MINISTERUL EDUCAȚIEI NAȚIONALE
ROMÂNIA
UNIVERSITATEA DE MEDICINĂ, FARMACIE, ȘTIINȚE ȘI TEHNOLOGIE „GEORGE EMIL PALADE" DIN TÂRGU MUREȘ

sciendo

Acta Marisiensis. SeriaTechnologica
Vol. 18 (XXXV) no. 1, 2021
ISSN 2668-4217, ISSN-L 2668-4217

10.2478/amset-2021-0006

EFFICIENCY ANALYSIS OF DEEPLEARNING4J NEURAL NETWORK CLASSIFIERS IN DEVELOPMENT OF TRANSITION BASED DEPENDENCY PARSERS

László CSÉPÁNYI-FÜRJES[1], **László KOVÁCS**[2]
[1,2]University of Miskolc
Institute of Information Science, 3515 Miskolc-Egyetemváros, Hungary
[1]csf.laszlo@gmail.com
[2]kovacs@iit.uni-miskolc.hu

Abstract

Dependency parsing is a complex process in natural language text processing, text to semantic transformation. The efficiency improvement of dependency parsing is a current and an active research area in the NLP community. The paper presents four transition-based dependency parser models with implementation using DL4J classifiers. The efficiency of the proposed models were tested with Hungarian language corpora. The parsing model uses a data representation form based on lightweight embedding and a novel morphological-description-vector format is proposed for the input layer. Based on the test experiments on parsing Hungarian text documents, the proposed list-based transitions parsers outperform the widespread stack-based variants.

Key words: NLP, dependency parser, word embedding, lightweight word embedding

1. Introduction

Dependency parsing is a complex and important component in various natural language processing (NLP) operations [1]. The dependency graph is a key structure to perform high level text processing operations like machine translation, semantic role labeling [2], relation extraction [3], question answering, sentiment analysis or text classification. Based on the real-life experiments, however, these systems are still far from perfect [4].

There are two main approaches in dependency parsing [5]. One direction is the transition-based approach that performs a sequence of simple basic transformation steps to find the optimal graph [6]. The other approach is a graph-based method where direct graph construction rules are applied on the input text [7].

Regarding dependency parsing, the key research directions relate to the development of new text representation methods and to optimization of the related machine learning methods. Currently there are several all-in-one NLP parser solutions available which are working as a pipeline and performing several text processing steps such as, tokenization, POS tagging, named entity recognition, etc. [8], [9]. In our experiments we were concentrating only on dependency parsing, therefore we left these other steps out of the scope by using the gold standard tokens and POS (Part of Speech) tags. In our investigation, we analyzed the efficiency of four transition-based dependency parser models with implementation using DL4J (Deeplearning4J) classifiers. In the parser models we implemented special word representation format using lightweight embedding and a novel morphological-description-vector format. In the tests, two main classifier neural network variants, namely the multi-layer perceptron model and the recurrent neural-network model were involved in the investigation. We used lightweight representation vectors on the input side of the neural networks and still achieved competitive results in terms of the UAS (unlabeled attachment score) indicator and F1 score. Our embedding library was also generated using DL4J's Word2Vec [10] module. We can mention here that in the last couple of years the contextualized word embedding, like BERT [11] achieved significant improvements in NLP tasks. As we wanted to use experimental embedding vectors such as POS-vectors and morphological-description-vectors together with the well-known word-vectors, we choose Word2Vec to generate these for us instead of using pre-trained ones.Regarding the implementation aspects, we used the DL4J [12] framework, which is an open source, distributed, deep learning library for the JVM developed by the Eclipse Foundation. However, at time of writing TensorFlow and Pytorch are the most popular Neural Network libraries, DL4J is getting more and more interest because it is competitive, production ready and provides commercial support.

The main contribution of the paper can be summarized in the followings:

- Efficiency analysis of four transition-based dependency parser models with implementation using DL4J classifiers.
- Implementation of special word representation format using lightweight embeddings.
- Introduction of a novel morphological-description-vector format.
- Comparative analysis between the widespread stack-based parsers and the list-based variants on Hungarian corpora

Test results showed that the extension of the feature set with morphological description of the words can significantly (4-5%) improve the accuracy in UAS score.

2. Survey on Dependency Parsing

One widely used approach to describe sentence structure in natural language is a graph structure where nodes are the words and labeled links connect the words. This approach is called dependency grammar [1] and the link structure is called dependency tree. There are two main methods for generating dependency trees, the graph method and the transition-based method. The transition-based method predicts an operation sequence starting from an initial configuration and ending at a predefined end configuration, which determines a dependency parse tree. The mentioned operation in other words is called a transition. The usual approach to predict the transition sequence involves ML (Machine Learning) techniques, such as classification [13]. In order to appropriately train the classifier, we need properly annotated training set called text corpus. Several different notation systems have been developed to provide the mentioned annotation, including UD (Universal Dependencies) [14] which aim is to provide a general-purpose language independent solution. Among others there is a Hungarian corpus available within the UD project [15]. Also, there is a much larger Hungarian text corpus that is following the file format that was defined in the UD project, however the annotation label, the morphological description and the POS tag set are different. It is called Szeged Dependency Treebank (SZDT) [16]. We performed test experiments with both corpora.

The transition-based dependency parser family [17] includes the best known and most widely used methods like Arc Standard Stack Based (ASSB) and Arc Eager Stack Based (AESB), as well as the Non-Projective List Based (NPLB) and Projective List Based (PLB) variants. The stack-based configuration can be described by a triple $c(\sigma, \beta, A)$ where σ denotes the stack of the pending tokens, β is the buffer of unprocessed tokens and A is the dependency tree itself.

The simplest implementation is the ASSB variant that uses the following transitions:

Left-Arc$_l$:

$$(\sigma \vee i, j \vee \beta, A) \Rightarrow (\sigma, j \vee \beta, A \cup \{j, l, i\}) \quad (1)$$

Right-Arc$_l$:

$$(\sigma \vee i, j \vee \beta, A) \Rightarrow (\sigma, i \vee \beta, A \cup \{i, l, j\}) \quad (2)$$

Shift:

$$(\sigma, i \vee \beta, A) \Rightarrow (\sigma \vee i, \beta, A) \quad (3)$$

The AESB variant aims to build the right-side dependency earlier (eagerly). It means that the Right-Arc transition does not remove the token from the stack. In order to process the whole sentence, it introduces a new transition, called reduce. The transition set looks like this:

Left-Arc$_l$:

$$(\sigma \vee i, j \vee \beta, A) \Rightarrow (\sigma, j \vee \beta, A \cup \{j, l, i\}) \quad (4)$$

Right-Arc$_l$:

$$(\sigma \vee i, j \vee \beta, A) \Rightarrow (\sigma, i \vee \beta, A \cup \{i, l, j\}) \quad (5)$$

Reduce:

$$(\sigma \vee i, \beta, A) \Rightarrow (\sigma, \beta, A) \quad (6)$$

Shift:

$$(\sigma, i \vee \beta, A) \Rightarrow (\sigma \vee i, \beta, A) \quad (7)$$

In the list-based system, the stack is replaced by two lists where one of the lists is used as a temporary storage. This configuration can be described by a quadruple $c(\lambda_1, \lambda_2, \beta, A)$ where (λ_1, λ_2) are representing the two mentioned lists. Up to now we have been discussing only projective dependency graph constructions where the arcs of the graph are not crossing each other. It is important to note that the Hungarian language contains large portion of non-projective structures as well [18]. In these structures the arcs can cross each other. Therefore, we want to experiment with a non-projective variant as well. The NPLB algorithm is one of the first non-projective systems for dependency parsing, it is using the following transition set:

Left-Arc$_l$:

$$\begin{gathered}(\lambda_1 \vee i, \lambda_2, j \vee \beta, A) \Rightarrow \\ (\lambda_1, i|\lambda_2, j|\beta, A \cup \{j, l, i\})\end{gathered} \quad (8)$$

Right-Arc$_l$:

$$(\lambda_1 \vee i, \lambda_2, j \vee \beta, A) \Rightarrow (\lambda_1, i|\lambda_2, j|\beta, A \cup \{j, l, i\}) \tag{9}$$

No-Arc:

$$(\lambda_1 \vee i, \lambda_2, \beta, A) \Rightarrow (\lambda_1, i \vee \lambda_2, \beta, A) \tag{10}$$

Shift:

$$(\lambda_1, \lambda_2, i \vee \beta, A) \Rightarrow (\lambda_1.\lambda_2 \vee i, [\ \ ], \beta, A) \tag{11}$$

In the Shift transition description, the $(\lambda_1.\lambda_2)$ formula means the concatenation of the two lists.

Finally, there is a list-based algorithm for strictly projective structures. This is the PLB variant which uses the following transitions:

Left-Arc$_l$:

$$(\lambda_1 \vee i, \lambda_2, j \vee \beta, A) \Rightarrow (\lambda_1, i|\lambda_2, j|\beta, A \cup \{j, l, i\}) \tag{12}$$

Right-Arc$_l$:

$$(\lambda_1 \vee i, \lambda_2, j \vee \beta, A) \Rightarrow (\lambda_1, i|\lambda_2, j|\beta, A \cup \{j, l, i\}) \tag{13}$$

No-Arc:

$$(\lambda_1 \vee i, \lambda_2, \beta, A) \Rightarrow (\lambda_1, i \vee \lambda_2, \beta, A) \tag{14}$$

Shift:

$$(\lambda_1, \lambda_2, i \vee \beta, A) \Rightarrow (\lambda_1.\lambda_2 \vee i, [\ \ ], \beta, A) \tag{15}$$

Nowadays the most frequently used ML technique to determine the transition sequence of a parse tree is the Neural Network prediction. Multi-Layer Perceptron (MLP) models produced pretty good outcome, but the best results have been achieved with Recurrent Neural Network (RNN) structures, in which the results of previous steps are also incorporated into the determination of the following output. Within the RNN model family, the Long-Short-Term-Memory (LSTM) can be considered the most efficient solution for analyzes and classification problems to be performed on word sequences. The used DL4J library contains both MLP (*DenseLayer* and *OutputLayer*) and RNN (*LSTM* and *RnnOutputLayer*) NN configurations (*MultiLayerConfiguration*), so we decided to incorporate both into our system.

3. Proposed Parsing Model

In the literature, we can find only a few experiments where some Hungarian corpus was also among the examined languages. Noji (2016) used pseudo-projective dependency parsing to process the non-projective language formulas [19]. For Hungarian they achieved 80,9% UAS with arc-eager and 79,1% with arc-standard algorithms. Gómez (2018) also tested the typical non-projective languages, and his result was 84,6% for Hungarian, which is one of the lowest among the examined languages [20]. Tálas (2019) examined how we can improve the UAS by applying changes to the annotation system of the Hungarian corpus. Their UAS result was 86,6% [21]. The main goal of our investigation was to perform a more general efficiency comparison of the main dependency parsing methods on a specific corpus containing documents in Hungarian language. The research motivation was twofold:

- to determine the efficiency of the general methods which were developed for the mainstream languages on a special agglutinative language,
- to develop novel optimization approach to improve the efficiency of the parsing methods on the agglutinative languages.

In order to achieve the research goals, we implemented two stack and two list-based dependency parser algorithms from the transition-based family.

In order to establish a baseline-experiment we trained both of our NN classifiers for all four algorithms using the UD Hungarian Szeged corpus. In order to compare the result to a non-NN classifier, also Multi-Class-Classifier (MCC) engine [22] was included into the tests where MCC was implemented with a simple decision-matrix based classifier. To be able to train the NN module, a training transition sequence set was built that describes the processing steps of the individual sentences in the actual corpus. One row of this sequence contains a gold transition along with a feature vector that determines the actual state of the processing. The feature vector is basically a concatenation of individual word/POS representations. We used one of the most common methods to build the mentioned representations, the Word2Vec algorithm. We denote the word and POS element sets as S^w, S^t respectively. The embedded representation set of these elements are E^w, E^t. The elements of these sets (e_i^w) will be concatenated to produce the feature vector. If $(n| \ \ |w, n_t)$ is the number of the selected elements then we add the $x^w = \left[e_{w_1}^w; e_{w_2}^w; \dots e_{w_{n_w}}^w\right]$ word vector to the feature vector, where the set of words is $S^w = \{w_1, w_2, \dots w_{n_w}\}$. Also, we add the $x^t = \left[e_{t_1}^t; e_{t_2}^t; \dots e_{t_{n_t}}^t\right]$ POS vector to the feature vector where the set of POS tags is $S^t = \{t_1, t_2, \dots t_{n_t}\}$.

Concretely in the stack-based algorithms the S^w set holds $n_w = 18$ elements. The top three words of the stack s_1, s_2, s_3 and the buffer b_1, b_2, b_3. The first and the second leftmost and rightmost dependents of the top two words from the stack and the buffer $lc_1(s_i), rc_1(s_i), lc_2(s_i), rc_2(s_i); (i = 1,2)$. The first and the second leftmost and rightmost dependents of these dependents $lc_1(lc_1(s_i)), rc_1(rc_1(s_i)) (i = 1,2)$. Including all the POS tags of these words.

In the list-based algorithms, the S^w set holds $n_w = 23$ elements. The first two words of λ^1 the first and the last words of λ_2 and the top three words of the buffer $\lambda_1^1, \lambda_2^1, \lambda_1^2, \lambda_n^2, b_1, b_2, b_3$. The leftmost and the rightmost dependents of the first two words from λ^1 and the leftmost and the rightmost dependents of the first word from λ^2, these are $lc_1(s_i), rc_1(s_i), lc_2(s_i), rc_2(s_i); (i = 1,2)$. The leftmost and the rightmost dependent of these first dependent words

$$lc_1(lc_1(\lambda_1^1)), rc_1(rc_1(\lambda_1^1)),$$
$$lc_1(lc_1(\lambda_1^2)), rc_1(rc_1(\lambda_1^2)).$$

We included the POS tag of the described words as well.

4. Environment and Results

Since the UD Hungarian Szeged corpus is relatively small, we did not expect state-of-the-art results in this case. The aim of this test was to implement the basic proof of our concept. Another aspect was the fact that UD treebank is the most widely available Hungarian corpus, so it was important to establish a baseline analysis with that dataset. The comparison result of the experiments can be seen in fig.1.

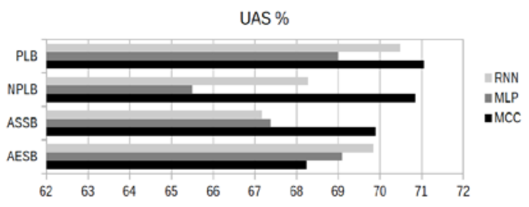

Fig. 1: Baseline test on UD Hungarian Szeged corpus using NN and non-NN classifiers

In the tests, we wanted to verify that our models with DL4J classifiers are appropriate tools, so we selected also a much larger corpus from the SZDT. We used the very same setup to do the training. The UD corpus was previously separated into test and train sub corpora. To compare the results we also separated the SZDT corpus excerpt into two disjunct train and test sub corpora in 9 to 1 portions. The result for the stack-based and the list-based algorithms are shown in table 1 and table 2 respectively. The evaluation was always done on the test portion that belongs to the actual train portion. Cross evaluation between the two corpora is not possible since the used POS tag set is not the same.

Table 1: Stack-based algorithm evaluations with MLP

	AESB		ASSB	
	UD	SZDT	UD	SZDT
Accuracy	0,867	0,945	0,850	0,933
Precision	0,866	0,945	0,846	0,931
Recall	0,860	0,945	0,833	0,929
F1 Score	0,863	0,945	0,839	0,930
UAS	0,691	0,847	0,674	0,831

Table 2: List-based algorithm evaluations with MLP

	NPLB		PLB	
	UD	SZDT	UD	SZDT
Accuracy	0,875	0,950	0,868	0,946
Precision	0,876	0,935	0,867	0,946
Recall	0,852	0,945	0,862	0,946
F1 Score	0,855	0,940	0,865	0,946
UAS	0,655	0,845	0,690	0,847

We used the usual NN metrics and the UAS score to compare the results. While the NN metrics are giving us information about the performance of the classifiers and the learning process itself, the UAS score provides us details about the accuracy of the produced dependency tree.

Beside the MLP tests we executed the evaluation with the RNN classifier as well, the result is shown in table 3 and table 4.

Table 3: Stack-based algorithm evaluations with RNN

	AESB		ASSB	
	UD	SZDT	UD	SZDT
Accuracy	0,878	0,954	0,876	0,954
Precision	0,880	0,954	0,869	0,953
Recall	0,871	0,953	0,858	0,951
F1 Score	0,875	0,953	0,864	0,952
UAS	0,698	0,869	0,672	0,840

Table 4: List-based algorithm evaluations with RNN

	NPLB		PLB	
	UD	SZDT	UD	SZDT
Accuracy	0,907	0,968	0,881	0,955
Precision	0,891	0,960	0,880	0,955
Recall	0,885	0,963	0,878	0,954
F1 Score	0,887	0,962	0,878	0,955
UAS	0,683	0,865	0,705	0,869

Based on the latest trends, the usual size of the embedded vector is over 100. In our case, we implemented a lightweight solution to test the effects of smaller size on the accuracy results. The aim was to achieve a good result without significantly increasing the required computational power. In the subsequent tests we applied two sizes: 30 and 50 and tested the NN classifiers on the SZDT corpus. The full sizes of the input vector of the stack-based networks were 1080 and 1800, the list-based networks' full sizes were 1380 and 2300 recpectively. Results can be seen in fig. 2 and fig.3.

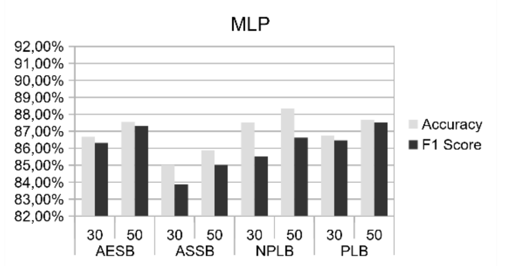

Fig. 2: Comparing embedding dimensions using Accuracy and F1 Score with MLP classifiers

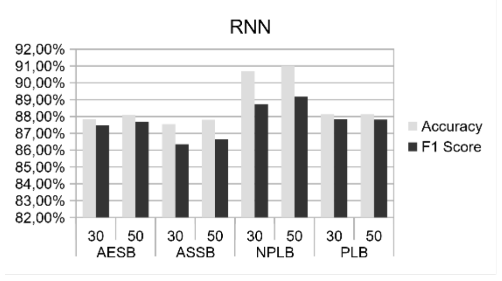

Fig. 3: Comparing embedding dimensions using Accuracy and F1 Score with RNN classifiers

In our final experiment we added the whole morphological description of the words in embedded format to the input vector. The change of the UAS indicator can be observed in fig. 4.

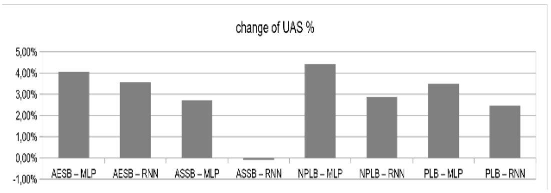

Fig. 4: Change of UAS after adding the whole morphological description to the input vector

5. Discussion

There are several available dependency parsers on the market that use their own strongly coupled classifier implementations [23], [24]. In our opinion this architecture does not support a flexible classifier usage, replacement in case a better one appears. Our aim was to develop a more modular system where the parser implementations as well as the classifiers are modular, replaceable.Considering the most important hyperparameters of model training, one key element is the size of our input vector. In the case of stack-based algorithms it is 1080, in case of list-based algorithms it is 1380. The hidden layer size is 200. We used a cube activation function, and we applied the XAVIER initialization method. The mini-batch size of the processing is 50. We also use the AdaGrad method with rate = 0,01 and dropout = 0,5. On the output layer we use the softmax activation. The output size always depends on the trained algorithm and it is the same as the number of the possible transitions. We applied standard back propagation in our MLP network.

The drawback of the MLP network is that it is not using information from the previous steps which means that it makes the actual decision in an isolation. Since the dependency parsing process is basically a transition sequence determination where the single transitions are not independent from each other, this network is not as optimal as the RNN network. When an RNN network makes a classification decision it is processing not only the input vector but also the result of the previous step. Our RNN network also uses a 3-layer setup and applies basically the same hyperparameters as the MLP network. The hidden layer is an LSTM with a tanh activation function. We used a size = 30 truncated backpropagation method which helps to limit the size of the back propagation operation. It resulted in better performance without significantly reducing the accuracy.

The UD Hungarian Szeged corpus is the one that is widely available for researchers since it is part of the Universal Dependencies project. In one of our previous research, we implemented a simple decision-matrix based classifier and we trained it with the very same treebank. Figure 1 shows the result of our NN models where the decision-matrix based classifier is also included. The simple decision-matrix based classifier was outperforming the NN models if the size of the train corpus was small, like in this case where the size of the train corpus is only 910. The decision-matrix based classifier uses Bag of Words (BOW) representation which means that if we enlarge the size of the text corpus then the size of the decision matrix becomes unmanageably big. From the stack-based NN models the AESB performs better, competing with the PLB algorithm, which proved to be the best overall for all three NN classifiers. This result is surprising, as the Hungarian language uses a significant amount of non-projective structure, based on which we would expect the success of the NPLB algorithm. But all together even the best result was significantly lower than the state of the art.In order to find out what our networks are capable of in the following experiment we also included a much larger Hungarian corpus, the SZDT. We selected the *8oelb* subcorpus from SZDT that contains 6817 train sentences. It is important to note that a perfect comparison would be a case where not only the language is matching, but also the train and test corpora are the same. Nevertheless, it is worth comparing the results even with different corpora because we can draw important conclusions from these numbers also.

Tables 1-4 are showing the comparison of the two Hungarian corpora. It can be clearly seen that the training on SZDT significantly increases the efficiency of the algorithms. The list-based ones prove to be better, although the AESB algorithm is also competitive, as we approach 85% UAS with this and the PLB algorithm. We managed to achieve even higher results with the RNN classifier. With that also the PLB and AESB algorithms proved to be the best in terms of the UAS metrics, we approached 87% with both. In terms of other NN metrics, the NPLB classifier was showing the best result. Of course, in all cases the evaluation was being done using the matching test corpus.

As we already mentioned our aim was to use lightweight word embeddings. We used only 30 dimensional embeddings in our previous evaluations and we still achieved a good result in terms of UAS indicator. In a repeated evaluation, we run all our tests on the UD corpus with an increased embeddings dimension (50). In Fig. 2-3 we can see that both the NN Accuracy and the F1 score is increasing slightly when we increase the dimensionality of the word embeddings. The value of the increase is greater for MLP classifiers than the RNN, but either case the improvement is not significant. We believe that there are industrial situations where these lightweight embeddings can be still efficiently used as our results are showing.

Our models used the basic features to construct the input vector. We suggested an improvement by adding the morphological description of the word to the feature set. In the annotated corpora this information is also available in a textual format. We built up the sentences using these morphological descriptions instead of words and executed the same Word2Vec algorithm on these altered sentences. Thanks to the improved algorithm we could map into the vector space not only words, but also other textual entities. The result was a morphological description library that contains 230 different entries for the UD corpus. We extended the input vector with this information in our next evaluation. The basic rule was simple, where we added the POS tag of a word, we also added the morphological description. With this improvement the input vector size of the stack-based algorithms became 1260, the size of the list-based ones became 1590. We could observe a 3-4% increase in UAS in fig. 4 in most of the algorithms with this improvement. This result suggested that extending the input vector features with different observations we can gain improved learning ability. Considering the implementation framework, the DL4J library appears to be easy to use and flexible. We used it in the preliminary phase as well to generate word embeddings and to generate train and test data sets. Our lightweight parser evaluated on CPU, but it is also possible to incorporate the GPU with just replacing a dependency library. We made a test with GPU, but it turned out that there is no benefit using it in case of a light input vector. The results show that it takes more time to deliver the data to the GPU and do the calculation in parallel than to do the calculation directly using the CPU. Nevertheless, DL4J has this feature which makes it more attractive. For complex n-dimensional array operations we used the ND4J library. The main advantage of this library is the optimized memory consumption. We observed the garbage collector performance during the training process, and it worked seamlessly. ND4J uses off-heap memory for data storage which is also a great advantage because of its performance and its interoperability with Basic Linear Algebra Subprograms (BLAS) libraries. We did not have to face the JVM array limits either. All our tests can run in 4GB heap plus 4GB off-heap memory setup.

Since DL4J library contains both MLP (*DenseLayer* and *OutputLayer*) and RNN (*LSTM* and *RnnOutputLayer*) NN configurations (*MultiLayerConfiguration*) we implemented the parser with both classifiers. Our MLP network contained 3 layers. The input layer processed the input vector which is basically a concatenation of all the useful feature vectors of the actual transition state.

6. Conclusion

The paper presented the efficiency analysis of four transition-based dependency parser models with implementation using DL4J classifiers. In the parser models we have implemented special word representation format using lightweight embedding and a novel morphological-description-vector format. In the tests, two main classifier neural network variants, namely the multi-layer perceptron model and the recurrent neural-network model were involved in the investigation.

Based on the test experiments we can say that the lightweight representation vectors on the input side of the neural networks can achieve competitive results considering both UAS and F1 score. Thus, in cases where the high computational power is not available using lightweight word embedding still can be an appropriate option. An important result of our analysis is that we could show that using a set of different small embedding (word, pos, morphological description, etc.) is more beneficial than using a single embedding with large vector size. Test results showed that the extension of the feature set with morphological description of the words can significantly (4-5%) improve the accuracy in UAS score. In the proposed model, we used the Word2Vec algorithm to generate embedded representation of the morphological descriptions. In the analysis we performed also comparative analysis between the widespread stack-based parsers and the list-based variants. Based on the test results, we can say that the list-based parsers are outperforming the stack-based ones for our Hungarian corpora. Based on our findings, we suggest putting more attention on the list-based dependency parsers in case of non-projective languages. From the viewpoint of the implementation, our goal was to prove that DL4J is a good candidate for a modular NLP framework. We used DL4J also to generate word embedding and to generate train and test data sets as well. From DL4J library, we tested MLP and RNN classifiers and performed several evaluations with Hungarian corpora to prove that the modular setup is competitive. Our experience is that DL4J is an appropriate tool in NLP tasks. The DL4J classifier can be easily configured and with the help of the built-in file iterators we can process the input text seamlessly. The memory management with utilization of the off-heap memory works more efficiently. This is a very important aspect in the world of NLP as the size of the corpus to process is usually very large or huge.

References

[1] Hays, D. G. (1964), Dependency Theory: A Formalism and Some Observations, *Language*, vol. 40, p. 511, 10.

[2] Marcheggiani, D. and Titov, I. (2017), Encoding sentences with graph convolutional networks for semantic role labeling, *Proceedings of the Conference on Empirical Methods for Natural Language Processing.*

[3] Zhang, Y., Qi, P. and Manning, C. D. (2018), Graph convolution over pruned dependency trees improves relation extraction, *Proceedings of the Conference on Empirical Methods for Natural Language Processing*.

[4] Qi, P., Dozat, T., Zhang, Y. and Manning, C. D. (2019), Universal Dependency Parsing from Scratch, arXiv:1901.10457.

[5] Choi, J. D. and Palmer, M. (2011), Getting the most out of transition-based dependency parsing.

[6] Nivre, J. (2009), Non-projective dependency parsing in expected linear time, *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL*, p. pp. 351–359.

[7] McDonald, R., Pereira, F., Ribarov, K. and Hajič, J. (2005), Non-projective dependency parsing using spanning tree algorithms, *Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing*, p. 523–530.

[8] Kanerva, J., Ginter, F., and Salakoski, T. (2020), Universal Lemmatizer: A sequence-to-sequence model for lemmatizing Universal Dependencies treebanks, *Natural Language Engineering*, pp. 1-30.

[9] Straka, M., and Straková, J. (2017), Tokenizing, POS Tagging, Lemmatizing and Parsing UD 2.0 with UDPipe, *in Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies.*

[10] Mikolov, T., Chen, K., Corrado, G. and Dean, J. (2014), Word2Vec, [Online]. Available: https://code.google.com/p/word2vec/

[11] Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K. (2019), BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, *in Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Volume 1 (Long and Short Papers), Minneapolis.

[12] Deeplearning4j: Open source distributed deep learning for the JVM (2017), Apache Software Foundation License 2.0., [Online]. Available: http://deeplearning4j.org/

[13] Nivre, J., Hall, J., Kübler, S., McDonald, R., Nilsson, J., Riedel, S. and Yuret, D. (2007), The CoNLL 2007 Shared Task on Dependency Parsing, *in Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, Prague.

[14] Nivre, J., de Marneffe, M.-C., Ginter, F., Goldberg, Y., Hajič, J., Manning, C. D. McDonald, R., Petrov, S., Pyysalo, S., Silveira, N., Tsarfaty, R. and Zeman, D. (2016), Universal Dependencies v1: A Multilingual Treebank Collection, *in Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, Portorož, Slovenia.

[15] Vincze, V., Simkó, K., Szántó, Z. and Farkas, R. (2017), Universal Dependencies and Morphology for Hungarian - and on the Price of Universality, *in Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics*: Volume 1, Long Papers.

[16] Vincze, V., Szauter, D., Almási, A., Móra, G. Alexin, Z. and Csirik, J. (2010), Hungarian Dependency Treebank, *in Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, Valletta.

[17] Nivre, J. (2008), Algorithms for Deterministic Incremental Dependency Parsing, *Computational Linguistics*, vol. 34, p. 513–553, 12.

[18] Sartorio, F. (2015), Improvements in Transition Based Systems for Dependency Parsing, PhD Thesis work.

[19] Noji, H. and Miyao, Y. (2015), Left-corner Parsing for Dependency Grammar, *Journal of Natural Language Processing*, p. 251–288.

[20] Gómez-Rodríguez, C., Shi, T. and Lee, L. (2018), Global Transition-based Non-projective Dependency Parsing, *in Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics* (Volume 1: Long Papers).

[21] Tálas, D. and Novák, A. (2019), Különböző függőségi elemzők teljesítményének vizsgálata magyar nyelven (Examining the performance of different dependency parsers for Hungarian language), *Conference of Hungarian Computational Linguistics*, p. 345–354.

[22] Kovács, L. and Csépányi-Fürjes, L. (2020), Feature reduction for dependency graph construction in computational linguistics, *in CEUR Workshop Proceedings*.

[23] Nivre, J. and Hall, J. (2018), MaltParser, [Online], Available: http://www.maltparser.org/

[24] Chen, D. and Manning, C. D. (2014), A Fast and Accurate Dependency Parser using Neural Networks, *in Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP).*