AN EMPIRICAL STUDY OF ENCODERS AND DECODERS IN GRAPH-BASED DEPENDENCY PARSING

Anonymous authors
Paper under double-blind review

ABSTRACT

Graph-based dependency parsing consists of two steps: first, an *encoder* produces a feature representation for each parsing substructure of the input sentence, which is then used to compute a score for the substructure; and second, a *decoder* finds the parse tree whose substructures have the largest total score. Over the past few years, powerful neural techniques have been introduced into the encoding step which substantially increases parsing accuracies. However, advanced decoding techniques, in particular high-order decoding, have seen a decline in usage. It is widely believed that contextualized features produced by neural encoders can help capture high-order decoding information and hence diminish the need for a high-order decoder. In this paper, we empirically evaluate the combinations of different neural and non-neural encoders with first- and second-order decoders and provide a comprehensive analysis about the effectiveness of these combinations with varied training data sizes. We find that: first, when there is large training data, a strong neural encoder with first-order decoding is sufficient to achieve high parsing accuracy and only slightly lags behind the combination of neural encoding and second-order decoding; second, with small training data, a non-neural encoder with a second-order decoder outperforms the other combinations in most cases.

1 INTRODUCTION

Dependency parsing (Kübler et al., 2009) is an important task in natural language processing (NLP) and a large number of methods have been proposed, most of which can be divided into two categories: graph-based methods (Dozat & Manning, 2017; Shi & Lee, 2018) and transition-based methods(Weiss et al., 2015; Andor et al., 2016; Ma et al., 2018) . In this paper, we focus on graph-based dependency parsing, which traditionally has higher parsing accuracy.

A typical graph-based dependency parser consists of two parts: first, an *encoder* that produces a feature representation for each parsing substructure of the input sentence and computes a score for the substructure based on its feature representation; and second, a *decoder* that finds the parse tree whose substructures have the largest total score.

Over the past few years, powerful neural techniques have been introduced into the encoding step that represent contextual features as continuous vectors. The introduction of neural methods leads to substantial increase in parsing accuracy (Kiperwasser & Goldberg, 2016). High-order decoding techniques, on the other hand, have seen a decline in usage. The common belief is that high-order information has already been captured by neural encoders in the contextual representations and thus the need for high-order decoding is diminished(Falenska & Kuhn, 2019).

In this paper, we empirically evaluate different combinations of neural and non-neural encoders with first- and second-order decoders to thoroughly examine their effect on parsing performance. From the experimental results we make the following observations: First, powerful neural encoders indeed diminish to some extend the necessity of high-order decoding when sufficient training data is provided. Second, with smaller training data, the advantage of a neural encoder with a second-order decoder begins to decrease, and its performance surpassed by other combinations in some treebanks. Finally, if we further limit the training data size to a few hundred sentences, the combination of a simple non-neural encoder and a high-order decoder emerges as the preferred choice for its robustness and relatively higher performance.

2 GRAPH-BASED DEPENDENCY PARSING

The main idea of graph-based dependency parsing is to formulate the task as a search for a maximum-spanning tree in a directed graph. The search comprises two steps: first computing the score for each substructure in the graph and then finding the tree with the highest score among substructure combinations.

Concretely, for a given input sentence $\mathbf{x} = x_1, \cdots, x_n$, the tokens and the dependency arcs between them are considered as vertexes and directed edges in a graph respectively. Assume that the whole graph is factorized into m substructures $\psi_1, \psi_2, \cdots, \psi_m$. Score these substructures with a function f. The desired max-spanning tree $\mathcal{T}^*$ is formulated as

$$\mathcal{T}^*) = \arg\max_{\mathcal{T}} \sum_{\psi_i \in \mathcal{T}} f(\psi_i, \mathbf{x})$$

where $\mathcal{T}$ ranges over all spanning trees in the graph.

The two steps of parsing described above are realized by an encoder and a decoder of the parser. The encoder implements the scoring function f. In a non-neural encoder, we define $f(\psi, \mathbf{x}) = \sum_d \mathbf{w}_d \cdot \mathbf{I}(d, \psi, \mathbf{x})$, where $\mathbf{I}$ is the indicator function determining whether a manually designed feature d is fired in substructure ψ and sentence $\mathbf{x}$, and $\mathbf{w}_d$ is a weight for feature d. In a neural encoder we use a neural network that takes sentence $\mathbf{x}$ as input and outputs the score for every substructure ψ.

The most common substructure factorization used in graph-based dependency parsing is to take each dependency arc as a basic substrucure. A decoder based on such factorization is called a *first-order* decoder. If the parse tree is restricted to be projective (i.e. no crossing between dependency arcs), the Eisner algorithm (Eisner, 1996) can be used as the decoder to produce the highest-scoring tree. For the non-projective case, the counterpart decoder is the Chu-Liu-Edmonds algorithm (Edmonds, 1967; Chu, 1965). The time complexity for the Eisner algorithms is $O(n^3)$, and $O(n^2)$ for Chu-Liu-Edmonds algorithm, where n is the sentence length.

If we consider two dependency arcs in combination as the basic substructure, for example, two dependency arcs with the head of one arc being the tail of the other (requiring three vertexes to index: *grandparent*, *head*, *child*), we require a *second-order* decoder. A modified version of the Eisner algorithm can be applied for second-order projective decoding in time complexity of $O(n^4)$. Since exact second-order decoding in non-projective dependency parsing is an NP-hard problem (McDonald & Pereira, 2006), we confine the scope of this paper to projective dependency parsing.

There exist different learning methods of graph-based dependency parsers. One method is to use a margin-based loss and maximize the margin between the score of the gold tree $\mathcal{T}$ and the incorrect tree with the highest score $\mathcal{T}'$:

$$\begin{aligned}\mathcal{L} = \max(0, 1 + &\sum_{\psi \in \mathcal{T}} f(\psi, \mathbf{x}) \\ &- \max_{\mathcal{T}' \neq \mathcal{T}} (\sum_{\psi \in \mathcal{T}'} f(\psi, \mathbf{x}) + \Delta(\mathcal{T}, \mathcal{T}')))\end{aligned}$$

$\Delta(\mathcal{T}, \mathcal{T}')$ represents the Hamming distance between $\mathcal{T}$ and $\mathcal{T}'$, which is the number of wrongly predicted dependency heads in $\mathcal{T}'$. Gradient-based methods such as Adam Kingma & Ba (2015) can be applied to optimize this objective function.

3 ENCODERS AND DECODERS

3.1 NEURAL AND NON-NEURAL ENCODERS

The expressive power of a neural encoder increases as the network gets more complex. In our experiment, we adopt two types of neural encoders: a one-layer LSTM encoder, and a two-layer LSTM encoder, in ascending order of expressive power.

One-Layer LSTM Encoder: This encoder employs a bi-directional long short-term memory (LSTM) network (Hochreiter & Schmidhuber, 1997) that takes in embedded word vectors as in-

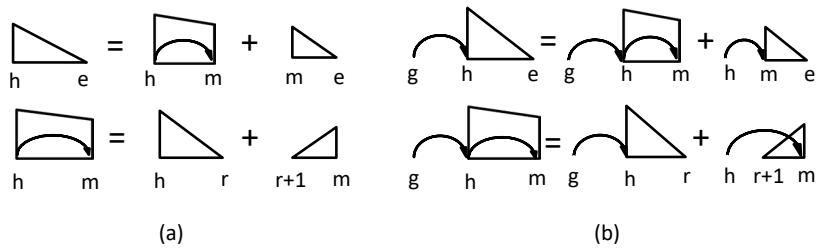


Figure 1: The combination of spans in the Eisner algorithm (a) and its second-order version (b). For brevity only one side is shown.

put. The hidden state representations output by the LSTM network are then used for substructure score computation with MLPs.

Two-Layer LSTM Encoder: This encoder is almost the same as the one-layer LSTM encoder except that there are two layers of LSTM networks. We limit the number of layers to 2 since it has been shown that adding more layers has little effect (Kiperwasser & Goldberg, 2016).

The computation of substructure scores follows that of Kiperwasser & Goldberg (2016) and Falenska & Kuhn (2019). For each input sentence $\mathbf{x} = x_1, \cdots, x_n$ we concatenate the embedding of word w and Part-Of-Speech (POS) tag t at position i as the representation for x_i:

$$x_i = e(w_i) \circ e(t_i)$$

By passing the representations through Bi-LSTM layers we obtain a feature vector containing contextual information for each position i:

$$h_i = BiLSTM(x_{1:n}, i)$$

The score of a substructure is computed based on these feature vectors. For each position in a substructure we pass the feature vectors at the position through a multi-layer perceptron (MLP).The MLP outputs at all the positions of the substructure are then concatenated to form the representation for the whole substructure. Another MLP is used to compute the substructure score from its representation.

Specifically, for the first-order case where the substructure is a dependency arc with head at position i and child at position j, its representation r_{ij} is:

$$r_{ij} = MLP_{head}(h_i) \circ MLP_{child}(h_j)$$

For the second-order case where the substructure contains two dependency arcs involving three positions i, j, k, its representation r_{ijk} is formulated as

$$r_{ijk} = MLP_{head}(h_i) \circ MLP_{child}(h_j) \circ MLP_{grandparent}(h_k)$$

Non-Neural Encoder: A non-neural encoder produces a sparse feature vector based on a manually designed feature template. We adopt the standard feature template used by McDonald & Pereira (2006) and Carreras (2007). The feature vector is then multiplied with a weight vector to produce the substructure score.

3.2 First- and Second-Order Decoders

Our first-order decoder is based on the Eisner algorithm, a widely used dynamic programming algorithm for first-order dependency parsing. The Eisner algorithm defines two types of structures for the dynamic programming procedure: *complete* spans and *incomplete* spans. A complete span, graphically represented as a triangle, stands for a subtree spanning towards one direction. An incomplete span, graphically represented as a trapezoid, stands for the sentence span covered by a dependency arc.

Starting from the base case, for every token in the sentence, there are two complete spans, one for each direction. Spans can be combined in the way shown in Figure 1(a). In each case, the decoder

Dataset	PTB	Czech-CAC	Russian	Chinese	Hebrew	Swedish
Training Data Size	36759	21899	3617	3578	4441	4215

Table 1: The number of sentences used in the experiment after pruning sentences longer than 40.

traverses all span pairs that can be combined and choose the pair with the highest score to form the new larger span. This is repeated from bottom up until the whole sentence is covered by a complete span. The dependency parse tree is recovered by backtracking the trace of forming the last complete span.

We apply a modified version of the Eisner algorithm in the second-order decoder. As shown in Figure 1(b), the basic structures used in second-order decoding are still complete spans and incomplete spans. The definitions of the spans remain the same except for the introduction of extra dependency arcs directing to them. The decoding procedure follows that of the original Eisner algorithm, but the index corresponding to the extra dependency arc results in the time complexity of the decoding algorithm increasing from $O(n^3)$ to $O(n^4)$.

4 Experiment setup

4.1 Dataset

We evaluate different combinations of encoders and decoders using datasets of six treebanks across languages with grammatical diversity: Penn Treebank (PTB), UD-Czech-CAC, UD-Russian, UD-Chinese, UD-Hebrew and UD-Swedish. Among these treebanks PTB consists of the English Wall Street Journal (WSJ) corpus (Marcus et al., 1993), annotated by Stanford Dependency (SD) (De Marneffe & Manning, 2008). The other treebanks are corpora annotated in the Universal Dependency (UD) v2.0 (Nivre et al., 2016).

Considering the fact that the second-order decoding time complexity of $O(n^4)$ makes the decoding procedure extremely slow when encountering long sentences, for efficiency purpose we prune all the sentences longer than 40 in training data. The numbers of remaining sentences after pruning are shown in Table 1. It can be seen that the treebanks we choose vary significantly in training data sizes. As discussed later in the paper, the training data size is a very important factor that determines the effectiveness of different encoder and decoder combinations.

4.2 Parser

We implement our parsers with different encoder and decoder configurations in PyTorch (Paszke et al., 2017). Our setting of hyperparameters in and network configuration and training generally follows that of Kiperwasser & Goldberg (2016) with minor modification to the batch size and the number of LSTM hidden units. We train our parsers for 40 epochs with the batch size of 10 sentences. The dimensions for word embedding, POS tag embedding and the number of LSTM hidden units are set respectively as to 100, 25 and 200. We adopt the Adam method for parameter optimization with the default hyperparameter setting in PyTorch.

4.3 Evaluation

There are two major evaluation metrics for dependency parsing: Unlabeled Attachment Score (UAS), the percentage of dependency heads being correctly predicted regardless of the dependency labels, and Labelled Attachment Score (LAS) the percentage of labeled dependency arc being correctly predicted. In this paper we focus our evaluation on UAS, because label prediction is typically independent from the decoding procedure, and therefore shall not be used to evaluate encoder-decoder combinations. Such evaluation criterion has also been adopted by previous works (Zhang & Zhao, 2015).

	LSTM-1+FO	LSTM-1+SO	LSTM-2+FO	LSTM-2+SO	Non-N+FO	Non-N+SO
pt	91.84	91.78	92.58	**92.68**	91.78	92.31
cs	87.59	87.62	87.76	**88.11**	86.19	87.16

Table 2: The UAS results of running different encoder-decoder combinations with the two largest treebanks, the best results are shown in bold. Here we denote pt = Penn Treebank, cs = UD-Czech-CAC, LSTM-n = n-layer LSTM encoder, Non-N = non-neural encoder, FO= first order decoder, SO = second order decoder.

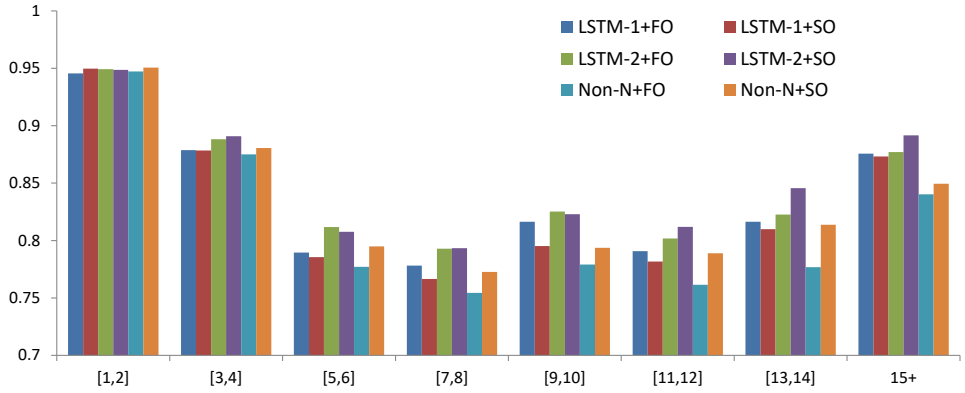


Figure 2: UAS of different encoder-decoder combinations relative dependency length

5 Results and Analysis

5.1 Large training data

It is widely believed that the more complex a model is, the more data it requires in parameter learning. Considering the complexity brought by combining neural encoders and high-order decoders, we start our evaluations from large training datasets (more than 20000 sentences) from which we expect all combinations will be effectively learned. Two treebanks meet the above require requirement: PTB and UD-Czech-CAC. In Table 2 we show the UAS results of running different encoder-decoder combinations on the two treebanks.

From the results, we see that with the decoder in the same order, increasing the complexity of the encoder generally leads to better performance. On the other hand, with the same encoder in most cases higher order decoder always outperforms the first-order decoder. However, for the non-neural encoder, the advantage of second order decoding seems more pronounced for that for the neural encoders, which implies that the introduction of powerful LSTM encoders does diminish the usefulness of high-order decoders to some extent, we investigate this assumption in following analysis.

We further study the performance of different encoder-decoder combinations on dependencies of different lengths.

The dependency length is defined as the number of tokens between the dependency head and its child (with the head itself also being counted). In Figure 2 we show the parsing accuracy of each encoder-decoder combination on gold dependencies of different dependency lengths in histogram (averaged over two treebanks). From the figure we see that the neural encoders generally outperform non-neural ones except when the dependency length is as short as 1 or 2. It is interesting to note the gap in parsing accuracy between **LSTM-2+FO** and **LSTM-2+SO** which becomes significant when the dependency length is longer than 10. We attribute this phenomenon to the inferiority of the first-order decoder in finding very long-term dependencies, even when equipped with powerful 2-layer LSTM encoder.

	LSTM-1+FO	LSTM-1+SO	LSTM-2+FO	LSTM-2+SO	Non-N+FO	Non-N+SO
pt(1/10)	87.68	87.80	**88.84**	88.54	88.09	88.79
cs(1/10)	81.01	81.76	82.58	**82.92**	82.08	82.78
ru	81.25	80.82	81.79	**82.17**	80.39	81.24
zh	79.78	79.84	80.86	**81.25**	76.28	78.65
he	82.92	82.67	**84.96**	84.68	82.81	83.72
sv	84.49	84.91	85.77	**86.16**	84.24	85.26

Table 3: The UAS results of running different encoder-decoder combinations with medium sized training data. The best results are shown in bold. Here ru = UD-Russian, zh = UD-Chinese, he = UD-Heberew, sv = UD-Swedish, 1/10 denotes that only 1/10 of the training data in this treebank is used for training, the other denotations are the same as in Table 2.

	LSTM-1-FO	LSTM-1-SO	LSTM-2-FO	LSTM-2-SO	Non-N-FO	Non-N-SO
pt(1/100)	78.85	78.55	80.17	78.88	81.02	**82.20**
cs(1/100)	71.75	68.83	72.22	70.53	76.26	**76.37**
ru(1/10)	73.92	72.46	74.75	73.03	74.88	**75.45**
zh(1/10)	69.30	69.65	**70.58**	70.21	67.42	68.58
he(1/10)	76.02	75.87	77.14	**77.24**	76.61	77.11
sv(1/10)	75.55	75.85	77.64	76.42	77.68	**78.74**

Table 4: The UAS results of running different encoder-decoder combinations with small training data. The best reults are shown in bold.

5.2 Medium sized training data results

For most languages, an annotated corpus in the size of PTB is unavailable. Among the treebanks collected by Universal Dependency (UD), training data size sets are commonly medium sized and restricted to thousands of sentences. We do evaluation on the four medium sized treebanks that we selected. Besides, to show the changes caused by training data sizes, we also do experiments with 1/10 of PTB and UD-Czech-CAC. We randomly sample 1/10 of the two treebanks for four times and report the average results.

We show the full results of evaluating all six combinations of encoders and decoders over the six treebanks in Table 3. Compared to the results in Table 2, the trend that the non-neural encoder being outperformed by the neural encoders regardless of the decoder order remains obvious, PTB(1/10) is an exception, for which, non-neural encoder is able to surpass the neural encoders when the decoder is set to second order. A possible explanation is that PTB has a fine-grained POS tag set containing more than 50 tags (compared with less than 20 tags in UD) and the fine-grained tag set can still provide informative features for the non-neural encoder while the neural encoder is negatively impacted by the decrease of the training data. Another noticeable change is that in some treebanks **LSTM-2+FO** replaces **LSTM-2+SO** as the best performing combination. It may also be a sign of the effe decreased training data on combinations of encoders and decoders with more complexity.

5.3 Small Training data results

Finally, we repeat our experiments in a low-resource setting, with a training set containing only a few hundred sentences. Due to the skill and labour required for treebank annotation, most of the existing languages in the world are low-resource in terms of treebanks. Studying the performance of NLP systems on low-resource corpora has drawn a lot of attention recently.

Again, we reduce the training data size by randomly picking sentences from the original training data. For UD-Russian, UD-Chinese, UD-Hebrew and UD-Swedish, we sample 1/10 of the original training data; for PTB and UD-Czech-CAC, we sample 1/100. For each treebank, we sample four random subsets and report the average results.

We give the full experimental results with small training data in Table 4, It can be seen that the results are significantly different from the previous results. The dominance of neural encoders no longer exists. Instead the combination of the non-neural encoder with the second order decoder shows superior performance with four out of six treebanks and competitive performance with one

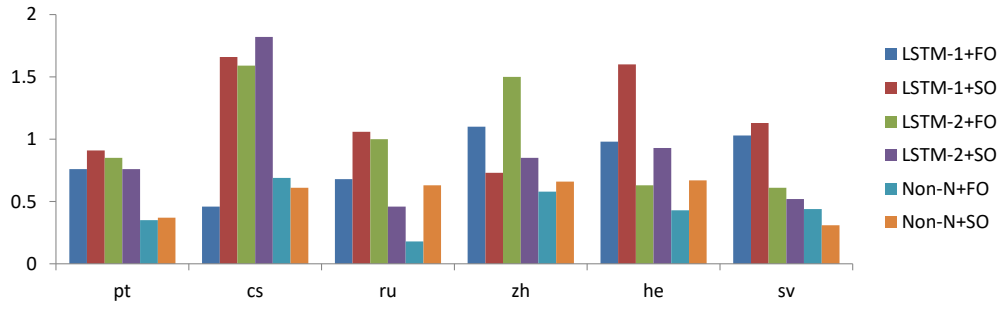
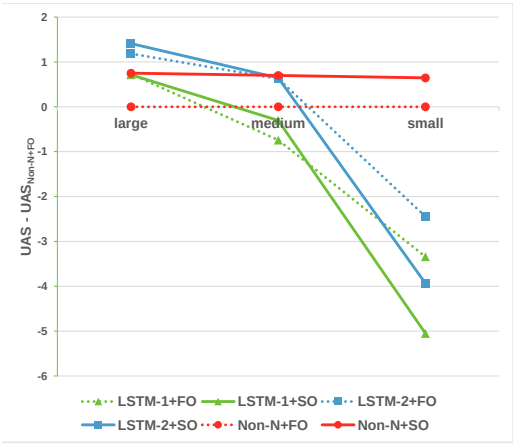
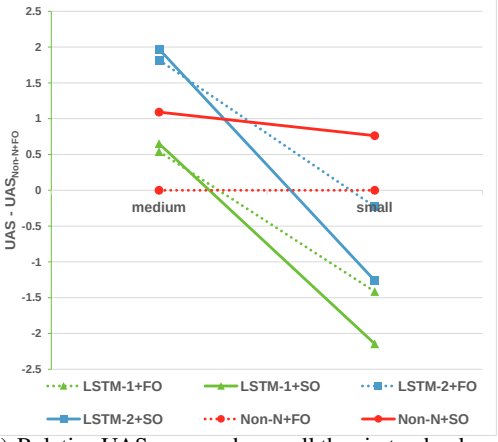


Figure 3: Standard deviation in UAS of four runs with small training data



(a) Relative UAS averaged over PTB and UD-Czech-CAC with all three training data sizes.



(b) Relative UAS averaged over all the six treebanks with medium and small data.

Figure 4: Average UAS relative to the combination of the non-neural encoder and the first-order decoder.

treebank. We speculate that this is due to the better data-efficiency of the non-neural encoder in comparison with the more data-hungry neural encoders.

Besides the parsing accuracy we also evaluate the standard deviation of 4 runs to examine the robustness of encoder-decoder combinations to data variation. We illustrate the results in Figure 3. It can be seen that the neural encoders have higher variation while the non-neural encoders remains relatively stable.

5.4 Summary of Results

Figure 4 shows, for each combination, the relative UAS averaged over treebanks with the baseline being the simplest combination, the non-neural encoder plus the first-order decoder.

We make the following observations:

- The LSTM encoders are better than the non-neural encoder with large data, but are significantly worse than the non-neural encoder with small data. There is no clear winner between the two types of encoders with medium data. These observations show that the LSTM encoders are clearly more data-hungry than the non-neural encoder.

- The one-layer LSTM encoder is consistently worse than the two-layer LSTM encoder. It seems that the number of LSTM layers may have limited impact on the data-efficiency of LSTM encoders.
- When combined with the non-neural encoder, the second-order decoder is consistently better than the first-order decoder, regardless of the data size.
- When combined with the LSTM encoders, the second-order decoder is slightly better than the first-order decoder with large or medium data, but is significantly worse than the first-order decoder with small data. Based on this and the previous observation, it seems that the data-efficiency of the first- and second-order decoders may depend on the encoder. The reason behind this phenomenon is still unclear to us and it may require further experimentation and analysis to find out.

6 CONCLUSIONS

We empirically evaluate the combinations of neural and non-neural encoders with first- and second-order decoders on six treebanks with varied data sizes. The results suggest that with sufficiently large training data (a few tens of thousands of sentences), one should use a neural encoder and perhaps a high-order decoder to achieve the best parsing accuracy; but with small training data (a few hundred sentences), one should use a traditional non-neural encoder plus a high-order decoder. Possible future work includes experimenting with second-order sibling decoding, third-order decoding, neural encoders with biaffine and triaffine score computation, and finally transition-based dependency parsers.

REFERENCES

Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. Globally normalized transition-based neural networks. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*, 2016.

Xavier Carreras. Experiments with a higher-order projective dependency parser. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, 2007.

Yoeng-Jin Chu. On the shortest arborescence of a directed graph. *Scientia Sinica*, 14:1396–1400, 1965.

Marie-Catherine De Marneffe and Christopher D Manning. Stanford typed dependencies manual. Technical report, Technical report, Stanford University, 2008.

Timothy Dozat and Christopher D. Manning. Deep biaffine attention for neural dependency parsing. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

Jack Edmonds. Optimum branchings. *Journal of Research of the national Bureau of Standards B*, 71(4):233–240, 1967.

Jason Eisner. Three new probabilistic models for dependency parsing: An exploration. In *16th International Conference on Computational Linguistics, Proceedings of the Conference, COLING 1996, Center for Sprogteknologi, Copenhagen, Denmark, August 5-9, 1996*, pp. 340–345, 1996.

Agnieszka Falenska and Jonas Kuhn. The (non-)utility of structural features in bilstm-based dependency parsers. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pp. 117–128, 2019.

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

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

Eliyahu Kiperwasser and Yoav Goldberg. Simple and accurate dependency parsing using bidirectional LSTM feature representations. *TACL*, 4:313–327, 2016.

Sandra Kübler, Ryan McDonald, and Joakim Nivre. Dependency parsing. *Synthesis Lectures on Human Language Technologies*, 1(1):1–127, 2009.

Xuezhe Ma, Zecong Hu, Jingzhou Liu, Nanyun Peng, Graham Neubig, and Eduard H. Hovy. Stack-pointer networks for dependency parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, pp. 1403–1414, 2018. doi: 10.18653/v1/P18-1130.

Mitchell Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. 1993.

Ryan T. McDonald and Fernando C. N. Pereira. Online learning of approximate dependency parsing algorithms. In *EACL 2006, 11st Conference of the European Chapter of the Association for Computational Linguistics, Proceedings of the Conference, April 3-7, 2006, Trento, Italy*, 2006.

Joakim Nivre, Marie-Catherine De Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajic, Christopher D Manning, Ryan T McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira, et al. Universal dependencies v1: A multilingual treebank collection. In *LREC*, 2016.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

Tianze Shi and Lillian Lee. Valency-augmented dependency parsing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pp. 1277–1291, 2018.

David Weiss, Chris Alberti, Michael Collins, and Slav Petrov. Structured training for neural network transition-based parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*, pp. 323–333, 2015.

Zhisong Zhang and Hai Zhao. High-order graph-based neural dependency parsing. In *Proceedings of the 29th Pacific Asia Conference on Language, Information and Computation, PACLIC 29, Shanghai, China, October 30 - November 1, 2015*, 2015.