Path-Based Neural Constituency Parsing

Katia Patkin

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2019-21
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-21.html

May 1, 2019

Copyright © 2019, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

UNIVERSITY of CALIFORNIA, BERKELEY

Path-Based Neural Constituency Parsing

by

Katia Patkin

A thesis submitted in partial fulfillment for the
degree of Masters of Science

in the
Department of Computer Science

December 2018

Path-Based Neural Constituency Parsing

by Katia Patkin

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor John DeNero
Research Advisor

(Date)

* * * * * * *

Professor Dan Klein
Second Reader

(Date)

Abstract

Path-Based Neural Constituency Parsing

by

Katia Patkin

Master of Science in Computer Science

University of California, Berkeley

Professor John DeNero, Chair

We describe an approach to constituency parsing that first predicts tree paths for each word in a sentence using a neural sequence model, then deterministically combines those paths into a parse using A* search. Our approach contrasts state-of-the-art neural parsers that combine independent predictions for each span and have no auto-regressive component. We show that using sequence prediction for paths, rather than independent classification for spans, leads to higher exact-match parse accuracy under a fixed sentence encoder.

Contents

List of Figures

List of Tables

Abbreviations

POS	**P**art **O**f **S**peech
CCG	**C**ombinatory **C**ategorical **G**rammar
LSTM	**L**ong **S**hort **T**erm **M**emory

Chapter 1

Introduction

Syntactic analysis has long served as a canonical example of a structured prediction task in natural language processing. However, modeling syntactic analysis as a collection of independent classifications was shown to be sufficient for part-of-speech tagging even before the rise in popularity of neural models [1, 2]. Recently, state-of-the-art constituency parsing performance has been achieved by a collection of independent span classification predictions that are combined into a tree with a structured search procedure [3–5]. These results call into question whether structured or auto-regressive modeling—for example with a neural sequence model—is necessary or appropriate for the syntactic analysis of natural language.

This work describes a novel sequence-based model for constituency parsing that provides a substantially higher exact-match parse accuracy than a span-based parser, when using the same sentence encoder. For section 23 of the Penn Treebank, F1 score decreases from 91.77% to 90.36% using our sequence model, but exact match accuracy increases from 37.46% to 40.11%. While 40.11% of sentences have a perfect F1 score, 32.82% of sentences have a F1 score of 90% or less. By contrast, the span-based parser has only 37.46% of sentences with a perfect F1 score, yet only 29.26% of the sentences have a F1 score of 90% or less, which contributes to its higher overall F1 score. Therefore, this model offers a way to trade off span accuracy for tree accuracy. Sequence prediction may be preferred over span classification in applications where correctly analyzing whole sentences is important for downstream performance.

Our approach is inspired by Combinatory Categorical Grammar (CCG), which employs a rich set of *supertags* to describe the role that each word plays in the structure of a sentence [6]. We introduce a method for decomposing a phrase-structure tree into a sequence of paths, one path per word, which describe all constituents for which a word is the syntactic head. These paths share many characteristics with CCG supertags. We train a model to predict a path for each word. This model can be viewed as a *seq2seq* model with attention [7]. Inspired by neural CCG parsing [8], we use A* search to find a sentence parse that combines the predicted paths for each word.

Our path-based model contrasts with other auto-regressive approaches to constituency parsing that sequentially generate the entire parse tree [9] or the entire parsed sentence [10]. Instead, our model makes independent sequential predictions for each word in the sentence, then combines these predictions via structured search. This degree of auto-regressive modeling offers an intermediate option between fully auto-regressive sequence-based or stack-based models and fully independent span models. Experiments show that our path-based parser not only yields higher exact-match parse accuracy, but also provides k-best lists that have dramatically higher oracle accuracy; 20-best oracle exact match improves from 51.78% for a span-based parser to 67.05% for our path-based parser.

Chapter 2

Representing Trees as Paths

Rather than predicting a parse tree directly, we propose to partition the tree into non-overlapping paths, one path per word, such that the path for each word describes all constituents for which that word is the syntactic head. We construct these paths so that each piece of information about the tree appears only once, but all information about the tree is captured in the paths.

We introduce a sequential representation called an *augmented path* that describes a path of nodes starting at a word, as well as gaps along the path indicating where paths from other words combine as siblings.

2.1 Augmented Paths

An augmented path consists of a sequence of *augmented labels* that include syntactic categories of constituents (e.g., NP), directional gap markers '/' and '\', and directional combination markers '<', '>'. The sequence of syntactic categories within an augmented path describes a partial path through the tree, starting at the corresponding word.

Gap markers are interleaved with the syntactic categories and describe the valence of each constituent along the path. The backward slash '\' and forward slash '/' indicate left and right gaps, respectively. A gap marker in an augmented path is similar to a functor in a CCG supertag, but does not encode the syntactic category of its arguments. Each gap corresponds to a sibling of the current node in the augmented path.

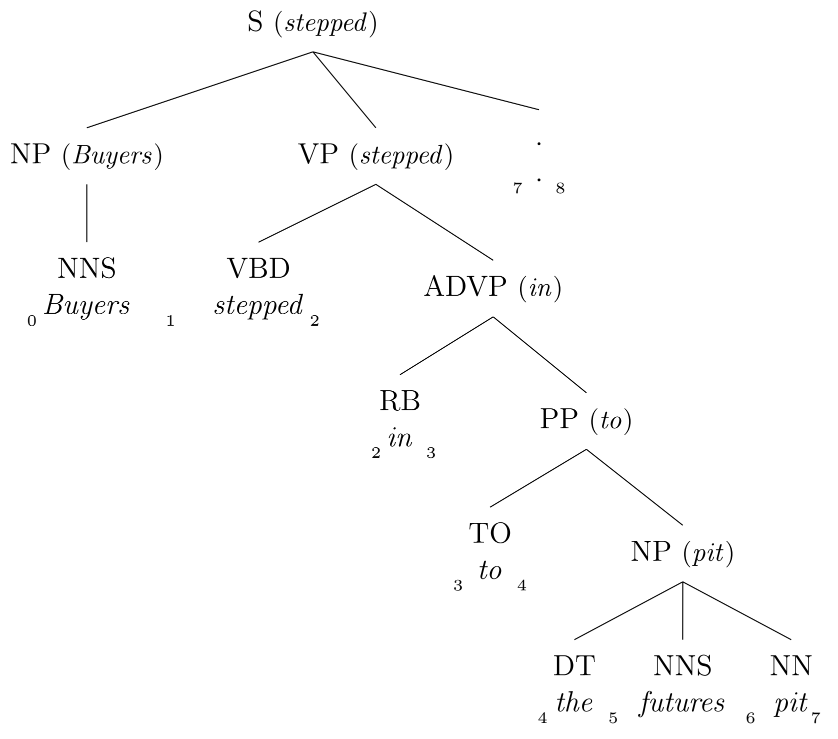


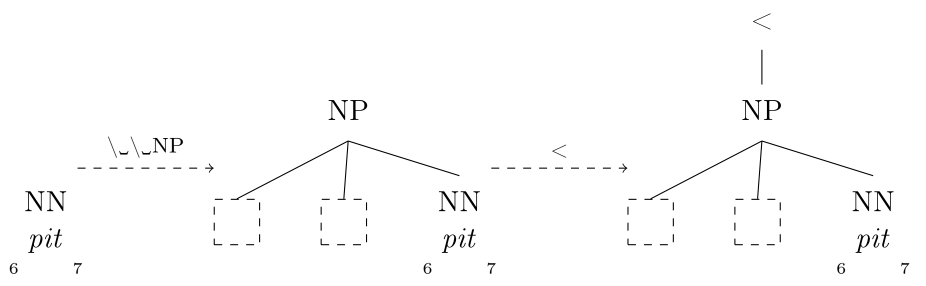
Figure 2.1: Constituency parse tree annotated by the syntactic heads for "*Buyers stepped in to the futures pit .*"

word	augmented path (syntactic head)
Buyers	`NNS␣NP␣>`
stepped	`VBD␣/␣VP␣\␣/␣S`
in	`RB␣/␣ADVP␣<`
to	`TO␣/␣PP␣<`
the	`DT␣>`
futures	`NNS␣>`
pit	`NN␣\␣\␣NP␣<`
.	`.␣<`

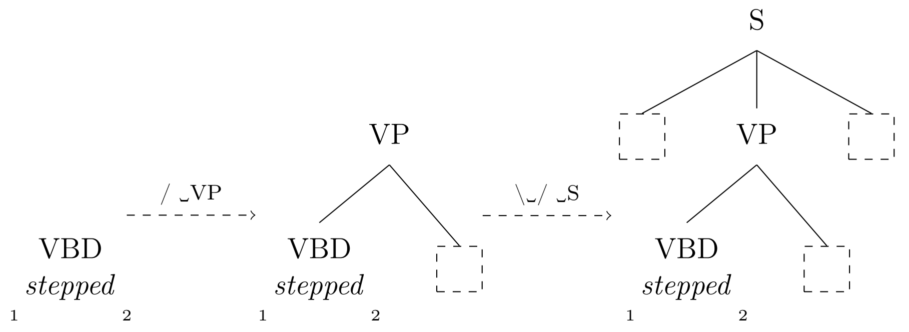
Table 2.1: Augmented paths for the sentence "Buyers stepped in to the futures pit ."

Combination markers only appear at the end of an augmented path. The angle brackets '<', '>' indicate the combination direction of the partial tree that is represented by the augmented path. The left angle bracket indicates combination into a sub-tree on the left and the right angle bracket indicates combination into a sub-tree on the right of the current augmented path. Only one combination marker can appear in an augmented path, and always at the end. When an augmented path terminates at the root of the whole tree, no combination marker is added because this augmented path is a partial representation of the entire tree and cannot combine into another sub-tree.

We chose head-based branching instead of other options, such as right-branching, because



(a) Construction of a partial tree for (NN *pit*) of the augmented path \␣\␣NP ␣< .



(b) Construction of a partial tree for (VBD *stepped*) of the augmented path: / ␣VP␣\␣/ ␣S .

Figure 2.2: Examples of converting augmented paths into a partial-tree representations. The empty dashed nodes indicate the "gap nodes". The dashed arrows indicate the next state of constructing the partial trees.

this led to more balanced path lengths. A right-branching partition would include only left gaps, and the path for each word would continue as long as it was the right-most word in its constituent. Instead, we propose to include in each augmented path only the constituents for which the source word is the syntactic head.

Figure 2.1 shows the syntactic head for each constituent in an example parse tree. The augmented paths for all the words in the sentence are summarized in Table 2.1.

2.2 Computing Augmented Paths

A tree can be partitioned into augmented paths by identifying the maximal sub-tree for each word, which is the sub-tree that contains all constituents that are headed by that word. The augmented path is a partial traversal of the maximal sub-tree of the word, starting from the leaf, i.e. the word, and traversing to the root of the maximal sub-tree without following branches. All the syntactic categories along the path from the word to the root are recorded into a sequence that also includes gap markers for the siblings that are not traversed. A combination marker is added at the end if the maximal sub-tree

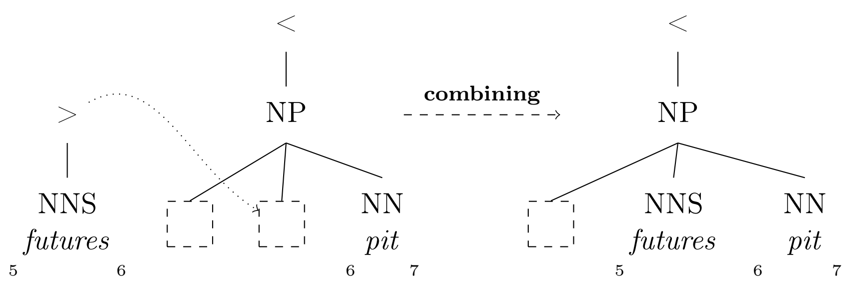


Figure 2.3: A legal combination of partial trees. Before combining: T_6, partial tree that spans over $(5, 6)$, and T_7, partial tree that spans over $(6, 7)$. After combining: $T_{5,7}$, partial tree that spans over $(5, 7)$.

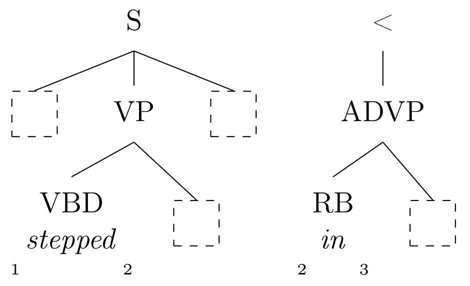
for a word is not the whole tree, but instead combines into another word's maximal sub-tree. Procedurally, we use the LTH Constituent-to-Dependency Conversion Tool [11] to identify the syntactic head of each constituent.

For example, we construct the augmented path for the word "*in*" in the sentence "*Buyers stepped in to the futures pit .*" as follows. Starting from the leaf and going to the root of the maximal sub-tree rooted at the `ADVP`, recording all the constituency types along the path as well as all the sibling gaps, we find `RB`, then a right gap '`/`', then `ADVP`. Finally '`<`' is added to the sequence to indicate that this sub-tree will combine into a partial tree on the left. Thus, the augmented path of "*in*" is `RB␣/␣ADVP␣<`. In this example, the PP constituent fills the gap on the right of RB, and ADVP is its parent, as well as the root of the sub-tree.

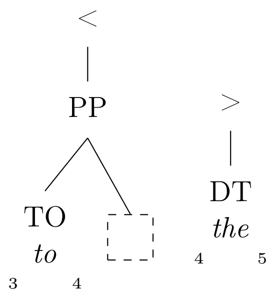
We construct augmented paths to include part-of-speech (POS) tags. However, when training the model to predict augmented paths, we treat POS tags as part of the input.

2.3 From Augmented Paths to Trees

A tree is reconstructed from a sequence of augmented paths by first converting all the augmented paths into partial trees. A partial tree is initialized as a single node for the pair of a word and its POS tag. Sibling and parent nodes are added to the partial tree by processing the augmented path from left to right: Gap markers are treated as sibling nodes of the current node, with the appropriate directionality. Once a constituency label is encountered, it is added as a parent node and becomes the new current node. This



(a) Partial trees T_2 and T_3 cannot combine because both have gaps.



(b) Partial trees T_4 and T_5 cannot combine because T_5 points to the right and T_4 is to the left of T_5.

Figure 2.4: Partial trees that cannot combine.

process is repeated until the entire augmented path is exhausted. If the last augmented label is a combination marker, it will become the root of the partial tree (cf. Figure 2.2).

Once all the augmented paths are converted into partial trees, the partial trees can be combined into a single complete tree. Partial trees are always combined in pairs. The tree that merges into another tree is called the *source partial tree* and the tree that is being merged into is called the *destination partial tree*.

A partial tree that spans all the words from i to j is denoted by $T_{i,j}$, and a partial tree that spans the i-th word in the sentence is denoted by T_i. The combined tree of any two partial trees, $T_{i,j}$ and $T_{j,k}$, is $T_{i,k}$ with $i < j < k$. From the construction of a partial tree, we observe that all of its leaves are either gap nodes or word-tag pairs.

Combining partial trees is possible only if the following conditions hold: (1) The partial trees are adjacent, i.e. the partial trees $T_{i,j}$ and $T_{j,k}$. (2) The source partial tree does not have gap nodes. (3) The root of the source partial tree is '>' ('<') if it combines to the right (left), and the destination partial tree has a gap node to the left (right) of the first (last) word in its span.

Once all the above conditions hold, the combination is done by removing the root combination marker and filling the appropriate gap in the destination tree with the source partial tree. For any destination tree, there is only one gap that can be filled first: the lowest and inner-most gap. Figure 2.3 illustrates the combination process. Figure 2.4 shows partial trees that cannot combine.

Given that all partial trees for the sentence are able to combine, the combination process generates a unique tree. Therefore, predicting the correct augmented path for each word is sufficient to reconstruct the entire correct parse tree.

Chapter 3

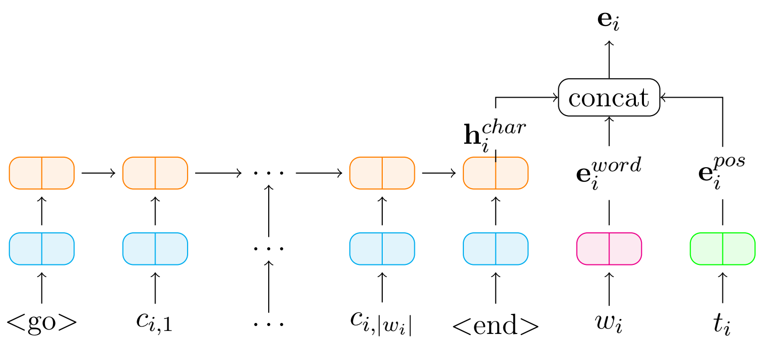
Predicting Augmented Paths

The neural model for independently predicting the augmented path for each word in a sentence is conditioned on the contents of the whole sentence. The architecture follows the encoder-decoder schema. The encoder is used to build context-sensitive vector encodings of the sentence, one focused on each word-tag pair. The decoder uses those encodings to generate conditional probability distributions over the augmented labels of the augmented path for each word.

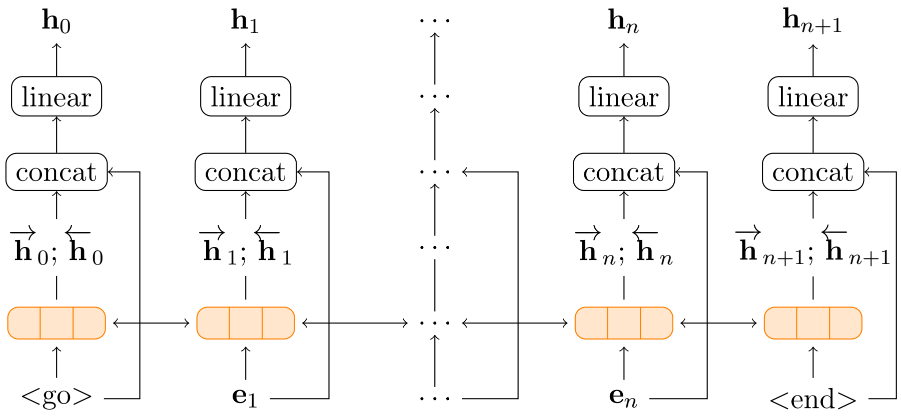
The input to our model is a sequence of word-tag pairs. For a sentence of length n, the output is n different augmented paths, which are each predicted as variable length sequences of augmented labels. Each augmented path is predicted independently of the rest, conditioned on the input sentence. Thus, for each word we have a sequence-to-sequence problem; the input sequence is the whole sentence, and the output sequence is the whole augmented path. Our encoder-decoder architecture follows Sutskever et al. [12], who introduced a neural network model for solving sequence-to-sequence problems, and by Bahdanau et al. [7], who proposed a related model with an attention mechanism that improves generation of long sequences.

3.1 Encoder

The encoder is a bi-directional Long Short-Term Memory (LSTM) [13, 14]. The inputs to the bi-directional LSTM are the concatenation of: (1) character-level word representations, (2) word-embeddings, and (3) POS tag embeddings. The character-level word



(a) Embedding for the i-th word-tag pair.



$\mathbf{h}_0$
$\mathbf{h}_1$
$\mathbf{h}_n$
$\mathbf{h}_{n+1}$
linear
concat
$\overrightarrow{\mathbf{h}}_0; \overleftarrow{\mathbf{h}}_0$
$\overrightarrow{\mathbf{h}}_1; \overleftarrow{\mathbf{h}}_1$
$\overrightarrow{\mathbf{h}}_n; \overleftarrow{\mathbf{h}}_n$
$\overrightarrow{\mathbf{h}}_{n+1}; \overleftarrow{\mathbf{h}}_{n+1}$
<go>
$\mathbf{e}_1$
$\mathbf{e}_n$
<end>

(b) Representation of the words in a sentence with n words.

Figure 3.1: The encoder architecture used in experiments.

representation is the final state of a separate LSTM over the character embeddings for a word. Figure 3.1a diagrams the input vector for the i-th word, $\vec{e}_i$. The sequence of all input vectors, $[\vec{e}_i]_{i=1}^n$, is passed through a two-layer bi-directional LSTM to obtain context-sensitive forward and backward encodings, $\overrightarrow{h}_i$ and $\overleftarrow{h}_i$.

Our representation of each word in the sentence is the concatenation of the forward and backward encodings. On top of this, per-position encodings are concatenated with the input vector and then passed through a fully-connected layer to reach a final sentence-level representation, $\vec{h} = [\vec{h}_i]_{i=1}^n$ (see Figure 3.1b):

$$\vec{h}_i = \vec{W} \cdot [\overrightarrow{h}_i; \overleftarrow{h}_i; \vec{e}_i] + \vec{b} \tag{3.1}$$

Recent work has used similar encoders. For example, Stern et al. [3] used a two-layer bi-directional LSTM over the inputs of words and POS tag embeddings for constituency parsing. Later, Gaddy et al. [4] showed that a bi-directional character LSTM of words is sufficient for constituency parsing. Kitaev and Klein [5] used a self-attentive architecture for the encoder, instead of an RNN, to achieve state of the art results. These different encoder architectures are all compatible with our approach.

3.2 Decoder

The decoder consist of a unidirectional LSTM with attention [7], followed by a feed-forward layer and softmax, as illustrated in Figure 3.2a. For each word in the sentence, the initial state of the LSTM is the word encoding $\vec{h}_i$. The decoder is trained to predict the augmented-path L_i for each word. At each time-step the input to the decoder is an augmented label $\ell_{i,j}$, the output from the decoder is the conditional probabilities of each possible next augmented label ℓ, given the word representation and the previous augmented-labels:

$$\vec{p}^{\,\ell}_{i,j+1} = Pr(\ell | \{\ell_{i,1}, \ldots, \ell_{i,j}\}, \vec{h}_i) \tag{3.2}$$

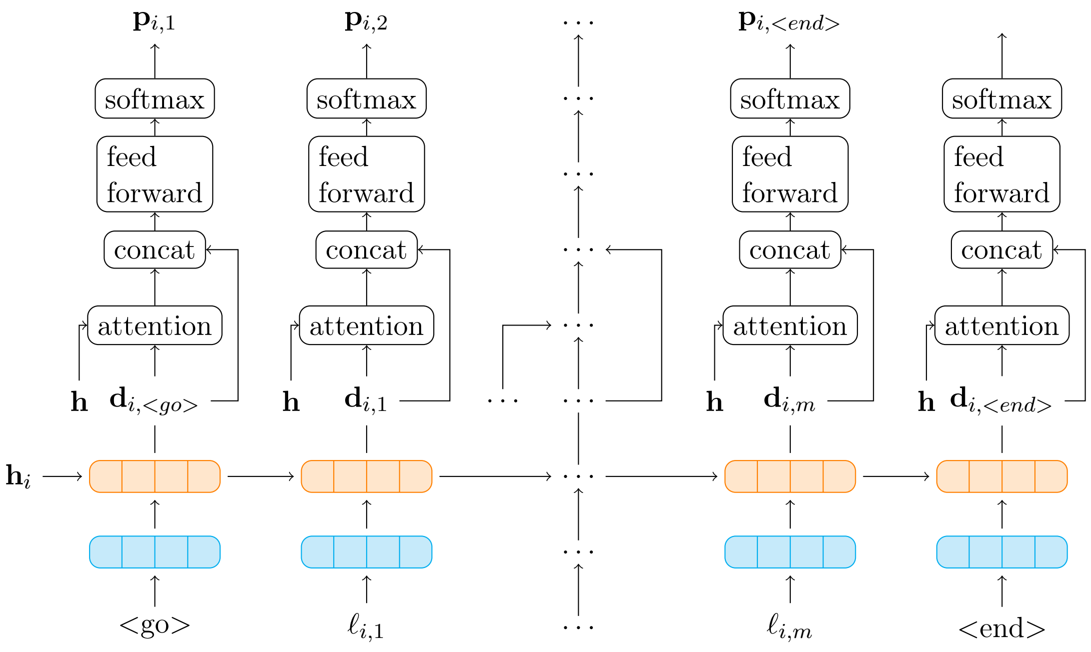
Notice that,

$$\begin{aligned} \vec{p}_{i,j+1} &= \text{softmax}(\vec{V} \cdot \vec{r}_{i,j}) \\ \vec{r}_{i,j} &= g(\vec{W}_r \cdot [\vec{c}_{i,j}; \vec{d}_{i,j}] + \vec{b}_r) \end{aligned} \tag{3.3}$$

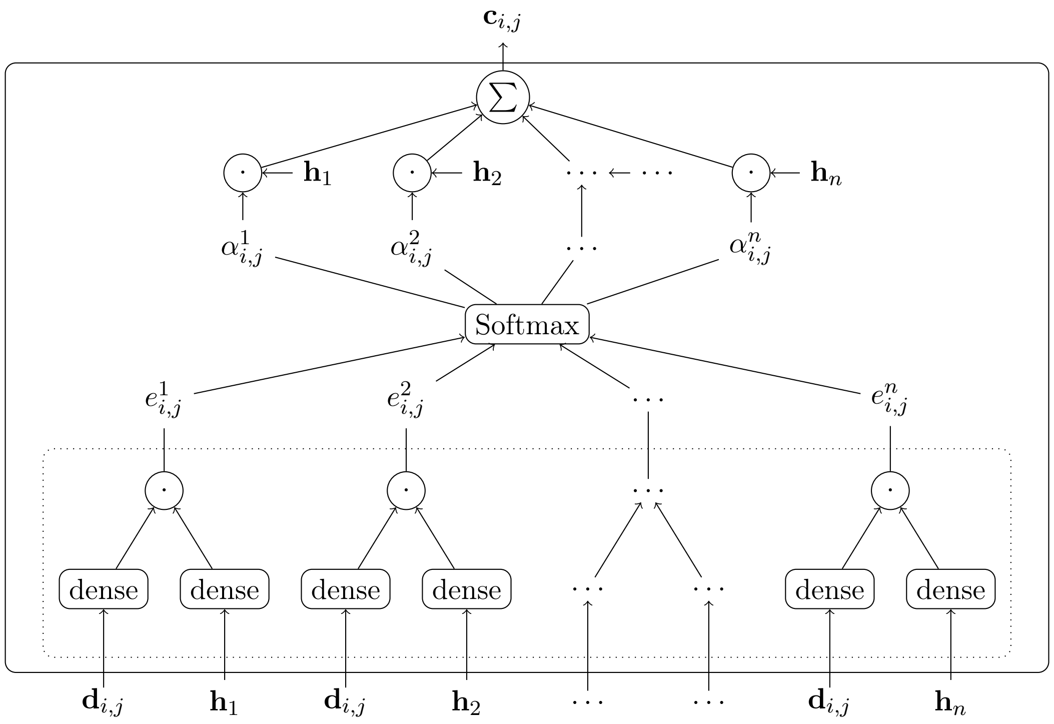
is a function of the output from the LSTM, $\vec{d}_{i,j}$ and the context output of the attention, $\vec{c}_{i,j}$. The context is a linear combination over the encoded sentence: $\vec{c}_{i,j} = \sum_{t=1}^{n} \alpha_{i,j}^{t} \cdot \vec{h}_t$. The attention weights are computed by a dot-product between the projection of $\vec{d}_{i,j}$ and each column of the projection of $\vec{h}$:

$$\begin{aligned} \vec{\alpha}_{i,j} &= \text{softmax}(\tilde{\vec{d}}_{i,j} \cdot \tilde{\vec{h}}) \\ \tilde{\vec{d}}_{i,j} &= g(\mathbf{W}_d \cdot \vec{d}_{i,j} + \vec{b}_d) \\ \tilde{\vec{h}} &= g(\mathbf{W}_e \cdot \vec{h} + \vec{b}_e) \end{aligned} \tag{3.4}$$

and $g(\cdot)$ is the ReLU non-linearity function.



(a) The decoder for word-tag pair i.



(b) The attention mechanism used in the decoder, for augmented label j for a word-tag pair i. The dashed box shows the sub-network that computes attention weights.

Figure 3.2: Graphical depiction of the decoder

Chapter 4

Searching for Parse Trees

Given a trained model applied to a sentence, parse trees are generated by a combination of the beam-search and A*-search algorithms. Beam search is used to generate a list of candidate augmented paths for each word. A* search is used to find complete parse trees constructed from these augmented paths.

4.1 Path Generation with Beam Search

The goal of beam-search is to find augmented paths that approximately maximize the joint conditional probability of the augmented labels for a word under the model.

For a beam search algorithm with a beam size of k, the algorithm maintains k partial hypotheses for each length, where a partial hypothesis is a prefix of some augmented path. For each word, the augmented paths are built starting with the '`<go>`' token, from left-to-right. At each step, the partial hypothesis in the beam is expanded with every possible next augmented label in the vocabulary and scored by the joint conditional probability of all augmented labels of the hypothesis. The top-k sequences with the highest probabilities according to the model (c.f. section 3.2) are selected, pruning all other candidates. Each time an '`<end>`' token is appended to a hypothesis, that hypothesis is removed from the beam, and the augmented path it describes is added to the set of complete hypotheses. The process is repeated until all surviving hypotheses are complete sequences and so the set of active partial hypotheses is empty. The top-k augmented paths for the i-th word are denoted by $\hat{L}_i^{(1)}, \dots, \hat{L}_i^{(k)}$. Each is a sequence of augmented labels and has the following total score:

$$s_i^{(r)} = \sum_{j=1}^{m} \log \left(Pr(\hat{\ell}_i^{(r_j)} | \{\hat{\ell}_i^{(r_1)}, \ldots, \hat{\ell}_i^{(r_{j-1})}\}, \vec{h}_i) \right)$$
$$\hat{L}_i^{(r)} = [\hat{\ell}_i^{(r_1)}, \ldots, \hat{\ell}_i^{(r_m)}], \quad r \in \{1, k\} \tag{4.1}$$

For each word i in the sentence, all the augmented paths are converted into partial trees as described in section 2.3, resulting in scored partial trees: $[(T_i^{(r)}, s_i^{(r)})]_{r=1}^{k}$, where i is the span and r is the rank, in descending order, with the highest scoring partial tree at rank one. We call these *basic partial trees* because they span only one word and have a score that is assigned by the neural model.

4.2 Tree Construction with A* Search

A* search is used to find a set of basic partial trees that combine into the highest scoring complete parse tree. The score of a tree is the sum of the scores of the basic partial trees used to construct it. Details on combining partial trees are found in section 2.3. A* maintains two main data structures: (1) A closed set, which records all the partial trees that have already been processed, and (2) an open set, a priority queue of partial trees waiting to be processed. Initially, the open set contains only basic partial trees and the closed set is empty. The main loop in A* involves removing a partial tree from the open set and combining it with all partial trees in the closed set, then adding it to the closed set. Successfully created partial trees are added to the open set.

4.2.0.1 State Space

Search states are partial trees. Each covers a span of the sentence and is built from basic partial trees for the words in that span. The score of the state is the sum of the scores for these basic partial trees. A* also requires a heuristic upper bound on the score of the completion for each state. In a sentence with n words, for a partial tree that spans (i, j), with basic partial trees that construct it, $T_i^{(r_i)}, \ldots, T_j^{(r_j)}$, the total score including the heuristic is:

$$C(r') = \underbrace{\sum_{t=i}^{j} s_t^{(r_t)}}_{\text{score}} + \underbrace{\sum_{t=1}^{i-1} s_t^{(1)} + \sum_{t=j+1}^{n} s_t^{(1)}}_{\text{heuristic}} \tag{4.2}$$

This heuristic optimistically assumes that a partial tree will successfully combine with all the highest-scoring basic partial trees outside of its span.

The open set is initialized with the highest scoring basic partial tree for each word from beam search. That is, for a sentence with n words, the start states are $\left[T_i^{(1)}\right]_{i=1}^{n}$.

The search concludes when a complete tree is found: a tree that spans the entire sentence and does not have any gaps. To find the k-best complete trees, we simply continue the search until k complete trees are found.

4.2.0.2 Successor Function

Partial trees that are removed from the open set are combined with trees from the closed set if the combination is legal (c.f. section 2.3). These new partial trees are added to the priority queue.

If the tree removed from the open set is a basic partial tree, then a basic partial tree with a higher rank is also a successor. That is, if $T_i^{(r)}$ was moved from the open set, then $T_i^{(r+1)}$ is added to the open set.

4.2.1 Search Time Limit

A* can take a long time to complete if there are high-scoring partial trees that cannot combine. We allow for a user-defined time-out to avoid a long run-time. If the time-out is reached or the open set is exhausted without finding a complete tree, then we choose the most complete tree discovered during search, breaking ties by model score, and generate a full parse by removing gaps and adding missing words as children of the root if necessary.

Chapter 5

Related Work

Lewis et al. [8] describe a similar approach to CCG parsing that combines conditionally independent predictions using A* PCFG parsing [15]. Lee et al. [16] showed that global features, a form of structure modeling, improved performance.

Vinyals et al. [9], and later Choe and Charniak [17], Liu et al. [18], showed that an encoder-decoder framework employing LSTM RNNs and an attention mechanism could be used to train a neural constituency parser that approached state-of-the-art performance while making few assumptions about the nature of the task. Several other auto-regressive neural models improved on this initial work, including shift-reduce parsers that conditioned on various parts of the output structure [10, 19, 20].

Later work questioned whether auto-regressive models were necessary. For example, [21] instead casts constituency parsing as a sequence labeling problem.

However, Stern et al. [3] showed that an auto-regressive neural model was not necessary to achieve state-of-the-art parser F1. Instead, this work proposed a model in which the predicted distribution over span labels were conditionally independent of each other, given the input sentence. Gaddy et al. [4] showed that representational changes could improve performance further, and Kitaev and Klein [5] demonstrated further improvements using a Transformer encoder and pre-trained embeddings. Hong [22] describes an alternative search procedure. This line of work has established span-based decoders that combine conditionally independent predictions using structured search as the current state-of-the-art approach to constituency parsing.

Chapter 6

Experiments

We perform a controlled comparison of our path-based decoder to the span-based decoder described in [4], which is a subtle improvement to the decoder in Stern et al. [3] and also used in [5]. This approach to decoding represents the state of the art in constituency parsing. We do not compare different encoders, but instead focus on comparing decoder performance using the fixed encoder described in section 3.1. We expect recent encoder and pre-training improvements to be orthogonal to this work.

We use the Penn Treebank [23] for our experiments with the standard splits of sections 2-21 for training, section 22 for development, and section 23 for testing.

6.1 Hyper-parameters and Training

We use the *ADAM* optimizer [24] with its default settings for optimization and a mini-batch size of 10 to train each model. We train each model for at least 20 epochs. We evaluate each model on the development set 4 times per epoch and choose the parameters with the best dev performance. The inputs to both parsers are concatenations of 100-dimensional word embeddings, 150-dimensional part-of-speech tag embeddings and 100-dimensional character-level LSTM states. The encoder is a two-layer bi-directional LSTM. The hidden dimension of the LSTM over words is 350. The hidden dimension of our decoder LSTM is 600. Dropout with ratio 0.4 was applied to all non-recurrent connections and inputs for the LSTMs. All parameters are randomly initialized. Hyper-parameters are chosen using the development set. During training, words are replaced

Parser	LR	LP	F1	Exact
Path-based @5	89.88	90.69	90.28	40.03
Path-based @10	89.93	90.72	90.33	40.11
Path-based @20	89.95	90.76	90.35	40.11
Path-based @32	89.96	90.76	90.36	40.11
Span-based	91.47	92.08	91.77	37.46

Table 6.1: Test set scores on Penn Treebank section 23 for different beam sizes of our parser using a two-minute-per-sentence time-out.

by `<UNK>` token with probability $1/(1 + freq(w))$, where $freq(w)$ is the frequency of w in the training data. At test time, we used the `<UNK>` token only for unknown words. Gold POS tag are used as input during training and test for both parsers. Both parsers are implemented with the DyNet library [25].

6.2 Results

We ran our path-based parser with different beam sizes from 5 to 32, the number of augmented label types. Table 6.1 shows that F1 score and exact match accuracy improve by only 0.08% as beam size increases. We conclude that performance is largely independent of beam size and use a beam size of 10 for further experiments. Compared to our retrained version of the span-based parser in [3, 4], performance is lower by 1.44% F1, but exact-match accuracy is higher by 2.65%.

The gap in F1 performance appears to materialize mostly for long sentences, as shown in Figure 6.1. The span-based decoder is better for all lengths, but the parsers are quite similar for shorter sentences. Figure 6.2 compares exact-match scores for different length bins. The path-based parser outperforms the span-based parser for lengths below 50. Interestingly, the greatest advantage appears for mid-length sentences between 20 and 40 words. In Figure 6.3, we plot the percentage of the sentences in the test set that exceed a certain F1 score. Our path-based parser produces more parses with F1 scores higher than 95%, compared to the span-based parser. However, for lower F1 score thresholds, the span-based parser is better, which explains the better overall F1 score.

We extended the chart parsing algorithm over spans to generate the k-best trees for each sentence: For each span, the chart parser finds the label with the maximal score, and finds the split with the maximal score. Our k-best extension finds the top-k labels for

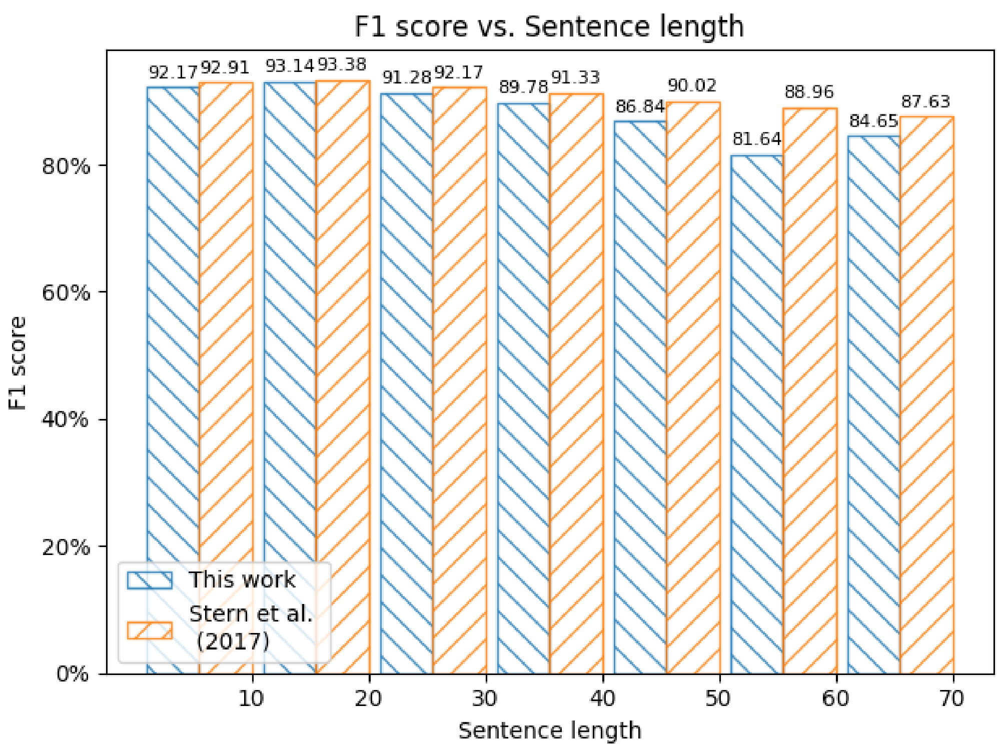


Figure 6.1: F1 score for subsets of the test set binned by sentence length.

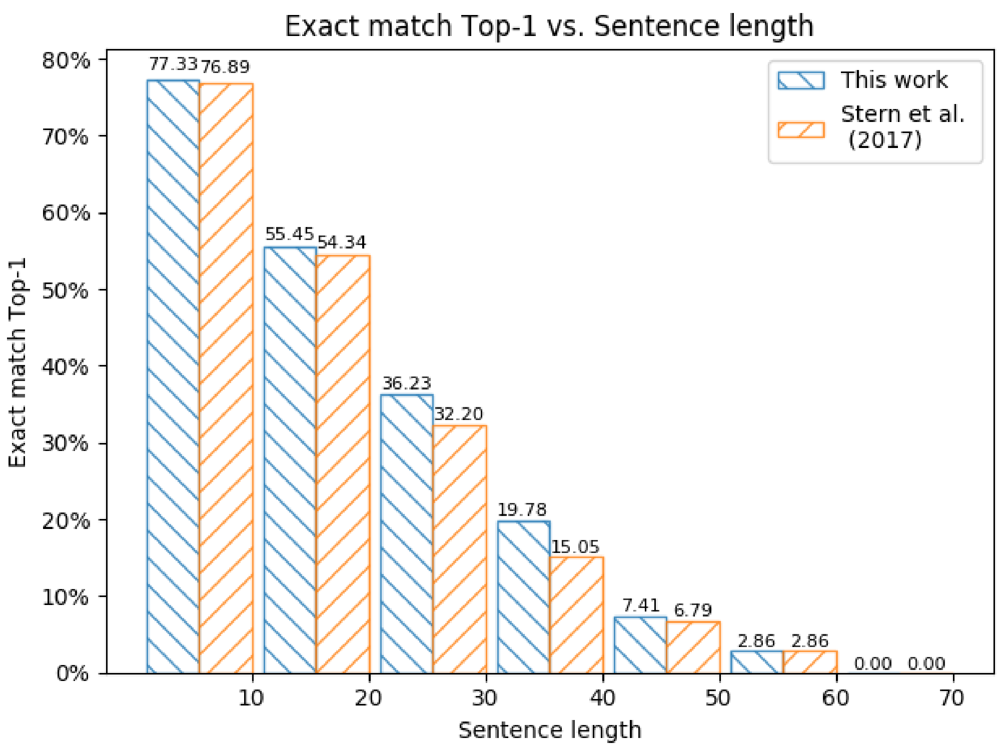


Figure 6.2: Oracle exact-match for subsets of the test set binned by sentence length for top-1 parses.

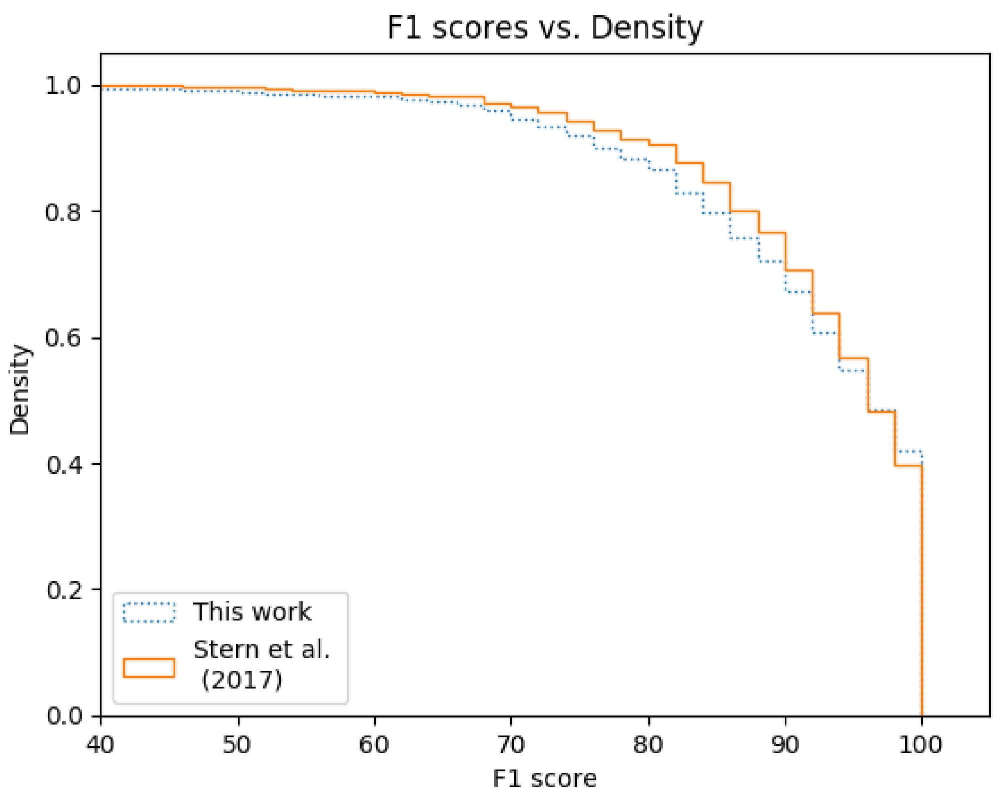


Figure 6.3: Percentage of test set sentences exceeding F1 score.

each span and uses A* search to find the top-k sub-trees for each span. For each parser, we generated the top 20 parses for each sentence and evaluated these top-k lists by oracle accuracy: a k-best list is correct if it contains the gold tree exactly as one of its entries. In Figure 6.4, we can see that our parser improves by 26.94% absolute to 67.05% when evaluating the top 20 parses instead of the top 1, while the span-based parser improves by only 14.32% to 51.78%.

Typical improvements apply to our path-based parser. For example, the length normalization technique in [26] gives a slightly better F1 score (90.43%) and helps with sentences longer than 50 (which increase by ∼2% F1). Exact match also improves to 40.19% for 1-best and 67.09% for 20-best lists. Using pre-trained word embeddings [27], improves F1 to 93.01%.

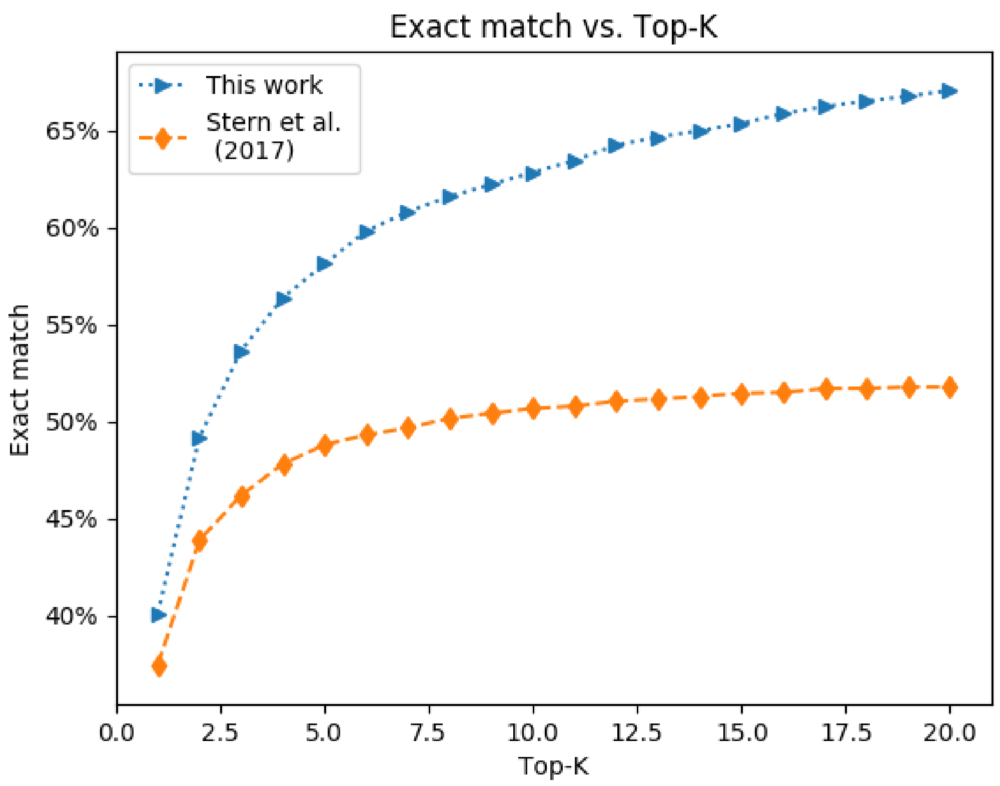


Figure 6.4: Oracle exact-match score for k-best lists.

Chapter 7

Conclusion

In this work, we proposed a novel way to generate phrase-structure trees by modeling paths from each word. Given a constant encoder, this approach improves exact match accuracy over a state-of-the-art span-based approach. Thus, auto-regressive models may have advantages over those that make conditionally independent predictions given the input. However, while more parses are exactly right, more parses have high error, especially for long sentences. We look forward to improving long sentence performance in future work.

Bibliography

[1] Percy Liang, Hal Daumé III, and Dan Klein. Structure Compilation: Trading Structure for Features. In *Proceedings of the 25th international conference on Machine learning*, pages 592–599, 2008. URL http://nlp.cs.berkeley.edu/pubs/Liang-Daume-Klein_2008_Structure_paper.pdf.

[2] Robert Moore. Fast High-Accuracy Part-of-Speech Tagging by Independent Classifiers. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pages 1165–1176, 2014. URL http://www.aclweb.org/anthology/C14-1110.

[3] Mitchell Stern, Jacob Andreas, and Dan Klein. A Minimal Span-Based Neural Constituency Parser. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pages 818–827, 2017. URL http://aclweb.org/anthology/P17-1076.

[4] David Gaddy, Mitchell Stern, and Dan Klein. What's Going On in Neural Constituency Parsers? An Analysis. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 999–1010, 2018. URL http://aclweb.org/anthology/N18-1091.

[5] Nikita Kitaev and Dan Klein. Constituency Parsing with a Self-Attentive Encoder. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, pages 2676–2686, 2018. URL http://aclweb.org/anthology/P18-1249.

[6] Mark Steedman. *The syntactic process*, volume 24. MIT press Cambridge, MA, 2000.

[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. In *International Conference on Learning Representations (ICLR)*, 2015. URL http://arxiv.org/abs/1409.0473.

[8] Mike Lewis, Kenton Lee, and Luke Zettlemoyer. LSTM CCG Parsing. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 221–231, 2016. URL http://www.aclweb.org/anthology/N16-1026.

[9] Oriol Vinyals, Łukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. Grammar as a Foreign Language. In *Advances in Neural Information Processing Systems*, pages 2773–2781, 2015. URL http://papers.nips.cc/paper/5635-grammar-as-a-foreign-language.pdf.

[10] Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A Smith. Recurrent Neural Network Grammars. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 199–209, 2016. URL http://www.aclweb.org/anthology/N16-1024.

[11] Richard Johansson and Pierre Nugues. Extended Constituent-to-dependency Conversion for English. In *Proceedings of the 16th Nordic Conference of Computational Linguistics (NODALIDA 2007)*, pages 105–112, 2007. URL http://aclweb.org/anthology/W07-2416.

[12] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems*, pages 3104–3112, 2014. URL http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf.

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

[14] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to Forget: Continual Prediction with LSTM. *Neural Computation*, 12:2451–2471, 1999.

[15] Dan Klein and Christopher D Manning. A parsing: fast exact viterbi parse selection. In *Proceedings of the 2003 Conference of the North American Chapter of the*

Association for Computational Linguistics on Human Language Technology-Volume 1, pages 40–47, 2003. URL http://www.aclweb.org/anthology/N03-1016.

[16] Kenton Lee, Mike Lewis, and Luke Zettlemoyer. Global Neural CCG Parsing with Optimality Guarantees. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2366–2376, 2016. URL http://aclweb.org/anthology/D16-1262.

[17] Do Kook Choe and Eugene Charniak. Parsing as Language Modeling. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2331–2336, 2016. URL http://aclweb.org/anthology/D16-1257.

[18] Lemao Liu, Muhua Zhu, and Shuming Shi. Improving Sequence-to-Sequence Constituency Parsing. In *AAAI Conference on Artificial Intelligence*, 2018. URL https://aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16347.

[19] James Cross and Liang Huang. Span-Based Constituency Parsing with a Structure-Label System and Provably Optimal Dynamic Oracles. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1–11, 2016. URL http://aclweb.org/anthology/D16-1001.

[20] Jiangming Liu and Yue Zhang. Shift-Reduce Constituent Parsing with Neural Lookahead Features. *Transactions of the Association for Computational Linguistics*, 5:45–58, 2017. URL http://aclweb.org/anthology/Q17-1004.

[21] Carlos Gómez-Rodríguez and David Vilares. Constituent Parsing as Sequence Labeling. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1314–1324, 2018. URL http://aclweb.org/anthology/D18-1162.

[22] Liang Hong, Junekiand Huang. Linear-time Constituency Parsing with RNNs and Dynamic Programming. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, pages 477–483, 2018. URL http://aclweb.org/anthology/P18-2076.

[23] 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. URL http://anthology.aclweb.org/J/J93/J93-2004.pdf.

[24] D Kingma and J Ba Adam. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations (ICLR)*, volume 5, 2015. URL https://arxiv.org/abs/1412.6980.

[25] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017. URL https://arxiv.org/abs/1701.03980.

[26] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016. URL https://arxiv.org/pdf/1609.08144.pdf.

[27] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2227–2237, 2018. URL http://aclweb.org/anthology/N18-1202.