Abstract

We introduce a novel architecture for dependency parsing: stack-pointer networks (STACKPTR). Combining pointer networks (Vinyals et al., 2015) with an internal stack, the proposed model first reads and encodes the whole sentence, then builds the dependency tree top-down (from root-to-leaf) in a depth-first fashion. The stack tracks the status of the depth-first search and the pointer networks select one child for the word at the top of the stack at each step. The STACKPTR parser benefits from the information of the whole sentence and all previously derived subtree structures, and removes the left-to-right restriction in classical transition-based parsers. Yet, the number of steps for building any (including non-projective) parse tree is linear in the length of the sentence just as other transition-based parsers, yielding an efficient decoding algorithm with $O(n^2)$ time complexity. We evaluate our model on 29 treebanks spanning 20 languages and different dependency annotation schemas, and achieve state-of-the-art performance on 21 of them.

1 Introduction

Dependency parsing, which predicts the existence and type of linguistic dependency relations between words, is a first step towards deep language understanding. Its importance is widely recognized in the natural language processing (NLP) community, with it benefiting a wide range of NLP applications, such as coreference resolution (Ng, 2010; Durrett and Klein, 2013; Ma et al., 2016), sentiment analysis (Tai et al., 2015), machine translation (Bastings et al., 2017), information extraction (Nguyen et al., 2009; Angeli et al., 2015; Peng et al., 2017), word sense disambiguation (Fauceglia et al., 2015), and low-resource language processing (McDonald et al., 2013; Ma and Xia, 2014). There are two dominant approaches to dependency parsing (Buchholz and Marsi, 2006; Nivre et al., 2007): local and greedy transition-based algorithms (Yamada and Matsumoto, 2003; Nivre and Scholz, 2004; Zhang and Nivre, 2011; Chen and Manning, 2014), and the globally optimized graph-based algorithms (Eisner, 1996; McDonald et al., 2005a,b; Koo and Collins, 2010).

Transition-based dependency parsers read words sequentially (commonly from left-to-right) and build dependency trees incrementally by making series of multiple choice decisions. The advantage of this formalism is that the number of operations required to build any projective parse tree is linear with respect to the length of the sentence. The challenge, however, is that the decision made at each step is based on local information, leading to error propagation and worse performance compared to graph-based parsers on root and long dependencies (McDonald and Nivre, 2011). Previous studies have explored solutions to address this challenge. Stack LSTMs (Dyer et al., 2015; Ballesteros et al., 2015, 2016) are capable of learning representations of the parser state that are sensitive to the complete contents of the parser’s state. Andor et al. (2016) proposed a globally normalized transition model to replace the locally normalized classifier. However, the parsing accuracy is still behind state-of-the-art graph-based parsers (Dozat and Manning, 2017).

Graph-based dependency parsers, on the other hand, learn scoring functions for parse trees and perform exhaustive search over all possible trees for a sentence to find the globally highest scoring
tree. Incorporating this global search algorithm with distributed representations learned from neural networks, neural graph-based parsers (Kiperwasser and Goldberg, 2016; Wang and Chang, 2016; Kuncoro et al., 2016; Dozat and Manning, 2017) have achieved the state-of-the-art accuracies on a number of treebanks in different languages. Nevertheless, these models, while accurate, are usually slow (e.g. decoding is $O(n^3)$ time complexity for first-order models (McDonald et al., 2005a,b) and higher polynomials for higher-order models (McDonald and Pereira, 2006; Koo and Collins, 2010; Ma and Zhao, 2012b,a)).

In this paper, we propose a novel neural network architecture for dependency parsing, stack-pointer networks (STACKPTR). STACKPTR is a transition-based architecture, with the corresponding asymptotic efficiency, but still maintains a global view of the sentence that proves essential for achieving competitive accuracy. Our STACKPTR parser has a pointer network (Vinyals et al., 2015) as its backbone, and is equipped with an internal stack to maintain the order of head words in tree structures. The STACKPTR parser performs parsing in an incremental, top-down, depth-first fashion; at each step, it generates an arc by assigning a child for the head word at the top of the internal stack. This architecture makes it possible to capture information from the whole sentence and all the previously derived sub-trees, while maintaining a number of parsing steps linear in the sentence length.

We evaluate our parser on 29 treebanks across 20 languages and different dependency annotation schemas, and achieve state-of-the-art performance on 21 of them. The contributions of this work are summarized as follows:

(i) We propose a neural network architecture for dependency parsing that is simple, effective, and efficient.
(ii) Empirical evaluations on benchmark datasets over 20 languages show that our method achieves state-of-the-art performance on 21 different treebanks\(^1\).
(iii) Comprehensive error analysis is conducted to compare the proposed method to a strong graph-based baseline using biaffine attention (Dozat and Manning, 2017).

2 Background
We first briefly describe the task of dependency parsing, setup the notation, and review Pointer Networks (Vinyals et al., 2015).

2.1 Dependency Parsing and Notations
Dependency trees represent syntactic relationships between words in the sentences through labeled directed edges between head words and their dependents. Figure 1 (a) shows a dependency tree for the sentence, “But there were no buyers”.

In this paper, we will use the following notation:

**Input**: $x = \{w_1, \ldots, w_n\}$ represents a generic sentence, where $w_i$ is the $i$th word.

**Output**: $y = \{p_1, p_2, \ldots, p_k\}$ represents a generic (possibly non-projective) dependency tree, where each path $p_i = \$w_{i,1}, w_{i,2}, \ldots, w_{i,l_i}\$ is a sequence of words from the root to a leaf. “$\$” is an universal virtual root that is added to each tree.

**Stack**: $\sigma$ denotes a stack configuration, which is a sequence of words. We use $\sigma[w]$ to represent a stack configuration that pushes word $w$ into the stack $\sigma$.

**Children**: $\text{ch}(w_i)$ denotes the list of all the children (modifiers) of word $w_i$.

2.2 Pointer Networks
Pointer Networks (PTR-NET) (Vinyals et al., 2015) are a variety of neural network capable of learning the conditional probability of an output sequence with elements that are discrete tokens corresponding to positions in an input sequence. This model cannot be trivially expressed by standard sequence-to-sequence networks (Sutskever et al., 2014) due to the variable number of input positions in each sentence. PTR-NET solves the problem by using attention (Bahdanau et al., 2015; Luong et al., 2015) as a pointer to select a member of the input sequence as the output.

Formally, the words of the sentence $x$ are fed one-by-one into the encoder (a multiple-layer bi-directional RNN), producing a sequence of encoder hidden states $s_t$. At each time step $t$, the decoder (a uni-directional RNN) receives the input from last step and outputs decoder hidden state $h_t$. The attention vector $a^t$ is calculated as follows:

$$
e^t = \text{score}(h_t, s_t)$$
$$a^t = \text{softmax}(e^t)$$  \hspace{1cm} (1)

where $\text{score}(\cdot, \cdot)$ is the attention scoring function, which has several variations such as dot-product,
concatenation, and biaffine (Luong et al., 2015). PTR-NET regards the attention vector \(a^t\) as a probability distribution over the source words, i.e. it uses \(a^t_h\) as pointers to select the input elements.

3 Stack-Pointer Networks

3.1 Overview

Similarly to PTR-NET, STACKPTR first reads the whole sentence and encodes each word into the encoder hidden state \(s_i\). The internal stack \(\sigma\) is always initialized with the root symbol \$. At each time step \(t\), the decoder receives the input vector corresponding to the top element of the stack \(\sigma\) (the head word \(w_h\), where \(h\) is the word index), generates the hidden state \(h_t\), and computes the attention vector \(a^t\) using Eq. (1). The parser chooses a specific position \(c\) according to the attention scores in \(a^t\) to generate a new dependency arc \((w_h, w_c)\) by selecting \(w_c\) as a child of \(w_h\). Then the parser pushes \(w_c\) onto the stack, i.e. \(\sigma \rightarrow \sigma|w_c\), and goes to the next step. At one step if the parser points \(w_h\) to itself, i.e. \(c = h\), it indicates that all children of the head word \(w_h\) have already been selected. Then the parser goes to the next step by popping \(w_h\) out of \(\sigma\).

At test time, in order to guarantee a valid dependency tree containing all the words in the input sentences exactly once, the decoder maintains a list of “available” words. At each decoding step, the parser selects a child for the current head word, and removes the child from the list of available words to make sure that it cannot be selected as a child of other head words.

For head words with multiple children, it is possible that there is more than one valid selection for each time step. In order to define a deterministic decoding process to make sure that there is only one ground-truth choice at each step (which is necessary for simple maximum likelihood estimation), a predefined order for each \(\text{ch}(w_i)\) needs to be introduced. The predefined order of children can have different alternatives, such as left-to-right or inside-out\(^2\). In this paper, we adopt the inside-out order\(^3\) since it enables us to utilize second-order sibling information, which has been proven beneficial for parsing performance (McDonald and Pereira, 2006; Koo and Collins, 2010) (see § 3.4 for details). Figure 1 (b) depicts the architecture of STACKPTR and the decoding procedure for the example sentence in Figure 1 (a).

3.2 Encoder

The encoder of our parsing model is based on the bi-directional LSTM-CNN architecture (BLSTM-CNNs) (Chiu and Nichols, 2016; Ma and Hovy, 2016) where CNNs encode character-level information of a word into its character-level representation.

---

\(^2\)Order the children by the distances to the head word on the left side, then the right side.

\(^3\)We also tried left-to-right order which obtained worse parsing accuracy than inside-out.
sentation and BLSTM models context information of each word. Formally, for each word, the CNN, with character embeddings as inputs, encodes the character-level representation. Then the character-level representation vector is concatenated with the word embedding vector to feed into the BLSTM network. To enrich word-level information, we also use POS embeddings. Finally, the encoder outputs a sequence of hidden states $s_i$.

### 3.3 Decoder

The decoder for our parser is a uni-directional LSTM. Different from previous work (Bahdanau et al., 2015; Vinyals et al., 2015) which uses word embeddings of the previous word as the input to the decoder, our decoder receives the encoder hidden state vector $(s_i)$ of the top element in the stack $\sigma$ (see Figure 1 (b)). Compared to word embeddings, the encoder hidden states contain more contextual information, benefiting both the training and decoding procedures. The decoder produces a sequence of decoder hidden states $h_i$, one for each decoding step.

### 3.4 Higher-order Information

As mentioned before, our parser is capable of utilizing higher-order information. In this paper, we incorporate two kinds of higher-order structures — grandparent and sibling. A sibling structure is a head word with two successive modifiers, and a grandparent structure is a pair of dependencies connected head-to-tail:

![Diagram of sibling and grandchild structures]

To utilize higher-order information, the decoder’s input at each step is the sum of the encoder hidden states of three words:

$$\beta_t = s_h + s_g + s_s$$

where $\beta_t$ is the input vector of decoder at time $t$ and $h, g, s$ are the indices of the head word and its grandparent and sibling, respectively. Figure 1 (b) illustrates the details. Here we use the element-wise sum operation instead of concatenation because it does not increase the dimension of the input vector $\beta_t$, thus introducing no additional model parameters.

### 3.5 Biaffine Attention Mechanism

For attention score function (Eq. (1)), we adopt the biaffine attention mechanism (Luong et al., 2015; Dozat and Manning, 2017):

$$e_i^t = h_i^T W s_i + U^T h_t + V^T s_i + b$$

where $W, U, V, b$ are parameters, denoting the weight matrix of the bi-linear term, the two weight vectors of the linear terms, and the bias vector.

As discussed in Dozat and Manning (2017), applying a multilayer perceptron (MLP) to the output vectors of the BLSTM before the score function can both reduce the dimensionality and overfitting of the model. We follow this work by using a one-layer perceptron to $s_i$ and $h_i$ with elu (Clevert et al., 2015) as its activation function.

Similarly, the dependency label classifier also uses a biaffine function to score each label, given the head word vector $h_i$ and child vector $s_i$ as inputs. Again, we use MLPs to transform $h_i$ and $s_i$ before feeding them into the classifier.

### 3.6 Training Objectives

The STACKPtr parser is trained to optimize the probability of the dependency trees given sentences: $P_{\theta}(y|x)$, which can be factorized as:

$$P_{\theta}(y|x) = \prod_{i=1}^{k} P_{\theta}(p_i|p_{<i}, x)$$

$$= \prod_{i=1}^{k} \prod_{j=1}^{l_i} P_{\theta}(c_{i,j}|c_{i,<j}, p_{<i}, x),$$

where $\theta$ represents model parameters. $p_{<i}$ denotes the preceding paths that have already been generated. $c_{i,j}$ represents the $j$th word in $p_i$ and $c_{i,<j}$ denotes all the proceeding words on the path $p_i$. Thus, the STACKPtr parser is an autoregressive model, like sequence-to-sequence models, but it factors the distribution according to a top-down tree structure as opposed to a left-to-right chain. We define $P_{\theta}(c_{i,j}|c_{i,<j}, p_{<i}, x) = a^t$, where attention vector $a^t$ (of dimension $n$) is used as the distribution over the indices of words in a sentence.

#### Arc Prediction

Our parser is trained by optimizing the conditional likelihood in Eq (2), which is implemented as the cross-entropy loss.

#### Label Prediction

We train a separated multi-class classifier in parallel to predict the dependency labels. Following Dozat and Manning (2017), the classifier takes the information of the
head word and its child as features. The label classifier is trained simultaneously with the parser by optimizing the sum of their objectives.

3.7 Discussion

Time Complexity. The number of decoding steps to build a parse tree for a sentence of length n is \(2n - 1\), linear in n. Together with the attention mechanism (at each step, we need to compute the attention vector \(a_t\), whose runtime is \(O(n)\)), the time complexity of decoding algorithm is \(O(n^2)\), which is more efficient than graph-based parsers that have \(O(n^3)\) or worse complexity when using dynamic programming or maximum spanning tree (MST) decoding algorithms.

Top-down Parsing. When humans comprehend a natural language sentence, they arguably do it in an incremental, left-to-right manner. However, when humans consciously annotate a sentence with syntactic structure, they rarely ever process in fixed left-to-right order. Rather, they start by reading the whole sentence, then seeking the main predicates, jumping back-and-forth over the sentence and recursively proceeding to the sub-tree structures governed by certain head words. Our parser follows a similar kind of annotation process: starting from reading the whole sentence, and processing in a top-down manner by finding the main predicates first and only then search for sub-trees governed by them. When making latter decisions, the parser has access to the entire structure built in earlier steps.

3.8 Implementation Details

Pre-trained Word Embeddings. For all the parsing models in different languages, we initialize word vectors with pretrained word embeddings. For Chinese, Dutch, English, German and Spanish, we use the structured-skipgram (Ling et al., 2015) embeddings. For other languages we use Polyglot embeddings (Al-Rfou et al., 2013).

Optimization. Parameter optimization is performed with the Adam optimizer (Kingma and Ba, 2014) with \(\beta_1 = \beta_2 = 0.9\). We choose an initial learning rate of \(\eta_0 = 0.001\). The learning rate \(\eta\) is annealed by multiplying a fixed decay rate \(\rho = 0.75\) when parsing performance stops increasing on validation sets. To reduce the effects of “gradient exploding”, we use gradient clipping of 5.0 (Pascaru et al., 2013).

Dropout Training. To mitigate overfitting, we apply dropout (Srivastava et al., 2014; Ma et al., 2017). For BLSTM, we use recurrent dropout (Gal and Ghahramani, 2016) with a drop rate of 0.33 between hidden states and 0.33 between layers. Following Dozat and Manning (2017), we also use embedding dropout with a rate of 0.33 on all word, character, and POS embeddings.

Hyper-Parameters. Some parameters are chosen from those reported in Dozat and Manning (2017). We use the same hyper-parameters across the models on different treebanks and languages, due to time constraints. The details of the chosen hyper-parameters for all experiments are summarized in Appendix A.

4 Experiments

4.1 Setup

We evaluate our StackPTR parser mainly on three treebanks: the English Penn Treebank (PTB version 3.0) (Marcus et al., 1993), the Penn Chinese Treebank (CTB version 5.1) (Xue et al., 2002), and the German CoNLL 2009 corpus (Hajič et al., 2009). We use the same experimental settings as Kuncoro et al. (2016).

To make a thorough empirical comparison with previous studies, we also evaluate our system on treebanks from CoNLL shared task and the Universal Dependency (UD) Treebanks\(^4\). For the CoNLL Treebanks, we use the English treebank from CoNLL-2008 shared task (Surdeanu et al., 2008) and all 13 treebanks from CoNLL-2006 shared task (Buchholz and Marsi, 2006). The experimental settings are the same as Ma and Hovy (2015). For UD Treebanks, we select 12 languages. The details of the treebanks and experimental settings are in §4.5 and Appendix B.

Evaluation Metrics Parsing performance is measured with five metrics: unlabeled attachment score (UAS), labeled attachment score (LAS), unlabeled complete match (UCM), labeled complete match (LCM), and root accuracy (RA). Following previous work (Kuncoro et al., 2016; Dozat and Manning, 2017), we report results excluding punctuations for Chinese and English. For each experiment, we report the mean values with corresponding standard deviations over 5 repetitions.

\(^4\)http://universaldependencies.org/
Baseline For fair comparison of the parsing performance, we re-implemented the graph-based Deep Biaffine (BiAF) parser (Dozat and Manning, 2017), which achieved state-of-the-art results on a wide range of languages. Our re-implementation adds character-level information using the same LSTM-CNN encoder as our model (§3.2) to the original BiAF model, which boosts its performance on all languages.

4.2 Main Results

We first conduct experiments to demonstrate the effectiveness of our neural architecture by comparing with the strong baseline BiAF. We compare the performance of four variations of our model with different decoder inputs — Org, +gpar, +sib and Full — where the Org model utilizes only the encoder hidden states of head words, while the +gpar and +sib models augments the original one with grandparent and sibling information, respectively. The Full model includes all the three information as inputs.

Figure 2 illustrates the performance (five metrics) of different variations of our STACKPTR parser together with the results of baseline BiAF re-implemented by us, on the test sets of the three languages. On UAS and LAS, the Full variation of STACKPTR with decoding beam size 10 outperforms BiAF on Chinese, and obtains competitive performance on English and German. An interesting observation is that the Full model achieves the best accuracy on English and Chinese, while performs slightly worse than +sib on German. This shows that the importance of higher-order information varies in languages. On LCM and UCM, STACKPTR significantly outperforms BiAF on all languages, showing the superiority of our parser on complete sentence parsing. The results of our parser on RA are slightly worse than BiAF. More details of results are provided in Appendix C.

4.3 Comparison with Previous Work

Table 1 illustrates the UAS and LAS of the four versions of our model (with decoding beam size 10) on the three treebanks, together with previous top-performing systems for comparison. Note that the results of STACKPTR and our re-implementation of BiAF are the average of 5 repetitions instead of a single run. Our Full model significantly outperforms all the transition-based parsers on all three languages, and achieves better results than most graph-based parsers.
Table 1: UAS and LAS of four versions of our model on test sets for three languages, together with top-performing parsing systems. “T” and “G” indicate transition- and graph-based models, respectively. For BiAF, we provide the original results reported in Dozat and Manning (2017) and our re-implementation. For STACKPTR and our re-implementation of BiAF, we report the average over 5 runs.

Table 2: Parsing performance on the test data of PTB with different versions of POS tags.

4.4 Error Analysis
In this section, we characterize the errors made by BiAF and STACKPTR by presenting a number of experiments that relate parsing errors to a set of linguistic and structural properties. For simplicity, we follow McDonald and Nivre (2011) and report labeled parsing metrics (either accuracy, precision, or recall) for all experiments.

4.4.1 Length and Graph Factors
Following McDonald and Nivre (2011), we analyze parsing errors related to structural factors.

**Sentence Length.** Figure 3 (a) shows the accuracy of both parsing models relative to sentence lengths. Consistent with the analysis in McDonald and Nivre (2011), STACKPTR tends to perform better on shorter sentences, which makes fewer parsing decisions, significantly reducing the chance of error propagation.

**Dependency Length.** Figure 3 (b) measures the precision and recall relative to dependency lengths. While the graph-based BiAF parser still performs better for longer dependency arcs and transition-based STACKPTR parser does better for shorter ones, the gap between the two systems is marginal, much smaller than that shown
in McDonald and Nivre (2011). One possible reason is that, unlike traditional transition-based parsers that scan the sentence from left to right, STACKPTR processes in a top-down manner, thus sometimes unnecessarily creating shorter dependency arcs first.

**Root Distance.** Figure 3 (c) plots the precision and recall of each system for arcs of varying distance to the root. Different from the observation in McDonald and Nivre (2011), STACKPTR does not show an obvious advantage on the precision for arcs further away from the root. Furthermore, the STACKPTR parser does not have the tendency to over-predict root modifiers reported in McDonald and Nivre (2011). This behavior can be explained using the same reasoning as above: the fact that arcs further away from the root are usually constructed early in the parsing algorithm of traditional transition-based parsers is not true for the STACKPTR parser.

### 4.4.2 Effect of POS Embedding

The only prerequisite information that our parsing model relies on is POS tags. With the goal of achieving an end-to-end parser, we explore the effect of POS tags on parsing performance. We run experiments on PTB using our STACKPTR parser with gold-standard and predicted POS tags, and without tags, respectively. STACKPTR in these experiments is the Full model with beam=10.

Table 2 gives results of the parsers with different versions of POS tags on the test data of PTB.

The parser with gold-standard POS tags significantly outperforms the other two parsers, showing that dependency parsers can still benefit from accurate POS information. The parser with predicted (imperfect) POS tags, however, performs even slightly worse than the parser without using POS tags. It illustrates that an end-to-end parser that doesn’t rely on POS information can obtain competitive (or even better) performance than parsers using imperfect predicted POS tags, even if the POS tagger is relatively high accuracy (accuracy > 97% in this experiment on PTB).

### 4.5 Experiments on Other Treebanks

#### 4.5.1 CoNLL Treebanks

Table 3 summarizes the parsing results of our model on the test sets of 14 treebanks from the CoNLL shared task, along with the state-of-the-art baselines. Along with BiAF, we also list the performance of the bi-directional attention based Parser (Bi-Att) (Cheng et al., 2016) and the neural MST parser (NeuroMST) (Ma and Hovy, 2017) for comparison. Our parser achieves state-of-the-art performance on both UAS and LAS on eight languages — Arabic, Czech, English, German, Portuguese, Slovene, Spanish, and Swedish. On Bulgarian and Dutch, our parser obtains the best UAS. On other languages, the performance of our parser is competitive with BiAF, and significantly better than others. The only exception is Japanese, on which NeuroMST obtains the best scores.
4.5.2 UD Treebanks

For UD Treebanks, we select 12 languages — Bulgarian, Catalan, Czech, Dutch, English, French, German, Italian, Norwegian, Romanian, Russian and Spanish. For all the languages, we adopt the standard training/dev/test splits, and use the universal POS tags (Petrov et al., 2012) provided in each treebank. The statistics of these corpora are provided in Appendix B.

Table 4 summarizes the results of the STACKPTR parser, along with BIAF for comparison, on both the development and test datasets for each language. First, both BIAF and STACKPTR parsers achieve relatively high parsing accuracies on all the 12 languages — all with UAS are higher than 90%. On nine languages — Catalan, Czech, Dutch, English, French, German, Norwegian, Russian and Spanish — STACKPTR outperforms BIAF for both UAS and LAS. On Bulgarian, STACKPTR achieves slightly better UAS while LAS is slightly worse than BIAF. On Italian and Romanian, STACKPTR obtains marginally better parsing performance than BIAF.

5 Conclusion

In this paper, we proposed STACKPTR, a transition-based neural network architecture, for dependency parsing. Combining pointer networks with an internal stack to track the status of the top-down, depth-first search in the decoding procedure, the STACKPTR parser is able to capture information from the whole sentence and all the previously derived subtrees, removing the left-to-right restriction in classical transition-based parsers, while maintaining linear parsing steps, w.r.t the length of the sentences. Experimental results on 29 treebanks show the effectiveness of our parser across 20 languages, by achieving state-of-the-art performance on 21 corpora.

There are several potential directions for future work. First, we intend to consider how to conduct experiments to improve the analysis of parsing errors qualitatively and quantitatively. Another interesting direction is to further improve our model by exploring reinforcement learning approaches to learn an optimal order for the children of head words, instead of using a predefined fixed order.

Acknowledgements

The authors thank Chunting Zhou, Di Wang and Zhengzhong Liu for their helpful discussions. This research was supported in part by DARPA grant FA8750-18-2-0018 funded under the AIDA program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References


Gabor Angeli, Melvin Jose Johnson Premkumar, and Christopher D. Manning. 2015. Leveraging linguistic structure for open domain information extraction.


Appendix: Stack-Pointer Network for Dependency Parsing

Appendix A: Hyper-Parameters

Table 5 summarizes the chosen hyper-parameters used for all the experiments in this paper. Some parameters are chosen directly or similarly from those reported in Dozat and Manning (2017). We use the same hyper-parameters across the models on different treebanks and languages, due to time constraints.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Hyper-parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNN</td>
<td>window size</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>number of filters</td>
<td>50</td>
</tr>
<tr>
<td>LSTM</td>
<td>encoder layers</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>encoder size</td>
<td>512</td>
</tr>
<tr>
<td></td>
<td>decoder layers</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>decoder size</td>
<td>512</td>
</tr>
<tr>
<td>MLP</td>
<td>arc MLP size</td>
<td>512</td>
</tr>
<tr>
<td></td>
<td>label MLP size</td>
<td>128</td>
</tr>
<tr>
<td>Dropout</td>
<td>embeddings</td>
<td>0.33</td>
</tr>
<tr>
<td></td>
<td>LSTM hidden states</td>
<td>0.33</td>
</tr>
<tr>
<td></td>
<td>LSTM layers</td>
<td>0.33</td>
</tr>
<tr>
<td>Learning</td>
<td>optimizer</td>
<td>Adam</td>
</tr>
<tr>
<td></td>
<td>initial learning rate</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>(β₁, β₂)</td>
<td>(0.9, 0.9)</td>
</tr>
<tr>
<td></td>
<td>decay rate</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td>gradient clipping</td>
<td>5.0</td>
</tr>
</tbody>
</table>

Table 5: Hyper-parameters for all experiments.
## Appendix B: UD Treebanks

Table 6 shows the corpora statistics of the treebanks for 12 languages. For evaluation, we report results excluding punctuation, which is any tokens with POS tags “PUNCT” or “SYM”.

<table>
<thead>
<tr>
<th>Corpora</th>
<th>#Sent</th>
<th>#Token (w.o punct)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bulgarian</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTB Training</td>
<td>8,907</td>
<td>124,336 (106,813)</td>
</tr>
<tr>
<td>Dev</td>
<td>1,115</td>
<td>16,089 (13,822)</td>
</tr>
<tr>
<td>Test</td>
<td>1,116</td>
<td>15,724 (13,456)</td>
</tr>
<tr>
<td>Catalan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AnCora Training</td>
<td>13,123</td>
<td>417,587 (371,981)</td>
</tr>
<tr>
<td>Dev</td>
<td>1,709</td>
<td>56,482 (50,452)</td>
</tr>
<tr>
<td>Test</td>
<td>1,846</td>
<td>57,738 (51,324)</td>
</tr>
<tr>
<td>Czech</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PDT, CAC Training</td>
<td>102,993</td>
<td>1,806,230 (1,542,805)</td>
</tr>
<tr>
<td>Dev</td>
<td>11,311</td>
<td>191,679 (163,387)</td>
</tr>
<tr>
<td>Test</td>
<td>12,203</td>
<td>205,597 (174,771)</td>
</tr>
<tr>
<td>Dutch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alpino Training</td>
<td>18,310</td>
<td>267,289 (234,104)</td>
</tr>
<tr>
<td>Dev</td>
<td>1,518</td>
<td>22,091 (19,042)</td>
</tr>
<tr>
<td>Test</td>
<td>1,396</td>
<td>21,126 (18,310)</td>
</tr>
<tr>
<td>LassySmall Training</td>
<td>12,543</td>
<td>204,585 (180,308)</td>
</tr>
<tr>
<td>Dev</td>
<td>2,002</td>
<td>25,148 (21,998)</td>
</tr>
<tr>
<td>Test</td>
<td>2,077</td>
<td>25,096 (21,898)</td>
</tr>
<tr>
<td>English</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EWT Training</td>
<td>14,554</td>
<td>356,638 (316,780)</td>
</tr>
<tr>
<td>Dev</td>
<td>1,478</td>
<td>35,768 (31,896)</td>
</tr>
<tr>
<td>Test</td>
<td>416</td>
<td>10,020 (8,795)</td>
</tr>
<tr>
<td>French</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GSD Training</td>
<td>13,841</td>
<td>263,536 (229,204)</td>
</tr>
<tr>
<td>Dev</td>
<td>799</td>
<td>12,348 (10,727)</td>
</tr>
<tr>
<td>Test</td>
<td>977</td>
<td>16,268 (13,929)</td>
</tr>
<tr>
<td>German</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GSD Training</td>
<td>12,838</td>
<td>270,703 (239,836)</td>
</tr>
<tr>
<td>Dev</td>
<td>564</td>
<td>11,908 (10,490)</td>
</tr>
<tr>
<td>Test</td>
<td>482</td>
<td>10,417 (9,237)</td>
</tr>
<tr>
<td>Italian</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ISDT Training</td>
<td>29,870</td>
<td>48,9217 (43,2597)</td>
</tr>
<tr>
<td>Dev</td>
<td>4,300</td>
<td>67,619 (59,784)</td>
</tr>
<tr>
<td>Test</td>
<td>3,450</td>
<td>54,739 (48,588)</td>
</tr>
<tr>
<td>Norwegian</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bokmaal Training</td>
<td>8,043</td>
<td>185,113 (161,429)</td>
</tr>
<tr>
<td>Dev</td>
<td>752</td>
<td>17,074 (14,851)</td>
</tr>
<tr>
<td>Test</td>
<td>729</td>
<td>16,268 (14,241)</td>
</tr>
<tr>
<td>Nynorsk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RRT Training</td>
<td>28,492</td>
<td>827,053 (730,062)</td>
</tr>
<tr>
<td>Dev</td>
<td>4,300</td>
<td>89,487 (78,951)</td>
</tr>
<tr>
<td>Test</td>
<td>2,174</td>
<td>64,617 (56,973)</td>
</tr>
</tbody>
</table>

Table 6: Corpora statistics of UD Treebanks for 12 languages. #Sent and #Token refer to the number of sentences and the number of words (w./w.o punctuations) in each data set, respectively.
## Appendix C: Main Results

Table 7 illustrates the details of the experimental results. For each STACKPRTR parsing model, we ran experiments with decoding beam size equals to 1, 5, and 10. For each experiment, we report the mean values with corresponding standard deviations over 5 runs.

<table>
<thead>
<tr>
<th>Model</th>
<th>beam</th>
<th>Dev</th>
<th>English</th>
<th>Test</th>
<th>Chinese</th>
<th>German</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>UAS</td>
<td>LAS</td>
<td>UCM</td>
<td>LCM</td>
<td>UAS</td>
</tr>
<tr>
<td>BiAF</td>
<td>–</td>
<td>95.73±0.04</td>
<td>93.97±0.06</td>
<td>60.58±0.77</td>
<td>47.47±0.63</td>
<td>95.84±0.06</td>
</tr>
<tr>
<td>Basic</td>
<td>1</td>
<td>95.71±0.02</td>
<td>93.88±0.03</td>
<td>62.33±0.33</td>
<td>47.75±0.32</td>
<td>95.71±0.06</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>95.71±0.04</td>
<td>93.88±0.05</td>
<td>62.40±0.45</td>
<td>47.80±0.44</td>
<td>95.76±0.11</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>95.72±0.03</td>
<td>93.89±0.04</td>
<td>62.40±0.45</td>
<td>47.80±0.44</td>
<td>95.77±0.11</td>
</tr>
<tr>
<td>+gpar</td>
<td>1</td>
<td>95.68±0.04</td>
<td>93.82±0.02</td>
<td>62.81±0.36</td>
<td>47.32±0.14</td>
<td>95.73±0.04</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>95.67±0.01</td>
<td>93.83±0.02</td>
<td>61.93±0.32</td>
<td>47.44±0.20</td>
<td>95.76±0.06</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>95.69±0.02</td>
<td>93.83±0.02</td>
<td>61.95±0.32</td>
<td>47.44±0.20</td>
<td>95.78±0.05</td>
</tr>
<tr>
<td>+sib</td>
<td>1</td>
<td>95.75±0.03</td>
<td>93.93±0.04</td>
<td>61.93±0.49</td>
<td>47.66±0.48</td>
<td>95.77±0.15</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>95.74±0.02</td>
<td>93.93±0.05</td>
<td>62.16±0.22</td>
<td>47.68±0.54</td>
<td>95.84±0.09</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>95.75±0.02</td>
<td>93.94±0.06</td>
<td>62.17±0.20</td>
<td>47.68±0.54</td>
<td>95.85±0.10</td>
</tr>
<tr>
<td>Full</td>
<td>1</td>
<td>95.63±0.08</td>
<td>93.78±0.08</td>
<td>61.56±0.63</td>
<td>47.12±0.36</td>
<td>95.79±0.04</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>95.75±0.06</td>
<td>93.90±0.08</td>
<td>62.06±0.42</td>
<td>47.43±0.36</td>
<td>95.87±0.04</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>95.75±0.06</td>
<td>93.90±0.08</td>
<td>62.08±0.39</td>
<td>47.43±0.36</td>
<td>95.87±0.04</td>
</tr>
</tbody>
</table>

Table 7: Parsing performance of different variations of our model on both the development and test sets for three languages, together with the BiAF parser as the baseline. Best results are highlighted with bold print.