Neural Representation Learning in Linguistic Structured Prediction

Lingpeng Kong

October 2016

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Noah A. Smith (co-Chair), Carnegie Mellon University / University of Washington
Chris Dyer (co-Chair), Carnegie Mellon University
Alan W. Black, Carnegie Mellon University
Michael Collins, Columbia University

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Abstract

Advances in neural network architectures and training algorithms have demonstrated the effectiveness of representation learning in natural language processing. This thesis stresses the importance of computationally modeling the structure in language, even when learning representations.

We propose that explicit structure representations and learned distributed representations can be efficiently combined for improved performance over (i) traditional approaches to structure and (ii) uninformed neural networks that ignore all but surface sequential structure. We demonstrate on three distinct problems how assumptions about structure can be integrated naturally into neural representation learners for NLP problems, without sacrificing computational efficiency.

First, we introduce an efficient model for inferring the phrase-structure trees using given dependency syntax trees as constraints and propose to extend the model, making it more expressive through non-linear (neural) representation learning.

Second, we propose segmental recurrent neural networks (SRNNs) which define, given an input sequence, a joint probability distribution over segmentations of the input and labelings of the segments and show that comparing to models that do not explicitly represent segments such as BIO tagging schemes and connectionist temporal classification (CTC), SRNNs obtain substantially higher accuracies.

Third, we consider the problem of Combinatory Categorial Grammar (CCG) supertagging. We propose to model the compositionality both inside these tags and between these tags. This enables the model to handle an unbounded number of supertags where structurally naive models simply fail.

The techniques proposed in this thesis automatically learn structurally informed representations of the inputs. These representations and components in the models can be better integrated with other end-to-end deep learning systems within and beyond NLP.

Contents

Chapter 1

Introduction

Computationally modeling the structure in language is crucial for two reasons. First, for the larger goal of automated language understanding, linguists have found that language meaning is derived through composition. Structure in language tells what are the atoms (e.g., words in a sentence) and how they fit together (e.g., syntactic or semantic parses) in composition. Second, from the perspective of machine learning, linguistic structures can be understood as a form of inductive bias, which helps learning succeed with less or less ideal data (Mitchell, 1980). For many tasks in NLP, only limited amounts of supervision is available. Therefore, getting the inductive bias right is particular important.

In recent years, neural models are state-of-the-art for many NLP tasks, including speech recognition (Graves et al., 2013), dependency parsing (Andor et al., 2016) and machine translation (Bahdanau et al., 2015). For many applications, the model architecture combines dense representations of words (Manning, 2016) and with sequential recurrent neural networks (Dyer et al., 2016). Thus, while they generally make use of information about what the words are (rather than operating on sequences of characters), they ignore syntactic and semantic structure and thus can be criticized as being structurally naive.

This thesis argues that explicit structure representations and learned distributed representations can be efficiently combined for improved performance over (i) traditional approaches to structure and (ii) uninformed neural networks that ignore all but surface sequential structure. As an example, Dyer et al. (2015) yields better accuracy than Chen and Manning (2014) by replacing the word representation on the stack with composed representations derived from linguistic structure.

In the first part of the thesis, first published as Kong et al. (2015b), we introduce an efficient model for inferring the phrase-structure trees using given dependency syntax trees as constraints. Dependency parsers are generally much faster, but less informative, since they do not produce constituents, which are often required by downstream applications. The approach we propose can be understood as a specially-trained coarse-to-fine decoding algorithm where the dependency parser provides "coarse" structure and the second stage refines it. Our algorithm achieved similar performance as state-of-the-art lexicalized phrase-structure parsers while doing it much more efficiently. This is an example of a structure-to-structure mapping problem that does not quite fit into the currently available neural frameworks that focus on sequences. We will extend the

previous work, making such model more expressive through non-linear (neural) representation learning, while still making use of the structural information available as input and producing well-formed structural output. This will serve as an example of integrating structural and neural models.

In the second part of this thesis, first published as (Kong et al., 2015a; Lu et al., 2016), we propose segmental recurrent neural networks (SRNNs) which define, given an input sequence, a joint probability distribution over segmentations of the input and labelings of the segments. Traditional neural solutions to this problem, e.g., connectionist temporal classification (CTC) (Graves et al., 2006a), reduce the segmental sequence labeling problem to a sequence labeling problem in the same spirit as *BIO* tagging. In our model, representations of the input segments (i.e., contiguous subsequences of the input) are computed by composing their constituent tokens using bidirectional recurrent neural nets, and these *segment embeddings* are used to define compatibility scores with output labels. Our model achieves competitive results in phone recognition, handwriting recognition, and joint word segmentation & POS tagging.

In the third part of the thesis, we consider the problem of Combinatory Categorial Grammar (CCG) supertagging (Steedman, 2000). CCG is widely used in semantic parsing, and parsing with it is very fast, especially if there's a good (probabilistic) tagger (Lewis and Steedman, 2014a; Lee et al., 2016). While the Penn Treebank (Marcus et al., 1993) has just 50 fixed POS tags, tagging the same corpus with CCG supertags uses over 400 lexical categories (i.e., supertags) and, in general, this number is unbounded since tags may be created productively. Current taggers (Lewis and Steedman, 2014a; Xu et al., 2015b; Lewis and Steedman, 2014b, *inter alia*) simply treat these supertags as discrete categories. We propose to model the compositionality both inside these tags and between these tags (even when not doing parsing, the compositionality between the tags can be modeled as a sequence level loss inside the model). Previous works suggests that explicitly modeling the compositional aspect of supertags can lead to better accuracy (Srikumar and Manning, 2014) and less supervision (Garrette et al., 2015).

In this thesis, we stress the importance of explicitly representing structure in neural models. We demonstrate on three distinct problems how assumptions about structure can be integrated naturally into neural representation learners for NLP problems, without sacrificing computational efficiency. We argue that, when comparing with structurally naive models, models that reason about the internal linguistic structure of the data demonstrate better generalization performance.

Chapter 2

Transforming Dependencies into Phrase Structures

Our prior work (Kong et al., 2015b) introduces an efficient model for inferring the phrase-structure trees using given dependency syntax trees as constraints. Because dependency parsers are generally much faster than phrase-structure parsers, we consider an alternate pipeline (Section 2.1): dependency parse first, then transform the dependency representation into a phrase-structure tree constrained to be consistent with the dependency parse. This idea was explored by Xia and Palmer (2001) and Xia et al. (2009) using hand-written rules. Instead, we present a data-driven algorithm using the structured prediction framework (Section 2.2). The approach can be understood as a specially-trained coarse-to-fine decoding algorithm where a dependency parser provides "coarse" structure and the second stage refines it (Charniak and Johnson, 2005; Petrov and Klein, 2007).

Our lexicalized phrase-structure parser, PAD, is asymptotically faster than parsing with a lexicalized context-free grammar: $O(n^2)$ plus dependency parsing, vs. $O(n^5)$ worst case runtime in sentence length n, with the same grammar constant. Experiments show that our approach achieves linear observable runtime, and accuracy similar to state-of-the-art phrase-structure parsers without reranking or semi-supervised training (Section 2.4).

We will extend the previous work, making the model more expressive through non-linear (neural) representation learning, while still making use of the structural information available as input and producing well-formed structural output. This will serve as an example of integrating structural and neural models.

2.1 Parsing Dependencies

2.1.1 Parsing Algorithm

Consider the classical problem of predicting the best phrase-structure parse under a CFG with head rules, known as lexicalized context-free parsing. Assume that we are given a binary CFG defining a set of valid phrase-structure parses $\mathcal{Y}(x)$. The parsing problem is to find the highest-scoring parse in this set, i.e. $\arg\max_{y \in \mathcal{Y}(x)} s(y; x)$ where s is a scoring function that factors over

Premise:

$$(\langle i, i\rangle, i, A) \quad \forall i \in \{1 \dots n\}, A \in \mathcal{N}$$

Rules:

For $i \leq h \leq k < m \leq j$, and rule $A \rightarrow \beta_1^* \ \beta_2$,

$$\frac{(\langle i, k\rangle, h, \beta_1) \quad (\langle k+1, j\rangle, m, \beta_2)}{(\langle i, j\rangle, h, A)}$$

For $i \leq m \leq k < h \leq j$, rule $A \rightarrow \beta_1 \ \beta_2^*$,

$$\frac{(\langle i, k\rangle, m, \beta_1) \quad (\langle k+1, j\rangle, h, \beta_2)}{(\langle i, j\rangle, h, A)}$$

Goal:

$$(\langle 1, n\rangle, m, r) \text{ for any } m$$

Premise:

$$(\langle i, i\rangle, i, A) \quad \forall i \in \{1 \dots n\}, A \in \mathcal{N}$$

Rules:

For all h, $m \in \mathcal{R}(h)$, rule $A \rightarrow \beta_1^* \ \beta_2$,
and $i \in \{m'_{\Leftarrow} : m' \in \mathcal{L}(h)\} \cup \{h\}$,

$$\frac{(\langle i, m_{\Leftarrow} - 1\rangle, h, \beta_1) \quad (\langle m_{\Leftarrow}, m_{\Rightarrow}\rangle, m, \beta_2)}{(\langle i, m_{\Rightarrow}\rangle, h, A)}$$

For all h, $m \in \mathcal{L}(h)$, rule $A \rightarrow \beta_1 \ \beta_2^*$,
and $j \in \{m'_{\Rightarrow} : m' \in \mathcal{R}(h)\} \cup \{h\}$,

$$\frac{(\langle m_{\Leftarrow}, m_{\Rightarrow}\rangle, m, \beta_1) \quad (\langle m_{\Rightarrow} + 1, j\rangle, h, \beta_2)}{(\langle m_{\Leftarrow}, j\rangle, h, A)}$$

Goal:

$$(\langle 1, n\rangle, m, r) \text{ for any } m \in \mathcal{R}(0)$$

Figure 2.1: The two algorithms written as deductive parsers. Starting from the *premise*, any valid application of *rules* that leads to a *goal* is a valid parse. Left: lexicalized CKY algorithm for CFG parsing with head rules. For this algorithm there are $O(n^5|\mathcal{G}|)$ rules where n is the length of the sentence. Right: the constrained CKY parsing algorithm for $\mathcal{Y}(x, d)$. The algorithm is nearly identical except that many of the free indices are now fixed given the dependency parse. Finding the optimal phrase-structure parse with the new algorithm now requires $O\left(\left(\sum_h |\mathcal{L}(h)||\mathcal{R}(h)|\right)|\mathcal{G}|\right)$ time where $\mathcal{L}(h)$ and $\mathcal{R}(h)$ are the left and right dependents of word h.

lexicalized tree productions.

This problem can be solved by extending the CKY algorithm to propagate head information. The algorithm can be compactly defined by the productions in Figure 2.1 (left). For example, one type of production is of the form

$$\frac{(\langle i, k\rangle, m, \beta_1) \quad (\langle k+1, j\rangle, h, \beta_2)}{(\langle i, j\rangle, h, A)}$$

for all rules $A \rightarrow \beta_1 \ \beta_2^* \in \mathcal{G}$ and spans $i \leq k < j$. This particular production indicates that rule $A \rightarrow \beta_1 \ \beta_2^*$ was applied at a vertex covering $\langle i, j\rangle$ to produce two vertices covering $\langle i, k\rangle$ and $\langle k+1, j\rangle$, and that the new head is index h has dependent index m. We say this production "completes" word m since it can no longer be the head of a larger span.

Running the algorithm consists of bottom-up dynamic programming over these productions. However, applying this version of the CKY algorithm requires $O(n^5|\mathcal{G}|)$ time (linear in the number of productions), which is not practical to run without heavy pruning. Most lexicalized parsers therefore make further assumptions on the scoring function which can lead to asymptotically faster algorithms (Eisner and Satta, 1999).

Instead, we consider the same objective, but constrain the phrase-structure parses to be consistent with a given dependency parse, d. By "consistent," we mean that the phrase-structure

parse will be converted by the head rules to this exact dependency parse.[1] Define the set of consistent phrase-structure parses as $\mathcal{Y}(x, d)$ and the constrained search problem as $\arg\max_{y \in \mathcal{Y}(x,d)} s(y; x, d)$.

Figure 2.1 (right) shows the algorithm for this new problem. The algorithm has several nice properties. All rules now must select words h and m that are consistent with the dependency parse (i.e., there is an arc (h, m)) so these variables are no longer free. Furthermore, since we have the full dependency parse, we can precompute the dependency span of each word $\langle m_{\Leftarrow}, m_{\Rightarrow} \rangle$. By our definition of consistency, this gives us the phrase-structure parse span of m before it is completed, and fixes two more free variables. Finally the head item must have its alternative side index match a valid dependency span. For example, if for a word h there are $|\mathcal{L}(h)| = 3$ left dependents, then when taking the next right-dependent there can only be 4 valid left boundary indices.

The runtime of the final algorithm reduces to $O(\sum_h |\mathcal{L}(h)||\mathcal{R}(h)||\mathcal{G}|)$. While the terms $|\mathcal{L}(h)|$ and $|\mathcal{R}(h)|$ could in theory make the runtime quadratic, in practice the number of dependents is almost always constant in the length of the sentence. This leads to linear observed runtime in practice as we will show in Section 2.4.

2.1.2 Pruning

In addition to constraining the number of phrase-structure parses, the dependency parse also provides valuable information about the labeling and structure of the phrase-structure parse. We can use this information to further prune the search space. We employ two pruning methods:

Method 1 uses the part-of-speech tag of x_h, $\mathrm{tag}(h)$, to limit the possible rule productions at a given span. We build tables $\mathcal{G}_{\mathrm{tag}(h)}$ and restrict the search to rules seen in training for a particular part-of-speech tag.

Method 2 prunes based on the order in which dependent words are added. By the constraints of the algorithm, a head word x_h must combine with each of its left and right dependents. However, the order of combination can lead to different tree structures (as illustrated in Figure 2.2).

In total there are $|\mathcal{L}(h)| \times |\mathcal{R}(h)|$ possible orderings of dependents.

In practice, though, it is often easy to predict which side, left or right, will come next. We do this by estimating the distribution,

$$p(\text{side} \mid \mathrm{tag}(h), \mathrm{tag}(m), \mathrm{tag}(m')),$$

where $m \in \mathcal{L}(h)$ is the next left dependent and $m' \in \mathcal{R}(h)$ is the next right dependent. If the conditional probability of left or right is greater than a threshold parameter γ, we make a hard decision to combine with that side next. This pruning further reduces the impact of outliers with multiple dependents on both sides.

We empirically measure how these pruning methods affect observed runtime and oracle parsing performance (i.e., how well a perfect scoring function could do with a pruned $\mathcal{Y}(x, d)$). Table 2.1 shows a comparison of these pruning methods on development data. The constrained parsing algorithm is much faster than standard lexicalized parsing, and pruning contributes even

[1] An alternative, soft version of consistency, might enforce that the phrase-structure parse is close to the dependency parse. While this allows the algorithm to potentially correct dependency parse mistakes, it is much more computationally expensive.

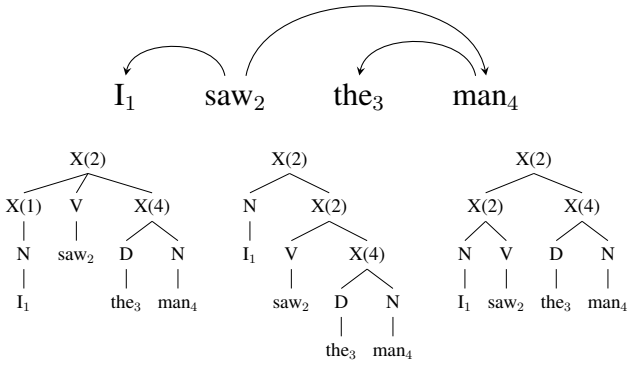


Figure 2.2: [Adapted from Collins et al. (1999).] A d-parse (left) and several c-parses consistent with it (right). Our goal is to select the best parse from this set.

Model	Complexity	Sent./s.	Oracle F_1
LEX CKY*	$n^5\|\mathcal{G}\|$	0.25	100.0
DEP CKY	$\sum_h \|\mathcal{L}(h)\|\|\mathcal{R}(h)\|\|\mathcal{G}\|$	71.2	92.6
PRUNE1	$\sum_h \|\mathcal{L}(h)\|\|\mathcal{R}(h)\|\|\mathcal{G}_{\mathrm{T}}\|$	336.0	92.5
PRUNE2	–	96.6	92.5
PRUNE1+2	–	425.1	92.5

Table 2.1: Comparison of three parsing setups: LEX CKY* is the complete lexicalized phrase-structure parser on $\mathcal{Y}(x)$, but limited to only sentences less than 20 words for tractability, DEP CKY is the constrained phrase-structure parser on $\mathcal{Y}(x, d)$, PRUNE1, PRUNE2, and PRUNE1+2 are combinations of the pruning methods described in Section 2.1.2. The oracle is the best labeled F_1 achievable on the development data (§22, see Section **??**).

greater speed-ups. The oracle experiments show that the dependency parse constraints do contribute a large drop in oracle accuracy, while pruning contributes a relatively small one. Still, this upper-bound on accuracy is high enough to make it possible to still recover phrase-structure parses at least as accurate as state-of-the-art phrase-structure parsers. We will return to this discussion in Section 2.4.

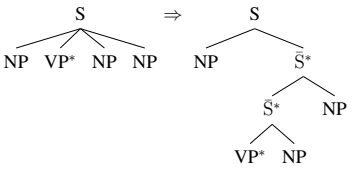
2.1.3 Binarization and Unary Rules

We have to this point developed the algorithm for a strictly binary-branching grammar; however, we need to produce trees have rules with varying size. In order to apply the algorithm, we binarize the grammar and add productions to handle unary rules.

Consider a non-binarized rule of the form $A \to \beta_1 \dots \beta_m$ with head child β_k^*. Relative to the head child β_k the rule has left-side $\beta_1 \dots \beta_{k-1}$ and right-side $\beta_{k+1} \dots \beta_m$. We replace this rule with new binary rules and non-terminal symbols to produce each side independently as a simple chain, left-side first. The transformation introduces the following new rules:[2] $A \to \beta_1\ \bar{A}^*$, $\bar{A} \to \beta_i\ \bar{A}^*$ for $i \in \{2, \dots, k\}$, and $\bar{A} \to \bar{A}^*\ \beta_i$ for $i \in \{k, \dots, m\}$.

As an example consider the transformation of a rule with four children:

[2]These rules are slightly modified when $k = 1$.



These rules can then be reversed deterministically to produce a non-binary tree.

We also explored binarization using horizontal and vertical markovization to include additional context of the tree, as found useful in unlexicalized approaches (Klein and Manning, 2003). Preliminary experiments showed that this increased the size of the grammar, and the runtime of the algorithm, without leading to improvements in accuracy.

Phrase-structure trees also include unary rules of the form $A \rightarrow \beta_1^*$. To handle unary rules we modify the parsing algorithms in Figure 2.1 to include a unary completion rule,

$$\frac{(\langle i, j \rangle, h, \beta_1)}{(\langle i, j \rangle, h, A)}$$

for all indices $i \leq h \leq j$ that are consistent with the dependency parse. In order to avoid unary recursion, we limit the number of applications of this rule at each span (preserving the runtime of the algorithm). Preliminary experiments looked at collapsing the unary rules into the nonterminal symbols, but we found that this hurt performance compared to explicit unary rules.

2.2 Structured Prediction

We learn the dependency parse to phrase-structure parse conversion using a standard structured prediction setup. Define the linear scoring function s for a conversion as $s(y; x, d, \theta) = \theta^\top f(x, d, y)$ where θ is a parameter vector and $f(x, d, y)$ is a feature function that maps parse productions to sparse feature vectors. While the parser only requires a dependency parse at prediction time, the parameters of this scoring function are learned directly from a treebank of phrase-structure parses and a set of head rules. The structured prediction model, in effect, learns to invert the head rule transformation.

2.2.1 Features

The scoring function requires specifying a set of parse features f which, in theory, could be directly adapted from existing lexicalized phrase-structure parsers. However, the structure of the dependency parse greatly limits the number of decisions that need to be made, and allows for a smaller set of features.

We model our features after two bare-bones parsing systems. The first set is the basic arc-factored features used by McDonald (2006). These features include combinations of: rule and top nonterminal, modifier word and part-of-speech, and head word and part-of-speech.

The second set of features is modeled after the span features described in the X-bar-style parser of Hall et al. (2014). These include conjunctions of the rule with: first and last word of current span, preceding and following word of current span, adjacent words at split of current span, and binned length of the span.

For a production $\dfrac{(\langle i,k\rangle, m, \beta_1) \quad (\langle k+1,j\rangle, h, \beta_2)}{(\langle i,j\rangle, h, A)}$

Nonterm Features
(A, β_1) $(A, \beta_1, \text{tag}(m))$
(A, β_2) $(A, \beta_2, \text{tag}(h))$
Span Features
(rule, x_i) (rule, x_{i-1})
(rule, x_j) (rule, x_{j+1})
(rule, x_k) (rule, x_{k+1})
$(\text{rule}, \text{bin}(j-i))$

Rule Features
(rule)
$(\text{rule}, x_h, \text{tag}(m))$
$(\text{rule}, \text{tag}(h), x_m)$
$(\text{rule}, \text{tag}(h), \text{tag}(m))$
(rule, x_h)
$(\text{rule}, \text{tag}(h))$
(rule, x_m)
$(\text{rule}, \text{tag}(m))$

Figure 2.3: The feature templates used in the function $f(x, d, y)$. For the span features, the symbol rule is expanded into both $A \rightarrow B\ C$ and backoff symbol A. The function $\text{bin}(i)$ partitions a span length into one of 10 bins.

The full feature set is shown in Figure 2.3. After training, there are a total of around 2 million non-zero features. For efficiency, we use lossy feature hashing. We found this had no impact on parsing accuracy but made the parsing significantly faster.

2.2.2 Training

The parameters θ are estimated using a structural support vector machine (Taskar et al., 2004a). Given a set of gold-annotated phrase-structure parse examples, $(x^1, y^1), \ldots, (x^D, y^D)$, and dependency parses $d^1 \ldots d^D$ induced from the head rules, we estimate the parameters to minimize the regularized empirical risk

$$\min_{\theta} \sum_{i=1}^{D} \ell(x^i, d^i, y^i, \theta) + \lambda ||\theta||_1$$

where we define ℓ as $\ell(x, d, y, \theta) = -s(y) + \max_{y' \in \mathcal{Y}(x,d)} \left(s(y') + \Delta(y, y')\right)$ and where Δ is a problem specific cost-function. In experiments, we use a Hamming loss $\Delta(y, y') = |y - y'|$ where y is an indicator for production rules firing over pairs of adjacent spans (i.e., i, j, k).

The objective is optimized using AdaGrad (Duchi et al., 2011). The gradient calculation requires computing a cost-augmented max-scoring phrase-structure parse for each training example which is done using the algorithm of Figure 2.1 (right).

2.3 Methods

We ran a series of experiments to assess the accuracy, efficiency, and applicability of our parser, PAD, to several tasks. These experiments use the following setup.

For English experiments we use the standard Penn Treebank (PTB) experimental setup (Marcus et al., 1993). Training is done on §2–21, development on §22, and testing on §23. We use the development set to tune the regularization parameter, $\lambda = 1e-8$, and the pruning threshold, $\gamma = 0.95$.

For Chinese experiments, we use version 5.1 of the Penn Chinese Treebank 5.1 (CTB) (Xue et al., 2005). We followed previous work and used articles 001–270 and 440–1151 for training, 301–325 for development, and 271–300 for test. We also use the development set to tune the regularization parameter, $\lambda = 1e - 3$.

Part-of-speech tagging is performed for all models using TurboTagger (Martins et al., 2013). Prior to training the dependency parser, the training sections are automatically processed using 10-fold jackknifing (Collins and Koo, 2005) for both dependency and phrase structure trees. Zhu et al. (2013) found this simple technique gives an improvement to dependency accuracy of 0.4% on English and 2.0% on Chinese in their system.

During training, we use the dependency parses induced by the head rules from the gold phrase-structure parses as constraints. There is a slight mismatch here with test, since these dependency parses are guaranteed to be consistent with the target phrase-structure parse. We also experimented with using 10-fold jacknifing of the dependency parser during training to produce more realistic parses; however, we found that this hurt performance of the parser.

Unless otherwise noted, in English the test dependency parsing is done using the RedShift implementation[3] of the parser of Zhang and Nivre (2011), trained to follow the conventions of Collins head rules (Collins, 2003). This parser is a transition-based beam search parser, and the size of the beam k controls a speed/accuracy trade-off. By default we use a beam of $k = 16$. We found that dependency labels have a significant impact on the performance of the RedShift parser, but not on English dependency conversion. We therefore train a labeled parser, but discard the labels.

For Chinese, we use the head rules compiled by Ding and Palmer (2005)[4]. For this data-set we trained the dependency parser using the YaraParser implementation[5] of the parser of Zhang and Nivre (2011), because it has a better Chinese implementation. We use a beam of $k = 64$. In experiments, we found that Chinese labels were quite helpful, and added four additional features templates conjoining the label with the non-terminals of a rule.

Evaluation for phrase-structure parses is performed using the `evalb`[6] script with the standard setup. We report labeled F_1 scores as well as recall and precision. For dependency parsing, we report unlabeled accuracy score (UAS).

We implemented the grammar binarization, head rules, and pruning tables in Python, and the parser, features, and training in C++. Experiments are performed on a Lenovo ThinkCentre desktop computer with 32GB of memory and Core i7-3770 3.4GHz 8M cache CPU.

2.4 Experiments

We ran experiments to assess the accuracy of the method, its runtime efficiency, the effect of dependency parsing accuracy, and the effect of the amount of annotated phrase-structure data.

[3] `https://github.com/syllog1sm/redshift`
[4] `http://stp.lingfil.uu.se/~nivre/research/chn_headrules.txt`
[5] `https://github.com/yahoo/YaraParser`
[6] `http://nlp.cs.nyu.edu/evalb`

Model	PTB §23 F_1	PTB §23 Sent./s.
Charniak (2000)	89.5	–
Stanford PCFG (2003)	85.5	5.3
Petrov (2007)	90.1	8.6
Zhu (2013)	90.3	39.0
Carreras (2008)	91.1	–
CJ Reranking (2005)	91.5	4.3
Stanford RNN (2013)	90.0	2.8
PAD	90.4	34.3
PAD (Pruned)	90.3	58.6

Model	CTB F_1
Charniak (2000)	80.8
Bikel (2004)	80.6
Petrov (2007)	83.3
Zhu (2013)	83.2
PAD	82.4

Table 2.2: Accuracy and speed on PTB §23 and CTB 5.1 test split. Comparisons are to state-of-the-art non-reranking supervised phrase-structure parsers (Charniak, 2000; Klein and Manning, 2003; Petrov and Klein, 2007; Carreras et al., 2008; Zhu et al., 2013; Bikel, 2004), and semi-supervised and reranking parsers (Charniak and Johnson, 2005; Socher et al., 2013).

Model	UAS	F_1	Sent./s.	Oracle
MALTPARSER	89.7	85.5	240.7	87.8
RS-K1	90.1	86.6	233.9	87.6
RS-K4	92.5	90.1	151.3	91.5
RS-K16	93.1	90.6	58.6	92.5
YARA-K1	89.7	85.3	1265.8	86.7
YARA-K16	92.9	89.8	157.5	91.7
YARA-K32	93.1	90.4	48.3	92.0
YARA-K64	93.1	90.5	47.3	92.2
TP-BASIC	92.8	88.9	132.8	90.8
TP-STANDARD	93.3	90.9	27.2	92.6
TP-FULL	93.5	90.8	13.2	92.9

Figure 2.4: The effect of dependency parsing accuracy (PTB §22) on PAD and an oracle converter. Runtime includes dependency parsing and phrase-structure parsing. Inputs include MaltParser (Nivre et al., 2006), the RedShift and the Yara implementations of the parser of Zhang and Nivre (2011) with various beam size, and three versions of TurboParser trained with projective constraints (Martins et al., 2013).

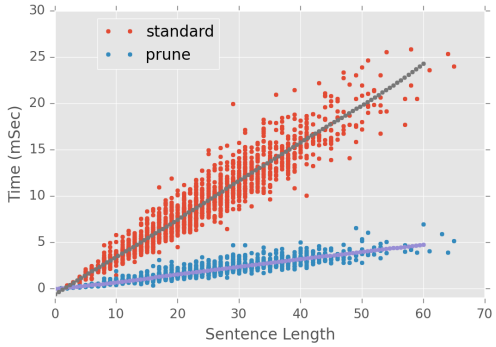


Figure 2.5: Empirical runtime of the parser on sentences of varying length, with and without pruning. Despite a worst-case quadratic complexity, observed runtime is linear.

Parsing Accuracy Table 2.2 compares the accuracy and speed of the phrase-structure trees produced by the parser. For these experiments we treat our system and the Zhang-Nivre parser as an independently trained, but complete end-to-end phrase-structure parser. Runtime for these experiments includes both the time for dependency parsing and conversion. Despite the fixed dependency constraints, the English results show that the parser is comparable in accuracy to many widely-used systems, and is significantly faster. The parser most competitive in both speed and accuracy is that of Zhu et al. (2013), a fast shift-reduce phrase-structure parser.

Furthermore, the Chinese results suggest that, even without making language-specific changes in the feature system we can still achieve competitive parsing accuracy.

Effect of Dependencies Table 2.4 shows experiments comparing the effect of different input dependency parses. For these experiments we used the same version of PAD with 11 different dependency parsers of varying quality and speed. We measure for each parser: its UAS, speed, and labeled F_1 when used with PAD and with an oracle converter.[7] The paired figure shows that there is a direct correlation between the UAS of the inputs and labeled F_1.

Runtime In Section 2.1 we considered the theoretical complexity of the parsing model and presented the main speed results in Table 2.1. Despite having a quadratic theoretical complexity, the practical runtime was quite fast. Here we consider the empirical complexity of the model by measuring the time spent on individual sentences. Figure 2.5 shows parser speed for sentences of varying length for both the full algorithm and with pruning. In both cases the observed runtime is linear.

2.5 Proposed Work

We will extend this work, making the model more expressive through non-linear (neural) representation learning. This can be potentially helpful when the model adopts semantics oriented

[7]For a gold parse y and predicted dependencies $\hat{d}$, define the oracle parse as $y' = \arg\min_{y' \in \mathcal{Y}(x,\hat{d})} \Delta(y, y')$

head-rules (e.g. Stanford head rules (De Marneffe and Manning, 2008)) where the context becomes more non-local.

Soft structural constraints in the model can be understood as a regularizer of learned representations (Zhang and Weiss, 2016). We will extend the previous work by exploring this general approach to incorporating the structural information in neural models.

Chapter 3

Segmental Recurrent Neural Networks

Our prior work (Kong et al., 2015a; Lu et al., 2016) introduces **segmental recurrent neural networks** (SRNNs) which define, given an input sequence, a joint probability distribution over segmentations of the input and labelings of the segments. Representations of the input segments (i.e., contiguous subsequences of the input) are computed by encoding their constituent tokens using bidirectional recurrent neural nets, and these "segment embeddings" are used to define compatibility scores with output labels. These local compatibility scores are integrated using a global semi-Markov conditional random field. Both fully supervised training—in which segment boundaries and labels are observed—as well as partially supervised training—in which segment boundaries are latent—are straightforward. Experiments show that, compared to models that do not explicitly represent segments such as BIO tagging schemes and connectionist temporal classification (CTC), SRNNs obtain substantially higher accuracies.

3.1 Model

Given a sequence of input observations $\boldsymbol{x} = \langle \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{|\boldsymbol{x}|} \rangle$ with length $|\boldsymbol{x}|$, a **segmental recurrent neural network** (SRNN) defines a joint distribution $p(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{x})$ over a sequence of labeled segments each of which is characterized by a duration ($z_i \in \mathbb{Z}_+$) and label ($y_i \in Y$). The segment durations constrained such that $\sum_{i=1}^{|\boldsymbol{z}|} z_i = |\boldsymbol{x}|$. The length of the output sequence $|\boldsymbol{y}| = |\boldsymbol{z}|$ is a random variable, and $|\boldsymbol{y}| \leq |\boldsymbol{x}|$ with probability 1. We write the starting time of segment i as $s_i = 1 + \sum_{j<i} z_j$.

To motivate our model form, we state several desiderata. First, we are interested in the following prediction problem,

$$\boldsymbol{y}^* = \arg\max_{\boldsymbol{y}} p(\boldsymbol{y} \mid \boldsymbol{x}) = \arg\max_{\boldsymbol{y}} \sum_{\boldsymbol{z}} p(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{x}) \approx \arg\max_{\boldsymbol{y}} \max_{\boldsymbol{z}} p(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{x}). \tag{3.1}$$

Note the use of joint maximization over $\boldsymbol{y}$ and $\boldsymbol{z}$ as a computationally tractable substitute for marginalizing out $\boldsymbol{z}$; this is commonly done in natural language processing.

Second, for problems where the explicit durations observations are unavailable at training time and are inferred as a latent variable, we must be able to use a marginal likelihood training

criterion,

$$\mathcal{L} = -\log p(\boldsymbol{y} \mid \boldsymbol{x}) = -\log \sum_{\boldsymbol{z}} p(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{x}). \tag{3.2}$$

In Eqs. 3.1 and 3.2, the conditional probability of the labeled segment sequence is (assuming kth order dependencies on $\boldsymbol{y}$):

$$p(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \prod_{i=1}^{|\boldsymbol{y}|} \exp f(y_{i-k:i}, z_i, \mathbf{x}) \tag{3.3}$$

where $Z(\boldsymbol{x})$ is an appropriate normalization function. To ensure the expressiveness of f and the computational efficiency of the maximization and marginalization problems in Eqs. 3.1 and 3.2, we use the following definition of f,

$$\begin{aligned} f(y_{i-k:i}, z_i, \mathbf{x}_{s_i:s_i+z_i-1}) = \mathbf{w}^\top \phi(\mathbf{V}[\mathbf{g}_y(y_{i-k}); \ldots; \mathbf{g}_y(y_i); \mathbf{g}_z(z_i); \\ \overrightarrow{\text{RNN}}(\mathbf{c}_{s_i:s_i+z_i-1}); \overleftarrow{\text{RNN}}(\mathbf{c}_{s_i:s_i+z_i-1})] + \mathbf{a}) + b \end{aligned} \tag{3.4}$$

where $\overrightarrow{\text{RNN}}(\mathbf{c}_{s_i:s_i+z_i-1})$ is a recurrent neural network that computes the **forward segment embedding** by "encoding" the z_i-length subsequence of $\boldsymbol{x}$ starting at index s_i,[1] and $\overleftarrow{\text{RNN}}$ computes the reverse segment embedding (i.e., traversing the sequence in reverse order), and $\mathbf{g}_y$ and $\mathbf{g}_z$ are functions which map the label candidate $\boldsymbol{y}$ and segmentation duration $\boldsymbol{z}$ into a vector representation. The notation $[\mathbf{a}; \mathbf{b}; \mathbf{c}]$ denotes vector concatenation. Finally, the concatenated segment duration, label candidates and segment embedding are passed through a affine transformation layer parameterized by $\mathbf{V}$ and $\mathbf{a}$ and a nonlinear activation function ϕ (e.g., $\tanh$), and a dot product with a vector $\mathbf{w}$ and addition by scalar b computes the log potential for the clique. Our proposed model is equivalent to a semi-Markov conditional random field with local features computed using neural networks. Figure 3.1 shows the model graphically.

We chose bidirectional LSTMs (Graves and Schmidhuber, 2005) as the implementation of the RNNs in Eq. 3.4. LSTMs (Hochreiter and Schmidhuber, 1997) are a popular variant of RNNs which have been seen successful in many representation learning problems (Graves and Jaitly, 2014; Karpathy and Fei-Fei, 2015). Bidirectional LSTMs enable effective computation for embedings in both directions and are known to be good at preserving long distance dependencies, and hence are well-suited for our task.

3.2 Inference with Dynamic Programming

We are interested in three inference problems: (i) finding the most probable segmentation/labeling for a model given a sequence $\boldsymbol{x}$; (ii) evaluating the partition function $Z(\boldsymbol{x})$; and (iii) computing the posterior marginal $Z(\boldsymbol{x}, \boldsymbol{y})$, which sums over all segmentations compatible with a reference

[1] Rather than directly reading the $\mathbf{x}_i$'s, each token is represented as the concatenation, $\mathbf{c}_i$, of a forward and backward over the sequence of raw inputs. This permits tokens to be sensitive to the contexts they occur in, and this is standardly used with neural net sequence labeling models (Graves et al., 2006b).

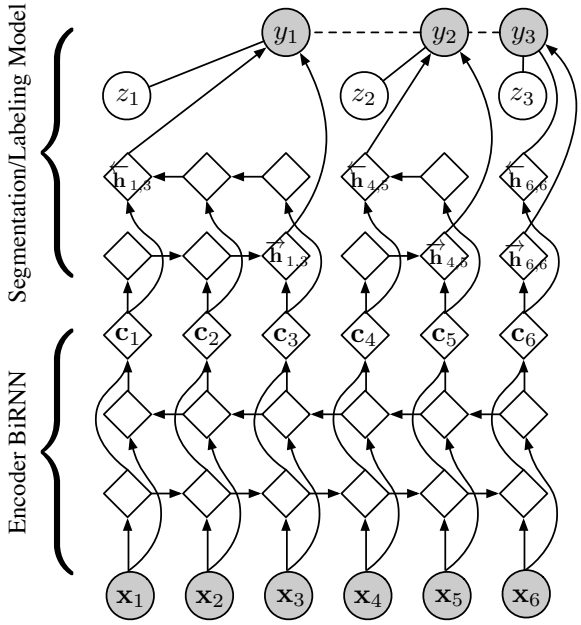


Figure 3.1: Graphical model showing a six-frame input and three output segments with durations $\boldsymbol{z} = \langle 3, 2, 1 \rangle$ (this particular setting of $\boldsymbol{z}$ is shown only to simplify the layout of this figure; the model assigns probabilities to all valid settings of $\boldsymbol{z}$). Circles represent random variables. Shaded nodes are observed in training; open nodes are latent random variables; diamonds are deterministic functions of their parents; dashed lines indicate optional statistical dependencies that can be included at the cost of increased inference complexity. The graphical notation we use here draws on conventions used to illustrate neural networks and graphical models.

sequence $\boldsymbol{y}$. These can all be solved using dynamic programming. For simplicity, we will assume zeroth order Markov dependencies between the y_is. Extensions to the kth order Markov dependencies should be straightforward. Since each of these algorithms relies on the forward and reverse segment embeddings, we first discuss how these can be computed before going on to the inference algorithms.

3.2.1 Computing Segment Embeddings

Let the $\overrightarrow{\mathbf{h}}_{i,j}$ designate the $\overrightarrow{\mathrm{RNN}}$ encoding of the input span (i, j), traversing from left to right, and let $\overleftarrow{\mathbf{h}}_{i,j}$ designate the reverse direction encoding using $\overleftarrow{\mathrm{RNN}}$. There are thus $O(|\boldsymbol{x}|^2)$ vectors that must be computed, each of length $O(|\boldsymbol{x}|)$. Naively this can be computed in time $O(|\boldsymbol{x}|^3)$, but the following dynamic program reduces this to $O(|\boldsymbol{x}|^2)$:

$$\begin{aligned}
\overrightarrow{\mathbf{h}}_{i,i} &= \overrightarrow{\mathrm{RNN}}(\overrightarrow{\mathbf{h}}_{0}, \mathbf{c}_i) \\
\overrightarrow{\mathbf{h}}_{i,j} &= \overrightarrow{\mathrm{RNN}}(\overrightarrow{\mathbf{h}}_{i,j-1}, \mathbf{c}_j) \\
\overleftarrow{\mathbf{h}}_{i,i} &= \overleftarrow{\mathrm{RNN}}(\overleftarrow{\mathbf{h}}_{0}, \mathbf{c}_i) \\
\overleftarrow{\mathbf{h}}_{i,j} &= \overleftarrow{\mathrm{RNN}}(\overleftarrow{\mathbf{h}}_{i+1,j}, \mathbf{c}_i)
\end{aligned}$$

The algorithm is executed by initializing in the values on the diagonal (representing segments of length 1) and then inductively filling out the rest of the matrix. In practice, we often can put a upper bound for the length of a eligible segment thus reducing the complexity of runtime to $O(|\boldsymbol{x}|)$. This savings can be substantial for very long sequences (e.g., those encountered in speech recognition).

3.2.2 Computing the most probable segmentation/labeling and $Z(\boldsymbol{x})$

For the input sequence $\boldsymbol{x}$, there are $2^{|\boldsymbol{x}|-1}$ possible segmentations and $O(|Y|^{|\boldsymbol{x}|})$ different labelings of these segments, making exhaustive computation entirely infeasible. Fortunately, the partition function $Z(\boldsymbol{x})$ may be computed in polynomial time with the following dynamic program:

$$\alpha_0 = 1$$
$$\alpha_j = \sum_{i<j} \alpha_i \times \sum_{y \in Y} \left(\exp \mathbf{w}^\top \phi(\mathbf{V}[g_y(y); g_z(z_i); \overrightarrow{\text{RNN}}(\mathbf{c}_{s_i:s_i+z_i-1}); \overleftarrow{\text{RNN}}(\mathbf{c}_{s_i:s_i+z_i-1})] + \mathbf{a}) + b \right).$$

After computing these values, $Z(\boldsymbol{x}) = \alpha_{|\boldsymbol{x}|}$. By changing the summations to a $\max$ operators (and storing the corresponding $\arg\max$ values), the maximal *a posteriori* segmentation/labeling can be computed.

Both the partition function evaluation and the search for the MAP outputs run in time $O(|\boldsymbol{x}|^2 \cdot |Y|)$ with this dynamic program. Adding nth order Markov dependencies between the y_is adds requires additional information in each state and increases the time and space requirements by a factor of $O(|Y|^n)$. However, this may be tractable for small $|Y|$ and n.

Avoiding overflow. Since this dynamic program sums over exponentially many segmentations and labelings, the values in the α_i chart can become very large. Thus, to avoid issues with overflow, computations of the α_i's must be carried out in log space.[2]

3.2.3 Computing $Z(\boldsymbol{x}, \boldsymbol{y})$

To compute the posterior marginal $Z(\boldsymbol{x}, \boldsymbol{y})$, it is necessary to sum over all segmentations that are compatible with a label sequence $\boldsymbol{y}$ given an input sequence $\boldsymbol{x}$. To do so requires only a minor modification of the previous dynamic program to track how much of the reference label sequence $\boldsymbol{y}$ has been consumed. We introduce the variable m as the index into $\boldsymbol{y}$ for this purpose. The

[2] An alternative strategy for avoiding overflow in similar dynamic programs is to rescale the forward summations at each time step (Rabiner, 1989; Graves et al., 2006b). Unfortunately, in a semi-Markov architecture each term in α_i sums over different segmentations (e.g., the summation for α_2 will have contain some terms that include α_1 and some terms that include only α_0), which means there are no common factors, making this strategy inapplicable.

modified recurrences are:

$$\gamma_0(0) = 1$$
$$\gamma_j(m) = \sum_{i<j} \gamma_i(m-1) \times \left(\exp \mathbf{w}^\top \phi(\mathbf{V}[g_y(y_i); g_z(z_i); \overrightarrow{\text{RNN}}(\mathbf{c}_{s_i:s_i+z_i-1}); \overleftarrow{\text{RNN}}(\mathbf{c}_{s_i:s_i+z_i-1})] + \mathbf{a}) + b\right).$$

The value $Z(\boldsymbol{x}, \boldsymbol{y})$ is $\gamma_{|\boldsymbol{x}|}(|\boldsymbol{y}|)$.

3.3 Parameter Learning

We consider two different learning objectives.

Supervised learning In the supervised case, both the segment durations ($\boldsymbol{z}$) and their labels ($\boldsymbol{y}$) are observed.

$$\begin{aligned}\mathcal{L} &= \sum_{(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})\in\mathcal{D}} -\log p(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{x}) \\ &= \sum_{(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})\in\mathcal{D}} \log Z(\boldsymbol{x}) - \log Z(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})\end{aligned}$$

In this expression, the unnormalized conditional probability of the reference segmentation/labeling, given the input $\boldsymbol{x}$ is written as $Z(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$.

Partially supervised learning In the partially supervised case, only the labels are observed and the segments (the $\boldsymbol{z}$) are unobserved and marginalized.

$$\begin{aligned}\mathcal{L} &= \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{D}} -\log p(\boldsymbol{y} \mid \boldsymbol{x}) \\ &= \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{D}} \sum_{\boldsymbol{z}\in\mathcal{Z}(\boldsymbol{x},\boldsymbol{y})} -\log p(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{x}) \\ &= \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{D}} \log Z(\boldsymbol{x}) - \log Z(\boldsymbol{x}, \boldsymbol{y})\end{aligned}$$

For both the fully and partially supervised scenarios, the necessary derivatives can be computed using automatic differentiation or (equivalently) with backward variants of the above dynamic programs (Sarawagi and Cohen, 2004).

3.4 Further Speedup

It is computationally expensive for RNNs to model long sequences, and the number of possible segmentations is exponential with the length of the input sequence as mentioned before. The

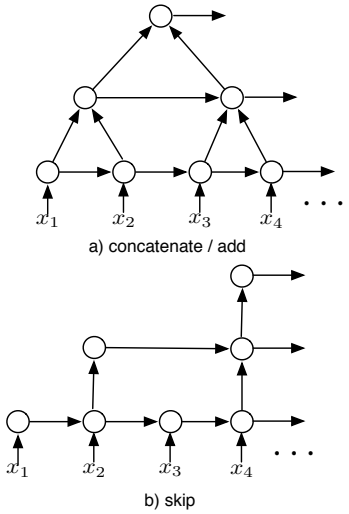


Figure 3.2: Hierarchical subsampling recurrent network (Graves, 2012) . The size of the subsampling window is two in this example.

computational cost can be significantly reduced by using the hierarchical subsampling RNN (Graves, 2012) to shorten the input sequences, where the subsampling layer takes a window of hidden states from the lower layer as input as shown in Figure 3.2. In our speech experiments (§3.5.2), we consider three variants: a) *concatenate* – the hidden states in the subsampling window are concatenated before been fed into the next layer; b) *add* – the hidden states are added into one vector for the next layer; c) *skip* – only the last hidden state in the window is kept and all the others are skipped. The last two schemes are computationally cheaper as they do not introduce extra model parameters.

3.5 Experiments

3.5.1 Online Handwriting Recognition

Dataset We use the handwriting dataset from (Kassel, 1995). This dataset is an online collection of hand-written words from 150 writers. It is recorded as the coordinates (x, y) at time t plus special pen-down/pen-up notations. We break the coordinates into strokes using the pen-down and pen-up notations. One character typically consists one or more contiguous strokes.[3]

The dataset is split into train, development and test set following (Kassel, 1995). Table 3.1 presents the statistics for the dataset.

A well-know variant of this dataset was introduced by Taskar et al. (2004b). Taskar et al. (2004b) selected a "clean" subset of about 6,100 words and rasterized and normalized the images of each letter. Then, the uppercased letters (since they are usually the first character in a word) are removed and only the lowercase letters are used. The main difference between our dataset and theirs is that their dataset is "offline" — Taskar et al. (2004b) mapped each character into

[3]There are infrequent cases where one stroke can go across multiple characters or the strokes which form the character can be not contiguous. We leave those cases for future work.

	#words	#characters
Train	4,368	37,247
Dev	1,269	10,905
Test	637	5,516
Total	6,274	53,668

Table 3.1: Statistics of the Online Handwriting Recognition Dataset

a bitmap and treated the segmentation of characters as a preprocessing step. We use the richer representation of the sequence of strokes as input.

Implementation We trained two versions of our model on this dataset, namely, the fully supervised model (§3.3), which takes advantage of the gold segmentations on training data, and the partially supervised model (§3.3) in which the gold segmentations are only used in the evaluation. A CTC model reimplemented on the top of our Encoder BiRNNs layer (Figure 3.1) is used as a baseline so that we can see the effect of explicitly representing the segmentation.[4] For the decoding of the CTC model, we simply use the best path decoding, where we assume that the most probable path will correspond to the most probable labeling, although it is known that prefix search decoding can slightly improve the results (Graves et al., 2006b).

As a preprocessing step, we first represented each point in the dataset using a 4 dimensional vector, $\mathbf{p} = (p_x, p_y, \Delta p_x, \Delta p_y)$, where p_x and p_y are the normalized coordinates of the point and Δp_x and Δp_y are the corresponding changes in the coordinates with respect to the previous point. Δp_x and Δp_y are meant to capture basic direction information. Then we map the points inside one stroke into a fixed-length vector using a bi-direction LSTM. Specifically, we concatenated the last position's hidden states in both directions and use it as the input vector $\mathbf{x}$ for the stroke.

In all the experiments, we use Adam (Kingma and Ba, 2014) with $\lambda = 1 \times 10^{-6}$ to optimize the parameters in the models. We train these models until convergence and picked the best model over the iterations based on development set performance then report performance on the test set.

We used 5 as the hidden state dimension in the bidirectional RNNs, which map the points into fixed-length stroke embeddings (hence the input vector size $5 \times 2 = 10$). We set the hidden dimensions of $\mathbf{c}$ in our model and CTC model to 24 and segment embedding $\mathbf{h}$ in our model as 18. These dimensions were chosen based on intuitively reasonable values, and it was confirmed on development data that they performed well. We tried to experiment with larger hidden dimensions and we found the performance did not vary much. Future work might more carefully optimize these parameters.

The results of the online handwriting recognition task are presented in Table 3.2. We see that both of our models outperform the baseline CTC model, which does not carry an explicit representation for the segments being labeled, by a significant margin. An interesting finding is, although the partially supervised model performs slightly worse in the development set, it

[4]The CTC interpretation rules specify that repeated symbols, e.g. `aa` will be interpreted as a single token of `a`. However since the segments in the handwriting recognition problem are extremely short, we use different rules and interpret this as `aa`. That is, only the blank symbol may be used to represent extended durations. Our experiments indicate this has little effect, and Graves (p.c.) reports that this change does not harm performance in general.

	Dev				Test			
	P_{seg}	R_{seg}	F_{seg}	Error	P_{seg}	R_{seg}	F_{seg}	Error
SRNNs (Partial)	98.7%	98.4%	98.6%	4.2%	99.2%	99.1%	99.2%	2.7%
SRNNs (Full)	98.9%	98.6%	98.8%	4.3%	98.8%	98.6%	98.6%	5.4%
CTC	-	-	-	15.2%	-	-	-	13.8%

Table 3.2: Hand-writing Recognition Task

subsampling	L	speedup
No	30	1
1 layer	15	$\sim$3x
2 layers	8	$\sim$10x

Table 3.3: Speedup by hierarchical subsampling networks.

actually outperforms the fully supervised model in the test set. Because the test set is written by different people from the train and development set, they exhibit different styles in their handwriting; our results suggest that the partially supervised model may generalize better across different writing styles.

3.5.2 End-to-end Speech Recognition

Hierarchical Subsampling We first demonstrate the results of the hierarchical subsampling recurrent network, which is the key to speed up our experiments. We set the size of the subsampling window to be 2, therefore each subsampling layer reduced the time resolution by a factor of 2. We set the maximum segment length (§3.2.1) to be 300 milliseconds, which corresponded to 30 frames of FBANKs (sampled at the rate of 10 milliseconds). With two layers of subsampling recurrent networks, the time resolution was reduced by a factor of 4, and the value of L was reduced to be 8, yielding around 10 times speedup as shown in Table 3.3.

Table 3.4 compares the three implementations of the recurrent subsampling network detailed in section 3.4. We observed that concatenating all the hidden states in the subsampling window did not yield lower phone error rate (PER) than using the simple *skipping* approach, which may

System	$d(\mathbf{w})$	$d(\mathbf{V})$	layers	hidden	PER(%)
`skip`	64	64	3	128	21.2
`conc`	64	64	3	128	21.3
`add`	64	64	3	128	23.2
`skip`	64	64	3	250	20.1
`conc`	64	64	3	250	20.5
`add`	64	64	3	250	21.5

Table 3.4: Results of hierarchical subsampling networks. $d(\mathbf{w})$ and $d(\mathbf{V})$ denote the dimension of $\mathbf{w}$ and $\mathbf{V}$ in Eqs. (3.4) respectively. `layers` denotes the number of LSTM layers and `hidden` is the dimension of the LSTM cells. `conc` is short for concatenating operation in the subsampling network.

Table 3.5: Results of tuning the hyperparameters.

Dropout	$d(\mathbf{w})$	$d(\mathbf{V})$	layers	hidden	PER
0.2	64	64	3	128	21.2
	64	32	3	128	21.6
	32	32	3	128	21.4
	64	64	3	250	20.1
	64	32	3	250	20.4
	32	32	3	250	20.6
	64	64	6	250	19.3
	64	32	6	250	20.2
	32	32	6	250	20.2
0.1	64	64	3	128	21.3
	64	64	3	250	20.9
	64	64	6	250	20.4
$\times$	64	64	6	250	21.9

Table 3.6: Results of three types of acoustic features.

Features	Deltas	$d(\mathbf{x})$	PER
24-dim FBANK	$\surd$	72	19.3
40-dim FBANK	$\surd$	120	18.9
Kaldi	$\times$	40	17.3

be due to the fact that the TIMIT dataset is small and it prefers a smaller model. On the other hand, adding the hidden states in the subsampling window together worked even worse, possibly due to that the sequential information in the subsampling window was flattened. In the following experiments, we sticked to the *skipping* method, and using two subsampling layers.

Hyperparameters We then evaluated the model by tuning the hyperparameters, and the results are given in Table 3.5. We tuned the number of LSTM layers, and the dimension of LSTM cells, as well as the dimensions of $\mathbf{w}$ and the segment vector $\mathbf{V}$. In general, larger models with dropout regularisation yielded higher recognition accuracy. Our best result was obtained using 6 layers of 250-dimensional LSTMs. However, without the dropout regularisation, the model can be easily overfit due to the small size of training set.

Features We then evaluated another two types of features using the same system configuration that achieved the best result in Table 3.5. We increased the number of FBANKs from 24 to 40, which yielded slightly lower PER. We also evaluated the standard Kaldi features — 39 dimensional MFCCs spliced by a context window of 7, followed by LDA and MLLT transform and with feature-space speaker-dependent MLLR, which were the same features used in the HMM-DNN baseline in Table 3.7. The well-engineered features improved the accuracy of our system by more than 1% absolute.

Results In Table 3.7, we compare our result to other reported results using segmental CRFs as well as recent end-to-end systems. Previous state-of-the-art result using segmental CRFs on the

System	LM	SD	PER
HMM-DNN	$\surd$	$\surd$	18.5
first-pass SCRF (Zweig, 2012)	$\surd$	$\times$	33.1
Boundary-factored SCRF (He and Fosler-Lussier, 2012)	$\times$	$\times$	26.5
Deep Segmental NN (Abdel-Hamid et al., 2013)	$\surd$	$\times$	21.9
Discriminative segmental cascade (Tang et al., 2015)	$\surd$	$\times$	21.7
+ 2nd pass with various features	$\surd$	$\times$	19.9
CTC (Graves et al., 2013)	$\times$	$\times$	18.4
RNN transducer (Graves et al., 2013)	-	$\times$	17.7
Attention-based RNN baseline (Chorowski et al., 2015)	$\times$	$\times$	18.7
+Conv. Features + Smooth Focus (Chorowski et al., 2015)	$\times$	$\times$	17.6
Segmental RNN	$\times$	$\times$	18.9
Segmental RNN	$\times$	$\surd$	17.3

Table 3.7: Comparison to Related Works. `LM` denote if the language model is used, and `SD` denotes feature space speaker-dependent transform. The HMM-DNN baseline was trained with cross-entropy using the Kaldi recipe. Sequence training did not improve it due to the small amount of data. RNN transducer can be viewed as a combination of the CTC network with a built-in RNN language model.

TIMIT dataset is reported in (Tang et al., 2015), where the first-pass decoding was used to prune the search space, and the second-pass was used to re-score the hypothesis using various features including neural network features. Besides, the ground-truth segmentation was used in (Tang et al., 2015). We achieved considerably lower PER with first-pass decoding, despite the fact that our CRF was zeroth-order, and we did not use any language model. Furthermore, our results are also comparable to that from the CTC and attention-based RNN end-to-end systems. The accuracy of segmental RNNs may be further improved by using higher-order CRFs or incorporating a language model into the decode step, using beam search to reduce the search error.

3.6 Conclusion

We have proposed a new model for segment labeling problems that learns explicit representations of segments of an input sequence. Experiments show that, compared to models that do not explicitly represent segments such as BIO tagging schemes and connectionist temporal classification (CTC), SRNNs obtain substantially higher accuracies.

Chapter 4

Modeling the Compositionality in CCG Supertagging

Combinatory categorical grammar (CCG) is widely used in semantic parsing, and parsing with it is very fast, especially if a good (probabilistic) tagger is available (Lewis and Steedman, 2014a; Lee et al., 2016). An example of a parse tree of CCG is shown in Figure 4.1 Each lexical token is associated with a structured category (i.e., supertag) in CCG. The number of lexical categories is unbounded in theory. In practice, previous work (Lewis and Steedman, 2014a; Xu et al., 2015b; Lewis and Steedman, 2014b, *inter alia*) use over 400 supertags and treat them as discrete categories, while Penn Treebank (Marcus et al., 1993) only has 50 predefined POS tags.

We propose to model the compositionality (§4.1) both inside these tags (*intra-*) and between these tags (*inter-*). Previous works suggests that explicit modeling of such information can leads to better accuracy (Srikumar and Manning, 2014) and less supervision (Garrette et al., 2015).

4.1 CCG and Supertagging

A CCG category ($\mathcal{C}$) follows the recursive definition:

$$\begin{aligned} \mathcal{C} &\rightarrow \{\text{S, N, NP, ...}\} \\ \mathcal{C} &\rightarrow \{\mathcal{C}/\mathcal{C}, \mathcal{C}\backslash\mathcal{C}\} \end{aligned}$$

It can either be an *atomic* category representing basic grammatical phrase (e.g. S, N, NP) or a *complex* category formed from the *combination* of two categories by one of the two *slash* operators (i.e. $/$ and $\backslash$).

CCG supertags can be *combined* recursively to form a *parse* of a sentence. There are only five rules used for *combining* the CCG supertags (Figure 4.2) in the CCGBank (Hockenmaier and Steedman, 2007).

4.2 Supertagging Transition System

We first introduce a novel supertagging transition system, which constructs a supertag in a top-down manner, following the recursive definition of the CCG categories (§4.1). This transition

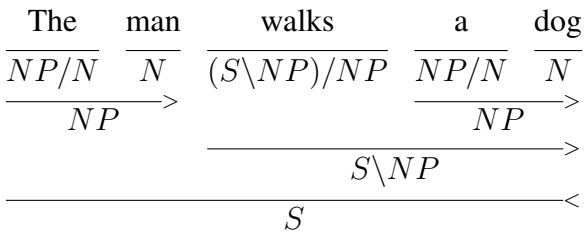


Figure 4.1: CCG parse for "The man walks a dog."

$$
\begin{array}{rrcrr}
\mathcal{X}/\mathcal{Y} & \mathcal{Y} & \Rightarrow & \mathcal{X} & (>) \\
\mathcal{Y} & \mathcal{X}\backslash\mathcal{Y} & \Rightarrow & \mathcal{X} & (<) \\
\mathcal{X}/\mathcal{Y} & \mathcal{Y}/\mathcal{Z} & \Rightarrow & \mathcal{X}/\mathcal{Z} & (> \mathbf{B}) \\
\mathcal{Y}\backslash\mathcal{Z} & \mathcal{X}\backslash\mathcal{Y} & \Rightarrow & \mathcal{X}\backslash\mathcal{Z} & (< \mathbf{B}) \\
\mathcal{Y}/\mathcal{Z} & \mathcal{X}\backslash\mathcal{Y} & \Rightarrow & \mathcal{X}/\mathcal{Z} & (< \mathbf{B}_{\times})
\end{array}
$$

Figure 4.2: Combination rules of CCG categories. Five rules are used in the CCG Bank, namely, forward application ($>$), backward application ($<$), forward composition ($> \mathbf{B}$), backward composition ($< \mathbf{B}$), and Backward crossing composition ($< \mathbf{B}_{\times}$).

system is inspired by the *unigram category generator* (Garrette et al., 2015) in the probabilistic grammar setting. Similar to the transition system for dependency parsing (Nivre, 2008), it uses a stack to keep track of the construction process.

Tagging Transitions There are 3 classes of transitions during the construction of a supertag.

- / (or \) transition introduces a "open binary CCG category" with the combinator / (or \) and pushes it onto the stack.
- REDUCE transition pops the top two completed CCG category (either *basic* or *complex*). Then the open binary CCG category is popped and used to combine the two completed categories. If the stack is not empty, this combined supertag will be pushed on to the stack. Otherwise, this combined supertag is the final result and the construction process ends.
- C transitions (where $X \in \{\text{S, N, NP, ...}\}$) pushes a *terminal* CCG category (i.e. basic grammatical phrase) C on the stack.

Model With the Tagging transition system, we can build a discriminative left-to-right tagger with the neural architecture illustrated in Figure 4.3. We use Stack LSTMs (Dyer et al., 2015) to model the stack structures and Bi-directional LSTMs to model the encoder of the lexical inputs. Three components are used when computing the probability of a single tagging transition $p(a_\tau)$.

- A stack (S_t) is the stack described in the transition system which is used to store partially built structures within a supertag.
- A (optional) stack (S_c) saves the history of completed CCG supertags built from the previous words in the sentence. After successfully built a supertag from S_t, the compositional representation of that tag will be pushed onto S_c.

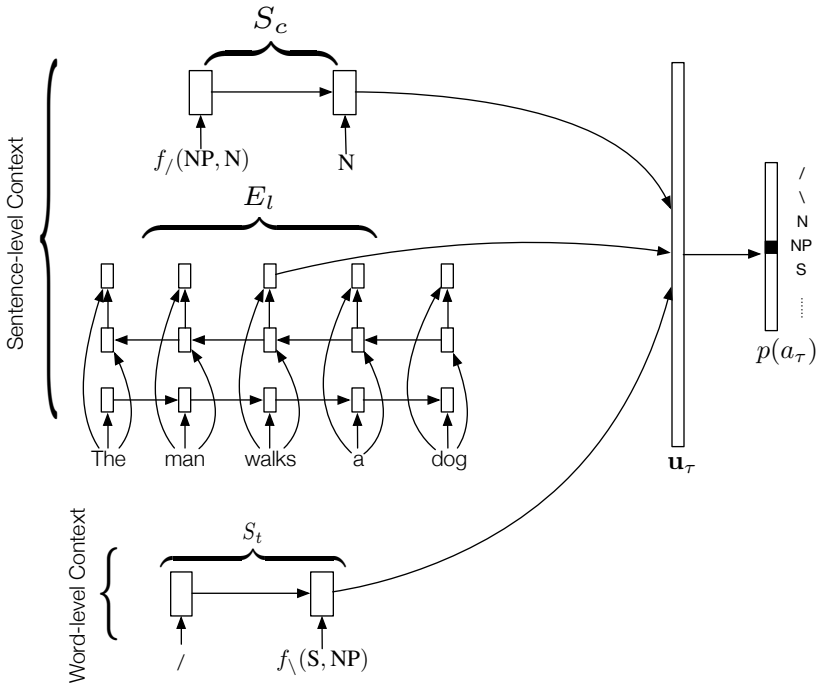


Figure 4.3: Neural architecture for defining a distribution over a_τ at given representations of the stack representing the previous completed supertags (S_c), the Bi-LSTM encoder of the current input words (E_l) and the stack stores the partial structure of the current building supertag (S_t). $f_/$ and $f_\backslash$ are the combination function we used inside the supertags for *forward*, *backward* encoding. In this example, we are about to build the rightmost NP part in the tag $(\mathrm{S}\backslash\mathrm{NP})/\mathrm{NP}$ of the word *walks* in the example in Figure 4.1.

- An encoder (E_l) is used to encode the words in the sentence. It provides the representation of the word when building the supertag.

The probability of the sequence of supertags ($\mathbf{y}$) given the input words ($\mathbf{x}$), is defined as the product of the probability of all the transitions in the building process of all the supertags in the sentence.

$$p(\mathbf{y}|\mathbf{x}) = \Pi_\tau p(a_\tau)$$

where $p(a_\tau)$ is computed from the *softmax* defined in the neural architecture. Note that τ is the index for the transitions, not the words in the sentence. The number of the transitions is usually larger than the number of the word in the sentence. The parameters in the model are learned to maximize the likelihood of a corpus of supertagged sequences.

Because we explicitly model the compositionality inside the supertags, the model we proposed has the ability to deal with an unbounded number of supertags, while the traditional approaches which treat the supertags as discrete categories simply fail.

Extensions Possible extensions for the tagging model can capture some potentially useful information in CCG supertagging.

First, due to the fact that the number of transitions taken to build one supertag can be very large. In practice, we may need to "group" common multiple transition series as a single transition. This is equivalent as treating some *complex* categories as "atoms" in the system.

Second, it is known that modifiers are more likely to appear in the CCG supertag sequences (Garrette et al., 2015). For example, (S\NP)/(S/NP) is more likely than (S\NP)/(NP\NP). To model this effect, one possible solution is to add a new transition COPY which basically copies the completed category on the top of the stack and push it onto the stack. This creates a "short-cut" for the model to remember the previous generated categories since originally this will involve a series of transitions.

Third, tag combinability is an important attribute in CCG. Its implication is that partial supertag structures will often recur during the construction process. To capture this intuition, we plan to introduce a cache of built partial categories (and possible other categories by applying the combination rules). Every time we predict a transition, we pull information from this cache using the attention mechanism (Xu et al., 2015a). This may help the model to build more combinable tags and avoiding invalid supertag sequences.

Another way to capture the tag combinability is to use a variant of the tagging transition system which consider a pair of adjacent tags together. In this variant, we mainly have two additional classes of transitions.

- RULE(R) where $R \in \{>, <, > \mathbf{B}, < \mathbf{B}, < \mathbf{B}_{\times}\}$ indicts which rule in Figure 4.2 is used combine the two adjacent supertags (in practice, the oracle can be generated in a left-to-right greedy manner). Following this action, the system constructs two supertags using the same transitions in the original tagging system but when a combinator or part is pre-determined by the applied rule, it simple uses the corresponding embedding without reconstructing them.
- NEW makes the system to construct a supertag as usual. This rule is used when we build tags that can't be combined with the previous one.

4.2.1 CCG Parsing Transition System

Bottom-up Transition System Zhang and Clark (2011) propose a bottom-up approach for CCG parsing. They introduce two classes of transitions operated between supertags[1]. The algorithm initializes the buffer with the supertag sequence of the sentence.

- COMBINE(X) pops out the top two supertags on the stack, combines them using rule X and pushes it back. X is one of the five operators combining the supertags (Figure 4.2).
- SHIFT pops the current supertag on the buffer and push it on to the stack.

With the existence of the previous tag history stack (E_c) in the tagging model, building a joint-parsing and supertagging model is easy if we use the bottom-up CCG parsing transitions. The model just alternate between parsing and tagging transitions where the SHIFT action in the parsing transitions lets the tagging transition to construct a new supertag and push it to the stack[2].

Top-down Transition System Hockenmaier and Steedman (2002) presents a generative model of CCG derivations, based on various techniques from the statistical parsing literature (Collins, 2003; Charniak, 1997; Goodman, 2000). Inspired by these work, we propose a novel transition

[1] In this section, we do not consider unary rules in the CCG, but extending them into the model is straight-forward.
[2] The parsing process may fail during the decoding time but tagging accuracy is still evaluable in that case.

system that builds the tree in a top-down manner. It can condition on all the previously generated structure with the use of neural models.

- CATEGORY($\mathcal{X}$) introduces an "open CCG category" $\mathcal{X}$ on top of the stack.
- REDUCE closes the open CCG category on the stack. It pops out its derivations from the stack and pushes the category symbol back onto the stack.
- RULE(R) where $R \in \{>, <, > \mathbf{B}, < \mathbf{B}, < \mathbf{B}_{\times}\}$ indicts which rule in Figure 4.2 should be used to break down the category on top of the stack. It picks the $\mathcal{Y}$ category for the rule and then pushes the first open category onto the stack. It has to wait until a "reduce" transition finishes the building of the first category and pushes the second category onto the stack.

With all these transition systems, the generative model for CCG supertagging and parsing can be easily derived. The main difference is every time the model completely generates a supertag, it needs to generate the word associated with it from the tag embedding. Dyer et al. (2016) observe, as hypothesized in Henderson (2004), that larger, unstructured, conditioning contexts are harder to learn from, and provide opportunities to overfit. However, the generative model with access to relatively fewer information may perform better. We plan to use this framework in our work to see whether that improves the final performance.

4.3 Conclusion

We propose to model the compositionality both inside and between the CCG supertags. In theory, our model has the ability to handle an unbounded number of supertags where structurally naive models simply fail. Previous works also suggests that explicit modeling of such information can leads to better accuracy (Srikumar and Manning, 2014) and less supervision (Garrette et al., 2015).

Chapter 5

Conclusion

The work presented in this thesis describes several instances of how assumptions about structure can be integrated naturally into neural representation learners for NLP problems, without sacrificing computational efficiency.

We stress the importance of explicitly representing structure in neural models. We argue that, when comparing with structurally naive models, models that reason about the internal linguistic structure of the data demonstrate better generalization performance.

The techniques proposed in thesis automatically learn the structurally informed representations of the inputs. These representations and components in the models can be better integrated with other end-to-end deep learning systems within and beyond NLP (Cho et al., 2014; Graves et al., 2013, *inter alia*).

We propose to shed light on the interpretability of neural models by further investigation of how the model understands more complex things from composing the small parts and taking advantage of the composition functions governed by linguistically sound assumptions on structures.

Chapter 6

Timeline

The timeline for this thesis will be organized around paper submission deadlines, and the overall goal is to complete this thesis within a year (defending in October 2017).

- Winter 2016: Work on extending the model in §2, Target: ACL 2017.
- Spring 2017: Work on modeling the compositionality in CCG supertagging in §4. Target: EMNLP 2017.
- Summer 2017: Write dissertation.
- October 2017: Thesis defense.

Bibliography

Ossama Abdel-Hamid, Li Deng, Dong Yu, and Hui Jiang. Deep segmental neural networks for speech recognition. In *Proc. INTERSPEECH*, pages 1849–1853, 2013. **??**

Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. Globally normalized transition-based neural networks. *Proc. ACL*, 2016. 1

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *Proc. ICLR*, 2015. 1

Daniel M Bikel. *On the parameter space of generative lexicalized statistical parsing models*. PhD thesis, University of Pennsylvania, 2004. 2.2

Xavier Carreras, Michael Collins, and Terry Koo. Tag, dynamic programming, and the perceptron for efficient, feature-rich parsing. In *Proceedings of the Twelfth Conference on Computational Natural Language Learning*, pages 9–16. Association for Computational Linguistics, 2008. 2.2

Eugene Charniak. Statistical parsing with a context-free grammar and word statistics. *AAAI/IAAI*, 2005(598-603):18, 1997. 4.2.1

Eugene Charniak. A maximum-entropy-inspired parser. In *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference*, pages 132–139. Association for Computational Linguistics, 2000. 2.2

Eugene Charniak and Mark Johnson. Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 173–180. Association for Computational Linguistics, 2005. 2, 2.2

Danqi Chen and Christopher D Manning. A fast and accurate dependency parser using neural networks. In *Proc. EMNLP*, 2014. 1

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *Pro. EMNLP*, 2014. 5

Jan K Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. Attention-based models for speech recognition. In *Advances in Neural Information Processing Systems*, pages 577–585, 2015. **??**, **??**

Michael Collins. Head-driven statistical models for natural language parsing. *Computational linguistics*, 29(4):589–637, 2003. 2.3, 4.2.1

Michael Collins and Terry Koo. Discriminative reranking for natural language parsing. *Computational Linguistics*, 31(1):25–70, 2005. 2.3

Michael Collins, Lance Ramshaw, Jan Hajič, and Christoph Tillmann. A statistical parser for czech. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, pages 505–512. Association for Computational Linguistics, 1999. 2.2

Marie-Catherine De Marneffe and Christopher D Manning. The stanford typed dependencies representation. In *Coling 2008: Proceedings of the workshop on Cross-Framework and Cross-Domain Parser Evaluation*, pages 1–8. Association for Computational Linguistics, 2008. 2.5

Yuan Ding and Martha Palmer. Machine translation using probabilistic synchronous dependency insertion grammars. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 541–548. Association for Computational Linguistics, 2005. 2.3

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011. 2.2.2

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A Smith. Transition-based dependency parsing with stack long short-term memory. In *Proc. ACL*, 2015. 1, 4.2

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A Smith. Recurrent neural network grammars. *Proc. NAACL*, 2016. 1, 4.2.1

Jason Eisner and Giorgio Satta. Efficient parsing for bilexical context-free grammars and head automaton grammars. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, pages 457–464. Association for Computational Linguistics, 1999. 2.1.1

Dan Garrette, Chris Dyer, Jason Baldridge, and Noah A Smith. Weakly-supervised grammar-informed bayesian ccg parser learning. In *AAAI*, pages 2246–2252, 2015. 1, 4, 4.2, 4.2, 4.3

Joshua Goodman. Probabilistic feature grammars. In *Advances in Probabilistic and Other Parsing Technologies*, pages 63–84. Springer, 2000. 4.2.1

Alex Graves. Hierarchical subsampling networks. In *Supervised Sequence Labelling with Recurrent Neural Networks*, pages 109–131. Springer, 2012. 3.2, 3.4

Alex Graves and Navdeep Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *Proc. ICML*, 2014. 3.1

Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, 18(5):602–610, 2005. 3.1

Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proc. ICML*, 2006a. 1

Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *Proc. ICML*, 2006b. 1, 2, 3.5.1

Alex Graves, A-R Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Proc. ICASSP*, pages 6645–6649. IEEE, 2013. 1, **??**, **??**, 5

David Hall, Greg Durrett, and Dan Klein. Less grammar, more features. In *ACL*, 2014. 2.2.1

Yanzhang He and Eric Fosler-Lussier. Efficient segmental conditional random fields for phone recognition. In *Proc. INTERSPEECH*, pages 1898–1901, 2012. **??**

James Henderson. Discriminative training of a neural network statistical parser. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, page 95. Association for Computational Linguistics, 2004. 4.2.1

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997. 3.1

Julia Hockenmaier and Mark Steedman. Generative models for statistical parsing with combinatory categorial grammar. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 335–342. Association for Computational Linguistics, 2002. 4.2.1

Julia Hockenmaier and Mark Steedman. Ccgbank: a corpus of ccg derivations and dependency structures extracted from the penn treebank. *Computational Linguistics*, 33(3):355–396, 2007. 4.1

Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proc. CVPR*, 2015. 3.1

Robert H. Kassel. *A comparison of approaches to on-line handwritten character recognition*. PhD thesis, Massachusetts Institute of Technology, 1995. 3.5.1

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 3.5.1

Dan Klein and Christopher D Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, pages 423–430. Association for Computational Linguistics, 2003. 2.1.3, 2.2

Lingpeng Kong, Chris Dyer, and Noah A Smith. Segmental recurrent neural networks. *Proc. ICLR*, 2015a. 1, 3

Lingpeng Kong, Alexander M Rush, and Noah A Smith. Transforming dependencies into phrase structures. In *Proc. NAACL*, 2015b. 1, 2

Kenton Lee, Mike Lewis, and Luke Zettlemoyer. Global neural ccg parsing with optimality guarantees. *arXiv preprint arXiv:1607.01432*, 2016. 1, 4

Mike Lewis and Mark Steedman. A* ccg parsing with a supertag-factored model. In *EMNLP*, pages 990–1000, 2014a. 1, 4

Mike Lewis and Mark Steedman. Improved ccg parsing with semi-supervised supertagging. *Transactions of the Association for Computational Linguistics*, 2:327–338, 2014b. 1, 4

Liang Lu, Lingpeng Kong, Chris Dyer, Noah A Smith, and Steve Renals. Segmental recurrent neural networks for end-to-end speech recognition. *Proc. Interspeech*, 2016. 1, 3

Christopher D Manning. Computational linguistics and deep learning. *Computational Linguis-*

tics, 2016. 1

Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993. 1, 2.3, 4

André FT Martins, Miguel Almeida, and Noah A Smith. Turning on the turbo: Fast third-order non-projective turbo parsers. In *ACL (2)*, pages 617–622, 2013. 2.3, 2.4

Ryan McDonald. *Discriminative learning and spanning tree algorithms for dependency parsing*. PhD thesis, University of Pennsylvania, 2006. 2.2.1

Tom M Mitchell. *The need for biases in learning generalizations*. Department of Computer Science, Laboratory for Computer Science Research, Rutgers Univ. New Jersey, 1980. 1

Joakim Nivre. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34(4):513–553, 2008. 4.2

Joakim Nivre, Johan Hall, and Jens Nilsson. Maltparser: A data-driven parser-generator for dependency parsing. In *Proceedings of LREC*, volume 6, pages 2216–2219, 2006. 2.4

Slav Petrov and Dan Klein. Improved inference for unlexicalized parsing. In *HLT-NAACL*, pages 404–411. Citeseer, 2007. 2, 2.2

Lawrence R. Rabiner. A tutorion on hidden Markov models and selected applications in speech recognition. *Proc. IEEE*, 77(2), 1989. 2

Sunita Sarawagi and William W. Cohen. Semi-Markov conditional random fields for information extraction. In *Proc. NIPS*, 2004. 3.3

Richard Socher, John Bauer, Christopher D Manning, and Andrew Y Ng. Parsing with compositional vector grammars. In *Proc. ACL*, pages 455–465, 2013. 2.2

Vivek Srikumar and Christopher D Manning. Learning distributed representations for structured output prediction. In *Advances in Neural Information Processing Systems*, pages 3266–3274, 2014. 1, 4, 4.3

Mark Steedman. *The syntactic process*, volume 24. MIT Press, 2000. 1

Hao Tang, Weiran Wang, Kevin Gimpel, and Karen Livescu. Discriminative segmental cascades for feature-rich phone recognition. In *Proc. ASRU*, 2015. **??**, 3.5.2

Ben Taskar, Carlos Guestrin, and Daphne Koller. Max-margin Markov networks. In *Advances in Neural Information Processing Systems 16*, 2004a. 2.2.2

Ben Taskar, Carlos Guestrin, and Daphne Koller. Max-margin Markov networks. *NIPS*, 16:25, 2004b. 3.5.1

Fei Xia and Martha Palmer. Converting dependency structures to phrase structures. In *Proceedings of the first international conference on Human language technology research*, pages 1–5. Association for Computational Linguistics, 2001. 2

Fei Xia, Owen Rambow, Rajesh Bhatt, Martha Palmer, and Dipti Misra Sharma. Towards a multi-representational treebank. In *The 7th International Workshop on Treebanks and Linguistic Theories. Groningen, Netherlands*, 2009. 2

Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov,

Richard S Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. *arXiv preprint arXiv:1502.03044*, 2015a. 4.2

Wenduan Xu, Michael Auli, and Stephen Clark. Ccg supertagging with a recurrent neural network. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*, volume 2, pages 250–255, 2015b. 1, 4

Naiwen Xue, Fei Xia, Fu-Dong Chiou, and Martha Palmer. The Penn Chinese TreeBank: Phrase structure annotation of a large corpus. *Natural Language Engineering*, 11(2):207–238, 2005. 2.3

Yuan Zhang and David Weiss. Stack-propagation: Improved representation learning for syntax. *arXiv preprint arXiv:1603.06598*, 2016. 2.5

Yue Zhang and Stephen Clark. Shift-reduce ccg parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pages 683–692. Association for Computational Linguistics, 2011. 4.2.1

Muhua Zhu, Yue Zhang, Wenliang Chen, Min Zhang, and Jingbo Zhu. Fast and accurate shift-reduce constituent parsing. In *ACL (1)*, pages 434–443, 2013. 2.3, 2.2, 2.4

Geoffrey Zweig. Classification and recognition with direct segment models. In *Proc. ICASSP*, pages 4161–4164. IEEE, 2012. **??**