Constituency parsing and nested named entity recognition (NER) are similar tasks since they both aim to predict a collection of nested and non-crossing spans. In this work, we cast nested NER to constituency parsing and propose a novel pointing mechanism for bottom-up parsing to tackle both tasks. The key idea is based on the observation that if we traverse a constituency tree in post-order, i.e., visiting a parent after its children, then two consecutively visited spans would share a boundary. Our model tracks the shared boundaries and predicts the next boundary at each step by leveraging a pointer network. As a result, it needs only linear steps to parse and thus is efficient. It also maintains a parsing configuration for structural consistency, i.e., always outputing valid trees. Experimentally, our model achieves the state-of-the-art performance on PTB among all BERT-based models (96.01 F1 score) and competitive performance on CTB7 in constituency parsing; and it also achieves strong performance on three benchmark datasets of nested NER: ACE2004, ACE2005, and GENIA.

1 Introduction
Constituency parsing is an important task in natural language processing, having many applications in downstream tasks, such as semantic role labeling (Fei et al., 2021), opinion mining (Xia et al., 2021), among others. Named entity recognition (NER) is a fundamental task in information extraction and nested NER has been receiving increasing attention due to its broader applications (Byrne, 2007).

Constituency parsing and nested NER are similar tasks since they both aim to predict a collection of nested and non-crossing spans (i.e., if two spans overlap, one must be a subspan of the other). Fig.1 shows example span representations of both tasks. The difference between the two tasks is that the collection of spans form a connected tree in constituency parsing, whereas they form several tree fragments in nested NER. However, we can add a node that spans the whole sentence to connect all tree fragments in nested NER to form a tree. Because of the similarity, there are some previous studies adapting methods from the constituency parsing literature to tackle nested NER (Finkel and Manning, 2009; Wang et al., 2018; Fu et al., 2021).

In this work, we focus on constituency parsing, but our proposed method tackles nested NER as well.

The two main paradigms in constituency pars-
ing are span-based and transition-based methods. Span-based methods (Stern et al., 2017; Kitaev and Klein, 2018; Zhang et al., 2020; Xin et al., 2021, *inter alia*) decompose the score of a constituency tree into the scores of constituent spans and use chart-based algorithms for inference. Built upon powerful neural encoders, they have obtained state-of-the-art results. However, they suffer from the high inference time complexity of exact algorithms or error propagation of top-down approximate algorithms. In contrast, transition-based methods (Dyer et al., 2016; Cross and Huang, 2016; Liu and Zhang, 2017, *inter alia*) conduct a series of local actions (e.g., shift and reduce) to build the final parse in linear steps, so they enjoy lower parsing time complexities. However, they suffer from the error propagation and exposure bias problems.

Recently, Nguyen et al. (2021a) propose a sequence-to-sequence (seq2seq) model with pointer networks (Vinyals et al., 2015a). They cast constituency parsing to a top-down splitting problem. First, they use neural encoders to obtain span representations, similar to span-based methods. Then they feed input parent span representations into the neural decoder recursively following the order shown in Fig. 2(a)\(^2\)—which amounts to pre-order traversal—to output a series of splitting points (i.e., boundaries) via pointer networks, so that each parent span is split into two child spans. Notably, Nguyen et al. (2020) propose a similar top-down pointing mechanism, but they design a chart-based parsing algorithm instead of adopting seq2seq modeling, and has been shown underperforming Nguyen et al. (2021a). Thanks to seq2seq modeling, Nguyen et al. (2021a)’s model achieves a competitive parsing performance with a lower parsing complexity compared with span-based methods.

However, their model has two main limitations. First, when generating each constituent, its subtree features cannot be exploited since its subsppans have not been realized yet (Liu and Zhang, 2017). Thus it is difficult for the model to predict the splitting point of a long span due to a lack of its subtree information, which exacerbates the error propagation problem and undermines the parsing performance. Second, since each parent span can only be split into two, their parsing algorithm can only output binary trees, thus needing binarization.

\(^2\)Slightly different from the figure, they do not feed spans of length 1 into the decoder for obvious reasons.

In this work, we devise a novel pointing mechanism for bottom-up parsing using (almost) the same seq2seq backbone as Nguyen et al. (2021a). Our model is able to overcome the two aforementioned limitations of Nguyen et al. (2021a). The main idea is based on the observation that if we traverse a constituency tree in post-order (i.e., visiting a parent after its children), two consecutively visited constituent spans would share a boundary. Fig. 2(b) shows an example: the right boundary of \(\text{1}\) is also the left boundary of \(\text{2}\) and the right boundary of \(\text{5}\) is also the right boundary of \(\text{6}\). Based on this observation, we propose to use a cursor to track the shared boundary boundaries and at each step, leverage a pointer network to predict the next boundary for generating the next constituent span and update the cursor to the right boundary of the new span. Our model generates one span at each step, thus needing only linear steps to parse a sentence, which is efficient. Besides, our model can leverage rich subtree features encoded in the neural decoder to generate parent constituent spans, which is especially helpful in predicting long spans. Finally, our model can output n-ary trees, enabling direct modeling of the original non-binary parse tree structures in treebanks and eliminating the need for binarization.
We conduct experiments on the benchmarking PTB and CTB for constituency parsing. On PTB, we achieve the state-of-the-art performance (96.01 F1 score) among all BERT-based models. On CTB, we achieve competitive performance. We also apply our method to nested NER and conduct experiments on three benchmark datasets: ACE2004, ACE2005, and GENIA. Our method achieves comparable performance to many tailored methods of nested NER, beating previous parsing-based methods. Our contributions can be summarized as the following:

- We propose a novel pointing mechanism for bottom-up n-ary tree parsing in linear steps.
- Our model achieves the state-of-the-art result on PTB in constituency parsing. We further show its application in nested NER where it achieves competitive results.

2 Methods

2.1 Preprocessing

It is known that constituency parsing can be regarded as a top-down splitting problem where parent spans are recursively split into pairs of subspans (Stern et al., 2017; Shen et al., 2018; Nguyen et al., 2020, 2021a). However, this formulation can output binary trees. We make an extension to cast constituency parsing as top-down segmentation, i.e., parent spans are segmented into $\geq 2$ subspans recursively, for the sake of outputting n-ary trees. To this end, we add some $\emptyset$ spans (we do not allow two adjacent $\emptyset$ spans to eliminate ambiguities) so that each span is either a bottommost span or can be segmented by its subspans. For instance, in Fig 2, 3 is a bottom-most span, and 7 can be segmented by 2, 3 and 6. We always include the whole-sentence span in order to cast other tasks, e.g., nested NER, to constituency parsing. We also collapse unary chains to atomic labels in constituency parsing, e.g., $S$-$\triangleright$VP $\rightarrow$ S+VP.

2.2 Parsing configuration

A problem of seq2seq constituency parsers is how to maintain structural consistency, i.e., outputting valid trees. To solve this problem, our pointing system maintains a parsing configuration, which is a quadruple $(c, A, p, S)$ where:

- $c$: index of the cursor.
- $A$: set of indices of all candidate boundaries.
- $p$: the left boundary of the newly created span, which is needed to maintain $A$.
- $S$: set of generated spans.

We can see from Fig. 3 that in the beginning, the cursor $c$ lies at 0. At each step, $c$ points to another boundary $a$ from $A$ to form a span $(\min(c, a), \max(c, a))$. There are two cases:

- $c < a$: a new bottom-most span is generated.
- $a < c$: several consecutive spans are merged into a larger span. It is worthy to note that we can merge $\geq 2$ spans in a single step, which allows our model to perform n-ary tree parsing.

In the first case, the new bottom-most span can combine with the very previous span to form a larger span whose left boundary is $p$, so we push $p$ back to $A$ (except for the case that $p = \text{null}$). In the later case, the very previous span is a subspan of the new span and thus $p$ cannot be pushed back. In both cases, all indices $\min(c, a) \leq i < \max(c, a)$ are removed from $A$ due to the post-order generation restriction; $p$ is updated to $\min(c, a)$ and $c$ is updated to $\max(c, a)$. The process stops when the whole-sentence span is generated. Table 1 formalises this process.

Oracle. The oracle pointing representations shown in Fig.1 can be generated by running a post-order traversal of the tree (e.g., Fig.2) and for each traversed span, pointing the cursor from its boundary shared with the previous span to its other boundary. If we do not allow two consecutive $\emptyset$ spans, the oracle is unique under our pointing system (we give a proof in Appendix A.1 by contradiction).

2.3 Model

Given a sentence $w = w_1, \ldots, w_n$, we add $<\text{bos}>$ (beginning of sentence) as $w_0$ and $<\text{eos}>$ (end of sentence) as $w_{n+1}$. The oracle is \{ $q_i \rightarrow p_i$, $y_i$ $\}$ $i=1,\ldots,m$, where $y_i$ is the span label and we use $l_i = \min(q_i, p_i)$ and $r_i = \max(q_i, p_i)$ to denote the left and right boundary of the $i$-th span, respectively.

Encoder. We feed the sentence into BERT (Devlin et al., 2019) and for each word $w_i$, we use the last subtoken embedding of the last layer as its dense representations $x_i$. Then we feed $x_0, \ldots, x_{n+1}$ into a three-layer bidirectional
Initial configuration \((c, A, p, s) = (0, \{1, 2, \ldots, n\}, \text{null}, \emptyset)\)

Goal \((0, n) \in S\)

<table>
<thead>
<tr>
<th>Pointing action</th>
<th>Input</th>
<th>Output</th>
<th>Precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEFT-POINT-a</td>
<td>((c, A, p, S))</td>
<td>((c, A \setminus {a, \ldots, c-1}, a, S \cup {(a, c)}))</td>
<td>(0 \leq a &lt; c)</td>
</tr>
<tr>
<td>RIGHT-POINT-a</td>
<td>((c, A, p, S))</td>
<td>((a, A \cup {p} \setminus {c, \ldots, a-1}, c, S \cup {(c, a)}))</td>
<td>(c &lt; a \leq n)</td>
</tr>
</tbody>
</table>

Table 1: Description of the parsing configuration.

Figure 3: Demonstration of the generation process and the neural architecture. Black arrows point to candidate boundaries that are not selected in each step.

LSTM (Hochreiter and Schmidhuber, 1997) (BiLSTM) to obtain \(c_0, \ldots, c_{n+1}\), where \(c_i = [f_i; g_i]\) and \(f_i\) and \(g_i\) are the forward and backward hidden states of the last BiLSTM layer at position \(i\) respectively.

Boundary and span representation. We use fencepost representation (Cross and Huang, 2016; Stern et al., 2017) to encode the \(i\)-th boundary lying between \(x_i\) and \(x_{i+1}\):

\[ b_i = [f_i; g_{i+1}] \]

then we represent span \((i, j)\) as:

\[ h_{i,j} = \text{MLP}_{\text{span}}(b_j - b_i) \]

Decoder. We use a unidirectional one-layer LSTM network as the decoder:

\[ d_t = \text{LSTM}(d_{t-1}, h_{t-1}, r_{t-1}; E_{y_{t-1}}), t \geq 2 \]  \hspace{1cm} (1)

where \(d_t\) is the hidden state of the LSTM decoder at time step \(t\), \(E\) is the label embedding matrix, \(;\) is the concatenation operation. For the first step, we feed a randomly initialized trainable vector \(d_0\) and a special \(<\text{START}>\) embedding into the decoder to obtain \(d_1\).

Pointing score. We use a deep biaffine function (Dozat and Manning, 2017) to estimate the pointing score \(s_t^i\) of selecting the \(i\)-th boundary at time step \(t\):

\[ d'_t = \text{MLP}_{\text{cursor}}(d_t) \]

\[ b'_i = \text{MLP}_{\text{point}}(b_i) \]

\[ s_t^i = [b'_i; 1]^\top W_{\text{point}}d'_t \]

where \(\text{MLP}_{\text{cursor}}\) and \(\text{MLP}_{\text{point}}\) are multi-layer perceptrons (MLPs) that project decoder states and boundary representations into \(k\)-dimensional spaces, respectively; \(W_{\text{point}} \in \mathbb{R}^{(k+1) \times (k)}\).
Label score. For a newly predicted span, we feed the concatenation of the span representation and the decoder state into another MLP to calculate the label score $e^t$:

$$
H = \text{MLP}_{\text{label}}([d^t; b_{t_1} - b_{t_1}]) \\
 e^t = HE^T
$$

Note that we reuse the label embedding matrix from Eq. 1 to facilitate parameter sharing.

Training objective. The training loss is decomposed into the pointing loss and the labeling loss:

$$
L = L_{\text{pointing}} + L_{\text{labeling}}
$$

$$
L_{\text{pointing}} = - \sum_{t=1}^{m} \log \frac{\exp{s^t_{p_t}}}{\sum_{j=0}^{n} \exp{s^t_j}}
$$

$$
L_{\text{labeling}} = - \sum_{t=1}^{m} \log \frac{\exp{e^t_{y_t}}}{\sum_{j=1}^{|L|} \exp{e^t_j}}
$$

where $|L|$ is the number of labels. Note that in the pointing loss we normalize over all boundaries instead of only accessible boundaries, because we find it performs better in our preliminary experiments.

Parsing. Our model follows the description in the previous subsection for parsing. For each time step $t$, it selects the highest-scoring accessible boundary to generate the span, then selects the highest-scoring label of the generated span, and updates the parsing configuration (Table 1).

3 Experiment setup

3.1 Data setup

Constituency parsing. We conduct experiments on Penn Treebank (PTB) 3.0 (Marcus et al., 1993) and Chinese Treebank (CTB) (Xue et al., 2005). Many previous researchers report that the results on CTB5.1 are unstable and of high variance (Zhang et al., 2020; Yang and Deng, 2020). So we follow the suggestion of Zhang et al. (2020) to conduct experiments on CTB7 instead of CTB5.1 for more robust evaluation as CTB7 has more test sentences and has a higher annotation quality. We use the standard data splits for both PTB and CTB.

Nested NER. We conduct experiments on three benchmark datasets: ACE2004 (Doddington et al., 2004), ACE2005 (Walker et al., 2006), and GENIA (Kim et al., 2003). We use the same data preprocessing as Shibuya and Hovy (2020) 3.

3.2 Evaluation

We report labeled recall/precision/F1 scores based on EVALB 4 for constituency parsing; span-level labeled recall/precision/F1 scores for nested NER. All reported results are averaged over three runs with different random seeds.

3.3 Implementation details

We use "bert-large-cased" (Devlin et al., 2019) for PTB, ACE2004 and ACE2005; "bert-chinese-based" for CTB; and "biobert-large-cased-v1.1" (Lee et al., 2020) for GENIA. We use no other external resources (e.g., predicted/gold POS tags, external static word embedding). The hidden size of LSTM is set to 1000 for both the encoder and the decoder. We add dropouts in LSTM/MLP layers. The dropout rate is set to 0.33. The hidden and output sizes of all MLPs are set to 500. The value of gradient clipping is set to 5. The number of training epochs is set to 10 for PTB, CTB, GENIA; 50 for ACE2004/2005. We use Adam (Kingma and Ba, 2015) as the optimizer with $\beta_1 = 0.9$ and $\beta_2 = 0.9$. The maximal learning rate is set to $5e^{-5}$ for BERT and $2.5e^{-3}$ for all other components. We use the first 10% epochs to linearly warmup the learning rates of each components to their maximum value and gradually decay them to zero for the rest of epochs. We batch sentences of similar lengths to make full use of GPUs and the number of tokens in a single batch is set to 3000.

4 Main result

On both PTB and CTB, we find incorporating $E_{y_{t-1}}$ in Eq. 1 leads to a slightly inferior performance (-0.02 F1 score on PTB and -0.05 F1 score on CTB), so we report results without this input feature.

Table 2 shows the results on PTB test set. Our method achieves 96.01 F1 score, outperforming the method of Nguyen et al. (2021a) by 0.31 F1 and having the same worst-case $O(n^2)$ parsing time complexity as theirs 5. It also outperforms all span-

\footnote{https://github.com/yahshibu/nested-ner-tacl2020-transformers}

\footnote{https://nlp.cs.nyu.edu/evalb}

\footnote{In their paper, they claim an $O(n)$ time complexity, which treats the complexity of a single pointing operation as $O(1)$. This calculation, however, assumes full GPU parallelization. Without parallelization, their method has a worst-case $O(n^2)$ time complexity as ours.}

<table>
<thead>
<tr>
<th>Model</th>
<th>P</th>
<th>R</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kitaev et al. (2019) [S]</td>
<td>95.46</td>
<td>95.73</td>
<td>95.59</td>
</tr>
<tr>
<td>Zhou and Zhao (2019) [S]</td>
<td>95.70</td>
<td>95.98</td>
<td>95.84</td>
</tr>
<tr>
<td>Zhang et al. (2020) [S]</td>
<td>95.85</td>
<td>95.53</td>
<td>95.69</td>
</tr>
<tr>
<td>Yang and Deng (2020) [T]</td>
<td>96.04</td>
<td>95.55</td>
<td>95.79</td>
</tr>
<tr>
<td>Nguyen et al. (2020) [S]</td>
<td>-</td>
<td>-</td>
<td>95.48</td>
</tr>
<tr>
<td>Wei et al. (2020) [S]</td>
<td>95.5</td>
<td>96.1</td>
<td>95.8</td>
</tr>
<tr>
<td>Tian et al. (2020) [S]</td>
<td>96.09</td>
<td>95.62</td>
<td>95.86</td>
</tr>
<tr>
<td>Xin et al. (2021) [S]</td>
<td>96.29</td>
<td>95.55</td>
<td>95.92</td>
</tr>
<tr>
<td>Nguyen et al. (2021a) [Q]</td>
<td>-</td>
<td>-</td>
<td>95.7</td>
</tr>
<tr>
<td>Cui et al. (2021) [S]</td>
<td>95.70</td>
<td>96.14</td>
<td>95.92</td>
</tr>
<tr>
<td>Ours [Q]</td>
<td>96.19</td>
<td>95.83</td>
<td>96.01</td>
</tr>
</tbody>
</table>

Table 3: Results on CTB7. All models use BERT as encoders.

<table>
<thead>
<tr>
<th>Model</th>
<th>P</th>
<th>R</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zhang et al. (2020) [S]</td>
<td>91.73</td>
<td>91.38</td>
<td>91.55</td>
</tr>
<tr>
<td>Ours [Q]</td>
<td>91.66</td>
<td>91.31</td>
<td>91.49</td>
</tr>
</tbody>
</table>

Table 4 shows the results on three benchmark datasets on nested NER. We find that incorporating $E_{\ell_{j-1}}$ is important, leading to +0.67 F1 score and +0.52 F1 score on ACE2004 and ACE2005, respectively. Although our method underperforms two recent state-of-the-art methods: Shen et al. (2021) and Tan et al. (2021), we find it has a competitive performance to other recent works (Wang et al., 2021; Yan et al., 2021; Fu et al., 2021). The most comparable one is the method of Fu et al. (2021), which belongs to parsing-based methods as ours. They adapt a span-based constituency parser to tackle nested NER using the CYK algorithm for training and inference. Our model outperforms theirs by 0.34 F1 and 0.13 F1 scores on ACE2004 and ACE2005 and has a similar performance to theirs on GENIA, meanwhile enjoying a lower inference complexity.

5 Analysis

Error analysis. As we discussed previously, bottom-up parsing can make use of the subtree features when predicting parent spans, so it is expected to have higher F1 scores on longer spans. To verify this, we plot Fig. 4 to show the changes of F1 scores with different constituent span lengths on the PTB test set. We can see that our method consistently outperforms the method of (Nguyen et al., 2021a) on all span lengths, but our advantage is most prominent for spans of length >30, which verifies our conjecture. In Fig. 5, we can see that when a constituent has multiple children (>3), our method performs much better than that of (Nguyen et al., 2021a), which validates the benefit of n-ary tree parsing. An intuitive explanation of this benefit is that our method predicts n-ary branching structures in a single step, whereas theirs needs multiple steps, which is more error-prone.
Table 4: Results on ACE2004, ACE2005 and GENIA. All models use BERT as encoders.

**Effect of beam search.** We also tried beam search but observed very slight improvement or even worse performance (e.g., +0.05 F1 score on PTB and -0.03 F1 score on CTB when we use a beam size 20). Hence we report all results using greedy decoding for simplicity. This suggests that greedy decoding can yield near-optimal solutions, indicating that our model is less prone to the error propagation problem.

**Effect of training loss.** As discussed in Sec. 2.3, we find that explicitly considering the structural consistency constraints when normalizing is harmful (-0.12 F1 score on PTB, -0.10 F1 score on CTB). We speculate that not enforcing the constraints during training can help the model to learn the constraints implicitly, which is helpful for the model to generalize better on the unseen test set. Notably, Nguyen et al. (2021a) also adopt this strategy, i.e., normalizing over all boundaries.

**Speed.** Similar to Nguyen et al. (2021a), the training process (i.e., teacher forcing) can be fully parallelized without resorting to structured inference, which could be compute-intensive or hard to parallelize. On PTB, it takes only 4.5 hours to train the model using BERT as the encoder with a single Titan V GPU. As for parsing, our method has the same parsing complexity as Nguyen et al. (2021a), i.e., worst-case $O(n^2)$. Table 5 shows the speed comparison on parsing the PTB test set (we report values based on a single Titan V GPU and not using BERT as encoder following Nguyen et al. (2021a)). We report the average number of pointing actions in Appendix A.2.

<table>
<thead>
<tr>
<th>System</th>
<th>Speed (Sents/s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Petrov and Klein (2007) (Berkeley)</td>
<td>6</td>
<td>1.0x</td>
</tr>
<tr>
<td>Zhu et al. (2013) (ZPar)</td>
<td>90</td>
<td>15.0x</td>
</tr>
<tr>
<td>Stern et al. (2017)</td>
<td>76</td>
<td>12.7x</td>
</tr>
<tr>
<td>Shen et al. (2018)</td>
<td>111</td>
<td>18.5x</td>
</tr>
<tr>
<td>Nguyen et al. (2020)</td>
<td>130</td>
<td>21.7x</td>
</tr>
<tr>
<td>Zhou and Zhao (2019)</td>
<td>159</td>
<td>26.5x</td>
</tr>
<tr>
<td>Wei et al. (2020)</td>
<td>220</td>
<td>36.7x</td>
</tr>
<tr>
<td>Gómez-Rodríguez and Vilares (2018)</td>
<td>780</td>
<td>130x</td>
</tr>
<tr>
<td>Kitaeve and Klein (2018) (GPU)</td>
<td>830</td>
<td>138.3x</td>
</tr>
<tr>
<td>Zhang et al. (2020)</td>
<td>924</td>
<td>154x</td>
</tr>
<tr>
<td>Nguyen et al. (2021a)</td>
<td>1127</td>
<td>187.3x</td>
</tr>
<tr>
<td>Ours</td>
<td>855</td>
<td>142.5x</td>
</tr>
</tbody>
</table>

Table 5: Speed comparison.

6 Related Work

**Constituency parsing.** There are many methods to tackle constituency parsing, such as transition-based methods (Dyer et al., 2016; Cross and Huang, 2016; Liu and Zhang, 2017; Yang and Deng, 2020), span-based methods (Stern et al., 2017; Kitaev and Klein, 2018; Kitaev et al., 2019; Zhang et al., 2020; Wei et al., 2020; Nguyen et al., 2020; Xin et al., 2021), sequence-to-sequence (seq2seq)-based methods (Vinyals et al., 2015b; Fernández-González and Gómez-Rodríguez, 2020), sequence-labeling-based methods (Gómez-Rodríguez and Vilares, 2018; Vilares et al., 2019; Kitaev and Klein, 2020), among others.

Our work belongs to the category of seq2seq-based methods. Previous seq2seq models linearize constituency trees into bracket sequences (Vinyals et al., 2015b) or shift-reduce action sequences (Ma et al., 2017; Fernández-González and Gómez-Rodríguez, 2020). However, they may produce invalid outputs and their performance lags behind span-based methods. Recently, seq2seq models lin-
earize constituency trees into sequences of spans in pre-order (Nguyen et al., 2021a) or in in-order (Wei et al., 2021). Our method generates sequences of spans in post-order instead, which has the advantage of utilizing rich subtree features and performing direct n-ary tree parsing.

Binarization is de facto in constituency parsing, but there is a recent trend toward n-ary parsing. Previous span-based methods adopt either explicit binarization (Zhang et al., 2020) or implicit binarization (Stern et al., 2017; Kitaev and Klein, 2018). Although the implicit binarization strategy eliminates the need for binarization in training, it can only output binary trees during decoding. Xin et al. (2021) propose an n-ary-aware span-based method by defining semi-Markov processes on each parent span so that the transition scores of adjacent sibling child-spans are explicitly considered in parsing. Fernández-González and Gómez-Rodríguez (2019); Yang and Deng (2020) propose novel transition systems to model n-ary trees. Our method outputs n-ary trees without the need for binarization via a novel pointing mechanism.

**Parsing with pointer networks.** Pointer Networks (Vinyals et al., 2015a) are introduced to the parsing literature by Ma et al. (2018) and quickly become popular in various parsing sub-tasks because they are flexible to predict various trees/graphs and can achieve very competitive performance. Ma et al. (2018) linearize a dependency tree in a top-down depth-first and inside-out manner and use a pointer network to predict the linearized dependency tree, which is then extended by Lin et al. (2019) to discourse parsing. Liu et al. (2019) add shortcuts between the decoder states of the previously generated parents/siblings to the current decoder states in both dependency and discourse parsing. Fernández-González and Gómez-Rodríguez (2019) propose a left-to-right dependency parser that predicts the heads of each word autoregressively, and later, they propose right-to-left and outside-in variants (Fernández-González and Gómez-Rodríguez, 2021a). They also adapt the left-to-right dependency parser to semantic dependency parsing (which predicts acyclic graphs instead of trees) (Fernández-González and Gómez-Rodríguez, 2020), discontinuous constituency parsing (by treating discontinuous constituency trees as augmented dependency trees) (Fernández-González and Gómez-Rodríguez, 2020), and joint dependency and constituency parsing (Fernández-González and Gómez-Rodríguez, 2020). They use a pointer network to reorder the sentence to reduce discontinuous constituency parsing to continuous constituency parsing (Fernández-González and Gómez-Rodríguez, 2021b). Nguyen et al. (2021a,b) cast (discourse) constituency/RST parsing as conditional splitting and use pointer networks to select the splitting points. Zhou et al. (2021) propose an action-pointer network for AMR parsing.

**Nested NER.** There are also many methods to tackle nested NER, such as hypergraph-based methods (Lu and Roth, 2015; Katiyar and Cardie, 2018; Wang and Lu, 2018), sequence-labeling-based methods (Shibuya and Hovy, 2020; Wang et al., 2021), parsing-based methods (Finkel and Manning, 2009; Wang et al., 2018; Fu et al., 2021), layered methods (Fisher and Vlachos, 2019; Wang et al., 2020; Luo and Zhao, 2020), span-based methods (Yu et al., 2020; Li et al., 2021), object-detection-based methods (Shen et al., 2021; Tan et al., 2021) etc.

Our work belongs to the category of parsing-based methods. Finkel and Manning (2009) insert named entities into a constituency tree and use a discriminative parser (Finkel et al., 2008) for learning and prediction. Wang et al. (2018) adapt a shift-reduce transition-based parser to output a constituency forest instead of a constituency tree for nested NER. Fu et al. (2021) adapt a span-based neural TreeCRF parser, treat nested named entities as the observed parts of a partially-observed constituency tree and develop a masked inside algorithm to marginalize all unobserved parts for maximizing the probability of the observed named entities. Our method has a better performance as well as a lower time complexity than Fu et al. (2021). Recently, Lou et al. (2022) extend the work of Fu et al. (2021), casting nested NER to lexicalized constituency parsing for leveraging headword information. They achieve a higher performance at the cost of a higher parsing complexity, i.e., \( O(n^4) \).

**7 Discussion and future work**

In the deep learning era, global optimization on trees becomes less important in both training and decoding. Teng and Zhang (2018) show that a span-based model trained with a local span classification loss performs well in conjunction with CYK decoding. Wei et al. (2020); Nguyen et al. (2020) show that top-down greedy decoding per-
forms comparably. In this work we have shown that greedy decoding works well. Thus it would also be a fruitful direction to design more powerful neural decoders which can leverage more subtree information and can maintain structural consistency. Also, it is a fruitful direction to devise more powerful span representations.

8 Conclusion

In this work we have presented a novel pointing mechanism and model for bottom-up constituency parsing, which allows n-ary tree parsing in linear steps. Experiments on multiple datasets show the effectiveness of our methods in both constituency parsing and nested NER.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. This work was supported by the National Natural Science Foundation of China (61976139).

References


Chao Lou, Songlin Yang, and Kewei Tu. 2022. Nested named entity recognition as latent lexicalized constituency parsing. In ACL.


David Vilares, Mostafa Abdou, and Anders Søgaard. 2019. Better, faster, stronger sequence tagging con-
Christopher Walker, Stephanie Strassel, Julie Medero,


A Appendix

A.1 Uniqueness of oracle

If there are two oracles $o_1$ and $o_2$ outputting the same tree. Their parsing configuration is $(c_1, A_1, p_1, S_1)$ and $(C_2, A_2, p_2, S_2)$, respectively. Assume that the first $k$th pointing actions of $o_1$ and $o_2$ are the same (so they share the same cursor $c$) and the $k + 1$th action is $(c \rightarrow a_1, y_1)$ and $(c \rightarrow a_2, y_2)$ respectively. We enumerate all possibilities:

1. $a_2 < c < a_1$, then $(a_2, c)$ exists in $S_2$. $c_1$ would be updated to $a_1$, so thereafter the end-point of the generated span is $\geq a_1$, thus $(a_2, c)$ cannot exist in $S_1$ since $c < a_1$.

2. $a_1 < c < a_2$, then $(a_1, c)$ exists in $S_2$. Similar to the previous case, we can conclude that $(a_1, c)$ cannot exist in $S_1$.

3. $a_1 < a_2 < c$, then $(a_2, c)$ exists in $S_2$, but $a_2 \notin A_1$ for all remaining steps, thus $(a_2, c)$ cannot exist in $S_1$.

4. $a_2 < a_1 < c$. This is similar to the previous case.

5. $c < a_1 < a_2$, then $(a_1, c)$ exists in $S_1$, but $a_1 \notin A_2$ for the remaining steps, thus $(a_1, c)$ cannot exist in $S_2$.

6. $c < a_2 < a_1$. This is similar to the previous case.

Hence there is exact one oracle and we have proved it by contradiction.

A.2 Number of actions

The system of Nguyen et al. (2021a) needs exact $n - 1$ actions to parse a length-$n$ sentence. While our model requires $2n - 1$ actions in the worst case because we generate one span at each step and there are at most $2n - 1$ spans if the corresponding constituency tree is a full binary tree. So there is a concern that our model needs twice time to parse. Empirically, since the constituency trees in the treebank are not full binary trees in most cases, we need less than $2n - 1$ steps to parse. Fig. 6 shows the number of actions needed to parse with different sentence lengths in PTB training set. The red line is $y = x - 1$ and the green line is $y = 2x - 1$. In average, our method needs 1.13 actions per token. Nguyen et al. (2021a) needs 0.96 action per token. So, our method is around 20% slower than theirs. Fig. 7 shows the case in nested NER. We only need 0.40 action per token since the spans in nested NER is more sparse than that.
in constituency parsing. Our method is expectedly faster than other parsing-based methods in nested NER, such as the transition system of Wang and Lu (2018), which needs at least one action per word; and the span-based method of Fu et al. (2021), which needs cubic time for CYK parsing.