Towards Better UD Parsing: Deep Contextualized Word Embeddings, Ensemble, and Treebank Concatenation

Wanxiang Che, Yijia Liu, Yuxuan Wang, Bo Zheng, Ting Liu
Research Center for Social Computing and Information Retrieval
Harbin Institute of Technology, China
{car,yjliu,yxwang,bzheng,tliu}@ir.hit.edu.cn

Abstract

This paper describes our system (HIT-SCIR) submitted to the CoNLL 2018 shared task on Multilingual Parsing from Raw Text to Universal Dependencies. We base our submission on Stanford's winning system for the CoNLL 2017 shared task and make two effective extensions: 1) incorporating deep contextualized word embeddings into both the part of speech tagger and parser; 2) ensembling parsers trained with different initialization. We also explore different ways of concatenating treebanks for further improvements. Experimental results on the development data show the effectiveness of our methods. In the final evaluation, our system was ranked first according to LAS (75.84%) and outperformed the other systems by a large margin.

1 Introduction

In this paper, we describe our system (HIT-SCIR) submitted to CoNLL 2018 shared task on Multilingual Parsing from Raw Text to Universal Dependencies (Zeman et al., 2018). We base our system on Stanford's winning system (Dozat et al., 2017, §2) for the CoNLL 2017 shared task (Zeman et al., 2017).

Dozat and Manning (2016) and its extension (Dozat et al., 2017) have shown very competitive performance in both the shared task (Dozat et al., 2017) and previous parsing works (Ma and Hovy, 2017; Shi et al., 2017a; Liu et al., 2018b; Ma et al., 2018). A nature question that raises is how can we further improve their part of speech (POS) tagger and parser via simple yet effective technique. In our system, we make two noteworthy extensions to their parser and tagger, which includes

- Incorporating the deep contextualized word embeddings (Peters et al., 2018, ELMo) into the word representaton (§3);
- Ensembling parsers trained with different initialization (§4).

Both these two extensions result in substantial improvements.

For some languages in the shared task, multiple treebanks of different domains are provided. Treebanks which are of the same language families are provided as well. Letting these treebanks help each other has been shown an effective way to improve parsing performance in both the cross-lingual-cross-domain parsing community and last year's shared tasks (Ammar et al., 2016; Guo et al., 2015; Che et al., 2017; Shi et al., 2017b; Björkelund et al., 2017). In our system, we apply the simple concatenation to the treebanks that are potentially helpful to each other and explore different ways of concatenation to improve the parser's performance (§5).

In dealing with the small languages and low-resource languages (§6), we adopt the word embedding transfer idea in the cross-lingual dependency parsing (Guo et al., 2015) and use the bilingual word vectors transformation technique (Smith et al., 2017)[1] to map *fasttext*[2] word embeddings (Bojanowski et al., 2016) of the source rich-resource language and target low-resource language into the same space. The transferred parser trained on the source language is used for the target low-resource language.

We conduct experiments on the development data to study the effects of ELMo, parser ensemble, and treebank concatenation. Experimental re-

[1] https://github.com/Babylonpartners/fastText_multilingual

[2] https://github.com/facebookresearch/fastText

sults show that these techniques substantially improve the parsing performance. Using these techniques, our system achieved an averaged LAS of 75.84 on the official test set and was ranked the first according to LAS (Zeman et al., 2018). This result outperforms the others by a large margin.

2 Deep Biaffine Parser

We based our system on the tagger and parser of Dozat et al. (2017). The core idea of the tagger and parser is using an LSTM network to produce the vector representation for each word and then predict POS tags and dependency relations using the representation. For the tagger whose input is the word alone, this representation is calculated as

$$\begin{aligned} \mathbf{r}_i &= \text{BiLSTM}(\mathbf{r}_0, (\mathbf{v}_1^{(word)}, ..., \mathbf{v}_n^{(word)}))_i \\ \mathbf{h}_i, \mathbf{c}_i &= \text{split}(\mathbf{r}_i), \end{aligned}$$

where $\mathbf{v}_i^{(word)}$ is the word embeddings. After getting $\mathbf{h}_i$, the scores of tags are calculated as

$$\begin{aligned} \mathbf{h}_i^{(pos)} &= \text{MLP}^{(pos)}(\mathbf{h}_i) \\ \mathbf{s}_i^{(pos)} &= W \cdot \mathbf{h}_i^{(pos)} + \mathbf{b}^{(pos)} \\ y_i^{(pos)} &= \operatorname*{argmax}_j s_{i,j}^{(pos)} \end{aligned}$$

where each element in $\mathbf{s}_i^{(pos)}$ represents the possibility that i-th word is assigned with corresponding tag.

For the parser whose inputs are the word and POS tag, such representation is calculated as

$$\begin{aligned} \mathbf{x}_i &= \mathbf{v}_i^{(word)} \oplus \mathbf{v}_i^{(tag)} \\ \mathbf{r}_i &= \text{BiLSTM}(\mathbf{r}_0, (\mathbf{x}_1, ..., \mathbf{x}_n))_i \\ \mathbf{h}_i, \mathbf{c}_i &= \text{split}(\mathbf{r}_i). \end{aligned}$$

And a pair of representations are fed into a biaffine classifier to predict the possibility that there is a dependency arc between these two words. The scores over all head words are calculated as

$$\begin{aligned} \mathbf{s}_i^{(arc)} &= H^{(arc\text{-}head)} W^{(arc)} \mathbf{h}_i^{(arc\text{-}dep)} \\ &\quad + H^{(arc\text{-}head)} \mathbf{b}^{(arc)} \\ y^{(arc)} &= \operatorname*{argmax}_j s_{i,j}^{(arc)} \end{aligned}$$

where $\mathbf{h}_i^{(arc\text{-}dep)}$ is computed by feeding $\mathbf{h}_i$ into an MLP and $H^{(arc\text{-}head)}$ is the stack of $\mathbf{h}_i^{(arc\text{-}head)}$ which is calculated in the same way with $\mathbf{h}_i^{(arc\text{-}dep)}$ but using another MLP. After getting the head $y^{(arc)}$ word, its relation with i-th word decided by calculating

$$\begin{aligned} \mathbf{s}_i^{(rel)} &= \mathbf{h}_{y^{(arc)}}^{T(rel-head)} \mathbf{U}^{(rel)} \mathbf{h}_i^{(rel-dep)} \\ &\quad + W^{(rel)} (\mathbf{h}_i^{(rel-dep)} \oplus \mathbf{h}_{y^{(arc)}}^{T(rel-head)}) \\ &\quad + \mathbf{b}^{(rel)}, \\ y^{(rel)} &= \operatorname*{argmax}_j s_{i,j}^{(rel)} \end{aligned}$$

where $\mathbf{h}^{(rel-head)}$ and $\mathbf{h}^{(rel-dep)}$ are calculated in the same way with $\mathbf{h}_i^{(arc\text{-}dep)}$ and $\mathbf{h}_i^{(arc\text{-}head)}$.

This decoding process can lead to cycles in the result. (Dozat et al., 2017) employed an iterative fixing methods on the cycles. We encourage the reader of this paper refer their paper for more training and decoding details.

For both the biaffine tagger and parser, the word embedding $\mathbf{v}_i^{(word)}$ is obtained by summarizing a fine-tuned token embedding $\mathbf{w}_i$, a fixed word2vec embedding $\mathbf{p}_i$, and an LSTM-encoded character representation $\hat{\mathbf{v}}_i$ as

$$\mathbf{v}_i^{(word)} = \mathbf{w}_i + \mathbf{p}_i + \hat{\mathbf{v}}_i$$

3 Deep Contextualized Word Embeddings

Deep contextualized word embeddings (Peters et al., 2018, ELMo) has shown to be very effective on a range of syntactic and semantic tasks and it's straightforward to achieve by using an LSTM network to encode words in a sentence and training the LSTM network with language modeling objective on large-scale raw text. More specifically, the $\mathbf{ELMo}_i$ is computed by first computing the hidden representation $\mathbf{h}_i^{(LM)}$ as

$$\begin{aligned} \mathbf{r}_i^{(LM)} &= \text{BiLSTM}^{(LM)}(\mathbf{r}_0^{(LM)}, (\tilde{\mathbf{v}}_1, ..., \tilde{\mathbf{v}}_n))_i \\ \mathbf{h}_i^{(LM)}, \mathbf{c}_i^{(LM)} &= \text{split}(\mathbf{r}_i^{(LM)}), \end{aligned}$$

where $\tilde{\mathbf{v}}_i$ is the output of a CNN over characters, then attentively summarizing and scaling different layers of $\mathbf{h}_{i,j}^{(LM)}$ with s_j and γ as

$$\mathbf{ELMo}_i = \gamma \sum_{j=0}^{L} s_j \mathbf{h}_{i,j}^{(LM)},$$

where L is the number of layers and $\mathbf{h}_{i,0}^{(LM)}$ is identical to $\tilde{\mathbf{v}}_i$. In our system, we follow Peters et al.

(2018) and use a two-layer bidirectional LSTM as our BiLSTM$^{(LM)}$.

In this paper, we study the usage of ELMo for improving both the tagger and parser and make several simplifications. Different from Peters et al. (2018), we treat the output of ELMo as a fixed representation and do not tune its parameters during tagger and parser training. Thus, we cancel the layer-wise attention scores s_j and the scaling factor γ, which means

$$\mathbf{ELMo}_i = \sum_{j=0}^{2} \mathbf{h}_{i,j}^{(LM)}.$$

In our preliminary experiments, using $\mathbf{h}_{i,0}^{(LM)}$ for $\mathbf{ELMo}_i$ yields better performance on some treebanks. In our final submission, we decide whether using $\sum_{j=0}^{2} \mathbf{h}_{i,j}^{(LM)}$ or $\mathbf{h}_{i,0}^{(LM)}$ by the development performance.

After getting $\mathbf{ELMo}_i$, we project it to the same dimension as $\mathbf{v}_i^{(word)}$ and use it as an additional word embeddings. The calculation of $\mathbf{v}_i^{(word)}$ becomes

$$\mathbf{v}_i^{(word)} = \mathbf{w}_i + \mathbf{p}_i + \hat{\mathbf{v}}_i + W^{(ELMo)} \cdot \mathbf{ELMo}_i$$

for both the tagger and parser. We need to note that training the tagger and parser includes $W^{(ELMo)}$. To avoid overfitting, we impose a dropout function on projected vector $W^{(ELMo)} \cdot \mathbf{ELMo}_i$ during training.

4 Parser Ensemble

According to Reimers and Gurevych (2017), neural network training can be sensitive to initialization and Liu et al. (2018a) shows that ensemble neural network trained with different initialization leads to performance improvements. We follow their works and train three parsers with different initialization, then ensemble these parsers by averaging their softmaxed output scores as

$$\mathbf{s}_i^{(rel)} = \frac{1}{3} \sum_{m=1}^{3} \text{softmax}(\mathbf{s}_i^{(m,rel)})$$

.

5 Treebank Concatenation

For 15 out of the 58 languages in the shared task, multiple treebanks from different domains are provided. There are also treebanks that comes from the same language families. Taking the advantages of the relation between treebanks has been shown a promising direction in both the research community (Ammar et al., 2016; Guo et al., 2015, 2016) and in the CoNLL 2017 shared task (Che et al., 2017; Björkelund et al., 2017; Shi et al., 2017b). In our system, we adopt the treebank concatenation technique as (Ammar et al., 2016) but limit that only a group of treebanks from the same language (*cross-domain concatenation*) or a pair of treebanks that are typologically or geographically correlated (*cross-lingual concatenation*) is concatenated.

In our system, we tried cross-domain concatenation on *nl*, *sv*, *ko*, *it*, *en*, *fr*, *gl*, *la*, *ru*, and *sl*. We also tried cross-lingual concatenation on *ug-tr*, *uk-ru*, *ga-en*, and *sme-fi* following (Che et al., 2017). However, due to the variance in vocabulary, grammatical genre, and even annotation, treebank concatenation does not guarantee to improve the model's performance. We decide the usage of concatenation by examining their development set performance. For some small treebanks which do not have development set, whether using treebank concatenation is decided through 5-fold cross validation.[3] We show the experimental results of treebank concatenation in Section 9.3.

target	br	fo	hy	kk	bxr	kmr	hsb	th
source	ga	no	et	tr	hi	fa	pl	zh

Table 1: Cross-lingual transfer settings for low-resource target languages.

6 Low Resources Languages

In the shared task, 5 languages are presented with training set of less than 50 sentences. 4 languages don't even have any training data. It's difficult to train reasonable parser on these low-resource languages. We deal with these treebanks by adopting the word embedding transfer idea of Guo et al. (2015). We transfer the word embeddings of the rich-resource language to the space of low-resource language using the bilingual word vectors transformation technique (Smith et al., 2017) and trained a parser using the source treebank with only pretrained word embeddings on the transformed space as $\mathbf{v}_i^{(word)} = \mathbf{p}_i$. The transforma-

[3] We use *udpipe* for the experiments of these part because we consider the effect of treebank concatenation as being irrelevant to the parser architecture and *udpipe* has the speed advantage in both training and testing.

tion matrix is automatically learned on the *fasttext* word embedding using the same tokens between two languages (like punctuations).

Table 1 shows our source languages for the target low-resource languages. For the treebank with a few training data, its source language is decided by testing the source parser's performance on the training data.[4] For the treebank without any training data, we choose the source language according to their language family.

Naija presents an exception for our method since it doesn't have *fasttext* word embeddings and embedding transformation is infeasible. Since it's a dialect of English, we use the full pipeline of *en_ewt* for *pcm_nsc*.

7 Preprocessing

Besides improving the tagger and parser, we also consider the preprocessing as an important factor to the final performance try to improve it by using the state-of-the-art system for sentence segmentation, and developing our own word segmentor for languages whose tokenizations are non-trival.

7.1 Sentence Segmentation

For some treebanks, sentence segmentation can be problematic since there is no explicitly sentence delimiters. de Lhoneux et al. (2017) and Shao (2017) presented a joint tokenization and sentence segmentation model[5] that outperformed the baseline model in last year's shared task (Zeman et al., 2017). We select a set of treebanks whose *udpipe* sentence segmentation F-scores are lower than 95 on the development set and use Uppsala segmentor instead.[6] Using the Uppsala segmentor leads to a development improvement of 7.67 F-score in these treebanks over *udpipe* baseline and it was ranked first according to sentence segmentation in the final evaluation.

7.2 Tokenization for Chinese, Japanese, and Vietnamese

Tokenization is non-trivial for languages which do not have explicit word boundary markers, like Chinese, Japanese, and Vietnamese. We develop our own tokenizer for these three languages.[7] Following Che et al. (2017) and Zheng et al. (2017), we model the tokenization as labeling the word boundary tag on character and use features derived from large-scale unlabeled data to further improve the performance.[8] In addition to the pointwise mutual information (PMI), we also incorporate the character ELMo into our tokenizer. These techniques lead to the best tokenization performance on all the related treebanks and the average improvement over *udpipe* baseline is 7.5 in tokenization F-score.[9]

7.3 Preprocessing for Thai

Thai language presents unique challenge in the preprocessing. Our survey on the Thai Wikipedia indicates that there is no explicit sentence delimiter for Thai language and obtaining Thai words requires tokenization. To this remedy, we use space as sentence delimiter and use the lexicon-based word segmentation – forward maximum matching algorithm for Thai tokenization. Our lexicon is derived from the *fasttext* word embeddings.

7.4 Lemmatization and Morphology Tagging

We did not make an effort on lemmatization and morphology tagging, but only use the baseline model. This lags our performance in the MLAS and BLEX evaluation, in which we were ranked 6th and 2nd correspondingly. However, since our method, especially incorporating ELMo, is not limited to particular task, we expect it to improve both the lemmatization and morphology tagging and achieve better MLAS and BLEX scores.

8 Implementation Details

8.1 Pretrained Word Embeddings

We use the 100-dimensional pretrained word embeddings released by the shared task for the large languages. For the small languages and low-resource languages where cross-lingual transfer is required, we use the 300-dimensional *fasttext* word embeddings. *fro_srcmf* presents the only exceptions and we use the French embeddings instead. For all the embeddings, we filter the top 10% frequent words.

[4] We use *udpipe* for this test.

[5] `https://github.com/yanshao9798/segmenter/`, noted as Uppsala segmentor henceforth.

[6] We use Uppsala segmentor for *it_postwita*, *got_proiel*, *la_poroiel*, *cu_proiel*, *grc_proiel*, *sl_ssj*, *nl_lassysmall*, *fi_tdt*, *pt_bosque*, *da_ddt*, *id_gsd*, *el_gdt*, and *et_edt*.

[7] note as SCIR segmentor henceforth.

[8] For Vietnamese where whitespaces occur both inter- and intra-words, we treat the whitespace-separated token as a character.

[9] *ja_gsd*, *ja_modern*, *vi_vtb*, and *zh_gsd*.

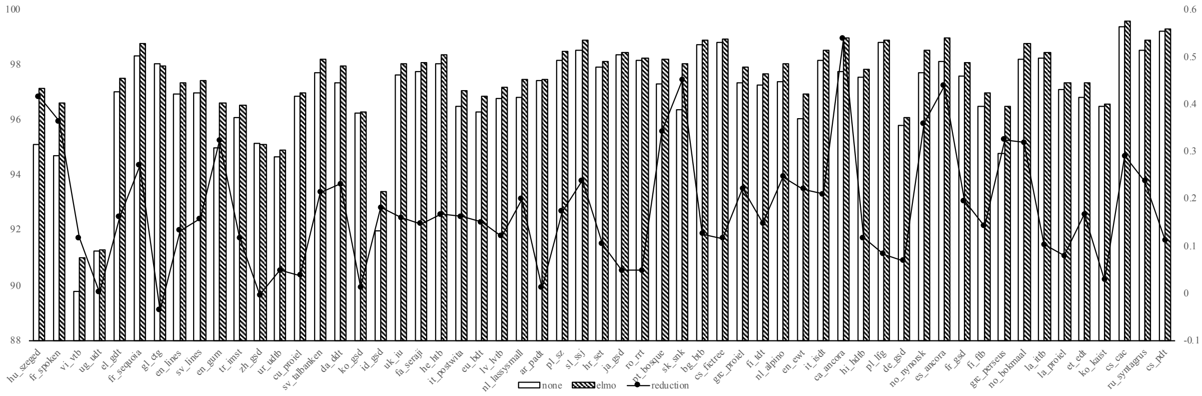

Figure 1: The effects of ELMo on POS tagging. Treebanks are sorted according to the number of training sentences from left to right.

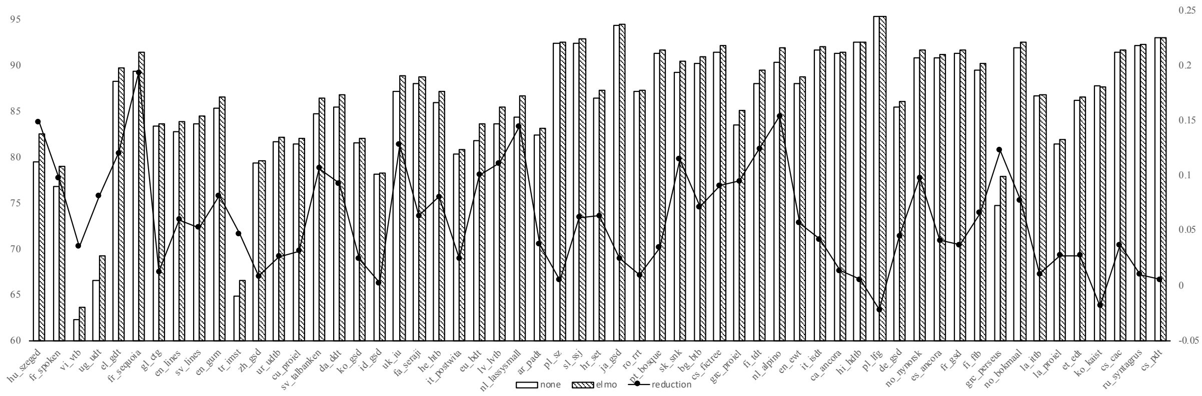

Figure 2: The effects of ELMo on parsing. Treebanks are sorted according to the number of training sentences from left to right.

8.2 ELMo

We use the same hyperparameter settings as Peters et al. (2018) for BiLSTM$^{(LM)}$ and the character CNN. We train their parameters as training a bidirectional language model on a randomly sampled subset of raw text released by the shared task. More specifically, for each language, we randomly sample 20 million words. Adam optimizer (Kingma and Ba, 2014) with default settings are used. The training of ELMo on one language takes roughly 3 days on an NVIDIA P100 GPU.

8.3 Biaffine Parser

We use the same hyperparameter settings as Dozat et al. (2017). When trained with ELMo, we use a dropout of 33% on the projected vectors.

8.4 SCIR Segmentor

We use a 50-dimensional character bigram embeddings as input and single layer LSTM for the sequence labeling model. The character ELMo is trained in the same way with the word ELMo. The final model is an ensemble of five single segmentors.

8.5 Uppsala Segmentor

We use the default settings for the Uppsala segmentor and the final model is an ensemble of three single segmentors.

9 Results

9.1 Effects of ELMo

We study the effect of ELMo on the large treebanks and report the results of singe tagger and parser with and without ELMo. Figure 1 shows the tagger results on the development set and Figure 2 shows the parser results. Using ELMo in the tagger leads to a macro-averaged improvement of 0.56% in UPOS and the macro-averaged error reduction is 17.83%. Using ELMo in the parser leads to a macro-averaged improvement of 0.84% in LAS and the macro-averaged error reduction is

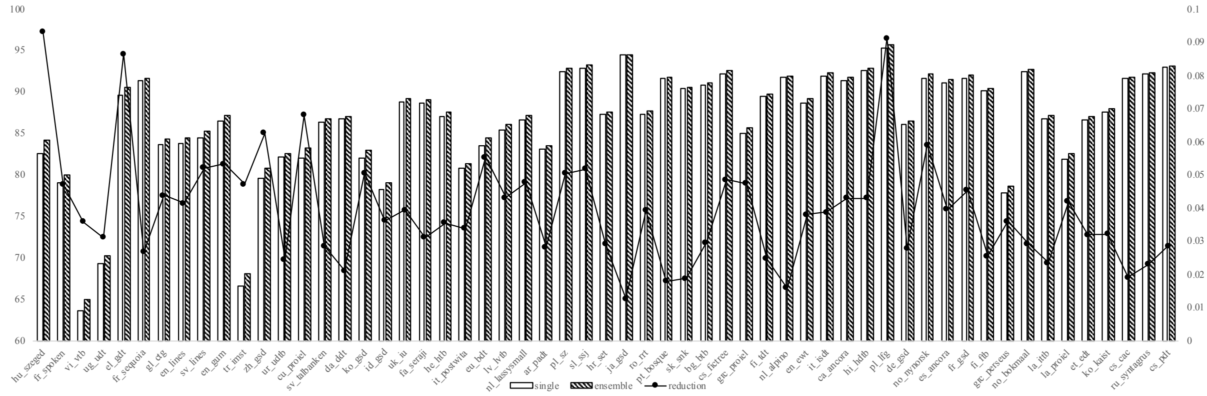

Figure 3: The effects of ensemble on parsing. Treebanks are sorted according to the number of training sentences from left to right.

7.88%.

ELMo improves the tagging performance almost on every treebank, except for *zh_gsd* and *gl_ctg*. Similar trends are witnessed in the parsing experiments with *ko_kaist* and *pl_lfg* being the only treebanks that ELMo slightly worsens the performance.

We also study the relative improvements against the size of the treebank. The line in Figure 1 and Figure 2 shows the error reduction from using ELMo on each treebank. However, no clear relation is revealed between the treebank size and the gains using ELMo.

9.2 Effects of Ensemble

We also test the effect of ensemble and show the results in Figure 3. Parser ensemble leads to an averaged improvement of 0.55% in LAS and the averaged error reduction is 4.0%. These results indicate that ensemble is an effective way to improve the parsing performance. The relationship between gains using ensemble and treebank size is also studied in this figure and the trend is that small treebank benefit more from ensemble. We address this to the fact the ensemble improve the model's generalization ability in which the parser trained on small treebank is weak due to overfitting.

9.3 Effects of Treebank Concatenation

As mentioned in Section 5, we study the effects of both the *cross-domain concatenation* and *cross-lingual concatenation*.

Cross-Domain Concatenation. For the treebanks which have development set, the development performances are shown in Table 2. Numbers of sentences in the training set are also shown in this table. The general trend is that for the treebank with small training set, cross-domain concatenation achieves better performance. While for those with large training set, concatenation doesn't improve the performance or even worsen the results.

For the small treebanks which do not have development set, the 5 fold cross validation results are shown in Table 3 in which concatenation improves most of the treebanks except for *gl_treegal*.

Cross-Lingual Concatenation. The experimental results of cross-lingual concatenation are shown in Table 4. Unfortunately, concatenating treebanks from different languages only achieves improved performance on *uk_iu*. This results also indicate that in cross lingual parsing, sophisticated methods like word embeddings transfer (Guo et al., 2015) and treebank transfer (Guo et al., 2016) are still necessary.

9.4 Effects of Better Preprocessing

We also study how preprocessing contributes to the final parsing performance. The experimental results on the development set are shown in Table 5. From this table, the performance of word segmentation is almost linearly correlated with the final performance. Similar trends on sentence segmentation performance are witnessed but *el_gdt* and *pt_bosque* presents some exceptions where better preprocess leads drop in the final parsing performance.

9.5 Time and Memory Consumption

A full run over the 82 test sets on the TIRA virtual machine (Potthast et al., 2014) takes about 40

nl	apino	lassysmall
# train	12.2	5.8
apino	91.87	
lassysmall		86.82
concat.	**92.08**	**89.34**

sv	lines	talbanken
# train	2.7	4.3
lines	84.64	
talbanken		86.39
concat.	**85.76**	**86.77**

ko	gsd	kaist
# train	4.4	23.0
gsd	82.05	
kaist		**87.83**
concat.	**83.73**	87.61

it	isdt	postwita
# train	13.1	5.4
isdt	**92.01**	
postwita		80.79
concat.	91.80	**82.54**

en	ewt	gum	lines
# train	12.5	2.9	2.7
ewt	**88.75**		
gum		**86.52**	
lines			83.86
concat.	88.74	85.65	**85.30**

fr	gsd	sequoia	spoken
# train	14.6	2.2	1.2
gsd	**91.64**		
sequoia		**91.44**	
spoken			79.06
concat.	91.44	90.51	**81.99**

Table 2: The developement performance with cross-domain concatenation for languages which has multiple treebanks. *# train* shows the number of training sentences in the treebank measured in thousand. We opt out *cs*, *fi*, and *pl* because all the treebanks of these languages are relatively large (# train > 10).

gl	treegal
# train	0.6
treegal	**66.71**
+ctg	56.73

la	perseus
# train	1.3
perseus	44.05
+proiel	**50.78**

no	nynorsklia
# train	0.3
nynorsklia	51.05
+nynorsk	**58.49**

ru	taiga
# train	0.9
taiga	54.70
+syntagrus	**60.75**

sl	sst
# train	2.1
sst	55.15
+ssj	**59.52**

Table 3: The 5-fold cross validation results for the cross-domain concatenation for treebank which doesn't have development set.

hours and consumes about 4G RAM memory.

10 Conclusion

Our system submitted to the CoNLL 2018 shared task made several improvements on last year's winning system from Dozat et al. (2017), including incorporating deep contextualized word embeddings, parser ensemble, and treebank concatenation. Experimental results on the development set show the effectiveness of our methods. Using these techniques, our system achieved an averaged LAS of 75.84% and obtained the first place in LAS in the final evaluation.

11 Credits

There are a few references we would like to give proper credit, especially to data providers: the core Universal Dependencies paper from LREC 2016 (Nivre et al., 2016), the UD version 2.2 datasets (Nivre et al., 2018), the baseline *udpipe* model released by Straka et al. (2016), the deep contextualized word embeddings code released by Peters et al. (2018), the biaffine tagger and parser released by Dozat et al. (2017), the joint sentence segmentor and tokenizer released by de Lhoneux et al. (2017), and the evaluation platform TIRA (Potthast et al., 2014).

Acknowledgments

This work was supported by the National Key Basic Research Program of China via grant 2014CB340503 and the National Natural Science Foundation of China (NSFC) via grant 61300113 and 61632011.

References

Waleed Ammar, George Mulcaire, Miguel Ballesteros, Chris Dyer, and Noah Smith. 2016. Many languages, one parser. *TACL* 4.

Anders Björkelund, Agnieszka Falenska, Xiang Yu, and Jonas Kuhn. 2017. Ims at the conll 2017 ud shared task: Crfs and perceptrons meet neural networks. In *Proc. of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*. http://www.aclweb.org/anthology/K17-3004.

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2016. Enriching word vectors with subword information. *CoRR* abs/1607.04606. http://arxiv.org/abs/1607.04606.

Wanxiang Che, Jiang Guo, Yuxuan Wang, Bo Zheng, Huaipeng Zhao, Yang Liu, Dechuan Teng, and Ting Liu. 2017. The hit-scir system for end-to-end parsing of universal dependencies. In *Proc. of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*. http://www.aclweb.org/anthology/K17-3005.

Miryam de Lhoneux, Yan Shao, Ali Basirat, Eliyahu Kiperwasser, Sara Stymne, Yoav Goldberg, and

	ug_udt		uk_iu		ga_idt		sme_giella
ug_udt	**69.27**	uk_iu	88.84	ga_idt	**62.84**	sme_giella	**66.33**
+tr_imst	19.27	+ru_syntagus	**90.74**	+en_ewt	51.00	+fi_ftb	59.86

Table 4: Cross-lingual concatenation results. The results for *ug_udt* and *uk_iu* are obtained on the development set. The results for *ga_idt* and *sme_giella* are obtained with *udpipe* by 5-fold cross validation.

	Δ-sent.	*udpipe*	*improved*
fi_tdt	+0.69	88.13	**88.67**
et_edt	+1.22	86.33	**86.36**
nl_lassysmall	+1.39	88.08	**88.60**
da_ddt	+1.56	86.21	**86.51**
el_gdt	+1.57	**90.08**	89.96
cu_proiel	+1.72	72.79	**74.04**
pt_bosque	+1.83	**90.73**	90.20
id_gsd	+2.46	74.14	**78.83**
la_proiel	+4.82	73.21	**74.22**
got_proiel	+5.36	67.55	**68.40**
grc_proiel	+5.86	79.67	**80.72**
sl_ssj	+18.81	88.43	**92.27**
it_postwita	+30.40	74.91	**79.26**
	Δ-word	*udpipe*	*improved*
ja_gsd	+4.07	80.53	**85.23**
zh_gsd	+7.16	66.16	**75.78**
vi_vtb	+9.02	48.58	**57.53**

Table 5: The effect of improved preprocessing. Δ-sent. shows the relative sentence segmentation improvement of the *improved* preprocess compared against *udpipe*. Δ-word shows the word segmentation improvement.

Joakim Nivre. 2017. From raw text to universal dependencies - look, no tags! In *Proc. of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*. http://www.aclweb.org/anthology/K17-3022.

Timothy Dozat and Christopher D. Manning. 2016. Deep biaffine attention for neural dependency parsing. *CoRR* abs/1611.01734.

Timothy Dozat, Peng Qi, and Christopher D. Manning. 2017. Stanford's graph-based neural dependency parser at the conll 2017 shared task. In *Proc. of CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*.

Jiang Guo, Wanxiang Che, Haifeng Wang, and Ting Liu. 2016. A universal framework for inductive transfer parsing across multi-typed treebanks. In *Proc. of Coling*. http://www.aclweb.org/anthology/C16-1002.

Jiang Guo, Wanxiang Che, David Yarowsky, Haifeng Wang, and Ting Liu. 2015. Cross-lingual dependency parsing based on distributed representations. In *Proc. of ACL*.

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *CoRR* abs/1412.6980.

Yijia Liu, Wanxiang Che, Huaipeng Zhao, Bing Qin, and Ting Liu. 2018a. Distilling knowledge for search-based structured prediction. *CoRR* abs/1805.11224. http://arxiv.org/abs/1805.11224.

Yijia Liu, Yi Zhu, Wanxiang Che, Bing Qin, Nathan Schneider, and Noah A. Smith. 2018b. Parsing tweets into universal dependencies. In *Proc. of NAACL*. http://aclweb.org/anthology/N18-1088.

Xuezhe Ma and Eduard Hovy. 2017. Neural probabilistic model for non-projective mst parsing. In *Proc. of IJCNLP*. http://www.aclweb.org/anthology/I17-1007.

Xuezhe Ma, Zecong Hu, Jingzhou Liu, Nanyun Peng, Graham Neubig, and Eduard H. Hovy. 2018. Stack-pointer networks for dependency parsing. *CoRR* abs/1805.01087. http://arxiv.org/abs/1805.01087.

Joakim Nivre, Marie-Catherine de Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajič, Christopher Manning, Ryan McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira, Reut Tsarfaty, and Daniel Zeman. 2016. Universal Dependencies v1: A multilingual treebank collection. In *Proc. of LREC-2016*.

Joakim Nivre et al. 2018. Universal Dependencies 2.2. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics, Charles University, Prague, `http://hdl.handle.net/11234/SUPPLYTHENEWPERMANENTIDHERE!`

Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In *Proc. of NAACL*. http://aclweb.org/anthology/N18-1202.

Martin Potthast, Tim Gollub, Francisco Rangel, Paolo Rosso, Efstathios Stamatatos, and Benno Stein. 2014. Improving the reproducibility of PAN's shared tasks: Plagiarism detection, author identification, and author profiling. In Evangelos Kanoulas, Mihai Lupu, Paul Clough, Mark Sanderson, Mark Hall, Allan Hanbury, and Elaine Toms, editors, *Information Access Evaluation meets Multilinguality, Multimodality, and Visualization. 5th International Conference of the CLEF Initiative (CLEF 14)*. Springer, Berlin Heidelberg New York, pages 268–299. https://doi.org/10.1007/978-3-319-11382-1_22.

Nils Reimers and Iryna Gurevych. 2017. Reporting score distributions makes a difference: Performance study of lstm-networks for sequence tagging. In *Proc. of EMNLP*.

Yan Shao. 2017. Cross-lingual word segmentation and morpheme segmentation as sequence labelling. *CoRR* abs/1709.03756. http://arxiv.org/abs/1709.03756.

Tianze Shi, Liang Huang, and Lillian Lee. 2017a. Fast(er) exact decoding and global training for transition-based dependency parsing via a minimal feature set. In *Proc. of EMNLP*. https://www.aclweb.org/anthology/D17-1002.

Tianze Shi, Felix G. Wu, Xilun Chen, and Yao Cheng. 2017b. Combining global models for parsing universal dependencies. In *Proc. of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*. http://www.aclweb.org/anthology/K17-3003.

Samuel L. Smith, David H. P. Turban, Steven Hamblin, and Nils Y. Hammerla. 2017. Offline bilingual word vectors, orthogonal transformations and the inverted softmax. *CoRR* abs/1702.03859. http://arxiv.org/abs/1702.03859.

Milan Straka, Jan Hajič, and Jana Straková. 2016. UDPipe: trainable pipeline for processing CoNLL-U files performing tokenization, morphological analysis, POS tagging and parsing. In *Proc. of LREC-2016*.

Daniel Zeman, Filip Ginter, Jan Hajič, Joakim Nivre, Martin Popel, and Milan Straka. 2018. CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies. In *Proc. of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*.

Daniel Zeman, Martin Popel, Milan Straka, Jan Hajič, Joakim Nivre, Filip Ginter, Juhani Luotolahti, Sampo Pyysalo, Slav Petrov, Martin Potthast, Francis Tyers, Elena Badmaeva, Memduh Gökırmak, Anna Nedoluzhko, Silvie Cinková, Jan Hajič jr., Jaroslava Hlaváčová, Václava Kettnerová, Zdeňka Urešová, Jenna Kanerva, Stina Ojala, Anna Missilä, Christopher Manning, Sebastian Schuster, Siva Reddy, Dima Taji, Nizar Habash, Herman Leung, Marie-Catherine de Marneffe, Manuela Sanguinetti, Maria Simi, Hiroshi Kanayama, Valeria de Paiva, Kira Droganova, Héctor Martínez Alonso, Çağrı Çöltekin, Umut Sulubacak, Hans Uszkoreit, Vivien Macketanz, Aljoscha Burchardt, Kim Harris, Katrin Marheinecke, Georg Rehm, Tolga Kayadelen, Mohammed Attia, Ali Elkahky, Zhuoran Yu, Emily Pitler, Saran Lertpradit, Michael Mandl, Jesse Kirchner, Hector Fernandez Alcalde, Jana Strnadova, Esha Banerjee, Ruli Manurung, Antonio Stella, Atsuko Shimada, Sookyoung Kwak, Gustavo Mendonça, Tatiana Lando, Rattima Nitisaroj, and Josie Li. 2017. CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies.

Bo Zheng, Wanxiang Che, Jiang Guo, and Ting Liu. 2017. Enhancing lstm-based word segmentation using unlabeled data. In *Chinese Computational Linguistics and Natural Language Processing Based on Naturally Annotated Big Data*.

ltcode	sent+tokenize	tagger	parser	LAS	Other LAS	rank	diff.
af_afribooms	udpipe: self	biaffine (none): self	biaffine (none)*3: self	85.47	85.45	1	0.02
ar_padt	udpipe: self	biaffine ($h_{0,1,2}$): self	biaffine ($h_{0,1,2}$)*3: self	73.63	77.06	2	-3.43
bg_btb	udpipe: self	biaffine (h_0): self	biaffine (h_0)*3: self	91.22	90.41	1	0.81
br_keb	udpipe: self	biaffine_trans: self+ga_idt	biaffine_trans*3: self+ga_idt	8.54	38.64	21	-30.1
bxr_bdt	udpipe: self	biaffine_trans: self+hi_hdtb	biaffine_trans*3: self+hi_hdtb	15.44	19.53	6	-4.09
ca_ancora	udpipe: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self	91.61	90.82	1	0.79
cs_cac	udpipe: self	biaffine (h_0): self	biaffine (h_0)*3: self	91.61	91.00	1	0.61
cs_fictree	udpipe: self	biaffine ($h_{0,1,2}$): self	biaffine ($h_{0,1,2}$)*3: self	92.02	91.83	1	0.19
cs_pdt	udpipe: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self	91.68	90.57	1	1.11
cs_pud	udpipe: cs_pdt	biaffine (h_0): cs_pdt	biaffine ($h_{0,1,2}$)*3: cs_pdt	86.13	85.35	1	0.78
cu_proiel	uppsala: self	biaffine ($h_{0,1,2}$): self	biaffine ($h_{0,1,2}$)*3: self	74.29	75.73	3	-1.44
da_ddt	uppsala: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self	86.28	84.88	1	1.40
de_gsd	udpipe: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self	80.36	79.03	1	1.33
el_gdt	uppsala: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self	89.65	89.59	1	0.06
en_ewt	udpipe: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self	84.57	84.02	1	0.55
en_gum	udpipe: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self	84.42	85.05	2	-0.63
en_lines	udpipe: self	biaffine ($h_{0,1,2}$): self	biaffine (h_0)*3: self+en_ewt+en_gum	81.97	81.44	1	0.53
en_pud	udpipe: en_ewt	biaffine (h_0): en_ewt	biaffine ($h_{0,1,2}$)*3: en_ewt	87.73	87.89	2	-0.16
es_ancora	udpipe: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self	90.93	90.47	1	0.46
et_edt	uppsala: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self	85.35	84.15	1	1.20
eu_bdt	udpipe: self	biaffine ($h_{0,1,2}$): self	biaffine (h_0)*3: self	84.22	83.13	1	1.09
fa_seraji	udpipe: self	biaffine ($h_{0,1,2}$): self	biaffine ($h_{0,1,2}$)*3: self	88.11	86.18	1	1.93
fi_ftb	udpipe: self	biaffine ($h_{0,1,2}$): self	biaffine (h_0)*3: self	88.53	87.86	1	0.67
fi_pud	udpipe: fi_tdt	biaffine (h_0): fi_tdt	biaffine (h_0)*3: fi_tdt	90.23	89.37	1	0.86
fi_tdt	uppsala: self	biaffine (h_0): self	biaffine (h_0)*3: self	88.73	87.64	1	1.09
fo_oft	udpipe: no_bokmaal	biaffine_trans: no_bokmaal	biaffine_trans*3: no_bokmaal	44.05	49.43	1	-5.38
fr_gsd	udpipe: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self	86.89	86.46	1	0.43
fr_sequoia	udpipe: self	biaffine ($h_{0,1,2}$): self	biaffine ($h_{0,1,2}$)*3: self	89.65	89.89	1	-0.24
fr_spoken	udpipe: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self+fr_gsd+fr_sequoia	75.78	74.31	1	1.47
fro_srcmf	udpipe: self	biaffine (none): self	biaffine (none)*3: self	87.07	87.12	2	-0.05
ga_idt	udpipe: self	biaffine (none): self	biaffine (none)*3: self	68.57	70.88	5	-2.31
gl_ctg	udpipe: self	biaffine (none): self	biaffine (none)*3: self	82.35	82.76	2	-0.41
gl_treegal	udpipe: self	biaffine (none): self	biaffine (none)*3: self	72.88	74.25	4	-1.37
got_proiel	uppsala: self	biaffine (none): self	biaffine (none)*3: self	69.26	69.55	3	-0.29
grc_perseus	udpipe: self	biaffine ($h_{0,1,2}$): self	biaffine ($h_{0,1,2}$)*3: self	79.39	74.29	1	5.10
grc_proiel	uppsala: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self	79.25	76.76	1	2.49
he_htb	udpipe: self	biaffine (h_0): self	biaffine (h_0)*3: self	67.05	76.09	3	-9.04
hi_hdtb	udpipe: self	biaffine ($h_{0,1,2}$): self	biaffine (h_0)*3: self	92.41	91.75	1	0.66
hr_set	udpipe: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self	87.36	86.76	1	0.60
hsb_ufal	udpipe: self	biaffine_trans: self+pl_lfg	biaffine_trans*3: self+pl_lfg	37.68	46.42	4	-8.74
hu_szeged	udpipe: self	biaffine (h_0): self	biaffine (h_0)*3: self	82.66	79.47	1	3.19
hy_armtdp	udpipe: self	biaffine_trans: self+et_edt	biaffine_trans*3: self+et_edt	33.90	37.01	3	-3.11
id_gsd	uppsala: self	biaffine ($h_{0,1,2}$): self	biaffine ($h_{0,1,2}$)*3: self	80.05	79.13	1	0.92
it_isdt	udpipe: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self	92.00	91.47	1	0.53
it_postwita	uppsala: self	biaffine ($h_{0,1,2}$): self	biaffine ($h_{0,1,2}$)*3: self+it_isdt	79.39	78.62	1	0.77
ja_gsd	udpipe+scir: self	biaffine (h_0): self	biaffine (h_0)*3: self	83.11	79.97	1	3.14
ja_modern	udpipe+scir: ja_gsd	biaffine (h_0): ja_gsd	biaffine (h_0)*3: ja_gsd	26.58	28.33	4	-1.75
kk_ktb	udpipe: self	biaffine_trans: self+tr_imst	biaffine_trans*3: self+tr_imst	23.92	31.93	10	-8.01
kmr_mg	udpipe: self	biaffine_trans: self+fa_seraji	biaffine_trans*3: self+fa_seraji	26.26	30.41	5	-4.15
ko_gsd	udpipe: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self	85.14	84.31	1	0.83
ko_kaist	udpipe: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self	86.91	86.84	1	0.07
la_ittb	udpipe: self	biaffine ($h_{0,1,2}$): self	biaffine ($h_{0,1,2}$)*3: self	87.08	86.54	1	0.54
la_perseus	udpipe: self	biaffine (h_0): self+la_proiel	biaffine ($h_{0,1,2}$)*3: self+la_proiel	72.63	68.07	1	4.56
la_proiel	uppsala: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self	73.61	71.76	1	1.85
lv_lvtb	udpipe: self	biaffine (h_0): self	biaffine (h_0)*3: self	83.97	81.85	1	2.12
nl_alpino	udpipe: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self+nl_lassysmall	89.56	87.49	1	2.07
nl_lassysmall	uppsala: self	biaffine ($h_{0,1,2}$): self	biaffine ($h_{0,1,2}$)*3: self+nl_alpino	86.84	84.27	1	2.57
no_bokmaal	udpipe: self	biaffine ($h_{0,1,2}$): self	biaffine ($h_{0,1,2}$)*3: self	91.23	90.37	1	0.86
no_nynorsk	udpipe: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self	90.99	89.46	1	1.53
no_nynorsklia	udpipe: self	biaffine (h_0): self+no_nynorsk	biaffine ($h_{0,1,2}$)*3: self+no_nynorsk	70.34	68.71	1	1.63
pcm_nsc	udpipe: en_ewt	biaffine (h_0): en_ewt	biaffine ($h_{0,1,2}$)*3: en_ewt	24.48	30.07	2	-5.59
pl_lfg	udpipe: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self	94.86	94.62	1	0.24
pl_sz	udpipe: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self	92.23	91.59	1	0.64
pt_bosque	uppsala: self	biaffine ($h_{0,1,2}$): self	biaffine ($h_{0,1,2}$)*3: self	87.61	87.81	3	-0.20
ro_rrt	udpipe: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self	86.87	86.33	1	0.54
ru_syntagrus	udpipe: self	biaffine ($h_{0,1,2}$): self	biaffine (h_0)*3: self	92.48	91.72	1	0.76
ru_taiga	udpipe: self	biaffine ($h_{0,1,2}$): self+ru_syntagrus	biaffine ($h_{0,1,2}$)*3: self+ru_syntagrus	71.81	74.24	3	-2.43
sk_snk	udpipe: self	biaffine ($h_{0,1,2}$): self	biaffine ($h_{0,1,2}$)*3: self	88.85	87.59	1	1.26
sl_ssj	uppsala: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self	91.47	91.26	1	0.21
sl_sst	udpipe: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self+sl_ssj	61.39	58.12	1	3.27
sme_giella	udpipe: self	biaffine (none): self	biaffine ($h_{0,1,2}$)*3: self	69.06	69.87	3	-0.81
sr_set	udpipe: self	biaffine (none): self	biaffine ($h_{0,1,2}$)*3: self	88.33	88.66	3	-0.33
sv_lines	udpipe: self	biaffine (h_0): self	biaffine (h_0)*3: self+sv_talbanken	84.08	81.97	1	2.11
sv_pud	udpipe: sv_lines	biaffine (h_0): sv_lines	biaffine (h_0)*3: sv_lines+sv_talbanken	80.35	79.71	1	0.64
sv_talbanken	udpipe: self	biaffine ($h_{0,1,2}$): self	biaffine (h_0)*3: self+sv_lines	88.63	86.45	1	2.18
th_pud	thai	biaffine_trans: zh_gsd	biaffine_trans*3: zh_gsd	0.64	13.70	14	-13.06
tr_imst	udpipe: self	biaffine (h_0): self	biaffine ($h_{0,1,2}$)*3: self	66.44	64.79	1	1.65
ug_udt	udpipe: self	biaffine (h_0): self	biaffine (h_0)*3: self	67.05	65.23	1	1.82
uk_iu	udpipe: self	biaffine (h_0): self	biaffine (h_0)*3: self+ru_syntagrus	88.43	85.16	1	3.27
ur_udtb	udpipe: self	biaffine (h_0): self	biaffine (h_0)*3: self	83.39	82.15	1	1.24
vi_vtb	udpipe+scir: self	biaffine ($h_{0,1,2}$): self	biaffine (h_0)*3: self	55.22	47.41	1	7.81
zh_gsd	udpipe+scir: self	biaffine (none): self	biaffine ($h_{0,1,2}$)*3: self	76.77	71.04	1	5.73

Table 6: The strategies used in the final submission. The *toolkit* and *model* are separated by colon (:). *uppsala* denotes the Uppsala segmentor; *scir* denotes our segmentor for Japanese, Vietnamese, and Chinese; *biaffine* denotes the biaffine tagger and parser; *biaffine_trans* denotes our transfer parser for low-resource languages. h_0 or $h_{0,1,2}$ in the brackets denotes the ELMo used to train the model where h_0 means using $\mathbf{h}_{i,0}^{(LM)}$ for ELMo and $h_{0,1,2}$ means using $\sum_{j=0}^{2} \mathbf{h}_{i,j}^{(LM)}$. *self* notes that the model is trained with the treebank itself. If the model field is not filled with *self*, the model is trained through treebank concatenation. We also show the test LAS and the difference against the other best system.