Neural Discontinuous Constituency Parsing

Miloš Stanojević and **Raquel G. Alhama**
Institute for Logic, Language and Computation (ILLC)
University of Amsterdam
{m.stanojevic, rgalhama}@uva.nl

Abstract

One of the most pressing issues in discontinuous constituency transition-based parsing is that the relevant information for parsing decisions could be located in any part of the stack or the buffer. In this paper, we propose a solution to this problem by replacing the structured perceptron model with a recursive neural model that computes a global representation of the configuration, therefore allowing even the most remote parts of the configuration to influence the parsing decisions. We also provide a detailed analysis of how this representation should be built out of sub-representations of its core elements (words, trees and stack). Additionally, we investigate how different types of swap oracles influence the results. Our model is the first neural discontinuous constituency parser, and it outperforms all the previously published models on three out of four datasets while on the fourth it obtains second place by a tiny difference.

1 Introduction

Research on constituency parsing has been mostly concentrated on projective trees, which can be modeled with Context-Free Grammars (CFGs). One of the main reasons for this is that modeling non-projective trees often requires richer grammar formalisms, which in practice implies slower runtime. For instance, the parsing algorithms for binary LCFRS—the most prominent grammar-based approach to parsing non-projective constituency trees—have computational complexity $O(n^{3k})$, where k is the *fan-out* of the grammar. For this reason, researchers turned to faster approximate methods. Approximations can be done in two ways: either on the types of structures that are predicted or on the parsing algorithm.

The first approach approximates discontinuous constituency structures with simpler structures for which more efficient algorithms exist. This method works as a pipeline: it converts the input to a simpler formalism, parses with it, and then converts it back. Relevant examples are the parsers by Hall and Nivre (2008) and Fernández-González and Martins (2015), who convert discontinuous constituents to dependencies, and Versley (2016), who also applied a conversion but in this case to the projective constituency trees.

The second approach—approximation on the parsing algorithm—consists of an approximate search for the most probable parse. This is analogous to the search done by transition-based parsers, which greedily search through the space of all possible parses, resulting in very fast models. The first transition-based discontinuous constituency parser of this sort was presented in Versley (2014), and it consists of a shift-reduce parser that handles discontinuities with swap transitions. This parser was very similar to dependency parsers with swap transitions (Nivre, 2009; Nivre et al., 2009), but unlike its dependency equivalents, it did not exhibit higher accuracy. Later work on discontinuous transition-based parsing was largely focused on finding alternative transitioning systems to handle discontinuity. Maier (2015) and Maier and Lichte (2016) proposed new types of swap operations (*CompoundSwap* and *SkipShift*) to make the transition sequences shorter—and therefore easier to learn. Coavoux and Crabbé (2017) went even further by modifying not only the transitions but the whole configuration structure by introducing an additional stack.

Over the years the transitioning system has seen some progress, but the learning model has remained the same : a sparse linear model trained with structured perceptron and early update strategy (Collins, 2002; Collins and Roark, 2004; Huang et al., 2012). This model requires heavy feature engineering and has a limited capacity in modeling interaction between the features.

Maier and Lichte (2016) argue that one of the biggest problems of transition based systems is precisely their greedy search, because they cannot recover from the bad decisions made in earlier parsing steps. Some researchers try to account for this problem by increasing the beam size, but there is a limit on how much the beam can be increased while remaining efficient for practical use (Coavoux and Crabbé, 2017).

The solution we propose is to use a probabilistic model that exploits the information from the whole configuration structure when making the decision for the next action. This can be achieved by using recurrent neural models that allow information to flow all the way from the individual characters, up trough the words, POS tags, subtrees, stack and buffer until the final configuration representation. Thanks to using a neural network model, which removes the need for feature engineering, we can concentrate on the question of which representations are more relevant for the model at each step of the flow. Thus, we reflect on how alternative representations should impact the task, and we report their relative contribution in an ablation study.

In our work, we also reduce the number of swap transitions by trying to postpone them as much as possible, in a style similar to the lazy-swap used in Nivre et al. (2009) —albeit with an even lower number of swaps. This change influences the model indirectly by introducing a helpful inductive bias.

Our model gets state-of-the-art results on Negra, Negra-30 and TigerSPMRL datasets, and on the TigerHN achieves the second best published result. To the best of our knowledge this is the first work that uses neural networks in the context of discontinuous constituency parsing.

2 Transition System

We base our transitioning system on the shift-promote-adjoin transitions proposed in Cross and Huang (2016), because they remove the need for explicit binarization. Transition-based parsers consist of two components: a configuration that represents a parsing state and a set of transitions between configurations.

The configuration consists of two data structures: a stack S that contains all the constituents built so far, and a buffer B of words that remain to be processed. The initial configuration consists of a buffer filled with words and an empty stack—presented as the *axiom* in Figure 1. The objective is to find a sequence of transitions that lead to a *goal* state in which the buffer is empty and the stack contains only one constituent with the ROOT label. The *shift* transition moves the first element from the buffer to the top of the stack. The *pro(X)* transition "promotes" the topmost element of the stack: it replaces it with a tree that has non-terminal X and the topmost element of the stack as its only child, which also becomes its head constituent. The *adj*$\curvearrowleft$ transition adjoins the second topmost element of the stack as a leftmost child of the topmost element of the stack. The *adj*$\curvearrowright$ transition is a mirror transition of the *adj*$\curvearrowleft$.

The transitions described so far are enough to handle projective constituency structures, and have been used with success for this task in Cross and Huang (2016). To make the parser able to process discontinuous constituents we need an additional transition that allows for constituents that are far apart on the stack to become close, so that they can be adjoined into a new constituent. For this we use the *swap* transition from Nivre (2009). This transition takes the second topmost element from the stack and puts it back to the buffer. To prevent infinite loops of *shift-swap* transitions, we put a constraint that *swap* can be applied only to constituents that have not been swapped before. To do this we use the linear ordering of constituents $<_{ind}$ based on the position of the leftmost word in their yield (Maier and Lichte, 2016).

2.1 Oracle

In the case of non-projective parsing, the extraction of the oracle is not trivial because there can be many possible oracles that would derive the same tree. Therefore it is common practice to use some heuristic to extract only one of the possible oracles.

To construct the oracle, we start with the initial configuration and apply the first transition whose conditions are satisfied. We keep applying transitions to the resulting configurations until the goal is reached. The transitions are determined as follows: first, we apply *adj*$\curvearrowleft$, *adj*$\curvearrowright$ or *pro(X)* if

$$
\begin{array}{rl}
\textit{axiom} & \langle\ [],\ \ [w_1, w_2, ..., w_n]\ \rangle \\[6pt]
\textit{shift} & \dfrac{\langle\ S,\ \ x|B\ \rangle}{\langle\ S|x,\ \ B\ \rangle} \\[6pt]
\textit{pro(X)} & \dfrac{\langle\ S|t,\ \ B\ \rangle}{\langle\ S|X(t),\ \ B\ \rangle} \\[6pt]
\textit{adj}\curvearrowleft & \dfrac{\langle\ S|t|X(t_1 \dots t_k),\ \ B\ \rangle}{\langle\ S|X(t, t_1 \dots t_k),\ \ B\ \rangle} \\[6pt]
\textit{adj}\curvearrowright & \dfrac{\langle\ S|X(t_1 \dots t_k)|t,\ \ B\ \rangle}{\langle\ S|X(t_1 \dots t_k, t),\ \ B\ \rangle} \\[6pt]
\textit{swap} & \dfrac{\langle\ S|t_1|t_2,\ \ B\ \rangle}{\langle\ S|t_2,\ \ t_1|B\ \rangle}\ t_1 <_{ind} t_2 \\[6pt]
\textit{goal} & \langle\ ROOT,\ \ \epsilon\ \rangle
\end{array}
$$

Figure 1: Transition System

one of those produces a constituent that is found in the tree; in case of failure, we check the condition for applying *swap*, which varies depending on the type of oracle, as we define next. If all these checks fail then a *shift* transition is performed.

2.1.1 Eager Oracle

Nivre (2009) introduced *swap* transitions with a very simple oracle. We can define the swapping condition for the extraction of the Eager oracle transition sequence as:

$$s_1 <_G s_0 \qquad (1)$$

where s_0 and s_1 are the topmost and second topmost elements of the stack respectively, and $<_G$ is the *projective ordering* of the nodes in the tree. That ordering can be computed by visiting the nodes in the tree in the postorder traversal.

This is the technique that has been used in most previous proposals on discontinuous constituency parsing (Maier, 2015; Maier and Lichte, 2016).

2.1.2 Lazy Oracle

Eager swapping strategy produces a large number of *swap* transitions which makes them difficult to predict. For this reason, Nivre et al. (2009) introduced a *lazy-swap* operation that postpones swapping by having an additional condition during the construction of an oracle. This technique was used successfully in Versley (2014) to improve over the eager swapping baseline. As an example, in Figure 2a word $w1$ should shift and swap many times to get to word $w5$ in order to construct constituent C. In contrast, a Lazy oracle would postpone swapping until constituent B is built so that only one swap operation over node B would be enough for word $w1$ to get to word $w5$.

In order to define that condition in the context of discontinuous constituency parsing, we need to define a few other terms. First of all, we call a *projective constituent* any constituent that yields a continuous span of words (marked with blue color in Figure 2). Note that a projective constituent might contain non-projective constituents as its descendants. A *fully projective constituent* is a constituent that is projective and whose descendants are all projective (marked with red in Figure 2). Finally, a *maximal fully projective constituent* is a fully projective constituent whose parent is not a fully projective constituent (marked green in Figure 2). Finally, we define a function $MPC(x)$ that returns the closest maximally projective constituent that is ascendant of a constituent x if there is one; otherwise, it returns x.

The condition for the lazy swap can now be expressed as:

$$s_1 <_G s_0 \quad \wedge \quad MPC(s_0) \neq MPC(b_0) \qquad (2)$$

where s_0 and b_0 are the topmost elements of the stack and buffer, respectively. This means that we do not allow *swap* to penetrate into maximally projective constituents, so swapping can be delayed until the maximally projective constituent has been built.

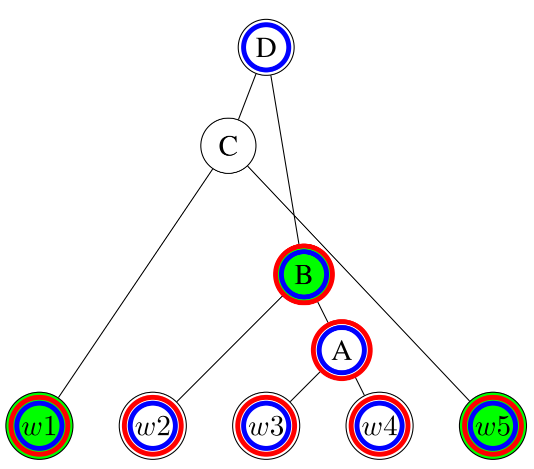
2.1.3 Lazier Oracle

The standard Lazy swap strategy helps in cases where MPC constituents exist, like in Figure 2a. But in cases like Figure 2b there are no MPC constituents (except for words), so Lazy would not show any improvement over Eager. Still, even in this case it is visible that swapping $w1$ should be postponed until B is built. We introduce an oracle strategy called Lazier that implements the heuristic of postponing swapping over projective constituents.[1]

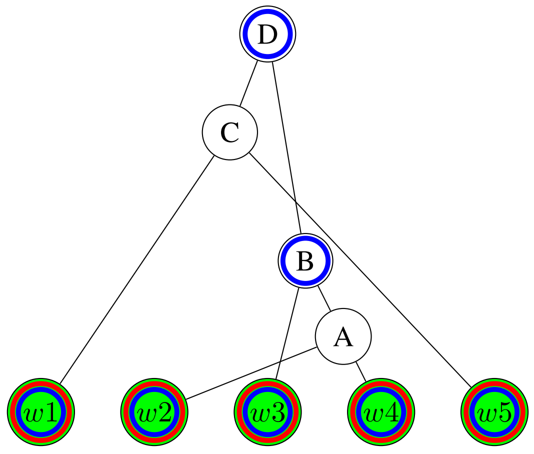
Let a function $CPC(x)$ return the closest projective constituent ascendant of a constituent x. The condition for swap operation can now be expressed with:

$$s_1 <_G s_0 \quad \wedge \quad CPC(s_0) = CPC(s_1) \qquad (3)$$

[1]The same intuition is followed in the Barriers strategy of Versley (2014).



(a) Tree with Maximal Fully Projective Node B



(b) Tree without any Maximal Fully Projective Nodes (other than words themselves)

Figure 2: Example tree structures

This constraint prevents *swap* from allowing constituents to escape their closest projective ancestor. If we know that the *swap* operation can be performed, i.e. if $s_1 <_G s_0$, it is easy to show that in that case $CPC(s_0) = CPC(s_1) \implies MPC(s_0) \neq MPC(b_0)$ or in other words that Lazy is a special case of Lazier. There are are only two cases to consider about $CPC(s_0)$: case a) s_0 and $CPC(s_0)$ are separated by a non-empty sequence of non-projective constituents and case b) s_0 is the immediate child of $CPC(s_0)$. In case a) from definition of maximal projective constituents follows that $MPC(s_0) = s_0$ and therefore $MPC(s_0) \neq MPC(b_0)$ since s_0 and b_0 are non-overlapping. In case b) we need to consider two possible options: b1) $CPC(s_0)$ is fully projective and b2) $CPC(s_0)$ is not fully projective. Case b1) is not possible because it leads to contradiction with original condition $s_1 <_G s_0$. Case b2) leads again to $MPC(s_0) = s_0$ and by that $MPC(s_0) \neq MPC(b_0)$.

3 Model

As mentioned before, our goal is to have a model that can have a global representation of the parsing state. In order to define this global representation of the configuration, we first need to analyze what are the proper representations of its subparts.

3.1 How to Represent Terminal Nodes?

The representations induced by neural networks are continuous vectors that encode the information that is relevant for the loss function. The initial nodes in the computation graph are often embeddings that represent the atomic inputs in the model. In our model, the embedding of a terminal node is computed by concatenating the following four embeddings and then applying the affine transformation to compress the result into a smaller vector:

- a trained word embedding
- a trained POS tag embedding
- a pre-trained word embedding
- a trained character embedding of the word

Trained embeddings (both word and POS tag embedding) are automatically trained by our model to better suit the task that we are solving. The usage of pre-trained embeddings has become standard in neural parsing models: these presentations are helpful because they bring additional contextual information from a bigger non-annotated corpora. The embeddings that we use in this work are the ones distributed with the Polyglot package (Al-Rfou et al., 2013).

The character embedding representation of a word is computed by composing the representations of each character in the word form. This can be useful to recover some of the morphological features present in the word, such as suffixes or prefixes. We compose character embeddings by running a bi-directional LSTM (Bi-LSTM) over the characters (Ling et al., 2015; Ballesteros et al., 2015).

The embeddings composed in this way express the properties of a word, but they ignore the context in which the word appears in the actual sentence. To address this we compute the final representation of the word by running a separate Bi-LSTM model over the initial vectors of the terminals in the same way as done by Kiperwasser and Goldberg (2016) and Cross and Huang (2016).

3.2 How to Represent Non-Terminal Nodes?

During the parsing process we need to produce the representations of the full subtrees that are going to be placed on the stack. In the dependency parsing literature, many approaches for representing dependency subtrees use the representation of the head word. If the representation of the head word is computed using a model that takes context into account, such as Bi-LSTM models, then this simple architecture can give good results (Kiperwasser and Goldberg, 2016; Cross and Huang, 2016). However, we believe that this is not the right approach for discontinuous constituency parsing. The reason is that, for the parser to know to which constituents it should attach the current constituent, it needs to know which arguments have already been attached and which ones are missing. In other words, even if the head of two different constituents is the same, their representation should be different because they have different requirements.

To address this we use a "composition function" approach where we recursively compute the representation of the constituent. Recursive neural networks (RecNN) (Goller and Küchler, 1996) are one way of accomplishing this. Dyer et al. (2015) use RecNN to compute the representation of the subtrees in the dependency structure. We adapt this model to our case in the following way. For binary constituents (i.e. outputs of *adj*$\curvearrowleft$ and *adj*$\curvearrowright$) the composition function takes the representation of the head constituent h_{head}, the representation of the complement h_{comp} and one single bit that represents the directionality of the $e_{\curvearrowleft}$ in the adjoining operation (0 for *adj*$\curvearrowleft$ and 1 for *adj*$\curvearrowright$). The resulting h_{new} representation is computed as follows:

$$h_{new} = tanh(W_{adj}[h_{head}; h_{comp}; e_{\curvearrowleft}] + b_{adj})$$

Here, semi-colon (;) represents vector concatenation, and W_{adj} and b_{adj} are the weight matrix and the bias vector that are trained together with the rest of the model, to optimize the desired loss function.

The transition *pro(X)* also creates new trees and its composition function can be seen as a function of a Simple RNN model:

$$h_{new} = tanh(W_{pro}[h_{head}; e_{nt}] + b_{pro})$$

Here e_{nt} is the embedding for the non-terminal to which constituent gets promoted. W_{pro} and b_{pro} are again the weight matrix and the bias vector whose values are estimated during training.

Simple RNN models have been shown to suffer from vanishing gradient problem, and for that reason they have been largely replaced with LSTM models (Hochreiter and Schmidhuber, 1997). The same holds for recursive neural network models. Le and Zuidema (2016) have shown that, for deep and complex hierarchical structures, the models that have a memory akin to the memory in LSTM are much more robust towards the vanishing gradient problem. Thus, in our work we use the Tree-LSTM neural architecture from Tai et al. (2015), but the alternative recursive version of LSTM by Le and Zuidema (2015) could be used as well.

In the Tree-LSTM model each constituent is represented by the hidden state h and the *memory cell* c. The composition function for the binary constituents with representations h_{head}, c_{head}, h_{comp} and c_{comp} computes the new representations h_{new} and c_{new} in the following way:

$$\begin{aligned}
f_{head} &= \sigma(W^{(f)}_{11} h_{head} + W^{(f)}_{12} h_{comp} + b^{(f)}_{\curvearrowleft}) \\
f_{comp} &= \sigma(W^{(f)}_{21} h_{head} + W^{(f)}_{22} h_{comp} + b^{(f)}_{\curvearrowleft}) \\
i &= \sigma(W^{(i)}_{1} h_{head} + W^{(i)}_{2} h_{comp} + b^{(i)}_{\curvearrowleft}) \\
o &= \sigma(W^{(o)}_{1} h_{head} + W^{(o)}_{2} h_{comp} + b^{(o)}_{\curvearrowleft}) \\
u &= tanh(W^{(u)}_{1} h_{head} + W^{(u)}_{2} h_{comp} + b^{(u)}_{\curvearrowleft}) \\
c_{new} &= i \odot u + f_{head} \odot c_{head} + f_{comp} \odot c_{comp} \\
h_{new} &= o \odot tanh(c_{new})
\end{aligned}$$

All the W matrices and the bias vectors b are trained parameters of the composition function. For each equation above there is an alternative equation that instead of bias $b_{\curvearrowleft}$ uses bias $b_{\curvearrowright}$. Which equation/bias will be used depends on the directionality of the adjoining operation.

For the promote transition, since it creates only one unary node, we can use almost the same computation as in the standard LSTM:

$$\begin{aligned}
f &= \sigma(W^{(f)}h_{head} + W_{nt}^{(f)}e_{nt} + b^{(f)}) \\
i &= \sigma(W^{(i)}h_{head} + W_{nt}^{(i)}e_{nt} + b^{(i)}) \\
o &= \sigma(W^{(o)}h_{head} + W_{nt}^{(o)}e_{nt} + b^{(o)}) \\
u &= tanh(W^{(u)}h_{head} + W_{nt}^{(u)}e_{nt} + b^{(u)}) \\
c_{new} &= i \odot u + f \odot c_{head} \\
h_{new} &= o \odot tanh(c_{new})
\end{aligned}$$

The main difference from the standard LSTM is that here we additionally use the information from the non-terminal embedding e_{nt} to which the constituent is promoted.

3.3 How to Represent a Configuration?

We have covered how to represent syntactic objects (terminal and non-terminal nodes) that are stored in the stack and the buffer, but we still need to decide how to combine these representations to make a final decision about the next transition.

One possibility is to first find a suitable representation for the stack and the buffer individually, concatenate these representations and then apply a multi-layer perceptron (MLP) to produce the probabilities for the next action.

The stack and the buffer can be seen as the same type of data structure: the buffer can be interpreted as a stack that is filled by pushing the words in a sentence from the last to the first. Therefore, we can use same approach for modeling stack and buffer.

The most common approach for representing a stack structure in transition based parsers (both in perceptron and neural models) is to take the representations of the first few top constituents on the top of the stack. Thus, this approach assumes that only the top of the stack and buffer are relevant for deciding the next action. Even though this assumption seems reasonable in the context of continuous constituency parsing, for discontinuous parsing it can be very harmful because the constituents that we want to merge might be very far from each other in the stack, as argued in (Maier and Lichte, 2016).

In our work, we explore an alternative model that could address this problem; namely, the Stack-LSTM model proposed in (Dyer et al., 2015). This model consists of an LSTM that processes the whole stack as a sequence, to obtain in this way a representation of the stack that includes *all* of its elements. This approach gave good results on continuous dependency parsing, but its properties should be even more important for discontinuous parsing, since it allows to keep in the stack a representation of all the constituents.

Given the stack h_{stack} and buffer h_{buffer} representations computed by Stack-LSTMs, we compute the configuration representation h_{conf} by concatenating these vectors and then applying an affine transformation followed by a $ReLU(\cdot)$ nonlinearity:

$$h_{conf} = ReLU(W_{conf}[h_{stack}; h_{buffer}] + b_{conf})$$

This vector representation encodes the whole configuration: the information flow passes trough every character, every POS tag, every constituent in the stack and in the buffer. From this vector representation we can compute the probability of the transition z from the set of possible transitions Z by applying one final softmax layer:

$$p(z|h_{conf}) = \frac{exp(w_z^T h_{conf} + b_{z_i})}{\sum_{z_i \in Z} exp(w_{z_i}^T h_{conf} + b_{z_i})}$$

The probability of the whole sequence of transitions is defined as the product of the probabilities of its transitions:

$$p(\mathbf{z}|\mathbf{w}) = \prod_{i=1}^{|z|} p(z_i|h_{conf\_i})$$

The parameters are optimized for maximum likelihood of the oracle sequence of transitions.

4 Experiments

We empirically test the performance of our parser on two German constituency treebanks: Negra and Tiger. The preprocessing applied to these treebanks follows the same methods used in other discontinuous constituency parsing literature, as described in Maier (2015) and implemented in the *tree-tools* software[2].

We use two different versions of the Negra treebank. The first version is filtered for the sentences up to 30 words, in order to remain comparable to previous grammar-based models; the second version includes sentences of all lengths. As for the Tiger treebank, we use two different splits: TigerHN (Hall and Nivre, 2008) and TigerSPMRL

[2]`https://github.com/wmaier/treetools`

name	value
trained word embedding dim.	100
pretrained word embedding dim.	64
POS embedding dim.	20
character embedding dim.	100
character Bi-LSTM layers	1
word Bi-LSTM layers	2
(non-)terminal node repr. dim.	40
configuration repr. dim.	100
stack LSTM dim.	100
stack LSTM layers	2
buffer LSTM dim.	100
buffer LSTM layers	2
optimizer	Adam
optimizer parameter b1	0.9
optimizer parameter b2	0.999
beam size	16

Table 1: Hyper-parameters of the model

	#swaps	jump size
Eager	43.17 ±49.21	1.00 ± 0.0
Lazy	10.96 ± 9.96	6.88 ±5.54
Lazier	5.40 ± 3.05	10.03 ±11.05

Table 2: Average number of swaps and jump sizes per sentence

(Maier, 2015). We evaluated the model with the evaluation module of *discodop*[3] parser.

Our model is implemented with DyNet (Neubig et al., 2017) and the code is available at `https://github.com/stanojevic/BadParser`. The concrete hyper-parameters of our model are shown in Table 1. We optimize the parameters with Adam optimizer on the training set, for 10 iterations with 100 random restarts, and we do model selection on the validation set for the F-score. During test time we use beam search with beam of size 16.

We conducted the development of our model on the TigerHN train and development sets. First we will analyze the effect of different model design decisions and then we show the results over the test set. The development set scores on TigerHN are shown in Table 3.

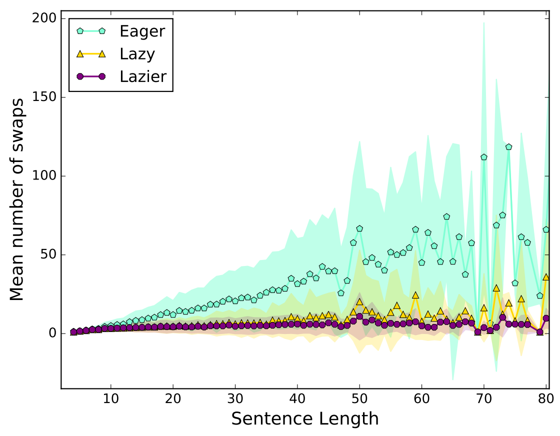
4.1 Which oracle is better?

The results in Table 3 show that the Eager oracle works better than Lazy for discontinuous constituents, but for continuous constituents (and over all constituents on average) Lazy works better. This can be explained by Lazy being very conservative about swaps: since their number is significantly reduced, the transition becomes difficult to predict, and thus the model gives up on predicting swaps and concentrates on the statistics for the projective operations. In other words, Lazy predicts swaps only if the statistical evidence for swaps is high. This can be seen by the contrast between high precision but very low recall on discontinuous constituents.

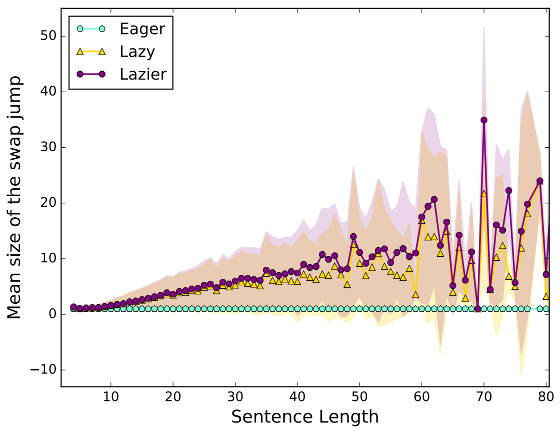
Eager works in the opposite direction: since it has observed many swaps it has a strong bias to predict them, which leads to a high recall but low precision. Lazier strikes a good balance between precision and recall on the discontinuous constituents, and because of that it outperforms both Eager and Lazy on F-score for both all constituents and discontinuous constituents.

The good result of Lazier cannot be subscribed only to the shorter transition sequences being easier to predict, because if it was up to the transition

[3]`https://github.com/andreasvc/disco-dop`



(a) Mean number of swaps per sentence length.



(b) Mean size of the swap jumps per sentence length.

Figure 3: The effect of different swap strategies of sentences with up to 80 words

Composition function	Bi-LSTM layers	Stack-LSTM	Oracle	All P	All R	All F	All E	Discontinuous P	Discontinuous R	Discontinuous F	Discontinuous E
Tree-LSTM	**2**	✓	**Lazier**	**84.85**	**83.81**	**84.33**	**46.64**	55.67	**46.54**	**50.70**	**37.48**
Head	2	✓	Lazier	61.86	53.91	57.61	13.87	16.43	6.63	9.45	3.49
RecNN	2	✓	Lazier	83.57	82.84	83.21	43.55	57.30	38.17	45.82	32.94
Tree-LSTM	0	✓	Lazier	81.31	80.30	80.80	39.85	48.17	33.67	39.64	27.37
Tree-LSTM	1	✓	Lazier	83.86	83.67	83.77	44.46	**58.12**	42.57	49.15	35.08
Tree-LSTM	2	top3	Lazier	82.29	81.07	81.68	40.64	54.39	31.49	39.89	26.57
Tree-LSTM	2	✓	Lazy	84.80	83.39	84.09	45.65	59.35	36.24	45.00	31.88
Tree-LSTM	2	✓	Eager	83.74	83.04	83.39	43.81	50.15	43.68	46.69	33.49

Table 3: Precision (P), Recall (R), F-score (F) and Exact (E), for our best model and ablated versions.

		Negra	**Negra-All**		**TigerHN**		**TigerSPMRL**
		All	**L≤40**	**All**	**L≤40**	**All**	
conv2dep	Hall and Nivre (2008)	-	-	-	79.93	-	-
	Fernández-González and Martins (2015)	82.56 *	81.08	80.52	**85.53**	**84.22**	80.62◇
LCFRS	van Cranenburgh (2012)	-	72.33	71.08	-	-	-
	van Cranenburgh and Bod (2013)	-	76.8	-	-	-	-
	Kallmeyer and Maier (2013)	75.75	-	-	-	-	-
Transition-Based	Versley (2014)	-	-	-	74.23	-	-
	Maier (2015)	76.95	-	-	79.52	-	74.71
	Maier and Lichte (2016)	-	-	-	80.02	-	76.46
	Coavoux and Crabbé (2017)	82.46	82.76	82.16	85.11	84.01	81.60
	This work	**83.29**	**83.39**	**82.87**	85.25	84.06	**81.64**

Table 4: Final results on test set, computed with *discodop* evaluation module. ⋆Trained on Negra-All. ◇Evaluated with SPRML scripts.

sequences length alone then Lazy would work better than Eager on the discontinuous constituents. The more likely explanation is that Lazier introduces an inductive bias in the model that is useful for generalization, and that allows the model to generalize better than Eager and Lazy.

We also quantified how many swaps are made by Eager, Lazy and Lazier. Figure 3 shows the statistics over the TigerHN training set for different sentence lengths; the aggregated statistics over all sentence lengths can be read in Table 2. We can observe in Figure 3(a) that, in the case of short sentences, all the swapping strategies give similar results, but as sentences get longer the number of swaps in Eager gets much higher and more unstable than lazier alternatives. We found that for some sentences Lazy and Lazier do with 2 swaps what Eager does with 126 swaps. Compared to Lazy, Lazier is much more stable in terms of the number of swaps, which can be seen by the standard deviation in Table 2. In Figure 3(b) shows a similar trend for the size of the jump of swap transitions. All the swaps of Eager make a jump of size 1, while the jumps of Lazier can go up to 91 words.

4.2 What is the best word representation?

We have tested whether the representation of a word based solely on its embeddings is enough to get good results or, instead, this representation should be refined by the bi-directional LSTM. Table 3 shows that adding layers to the bi-directional LSTM consistently improves the scores. The difference between not using a bi-directional LSTM and using 2 layers of bi-directional LSTM is 3.63

F-score, which is a big margin. Adding a third layer did not improve scores significantly.

4.3 What is the best composition function?

We have tried three options for composition functions: Head (use only the head word embedding instead of a composition function), RecNN and TreeLSTM – all presented in Section 3.2. As we expected, the head representation alone did not perform well, which shows that some type of composition function is needed. We find that using a recursive model with a memory cell improves results by 1.12 F-score, and thus we settle for the TreeLSTM composition function.

4.4 What is the best configuration representation?

We tested two configuration representations: the first one – top3 – takes the 3 topmost elements from the stack and the buffer as the representatives, while the second one – Stack-LSTM – models the whole content of the configuration via recurrent neural models. In line with our intuitions, the Stack-LSTM, thanks to considering the whole stack and buffer structure instead of only a few elements, outperforms top3 by a margin of 2.65 F-score points.

4.5 Comparison with other models

We took the version of our model that performed the best on the TigerHN development set and compared it on the four different datasets (two treebanks with two different splits) with other parsers.

In Table 4 we show the results compared to the other works published on these datasets. Our parser outperforms all the previously published models on all datasets except TigerHN, where it ends up second best after Fernández-González and Martins (2015). As shown in our previous analysis, exploring alternative representations of the different components has allowed us to construct a better model. We must also notice that, when comparing to other models, one influential cause of the good performance may be the capacity of our model, provided by the neural architectures. Neural networks allow modeling relations from input to output that are much more complex than those captured by the approaches we compare to, most of which use linear models based on perceptron or simple PCFG type of generative models.

We have also tested our model on the predicted POS tags from TigerSPMRL split, as provided in the shared task (Seddah et al., 2013). The results are shown in Table 5. The biggest strength of our model—its capacity— is in this case its biggest weakness: it causes the model parameters to overfit the noisy predicted tags during training, because we have not used any form of regularization. Model combinations like the one in Versley (2016) do not suffer from this because they implicitly do strong regularization. Our model could probably achieve better results on this dataset with stronger regularization, which we leave for future research.

TigerSPRML	F1 (spmrl.prm)	
	≤ 70	All
Versley (2014)	73.90	–
This work	77.25	76.96
F&M (2015)	77.72	77.32
Coavoux&Crabbé (2017)	79.44	79.26
Versley (2016)	79.84	79.50

Table 5: Results on SPMRL data with predicted tags.

5 Conclusion

We have presented the first neural model for discontinuous constituency parsing that achieves state-of-the-art results in three out of four standard datasets for discontinuous parsing with gold POS tags. Our findings suggest that i) bidirectional LSTM should be used for refining the representations of terminals even in the cases when they are going to be combined by a recursive model, ii) the performance of the composition function depends to a big extent on the availability of the memory cells, to prevent the vanishing gradient, iii) it is crucial to use all the elements in the stack and buffer in the decision process instead of just few elements on the top and iv) Lazier oracle gives better and more stable results than Eager and Lazy oracles on both continuous and discontinuous constituents.

Acknowledgments

We are thankful to Andreas van Cranenburgh and Wolfgang Maier for sharing the data and preprocessing scripts, and to the anonymous reviewers for suggestions that improved the paper. We are grateful to the ILLC for providing us with traveling funds to attend the conference.

References

Rami Al-Rfou, Bryan Perozzi, and Steven Skiena. 2013. Polyglot: Distributed word representations for multilingual nlp. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*. Association for Computational Linguistics, Sofia, Bulgaria, pages 183–192. http://www.aclweb.org/anthology/W13-3520.

Miguel Ballesteros, Chris Dyer, and Noah A. Smith. 2015. Improved transition-based parsing by modeling characters instead of words with lstms. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Lisbon, Portugal, pages 349–359. http://aclweb.org/anthology/D15-1041.

Maximin Coavoux and Benoit Crabbé. 2017. Incremental discontinuous phrase structure parsing with the gap transition. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, Valencia, Spain.

Michael Collins. 2002. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*. Association for Computational Linguistics, pages 1–8.

Michael Collins and Brian Roark. 2004. Incremental parsing with the perceptron algorithm. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, page 111.

James Cross and Liang Huang. 2016. Incremental parsing with minimal features using bi-directional lstm. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Association for Computational Linguistics, Berlin, Germany, pages 32–37. http://anthology.aclweb.org/P16-2006.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. 2015. Transition-based dependency parsing with stack long short-term memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Beijing, China, pages 334–343. http://www.aclweb.org/anthology/P15-1033.

Daniel Fernández-González and André F. T. Martins. 2015. Parsing as reduction. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Beijing, China, pages 1523–1533. http://www.aclweb.org/anthology/P15-1147.

Christoph Goller and Andreas Küchler. 1996. Learning task-dependent distributed representations by backpropagation through structure. In *Proceedings of the International Conference on Neural Networks*. IEEE, pages 347–352.

Johan Hall and Joakim Nivre. 2008. Parsing discontinuous phrase structure with grammatical functions. In *Proceedings of the 6th International Conference on Natural Language Processing (GoTAL)*. pages 169–180.

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

Liang Huang, Suphan Fayong, and Yang Guo. 2012. Structured perceptron with inexact search. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Stroudsburg, PA, USA, NAACL HLT '12, pages 142–151. http://dl.acm.org/citation.cfm?id=2382029.2382049.

Laura Kallmeyer and Wolfgang Maier. 2013. Data-driven parsing using probabilistic linear context-free rewriting systems. *Computational Linguistics* 39(1):87–119.

Eliyahu Kiperwasser and Yoav Goldberg. 2016. Simple and accurate dependency parsing using bidirectional lstm feature representations. *Transactions of the Association for Computational Linguistics* 4:313–327. https://transacl.org/ojs/index.php/tacl/article/view/885.

Phong Le and Willem Zuidema. 2015. Compositional distributional semantics with long short term memory. In *Proceedings of the Fourth Joint Conference on Lexical and Computational Semantics*. Association for Computational Linguistics, Denver, Colorado, pages 10–19. http://www.aclweb.org/anthology/S15-1002.

Phong Le and Willem Zuidema. 2016. Quantifying the vanishing gradient and long distance dependency problem in recursive neural networks and recursive LSTMs. In *ACL Workshop on Representation Learning for NLP*.

Wang Ling, Chris Dyer, Alan W Black, Isabel Trancoso, Ramon Fermandez, Silvio Amir, Luis Marujo, and Tiago Luis. 2015. Finding function in form: Compositional character models for open vocabulary word representation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Lisbon, Portugal, pages 1520–1530. http://aclweb.org/anthology/D15-1176.

Wolfgang Maier. 2015. Discontinuous incremental shift-reduce parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint*

Conference on Natural Language Processing (Volume 1: Long Papers). Association for Computational Linguistics, Beijing, China, pages 1202–1212. http://www.aclweb.org/anthology/P15-1116.

Wolfgang Maier and Timm Lichte. 2016. Discontinuous parsing with continuous trees. In *Proceedings of the Workshop on Discontinuous Structures in Natural Language Processing*. Association for Computational Linguistics, San Diego, California, pages 47–57. http://www.aclweb.org/anthology/W16-0906.

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

Joakim Nivre. 2009. Non-projective dependency parsing in expected linear time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*. pages 351–359. http://www.aclweb.org/anthology/P/P09/P09-1040.pdf.

Joakim Nivre, Marco Kuhlmann, and Johan Hall. 2009. An improved oracle for dependency parsing with online reordering. In *Proceedings of the 11th International Conference on Parsing Technologies (IWPT)*. pages 73–76.

Djamé Seddah, Reut Tsarfaty, Sandra Kübler, Marie Candito, Jinho D. Choi, Richárd Farkas, Jennifer Foster, Iakes Goenaga, Koldo Gojenola Galletebeitia, Yoav Goldberg, Spence Green, Nizar Habash, Marco Kuhlmann, Wolfgang Maier, Joakim Nivre, Adam Przepiórkowski, Ryan Roth, Wolfgang Seeker, Yannick Versley, Veronika Vincze, Marcin Woliński, Alina Wróblewska, and Eric Villemonte de la Clergerie. 2013. Overview of the SPMRL 2013 shared task: A cross-framework evaluation of parsing morphologically rich languages. In *Proceedings of the Fourth Workshop on Statistical Parsing of Morphologically-Rich Languages*. Association for Computational Linguistics, Seattle, Washington, USA, pages 146–182. http://www.aclweb.org/anthology/W13-4917.

Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Beijing, China, pages 1556–1566. http://www.aclweb.org/anthology/P15-1150.

Andreas van Cranenburgh. 2012. Efficient parsing with linear context-free rewriting systems. In *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, Avignon, France, pages 460–470. http://www.aclweb.org/anthology/E12-1047.

Andreas van Cranenburgh and Rens Bod. 2013. Discontinuous parsing with an efficient and accurate DOP model. In *Proceedings of the 13th International Conference on Parsing Technologies (IWPT)*.

Yannick Versley. 2014. Experiments with easy-first nonprojective constituent parsing. In *Proceedings of the First Joint Workshop on Statistical Parsing of Morphologically Rich Languages and Syntactic Analysis of Non-Canonical Languages*. Dublin City University, Dublin, Ireland, pages 39–53. http://www.aclweb.org/anthology/W14-6104.

Yannick Versley. 2016. Discontinuity re^2-visited: A minimalist approach to pseudoprojective constituent parsing. In *Proceedings of the Workshop on Discontinuous Structures in Natural Language Processing*. Association for Computational Linguistics, San Diego, California, pages 58–69. http://www.aclweb.org/anthology/W16-0907.