the first two terms in Eq. 4.7 stays the same for all $y_{ij}$ in a $y_i$, which is exploited later in the implementation (§4.3.6).

A naive implementation of the XF1 training procedure would backpropagate the error gradients individually for each $y_i$ in $\Lambda_{x_n}$. To improve efficiency, observe that the unfolded network in the beam containing all $y_i$ becomes a DAG (with one hidden state leading to one or more resulting hidden states) and BPTS (§4.1.3) can be used to obtain all the backpropagated errors required for SGD. This optimization along with an efficient beam search implementation is used to achieve faster training.

### 4.3.5 More Efficient Beam Search with a Tree-Structured Stack

In theory, shift-reduce parsers have a runtime $\mathcal{O}(n)$ linear in the input length $n$. With beam search, as the number of parser actions, parser states and basic operations such as feature extractions become proportional to the size of the beam $k$, the linear runtime is often more accurately expressed as $\mathcal{O}(kn)$. In practice, however, unlike greedy shift-reduce parsers (Yamada and Matsumoto, 2003; Nivre and Scholz, 2004), which maintain only a single stack and queue for each input, beam search entails state-copying operations to duplicate the contents of the current beam into the next before modifying the copied parser states. As the size of the stack can also grow linearly in $n$, a beam search implementation that follows this idiom will instead have a runtime $\mathcal{O}(kn^2)$ quadratic in the sentence length. ZPAR first implemented a few optimizations to remedy this issue (Zhang and Clark, 2011b), and similar to Goldberg et al. (2013), the main ingredient is a data structure called tree-structured stack (TSS) building upon the graph-structured stack (Tomita, 1985). This optimization is also implemented in all models in this thesis.

The key idea of TSS is state sharing akin to a lattice structure, which gives a distributed representation of parser states. TSS can be conveniently implemented as an array, where each cell of the array contains a single incomplete parser state except the first cell that contains the complete initial state. Each state other than the first is incomplete in the sense that only a single shift-reduce action and the relevant subtree headed by it are captured, and a complete state is distributed across multiple incomplete states in the array.

Implementing TSS boils down to maintaining two pointers on each incomplete parser state. The first pointer is named PREVIOUS_STATE, which as the name suggests, always points to the (incomplete) state from which the current state is directly derived. For example, as depicted in Fig. 4-3a, the initial state • (in the TSS array cell 0) is the PREVIOUS_STATE of the second state, which is the PREVIOUS_STATE of the third state, and similarly for the remaining states. Therefore, from any state other than the initial state, by following the PREVIOUS_STATE pointers, the trace of the shift-reduce action sequence can be obtained. In Fig. 4-3a, this trace is SHIFT (sh), SHIFT (sh), REDUCE (re), UNARY (un).

For shift-reduce parsing, it is also common that an output stack contains more than one subtree. In this case, the root of each subtree can be more readily reached through

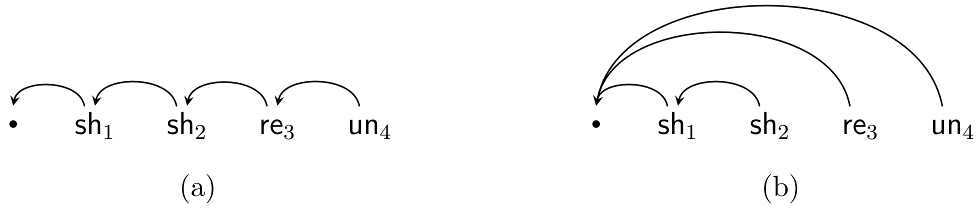


Figure 4-3: A TSS array with PREVIOUS_STATE (a) and PREVIOUS_STACK (b) pointers. Subscripts indicate array cell indexes. Index for the initial state • is omitted.

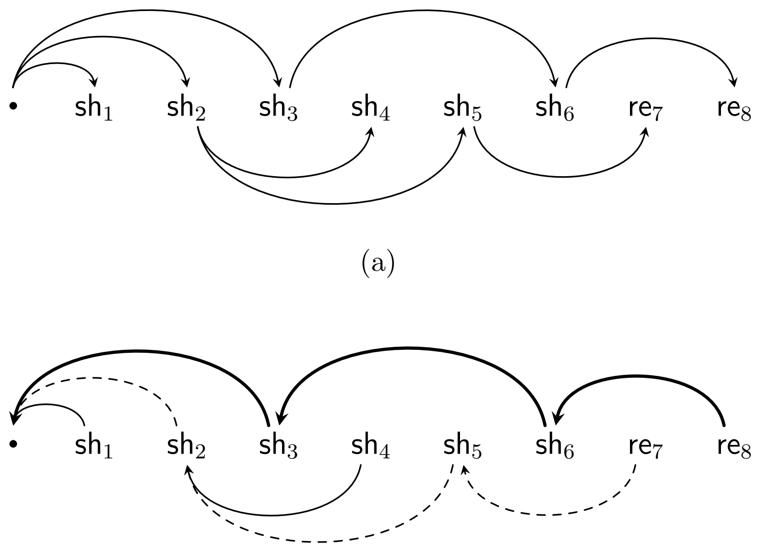
the second pointer PREVIOUS_STACK maintained on each incomplete state, and this pointer is updated according to the action type applied to a state. For a SHIFT action, because no existing subtrees would be consumed, PREVIOUS_STACK always points to the same state as PREVIOUS_STATE. For a REDUCE action, because two subtrees are consumed (the one from the current state and the other from PREVIOUS_STATE), PREVIOUS_STACK of the newly created state points to the PREVIOUS_STACK of the PREVIOUS_STACK of the current state. A UNARY action can be taken as consuming the top and the only subtree from the current state, hence the PREVIOUS_STACK pointer for the state resulting from a UNARY action points to the state given by the PREVIOUS_STACK of the current item. With the same example as in Fig. 4-3a, the PREVIOUS_STACK pointers are illustrated in Fig. 4-3b.

I show in the experimental section (§4.7) that beam search implemented using TSS can be very efficient, even for RNN parsing models.

### 4.3.6 More Efficient Training for Dynamically Shaped RNNs

Beam search implemented as a TSS can be exploited for a more efficient implementation of XF1 training. The key observation is that an unrolled RNN, as generated by beam search, can be represented with a TSS array by coupling its hidden states with the incomplete parser states. As parsing proceeds, beam search then generates both the TSS array and a dynamically shaped RNN for each training instance.

The first step for BPTS over the RNN contained in a TSS array is to mark out the hidden states. This can be achieved by marking out the (incomplete) states for each parse (or equivalently, each shift-reduce action sequence) in $\Lambda_{x_n}$ by following their PREVIOUS_STATE pointers in the TSS array. The pseudocode for doing this is



(a)

(b)

Figure 4-4: Example unrolled RNN in a TSS array (a), and the associated parser states with PREVIOUS_STATE pointers (b), with inverted arc directions; subscripts indicate array cell indexes. The dark ($\mathsf{sh}_3$ $\mathsf{sh}_6$ $\mathsf{re}_8$) and dashed ($\mathsf{sh}_2$ $\mathsf{sh}_5$ $\mathsf{re}_7$) arcs in (b) denote two shift-reduce action sequences. Another possible but not explicitly marked sequence is $\mathsf{sh}_2$ $\mathsf{sh}_4$.

shown in Fig. 4-5. An example unrolled RNN in a TSS is also shown in Fig. 4-4a, along with the PREVIOUS_STATE pointers (Fig. 4-4b).

For each parse in $\Lambda_{x_n}$, Mark-State in Fig. 4-5 does the actual marking (line 4). First, State is called to obtain the last parser state of a parse $y_i$, denoted as $s_{|y_i|}$ (line 5) which is marked (line 8) and acts as the handle for tracing back the TSS array, denoted as $\mathcal{L}_{x_n}$. Whenever a PREVIOUS_STATE $s_{\text{prev}}$ is encountered, it is also marked (line 10). On each marked state, a hashmap $\mathcal{A}$ is maintained, in which the key is a shift-reduce action index to a node in the output layer of the RNN stored during parsing at $s_{\text{curr}}$ as $s_{\text{curr}}.act$, which denotes the index of the action that generated $s_{\text{curr}}$ (line 11), and the value is a list of $k$-best parse indexes. During the marking process, this hashmap keeps track of all the shift-reduce actions outgoing from the hidden state of $s_{\text{prev}}$, which also survived beam search and ended up being part of a parse in $\Lambda_{x_n}$. For example, in Fig. 4-4b, both $\mathsf{sh}_4$ and $\mathsf{sh}_5$ are outgoing actions from their $s_{\text{prev}}$, $\mathsf{sh}_2$. Assuming $\mathsf{sh}_4$ is in parse $\hat{y}_1$ and $\mathsf{sh}_5$ is in parse $\hat{y}_2$, where both $\hat{y}_1$ and $\hat{y}_2$ are

```
function MARK($\Lambda_{x_n}$, $\mathcal{L}_{x_n}$)    ▷ input: $k$-best list $\Lambda_{x_n}$ and TSS array $\mathcal{L}_{x_n}$
  for each $\hat{y}_i \in \Lambda_{x_n}$ do
    MARK-STATE($\hat{y}_i$)
procedure MARK-STATE($y_i$)    ▷ the $i$th parse in $\Lambda_{x_n}$
  $s_{|y_i|} \leftarrow \mathcal{L}_{x_n}$.STATE($y_i$)    ▷ the last state of parse $y_i$ in the TSS array $\mathcal{L}_{x_n}$
  $s_{\text{curr}} \leftarrow s_{|y_i|}$
  $s_{\text{prev}} \leftarrow s_{|y_i|}$.PREVIOUS_STATE
  $s_{\text{curr}}.\mathit{mark} \leftarrow \mathit{true}$
  while true do
    $s_{\text{prev}}.\mathit{mark} \leftarrow \mathit{true}$
    $s_{\text{prev}}.\mathcal{A}[s_{\text{curr}}.\mathit{act}]$.INSERT($i$)    ▷ $\mathcal{A}$ is a hashmap, the key is an action index
    if $s_{\text{prev}} == s_0$ then    ▷ $s_0$ is the initial state
      break
    $s_{\text{curr}} \leftarrow s_{\text{prev}}$
    $s_{\text{prev}} \leftarrow s_{\text{prev}}$.PREVIOUS_STATE
```

Figure 4-5: Pseudocode for marking all parser states in a TSS array $\mathcal{L}_{x_n}$ associated with all the parses in a $k$-best list $\Lambda_{x_n}$.

```
function L-BPTS($\mathcal{L}_{x_n}$)    ▷ input is the TSS array $\mathcal{L}_{x_n}$
  for $r$ in $\mathcal{L}_{x_n}.\mathit{last} \ldots 0$ do    ▷ from the last index in $\mathcal{L}_{x_n}$
    $s_r \leftarrow \mathcal{L}_{x_n}[r]$
    if $s_r.\mathit{mark}$ and $|s_r.\mathit{succ}| > 0$ then
      $\boldsymbol{\delta}^u_{s_r} = \sum_{s' \in s_r.\mathit{succ}} s'.\boldsymbol{\delta}^h_{s'}$    ▷ aggregate errors from child states
      $\boldsymbol{\delta}^a_{s_r} = \text{BPTS}(s_r.\mathcal{A})$    ▷ aggregate errors from outgoing actions
      $\boldsymbol{\delta}^h_{s_r} = \mathbf{d}^h_{s_r} \circ (\mathbf{W}\boldsymbol{\delta}^u_{s_r} + \mathbf{V}\boldsymbol{\delta}^a_{s_r})$    ▷ aggregate all errors at $s_r$
```

Figure 4-6: Pseudocode for BPTS over a TSS array $\mathcal{L}_{x_n}$. $\mathbf{d}^h_{s_r}$ is the activation derivative vector for the hidden state of $s_r$; $\circ$ is element-wise product.

in $\Lambda_{x_n}$, then the hashmap $\mathsf{sh}_2.\mathcal{A}$ will contain *at least* two entries, namely $(\mathsf{sh}_4.\mathit{act}, 1)$ and $(\mathsf{sh}_5.\mathit{act}, 2)$.

Note that a single hidden state in an unrolled RNN contained in the beam can contribute to multiple parses in $\Lambda_{x_n}$. That is, a single hidden state may potentially be marked out more than once. This is indeed necessary for BPTS that needs to aggregate backpropagated errors from all actions "leaving" a hidden state that ended up in one of the action sequences of the parses in $\Lambda_{x_n}$. The aforementioned $\mathsf{sh}_2$ is such an example, which will be marked from both $\mathsf{sh}_4$ and $\mathsf{re}_7$.

Once $\Lambda_{x_n}$ is marked, I traverse $\mathcal{L}_{x_n}$ backwards from the last cell of the TSS array to do the actual BPTS computations. This traversal visits each cell of TSS only once, such that the PREVIOUS_STATE DAG (e.g., the one in Fig. 4-4b) is visited in a reverse topological order, and all BPTS is done in one single traversal of $\Lambda_{x_n}$. I show the pseudocode for doing this in Fig. 4-6. On line 3, the parser state $s_r$ at position $r$ in $\mathcal{L}_{x_n}$ is obtained. If $s_r$ has been marked by $\textsc{Mark}(\Lambda_{x_n}, \mathcal{L}_{x_n})$, and it leads to at least one child state in the unrolled RNN ($|s_r.\mathit{succ}| > 0$ on line 4, in which *succ* denotes the set of child states), the cumulative backpropagated error $\boldsymbol{\delta}^h_{s_r}$ at the hidden state of $s_r$ is computed (line 7). This involves aggregating errors from all the child states $u$ contained in $s_r.\mathit{succ}$ (line 5) and all the action indexes (or keys) in $s_r.\mathcal{A}$, where for each index, errors are aggregated from all $k$-best parses it is part of, s.t. (on line 6):

$$\begin{aligned}
\boldsymbol{\delta}^a_{s_r} &= \text{BPTS}(s_r.\mathcal{A}) \\
&= \sum_{m \in s_r.\mathcal{A}.\mathit{keys}} \sum_{i \in s_r.\mathcal{A}[m]} \boldsymbol{\delta}_m \boldsymbol{\delta}_{im} \\
&= \sum_{m \in s_r.\mathcal{A}.\mathit{keys}} \sum_{i \in s_r.\mathcal{A}[m]} \boldsymbol{\delta}_m \underbrace{p(y_i|\theta)(\text{XF1}(\theta) - \text{F1}(\Delta_{y_i}, \Delta_{x_n}))\frac{1}{z_m}}_{\text{XF1 gradient per action, Eq. 4.7}},
\end{aligned} \tag{4.8}$$

where according to Eq. 4.2:

$$\boldsymbol{\delta}_m = \begin{pmatrix} z_1(\alpha_{1m} - z_m)\beta_1 \\ z_2(\alpha_{2m} - z_m)\beta_2 \\ \vdots \\ z_l(\alpha_{lm} - z_m)\beta_l \\ \vdots \\ z_C(\alpha_{Cm} - z_m)\beta_C \end{pmatrix},$$

$$\begin{cases} \alpha_{lm} = 1 & \text{if } l == m \\ \alpha_{lm} = 0 & \text{otherwise,} \end{cases} \quad \text{and} \quad \begin{cases} \beta_l = 1 & \text{if } l \in \mathcal{T}(s_r) \\ \beta_l = 0 & \text{otherwise,} \end{cases}$$

where $C$ is the total number of possible actions in the model (§4.4), $1 \le m \le C$, $1 \le l \le C$, and $\mathcal{T}(s_r)$ is the set of feasible shift-reduce actions at state $s_r$. Note that by definition, $\forall m$ s.t. $m \in s_r.\mathcal{A}.\mathit{keys}$, $m \in \mathcal{T}(s_r)$.

|  |  |  |  |
|---|---|---|---|
| $s_0$.w | $s_1$.w | $s_2$.w | $s_3$.w |
| $s$.w$_0$ | $s_.$w$_1$ | $s$.w$_2$ | $s$.w$_3$ |
| $s_0$.l.w | $s_1$.l.w | $s_0$.r.w | $s_1$.r.w |
| $q_0$.w | $q_1$.w | $q_2$.w | $q_3$.w |
| $s_0$.c | $s_0$.l.c | $s_0$.r.c |  |
| $s_1$.c | $s_1$.l.c | $s_1$.r.c |  |
| $s_2$.c | $s_3$.c |  |  |

Table 4.1: Atomic feature templates.

## 4.4 CCG and the RNN Shift-Reduce Parsing Model

The total number of output units in the RNN is equal to the number of lexical categories (i.e., all possible SHIFT actions), plus 10 units for REDUCE and 18 units for UNARY actions, corresponding to the CCG rules in Clark and Curran (2007).[7]

The same as the structured perceptron normal-form model (Zhang and Clark, 2011a), all features in the RNN model fall into three types: word, POS and CCG category. Table 4.1 shows the atomic feature templates, where $|f_w| = 16$, $|f_p| = 16$ and $|f_c| = 8$ (with all word-based features generalized to POS features). Similarly, each template has two parts: the first part denotes parser state context and the second part denotes the feature type. Recall also that $s$ denotes stack contexts and $q$ denotes queue contexts. For example, $s_0$ is the top subtree on the stack, with $s_0$.l being its left child and $s_0$.r being its right child.

As an extra addition, the RNN models define a feature type denoted as w$_n$ ($0 \leq n \leq 3$), which represents the $n$th word of the input string that has been shifted onto the stack. But disregarding this addition, the set of atomic feature templates is chosen such that each higher-order feature template of the normal-form model (Table 3.2, §3.3) can be obtained as the conjunction of some atomic feature templates.

Note, however, unlike the structured perceptron models, including both the normal-form and dependency models, features in the RNN models are not conjoined with action types.

[7] In principle, only 1 re unit is needed, but 9 additional units are used to handle non-standard CCG rules in the treebank.

## 4.5 Bidirectional RNN Supertagging

I extend the RNN supertagging model in the previous chapter by using a bidirectional RNN (BRNN). The BRNN processes an input in both directions with two separate hidden layers, which are then fed to one output layer to make predictions. At each time step $t$, I compute the *forward* hidden state $\mathbf{h}_t$ for $t = (0, 1, \ldots, n-1)$; the *backward* hidden state $\hat{\mathbf{h}}_t$ is computed similarly but from the reverse direction for $t = (n-1, n-2, \ldots, 0)$ as

$$\hat{\mathbf{h}}_t = f(\hat{\mathbf{U}}\mathbf{x}_t + \hat{\mathbf{W}}\mathbf{h}_{t+1}),$$

and the output layer, for $t = (0, 1, \ldots, n-1)$, is computed as

$$\mathbf{y}_t = f(\hat{\mathbf{V}}[\mathbf{h}_t; \hat{\mathbf{h}}_t]).$$

The BRNN introduces two new parameter matrices $\hat{\mathbf{U}}$ and $\hat{\mathbf{W}}$ and replaces the old hidden-to-output matrix $\mathbf{V}$ with $\hat{\mathbf{V}}$ to take two hidden layers as input. I use the same three feature embedding types as the unidirectional model (§4.2), and all features are extracted from a context window surrounding and including the current word.

## 4.6 Experiments: Chart Parsing

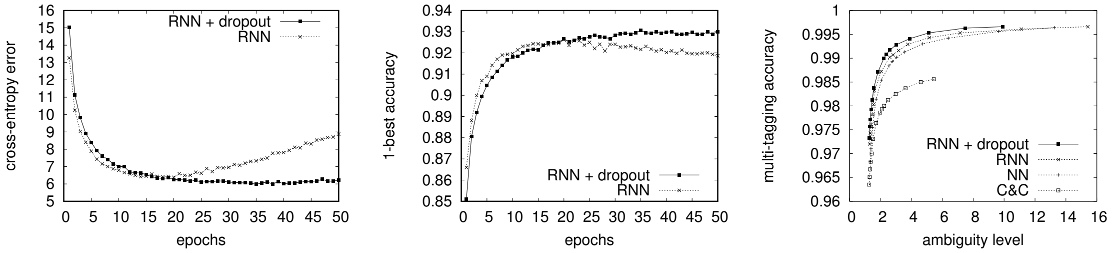
**Datasets and Baseline.** In addition to CCGBank (Hockenmaier and Steedman, 2007), the Wikipedia dataset of Honnibal et al. (2009) and the BioInfer dataset of Pyysalo et al. (2007) are used as two out-of-domain test sets. I compared supertagging accuracy with the MaxEnt C&C supertagger and the neural network tagger of Lewis and Steedman (2014b) (henceforth NN), and I also evaluated parsing accuracy using these three supertaggers as a front-end to the C&C parser. The same 425 supertag set used in the C&C parser and NN are used in all models. The MaxEnt C&C supertagger uses POS tag features and a tag dictionary, neither of which are used by other supertaggers.

**Embedding Preprocessing.** Pre-trained word embeddings from Turian et al. (2010) were used to initialize look-up table $\mathcal{L}_w$, and a set of word pre-processing techniques was applied at both training and test time. All words are first lower-cased, and all numbers are collapsed into a single digit '0'. If a lower-cased hyphenated word does not have an entry in the pre-trained word embeddings, pre-processing is backed off to the substring after the last hyphen. For compound words and numbers delimited by "\/", pre-processing is backed off to the substring after the delimiter. After pre-processing, the Turian embeddings have a coverage of 94.25% on the training data; for out-of-vocabulary words, three separate randomly initialized embeddings are used for lower-case alphanumeric words, upper-case alphanumeric words, and non-alphanumeric symbols. For padding at the start and end of a sentence, the "unknown" entry from the pre-trained embeddings is used. Look-up tables $\mathcal{L}_s$ and $\mathcal{L}_c$ were also randomly initialized, and all look-up tables were modified during training.

**Hyperparameters and Training.** For $\mathcal{L}_w$, I used the scaled 50-dimensional Turian embeddings ($n = 50$ for $\mathcal{L}_w$) as initialization. I experimented during development with using 100-dimensional embeddings and found no improvements in the resulting model. Out-of-vocabulary embedding values in $\mathcal{L}_w$ and all embedding values in $\mathcal{L}_s$ and $\mathcal{L}_c$ were initialized with a uniform distribution in the interval $[-2.00, 2.00]$. The embedding dimension size $m$ of $\mathcal{L}_s$ and $\mathcal{L}_c$ was set to 5. Other parameters of the network $\{\mathbf{U}, \mathbf{V}, \mathbf{W}\}$ were also initialized a uniform distribution in the interval $[-2.00, 2.00]$, and were then scaled by their corresponding input vector size. I experimented with context window sizes of 3, 5, 7, 9 and 11 during development and found a window size of 7 gave the best performing model on the dev set. I used a fixed learning rate of $0.25 \times 10^{-2}$ and a hidden state size of 200.

To train the model, I optimized cross-entropy loss with SGD using mini-batched BPTT, where the mini-batch size was set to 9, by tuning on the dev set.

**Embedding Dropout Regularization.** Without any regularization, I found cross-entropy error on the dev set started to increase while the error on the training set was continuously driven to a very small value (Fig. 4-7a). With the suspicion of



(a) (b) (c)

Figure 4-7: Learning curve (a) and 1-best tagging accuracy (b) of the RNN model on CCGBank Section 00. Plot (c) shows ambiguity vs. multi-tagging accuracy for all supertaggers (auto POS).

| | Accuracy | Time |
|---|---|---|
| C&C (gold POS) | 92.60 | - |
| C&C (auto POS) | 91.50 | **0.57** |
| NN | 91.10 | 21.00 |
| RNN | 92.63 | - |
| RNN+dropout | **93.07** | 2.02 |

Table 4.2: 1-best tagging accuracy and speed comparison on CCGBank Section 00 with a single CPU core (1,913 sentences), tagging time in secs.

overfitting, I experimented with $\ell_1$ and $\ell_2$ regularization and learning rate decay but none of these techniques gave any noticeable improvements for the model. Following Legrand and Collobert (2015), I instead implemented word embedding dropout as a regularization for all the look-up tables. As a result, I observed more stable learning and better generalization. Similar to other forms of droput (Srivastava et al., 2014), I randomly dropped units and their connections to other units at training time. Concretely, I applied a binary dropout mask to $x_t$, with a dropout rate of 0.25, and at test time no mask is applied, but the input to the network, $x_t$, at each word position is scaled by 0.75.

### 4.6.1 Supertagging Results

The best non-regularized model was obtained after the 20th epoch, and it took 35 epochs for the dropout model to peak (Fig. 4-7b). I report results obtained with the dropout model for all experiments.

Table 4.2 shows 1-best supertagging accuracies on the dev set. The accuracy

| | Section 23 | Wiki | Bio |
|---|---|---|---|
| C&C (gold POS) | 93.32 | 88.80 | 91.85 |
| C&C (auto POS) | 92.02 | 88.80 | **89.08** |
| NN | 91.57 | 89.00 | 88.16 |
| RNN | **93.00** | **90.00** | 88.27 |

Table 4.3: 1-best tagging accuracy comparison on CCGBank Section 23 (2,407 sentences), Wikipedia (200 sentences) and Bio-GENIA (1,000 sentences).

of the C&C supertagger drops about 1.00% with automatically assigned POS tags, while the RNN model gives higher accuracy (+0.47%) than the C&C supertagger with gold POS tags. All timing values were obtained on an Intel i7-4790K CPU, and all implementations are in C++ except NN which is implemented in TORCH and JAVA.

Table 4.4 compares different supertagging models for multi-tagging accuracy at the default $\beta$ levels used by the C&C parser on the dev set. At the first $\beta$ level ($0.75 \times 10^{-1}$), the three supertagging models give very close ambiguity levels, but the RNN model clearly outperforms NN and C&C (auto POS) in both word (WORD) and sentence (SENT) level accuracies, giving similar word-level accuracy as C&C (gold POS). For other $\beta$ levels (except $\beta = 0.01 \times 10^{-1}$), the RNN model gives comparable ambiguity levels to the C&C model which uses a tagdict, while being much more accurate than both the other two models.

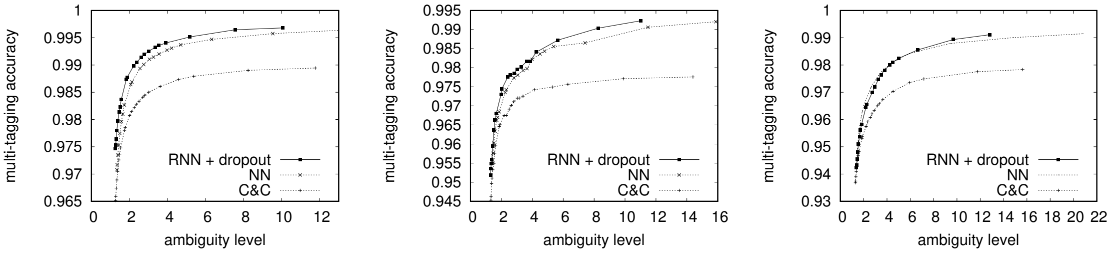
Fig. 4-7c compares multi-tagging accuracies of all the models on the dev set. For all models, the same $\beta$ levels were used (ranging from $0.75 \times 10^{-1}$ to $10^{-4}$, and all C&C default values were included). The RNN model consistently outperforms other models across different ambiguity levels.

Table 4.3 shows 1-best accuracies of all models on the test data sets (following Lewis and Steedman (2014b), results for Bio were obtained with Bio-GENIA gold standard CCG lexical category data from Rimell and Clark (2008)). With gold standard POS tags, the C&C model outperforms both the NN and RNN models on CCGBank and Bio-GENIA; while with auto POS, the accuracy of the C&C model drops significantly.

Fig. 4-8 shows multi-tagging accuracies on all test data (using $\beta$ levels ranging

|  | RNN |  |  | NN |  |  | C&C (auto POS) |  |  | C&C (gold POS) |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\beta \times 10^{-1}$ | WORD | SENT | ambi. | WORD | SENT | ambi. | WORD | SENT | ambi. | WORD | SENT | ambi. |
| 0.75 | 97.33 | 66.07 | 1.27 | 96.83 | 61.27 | 1.34 | 96.34 | 60.27 | 1.27 | **97.34** | **67.43** | 1.27 |
| 0.30 | **98.12** | **74.39** | 1.46 | 97.81 | 70.83 | 1.58 | 97.05 | 65.50 | 1.43 | 97.92 | 72.87 | 1.43 |
| 0.10 | **98.71** | **81.70** | 1.84 | 98.54 | 79.25 | 2.06 | 97.63 | 70.52 | 1.72 | 98.37 | 77.73 | 1.72 |
| 0.05 | **99.01** | **84.79** | 2.22 | 98.84 | 83.38 | 2.55 | 97.86 | 72.24 | 1.98 | 98.52 | 79.25 | 1.98 |
| 0.01 | **99.41** | **90.54** | 3.90 | 99.29 | 89.07 | 4.72 | 98.25 | 80.24 | 3.57 | 99.17 | 87.19 | 3.00 |

Table 4.4: Multi-tagging accuracy vs. ambiguity (supertags/word) at the default C&C $\beta$ levels on CCGBank Section 00.



(a) (b) (c)

Figure 4-8: Multi-tagging accuracy for all supertagging models on CCGBank Section 23 (a), Wikipedia (b) and Bio-GENIA (c) (auto POS).

| | LP | LR | LF | SENT | CAT | cov. |
|---|---|---|---|---|---|---|
| C&C (normal) | 85.18 | 82.53 | 83.83 | 31.42 | 92.39 | 100 |
| C&C (hybrid) | 86.07 | 82.77 | 84.39 | 32.62 | 92.57 | 100 |
| C&C (normal + RNN) | 86.74 | 84.58 | 85.65 | 34.13 | 93.60 | 100 |
| C&C (hybrid + RNN) | **87.73** | **84.83** | **86.25** | **34.97** | **93.84** | 100 |
| C&C (normal) | 85.18 | 84.32 | 84.75 | 31.73 | 92.83 | 99.01 (C&C cov.) |
| C&C (hybrid) | 86.07 | 84.49 | 85.28 | 32.93 | 93.02 | 99.06 (C&C cov.) |
| C&C (normal + RNN) | 86.81 | 86.01 | 86.41 | 34.37 | 93.80 | 99.01 (C&C cov.) |
| C&C (hybrid + RNN) | **87.77** | **86.25** | **87.00** | **35.20** | **94.04** | 99.06 (C&C cov.) |
| C&C (normal + RNN) | 86.74 | 86.15 | 86.45 | 34.33 | 93.81 | **99.42** |
| C&C (hybrid + RNN) | 87.73 | 86.41 | 87.06 | 35.17 | 94.05 | **99.42** |

Table 4.5: Parsing development results on CCGBank Section 00 (auto POS). No separate development experiments for the Wikipedia and BioInfer data sets.

from $0.75 \times 10^{-1}$ to $10^{-6}$, and all C&C default values are included). On CCGBank, the RNN model has a clear accuracy advantage, while on the other two data sets, the accuracies given by the NN model are closer to the RNN model at some ambiguity levels. However, both the NN and RNN models are more robust than the C&C model on the two out-of-domain data sets.

### 4.6.2 Parsing Results

I integrated the supertagging model into the C&C parser, at both training and test time, using all default settings; C&C hybrid model was used for CCGBank and Wikipedia; the normal-form model was used for the BioInfer data, following Lewis and Steedman (2014b) and Rimell and Clark (2008). Parsing development results are shown in Table 4.5; for out-of-domain data sets, no separate development ex-

| | LP | LR | LF | SENT | CAT | cov. |
|---|---|---|---|---|---|---|
| C&C (hybrid) | 86.24 | 84.17 | 85.19 | 33.24 | 93.00 | 100 |
| C&C (hybrid + NN) | 86.71 | 85.40 | 86.05 | 33.32 | 93.31 | 100 |
| C&C (hybrid + RNN) | **87.68** | **86.41** | **87.04** | **35.02** | **93.87** | 100 |
| C&C (hybrid) | 86.24 | 84.85 | 85.54 | 33.43 | 93.03 | 99.42 (C&C cov.) |
| C&C (hybrid + NN) | 86.71 | 85.56 | 86.13 | 33.35 | 93.32 | 99.92 |
| C&C (hybrid + RNN) | **87.68** | **86.47** | **87.07** | **35.04** | **93.87** | **99.96** |

Table 4.6: Parsing results on CCGBank Section 23 (auto POS).

| | LP | LR | LF | SENT | CAT | cov. |
|---|---|---|---|---|---|---|
| C&C (hybrid) | 81.58 | 79.48 | 80.52 | 25.50 | 89.83 | 100 |
| C&C (hybrid + NN) | 82.65 | 81.36 | 82.00 | **29.50** | 90.41 | 100 |
| C&C (hybrid + RNN) | **83.22** | **81.78** | **82.49** | 29.00 | **90.73** | 100 |
| C&C (hybrid) | 81.58 | 80.08 | 80.83 | 25.63 | 89.87 | 99.50 (C&C cov.) |
| C&C (hybrid + NN) | 82.55 | 81.24 | 81.89 | 29.65 | 90.38 | 99.50 (C&C cov.) |
| C&C (hybrid + RNN) | **83.18** | **81.72** | **82.45** | **33.67** | **90.69** | 99.50 (C&C cov.) |

Table 4.7: Parsing results on Wikipedia-200 (auto POS; both NN and RNN have 100% coverage with --force-words option of C&C set to false).

| | LP | LR | LF | cov. |
|---|---|---|---|---|
| C&C (normal) | 77.78 | 76.07 | 76.91 | 95.40 |
| C&C (normal + NN) | 79.77 | **78.62** | **79.19** | 97.40 |
| C&C (normal + RNN) | **80.10** | 78.21 | 79.14 | **97.80** |
| C&C (normal) | 77.78 | 71.44 | 74.47 | 100 |
| C&C (normal + NN) | 79.77 | 75.35 | 77.50 | 100 |
| C&C (normal + RNN) | **80.10** | **75.52** | **77.74** | 100 |

Table 4.8: Parsing results on BioInfer (auto POS).

periments were done. Final results are shown in Table 4.6 (CCGBank Section 23), Table 4.7 (Wikipedia) and Table 4.8 (BioInfer). As can be seen, parsing accuracies on CCGBank and Wikipedia are substantially improved, and the accuracy on CCGBank represents an F1 improvement of 1.53%/1.85%, which are state-of-art parsing results for the C&C models.

## 4.7 Experiments: Shift-Reduce Parsing

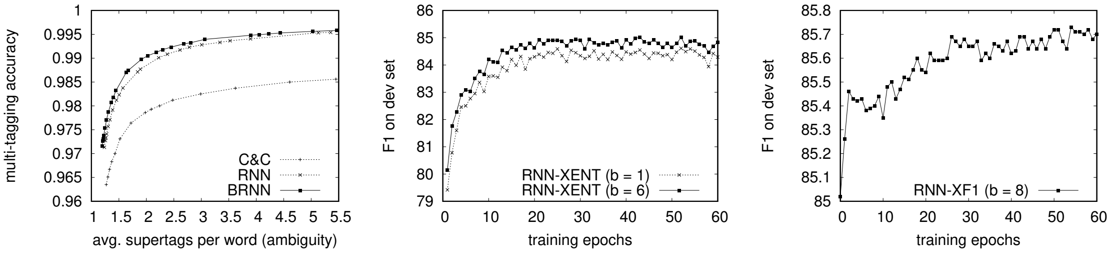
**Baselines.** The baselines are the same as in the previous chapter including the normal-form shift-reduce model (Zhang and Clark, 2011a), along with a reimplementation of it for additional reference, and the C&C normal-form and hybrid-models. I also include the feed-forward shift-reduce CCG parser of Ambati et al. (2016), which is a beam-search shift-reduce parser based on Chen and Manning (2014) and Weiss et al. (2015).

**Hyperparameters.** For the BRNN supertagging model, I used identical hyperparameter settings as the RNN supertagging model (§4.6). For all RNN parsing models, the weights were initialized using a uniform distribution in the interval $[-2.00, 2.00]$, and scaled by their fan-in (Bengio, 2012); the hidden layer size was 220, and 50-dimensional embeddings were used for all feature types and scaled Turian embeddings were used (Turian et al., 2010) for word embeddings. The same word embedding processing applied for the RNN supertagging model was used (§4.6). I also pretrained CCG lexcial category and POS embeddings by using the GENSIM word2vec implementation.[8] The data used for this was obtained by parsing a Wikipedia dump using the C&C parser and concatenating the output with CCGBank Sections 02-21. Embeddings for unknown words and CCG categories outside of the lexical category set were also uniformly initialized ($[-2.00, 2.00]$) without scaling.

To train all the models, I used a fixed learning rate of $0.25 \times 10^{-2}$ and did not truncate the gradients for BPTT, except for training the baseline cross-entropy RNN parsing model where a BPTT step size of 9 was used. Embedding dropout was also applied (§4.6), with a dropout rate of 0.25 for the supertagger and 0.30 for the parser.

10-fold jackknifing was used for both POS tagging and supertagging. For both development and test experiments, automatically assigned POS tags given by the C&C POS tagger were used. The BRNN supertagging model was used by all RNN parsing models for both training and testing. For training only, if the gold standard lexical category is not available for a word, it is added to the list of categories.

[8] https://radimrehurek.com/gensim/



(a) (b) (c)

Figure 4-9: Experiment results on the dev set. Multi-tagging accuracy (a). F1 vs. epoch for the Rnn-Xent models with beam size $b \in \{1, 6\}$ (b). F1 vs. epoch for the Rnn-XF1 models with $b = 8$ (c).

| | Dev | Test |
|---|---|---|
| c&c (gold POS) | 92.60 | 93.32 |
| c&c (auto POS) | 91.50 | 92.02 |
| Rnn | 93.07 | 93.00 |
| Brnn | **93.49** | **93.52** |

Table 4.9: 1-best supertagging accuracy comparison.

### 4.7.1 Supertagging Results

Table 4.9 shows 1-best supertagging results. On the test set, the Brnn supertagger achieves a 1-best accuracy of 93.52%, an absolute improvement of 0.52% over the RNN model.

Fig. 4-9a shows multi-tagging accuracy comparison for the three supertaggers by varying the variable-width beam probability cut-off value $\beta$ for each supertagger (§2.2). For this experiment $\beta$ values ranging from 0.09 to $2 \times 10^{-4}$ were used and it can be seen that the Brnn supertagger consistently achieves better accuracies at similar ambiguity levels.

### 4.7.2 Parsing Results

To train the cross-entropy locally normalized model (Rnn-Xent), the Brnn supertagger was used with a supertagger $\beta$ value of $0.25 \times 10^{-3}$ (with an average ambiguity of 5.02). SGD training was ran until no accuracy gains were observed, and the best Rnn-Xent model was obtained after the 52nd epoch (Fig. 4-9b).

It was also found that using a relatively smaller supertagger $\beta$ value (higher am-

|  | 0.09 | 0.08 | 0.07 | 0.06 |
|---|---|---|---|---|
| $b = 1$ | 84.61 | 84.58 | 84.55 | 84.50 |
| $b = 2$ | 84.94 | 84.86 | 84.86 | 84.81 |
| $b = 4$ | 85.01 | 84.95 | 84.92 | 84.92 |
| $b = 6$ | **85.02** | 84.96 | 84.94 | 84.93 |
| $b = 8$ | **85.02** | 84.99 | 84.96 | 84.95 |
| $b = 16$ | 85.01 | 84.95 | 84.97 | 84.98 |

Table 4.10: The effect on dev F1 by varying the beam size (b) and supertagger $\beta$ value for the Rnn-Xent model.

biguity) for training, and a larger $\beta$ value (lower ambiguity) for testing, resulted in more accurate models, and the final $\beta$ value was chosen to be 0.09 using development tuning (Table 4.10). This observation was different from the normal-form shift-reduce model (Zhang and Clark, 2011a) and the dependency model (§3), which used the same $\beta$ values for training and testing.

I also experimented with using different beam sizes at test time for the Rnn-Xent model (Table 4.10): with $b = 6$, I obtained an accuracy of 85.02%, an improvement of 0.41% over $b = 1$ (with a $\beta$ value of 0.09). Accuracy gains were seen up to $b = 8$ (with very minimal gains with $b = 16$ for $\beta$ values 0.06 and 0.07), after which the accuracy started to drop. F1 on dev with $b = 6$ across all training epochs are also shown in Fig. 4-9b, and the best model was obtained after the 43rd epoch.

For the XF1 model (Rnn-XF1), I used $b = 8$ and a supertagger $\beta$ value of 0.09 for both training and testing. Fig.4-9c shows dev F1 versus the number of training epochs. The best dev F1 was obtained after the 54th epoch with an accuracy of 85.73%, 1.12% higher than that of the Rnn-Xent model with $b = 1$ and 0.71% higher than the Rnn-Xent model with $b \in \{6, 8\}$. This result improves over the normal-form (Zhang and Clark, 2011a) and the dependency model (§3) by 0.73% and 0.55%, respectively (Table 4.11).

On the test set (Table 4.12), the beam-search Rnn-XF1 model achieves a final F1 of 86.42%, improving over the Rnn-Xent baseline by 1.47%.

Finally, to speed up the RNN models, I implemented precomputation (Devlin et al., 2014) to cache the top 20K word embeddings and all POS embeddings. This

|  | beam | LP | LR | LF | SENT | CAT |
|---|---|---|---|---|---|---|
| C&C (normal) | - | 85.18 | 82.53 | 83.83 | 31.42 | 92.39 |
| C&C (hybrid) | - | 86.07 | 82.77 | 84.39 | 32.62 | 92.57 |
| Zhang and Clark (2011a) | 16 | 87.15 | 82.95 | 85.00 | 33.82 | 92.77 |
| Zhang and Clark (2011a)* | 16 | 86.76 | 83.15 | 84.92 | 33.72 | 92.64 |
| SHIFT-REDUCE-DEP | 128 | 86.29 | **84.09** | 85.18 | 34.40 | 92.75 |
| Ambati et al. (2016)† | 1 | - | - | 82.65 | - | 91.72 |
| Ambati et al. (2016)‡ | 16 | - | - | 85.69 | - | 93.02 |
| RNN-XENT | 1 | 88.12 | 81.38 | 84.61 | 33.82 | 93.42 |
| RNN-XENT | 6 | 87.96 | 82.27 | 85.02 | 34.29 | 93.47 |
| RNN-XF1 | 8 | **88.20** | 83.40 | **85.73** | **34.97** | **93.56** |

Table 4.11: Parsing results on Section 00 (100% coverage and auto POS). $*$ = reimplementation; † = cross-entropy; ‡ = cross-entropy + structured perceptron.

|  | beam | LP | LR | LF | SENT | CAT | Speed |
|---|---|---|---|---|---|---|---|
| C&C (normal) | - | 85.45 | 83.97 | 84.70 | 32.82 | 92.83 | 97.90 |
| C&C (hybrid) | - | 86.24 | 84.17 | 85.19 | 33.24 | 93.00 | 95.25 |
| Zhang and Clark (2011a) | 16 | 87.43 | 83.61 | 85.48 | 35.19 | 93.12 | - |
| Zhang and Clark (2011a)* | 16 | 87.04 | 84.14 | 85.56 | 34.98 | 92.95 | 49.54 |
| SHIFT-REDUCE-DEP | 128 | 87.03 | 85.08 | 86.04 | 35.69 | 93.10 | 12.85 |
| Ambati et al. (2016)† | 1 | - | - | 83.27 | - | 91.89 | **350.00** |
| Ambati et al. (2016)‡ | 16 | - | - | 85.57 | - | 92.86 | - |
| RNN-XENT | 1 | 88.53 | 81.65 | 84.95 | 32.99 | 93.57 | 337.45 |
| RNN-XENT | 6 | 88.54 | 82.77 | 85.56 | 34.19 | 93.68 | 96.04 |
| RNN-XF1 | 8 | **88.74** | 84.22 | **86.42** | 34.73 | **93.87** | 67.65 |

Table 4.12: Parsing results on Section 23 (100% coverage and auto POS). $*$ = reimplementation; † = cross-entropy; ‡ = cross-entropy + structured perceptron. Speed (sents/sec; result for Ambati et al. (2016) is taken from the paper).

made the RNN-XENT parser more than three times faster than the C&C parser.[9]

## 4.8 Related Work

**Sequence Labelling with Neural Networks.** A myriad of neural network models have been proposed for sequence labeling tasks such as chunking, POS tagging and NER (Collobert et al., 2011; Lewis and Steedman, 2014b; Chiu and Nichols, 2015; Huang et al., 2015; Labeau et al., 2015; Ma and Hovy, 2016; Lample et al.,

[9] Speed measured with an Intel i7-4790K CPU.

2016; Lewis et al., 2016; Yang et al., 2016). A crucial aspect of all these models, as with most current neural network models for language processing, is the utilization of token-level (word or character) embeddings to mitigate the feature sparsity problem of more traditional one-hot indicator feature representations. In addition, the combination of embeddings and various neural network architectures results in models that have demonstrated high efficacy for sequence labelling. Primarily, such models have the advantage of being able to naturally capture long-range label dependencies—either through the use of recurrent neural networks alone (Huang et al., 2015; Labeau et al., 2015), or through a combination of neural networks and additional enhancements such as CRFs (Collobert et al., 2011; Huang et al., 2015; Lample et al., 2016; Ma and Hovy, 2016). In addition, they allow for end-to-end task-independent models that are entirely free of task-specific feature engineering (Collobert et al., 2011; Huang et al., 2015) as well as unified language-independent models that can generalize across languages by learning morphological and orthographic distributed representations (Lample et al., 2016; Yang et al., 2016).

The RNN model I have explored in this chapter is attractively simple and the foremost goal here is to test its viability in capturing sequence-level long-term dependencies for supertagging; more importantly, I have shown such a model has a positive impact on parsing accuracy. Recent work has also shown that neural network supertaggers can be used in an A* supertag-factored framework, in which CCG derivations are scored solely based on inside and outside probabilities obtained from CCG lexical category sequences (Lewis and Steedman, 2014a; Lewis et al., 2016). In such models, there is no explicit statistical model of the derivation or dependencies, and parsing accuracy is almost exclusively determined by supertagging accuracy. By contrast, I have found in my experiments that integrating the RNN supertagger with the parsing models at both training and inference time further improves parsing accuracy. I owe this to the fact that the supertagger determines the derivation space of the C&C parser, and hence affects parsing model estimation. This is further demonstrated by Vaswani et al. (2016), who obtain significant parsing accuracy improvements for the JAVA version of the C&C parser (Clark et al., 2015) by using an

LSTM supertagger in a similar way.

On the other hand, with only existing supertagging and parsing models, Auli and Lopez (2011a) showed that a tighter integration of these two components leads to a more accurate parser. In particular, they integrated the C&C two-stage pipeline more closely using belief propagation and dual decomposition at inference time to outperform the baseline. While this suggests the potential for further improvements by integrating a more accurate supertagging model into their parser, it still leaves the problem of fully integrated supertagging and parsing wide open, which should be a fruitful avenue for future work.

Another strand of work that ties to the current work is on improving supervised parsing models with additional labeled or unlabeled data. For instance, by using self-training (Honnibal et al., 2009; Kuhlmann et al., 2010); by doing extra annotations (Rimell and Clark, 2008); or by augmenting the model with unsupervised word cluster or word embedding features (Koo et al., 2008; Andreas and Klein, 2014). Being orthogonal to such methods, I have shown supertagging accuracy is still a major bottleneck for the C&C parser, and attacking this bottleneck further is a more direct and highly effective approach, on both in- and out-of-domain data sets, largely outperforming competitive techniques (Rimell and Clark, 2008; Honnibal et al., 2009; Kuhlmann et al., 2010).

**Optimizing for Task-specific Metrics.** The XF1 training objective is largely inspired by task-specific optimization for parsing and MT. Goodman (1996) proposed algorithms for optimizing a parser for various constituent matching criteria, and it was one of the earliest works which optimizes a parser for evaluation metrics. Smith and Eisner (2006) proposed a framework for minimizing expected loss for log-linear models and applied it to dependency parsing by optimizing for labeled attachment scores. Auli and Lopez (2011b) optimized the C&C parser for F-measure. However, they used the softmax-margin (Gimpel and Smith, 2010) objective, which required decomposing precision and recall statistics over parse forests. Instead, I directly optimize for an F-measure loss. In MT, task-specific optimization has also received much attention (e.g., see Och (2003)). Closely related to my work, Gao and He (2013) proposed training a Markov random field translation model as an additional component in a log-linear phrase-based translation system using a $k$-best list-based expected BLEU objective; using the same objective, Auli et al. (2014) and Auli and Gao (2014) trained a large scale phrase-based reordering model and a RNN language model respectively, all as additional components within a log-linear translation model. In contrast, my RNN parsing model is trained in an end-to-end fashion with an expected F-measure loss and all parameters of the model are optimized using backpropagation and SGD.

**Parsing with RNNs.** A line of work is devoted to parsing with RNN models, including using RNNs (Miikkulainen, 1996; Mayberry and Miikkulainen, 1999; Legrand and Collobert, 2015; Watanabe and Sumita, 2015) and LSTMs (Vinyals et al., 2015; Ballesteros et al., 2015; Dyer et al., 2015; Kiperwasser and Goldberg, 2016). Legrand and Collobert (2015) used RNNs to learn joint tagging and parsing models; Vinyals et al. (2015) explored sequence-to-sequence learning (Sutskever et al., 2014) for parsing; Ballesteros et al. (2015) utilized character-level representations and Kiperwasser and Goldberg (2016) built an easy-first dependency parser using tree-structured compositional LSTMs. However, all these parsers use greedy search and are trained using the maximum likelihood criterion (except Kiperwasser and Goldberg (2016), who used a margin-based objective). For learning global models, Watanabe and Sumita (2015) used a margin-based objective, which was not optimized for the evaluation metric; although not using RNNs, Weiss et al. (2015) proposed a method based on the structured perceptron (Collins, 2002; Collins and Roark, 2004; Zhang and Clark, 2008), which required fixing the neural network representations, and thus their model parameters were not learned using end-to-end backpropagation. Different from all aforementioned works which all consider discriminative modelling, Dyer et al. (2016) recently proposed a generative LSTM parser for constituency parsing, which showed state-of-the-art results.

Finally, in line with the present work, a number of recent works have also independently explored training neural network models for parsing and other tasks to tackle label bias (Ranzato et al., 2016; Wiseman and Rush, 2016), loss-evaluation mismatch (Ranzato et al., 2016; Wiseman and Rush, 2016), and additionally exposure bias (Bengio et al., 2015; Vaswani and Sagae, 2016; Ballesteros et al., 2016); the benefits of moving beyond fully locally normalized models have already been repeatedly demonstrated.

## 4.9 Summary

Neural network shift-reduce parsers are often trained by maximizing the log-likelihood of isolated shift-reduce actions, which is susceptible to the label bias problem and does not optimize towards the final evaluation metric. As the main contribution of this chapter, I addressed this by proposing expected F-measure training for shift-reduce parsing with RNNs. I have demonstrated the efficacy of the method on shift-reduce parsing for CCG, achieving higher accuracies than all previous shift-reduce CCG parsers and the C&C parser.[10] In the next chapter, I show how this training framework is applied to a different neural network architecture.

[10] Auli and Lopez (2011b) present higher accuracies but on a different coverage to enable a comparison to Fowler and Penn (2010). Their results are thus not directly comparable.

# Chapter 5

# LSTM Shift-Reduce CCG Parsing

Recent work on using recurrent neural networks for shift-reduce parsing has utilized both Elman RNNs and LSTMs to derive explicit representations for shift-reduce actions and parser states, and it has been shown that recursive tree-structured networks are well-suited to model the incremental process that derives syntactic structures in a shift-reduce parser (Dyer et al., 2015; Watanabe and Sumita, 2015). One of the main assumptions underlying these systems is that representations learned by tree-structured networks better mirror the hierarchical nature of syntax (Chomsky, 1957) and the linguistic principle of compositionality (Frege, 1892).

In keeping with such motivations, I define a model to learn representations for the complete derivation process. However, departing from tree-structured networks, I encode tree structures implicitly with multiple sequential LSTMs, which allow the incremental linearization of the complete derivation history, with no feature engineering (Zhang and Clark, 2011a; Xu et al., 2014), no atomic feature sets (Chen and Manning, 2014), and without relying on any additional control operations (Dyer et al., 2015) or explicit recursive structures (Goller and Kuchler, 1996; Socher et al., 2010; Socher et al., 2011; Socher et al., 2013).

In conjunction with two different training objectives, I obtain state-of-the-art results by combining the model with an attention-based supertagger in i) a locally normalized greedy parser; ii) a globally normalized beam-search parser; and iii) a globally normalized greedy parser. In each case, the resulting parser is sensitive to all aspects of the parsing history, including arbitrary positions in the input (encoded by a bidirectional LSTM), at any step during the parsing process.

## 5.1 Background

### 5.1.1 LSTM

Simple RNNs suffer from the vanishing gradient problem (Bengio et al., 1994; Hochreiter, 1998). Informally this means that BPTT (§4.1.4) could make the gradients negligibly small, diminishing the effects of backpropagated errors on temporal events further apart and making modelling long-range dependencies more difficult. For the Elman RNN (Elman, 1990), the source of this problem is the scaling effect of the hidden state activation derivatives as the backpropagated errors pass through the unrolled network. Because if the derivatives are sufficiently small, such as in the case of sigmoid activation function $\sigma$ with $0 < \sigma' \leq 0.25$, the gradients could decrease exponentially in the number of time steps. One of the main motivations of LSTMs is to mitigate this, thereby increasing sensitivity over longer time delays and allowing dependencies among temporal events that are arbitrarily far in a sequence to be captured.

Recall that an Elman RNN is factored into an input layer $\mathbf{x}_t$ and a hidden state (layer) $\mathbf{h}_t$ with recurrent connections, and it can be represented by the recurrence:

$$\mathbf{h}_t = \Phi(\mathbf{x}_t, \mathbf{h}_{t-1}),$$

where $\mathbf{x_t}$ is the current input, $\mathbf{h}_{t-1}$ is the previous hidden state and $\Phi$ is a set of affine transformations coupled with nonlinear activations. In LSTMs, the recurrence becomes

$$\mathbf{h}_t, \mathbf{c}_t = \Phi(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{c}_{t-1}), \tag{5.1}$$

where $\mathbf{c}_t$ is the cell state. Intuitively, the cell state functions as an explicit memory, and the parameters controlling how and when information is “written to” or “read from” it are learned during training. Specifically, these parameters are modelled as *gates* to regulate information flow and enable controlled “reading” and “writing”.

This is different from Elman RNNs, which use the complete hidden state from the previous step ("reading") and the complete input from the current step ("writing") to fully determine the hidden state at the current step. In other words, "reading" and "writing" are uncontrolled in an Elman RNN but controlled and adaptive in LSTMs.

A typical LSTM contains three gates, namely the forget ($\mathbf{f}_t$), input ($\mathbf{i}_t$) and output ($\mathbf{o}_t$) gates. Each gate is modelled as a single-layer feed-forward network. Concretely, the forget gate acts as a reset switch, determining how much information is "erased" from the previous cell state; the input gate determines how much information from the current input is used to update the cell state $\mathbf{c}_t$; and the output gate "filters" the cell state for the next time step. All three gates learn to condition on the current input $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$, and together with the cell state, a typical LSTM can be formulated as (Hochreiter and Schmidhuber, 1997):

$$\begin{aligned}
\mathbf{i}_t &= \sigma(\mathbf{W}_{ix}\mathbf{x}_t + \mathbf{W}_{ih}\mathbf{h}_{t-1} + \mathbf{b}_i)\\
\mathbf{f}_t &= \sigma(\mathbf{W}_{fx}\mathbf{x}_t + \mathbf{W}_{fh}\mathbf{h}_{t-1} + \mathbf{b}_f)\\
\hat{\mathbf{c}}_t &= \tanh(\mathbf{W}_{cx}\mathbf{x}_t + \mathbf{W}_{ch}\mathbf{h}_{t-1} + \mathbf{b}_c)\\
\mathbf{o}_t &= \sigma(\mathbf{W}_{ox}\mathbf{x}_t + \mathbf{W}_{oh}\mathbf{h}_{t-1} + \mathbf{b}_o)\\
\mathbf{c}_t &= \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \hat{\mathbf{c}}_t\\
\mathbf{h}_t &= \mathbf{o}_t \circ \tanh(\mathbf{c}_t),
\end{aligned}$$

where $\sigma$ is the sigmoid activation, and $\circ$ is the element-wise product.

In addition to the gates, the identity activation function of the cell state is also highly relevant in tackling vanishing gradients, as any error passing through is preserved due to its unity derivative. It has also been shown empirically that the forget gate plays a significant role in an LSTM for a number of tasks except language modelling; while the output gate is the least important, and can even be removed with little impact on the performance of an LSTM (Greff et al., 2016).

### 5.1.2 LSTM Backpropagation

For completeness, I derive the gradients for LSTM backpropagation. First, I rewrite the above LSTM formulation by arranging the first four equations into a matrix form (omitting the bias terms and keeping the last two equations unchanged):

$$\begin{pmatrix} \mathbf{i}_t \\ \mathbf{f}_t \\ \hat{\mathbf{c}}_t \\ \mathbf{o}_t \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \tanh \\ \sigma \end{pmatrix} \begin{pmatrix} \mathbf{w}_{ix}\mathbf{w}_{ih} \\ \mathbf{w}_{fx}\mathbf{w}_{fh} \\ \mathbf{w}_{cx}\mathbf{w}_{ch} \\ \mathbf{w}_{ox}\mathbf{w}_{oh} \end{pmatrix} \begin{pmatrix} \mathbf{x}_t \\ \mathbf{h}_{t-1} \end{pmatrix}$$

$$\begin{aligned} \mathbf{c}_t &= \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \hat{\mathbf{c}}_t \\ \mathbf{h}_t &= \mathbf{o}_t \circ \tanh(\mathbf{c}_t). \end{aligned}$$

Further, let the activations vector be $a$, where the activation functions are applied element-wise,[1] and split the block matrix of weights into a left block $\mathbf{U}_x$ (associated with $\mathbf{x}_t$) and a right block $\mathbf{U}_h$ (associated with $\mathbf{h}_{t-1}$), I rewrite the above as

$$\begin{aligned} \mathbf{g}_t &= a(\mathbf{U}\mathbf{b}_t) \\ &= a(\mathbf{U}_x\mathbf{x}_t + \mathbf{U}_h\mathbf{h}_{t-1}). \end{aligned}$$

Let $\boldsymbol{\delta}_{\mathbf{h}_t}$ be the error vector at $\mathbf{h}_t$, by applying BPTS (§4.1.3) and observing $\circ$ is commutative, the following error vectors can be derived:

$$\begin{aligned} \boldsymbol{\delta}_{\mathbf{o}_t} &= \tanh(\mathbf{c}_t) \circ \boldsymbol{\delta}_{\mathbf{h}_t}, \\ \boldsymbol{\delta}_{\mathbf{c}_t} &= \tanh'(\mathbf{c}_t) \circ \mathbf{o}_t \circ \boldsymbol{\delta}_{\mathbf{h}_t} + \boldsymbol{\delta}_{\mathbf{c}_{t+1}}, \\ \boldsymbol{\delta}_{\mathbf{i}_t} &= \hat{\mathbf{c}}_t \circ \boldsymbol{\delta}_{\mathbf{c}_t}, \\ \boldsymbol{\delta}_{\mathbf{f}_t} &= \mathbf{c}_{t-1} \circ \boldsymbol{\delta}_{\mathbf{c}_t}, \\ \boldsymbol{\delta}_{\hat{\mathbf{c}}_t} &= \mathbf{i}_t \circ \boldsymbol{\delta}_{\mathbf{c}_t}, \\ \boldsymbol{\delta}_{\mathbf{c}_{t-1}} &= \mathbf{f}_t \circ \boldsymbol{\delta}_{\mathbf{c}_t}. \end{aligned}$$

[1] This is a notation abuse.

Let

$$\boldsymbol{\delta}_{\mathbf{g}_t} = \begin{pmatrix} \boldsymbol{\delta}_{\mathbf{i}_t} \\ \boldsymbol{\delta}_{\mathbf{f}_t} \\ \boldsymbol{\delta}_{\hat{\mathbf{c}}_t} \\ \boldsymbol{\delta}_{\mathbf{o}_t} \end{pmatrix},$$

and $\boldsymbol{\delta}_{\mathbf{z}_t} = a'(\mathbf{U}\mathbf{b}_t) \circ \boldsymbol{\delta}_{\mathbf{g}_t}$; the following error vectors can be derived:

$$\boldsymbol{\delta}_{\mathbf{x}_t} = \mathbf{U}_x^\top \boldsymbol{\delta}_{\mathbf{z}_t},$$

$$\boldsymbol{\delta}_{\mathbf{h}_{t-1}} = \mathbf{U}_h^\top \boldsymbol{\delta}_{\mathbf{z}_t}.$$

Finally, the gradients for $\mathbf{U}_x$ and $\mathbf{U}_h$ are

$$\nabla_{\mathbf{U}_x} = \boldsymbol{\delta}_{\mathbf{z}_t} \mathbf{x}_t^\top,$$

$$\nabla_{\mathbf{U}_h} = \boldsymbol{\delta}_{\mathbf{z}_t} \mathbf{h}_{t-1}^\top.$$

## 5.2 LSTM Shift-Reduce Parsing

Several extensions to the vanilla LSTM have been proposed over time, each with a modified instantiation of $\Phi_\theta$ that exerts refined control over various elements in an LSTM (Gers et al., 2000; Gers and Schmidhuber, 2000). The instantiation for all LSTMs throughout this chapter is as follows:

$$\begin{aligned}
\mathbf{i}_t &= \sigma(\mathbf{W}_{ix}\mathbf{x}_t + \mathbf{W}_{ih}\mathbf{h}_{t-1} + \mathbf{W}_{ic}\mathbf{c}_{t-1} + \mathbf{b}_i) \\
\mathbf{f}_t &= \sigma(\mathbf{W}_{fx}\mathbf{x}_t + \mathbf{W}_{fh}\mathbf{h}_{t-1} + \mathbf{W}_{fc}\mathbf{c}_{t-1} + \mathbf{b}_f) \\
\mathbf{c}_t &= \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \tanh(\mathbf{W}_{cx}\mathbf{x}_t + \mathbf{W}_{ch}\mathbf{h}_{t-1} + \mathbf{b}_c) \\
\mathbf{o}_t &= \sigma(\mathbf{W}_{ox}\mathbf{x}_t + \mathbf{W}_{oh}\mathbf{h}_{t-1} + \mathbf{W}_{oc}\mathbf{c}_t + \mathbf{b}_o) \\
\mathbf{h}_t &= \mathbf{o}_t \circ \tanh(\mathbf{c}_t), \qquad (5.2)
\end{aligned}$$

which adopts the so-called “peephole connections” to allow the gates to look at the cell state, with the motivation of learning more precise timings of events (Gers and

Schmidhuber, 2000).

In addition to unidirectional LSTMs that model an input sequence $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}$ in a strict left-to-right order, I also use bidirectional LSTMs (BLSTMs; Graves and Schmidhuber, 2005), which read the input from both directions with two independent LSTMs. This is the same bidirectional architecture as in §4.5, with RNNs replaced with LSTMs. At each step, the forward hidden state $\mathbf{h}_t$ is computed using Eq. 5.1 for $t = (0, 1, \ldots, n-1)$; the backward hidden state $\hat{\mathbf{h}_t}$ is computed similarly but from the reverse direction for $t = (n-1, n-2, \ldots, 0)$. Together, the two hidden states at each step $t$ capture both past and future contexts, and the representation for each $\mathbf{x}_t$ is obtained as the concatenation $[\mathbf{h}_t; \hat{\mathbf{h}_t}]$.

### 5.2.1 Embeddings

The neural network model employed by Chen and Manning (2014), and followed by a number of other parsers (Weiss et al., 2015; Zhou et al., 2015; Ambati et al., 2016; Andor et al., 2016; Xu et al., 2016) allows higher-order feature conjunctions to be automatically discovered from a set of dense feature embeddings. However, as we have seen in the previous chapter, a set of atomic feature templates (Table 4.1, §4.4), which are only sensitive to contexts from the top few elements on the stack and queue are still needed to dictate the choice of these embeddings. Here I dispense with such templates and seek to design a model that is sensitive to both local and non-local contexts, on both the stack and queue.

Consequently, embeddings represent atomic input units that are added to the parser and are preserved throughout parsing. In total four types of embeddings are used, namely, word, CCG category, POS and action, where each has an associated look-up table. The look-up table for word embeddings is $\mathcal{L}_w \in \mathbb{R}^{k \times |w|}$, where $k$ is the embedding dimension and $|w|$ is the vocabulary size. Similarly, additional look-up tables are maintained for CCG categories, $\mathcal{L}_c \in \mathbb{R}^{l \times |c|}$, for the three types of actions, $\mathcal{L}_a \in \mathbb{R}^{m \times 3}$, and for POS tags, $\mathcal{L}_p \in \mathbb{R}^{n \times |p|}$.

input : $w_0 \dots w_{n-1}$
axiom : $0 : (0, \epsilon, \beta, \phi)$
goal : $2n - 1 + \mu : (n, \delta, \epsilon, \Delta)$

$$\frac{\omega : (j, \delta, x_{w_j}|\beta, \Delta)}{\omega + 1 : (j+1, \delta|x_{w_j}, \beta, \Delta)} \qquad (\textsc{shift}; 0 \leq j < n)$$

$$\frac{\omega : (j, \delta|s_1|s_0, \beta, \Delta)}{\omega + 1 : (j, \delta|x, \beta, \Delta \cup \langle x \rangle))} \qquad (\textsc{reduce}; s_1 s_0 \rightarrow x)$$

$$\frac{\omega : (j, \delta|s_0, \beta, \Delta)}{\omega + 1 : (j, \delta|x, \beta, \Delta)} \qquad (\textsc{unary}; s_0 \rightarrow x)$$

Figure 5-1: The shift-reduce deduction system.

### 5.2.2 Model

**Parser.** Fig. 5-1 reproduces the deduction system of shift-reduce CCG parsing (§2.3.1). Recall that each parser state is denoted as $(j, \delta, \beta, \Delta)$, where $j$ is the positional index of the word at the front of the queue, $\delta$ is the stack (with its top element $s_0$ to the right), and $\beta$ is the queue (with its top element $w_j$ to the left) and $\Delta$ is the set of CCG dependencies realized for the input consumed so far. Each state is also associated with a step indicator $t$, signifying the number of actions applied to it and the goal is reached in $2n-1+\mu$ steps, where $\mu$ is the total number of UNARY actions.

In the LSTM parser, I define each action as a 4-tuple $(\tau_t, c_t, w_{c_t}, p_{w_{c_t}})$, where $\tau_t \in \{\textsc{shift}, \textsc{reduce}, \textsc{unary}\}$ for $t \geq 1$, $c_t$ is the resulting category of $\tau_t$, and $w_{c_t}$ is the head word attached to $c_t$ with $p_{w_{c_t}}$ being its POS tag.[2]

**LSTM model.** LSTMs are designed to handle time-series data, in a purely sequential fashion, and I try to exploit this by completely linearizing all aspects of the parsing history. Concretely, I factor the model as five LSTMs, comprising four unidirectional ones, denoted as U, V, X and Y, and an additional BLSTM, denoted as W (Fig. 5-2). Before parsing each sentence, W is fed with the complete input (padded

[2] In case of multiple heads, I always choose the first one in the order they are created.

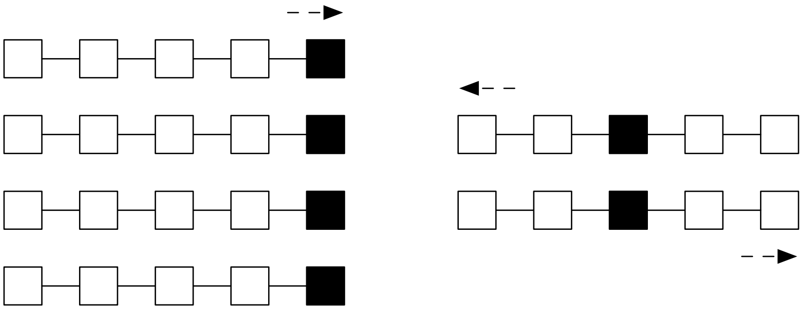


Figure 5-2: Example representation for a parser state at time step $t$, with the four unidirectional LSTMs (left) and the bidirectional LSTM (right). The shaded cells on the left represent $\boldsymbol{\delta}_t = [\mathbf{h}_t^{\mathrm{U}}; \mathbf{h}_t^{\mathrm{V}}; \mathbf{h}_t^{\mathrm{X}}; \mathbf{h}_t^{\mathrm{Y}}]$ (Eq. 5.3), and the shaded cells on the right represent $\mathbf{w}_j = [\mathbf{h}_j^{\mathrm{W}}; \hat{\mathbf{h}}_j^{\mathrm{W}}]$.

with a special embedding $\perp$ as the end of sentence token), and in subsequent steps, $\mathbf{w}_j = [\mathbf{h}_j^{\mathrm{W}}; \hat{\mathbf{h}}_j^{\mathrm{W}}]$ is used to represent $w_j$.[3] At initialization, a $\perp$ symbol is also added to the other four unidirectional LSTMs.

Given this 5-LSTM factorization, the stack representation for a parser state at step $t$, for $t \geq 1$, is obtained as

$$\boldsymbol{\delta}_t = [\mathbf{h}_t^{\mathrm{U}}; \mathbf{h}_t^{\mathrm{V}}; \mathbf{h}_t^{\mathrm{X}}; \mathbf{h}_t^{\mathrm{Y}}], \tag{5.3}$$

and together with $\mathbf{w}_j$, $[\boldsymbol{\delta}_t; \mathbf{w}_j]$ gives a representation for the complete parser state. For the initial state, it is represented as $[\boldsymbol{\delta}_0; \mathbf{w}_0]$, where $\boldsymbol{\delta}_0 = [\mathbf{h}_\perp^{\mathrm{U}}; \mathbf{h}_\perp^{\mathrm{V}}; \mathbf{h}_\perp^{\mathrm{X}}; \mathbf{h}_\perp^{\mathrm{Y}}]$.

During parsing, whenever the parser applies an action $(\tau_t, c_t, w_{c_t}, p_{w_{c_t}})$, the model is updated by adding the embedding of $\tau_t$, denoted as $\mathcal{L}_a(\tau_t)$, onto U, and also by adding the other three embeddings of the action 4-tuple, that is, $\mathcal{L}_c(c_t)$, $\mathcal{L}_w(w_{c_t})$ and $\mathcal{L}_p(p_{w_{c_t}})$, onto V, X and Y respectively.

To predict the next action, an action hidden layer $\mathbf{b}_t$ is first derived, by passing the parser state representation $[\boldsymbol{\delta}_{t-1}; \mathbf{w}_j]$ through an affine transformation and a nonlinearity, s.t.

$$\mathbf{b}_t = f(\mathbf{B}[\boldsymbol{\delta}_{t-1}; \mathbf{w}_j] + \mathbf{r}), \tag{5.4}$$

where $\mathbf{B}$ is a parameter matrix of the model, $\mathbf{r}$ is a bias vector and $f$ is a ReLU

[3] Word and POS embeddings are concatenated at each input position $j$, for $0 \leq j < n$; $\mathbf{w}_n = [\mathbf{h}_\perp^{\mathrm{W}}; \hat{\mathbf{h}}_\perp^{\mathrm{W}}]$.

nonlinearity (Nair and Hinton, 2010). Finally, another affine transformation (with $\mathbf{A}$ as the weights and $\mathbf{s}$ as the bias) and a nonlinearity is applied to $\mathbf{b}_t$:

$$\mathbf{a}_t = f(\mathbf{A}\mathbf{b}_t + \mathbf{s}),$$

before the probability of the $i$th action in $\mathbf{a}_t$ is obtained as

$$p(\tau_t^i|\mathbf{b}_t) = \frac{\exp\{\mathbf{a}_t^i\}}{\sum_{\tau_t^k \in \mathcal{T}(\langle\delta,\beta\rangle_{t-1})} \exp\{\mathbf{a}_t^k\}},$$

where $\mathcal{T}(\langle\delta,\beta\rangle_{t-1})$ is the set of feasible actions for $\langle\delta,\beta\rangle_{t-1}$, and $\tau_t^i \in \mathcal{T}(\langle\delta,\beta\rangle_{t-1})$.

### 5.2.3 Stack-LSTM, Derivations and Dependency Structures

My LSTM parser architecture is inspired by the stack-LSTM (Dyer et al., 2015), which is an extension of sequential LSTMs to allow the modelling of structures beyond a strict left-to-right order. The stack-LSTM has been used in the dependency parser of Dyer et al. (2015), for which it was crucial in allowing representations for the entire parsing history to be constructed in an incremental fashion.

In essence, a stack-LSTM has the same recurrence as its sequential counterpart (Eq. 5.1), but two additional control operations are provided to make it more flexible. The first of these two operations is PUSH, which operates in a similar fashion as in a sequential LSTM: a new LSTM cell is added by extending a "previous" LSTM cell. However, unlike in a sequential LSTM, this "previous" cell is not necessarily the rightmost, but can be any cell in the stack-LSTM. The second operation is POP, which allows "rewinding" and hence the selection of the "previous" LSTM cell that provides $\mathbf{c}_{t-1}$ and $\mathbf{h}_{t-1}$ when deriving Eq. 5.1. POP is a simple non-destructive operation, and it can also be understood as updating the stack-top pointer, maintained in a stack-LSTM, to indicate the position of the "previous" cell. Each POP rewinds the stack-top pointer back for one step, and like PUSH, multiple POP can be applied in succession. A few examples of these two operations are shown in Fig. 5-3.

To apply the stack-LSTM to arc-standard dependency parsing, Dyer et al. (2015) combine it with recursive neural representations of dependency trees. In doing so,

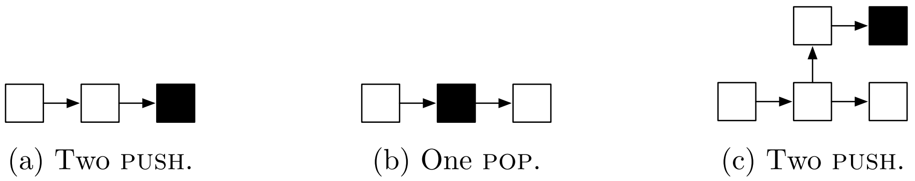

(a) Two PUSH.
(b) One POP.
(c) Two PUSH.

Figure 5-3: Example stack-LSTM operations. Five operations are applied consecutively; stack top is shaded.

they obtain parser state representations consisting of the complete summary of the contents on the parser stack, including (partial or complete) dependency trees. That is, the model exhibits "global" sensitivity to the parser state history at any given step in the parsing process (a property also found in the LSTM parser of this chapter). The key observation that allows them to achieve this is that transitions of the chosen arc-standard dependency parser (Nivre, 2004) can be mapped to isomorphic operations in a stack-LSTM. This mapping is straightforward for SHIFT transitions, and slightly more involved for LEFT-ARC and RIGHT-ARC. In the former case, one PUSH operation is sufficient, where the first word from the queue is appended to the stack-LSTM, extending it one step to the right. In the latter case, three stack-LSTM operations applied in succession are needed. This is because a word is first removed from the stack whenever a LEFT-ARC or RIGHT-ARC is applied in the parser; in the stack-LSTM, this corresponds to "removing" the top-two words by rewinding it two steps back (two POP), and pushing the result of composing the "removed" words back to the stack-LSTM (one PUSH). This three-operation process, in combination with projective arc-standard parsing, ensures that the representations for dependency trees can always be constructed in a recursive bottom-up fashion, as an integral part of a stack-LSTM that also learns representations for all other aspects of the parsing process. This ability to learn both non-recursive and recursive representations jointly is a notable feature of the stack-LSTM parser.

In contrast, the present LSTM parser uses a simpler architecture that is free from explicit recursive representations, and it naturally linearizes CCG derivations "incrementally" (Ambati et al., 2015) following their post-order traversals using four unidirectional LSTMs whose hidden state concatenation at each step represents an action 4-tuple (Eq. 5.3) in a CCG derivation. In line with most existing CCG parsing models, including dependency models, I have also chosen to model CCG derivations, rather than dependency structures.[4] This allows me to sidestep the issues that would arise from deriving explicit representations for CCG dependencies, due to the large amount of flexibility in how dependencies are realized in CCG (§2.1.4). The main motivation of this design is related to the parser deduction system, which explicitly regulates parser actions and implicitly defines an ordering for all the linearized elements on the four unidirectional LSTMs. Because of this, it is an open empirical question whether explicit tree-structured representations can further improve the model. As preliminary pieces of evidence showing the efficacy of the chosen design, the experiments below show that the presence of action embeddings helps the model very little, while the category embeddings, which linearize the derivations, have a significant impact on the performance of the parser (§5.4.2).

### 5.2.4 Training

As a baseline, I first train a locally normalized model, in which the log-likelihood of each target action in the training data is maximized as in the local RNN model (§4.3.2). More specifically, let $(\tau_1^g, \dots, \tau_{T_n}^g)$ be the gold standard action sequence for a training sentence $n$, a cross-entropy criterion is used to obtain the error gradients, and for each sentence, training involves minimizing

$$L(\theta) = -\log \prod_{t=1}^{T_n} p(\tau_t^g|\mathbf{b}_t) = -\sum_{t=1}^{T_n} \log p(\tau_t^g|\mathbf{b}_t),$$

where $\theta$ is the set of all parameters in the model.

As discussed in §4.3.2, locally normalized models suffer from the label bias problem, even with LSTMs.[5] Here, I extend the local LSTM model into a global one by adapting the expected F-measure training previously introduced for the RNN model (§4.3.3), which is briefly recapped below. To the best of my knowledge, this is the

[4] Most CCG dependency models (e.g., see Clark and Curran (2007) and Xu et al. (2014)) model CCG derivations with dependency features.

[5] The proof in Andor et al. (2016) does not directly apply to BLSTMs (see footnote 4 in §4.3.2).

first approach to train a global LSTM shift-reduce parser.

Let $\theta = \{\mathbf{U}, \mathbf{V}, \mathbf{X}, \mathbf{Y}, \mathbf{W}, \mathbf{B}, \mathbf{A}\}$ be the weights of the baseline greedy model,[6] the weights of the global model, which has the same architecture as the baseline, are initialized to $\theta$, and reoptimized in multiple training epochs as follows:

1. Pick a sentence $x_n$ from the training set, decode it with beam search, and generate a $k$-best list of output parses with the *current* $\theta$, denoted as $\Lambda_{x_n}$.[7]

2. For each parse $y_i$ in $\Lambda_{x_n}$), compute its sentence-level F1 using the set of dependencies in the $\Delta$ field of its parser state. In addition, let $|y_i|$ be the total number of actions that derived $y_i$ and $s_\theta(y_{ij})$ be the softmax action score of the $j$th action, given by the LSTM model. Compute the log-linear score of its action sequence as $\rho(y_i) = \sum_{j=1}^{|y_i|} \log s_\theta(y_{ij})$.

3. Compute the negative expected F1 objective (defined below) for $x_n$ and minimize it using SGD (maximizing expected F1). Repeat these three steps for the remaining sentences.

More formally, the loss $J(\theta)$, is defined as

$$\begin{aligned} J(\theta) &= -\mathrm{XF1}(\theta) \\ &= -\sum_{y_i \in \Lambda_{x_n}} p(y_i|\theta)\mathrm{F1}(\Delta_{y_i}, \Delta^G_{x_n}), \end{aligned}$$

where $\mathrm{F1}(\Delta_{y_i}, \Delta^G_{x_n})$ is the sentence level F1 of the parse derived by $y_i$, with respect to the gold standard dependency structure $\Delta^G_{x_n}$ of $x_n$; $p(y_i|\theta)$ is the normalized probability score of the action sequence $y_i$, computed as

$$p(y_i|\theta) = \frac{\exp\{\gamma\rho(y_i)\}}{\sum_{y \in \Lambda_{x_n}} \exp\{\gamma\rho(y)\}},$$

where $\gamma$ is a parameter that sharpens or flattens the distribution (Tromble et al., 2008).[8] Different from the maximum-likelihood objective, XF1 optimizes the model

[6] Boldface letters are used to designate LSTM weights, and bias terms are omitted for brevity.

[7] $k$ was not preset like in §4.3.3, and $k = 11.06$ on average with a beam size of 8.

[8] $\gamma = 1$ for all experiments.

on a sequence level and towards the final evaluation metric, by taking into account all action sequences in $\Lambda_{x_n}$.

## 5.3 Attention-Based LSTM Supertagging

In addition to the size of the label space, supertagging is difficult because CCG categories can encode long-range dependencies and tagging decisions frequently depend on non-local contexts. For example, in *He went to the zoo with a cat*, a possible category for *with*, $(S\backslash NP)\backslash(S\backslash NP)/NP$, depends on the word *went* further back in the sentence.

Recently a number of RNN models have been proposed for CCG supertagging (Xu et al., 2015; Lewis et al., 2016; Vaswani et al., 2016; Xu et al., 2016), and such models show dramatic improvements over non-recurrent models (Lewis and Steedman, 2014b). Although the underlying models differ in their exact architectures, all of them make each tagging decision using only the hidden states at the current input position, and this imposes a potential bottleneck in the model. To mitigate this, I generalize the attention mechanisms of Bahdanau et al. (2015) and Luong et al. (2015), and adapt them to supertagging, by allowing the model to explicitly use hidden states from more than one input positions for tagging each word. Similar to Bahdanau et al. (2015) and Luong et al. (2015), a key feature in the model is a soft alignment vector that weights the relative importance of the considered hidden states.

For an input sentence $w_0, w_1, \ldots, w_{n-1}$, $\mathbf{w}_t = [\mathbf{h}_t; \hat{\mathbf{h}}_t]$ (Eq. 5.2) is used as the representation for the $t$th word ($0 \leq t < n$, $\mathbf{w}_t \in \mathbb{R}^{2d\times 1}$), given by a BLSTM with a hidden state size $d$ for both its forward and backward layers.[9] Let $k$ be a context window size hyperparameter, define $\mathbf{H}_t \in \mathbb{R}^{2d\times(k-1)}$ as

$$\mathbf{H}_t = [\mathbf{w}_{t-\lfloor k/2\rfloor}, \ldots, \mathbf{w}_{t-1}, \mathbf{w}_{t+1}, \ldots, \mathbf{w}_{t+\lfloor k/2\rfloor}],$$

which contains representations for all words in the size $k$ window except $\mathbf{w}_t$. At each position $t$, the attention model derives a context vector $\mathbf{c}_t \in \mathbb{R}^{2d\times 1}$ (defined below)

[9] Unlike in the parsing model, POS tags are excluded.

from $\mathbf{H}_t$, which is used in conjunction with $\mathbf{w}_t$ to produce an attentional hidden layer:

$$\mathbf{x}_t = f(\mathbf{M}[\mathbf{c}_t; \mathbf{w}_t] + \mathbf{m}), \tag{5.5}$$

where $f$ is a ReLU nonlinearity, $\mathbf{M} \in \mathbb{R}^{g \times 4d}$ is a learned weight matrix, $\mathbf{m}$ is a bias term, and $g$ is the size of $\mathbf{x}_t$. Then $\mathbf{x}_t$ is used to produce another hidden layer (with $\mathbf{N}$ as the weights and $\mathbf{n}$ as the bias):

$$\mathbf{z}_t = \mathbf{N}\mathbf{x}_t + \mathbf{n},$$

and the predictive distribution over categories is obtained by feeding $\mathbf{z}_t$ through a softmax activation.

In order to derive the context vector $\mathbf{c}_t$, I first compute $\mathbf{b}_t \in \mathbb{R}^{(k-1)\times 1}$ from $\mathbf{H}_t$ and $\mathbf{w}_t$ using $\boldsymbol{\alpha} \in \mathbb{R}^{1\times 4d}$, s.t. the $i$th entry in $\mathbf{b}_t$ is

$$\mathbf{b}_t^i = \boldsymbol{\alpha}[\mathbf{w}_{T[i]}; \mathbf{w}_t],$$

for $i \in [0, k-1)$, $T = [t - \lfloor k/2 \rfloor, \ldots, t-1, t+1, \ldots, t + \lfloor k/2 \rfloor]$; then $\mathbf{c}_t$ is derived as:

$$\begin{aligned}\mathbf{a}_t &= \text{softmax}(\mathbf{b}_t),\\ \mathbf{c}_t &= \mathbf{H}_t\mathbf{a}_t,\end{aligned}$$

where $\mathbf{a}_t$ is the alignment vector. I also experiment with two types of attention reminiscent of the *global* and *local* models in Luong et al. (2015), where the first attends over all input words ($k = n$) and the second over a local window.

It is worth noting that two other works have concurrently tackled supertagging with BLSTM models. In Vaswani et al. (2016), a language model layer is added on top of a BLSTM, which allows embeddings of previously predicted tags to propagate through and influence the pending tagging decision. However, the language model layer is only effective when both scheduled sampling for training (Bengio et al., 2015) and beam search for inference are used. I show my attention-based models can match their performance, with only standard training and greedy decoding. Additionally, Lewis et al. (2016) presented a BLSTM model with two layers of stacking in each direction; as an internal baseline, I show a non-stacking BLSTM without attention can achieve the same accuracy.

## 5.4 Experiments

**Baselines.** For supertagging, the baselines are the RNN and BRNN models from the previous chapter, and the BLSTM models in Vaswani et al. (2016) and Lewis et al. (2016). For parsing, the same baselines from the previous two chapters are included. Additionally, I compared with the RNN shift-reduce models (§4).

**Model and training parameters.**[10] All LSTM models are non-stacking with a single layer.[11] For the supertagging models, the LSTM hidden state size is 256, and the size of the attentional hidden layer is 200 ($\mathbf{x}_t$, Eq. 5.5). All parsing model LSTMs have a hidden state size of 128, and action hidden layer size is 80 ($\mathbf{b}_t$, Eq. 5.4).

Pretrained word embeddings for all models are 100-dimensional (Turian et al., 2010), and all other embeddings are 50-dimensional. Like for the RNN models, I also pretrained CCG lexical category and POS embeddings on the concatenation of the training data and a Wikipedia dump parsed with C&C (§4.7). All other parameters were uniformly initialized in $\pm\sqrt{6/(r+c)}$, where $r$ and $c$ are the number of rows and columns of a matrix (Glorot and Bengio, 2010).

For training, I used plain non-minibatched SGD with an initial learning rate $\eta_0 = 0.1$ and training was stopped when accuracy no longer increases on the dev set. For all models, a learning rate schedule $\eta_e = \eta_0/(1 + \lambda e)$ with $\lambda = 0.08$ was used for $e \geq 11$. Gradients were clipped whenever their norm exceeds 5. Dropout training as suggested by Zaremba et al. (2014), with a dropout rate of 0.3, and an $\ell_2$ penalty of $1.00 \times 10^{-5}$, were applied to all models.

Like in the previous two chapters, for training the parsing models, 10-fold jack-

[10] All models in this chapter are implemented with the CNN toolkit: `https://github.com/clab/cnn`.

[11] The BLSTMs have a single layer in each direction. I experimented with 2 layers in all models during development and found negligible improvements.

|  | Dev | Test |
|---|---|---|
| C&C | 91.50 | 92.02 |
| RNN | 93.07 | 93.00 |
| BRNN | 93.49 | 93.52 |
| Lewis et al. (2016) | 94.1 | 94.3 |
| Vaswani et al. (2016) | 94.08 | - |
| Vaswani et al. (2016) +LM +beam | 94.24 | 94.50 |
| BLSTM | 94.11 | 94.29 |
| BLSTM-LOCAL | **94.31** | **94.46** |
| BLSTM-GLOBAL | 94.22 | 94.42 |

Table 5.1: 1-best supertagging results on both the dev and test sets. BLSTM is the baseline model without attention; BLSTM-LOCAL and -GLOBAL are the two attention-based models.

knifing was used for both POS tagging and supertagging. For training only, if the gold standard lexical category is not supplied by the supertagger for a word, it is added to its list of categories.

### 5.4.1 Supertagging Results

Table 5.1 summarizes 1-best supertagging results. The baseline BLSTM model without attention achieves the same level of accuracy as Lewis et al. (2016) and the baseline BLSTM model of Vaswani et al. (2016), with a hidden state 50% smaller than the latter (256 vs. 512).

For training and testing the local attention model (BLSTM-LOCAL), an attention window size of 5 was used (tuned on the dev set), and it gives an improvement of 0.94% over the BRNN supertagger (§4.5), achieving an accuracy on par with the beam-search (size 12) model of Vaswani et al. (2016) that is enhanced with a language model. Despite being able to consider wider contexts than the local model, the global attention model (BLSTM-GLOBAL) did not show further gains, hence BLSTM-LOCAL was used for all parsing experiments below.

### 5.4.2 Parsing Results

**The XENT model.** For the locally normalized cross-entropy model, the same as the RNN-XENT model (§4.7.2) I found using a small $\beta$ value (bigger ambiguity) for

| | 0.09 | 0.07 | 0.06 | 0.01 | 0.001 |
|---|---|---|---|---|---|
| b = 1 | 86.49 | 86.52 | 86.56 | 86.26 | 85.80 |
| b = 2 | 86.55 | 86.58 | 86.63 | 86.39 | 86.01 |
| b = 8 | 86.61 | 86.64 | **86.67** | 86.40 | 86.07 |

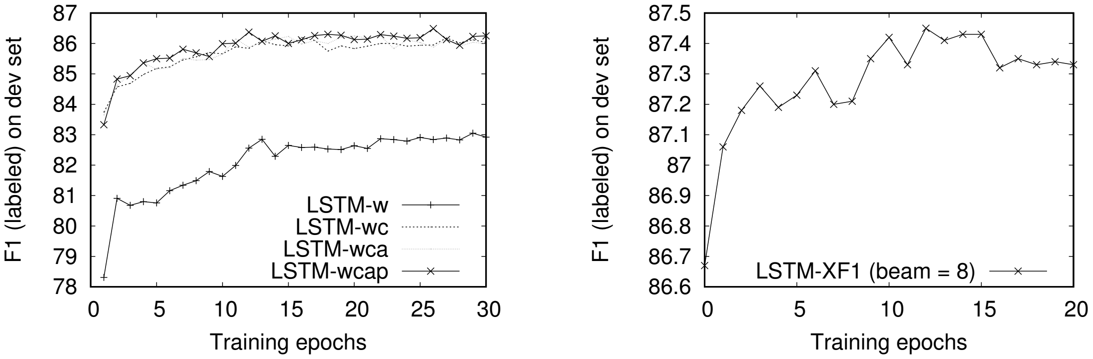
Table 5.2: The effect on dev F1 by varying the beam size (b) and supertagger $\beta$ value for the LSTM-XENT model.

| | beam | LP | LR | LF | SENT | CAT |
|---|---|---|---|---|---|---|
| C&C (normal) | - | 85.18 | 82.53 | 83.83 | 31.42 | 92.39 |
| C&C (hybrid) | - | 86.07 | 82.77 | 84.39 | 32.62 | 92.57 |
| Zhang and Clark (2011a) | 16 | 87.15 | 82.95 | 85.00 | 33.82 | 92.77 |
| Zhang and Clark (2011a)* | 16 | 86.76 | 83.15 | 84.92 | 33.72 | 92.64 |
| SHIFT-REDUCE-DEP | 128 | 86.29 | 84.09 | 85.18 | 34.40 | 92.75 |
| Ambati et al. (2016)† | 1 | - | - | 82.65 | - | 91.72 |
| Ambati et al. (2016)‡ | 16 | - | - | 85.69 | - | 93.02 |
| RNN-XENT | 1 | 88.12 | 81.38 | 84.61 | 33.82 | 93.42 |
| RNN-XF1 | 8 | 88.20 | 83.40 | 85.73 | 34.97 | 93.56 |
| LSTM-XENT | 1 | 89.43 | 83.86 | 86.56 | **48.98** | **94.47** |
| LSTM-XF1 | 1 | **89.68** | 85.29 | 87.43 | 48.09 | 94.41 |
| LSTM-XF1 | 8 | 89.54 | **85.46** | **87.45** | 47.99 | 94.39 |

Table 5.3: Parsing results on Section 00 (100% coverage and auto POS), with all LSTM models using the BLSTM-LOCAL supertagging model. All LSTM parsers are the full model with all four types of embeddings. $*$ = reimplementation. † = cross-entropy; ‡ = cross-entropy + structured perceptron.

training significantly improved accuracy, and $\beta = 1.00 \times 10^{-5}$ for training (with an ambiguity of 5.22 after jackknifing) and $\beta = 0.06$ (with an ambiguity of 1.09) for testing were chosen via development experiments (Table 5.2). Achieving an F1 of 86.56% on the dev set, this model surpasses all previous shift-reduce models (LSTM-XENT, Table 5.3).

Table 5.4 shows elaborated dev set results for the same model, where the four types of embeddings, that is, word (w), CCG category (c), action (a) and POS (p), were gradually introduced to study their individual contribution. As can be seen, category embeddings (LSTM-w+c) yielded a large gain over using word embeddings alone (LSTM-w), and action embeddings (LSTM-w+c+a) provided little improvement, but further adding POS embeddings (LSTM-w+c+a+p) gave noticeable recall (+0.61%)



(a) (b)

Figure 5-4: Learning curves for the cross-entropy models (a) and the XF1 model, with a beam size of 8 (b).

| | LP | LR | LF | CAT |
|---|---|---|---|---|
| Lstm-w | **90.13** | 76.99 | 83.05 | 94.24 |
| Lstm-w+c | 89.37 | 83.25 | 86.20 | 94.34 |
| Lstm-w+c+a | 89.31 | 83.39 | 86.25 | 94.38 |
| Lstm-w+c+a+p | 89.43 | **83.86** | **86.56** | **94.47** |

Table 5.4: F1 on dev for all the cross-entropy models.

and F1 improvements (+0.36%) over Lstm-w+c.

Fig. 5-4a shows the learning curves for all cross-entropy models, where all of them converged in under 30 epochs.

**The XF1 model.** With a beam size of 8, training took 12 epochs to converge (Fig. 5-4b), and an F1 of 87.45% on the dev set was obtained (Table 5.3). On the test set, the final F1 is 87.76%, which improves over the beam-search Rnn-XF1 model by 1.34% (Table 5.5).

Notably, decoding the XF1 model with greedy inference only slightly decreased recall and F1, and this resulted in a highly accurate greedy parser (Lstm-XF1, beam = 1, Table 5.5), outperforming the Rnn-Xent model (§4.7.2) by 2.67% F1.

**Effect of the supertagger.** To isolate the parsing model from the supertagging model, I first experimented with the BRNN supertagging model (§4.5) for both training and testing the Lstm-Xent parser with greedy inference (Table 5.6). Using this supertagger, the highest F1 (85.86%) was still achieved on the dev set (Lstm-

| | beam | LP | LR | LF | SENT | CAT | Speed |
|---|---|---|---|---|---|---|---|
| c&c (normal) | - | 85.45 | 83.97 | 84.70 | 32.82 | 92.83 | 97.90 |
| c&c (hybrid) | - | 86.24 | 84.17 | 85.19 | 33.24 | 93.00 | 95.25 |
| Zhang and Clark (2011a) | 16 | 87.43 | 83.61 | 85.48 | 35.19 | 93.12 | - |
| Zhang and Clark (2011a)* | 16 | 87.04 | 84.14 | 85.56 | 34.98 | 92.95 | 49.54 |
| Shift-Reduce-Dep | 128 | 87.03 | 85.08 | 86.04 | 35.69 | 93.10 | 12.85 |
| Ambati et al. (2016)[†] | 1 | - | - | 83.27 | - | 91.89 | **350.00** |
| Ambati et al. (2016)[‡] | 16 | - | - | 85.57 | - | 92.86 | - |
| Rnn-Xent | 1 | 88.53 | 81.65 | 84.95 | 32.99 | 93.57 | 337.45 |
| Rnn-XF1 | 8 | 88.74 | 84.22 | 86.42 | 34.73 | 93.87 | 67.65 |
| Lstm-Xent | 1 | 89.75 | 84.10 | 86.83 | **49.94** | **94.63** | 40.76 |
| Lstm-XF1 | 1 | **89.85** | 85.51 | 87.62 | 48.94 | 94.53 | 38.60 |
| Lstm-XF1 | 8 | 89.81 | **85.81** | **87.76** | 49.07 | 94.57 | 10.40 |

Table 5.5: Parsing results on Section 23 (100% coverage and auto POS), with all LSTM models using the BLstm-local supertagging model. All LSTM parsers are the full model with all four types of embeddings. $*$ = reimplementation. † = cross-entropy; ‡ = cross-entropy + structured perceptron. Speed measured on an Intel i7-4790K CPU (sents/sec; result for Ambati et al. (2016) is taken from the paper).

| | Dev | Test |
|---|---|---|
| Lstm-BRnn | 85.86 | 86.37 |
| Lstm-BLstm | 86.26 | 86.64 |
| Lstm-Xent | **86.56** | **86.83** |

Table 5.6: The effect of different supertaggers on the full greedy parser. LSTM-Xent is the same parser as in Table 5.3 and 5.5, which uses the BLstm-local supertagger.

BRnn) in comparison with all previous shift-reduce models; moreover, an improvement of 1.42% F1 over the Rnn-Xent model (§4.7.2) was obtained on the test set (Table 5.5). I then experimented with using the baseline BLSTM supertagging model for parsing (LSTM-BLstm), and observed the attention-based setup (Lstm-Xent) outperformed it, despite the attention-based supertagger BLstm-local not giving better overall multi-tagging accuracy. I owe this to the fact that larger $\beta$ cutoff values—resulting in almost deterministic supertagging decisions on average—were more beneficial for greedy inference.[12]

[12] All $\beta$ cutoffs were tuned on the dev set; for Brnn, the same $\beta$ settings as in §4.5 were found to be optimal; for BLSTM, $\beta = 4 \times 10^{-5}$ for training (with an ambiguity of 5.27) and $\beta = 0.02$ for testing (with an ambiguity of 1.17).

| | LP | LR | LF |
|---|---|---|---|
| C&C+RNN | 87.68 | 86.41 | 87.04 |
| Lewis et al. (2016) | 87.7 | 86.7 | 87.2 |
| Lewis et al. (2016)$^\star$ | 88.6 | 87.5 | 88.1 |
| Vaswani et al. (2016)$^*$ | - | - | 88.32 |
| Lee et al. (2016) | - | - | **88.7** |
| LSTM-XF1 (beam = 1) | 89.85 | 85.51 | 87.62 |
| LSTM-XF1 (beam = 8) | 89.81 | 85.81 | 87.76 |

Table 5.7: Comparison of the XF1 models with chart-based parsers on the test set. $\star$ = tri-training using external data; $*$ = a different POS tagger.

**Comparison with chart-based models.** For completeness and to put the results in perspective, I compared the XF1 models with other CCG parsers in the literature (Table 5.7): C&C+RNN is the log-linear C&C dependency hybrid model with an RNN supertagger front-end (§4.6.2); Lewis et al. (2016) is an LSTM supertagger-factored parser using the A$^*$ CCG parsing algorithm of Lewis and Steedman (2014a); Vaswani et al. (2016) combine a BLSTM supertagger with the recently developed JAVA version of the C&C parser (Clark et al., 2015) that uses a max-violation structured perceptron, which significantly improves over the original C&C models; and finally, a global recursive neural network model with A$^*$ decoding (Lee et al., 2016). Note that all these alternative models—with the exception of C&C+RNN and Lewis et al. (2016)—use structured learning that accounts for violations of the gold standard, which could potentially provide further improvements for the LSTM shift-reduce models.[13]

## 5.5 Summary

The XF1 training framework developed in the previous chapter was applied to an LSTM architecture with a factorization allowing the incremental linearization of the complete parsing history. It has also been demonstrated that global normalization benefits an LSTM shift-reduce model, and contrary to the structured perceptron global shift-reduce CCG models (Zhang and Clark, 2011a; Xu et al., 2014), beam-

[13] XF1 training considers shift-reduce action sequences, but not violations of the gold standard (e.g., see Huang et al. (2012), Watanabe and Sumita (2015), Zhou et al. (2015) and Andor et al. (2016)).

search inference was shown to be unnecessary, which can be partly attributed to the relatively high 1-best accuracy of the LSTM supertagger. For future work, a natural direction is to explore integrated supertagging and parsing in a single neural model.

# Chapter 6

# Conclusion

> “It doesn’t matter how beautiful your theory is, it doesn’t matter how smart you are. If it disagrees with experiment, it’s wrong.”
>
> —Richard P. Feynman

> “If your intuitions are good, you should follow them and you’ll eventually be successful. If your intuitions are not good, it doesn’t matter what you do.”
>
> —Geoffrey E. Hinton

Statistical shift-reduce parsing poses a number of challenges. It is a representation learning challenge, usually boiled down to learning richer representations for various aspects of parser states. It is a structured learning challenge, demanding models to take into account the structural properties of the output. These two interrelated and orthogonal challenges are further compounded by inexact search, and together they constitute three essential elements in any shift-reduce model. While improving each element independently has shed light on their individual significance, approaches that treat them holistically have shown their merits.

The holistic approach is the one that has been considered in this study, which began with the question: Should we model the derivations, the dependencies, or both, for shift-reduce CCG parsing? The resulting algorithmic and structured learning issues that arose in the context of the linear structured perceptron dependency model were addressed by a novel dependency oracle and by extending and generalizing provably correct theories marrying the structured perceptron and inexact search (Huang et al., 2012), while at the same time preserving and capitalizing on, rather than interfering with, the strengths of the normal-form shift-reduce CCG model (Zhang and Clark, 2011a), such as global structured learning and its rich feature sets (§3).

Drawing on recent work on using feed-forward neural networks for learning feature representations for parser states (Chen and Manning, 2014), I then described a framework for training RNN shift-reduce parsing models optimized for a task-specific loss based on expected F-measure (§4). Being agnostic to the underlying neural network architecture, this framework was also applied to an LSTM parser inspired by the stack-LSTM of Dyer et al. (2015) (§5). In both cases, the models were also globally normalized and the three aforementioned essential elements were tightly integrated.

Empirically, extensive experiments were performed throughout, and results were state-of-the-art. Clearly, however, the present study is far from complete.

First, all shift-reduce parsing models introduced still required a separate supertagging model, which has a large impact on the final parsing accuracy. Further conjoined with the POS tagging model, this pipelined approach has been dominating in CCG parsing for over a decade. But with the flexibility provided by neural networks, and as suggested by some recent works (Zhang and Weiss, 2016; Søgaard and Goldberg, 2016), it is reasonable to expect that an end-to-end neural model for CCG parsing can be derived, ideally even without being confined to a specific parsing paradigm, either chart-based or shift-reduce.

Second, I have just barely scratched the surface of investigating structured learning for neural shift-reduce models. Combining structured learning with deep learning is an emerging theme, and the same intuition applies to the models presented above, especially in devising both empirical and formal methods that faithfully take into account the respective neural models, instead of relying upon techniques originally developed for other models (Watanabe and Sumita, 2015; Weiss et al., 2015; Andor et al., 2016). Moreover, how to integrate such methods with a framework like expected F-measure training is of particular interest.

Aside from possible extensions, however, it is debatable under what guiding principles should the present study be further extended.

Statistical parsing is a well-defined research area that has attracted a lot of attention in computational linguistics, and it has generated a set of techniques many of which have been used for or adapted to other tasks (Wu, 1997; Chelba and Jelinek, 1998; Goodman, 1999; Ramshaw et al., 2001; Collins and Roark, 2004; Huang and Chiang, 2005; Zettlemoyer and Collins, 2005; Chiang, 2007; Dyer, 2010). Statistical parsers (Collins, 1997; Klein and Manning, 2003; Briscoe et al., 2006; McDonald, 2006; Nivre et al., 2006; Curran et al., 2007), on the other hand, have mainly played the role of feature generators, producing annotations shown to be useful in various settings (Yamada and Knight, 2001; Chiang et al., 2008; Marton and Resnik, 2008; Mi et al., 2008; Chiang et al., 2009; Dyer and Resnik, 2010). In addition to being used in such capacities, however, it is apparent that their future role is likely to be limited in end-to-end language processing approaches, and in the endeavour to achieve automated human-level language understanding—beyond formalism-dependent parsing and natural language text processing—which remains elusive with currently available language technologies.

As a related issue, the practical implications for formalisms like CCG also calls for revisiting, with one notable reason being that alternatives such as dependency grammars have usually demonstrated their superior simplicity and cross-lingual scalability that are preferred in production systems (McDonald et al., 2013; Andor et al., 2016), which put more emphasis on empirical results rather than the linguistic properties of a grammar.

But more importantly on a higher level, it might be illuminating to ask: How useful is syntax for language understanding, is it relevant at all?

Overall, this thesis does not intend to serve as a proponent for parsing nor CCG. Instead, it purposes itself as an exploration of structured learning with inexact search—in the context of shift-reduce CCG parsing. It is hoped that future work will continue in this spirit.

## References

Anthony Ades and Mark Steedman. 1982. On the order of words. In *Linguistics and philosophy*. Springer.

Alfred Aho and Jeffrey Ullman. 1972. *The theory of parsing, translation, and compiling*. Prentice-Hall.

Bharat Ram Ambati, Tejaswini Deoskar, Mark Johnson, and Mark Steedman. 2015. An incremental algorithm for transition-based CCG parsing. In *Proc. of NAACL*.

Bharat Ram Ambati, Tejaswini Deoskar, and Mark Steedman. 2016. Shift-reduce CCG parsing using neural network models. In *Proc. of NAACL (Vol. 2)*.

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

Jacob Andreas and Dan Klein. 2014. How much do word embeddings encode about syntax? In *Proc. of ACL (Vol. 2)*.

Michael Auli and Jianfeng Gao. 2014. Decoder integration and expected BLEU training for recurrent neural network language models. In *Proc. of ACL (Vol. 2)*.

Michael Auli and Adam Lopez. 2011a. A comparison of loopy belief propagation and dual decomposition for integrated CCG supertagging and parsing. In *Proc. of ACL*.

Michael Auli and Adam Lopez. 2011b. Training a log-linear parser with loss functions via softmax-margin. In *Proc. of EMNLP*.

Michael Auli, Michel Galley, and Jianfeng Gao. 2014. Large-scale expected BLEU training of phrase-based reordering models. In *Proc. of EMNLP*.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *Proc. of ICLR*.

Miguel Ballesteros, Chris Dyer, and Noah A. Smith. 2015. Improved transition-based parsing by modeling characters instead of words with LSTMs. In *Proc. of EMNLP*.

Miguel Ballesteros, Yoav Goldberg, Chris Dyer, and Noah A. Smith. 2016. Training with exploration improves a greedy stack-LSTM parser. In *Proc. of EMNLP (Vol. 2)*.

Srinivas Bangalore and Aravind Joshi. 1999. Supertagging: An approach to almost parsing. In *Computational linguistics*. MIT Press.

Yehoshua Bar-Hillel. 1953. A quasi-arithmetical notation for syntactic description. In *Language*. JSTOR.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. 1994. Learning long-term dependencies with gradient descent is difficult. In *IEEE Transactions on Neural Networks*. IEEE.

Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam M. Shazeer. 2015. Scheduled sampling for sequence prediction with recurrent neural networks. In *Proc. of NIPS*.

Yoshua Bengio. 2012. Practical recommendations for gradient-based training of deep architectures. In *Neural Networks: Tricks of the Trade*. Springer.

Bernd Bohnet. 2010. Very high accuracy and fast dependency parsing is not a contradiction. In *Proc. of COLING*.

Leon Bottou. 1991. *Une approche theorique de l'apprentissage connexionniste et applications a la reconnaissance de la parole*. Ph.D. thesis, Universite de Paris XI.

Ted Briscoe and John Carroll. 1993. Generalized probabilistic LR parsing of natural language (corpora) with unification-based grammars. In *Computational linguistics*. MIT Press.

Ted Briscoe and John Carroll. 2006. Evaluating the accuracy of an unlexicalized statistical parser on the PARC DepBank. In *Proc. of COLING/ACL*.

Ted Briscoe, John Carroll, and Rebecca Watson. 2006. The second release of the RASP system. In *Proc. of COLING/ACL Interactive presentation sessions*.

Xavier Carreras, Michael Collins, and Terry Koo. 2008. TAG, dynamic programming, and the perceptron for efficient, feature-rich parsing. In *Proc. of CoNLL*.

Xavier Carreras. 2007. Experiments with a higher-order projective dependency parser. In *Proc. of the CoNLL Shared Task Session of EMNLP/CoNLL*.

Daniel Cer, Marie-Catherine De Marneffe, Daniel Jurafsky, and Christopher D. Manning. 2010. Parsing to stanford dependencies: Trade-offs between speed and accuracy. In *Proc. of LREC*.

Yin-Wen Chang and Michael Collins. 2011. Exact decoding of phrase-based translation models through lagrangian relaxation. In *Proc. of EMNLP*.

Eugene Charniak and Mark Johnson. 2005. Coarse-to-fine n-best parsing and MaxEnt discriminative reranking. In *Proc. of ACL*.

Ciprian Chelba and Frederick Jelinek. 1998. Exploiting syntactic structure for language modeling. In *Proc. of ACL*.

Danqi Chen and Christopher D. Manning. 2014. A fast and accurate dependency parser using neural networks. In *Proc. of EMNLP*.

David Chiang, Yuval Marton, and Philip Resnik. 2008. Online large-margin training of syntactic and structural translation features. In *Proc. of EMNLP*.

David Chiang, Kevin Knight, and Wei Wang. 2009. 11,001 new features for statistical machine translation. In *Proc. of NAACL*.

David Chiang. 2007. Hierarchical phrase-based translation. In *Computational Linguistics*. MIT Press.

Jason Chiu and Eric Nichols. 2015. Named entity recognition with bidirectional LSTM-CNNs. In *Transactions of the Association for Computational Linguistics*. ACL.

Noam Chomsky. 1957. *Syntactic structures*. The Hague: Mouton.

Stephen Clark and James Curran. 2004a. The importance of supertagging for wide-coverage CCG parsing. In *Proc. of COLING*.

Stephen Clark and James Curran. 2004b. Parsing the WSJ using CCG and log-linear models. In *Proc. of ACL*.

Stephen Clark and James Curran. 2006. Partial training for a lexicalized-grammar parser. In *Proc. of NAACL*.

Stephen Clark and James Curran. 2007. Wide-coverage efficient statistical parsing with CCG and log-linear models. In *Computational Linguistics*. MIT Press.

Stephen Clark and Julia Hockenmaier. 2002. Evaluating a wide-coverage CCG parser. In *Proc. of the LREC 2002 Beyond Parseval Workshop*.

Stephen Clark, Julia Hockenmaier, and Mark Steedman. 2002. Building deep dependency structures with a wide-coverage CCG parser. In *Proc. of ACL*.

Stephen Clark, Darren Foong, Luana Bulat, and Wenduan Xu. 2015. The Java version of the C&C parser. Technical report, University of Cambridge Computer Laboratory.

Stephen Clark. 2002. Supertagging for Combinatory Categorial Grammar. In *Proceedings of the 6th International Workshop on Tree Adjoining Grammars and Related Frameworks (TAG+6)*.

Michael Collins and Brian Roark. 2004. Incremental parsing with the perceptron algorithm. In *Proc. of ACL*.

Michael Collins. 1997. Three generative, lexicalised models for statistical parsing. In *Proc. of ACL*.

Michael Collins. 1999. *Head-driven statistical models for natural language parsing*. Ph.D. thesis, University of Pennsylvania.

Michael Collins. 2000. Discriminative reranking for natural language parsing. In *Proc. of ICML*.

Michael Collins. 2002. Discriminative training methods for Hidden Markov Models: Theory and experiments with perceptron algorithms. In *Proc. of EMNLP*.

Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural language processing (almost) from scratch. In *The Journal of Machine Learning Research*. JMLR.

James Curran, Stephen Clark, and David Vadas. 2006. Multi-tagging for lexicalized-grammar parsing. In *Proc. of COLING/ACL*.

James Curran, Stephen Clark, and Johan Bos. 2007. Linguistically motivated large-scale NLP with C&C and Boxer. In *Proc. of the ACL Interactive Poster and Demonstration Sessions*.

Jacob Devlin, Rabih Zbib, Zhongqiang Huang, Thomas Lamar, Richard Schwartz, and John Makhoul. 2014. Fast and robust neural network joint models for statistical machine translation. In *Proc. of ACL*.

Greg Durrett and Dan Klein. 2015. Neural CRF parsing. In *Proc. of ACL/IJCNLP*.

Chris Dyer and Philip Resnik. 2010. Context-free reordering, finite-state translation. In *Proc. of NAACL*.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. 2015. Transition-based dependency parsing with stack long short-term memory. In *Proc. of ACL*.

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

Chris Dyer. 2010. Two monolingual parses are better than one (synchronous parse). In *Proc. of NAACL*.

Jason Eisner. 1996a. Efficient normal-form parsing for Combinatory Categorial Grammar. In *Proc. of ACL*.

Jason Eisner. 1996b. Three new probabilistic models for dependency parsing: An exploration. In *Proc. of ACL*.

Jeffrey Elman. 1990. Finding structure in time. In *Cognitive Science*. Elsevier.

Timothy Fowler and Gerald Penn. 2010. Accurate context-free parsing with Combinatory Categorial Grammar. In *Proc. of ACL*.

Gottlob Frege. 1892. Über sinn und bedeutung. In *Zeitschrift fr Philosophie und philosophische Kritik, N.F.* C.E.M. Pfeffer.

Yoav Freund and Robert E. Schapire. 1999. Large margin classification using the perceptron algorithm. In *Machine Learning*. Springer.

Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Nguyen. 1993. Directed hypergraphs and applications. In *Discrete Applied Mathematics*. Elsevier.

Jianfeng Gao and Xiaodong He. 2013. Training MRF-based phrase translation models using gradient ascent. In *Proc. of NAACL*.

Jianfeng Gao, Xiaodong He, Wen-tau Yih, and Li Deng. 2014. Learning continuous phrase representations for translation modeling. In *Proc. of ACL*.

Stuart Geman and Mark Johnson. 2002. Dynamic programming for parsing and estimation of stochastic unification-based grammars. In *Proc. of ACL*.

Felix Gers and Jürgen Schmidhuber. 2000. Recurrent nets that time and count. In *Neural Networks*. IEEE.

Felix Gers, Jürgen Schmidhuber, and Fred Cummins. 2000. Learning to forget: Continual prediction with LSTM. In *Neural Computation*. MIT Press.

Kevin Gimpel and Noah A. Smith. 2010. Softmax-margin CRFs: training log-linear models with cost functions. In *Proc. of NAACL*.

Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proc. of AISTATS*.

Yoav Goldberg and Joakim Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. In *Proc. of COLING*.

Yoav Goldberg, Kai Zhao, and Liang Huang. 2013. Efficient implementation for beam search incremental parsers. In *Proc. of ACL (Vol. 2)*.

Christoph Goller and Andreas Kuchler. 1996. Learning task-dependent distributed representations by backpropagation through structure. In *Proc. of ICNN*.

Joshua Goodman. 1996. Parsing algorithms and metrics. In *Proc. of ACL*.

Joshua Goodman. 1999. Semiring parsing. In *Computational Linguistics*. MIT Press.

Alex Graves and Jürgen Schmidhuber. 2005. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. In *Neural Networks*. Elsevier.

Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. 2016. LSTM: A search space odyssey. In *IEEE Transactions on Neural Networks and Learning Systems*. IEEE.

Xiaodong He and Li Deng. 2012. Maximum expected BLEU training of phrase and lexicon translation models. In *Proc. of ACL*.

James Henderson. 2003. Inducing history representations for broad coverage statistical parsing. In *Proc. of NAACL.*

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. In *Neural Computation*. MIT Press.

Sepp Hochreiter. 1998. The vanishing gradient problem during learning recurrent neural nets and problem solutions. In *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*. World Scientific.

Julia Hockenmaier and Yonatan Bisk. 2010. Normal-form parsing for Combinatory Categorial Grammars with generalized composition and type-raising. In *Proc. of COLING.*

Julia Hockenmaier and Mark Steedman. 2007. CCGBank: A corpus of CCG derivations and dependency structures extracted from the Penn Treebank. In *Computational Linguistics*. MIT Press.

Julia Hockenmaier. 2003. *Data and Models for Statistical Parsing with Combinatory Categorial Grammar*. Ph.D. thesis, University of Edinburgh.

Matthew Honnibal, Joel Nothman, and James Curran. 2009. Evaluating a statistical CCG parser on Wikipedia. In *Proc. of the Workshop on The People's Web Meets NLP: Collaboratively Constructed Semantic Resources.*

Liang Huang and David Chiang. 2005. Better k-best parsing. In *Proc. of IWPT.*

Liang Huang and David Chiang. 2007. Forest rescoring: Faster decoding with integrated language models. In *Proc. of ACL.*

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

Liang Huang, Suphan Fayong, and Yang Guo. 2012. Structured perceptron with inexact search. In *Proc. of NAACL.*

Zhiheng Huang, Wei Xu, and Kai Yu. 2015. Bidirectional LSTM-CRF models for sequence tagging. *arXiv:1508.01991.*

Liang Huang. 2008. Forest reranking: Discriminative parsing with non-local features. *Proc. of ACL.*

Mark Johnson. 2001. Joint and conditional estimation of tagging and parsing models. In *Proc. of ACL.*

Eliyahu Kiperwasser and Yoav Goldberg. 2016. Easy-first dependency parsing with hierarchical tree LSTMs. In *Transactions of the Association for Computational Linguistics*. ACL.

Dan Klein and Christopher D. Manning. 2003. Fast exact inference with a factored model for natural language parsing. In *Proc. of NIPS*.

Terry Koo and Michael Collins. 2010. Efficient third-order dependency parsers. In *Proc. of ACL*.

Terry Koo, Xavier Carreras, and Michael Collins. 2008. Simple semi-supervised dependency parsing. In *Proc. of ACL/HLT*.

Terry Koo, Alexander M. Rush, Michael Collins, Tommi Jaakkola, and David Sontag. 2010. Dual decomposition for parsing with non-projective head automata. In *Proc. of EMNLP*.

Marco Kuhlmann and Giorgio Satta. 2014. A new parsing algorithm for Combinatory Categorial Grammar. In *Transactions of the Association for Computational Linguistics*. ACL.

Marco Kuhlmann, Alexander Koller, and Giorgio Satta. 2010. The importance of rule restrictions in CCG. In *Proc. of ACL*.

Marco Kuhlmann, Carlos Gómez-Rodríguez, and Giorgio Satta. 2011. Dynamic programming algorithms for transition-based dependency parsers. In *Proc. of ACL*.

Marco Kuhlmann, Giorgio Satta, and Peter Jonsson. 2017. On the complexity of CCG parsing. *arXiv:1702.06594*.

Jonathan Kummerfeld, Jessika Roesner, Tim Dawborn, James Haggerty, James Curran, and Stephen Clark. 2010. Faster parsing by supertagger adaptation. In *Proc. of ACL*.

Matthieu Labeau, Kevin Löser, Alexandre Allauzen, and Rue John von Neumann. 2015. Non-lexical neural architecture for fine-grained POS tagging. In *Proc. of EMNLP*.

John Lafferty, Andrew McCallum, and Fernando Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proc. of ICML*.

Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. 2016. Neural architectures for named entity recognition. In *Proc. of NAACL*.

Kenton Lee, Mike Lewis, and Luke Zettlemoyer. 2016. Global neural CCG parsing with optimality guarantees. In *Proc. of EMNLP*.

Joël Legrand and Ronan Collobert. 2015. Joint RNN-based greedy parsing and word composition. In *Proc. of ICLR*.

Mike Lewis and Mark Steedman. 2014a. A* CCG parsing with a supertag-factored model. In *Proc. of EMNLP.*

Mike Lewis and Mark Steedman. 2014b. Improved CCG parsing with semi-supervised supertagging. In *Transactions of the Association for Computational Linguistics.* ACL.

Mike Lewis, Kenton Lee, and Luke Zettlemoyer. 2016. LSTM CCG parsing. In *Proc. of NAACL.*

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proc. of EMNLP.*

Xuezhe Ma and Eduard Hovy. 2016. End-to-end sequence labeling via bi-directional LSTM-CNNs-CRF. In *Proc. of ACL.*

Mitchell Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. In *Computational Linguistics.* MIT.

André F.T. Martins, Noah A. Smith, and Eric Xing. 2009. Concise integer linear programming formulations for dependency parsing. In *Proc. of ACL/AFNLP.*

André F.T. Martins, Noah A. Smith, Eric Xing, Pedro M.Q. Aguiar, and Mário A.T. Figueiredo. 2010. Turbo parsers: Dependency parsing by approximate variational inference. In *Proc. of EMNLP.*

André F.T. Martins, Noah A. Smith, Pedro M.Q. Aguiar, and Mário A.T. Figueiredo. 2011. Dual decomposition with many overlapping components. In *Proc. of EMNLP.*

Yuval Marton and Philip Resnik. 2008. Soft syntactic constraints for hierarchical phrased-based translation. In *Proc. of ACL.*

Marshall R. Mayberry and Risto Miikkulainen. 1999. SARDSRN: A neural network shift-reduce parser. In *Proc. of IJCAI.*

Andrew McCallum, Dayne Freitag, and Fernando Pereira. 2000. Maximum entropy Markov models for information extraction and segmentation. In *Proc. of ICML.*

Ryan McDonald and Joakim Nivre. 2007. Characterizing the errors of data-driven dependency parsing models. In *Proc. of EMNLP/CoNLL.*

Ryan McDonald and Fernando Pereira. 2006. Online learning of approximate dependency parsing algorithms. In *Proc. of EACL.*

Ryan McDonald, Joakim Nivre, Yvonne Quirmbach-Brundage, Yoav Goldberg, Dipanjan Das, Kuzman Ganchev, Keith Hall, Slav Petrov, Hao Zhang, Oscar Täckström, Claudia Bedini, Núria Bertomeu Castelló, and Jungmee Lee. 2013. Universal dependency annotation for multilingual parsing. In *Proc. of ACL.*

Ryan McDonald. 2006. *Discriminative learning and spanning tree algorithms for dependency parsing.* Ph.D. thesis, University of Pennsylvania.

Haitao Mi, Liang Huang, and Qun Liu. 2008. Forest-based translation. In *Proc. of ACL.*

Risto Miikkulainen. 1996. Subsymbolic case-role analysis of sentences with embedded clauses. In *Cognitive Science.* Elsevier.

Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Proc. of INTERSPEECH.*

Yusuke Miyao and Jun'ichi Tsujii. 2002. Maximum entropy estimation for feature forests. In *Proc. of HLT.*

Vinod Nair and Geoffrey E. Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proc. of ICML.*

Joakim Nivre and Ryan McDonald. 2008. Integrating graph-based and transition-based dependency parsers. In *Proc. of ACL/HLT.*

Joakim Nivre and Mario Scholz. 2004. Deterministic dependency parsing of English text. In *Proc. of COLING.*

Joakim Nivre, Johan Hall, and Jens Nilsson. 2006. Maltparser: A data-driven parser-generator for dependency parsing. In *Proc. of LREC.*

Joakim Nivre, Laura Rimell, Ryan McDonald, and Carlos Gomez-Rodriguez. 2010. Evaluation of dependency parsers on unbounded dependencies. In *Proc. of COLING.*

Joakim Nivre. 2004. Incrementality in deterministic dependency parsing. In *Proc. of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together.*

Joakim Nivre. 2008. Algorithms for deterministic incremental dependency parsing. In *Computational Linguistics.* MIT Press.

Franz Och. 2003. Minimum error rate training in statistical machine translation. In *Proc. of ACL.*

Sampo Pyysalo, Filip Ginter, Juho Heimonen, Jari Björne, Jorma Boberg, Jouni Järvinen, and Tapio Salakoski. 2007. Bioinfer: a corpus for information extraction in the biomedical domain. In *BMC bioinformatics.* BioMed Central.

Lance Ramshaw, Elizabeth Boschee, Sergey Bratus, Scott Miller, Rebecca Stone, Ralph Weischedel, and Alex Zamanian. 2001. Experiments in multi-modal automatic content extraction. In *Proc. of HLT.*

Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. 2016. Sequence level training with recurrent neural networks. In *Proc. of ICLR*.

Sebastian Riedel and James Clarke. 2006. Incremental integer linear programming for non-projective dependency parsing. In *Proc. of EMNLP*.

Laura Rimell and Stephen Clark. 2008. Adapting a lexicalized-grammar parser to contrasting domains. In *Proc. of EMNLP*.

Laura Rimell, Stephen Clark, and Mark Steedman. 2009. Unbounded dependency recovery for parser evaluation. In *Proc. of EMNLP*.

Frank Rosenblatt. 1958. The perceptron: A probabilistic model for information storage and organization in the brain. In *Psychological Review*. American Psychological Association.

Antti-Veikko Rosti, Bing Zhang, Spyros Matsoukas, and Richard Schwartz. 2010. BBN system description for WMT10 system combination task. In *Proc. of the Joint Fifth Workshop on Statistical Machine Translation and MetricsMATR*.

David Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1988. Learning representations by back-propagating errors. In *Nature*.

Conrad Sanderson. 2010. Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments. Technical report, NICTA.

Nathan Schneider, Brendan O'Connor, Naomi Saphra, David Bamman, Manaal Faruqui, Noah A. Smith, Chris Dyer, and Jason Baldridge. 2013. A framework for (under)specifying dependency syntax without overloading annotators. In *Proc. of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*.

David Smith and Jason Eisner. 2006. Minimum-risk annealing for training log-linear models. In *Proc. of COLING/ACL*.

Noah A. Smith and Mark Johnson. 2007. Weighted and probabilistic context-free grammars are equally expressive. In *Computational Linguistics*. MIT Press.

Richard Socher, Christopher D. Manning, and Andrew Ng. 2010. Learning continuous phrase representations and syntactic parsing with recursive neural networks. In *Proc. of the NIPS Deep Learning and Unsupervised Feature Learning Workshop*.

Richard Socher, Cliff Lin, Christopher D. Manning, and Andrew Ng. 2011. Parsing natural scenes and natural language with recursive neural networks. In *Proc. of ICML*.

Richard Socher, John Bauer, Christopher D. Manning, and Andrew Ng. 2013. Parsing with compositional vector grammars. In *Proc. of ACL*.

Anders Søgaard and Yoav Goldberg. 2016. Deep multi-task learning with low level tasks supervised at lower layers. In *Proc. of ACL (Vol. 2)*.

Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. In *The Journal of Machine Learning Research*. JMLR.

Mark Steedman. 1996. A very short introduction to CCG. *Unpublished paper: http://www.inf.ed.ac.uk/teaching/courses/ics/papers/ccgintro.pdf*.

Mark Steedman. 2000. *The Syntactic Process*. MIT Press.

Ilya Sutskever, Oriol Vinyals, and Quoc Le. 2014. Sequence to sequence learning with neural networks. In *Proc. of NIPS*.

Ben Taskar, Carlos Guestrin, and Daphne Koller. 2003. Max margin Markov networks. In *Proc. of NIPS*.

Ben Taskar, Dan Klein, Michael Collins, Daphne Koller, and Christopher D. Manning. 2004. Max-margin parsing. In *Proc. of EMNLP*.

Masaru Tomita. 1985. An efficient context-free parsing algorithm for natural languages. In *Proc. of IJCAI*.

Roy Tromble, Shankar Kumar, Franz Och, and Wolfgang Macherey. 2008. Lattice minimum bayes-risk decoding for statistical machine translation. In *Proc. of EMNLP*.

Ioannis Tsochantaridis, Thorsten Joachims, Thomas Hofmann, and Yasemin Altun. 2005. Large margin methods for structured and interdependent output variables. In *The Journal of Machine Learning Research*. JMLR.

Joseph Turian, Lev Ratinov, and Yoshua Bengio. 2010. Word representations: a simple and general method for semi-supervised learning. In *Proc. of ACL*.

Ashish Vaswani and Kenji Sagae. 2016. Efficient structured inference for transition-based parsing with neural networks and error states. In *Transactions of the Association for Computational Linguistics*. ACL.

Ashish Vaswani, Yonatan Bisk, Kenji Sagae, and Ryan Musa. 2016. Supertagging with LSTMs. In *Proc. of NAACL (Vol. 2)*.

Krishnamurti Vijay-Shanker and David Weir. 1993. Parsing some constrained grammar formalisms. In *Computational Linguistics*. MIT Press.

Krishnamurti Vijay-Shanker and David J Weir. 1994. The equivalence of four extensions of context-free grammars. In *Theory of Computing Systems*. Springer.

Oriol Vinyals, Lukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey E. Hinton. 2015. Grammar as a foreign language. In *Proc. of NIPS*.

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

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

Paul Werbos. 1974. *Beyond regression: New tools for prediction and analysis in the behavioral sciences.* Ph.D. thesis, Harvard University.

Paul Werbos. 1990. Backpropagation through time: what it does and how to do it. In *Proc. of the IEEE.* IEEE.

Sam Wiseman and Alexander M. Rush. 2016. Sequence-to-sequence learning as beam-search optimization. In *Proc. of EMNLP.*

Dekai Wu. 1997. Stochastic inversion transduction grammars and bilingual parsing of parallel corpora. In *Computational linguistics.* MIT Press.

Wenduan Xu, Stephen Clark, and Yue Zhang. 2014. Shift-reduce CCG parsing with a dependency model. In *Proc. of ACL.*

Wenduan Xu, Michael Auli, and Stephen Clark. 2015. CCG supertagging with a recurrent neural network. In *Proc. of ACL (Vol. 2).*

Wenduan Xu, Michael Auli, and Stephen Clark. 2016. Expected F-measure training for shift-reduce parsing with recurrent neural networks. In *Proc. of NAACL.*

Wenduan Xu. 2016. LSTM shift-reduce CCG parsing. In *Proc. of EMNLP.*

Kenji Yamada and Kevin Knight. 2001. A syntax-based statistical translation model. In *Proc. of ACL.*

Hiroyasu Yamada and Yuji Matsumoto. 2003. Statistical dependency analysis using support vector machines. In *Proc. of IWPT.*

Zhilin Yang, Ruslan Salakhutdinov, and William Cohen. 2016. Multi-task cross-lingual sequence tagging from scratch. *arXiv:1603.06270.*

Heng Yu, Liang Huang, Haitao Mi, and Kai Zhao. 2013. Max-violation perceptron and forced decoding for scalable MT training. In *Proc. of EMNLP.*

Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2014. Recurrent neural network regularization. In *Proc. of ICLR.*

Luke Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Proc. of UAI.*

Yue Zhang and Stephen Clark. 2008. A tale of two parsers: investigating and combining graph-based and transition-based dependency parsing using beam-search. In *Proc. of EMNLP.*

Yue Zhang and Stephen Clark. 2011a. Shift-reduce CCG parsing. In *Proc. of ACL*.

Yue Zhang and Stephen Clark. 2011b. Syntactic processing using the generalized perceptron and beam search. In *Computational Linguistics*. MIT Press.

Hao Zhang and Ryan McDonald. 2012. Generalized higher-order dependency parsing with cube pruning. In *Proc. of EMNLP/CoNLL*.

Hao Zhang and Ryan McDonald. 2014. Enforcing structural diversity in cube-pruned dependency parsing. In *Proc. of ACL (Vol. 2)*.

Yue Zhang and Joakim Nivre. 2011. Transition-based dependency parsing with rich non-local features. In *Proc. of ACL (Vol. 2)*.

Yuan Zhang and David Weiss. 2016. Stack-propagation: Improved representation learning for syntax. In *Proc. of ACL*.

Hao Zhang, Liang Huang, Kai Zhao, and Ryan McDonald. 2013. Online learning for inexact hypergraph search. In *Proc. of EMNLP*.

Hao Zhou, Yue Zhang, Shujian Huang, and Jiajun Chen. 2015. A neural probabilistic structured-prediction model for transition-based dependency parsing. In *Proc. of ACL*.

Muhua Zhu, Yue Zhang, Wenliang Chen, Min Zhang, and Jingbo Zhu. 2013. Fast and accurate shift-reduce constituent parsing. In *Proc. of ACL*.