

Efficient Second-Order TreeCRF for Neural Dependency Parsing

Yu Zhang, Zhenghua Li*, Min Zhang

Institute of Artificial Intelligence, School of Computer Science and Technology,
Soochow University, Suzhou, China

yzhang.cs@outlook.com, {zhli13,minzhang}@suda.edu.cn

Abstract

In the deep learning (DL) era, parsing models are extremely simplified with little hurt on performance, thanks to the remarkable capability of multi-layer BiLSTMs in context representation. As the most popular graph-based dependency parser due to its high efficiency and performance, the biaffine parser directly scores single dependencies under the arc-factorization assumption, and adopts a very simple local token-wise cross-entropy training loss. This paper for the first time presents a second-order TreeCRF extension to the biaffine parser. For a long time, the complexity and inefficiency of the inside-outside algorithm hinder the popularity of TreeCRF. To address this issue, we propose an effective way to batchify the inside and Viterbi algorithms for direct large matrix operation on GPUs, and to avoid the complex outside algorithm via efficient back-propagation. Experiments and analysis on 27 datasets from 13 languages clearly show that techniques developed before the DL era, such as structural learning (global TreeCRF loss) and high-order modeling are still useful, and can further boost parsing performance over the state-of-the-art biaffine parser, especially for partially annotated training data. We release our code at <https://github.com/yzhangcs/crfpar>.

1 Introduction

As a fundamental task in NLP, dependency parsing has attracted a lot of research interest due to its simplicity and multilingual applicability in capturing both syntactic and semantic information (Nivre et al., 2016). Given an input sentence $x = w_0 w_1 \dots w_n$, a dependency tree, as depicted in Figure 1, is defined as $y = \{(i, j, l), 0 \leq i \leq n, 1 \leq j \leq n, l \in \mathcal{L}\}$, where (i, j, l) is a dependency from the head word w_i to the modifier word

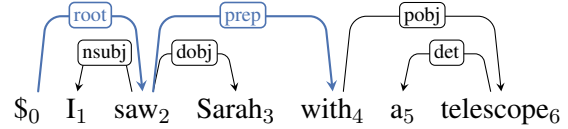


Figure 1: An example full dependency tree. In the case of partial annotation, only some (not all) dependencies are annotated, for example, the two thick (blue) arcs.

w_j with the relation label $l \in \mathcal{L}$. Between two mainstream approaches, this work focuses on the graph-based paradigm (vs. transition-based).

Before the deep learning (DL) era, graph-based parsing relies on many hand-crafted features and differs from its neural counterpart in two major aspects. First, structural learning, i.e., explicit awareness of tree structure constraints during training, is indispensable. Most non-neural graph-based parsers adopt the max-margin training algorithm, which first predicts a highest-scoring *tree* with the current model, and then updates feature weights so that the correct tree has a higher score than the predicted tree.

Second, high-order modeling brings significant accuracy gains. The basic first-order model factors the score of a tree into independent scores of single dependencies (McDonald et al., 2005a). Second-order models were soon proposed to incorporate scores of dependency pairs, such as adjacent-siblings (McDonald and Pereira, 2006) and grand-parent-child (Carreras, 2007; Koo and Collins, 2010), showing significant accuracy improvement yet with the cost of lower efficiency and more complex decoding algorithms.¹

In contrast, neural graph-based dependency parsing exhibits an opposite development trend. Pei et al. (2015) propose to use feed-forward neural

*Corresponding author

¹Third-order and fourth-order models show little accuracy improvement probably due to the feature sparseness problem (Koo and Collins, 2010; Ma and Zhao, 2012).

networks for automatically learning combinations of dozens of atomic features similar to [Chen and Manning \(2014\)](#), and for computing subtree scores. They show that incorporating second-order scores of adjacent-sibling subtrees significantly improved performance. Then, both [Wang and Chang \(2016\)](#) and [Kiperwasser and Goldberg \(2016\)](#) propose to utilize BiLSTM as an encoder and use minimal feature sets for scoring single dependencies in a first-order parser. These three representative works all employ global max-margin training. [Dozat and Manning \(2017\)](#) propose a strong and efficient biaffine parser and obtain state-of-the-art accuracy on a variety of datasets and languages. The biaffine parser is also first-order and employs simpler and more efficient non-structural training via local head selection for each token ([Zhang et al., 2017](#)).

Observing such contrasting development, we try to make a connection between pre-DL and DL techniques for graph-based parsing. Specifically, **the first question** to be addressed in this work is: *can previously useful techniques such as structural learning and high-order modeling further improve the state-of-the-art² biaffine parser, and if so, in which aspects are they helpful?*

For structural learning, we focus on the more complex and less popular TreeCRF instead of max-margin training. The reason is two-fold. First, estimating probability distribution is the core issue in modern data-driven NLP methods ([Le and Zuidema, 2014](#)). The probability of a tree, i.e., $p(\mathbf{y} \mid \mathbf{x})$, is potentially more useful than an unbounded score $s(\mathbf{x}, \mathbf{y})$ for high-level NLP tasks when utilizing parsing outputs. Second, as a theoretically sound way to measure model confidence of subtrees, marginal probabilities can support Minimum Bayes Risk (MBR) decoding ([Smith and Smith, 2007](#)), and are also proven to be crucial for the important research line of token-level active learning based on partial trees ([Li et al., 2016](#)).

One probable reason for the less popularity of TreeCRF, despite its usefulness, is due to the complexity and inefficiency of the inside-outside algorithm, especially the outside algorithm. As far as we know, all existing works compute the inside and outside algorithms on CPUs. The inefficiency issue becomes more severe in the DL era, due to

the unmatched speed of CPU and GPU computation. This leads to **the second question**: *can we batchify the inside-outside algorithm and perform computation directly on GPUs?* In that case, we can employ efficient TreeCRF as a built-in component in DL toolkits such as PyTorch for wider applications ([Cai et al., 2017](#); [Le and Zuidema, 2014](#)).

Overall, targeted at the above two questions, this work makes the following contributions.

- We for the first time propose second-order TreeCRF for neural dependency parsing. We also propose an efficient and effective triaffine operation for scoring second-order subtrees.
- We propose to batchify the inside algorithm via direct large tensor computation on GPUs, leading to very efficient TreeCRF loss computation. We show that the complex outside algorithm is no longer needed for the computation of gradients and marginal probabilities, and can be replaced by the equally efficient back-propagation process.
- We conduct experiments on 27 datasets from 13 languages. The results and analysis show that both structural learning and high-order modeling are still beneficial to the state-of-the-art biaffine parser in many ways in the DL era.

2 The Basic Biaffine Parser

We re-implement the state-of-the-art biaffine parser ([Dozat and Manning, 2017](#)) with two modifications, i.e., using CharLSTM word representation vectors instead of POS tag embeddings, and the first-order Eisner algorithm ([Eisner, 2000](#)) for projective decoding instead of the non-projective MST algorithm.

Scoring architecture. Figure 2 shows the scoring architecture, consisting of four components.

Input vectors. The i th input vector is composed of two parts: the word embedding and the CharLSTM word representation vector of w_i .

$$\mathbf{e}_i = \text{emb}(w_i) \oplus \text{CharLSTM}(w_i) \quad (1)$$

where $\text{CharLSTM}(w_i)$ is obtained by feeding w_i into a BiLSTM and then concatenating the two last hidden vectors ([Lample et al., 2016](#)). We find that replacing POS tag embeddings with

² Though many recent works report higher performance with extra resources, for example contextualized word representations learned from large-scale unlabeled texts under language model loss, they either adopt the same architecture or achieve similar performance under fair comparison.

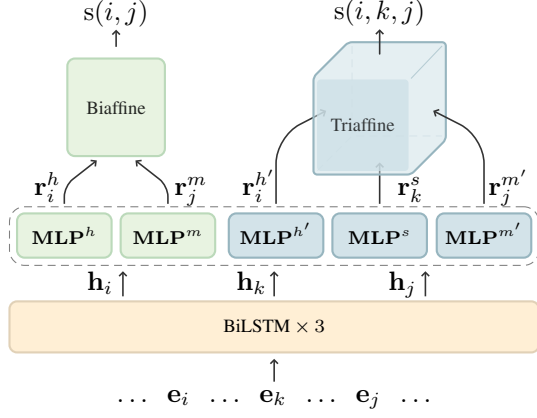


Figure 2: Scoring architecture with second-order extension.

CharLSTM(w_i) leads to consistent improvement, and also simplifies the multilingual experiments by avoiding POS tag generation (especially n-fold jackknifing on training data).

BiLSTM encoder. To encode the sentential contexts, the parser applies three BiLSTM layers over $e_0 \dots e_n$. The output vector of the top-layer BiLSTM for the i th word is denoted as h_i .

MLP feature extraction. Two shared MLPs are applied to h_i , obtaining two lower-dimensional vectors that detain only syntax-related features:

$$r_i^h; r_i^m = \text{MLP}^{h/m}(h_i) \quad (2)$$

where r_i^h and r_i^m are the representation vector of w_i as a head word and a modifier word respectively.

Biaffine scorer. Dozat and Manning (2017) for the first time propose to compute the score of a dependency $i \rightarrow j$ via biaffine attention:

$$s(i, j) = \begin{bmatrix} r_j^m \\ 1 \end{bmatrix}^T \mathbf{W}^{biaffine} r_i^h \quad (3)$$

where $\mathbf{W}^{biaffine} \in \mathbb{R}^{d \times d}$. The computation is extremely efficient on GPUs.

Local token-wise training loss. The biaffine parser adopts a simple non-structural training loss, trying to independently maximize the local probability of the correct head word for each word. For a gold-standard head-modifier pair (w_i, w_j) in a training instance, the cross-entropy loss is

$$L(i, j) = -\log \frac{e^{s(i, j)}}{\sum_{0 \leq k \leq n} e^{s(k, j)}} \quad (4)$$

In other words, the model is trained based on simple head selection, without considering the tree structure at all, and losses of all words in a mini-batch are accumulated.

Decoding. Having scores of all dependencies, we adopt the first-order Eisner algorithm with time complexity of $O(n^3)$ to find the optimal tree.

$$y^* = \arg \max_y \left[s(\mathbf{x}, y) \equiv \sum_{i \rightarrow j \in y} s(i, j) \right] \quad (5)$$

Handling dependency labels. The biaffine parser treats skeletal tree searching and labeling as two independent (training phase) and cascaded (parsing phase) tasks. This work follows the same strategy for simplicity. Please refer to Dozat and Manning (2017) for details.

3 Second-order TreeCRF

This work substantially extends the biaffine parser in two closely related aspects: using probabilistic TreeCRF for structural training and explicitly incorporating high-order subtree scores. Specifically, we further incorporate adjacent-sibling subtree scores into the basic first-order model:³

$$s(\mathbf{x}, y) = \sum_{i \rightarrow j \in y} s(i, j) + \sum_{i \rightarrow \{k, j\} \in y} s(i, k, j) \quad (6)$$

where k and j are two adjacent modifiers of i and satisfy either $i < k < j$ or $j < k < i$.

As a probabilistic model, TreeCRF computes the conditional probability of a tree as

$$p(y | \mathbf{x}) = \frac{e^{s(\mathbf{x}, y)}}{Z(\mathbf{x}) \equiv \sum_{y' \in \mathcal{Y}(\mathbf{x})} e^{s(\mathbf{x}, y')}} \quad (7)$$

where $\mathcal{Y}(\mathbf{x})$ is the set of all legal (projective) trees for \mathbf{x} , and $Z(\mathbf{x})$ is commonly referred to as the normalization (or partition) term.

During training, TreeCRF employs the following structural training loss to maximize the conditional probability of the gold-standard tree y given \mathbf{x} .

$$\begin{aligned} L(\mathbf{x}, y) &= -\log p(y | \mathbf{x}) \\ &= -s(\mathbf{x}, y) + \log Z(\mathbf{x}) \end{aligned} \quad (8)$$

³ This work can be further extended to incorporate grandparent-modifier subtree scores based on the viterbi algorithm of $O(n^4)$ time complexity proposed by Koo and Collins (2010), which we leave for future work.

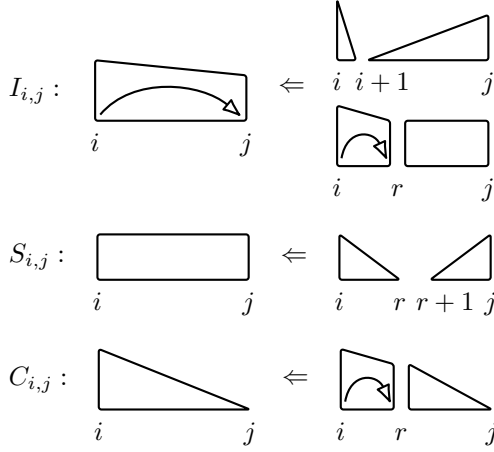


Figure 3: Diagrams of the second-order inside algorithm based on bottom-up dynamic programming.

3.1 Scoring Second-order Subtrees

To avoid major modification to the original scoring architecture, we take a straightforward extension to obtain scores of adjacent-sibling subtrees. First, we employ three extra MLPs to perform similar feature extraction.

$$\mathbf{r}_i^{h'}; \mathbf{r}_i^s; \mathbf{r}_i^{m'} = \text{MLP}^{h'/s/m'}(\mathbf{h}_i) \quad (9)$$

where $\mathbf{r}_i^{h'}; \mathbf{r}_i^s; \mathbf{r}_i^{m'}$ are the representation vectors of w_i as head, sibling, and modifier respectively.⁴

Then, we propose a natural extension to the bi-affine equation, and employ triaffine for score computation over three vectors.⁵

$$s(i, k, j) = \begin{bmatrix} \mathbf{r}_k^s \\ 1 \end{bmatrix}^T \mathbf{r}_i^{h'}{}^T \mathbf{W}^{\text{triaffine}} \begin{bmatrix} \mathbf{r}_j^{m'} \\ 1 \end{bmatrix} \quad (10)$$

where $\mathbf{W}^{\text{triaffine}} \in \mathbb{R}^{d' \times d' \times d'}$ is a three-way tensor. The triaffine computation can be quite efficiently performed with the einsum function on PyTorch.

3.2 Computing TreeCRF Loss Efficiently

The key to TreeCRF loss is how to efficiently compute $\log Z(\mathbf{x})$, as shown in Equation 8. This problem has been well solved long before the DL era for non-neural dependency parsing. Straightforwardly, we can directly extend the viterbi decoding algorithm by replacing max product with sum

⁴ Another way is to use one extra MLP for sibling representation, and re-use head and modifier representation from the basic first-order components, which however leads to inferior performance in our preliminary experiments.

⁵ We have also tried the approximate method of Wang et al. (2019), which uses three biaffine operations to simulate the interactions of three input vectors, but observed inferior performance. We omit the results due to the space limitation.

Algorithm 1 Second-order Inside Algorithm.

```

1: define:  $I, S, C \in \mathbb{R}^{n \times n \times B}$   $\triangleright B$  is #sents in a batch
2: initialize:  $C_{i,i} = \log e^0 = 0, 0 \leq i \leq n$ 
3: for  $w = 1$  to  $n$  do  $\triangleright$  span width
4:   Batchify:  $0 \leq i; j = i + w \leq n$ 
5:      $I_{i,j} = \log \left( \sum_{i < r < j} e^{C_{i,i} + C_{j,i+1} + I_{i,r} + S_{r,j} + s(i,r,j)} \right) + s(i,j)$ 
6:      $S_{i,j} = \log \sum_{i \leq r < j} e^{C_{i,r} + C_{j,r+1}}$ 
7:      $C_{i,j} = \log \sum_{i < r \leq j} e^{I_{i,r} + C_{r,j}}$ 
8:   end for  $\triangleright$  refer to Figure 3
9: return  $C_{0,n} \equiv \log Z$ 

```

product, and naturally obtain $\log Z(\mathbf{x})$ in the same polynomial time complexity. However, it is not enough to solely perform the inside algorithm for non-neural parsing, due to the inapplicability of the automatic differentiation mechanism. In order to obtain marginal probabilities and then feature weight gradients, we have to realize the more sophisticated outside algorithm, which is usually at least twice slower than the inside algorithm. This may be the major reason for the less popularity of TreeCRF (vs. max-margin training) before the DL era.

As far as we know, all previous works on neural TreeCRF parsing explicitly implement the inside-outside algorithm for gradient computation (Zhang et al., 2019; Jiang et al., 2018). To improve efficiency, computation is transferred from GPUs to CPUs with Cython programming.

This work shows that the inside algorithm can be effectively batchified to fully utilize the power of GPUs. Figure 3 and Algorithm 1 together illustrate the batchified version of the second-order inside algorithm, which is a direct extension of the second-order Eisner algorithm in McDonald and Pereira (2006) by replacing max product with sum product. We omit the generations of incomplete, complete, and sibling spans in the opposite direction from j to i for brevity.

Basically, we first pack the scores of same-width spans at different positions (i, j) for all B sentences in the data batch into large tensors. Then we can do computation and aggregation simultaneously on GPUs via efficient large tensor operation.

Similarly, we also batchify the decoding algorithm. Due to space limitation, we omit the details.

It is noteworthy that the techniques described here are also applicable to other grammar formulations such as CKY-style constituency parsing (Finkel et al., 2008; Drozdov et al., 2019).

3.3 Outside via Back-propagation

Eisner (2016) proposes a theoretical proof on the equivalence between the back-propagation mechanism and the outside algorithm in the case of constituency (phrase-structure) parsing. This work empirically verifies this equivalence for dependency parsing.

Moreover, we also find that marginal probabilities $p(i \rightarrow j \mid \mathbf{x})$ directly correspond to gradients after back-propagation with $\log Z(\mathbf{x})$ as the loss:

$$\frac{\partial \log Z}{\partial s(i, j)} = \sum_{\mathbf{y}: (i, j) \in \mathbf{y}} p(\mathbf{y} \mid \mathbf{x}) = p(i \rightarrow j \mid \mathbf{x}) \quad (11)$$

which can be easily proved. For TreeCRF parsers, we perform MBR decoding (Smith and Smith, 2007) by replacing scores with marginal probabilities in the decoding algorithm, leading to a slight but consistent accuracy increase.

3.4 Handling Partial Annotation

As an attractive research direction, studies show that it is more effective to construct or even collect partially labeled data (Nivre et al., 2014; Hwa, 1999; Pereira and Schabes, 1992), where a sentence may correspond to a partial tree $|\mathbf{y}^p| < n$ in the case of dependency parsing. Partial annotation can be very powerful when combined with active learning, because annotation cost can be greatly reduced if annotators only need to annotate sub-structures that are difficult for models. Li et al. (2016) present a detailed survey on this topic. Moreover, Peng et al. (2019) recently released a partially labeled multi-domain Chinese dependency treebank based on this idea.

Then, the question is how to train models on partially labeled data. Li et al. (2016) propose to extend TreeCRF for this purpose and obtain promising results in the case of non-neural dependency parsing. This work applies their approach to the neural biaffine parser. We are particularly concerned at the influence of structural learning and high-order modeling on the utilization of partially labeled training data.

For the basic biaffine parser based on first-order local training, it seems the only choice is omitting losses of unannotated words. In contrast, tree constraints allow annotated dependencies to influence the probability distributions of unannotated words, and high-order modeling further helps by promoting inter-token interaction. Therefore, both

structural learning and high-order modeling are intuitively very beneficial.

Under partial annotation, we follow Li et al. (2016) and define the training loss as:

$$\begin{aligned} L(\mathbf{x}, \mathbf{y}^p) &= -\log \sum_{\mathbf{y} \in \mathcal{Y}(\mathbf{x}); \mathbf{y} \supseteq \mathbf{y}^p} p(\mathbf{y} \mid \mathbf{x}) \\ Z(\mathbf{x}, \mathbf{y}^p) &\equiv \sum_{\mathbf{y} \in \mathcal{Y}(\mathbf{x}); \mathbf{y} \supseteq \mathbf{y}^p} e^{s(\mathbf{x}, \mathbf{y})} \\ &= -\log \frac{Z(\mathbf{x}, \mathbf{y}^p)}{Z(\mathbf{x})} \end{aligned} \quad (12)$$

where $Z(\mathbf{x}, \mathbf{y}^p)$ only considers all legal trees that are compatible with the given partial tree and can also be efficiently computed like $Z(\mathbf{x})$.

4 Experiments

Data. We conduct experiments and analysis on 27 datasets from 13 languages, including two widely used datasets: the English Penn Treebank (PTB) data with Stanford dependencies (Chen and Manning, 2014), and the Chinese data at the CoNLL09 shared task (Hajič et al., 2009).

We also adopt the Chinese dataset released at the NLPCC19 cross-domain dependency parsing shared task (Peng et al., 2019), containing one source domain and three target domains. For simplicity, we directly merge the train/dev/test data of the four domains into larger ones respectively. One characteristic of the data is that most sentences are partially annotated based on active learning.

Finally, we conduct experiments on Universal Dependencies (UD) v2.2 and v2.3 following Ji et al. (2019) and Zhang et al. (2019) respectively. We adopt the 300d multilingual pretrained word embeddings used in Zeman et al. (2018) and take the CharLSTM representations as input. For UD2.2, to compare with Ji et al. (2019), we follow the raw text setting of the CoNLL18 shared task (Zeman et al., 2018), and directly use their sentence segmentation and tokenization results. For UD2.3, we also report the results of using gold-standard POS tags to compare with Zhang et al. (2019).

Evaluation metrics. We use unlabeled and labeled attachment score (UAS/LAS) as the main metrics. Punctuations are omitted for PTB. For the partially labeled NLPCC19 data, we adopt the official evaluation script, which simply omits the words without gold-standard heads to accommodate partial annotation. We adopt Dan Bikel’s randomized parsing evaluation comparator for significance test.

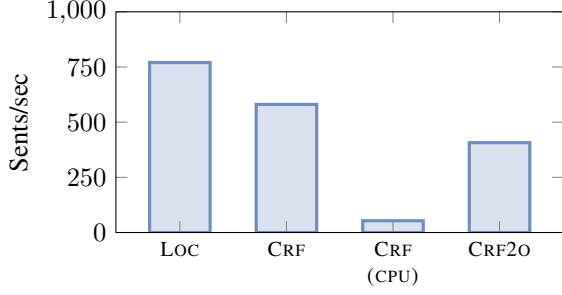


Figure 4: Parsing speed comparison on PTB-test.

Parameter settings. We directly adopt most parameter settings of [Dozat and Manning \(2017\)](#), including dropout and initialization strategies. For CharLSTM, the dimension of input char embeddings is 50, and the dimension of output vector is 100, following [Lample et al. \(2016\)](#). For the second-order model, we set the dimensions of $\mathbf{r}_i^{h'/s/m'}$ to 100, and find little accuracy improvement when increasing to 300. We trained each model for at most 1,000 iterations, and stop training if the peak performance on the dev data does not increase in 100 consecutive epochs.

Models. LOC uses local cross-entropy training loss and employs the Eisner algorithm for finding the optimal projective tree. CRF and CRF2O denote the first-order and second-order TreeCRF model respectively. LOC_{MST} denotes the basic local model that directly produces non-projective tree based on the MST decoding algorithm of [Dozat and Manning \(2017\)](#).

4.1 Efficiency Comparison

Figure 4 compares the parsing speed of different models on PTB-test. For a fair comparison, we run all models on the same machine with Intel Xeon CPU (E5-2650v4, 2.20GHz) and GeForce GTX 1080 Ti GPU. “CRF (CPU)” refers to the model that explicitly performs the inside-outside algorithm using Cython on CPUs. Multi-threading is employed since sentences are mutually independent. However, we find that using more than 4 threads does not further improve the speed.

We can see that the efficiency of TreeCRF is greatly improved by batchifying the inside algorithm and implicitly realizing the outside algorithm by back-propagation on GPUs. For the first-order CRF model, our implementation can parse about 500 sentences per second, over 10 times faster than the multi-thread “CRF (CPU)”. For the second-order CRF2O, our parser achieves the speed of 400

	Dev		Test	
	UAS	LAS	UAS	LAS
PTB				
Biaffine17	-	-	95.74	94.08
F&K19	-	-	-	91.59
Li19	95.76	93.97	95.93	94.19
Ji19	95.88	93.94	95.97	94.31
Zhang19	-	-	-	93.96
LOC	95.82	93.99	96.08	94.47
CRF w/o MBR	95.74	93.96	96.04	94.34
CRF	95.76	93.99	96.02	94.33
CRF2O w/o MBR	95.92	94.16	96.14	94.49
CRF2O	95.90	94.12	96.11	94.46
CoNLL09				
Biaffine17	-	-	88.90	85.38
Li19	88.68	85.47	88.77	85.58
LOC	89.07	86.10	89.15	85.98
CRF w/o MBR	89.04	86.04	89.14	86.06
CRF	89.12	86.12	89.28	86.18 [†]
CRF2O w/o MBR	89.29	86.24	89.49	86.39
CRF2O	89.44	86.37	89.63[‡]	86.52[‡]
NLPCC19				
LOC	77.01	71.14	76.92	71.04
CRF w/o MBR	77.40	71.65	77.17	71.58
CRF	77.34	71.62	77.53 [‡]	71.89 [‡]
CRF2O w/o MBR	77.58	71.92	77.89	72.25
CRF2O	78.08	72.32	78.02[‡]	72.33[‡]

Table 1: Main results. We perform significance test against LOC on the test data, where “[†]” means $p < 0.05$ and “[‡]” means $p < 0.005$. Biaffine17: [Dozat and Manning \(2017\)](#); F&K19: [Falenska and Kuhn \(2019\)](#); Li19: [Li et al. \(2019\)](#); Ji19: [Ji et al. \(2019\)](#); Zhang19: [Zhang et al. \(2019\)](#).

sentences per second, which is able to meet the requirements of a real-time system. More discussions on efficiency are presented in Appendix A.

4.2 Main Results

Table 1 lists the main results on the dev and test data. The trends on dev and test are mostly consistent. For a fair comparison with previous works, we only consider those without using extra resources such as ELMo ([Peters et al., 2018](#)) and BERT ([Devlin et al., 2019](#)). We can see that our baseline LOC achieves the best performance on both PTB and CoNLL09.

On PTB, both CRF and CRF2O fail to improve

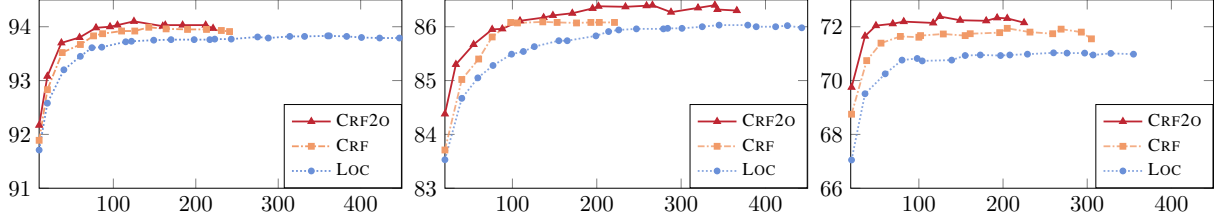


Figure 5: Convergence curves (LAS vs. training epochs) on dev data of PTB, CoNLL09, and NLPCC19.

	P	SIB R	F	UCM	LCM
PTB					
LOC	91.16	90.80	90.98	61.59	50.66
CRF	91.24	90.92	91.08	61.92	50.33
CRF2O	91.56	91.11	91.33	63.08	50.99
CoNLL09					
LOC	79.20	79.02	79.11	40.10	28.91
CRF	79.17	79.55	79.36	40.61	29.38
CRF2O	81.00	80.63	80.82	42.53	30.09

Table 2: Sub- and full-tree performance on test data.

the parsing accuracy further, probably because the performance is already very high. However, as shown by further analysis in Section 4.3, the positive effect is actually introduced by structural learning and high-order modeling.

On CoNLL09, CRF significantly outperforms LOC, and CRF2O can further improve the performance.

On the partially annotated NLPCC19 data, CRF outperforms LOC by a very large margin, indicating the usefulness of structural learning in the scenario of partial annotation. CRF2O further improves the parsing performance by explicitly modeling second-order subtree features. These results confirm our intuitions discussed in Section 3.4. Please note that the parsing accuracy looks very low because the partially annotated tokens are usually difficult for models.

4.3 Analysis

Impact of MBR decoding. For CRF and CRF2O, we by default to perform MBR decoding, which employs the Eisner algorithm over marginal probabilities (Smith and Smith, 2007) to find the best tree.

$$\mathbf{y}^* = \arg \max_{\mathbf{y}} \left[\sum_{i \rightarrow j \in \mathbf{y}} p(i \rightarrow j | \mathbf{x}) \right] \quad (13)$$

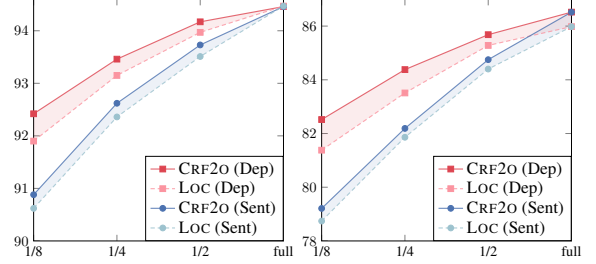


Figure 6: LAS on PTB (left) and CoNLL09-test (right) regarding the amount of training data (dependencies vs. sentences).

Table 1 reports the results of directly finding 1-best trees according to dependency scores. Except for PTB, probably due to the high accuracy already, MBR decoding brings small yet consistent improvements for both CRF and CRF2O.

Convergence behavior. Figure 5 compares the convergence curves. For clarity, we plot one data point corresponding to the peak LAS every 20 epochs. We can clearly see that both structural learning and high-order modeling consistently improve the model. CRF2O achieves steadily higher accuracy and converges much faster than the basic LOC.

Performance at sub- and full-tree levels. Beyond the dependency-wise accuracy (UAS/LAS), we would like to evaluate the models regarding performance at sub-tree and full-tree levels. Table 2 shows the results. We skip the partially labeled NLPCC19 data. UCM means unlabeled complete matching rate, i.e., the percent of sentences obtaining whole correct skeletal trees, while LCM further requires that all labels are also correct.

For SIB, we evaluate the model regarding unlabeled adjacent-sibling subtrees (system outputs vs. gold-standard references). According to Equation 6, (i, k, j) is an adjacent-sibling subtree, if and only if w_k and w_j are both children of w_i at the same side, and there are no other children of w_i between them. Given two trees, we can col-

	bg	ca	cs	de	en	es	fr	it	nl	no	ro	ru	Avg.
UD2.2													
LOC _{MST}	90.32	90.77	90.80	80.48	86.87	90.63	87.66	91.85	87.81	90.51	86.39	93.11	88.93
LOC	90.37	90.97	90.87	80.23	86.93	90.54	87.72	91.90	87.90	90.79	86.41	93.10	88.98
CRF	90.50	91.16 [†]	91.16	80.44	86.84	90.72 [†]	87.92	91.89	88.74 [‡]	90.61	86.42	93.04	89.12 [‡]
CRF2O	90.66	91.39[‡]	91.01	80.52	87.07	91.03[‡]	88.36[†]	91.83	89.04[‡]	90.99	86.52	93.33[‡]	89.31[‡]
using raw text													
Ji19	88.28	89.90	89.85	77.09	81.16	88.93	83.73	88.91	84.82	86.33	84.44	86.62	85.83
CRF2O	89.72	91.27	90.94	78.26	82.88	90.79	86.33	91.02	87.92	90.17	85.71	92.49	88.13
UD2.3													
LOC _{MST}	90.56	91.03	91.98	81.59	86.83	90.64	88.23	91.67	88.20	90.63	86.51	93.03	89.23
LOC	90.57	91.10	91.85	81.68	86.54	90.47	88.40	91.53	88.18	90.65	86.31	92.91	89.19
CRF	90.52	91.19	92.02	81.43	86.88 [†]	90.76 [†]	88.75	91.76	88.08	90.79	86.54	93.16 [‡]	89.32 [‡]
CRF2O	90.76	91.12	92.15[‡]	81.94	86.93[†]	90.81[‡]	88.83[†]	92.34[‡]	88.21[†]	90.78	86.62	93.22[‡]	89.48[‡]
using gold POS tags													
Zhang19	90.15	91.39	91.10	83.39	88.52	90.84	88.59	92.49	88.37	92.82	84.89	93.11	89.85
CRF2O	91.32	92.57	92.66	84.56	88.98	91.88	89.83	92.94	89.85	93.26	87.39	93.86	90.76

Table 3: LAS on UD2.2 and UD2.3 test datasets. Again, [†] and [‡] means significance level at $p < 0.05$ and $p < 0.005$ respectively against the LOC parser.

lect all adjacent-sibling subtrees and compose two sets of triples. Then we evaluate the P/R/F values. Please note that it is impossible to evaluate SIB for partially annotated references.

We can clearly see that by modeling adjacent-sibling subtree scores, the SIB performance obtains larger improvement than both CRF and LOC, and this further contributes to the large improvement on full-tree matching rates (UCM/LCM).

Capability to learn from partial trees. To better understand why CRF2O performs very well on partially annotated NLPCC19, we design more comparative experiments by retaining either a proportion of random training sentences (full trees) or a proportion of random dependencies for each sentence (partial trees). Figure 6 shows the results.

We can see that the performance gap is quite steady when we gradually reduce the number of training sentences. In contrast, the gap clearly becomes larger when each training sentence has less annotated dependencies. This shows that CRF2O is superior to the basic LOC in utilizing partial annotated data for model training.

4.4 Results on Universal Dependencies

Table 3 compares different models on UD datasets, which contain a lot of non-projective trees. We adopt the pseudo-projective approach (Nivre and

Nilsson, 2005) for handling the ubiquitous non-projective trees of most languages. Basically, the idea is to transform non-projective trees into projective ones using more complex labels for post-processing recovery.

We can see that for the basic local parsers, the direct non-projective LOC_{MST} and the pseudo-projective LOC achieve very similar performance.

More importantly, both CRF and CRF2O produce consistent improvements over the baseline in many languages. On both UD2.2 and UD2.3, Our proposed CRF2O model achieves the highest accuracy for 10 languages among 12, and obtains significant improvement in more than 5 languages. Overall, the averaged improvement is 0.33 and 0.29 on UD2.2 and UD2.3 respectively, which is also significant at $p < 0.005$.

On average, our CRF2O parser outperforms Ji et al. (2019) by 2.30 on UD2.2 raw texts following CoNLL-2018 shared task setting, and Zhang et al. (2019) by 0.91 on UD2.3 data with gold POS tags. It is noteworthy that the German (de) result is kindly provided by Tao Ji after rerunning their parser with predicted XPOS tags, since their reported result in Ji et al. (2019) accidentally used gold-standard sentence segmentation, tokenization, and XPOS tags. Our CRF2O parser achieves an average LAS of 87.64 using their XPOS tags.

5 Related Works

Batchification has been widely used in linear-chain CRF, but is rather complicated for tree structures. [Eisner \(2016\)](#) presents a theoretical proof on the equivalence of outside and back-propagation for constituent tree parsing, and also briefly discusses other formalisms such as dependency grammar. Unfortunately, we were unaware of Eisner’s great work until we were surveying the literature for paper writing. As an empirical study, we believe this work is valuable and makes it practical to deploy TreeCRF models in real-life systems.

[Falenska and Kuhn \(2019\)](#) present a nice analytical work on dependency parsing, similar to [Gaddy et al. \(2018\)](#) on constituency parsing. By extending the first-order graph-based parser of [Kiperwasser and Goldberg \(2016\)](#) into second-order, they try to find out how much structural context is implicitly captured by the BiLSTM encoder. They concatenate three BiLSTM output vectors (i, k, j) for scoring adjacent-sibling subtrees, and adopt max-margin loss and the second-order Eisner decoding algorithm ([McDonald and Pereira, 2006](#)). Based on their negative results and analysis, they draw the conclusion that high-order modeling is redundant because BiLSTM can implicitly and effectively encode enough structural context. They also present a nice survey on the relationship between RNNs and syntax. In this work, we use a much stronger basic parser and observe more significant UAS/LAS improvement than theirs. Particularly, we present an in-depth analysis showing that explicitly high-order modeling certainly helps the parsing model and thus is complementary to the BiLSTM encoder.

[Ji et al. \(2019\)](#) employ graph neural networks to incorporate high-order structural information into the biaffine parser implicitly. They add a three-layer graph attention network (GAT) component ([Veličković et al., 2018](#)) between the MLP and Bi-affine layers. The first GAT layer takes \mathbf{r}_i^h and \mathbf{r}_i^m from MLPs as inputs and produces new representation \mathbf{r}_i^{h1} and \mathbf{r}_i^{m1} by aggregating neighboring nodes. Similarly, the second GAT layer operates on \mathbf{r}_i^{h1} and \mathbf{r}_i^{m1} , and produces \mathbf{r}_i^{h2} and \mathbf{r}_i^{m2} . In this way, a node gradually collects multi-hop high-order information as global evidence for scoring single dependencies. They follow the original local head-selection training loss. In contrast, this work adopts global TreeCRF loss and explicitly incorporates high-order scores into the biaffine parser.

[Zhang et al. \(2019\)](#) investigate the usefulness of structural training for the first-order biaffine parser. They compare the performance of local head-selection loss, global max-margin loss, and TreeCRF loss on multilingual datasets. They show that TreeCRF loss is overall slightly superior to max-margin loss, and LAS improvement from structural learning is modest but significant for some languages. They also show that structural learning (especially TreeCRF) substantially improves sentence-level complete matching rate, which is consistent with our findings. Moreover, they explicitly compute the inside and outside algorithms on CPUs via Cython programming. In contrast, this work proposes an efficient second-order TreeCRF extension to the biaffine parser, and presents much more in-depth analysis to show the effect of both structural learning and high-order modeling.

6 Conclusions

This paper for the first time presents second-order TreeCRF for neural dependency parsing using tri-affine for explicitly scoring second-order subtrees. We propose to batchify the inside algorithm to accommodate GPUs. We also empirically verify that the complex outside algorithm can be implicitly performed via efficient back-propagation, which naturally produces gradients and marginal probabilities. We conduct experiments and detailed analysis on 27 datasets from 13 languages, and find that structural learning and high-order modeling can further enhance the state-of-the-art biaffine parser in various aspects: 1) better convergence behavior; 2) higher performance on sub- and full-tree levels; 3) better utilization of partially annotated data.

Acknowledgments

The authors would like to thank: 1) the anonymous reviewers for the helpful comments, 2) Wenliang Chen for helpful discussions on high-order neural dependency parsing, 3) Tao Ji for kindly sharing the data and giving beneficial suggestions for the experiments on CoNLL18 datasets, 4) Wei Jiang, Yahui Liu, Haoping Yang, Houquan Zhou and Mingyue Zhou for their help in paper writing and polishing. This work was supported by National Natural Science Foundation of China (Grant No. 61876116, 61525205, 61936010) and a Project Funded by the Priority Academic Program Development (PAPD) of Jiangsu Higher Education Institutions.

References

- Jiong Cai, Yong Jiang, and Kewei Tu. 2017. CRF autoencoder for unsupervised dependency parsing. In *Proceedings of EMNLP*, pages 1638–1643.
- Xavier Carreras. 2007. Experiments with a higher-order projective dependency parser. In *Proceedings of EMNLP*, pages 957–961.
- Danqi Chen and Christopher Manning. 2014. A fast and accurate dependency parser using neural networks. In *Proceedings of EMNLP*, pages 740–750.
- Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL*, pages 4171–4186.
- Timothy Dozat and Christopher D. Manning. 2017. Deep biaffine attention for neural dependency parsing. In *Proceedings of ICLR*.
- Timothy Dozat, Peng Qi, and Christopher D. Manning. 2017. Stanford’s graph-based neural dependency parser at the CoNLL 2017 shared task. In *Proceedings of CoNLL*, pages 20–30.
- Andrew Drozdov, Patrick Verga, Mohit Yadav, Mohit Iyyer, and Andrew McCallum. 2019. Unsupervised latent tree induction with deep inside-outside recursive auto-encoders. In *Proceedings of NAACL*, pages 1129–1141.
- Jason Eisner. 2000. Bilexical grammars and their cubic-time parsing algorithms. In *Advances in Probabilistic and Other Parsing Technologies*.
- Jason Eisner. 2016. Inside-outside and forward-backward algorithms are just backprop (tutorial paper). In *Proceedings of WS*, pages 1–17.
- Agnieszka Falenska and Jonas Kuhn. 2019. The (non-)utility of structural features in BiLSTM-based dependency parsers. In *Proceedings of ACL*, pages 117–128.
- Jenny Rose Finkel, Alex Kleeman, and Christopher D. Manning. 2008. Efficient, feature-based, conditional random field parsing. In *Proceedings of ACL*, pages 959–967.
- David Gaddy, Mitchell Stern, and Dan Klein. 2018. What’s going on in neural constituency parsers? an analysis. In *Proceedings of NAACL*, pages 999–1010.
- Jan Hajič, Massimiliano Ciaramita, Richard Johansson, Daisuke Kawahara, Maria Antònia Martí, Lluís Màrquez, Adam Meyers, Joakim Nivre, Sebastian Padó, Jan Štěpánek, Pavel Straňák, Mihai Surdeanu, Nianwen Xue, and Yi Zhang. 2009. The CoNLL-2009 shared task: Syntactic and semantic dependencies in multiple languages. In *Proceedings of CoNLL*, pages 1–18.
- Rebecca Hwa. 1999. Supervised grammar induction using training data with limited constituent information. In *Proceedings of ACL*, pages 73–79.
- Tao Ji, Yuanbin Wu, and Man Lan. 2019. Graph-based dependency parsing with graph neural networks. In *Proceedings of ACL*, pages 2475–2485.
- Xinzhou Jiang, Zhenghua Li, Bo Zhang, Min Zhang, Sheng Li, and Luo Si. 2018. Supervised treebank conversion: Data and approaches. In *Proceedings of ACL*, pages 2706–2716.
- Eliyahu Kiperwasser and Yoav Goldberg. 2016. Simple and accurate dependency parsing using bidirectional LSTM feature representations. *Transactions of ACL*, pages 313–327.
- Terry Koo and Michael Collins. 2010. Efficient third-order dependency parsers. In *Proceedings of ACL*, pages 1–11.
- Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. 2016. Neural architectures for named entity recognition. In *Proceedings of NAACL*, pages 2475–2485.
- Phong Le and Willem Zuidema. 2014. The inside-outside recursive neural network model for dependency parsing. In *Proceedings of EMNLP*, pages 729–739.
- Ying Li, Zhenghua Li, Min Zhang, Rui Wang, Sheng Li, and Luo Si. 2019. Self-attentive biaffine dependency parsing. In *Proceedings of IJCAI*, pages 5067–5073.
- Zhenghua Li, Min Zhang, Yue Zhang, Zhanyi Liu, Wenliang Chen, Hua Wu, and Haifeng Wang. 2016. Active learning for dependency parsing with partial annotation. In *Proceedings of ACL*, pages 344–354.
- Xuezhe Ma and Hai Zhao. 2012. Fourth-order dependency parsing. In *Proceedings of COLING*, pages 785–796.
- Ryan McDonald, Koby Crammer, and Fernando Pereira. 2005a. Online large-margin training of dependency parsers. In *Proceedings of ACL*, pages 91–98.
- Ryan McDonald and Fernando Pereira. 2006. Online learning of approximate dependency parsing algorithms. In *Proceedings of EACL*, pages 81–88.
- Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. 2005b. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of EMNLP*, pages 523–530.
- Joakim Nivre, Yoav Goldberg, and Ryan McDonald. 2014. Squibs: Constrained arc-eager dependency parsing. *CL*, pages 249–257.

- Joakim Nivre, Marie-Catherine de Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajič, Christopher D. Manning, Ryan McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira, Reut Tsarfaty, and Daniel Zeman. 2016. Universal dependencies v1: A multilingual treebank collection. In *Proceedings of LREC*, pages 1659–1666.
- Joakim Nivre and Jens Nilsson. 2005. Pseudo-projective dependency parsing. In *Proceedings of ACL*, pages 99–106.
- Wenzhe Pei, Tao Ge, and Baobao Chang. 2015. An effective neural network model for graph-based dependency parsing. In *Proceedings of ACL-IJCNLP*, pages 313–322.
- Xue Peng, Zhenghua Li, Min Zhang, Rui Wang, Yue Zhang, and Luo Si. 2019. Overview of the nlpcc 2019 shared task: cross-domain dependency parsing. In *Proceedings of NLPCC*, pages 760–771.
- Fernando Pereira and Yves Schabes. 1992. Inside-outside reestimation from partially bracketed corpora. In *Proceedings of ACL*, pages 128–135.
- Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In *Proceedings of NAACL*, pages 2227–2237.
- David A. Smith and Noah A. Smith. 2007. Probabilistic models of nonprojective dependency trees. In *Proceedings of EMNLP*, pages 132–140.
- Robert Endre Tarjan. 1977. Finding optimum branchings. *Networks*, pages 25–35.
- Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph attention networks. In *Proceedings of ICLR*.
- Wenhui Wang and Baobao Chang. 2016. Graph-based dependency parsing with bidirectional LSTM. In *Proceedings of ACL*, pages 2475–2485.
- Xinyu Wang, Jingxian Huang, and Kewei Tu. 2019. Second-order semantic dependency parsing with end-to-end neural networks. In *Proceedings of ACL*, pages 4609–4618.
- Daniel Zeman, Jan Hajič, Martin Popel, Martin Potthast, Milan Straka, Filip Ginter, Joakim Nivre, and Slav Petrov. 2018. CoNLL 2018 shared task: Multilingual parsing from raw text to universal dependencies. In *Proceedings of CoNLL*, pages 1–21.
- Xingxing Zhang, Jianpeng Cheng, and Mirella Lapata. 2017. Dependency parsing as head selection. In *Proceedings of EACL*, pages 665–676.
- Zhisong Zhang, Xuezhe Ma, and Eduard Hovy. 2019. An empirical investigation of structured output modeling for graph-based neural dependency parsing. In *Proceedings of ACL*, pages 5592–5598.

A More on Efficiency

Training speed. During training, we greedily find the 1-best head for each word without tree constraints. Therefore, the processing speed is faster than the evaluation phase. Specifically, for LOC, CRF and CRF2O, the average one-iteration training time is about 1min, 2.5min and 3.5min on PTB. In other words, the parser consumes about 700/300/200 sentences per second.

MST decoding. As Dozat et al. (2017) pointed out, they adopted an ad-hoc approximate algorithm which does not guarantee to produce the highest-scoring tree, rather than the ChuLiu/Edmonds algorithm for MST decoding. The time complexity of the ChuLiu/Edmonds algorithm is $O(n^2)$ under the optimized implementation of Tarjan (1977). Please see the discussion of McDonald et al. (2005b) for details.

For LOC_{MST}, we directly borrow the MST decoding approach in the original parser of Dozat and Manning (2017). LOC_{MST} achieves 94.43 LAS on PTB-test (inferior to 94.47 of LOC, see Table 1), and its parsing speed is over 1000 sentences per second.

Faster decoding strategy. Inspired by the idea of ChuLiu/Edmonds algorithm, we can further improve the efficiency of the CRF parsing models by avoiding the Eisner decoding for some sentences. The idea is that if by greedily assigning a local max-scoring head word to each word, we can already obtain a legal projective tree, then we omit the decoding process for the sentence. We can judge whether an output is a legal tree (single root and no cycles) using the Tarjan algorithm in $O(n)$ time complexity. Further, we can judge whether a tree is a projective tree also in a straightforward way very efficiently. In fact, we find that more than 99% sentences directly obtain legal projective trees on PTB-test by such greedy assignment on marginal probabilities first. We only need to run the decoding algorithm for the left sentences.