Dependency parser for Bulgarian

Atanas Atanasov
Faculty of Slavic Studies,
Sofia University "St. Kliment Ohridski"
atanasov@slav.uni-sofia.bg

Abstract

This paper delves into the implementation of a Biaffine Attention Model, a sophisticated neural network architecture employed for dependency parsing tasks. Proposed by Dozat and Manning, this model is applied to Bulgarian language processing. The model's training and evaluation are conducted using the Bulgarian Universal Dependencies dataset. The paper offers a comprehensive explanation of the model's architecture and the data preparation process, aiming to demonstrate that for highly inflected languages, the inclusion of two additional input layers - lemmas and language-specific morphological information - is beneficial. The results of the experiments are subsequently presented and discussed. The paper concludes with a reflection on the model's performance and suggestions for potential future work.

Keywords: syntactic parsing, Universal Dependencies, Biaffine Attention, Bulgarian.

1 Introduction

The paper presents an implementation of a neural network-based dependency parser using TensorFlow and Keras[1]. The parser is trained and evaluated on the Bulgarian Universal Dependencies dataset.

The article introduces a model in its intermediate stage of development, with the current focus entirely on dependency analysis. At this stage, it does not aim to predict parts of speech, lemmas, etc. (these are expected to be added to the model later). For this reason, only the results for Labeled and Unlabeled Attachment Scores will be presented.

1.1 Linguistic background

From a linguistic perspective, Generative Grammar (GG) and Dependency Grammar (DG) have emerged as the two primary approaches to syntax study over the past few decades[2]. GG is a grammar model that operates on the premise that a sentence's syntactic structure is generated by a set of rules. These rules are applied to a set of terminal nodes, which are the words (or potentially empty functional categories) of the sentence. The rules are recursively applied until the sentence is parsed into its smallest constituents. The result of constituency parsing is a tree, known as a constituency parse tree, which represents the sentence's syntactic structure.

On the other hand, DG is a grammar model that posits that a sentence's syntactic structure is represented by a set of dependencies between the sentence's words. These dependencies are directed links between the words of the sentence and are represented by a tree, known as a dependency parse tree. The nodes of the dependency parse tree are the words of the sentence, and the edges (arcs) represent the dependencies between the words. The dependency parse tree has a root node, typically the main verb (predicate) of the sentence. This root node has no incoming edges but can have multiple outgoing edges. The dependency parse tree is a directed acyclic graph (DAG), meaning there are no cycles in the graph and only one path exists between any two nodes in the tree.

[1] The code, written in Python 3.11, has been tested on Ubuntu 22.04.3 LTS running on WSL2. It is accessible at https://github.com/nassoo/dependecy_parser. The necessary packages are enumerated in the requirements.txt file. Additionally, an environment.yml file is provided for convenience. Note that the virtual environment includes packages not directly used in the notebook, such as PyTorch and Transformers, which are for side experiments. If disk space is limited, consider manually installing only the necessary packages.

[2] In fact, the concept behind the dependency approach dates back several thousand years (Kruijff, 2002: 7-17). However, it wasn't until the 20th century that it was formalized and evolved into a comprehensive theory.

In recent decades, Generative Grammar (GG) has been the preferred approach for representing syntactic structures in most linguistic studies. However, due to advancements in Natural Language Processing (NLP), particularly the Universal Dependencies (UD) project and deep learning models based on it, DG is gaining popularity. The primary advantage of DG lies in its simplicity and intuitiveness, despite GG's greater expressiveness.

1.2 Universal Dependencies

Marneffe et al. (2021) point out that "Universal dependencies (UD) is a framework for morphosyntactic annotation of human language, which to date has been used to create treebanks for more than 100 languages" and "the linguistic theory of the UD framework ... draws on a long tradition of typologically oriented grammatical theories. Grammatical relations between words are centrally used to explain how predicate–argument structures are encoded morphosyntactically in different languages while morphological features and part-of-speech classes give the properties of words". The UD project aims to facilitate multilingual parser development, cross-lingual learning, and parsing research from a language typology perspective. It has significantly influenced the field of Natural Language Processing (NLP). The project provides a set of universal guidelines, applicable to all languages, with language-specific extensions. These guidelines cover annotation at both the word level (morphology) and the sentence level (syntax). The UD treebanks, collections of annotated sentences, serve as a valuable resource for training and evaluating models such as part-of-speech taggers and dependency parsers.

1.3 Dependency parsing background

Two primary data-driven approaches exist for dependency parsing: transition-based and graph-based methods (Kübler et al., 2009). The transition-based (shift-reduce) approach is a greedy algorithm that builds a dependency tree by applying a sequence of actions to a partially built tree. They maintain a stack and a buffer of words to be processed. The parsing process starts with all words in the buffer and an empty stack. The parser can perform three types of actions: 'SHIFT' (moves a word from the buffer to the stack); 'REDUCE' (removes a word from the stack); 'ARC' (creates a dependency relation between two words, one from the stack and one from the buffer). The parser makes these actions based on a set of features extracted from the current state of the stack and buffer, and the previously built dependency relations. The process continues until the buffer is empty and all words have been incorporated into the dependency tree. These parsers do not use deep learning algorithms for direct prediction of dependencies, instead they use it to predict the next action to be taken.

Graph-based parsers, on the other hand, work by considering all possible dependency trees for a sentence and choosing the one with the highest score. The score of a tree is typically computed as the sum of the scores of its individual dependencies. The scoring function is learned from a treebank during training. The learning process involves finding weights for the features of the dependencies such that the correct trees in the training data get higher scores than incorrect trees.

Transition-based parsers are generally more efficient, as they parse a sentence in linear time. This makes them suitable for real-time applications or large-scale data processing. However, these parsers suffer from error propagation. The decisions are made greedily, and once a parsing action is taken, it cannot be undone. This means that an error early in the parsing process can affect the rest of the parse. Graph-based parsers often achieve higher accuracy than transition-based parsers. They globally optimize the parse tree of a sentence, considering all possible trees before making a decision. However, this global optimization comes at a cost. Graph-based parsers have a higher time complexity (usually cubic in the length of the sentence), making them slower than transition-based parsers.

The development of dependency parsing has been significantly influenced by the CoNLL Shared Tasks, which have provided standardized datasets and evaluation benchmarks. The CoNLL 2017 (Zeman et al., 2017) and 2018 (Zeman et al., 2018) Shared Tasks, in particular, focused on multilingual dependency parsing, advancing the field through cross-lingual comparisons. While this work concentrates solely on the dependency analysis of Bulgarian, insights gained from these shared tasks have informed the approach to model development and evaluation.

1.4 Deep Biaffine Attention

This paper presents an implementation of a graph-based dependency parser, following the algorithm proposed by Dozat and Manning (2017). Their architecture utilizes a bi-LSTM (Bidirectional Long Short-Term Memory) to read the input sentence from both directions, thereby capturing a rich set of syntactic and semantic features. Furthermore, the authors introduce a biaffine attention mechanism, which is a bilinear function supplemented with an affine transformation. Instead of employing shallow bilinear attention that operates directly on recurrent state representations, deep biaffine attention uses a multi-layer perceptron (MLP) to project these representations into a higher-dimensional space prior to applying the bilinear attention function. This approach enables the model to capture more complex relationships between words and their potential dependencies.

The parser proposed in this paper is trained and evaluated using the Bulgarian UD treebank (Osenova and Simov, 2015).

2 Data preparation

The UD treebanks come in various formats, including CoNLL-U (a format used for linguistic treebanks in the Conference on Natural Language Learning), TensorFlow Datasets (TFDS), and HuggingFace Datasets. Given that this project uses TensorFlow, TFDS is the most convenient format. The treebanks are always split into training, development, and test sets.

The Bulgarian UD dataset includes 8,907 training sentences, 1,115 development sentences, and 1,116 test sentences. This distribution, which approximates an 80-10-10 split for training, validation, and testing respectively, is typical for UD treebanks. Given its suitability for developing a neural network parser, no additional splitting of the data is required.

In Dozat and Manning's architecture, only specific parts of the data are utilized. The *tokens* column, which contains the tokenized sentence, and the *upos* column, which includes Universal Part-of-Speech tags, are used to train the parser. The *deprel* column, with dependency relations, and the *head* column, indicating word head indices, serve as the parser's targets. Given the focus on using the parser for Bulgarian, as highlighted in the Experiments and Results section, two additional columns are incorporated into the input layers: *lemmas* (containing the lemmatized forms of the words) and *xpos* (providing more detailed morphological language-specific information, although these tags are not consistent across languages).

After loading the data, the next step is to construct vocabularies. These vocabularies consist of unique words, lemmas, Universal Part-of-Speech tags, language-specific tags, and dependency relations from the training set, all of which are converted into numerical representations for model processing. The vocabularies facilitate the conversion of words and tags into these representations, while the dependency relations are used to construct the target vectors for the parser. The unique values from the dataset are extracted and stored in TensorFlow's hash tables for more efficient tensor handling. Special tokens are added to equalize input sequence lengths, represent the root of the dependency parse tree (the dummy token that governs the main verb of the sentence), and denote unknown words, tags, or relations. These tables are utilized to convert between numerical IDs and their corresponding labels during both training and prediction.

A configuration management module was implemented to streamline experimentation and reproducibility. This module incorporates hyperparameter loading from a JSON file and manages essential data structures, such as hash tables, necessary for both model training and data preprocessing. By centralizing configuration parameters, the development process was optimized, facilitating efficient exploration of the hyperparameter space.

The vectorization process is a critical step in preparing the data for the neural network. This process transforms the dataset into a format where each element is a tuple of inputs and outputs.

The inputs consist of tokenized sentences, Universal Part-of-Speech tags, lemmas, language-specific tags, and sentence lengths. The outputs are word head indices and dependency relations. Word heads are cast to numbers (specifically, tf.int32 to meet the requirements of tf.lookup.StaticHashTable), while the remaining elements are encoded as integers using their respective hash tables. To ensure uniformity, sequences are either padded or truncated to a predetermined length, as specified in the configuration. Each sequence begins with a

dummy value, and an additional token is included in the sentence length to represent the root of the dependency parse tree.

The vectorized inputs and outputs are then used to generate a new dataset.

The data preparation process includes batching and shuffling steps to optimize the training of the neural network. In the batching step, the dataset is divided into smaller groups or *batches*. Each batch contains a certain number of examples that the model will process simultaneously. The size of these batches is a configurable parameter and can be adjusted based on the computational resources available.

The shuffling step randomizes the order of the examples in the dataset. This is done to ensure that the model does not learn any unintended patterns from the order of the examples, which could lead to overfitting. This step is performed before each epoch, i.e., each pass through the entire dataset, to ensure that the model is exposed to a different order of examples in each epoch.

This dataset, vectorized and batched, can be used for model training or prediction.

3 The model

This chapter introduces the primary neural network model, designed for dependency parsing.

Built with TensorFlow and the Keras API, the model incorporates several components. These include embedding layers for words, lemmas, POS, and language-specific tags, BiLSTM layers for sentence encoding, and MLP layers for transforming the sentence encodings into a form suitable for predicting arcs and relations between words. The model also features Biaffine layers, which take these transformed outputs and make the actual predictions of arcs and relations between words.

The model is equipped with a custom loss function and utilizes the Adam optimizer for training. It also includes metrics for monitoring the loss and accuracy during both training and evaluation.

3.1 Model Architecture

As it was already mentioned, the model is based on the architecture proposed in *Deep Biaffine Attention for Neural Dependency Parsing* (Dozat and Manning, 2017), which builds off the work from Kiperwasser and Goldberg (2016) with a few modifications. The graph-based algorithm is illustrated in Figure 1.

Dozat and Manning use a larger but more thoroughly regularized parser, with biaffine classifiers to predict arcs and labels. They use biaffine attention instead of bilinear or traditional MLP-based attention; a biaffine dependency label classifier; and apply dimension-reducing MLPs to each recurrent output vector r_i before applying the biaffine transformation. The biaffine mechanism is similar to traditional affine classifiers, where the vector of scores s_i for all classes equals the weight matrix W multiplied by single LSTM output state r_i (or other vector input) plus the bias term b:

$$s_i = W r_i + \mathrm{b} \quad \text{(Fixed-class affine classifier)} \tag{1}$$

In the biaffine mechanism, the weight matrix W in (1) is replaced by a $(d \times d)$ linear transformation of the stacked LSTM output $RU^{(1)}$ in (2) and a $(d \times d)$ transformation $Ru^{(2)}$ replaces the bias term b:

$$s_i^{(arc)} = \left(RU^{(1)}\right)r_i + \left(Ru^{(2)}\right) \quad \text{(Variable-class biaffine classifier)} \tag{2}$$

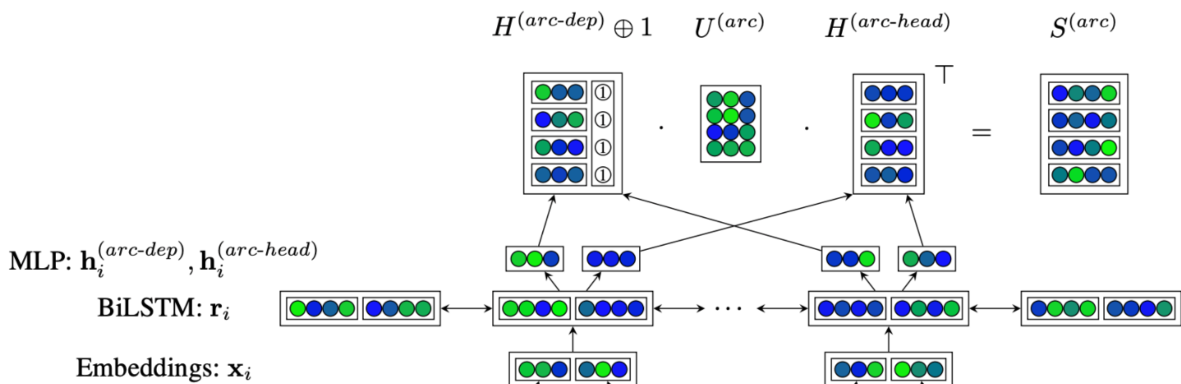

Figure 1: The graph-based architecture, proposed by Dozat and Manning.

Along with being simpler than the MLP-based approach, this has the conceptual advantage of directly modeling both the prior probability of a word j receiving any dependents in the term $r_j^{\top}u^{(2)}$ and the likelihood of j receiving a specific dependent i in the term $r_j^{\top}U_i^{(1)r}$. The authors also use a biaffine classifier to predict dependency labels given the gold or predicted head y_i:

$$s_i^{(label)} = r_{y_i}^{\top}U^{(1)}r_i + \left(r_{y_i} \otimes r_i\right)^{\top}U^{(2)} + b \tag{3}$$

(Fixed-class biaffine classifier)

Dozat and Manning point out that applying smaller MLPs to the recurrent output states before the biaffine classifier has the advantage of stripping away information not relevant to the current decision. They also claim that reducing dimensionality and applying a nonlinearity (4, 5, 6) increases parsing speed and decreases the risk of overfitting.

$$h_i^{(arc-dep)} = \text{MLP}^{(arc-dep)}(r_i) \tag{4}$$

$$h_j^{(arc-head)} = \text{MLP}^{(arc-head)}(r_j) \tag{5}$$

$$s_i^{(arc)} = H^{(arc-head)}U^{(1)}h_i^{(arc-dep)} + H^{(arc-head)}u^{(2)} \tag{6}$$

They call this a **deep** bilinear attention mechanism, as opposed to **shallow** bilinear attention, which uses the recurrent states directly.

MLPs are applied to the recurrent states before using them in the label classifier as well.

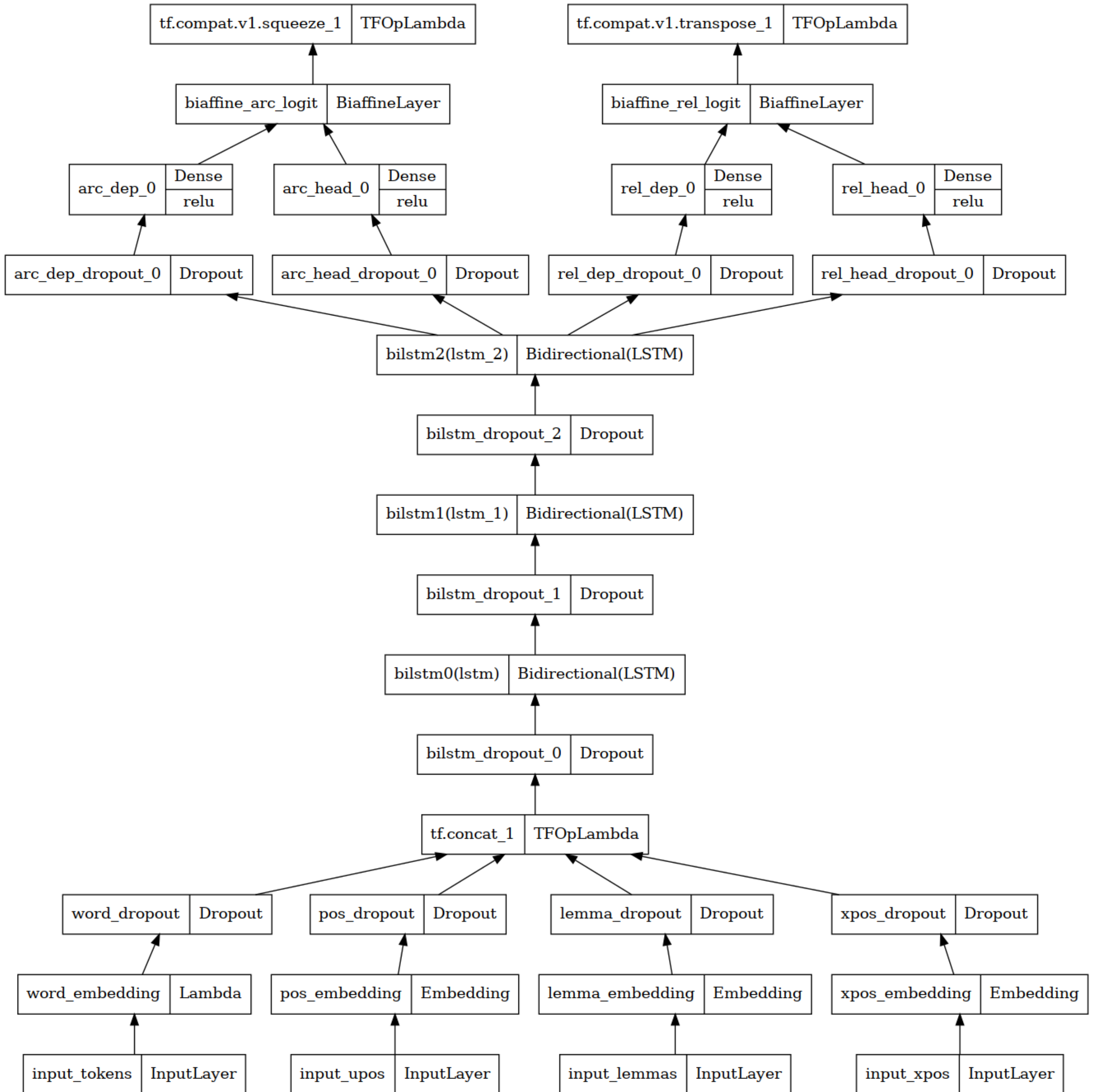

Figure 2: The architecture of the model with additional input layers.

3.2 Model Implementation

The parser proposed here is implemented following this architecture, with some modifications. Given that Bulgarian is a morphologically rich language, the model includes optional input layers for lemmas and morphological tags. These additions could potentially enhance its performance and warrant further evaluation. The model is constructed using the Keras API and features several custom components.

The primary component is responsible for constructing the neural network model. It includes methods for building the model's components, such as the embedding, BiLSTM, MLP, and biaffine attention layers. The model is compiled with a custom loss function, based on the sparse categorical cross-entropy loss object. This function computes the loss between the true and predicted values for the arcs and relations, returning a tensor that represents the average loss per example in the batch. This average loss is used during training to update the model's weights.

While the Adam optimizer is a popular choice for training deep learning models due to its adaptive learning rate, this implementation also includes the option to use exponential decay to potentially improve results.

The class tracks three metrics: the mean loss, the unlabeled attachment score (UAS), and the labeled attachment score (LAS).

Since TensorFlow does not provide a built-in biaffine layer, a custom one is implemented to perform the transformations. It follows the formulas for calculating the scores for potential arcs (6) and labels (3), and computes the weighted sum of the input tensors according to the weight matrix.

The model, comprising 31,036,906 parameters, is depicted in Figure 2.

The training process includes an option to log summaries in both TensorBoard and MLflow. This feature allows for the monitoring and comparison of all hyperparameter changes.

4 Experiments and Results

The model underwent testing on the Bulgarian UD treebank with varying hyperparameters, and the results were evaluated using the Unlabeled Attachment Score (UAS) and Labeled Attachment Score (LAS). The UAS measures the proportion of words correctly attached to their head, while the LAS measures the proportion of words correctly attached to their head with the correct dependency relation.

Optimal performance was achieved with a higher dropout rate of 0.5, as opposed to the 0.33 reported by Dozat and Manning. This can be attributed to the smaller size of the Bulgarian UD treebank compared to the treebanks used in their study, necessitating stronger regularization to prevent overfitting and enhance generalization.

Modifications to the learning and decay rates did not yield improved results. The best scores achieved using exponential decay, tested with values between 0.075 and 0.95, were 0.14% for UAS and 0.08% lower than the scores reported in Table 1. Similarly, increasing the batch size (and correspondingly the number of epochs) did not significantly affect performance. Specifically, training with a batch size of 512 and 160 epochs resulted in scores that were 1.04% lower for UAS and 1.02% lower for LAS compared to training with a batch size of 128 and 80 epochs with the same hyperparameters.

A substantial improvement (0.86% for UAS and 1.31% for LAS) was observed upon the inclusion of lemmas and language-specific morphological information as input layers. This enhancement is anticipated given the complexity of the Bulgarian language and the significance of morphological information in parsing it.

Another enhancement involved replacing the traditional embedding layer for input tokens with RoBERTa embeddings. This change leverages the pre-trained model (Liu et al., 2019) to generate contextualized embeddings, which capture richer semantic and syntactic information. Since RoBERTa is used in the current version of the parser only for token vectorization, the improvement is not particularly large (0.21% for UAS and 0.49% for LAS). However, it still demonstrates the advantage of using large language models.

The model achieved a UAS of 93.32 and a LAS of 89.73 on the test dataset, thereby demonstrating its ability to accurately predict the dependency parse tree of a sentence. Table 1 compares the performance of this model with other models. It surpasses the NLP pipeline for Bulgarian, developed within the spaCy framework (Popov et al., 2020). The model also yields superior results (with a 1.90% increase on UAS and a 2.72%

increase on LAS) compared to the model by Dozat and Manning, which uses only two input layers and a dropout rate of 0.33. It is also ahead of NLP-Cube (Boros et al., 2018) and UDPipe 2.0 (Straka, 2018). However, it still falls short of UDify (Kondratyuk and Straka, 2019). One reason for this could be that UDify is trained multilingually. Nevertheless, even when trained solely on Bulgarian, UDify's results are closely matched, suggesting that its primary advantage lies in the use of the BERT self-attention model. It is worth noting that the presented model is considerably smaller in size (especially when trained without RoBERTa embeddings) compared to UDify.

Model	UAS	LAS
spaCy	88.95	83.03
Biaffine w/o morph	91.21	86.52
NLP-Cube	92.47	88.93
UDPipe 2.0	92.82	89.70
Biaffine with morph	93.32	89.73
UDify	95.54	92.40

Table 1: Results on Bulgarian UD dataset

5 Future Work

The model performs well on the Bulgarian UD treebank, with its results approaching those of state-of-the-art parsers. However, further improvements are necessary. Potential areas for enhancement include:

- Hyperparameter tuning: The model's hyperparameters can be further optimized to improve its performance. This includes (but not restricted to) the learning rate, the number of layers in the BiLSTM, the number of units in the MLPs, and the dropout rate.
- Better embeddings: The model can be improved by using better word embeddings, leveraging pre-trained large language models.

Another crucial step involves integrating POS and XPOS annotations. Currently, the model utilizes the CLASSLA library (Ljubešić and Dobrovoljc, 2019; Terčon and Ljubešić, 2023) to perform POS and XPOS tagging, which are then used as inputs for sentence prediction. Therefore, the next development step is to directly incorporate this functionality by training a morphological tagger.

While the focus of this study was specifically on Bulgarian, the model should be evaluated with other highly inflected languages to determine if the inclusion of lemmas and morphological tags improves performance for these languages as well.

6 Conclusion

The implementation of the neural network-based dependency parser, utilizing TensorFlow and Keras, gave near state-of-the-art results (UAS: 93.32, LAS: 89.73). The parser underwent training and evaluation on the Bulgarian Universal Dependencies dataset, yielding competitive results and thereby demonstrating the efficacy of the proposed architecture. Although the original model by Dozat and Manning was evaluated on considerably larger datasets and languages with simpler morphological structures, the results are comparable. For instance, the parser's results for Bulgarian outperforms the original model's scores for Chinese and Czech, the latter possessing one of the largest treebanks. The parser can predict the dependency syntax structure of Bulgarian sentences, and the *displacy* module from the *spacy* library can visualize these predictions. The parser's performance can be further enhanced by optimizing its hyperparameters and employing superior word embeddings.

References

Tiberiu Boros, Stefan Daniel Dumitrescu, and Ruxandra Burtica. 2018. NLP-Cube: End-to-End Raw Text Processing With Neural Networks. In *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 171–179, Brussels, Belgium. Association for Computational Linguistics.

Timothy Dozat and Christopher D. Manning. 2017. Deep Biaffine Attention for Neural Dependency Parsing. In *International Conference on Learning Representations*.

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Eliyahu Kiperwasser and Yoav Goldberg. 2016. Simple and accurate dependency parsing using bidirectional LSTM feature representations. *Transactions of the Association for Computational Linguistics* 4: 313-327. MIT Press.

Dan Kondratyuk and Milan Straka. 2019. 75 Languages, 1 Model: Parsing Universal Dependencies Universally. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2779–2795, Hong Kong, China. Association for Computational Linguistics.

Geert-Jan Kruijff. 2002. Formal and computational aspects of dependency grammar: History and development of DG. Technical report, ESSLLI-2002.

Sandra Kübler, Ryan McDonald and Joakim Nivre, 2009. Dependency Parsing. *Synthesis lectures on human language technologies 2*. Morgan & Claypool Publishers

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *https://arxiv.org/abs/1907.11692*

Nikola Ljubešić and Kaja Dobrovoljc. 2019. What does Neural Bring? Analysing Improvements in Morphosyntactic Annotation and Lemmatisation of Slovenian, Croatian and Serbian. In *Proceedings of the 7th Workshop on Balto-Slavic Natural Language Processing*, pages 29–34, Florence, Italy. Association for Computational Linguistics.

Marie-Catherine de Marneffe, Christopher Manning, Joakim Nivre, and Daniel Zeman. 2021. Universal Dependencies. *Computational Linguistics 2021; 47 (2)*: 255–308.

Petya Osenova and Kiril Simov. 2015. Universalizing BulTreeBank: a Linguistic Tale about Glocalization. In *Proceedings of BSNLP 2015*, Hissar, Bulgaria.

Alexander Popov, Petya Osenova, and Kiril Simov. 2020. Implementing an End-to-End Treebank-Informed Pipeline for Bulgarian. In *Proceedings of the 19th International Workshop on Treebanks and Linguistic Theories*, pages 162–167, Düsseldorf, Germany. Association for Computational Linguistics.

Milan Straka. 2018. UDPipe 2.0 Prototype at CoNLL 2018 UD Shared Task. In *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 197–207, Brussels, Belgium. Association for Computational Linguistics.

Luka Terčon and Nikola Ljubešić. 2023. CLASSLA-Stanza: The Next Step for Linguistic Processing of South Slavic Languages. arXiv:2308.04255.

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 Gokirmak, Anna Nedoluzhko, Silvie Cinková, Jan Hajič jr., Jaroslava Hlaváčová, Václava Kettnerová, Zdeňka Urešová, et al. 2017. CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 1–19, Vancouver, Canada. Association for Computational Linguistics.

Daniel Zeman, Jan Hajič, Martin Popel, Martin Potthast, Milan Straka, Filip Ginter, Joakim Nivre, and Slav Petrov. 2018. CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies. In *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 1–21, Brussels, Belgium. Association for Computational Linguistics.