Department of Informatics

MSc in Computer Science

MSc thesis

"Exploring deep neural network models of syntax with a focus on Greek "

Emmanouil Kyriakakis

EY1604

Supervisors: Ion Androutsopoulos, Ryan McDonald

Athens, June 2018

Acknowledgments

I would like to thank my supervisors Ion Androutsopoulos and Ryan McDonald for providing me the chance to get involved in a very interesting field of NLP and for their valuable guidance throughout the development of my thesis. Also, I would like to thank AUEB's Natural Language Processing Group for the hardware resources they provided me during the work of my thesis.

Abstract

Part-of-speech tagging (POS tagging) and dependency parsing are fundamental syntax tasks in the field of Natural Language Processing. POS tagging is the process of assigning a tag (e.g. noun, verb, etc.) to each word in a sentence. Dependency parsing aims to automatically analyze the dependency structure of a sentence. This thesis involves the implementation and evaluation of a POS tagger and a dependency parser, with a focus on Greek. Both models utilize deep neural network architectures in order to produce state of the art performance. Our POS tagger produces near state of the art performance in the Greek language. Our dependency parser produces state of the art performance in the Greek and near state of the art performance in English and Bulgarian.

Περίληψη

Η επισημείωση μερών του λόγου (part-of-speech tagging) και η εξαγωγή συντακτικών δέντρων εξαρτήσεων (dependency parsing) είναι θεμελιώδεις συντακτικές διεργασίες του πεδίου της Επεξεργασίας Φυσικής Γλώσσας (ΕΦΓ). Η επισημείωση μερών του λόγου είναι η διαδικασία ανάθεσης ετικετών (π.χ ρήμα, ουσιαστικό, άρθρο κ.λπ.) σε κάθε λέξη μίας πρότασης. Η εξαγωγή συντακτικών δέντρων εξαρτήσεων στοχεύει στην αυτόματη ανάλυση της συντακτικής δομής μιας πρότασης. Στη διάρκεια της συγκεκριμένης διπλωματικής εργασίας αναπτύξαμε και αξιολογήσαμε ένα μοντέλο επισημείωσης μερών του λόγου, καθώς και ένα μοντέλο εξαγωγής συντακτικών δέντρων εξαρτήσεων εστιάζοντας κυρίως στην ελληνική γλώσσα. Και τα δύο μοντέλα χρησιμοποιούν αρχιτεκτονικές βαθέων νευρωνικών δικτύων (deep learning) που οδηγούν σε διεθνώς ανταγωνιστικές επιδόσεις.

Η επίδοση του επισημειωτή μερών του λόγου που αναπτύξαμε είναι ελάχιστα κατώτερη από την καλύτερη επίδοση που αναφέρεται στην βιβλιογραφία για την ελληνική γλώσσα. Η επίδοση του μοντέλου μας για την εξαγωγή συντακτικών δέντρων εξαρτήσεων είναι ανώτερη από την καλύτερη επίδοση που αναφέρεται στην βιβλιογραφία για την ελληνική γλώσσα. Τέλος, το μοντέλο εξαγωγής συντακτικών δέντρων εξαρτήσεων που αναπτύξαμε επιτυγχάνει ανταγωνιστικά αποτελέσματα σε σύγκριση με το καλύτερο μοντέλο που αναφέρεται στην βιβλιογραφία για την αγγλική και την βουλγαρική γλώσσα.

Table of Contents

1. Introduction....6
    1.1 Part-of-Speech tagging....6
    1.2 Dependency parsing....7
    1.3 Dependency trees....9
    1.4 Projectivity....9
    1.5 Graph-based parsing....10
    1.6 Evaluation metrics....12
2. Related work....13
  2.1 The parser of Kiperwasser et al....14
    2.1.1 Input encoder....14
    2.1.2 MLP score function, parsing and hinge loss objective....16
  2.2 The parser of Dozat et al....17
    2.2.1 Character-level embeddings....17
    2.2.2 Input encoder and head/modifier ReLu layers....18
    2.2.3 Biaffine arc/label classifiers....20
    2.2.4 Training details....22
    2.2.5 The POS/XPOS tagger of Dozat et al....22
3. Our work....25
  3.2 POS/XPOS tagger....26
    3.2.1 Input and Model architecture....26
    3.2.2 Training details....27
  3.3 Dependency parser....28
    3.3.1 Neural architectures investigation....28
    3.3.2 Input encoder and head/modifier ReLu layers....29
    3.3.3 MLP score function....32
    3.3.4 Parsing using Edmond’s decoder....33
    3.3.5 Training details....34
4. Experiments....35
    4.1 CoNLL-U format....35
    4.2 POS/XPOS tagger results....37
    4.3 Dependency parser results....37
5. Conclusions and future work....40
    5.1 Conclusions....40
    5.2 Future work....41
References....42

1. Introduction

In this chapter we describe briefly the Natural Language Processing (NLP) tasks of part-of-speech (POS) tagging and dependency parsing. The dependency parsing section draws heavily on terminology and characterizations outlined in the “Dependency parsing” book (Kubler et al., 2009). POS tagging is a fundamental NLP task that assigns a tag to each word in a sentence. Dependency parsing is another NLP task that aims to automatically analyze the dependency structure of a given input sentence. A dependency structure can be represented as a directed acyclic graph (DAG) where nodes and edges correspond to words and dependency arcs respectively. We explain also the notion of projectivity regarding the dependency structures. Finally, we focus on data-driven dependency parsing approaches. These approaches include transition-based and graph-based parsing algorithms.

1.1 Part-of-Speech tagging

POS tagging is the process of assigning a tag to each word in a sentence. *Sentence splitting* and word *tokenization* are usually performed before, or as part of the tagging process. Sentence splitting or *sentence boundary disambiguation* is an NLP task that decides where sentences begin and end. Tokenization is the process of breaking up a sentence into pieces (tokens) like words, punctuation symbols etc.

Tagging is a disambiguation task. Words are ambiguous since they have more than one part-of-speech tags and the goal is to find the correct tag for each situation. For example, the word *book* is a verb in the sentence “book that ticket” but a noun in the sentence “read the green book”. The problem of POS tagging is to resolve these ambiguities. A tagging algorithm takes as input a sequence of tokens and produces a sequence of tags as output. Each token of the sentence is assigned a tag. Common algorithms that are used for POS tagging include Hidden Markov Models (Eisner 1996), Conditional Random Fields (Lafferty et al., 2002) and deep neural networks (Goldberg 2017).

POS tags are very useful because they encode a large amount of information regarding a word and its neighbors. Knowing whether a word is a verb or noun gives us a lot of information for the neighboring words. For example, nouns are preceded by determiners and adjectives whereas verbs are usually preceded by nouns. Also, knowing the POS tag of a word tells us a lot about the syntactic structure around the word. For instance, nouns are usually parts of noun phrases.

The information encoded in POS tags is very useful for other NLP tasks such as dependency parsing (Dozat et al., 2017), information retrieval (Chowdhury et al., 1993) and speech synthesis (Ming et al., 2011). The POS tags influence the possible morphological affixes and so can influence stemming for information retrieval. A POS tag of a word is important for producing pronunciations in speech synthesis or recognition. For example, the word *content* is pronounced differently when it is a noun (CONtent) and differently when it is an adjective (conTENT).

1.2 Dependency parsing

Dependency representations of syntax are well established in the field of descriptive linguistics, with many different formalisms such as Word Grammars (Hudson 1984), and Meaning Text Theory (Mel'čuk 1987). The basic assumption of all dependency grammar varieties is that syntactic structure consists of words linked by binary, asymmetrical relations called dependency relations. In every dependency relation there is a word, called the dependent, and another word on which it depends, called the head. The image below depicts a dependency structure for a simple English sentence. Dependency relations are represented by arrows pointing from the head to the dependent. Furthermore, each arrow has a label, indicating the dependency type.

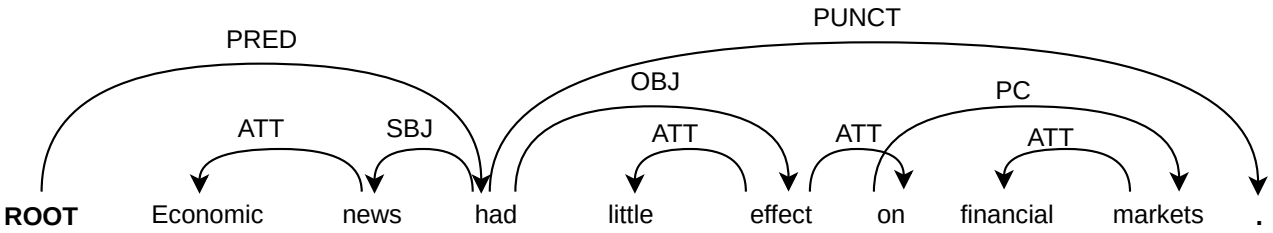

Figure 1.1: Dependency structure of an English sentence. Example from Kubler et al. (2009).

Dependency parsing has attracted considerable interest from researchers and developers in the NLP field. One reason for this popularity is the fact that dependency-based syntactic representations seem to be useful in other applications of language technology, such as machine translation (Quirk et al., 2006). The task of a dependency parser is to produce a labeled dependency structure (figure 1.1), where the words of the sentence (including the artificial word ROOT) are connected by typed dependency relations (e.g. object, subject etc). Thus, we can define the dependency parsing problem as that of mapping an input sentence *S,* consisting of the words $w_0, w_1, \ldots, w_n$ (where w_0 is the ROOT), to its dependency graph *G*.

The approaches to solve the parsing problem can be divided into two classes, *data-driven* and *grammar-based* respectively. A *data-driven* approach uses *machine learning* on linguistic data in order to parse new sentences. A *grammar-based* approach relies on a formal grammar, defining a formal language, in order to decide whether a given input sentence is in the language defined by the grammar or not. *Data-driven* methods for dependency parsing have attracted the most attention in recent years.

Data-driven or supervised methods presume that the input sentences used for machine learning have been annotated with their correct dependency structures. In supervised dependency parsing, there are two different problems that need to be solved computationally. The first is the *learning problem*, which is the task of learning a *parsing model* from a representative sample of sentences and their dependency structures. The second is the *parsing problem,* which is the task of applying the learned model in order to parse a new sentence. So we can view supervised dependency parsing as two problems, following the discussion of Kubler et al. (2009):

- **Learning:** Given a training set *S* of sentences (annotated with the correct dependency structures), learn a parsing model *M* that can be used to parse new sentences.
- **Parsing:** Given a parsing model *M* and a sentence *s,* derive the optimal dependency graph *G* for *s* according to *M.*

The two main classes of data-driven methods are *graph-based* and *transition-based* respectively.

Graph-based approaches define a space of candidate dependency graphs for a sentence. The learning problem is to derive a model for assigning scores to the head-modifier pairs. The aforementioned model will calculate $(n+1)^2$ arc-scores for a sentence of length n (+1 for the artificial word ROOT), i.e., the model will score every possible head-modifier pair in the sentence. The parsing or decoding problem is the task of finding the highest-scoring dependency graph for the input sentence, given the scores produced by the learned model. This is often called maximum spanning tree parsing, since the problem of finding the highest-scoring dependency graph is equivalent to the problem of finding a maximum spanning tree in a dense graph.

Transition-based approaches is another paradigm that is well studied (Nivre et al., 2008). Transition-based parsing has configurations and transitions between configurations. A sequence of configuration-transition pairs defines a tree and the goal is to learn the correct transitions out of configurations. The idea is that a sequence of valid transitions, starting in the initial configuration for a given sentence and ending in one of several terminal configurations, defines a valid dependency tree for an input sentence. There are many instantiations for transition-based parsing but the most common is the shift-reduce parsing technique (Shieber et al., 1983).

1.3 Dependency trees

A common assumption is that valid dependency graphs are represented as trees. Following the discussion of Jurafsky et al. (2017), a dependency tree is a graph G = (V, E), where V is a set of vertices and E is a set of pairwise connections (arcs) of the vertices in V. The number of vertices is equal to the number of words in a given sentence plus one more vertex for the artificial word ROOT. The arcs of the set E capture the head-dependent and grammatical function relationships between the vertices (words) in the V set.

A valid dependency tree is a directed acyclic graph (DAG) that satisfies the following constraints:

1. There is a unique ROOT node with no incoming arcs.
2. Each vertex has exactly one incoming arc.
3. There is a unique path from the ROOT node to each vertex (word) in V.

The aforementioned constraints ensure that each word has a single head word (incoming arc) but a head word may have multiple dependent words (outgoing arcs). Also, the dependency structure is connected and there is a unique root node from which there is a unique directed path to every other vertex in the graph.

1.4 Projectivity

Projectivity is an additional constraint that is derived from the order of the words in a given sentence. Following the discussion of Jurafsky et al. (2017), an arc from a head to a dependent is projective if there is a path from the head to every other word that lies between the head and the dependent in the sentence. If all arcs of a dependency tree are projective then the dependency tree is projective. However, if a dependency tree has at least one non-projective arc then the dependency tree is called non-projective. Below we have drawn two dependency trees of English sentences that are projective (figure 1.2) and non-projective (figure 1.3) respectively.

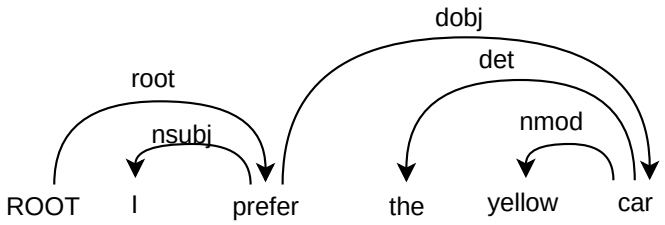

Figure 1.2: Projective English sentence.

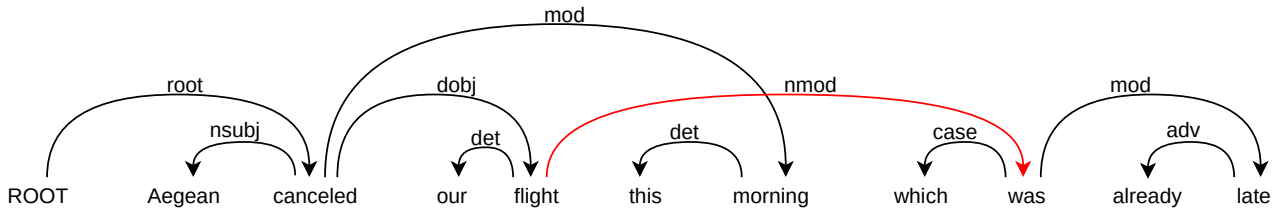

Figure 1.3: Non-projective English sentence. Example from Jurafsky et al. (2017).

As we can see in the first (projective) sentence all arcs satisfy the projectivity constraint. In contrast, the second (non-projective) sentence has an arc from the head *flight* to the dependent *was* that is non-projective since there is no path from the head *flight* to the intervening words *this* and *morning*. In general, a dependency tree is projective if it can be drawn with no crossing edges. In our example there is no way to connect *flight* to its dependent *was* without crossing the arc that links *morning* to its head.

Our concern with projectivity arises mainly because there are computational limitations regarding the transition-based approaches. Projectivity usually allows for polynomial run time parsing algorithms.

1.5 Graph-based parsing

Graph-based dependency parsing approaches search through the space of possible trees of a given sentence to find a tree (or trees) that maximize some score. Graph-based methods encode the search space as directed graphs and sometimes utilize methods from graph theory to search the space for optimal solutions. Unlike transition-based methods that parametrize models over transitions used to construct a tree, graph-based methods parametrize models over substructures of a dependency tree.

We can define a graph-based method through a model $M=(\lambda, \Gamma, h)$ where λ is the model parameters (we learn them during training), Γ is a set of constraints on permissible structures and h is a parsing algorithm. Γ is simply a set of constraints that force the model to produce well-structured dependency graphs (spanning trees originated from a unique root).

A fundamental part of graph-based parsing systems is the notion of the score of a dependency tree $G=(V,E)\in G_s$, where G_s is the set of all possible trees for a sentence s:

$$score(G)=score(V,E)\in\mathbb{R}$$

This score (real value) represents how likely it is that a particular tree is the correct analysis of the sentence s. The fundamental property of graph-based parsers is that this score is assumed to factor through the scores of sub-graphs of G:

$$score(G) = f(\Psi_1, \Psi_2, ..., \Psi_n) \forall \Psi_i \in \Psi_G$$

The function *f* is some function over sub-graphs Ψ and Ψ_G represents the relevant set of subgraphs of G. Most commonly, the function *f* is equivalent to a summation over factor parameters and the score formula can be rewritten as:

$$score(G) = \sum_{\Psi \in \Psi_G} \varphi_\Psi$$

The most common graph-based approach is the *arc-factored* models which use the smallest and most basic parametrization over single dependency arcs. We can define arc-factored models for a given dependency tree $G = (V, E)$ as follows:

1. $\Psi_G = E$
2. $\varphi_\Psi = \varphi_{(w_i, r, w_j)} \in \mathbb{R} \forall (w_i, r, w_j) \in E$

Thus, arc-factored models assign a real value parameter to every labeled arc in the tree. So we can rewrite the score formula as follows:

$$score(G) = \sum_{(w_i, r, w_j) \in E} \varphi(w_i, r, w_j)$$

With $\varphi(w_i, r, w_j)$ we denote the score that the model assigns to the dependency arc (w_i, r, w_j).

To solve the parsing problem we need an algorithm that finds the tree whose arc scores sum to the maximum value. We can define the parsing (decoding) function as:

$$h(S, \Gamma, \lambda) = argmax_{G=(V,E) \in G_S} \sum_{(w_i, r. w_j) \in E} \varphi(w_i, r, w_j)$$

The argmax problem is equivalent to find the maximum spanning tree with the additional constraint that one tree node (root) should have no incoming edges in order to form a valid dependence tree.

There are two polynomial time algorithms to solve the parsing problem. First, Eisner's decoder (Eisner 1996) which is a CKY-like $O(n^3)$ dynamic programming algorithm that is used for projective dependency parsing. Second, in order to parse non-projective sentences we can use Edmond's decoder (Chu and Liu, 1965 and Edmonds, 1967) which finds the maximum spanning tree over all possible spanning trees of a sentence with $O(n^3)$ complexity. Given a sentence of length n we first select the n-1 highest-scored arcs. If there is no cycle on the selected arcs we are done. If a cycle is detected we contract it by merging in a new node all the nodes that it contains. We also recalculate the ingoing and outgoing arcs for the newly created node and keep pointers to the original arcs. We recursively repeat the same procedure until we find a solution with no cycles. Then, we

can backtrack using the pointers we have stored during cycle(s) contraction and find the edges of the maximum spanning tree.

Beyond the 1st order arc-factored models where each head-modifier arc is assumed to be independent other higher-order factorizations have been investigated in the literature. Though arc-factored models are appealing computationally, they are not justified linguistically as their underlying arc independence assumption is simply not valid. For the purpose of this thesis, though, we focus on 1st order arc-factored models that are the most commonly used.

1.6 Evaluation metrics

In order to evaluate the performance of a dependency parser, we compare the produced output of the parser on a test treebank with the gold standard annotation found in the treebank. The two most commonly used evaluation metrics are:

- **Unlabeled Attachment Score (UAS):** the percentage of words in an input that are assigned the correct head.
- **Labeled Attachment Score (LAS):** the percentage of words in an input that are assigned the correct head **and** the correct dependency relation.

2. Related work

The creation of the tagged Brown Corpus and the Penn treebank (Marcus et al., 1993) led to a number of advances in POS tagging. The Brill tagger (Brill 1992) was an early accurate rule based system, that used a form of machine learning (called "transformation based learning") to learn its rules. The Ratnaparkhi tagger (Ratnaparkhi 1996) was one of the first linear discriminative models (maximum entropy/logistic regression) that utilized rich features to predict POS tags. Furthermore, the TnT tagger (Brants 2000) was a generative HMM model. These taggers were highly used for years, with the Stanford tagger (Toutanova et al., 2003) probably being close to the state-of-the art for a decade.

The first statistical models for dependency parsing that were not based on context-free grammars (CFG) were proposed by Eisner (1996). Eisner developed three different probabilistic models for dependency parsing and an efficient $O(n^3)$ dynamic programming parsing algorithm. He presented a flexible probabilistic parser that simultaneously assigned both POS tags and a bare-bones dependency structure. Each word was linked to a single parent and the head of the sentence was linked to the EOS (end of the sentence). Crossing links and cycles were not allowed.

Arc-factored discriminative linear models with rich features were proposed by McDonald et al. (2005a). The proposed arc-factored model had a rich feature set including word and POS tag information for parent and child nodes, POS tag information of the surrounding and between parent-child nodes and distance/direction information of parent-child dependencies. Eisner's parsing algorithm was used for parsing. The same year, McDonald et al. (2005b) formalized weighted dependency parsing as searching for maximum spanning trees (MSTs) in directed graphs. The Chiu-Liu-Edmonds algorithm was utilized for MST extraction. The proposed MST parsing algorithm was capable of non-projective dependency parsing in contrast to Eisner's algorithm that can produce only projective dependency structures.

In subsequent work McDonald et al. (2006, 2007) investigated several non-projective parsing algorithms for dependency parsing, providing novel polynomial time solutions under the assumption that each dependency decision is independent of all the others (arc-factored models). Furthermore, they also investigated higher-order non-projective algorithms and found that exact non-projective dependency parsing is intractable for any model richer than the edge-factored model (1^{st} order), but the intractability can be circumvented with new approximate parsing algorithms.

Koo et al. (2010) presented algorithms for higher-order projective dependency parsing. The proposed parsers utilized both sibling-style and grandchild-style interactions and were efficient since they required $O(n^4)$ time. From 2011 until Kiperwasser et al.'s

(2016) model most of the field focused on incremental improvements to graph-based parsing and on multilingual and low-resource parsing.

2.1 The parser of Kiperwasser et al.

To the best of our knowledge Kiperwasser et al. (2016) were the first that utilized LSTMs as feature extractors for the task of dependency parsing. They presented a transition-based and a graph-based approach for dependency parsing using bidirectional LSTMs as feature extractors. The aforementioned architecture was then used by many other authors as the backbone for the development of their dependency parsing systems. The reported results of their transition-based and graph-based implementations on English and Chinese treebanks were almost the same. We have chosen to present their graph-based approach since graph-based approaches can parse also non-projective sentences which is very important for highly non-projective languages like Greek.

2.1.1 Input encoder

Traditional first-order graph-based parsers use a core of features that usually take into account the word and the POS tag of the head and the modifier, as well as POS-tags of the words around the head and the modifier, POS tags of the words between the head and the modifier and the distance and direction between the head and the modifier. In the literature there are such feature sets based on hand-crafted feature functions. But, feature-engineering is a tedious task that requires a lot of expertise in the domain of the specific task. Kiperwasser et al.'s model attempts to alleviate parts of the feature function design problem by moving to neural network models using recurrent inputs, enabling the modeler to focus on a small set of "core" features and leaving it up to the machine-learning machinery to come up with good feature combinations. The proposed feature extractors are based on bidirectional LSTMs that take into account both the past $x_{1:i}$ and the future $x_{i:n}$ elements regarding the i_{th} element of a sequence (sentence). Long short-term memory (LSTM) is a specific type of recurrent neural network (RNN) that can learn longer-term dependencies. For example, given an n-words input sentence s with words $w_1, w_2, ..., w_n$ together with the corresponding POS tags $t_1, t_2, ..., t_n$. Each word w_i and POS tag t_i are associated with embeddings vectors $e(w_i)$ and $e(t_i)$ respectively. A sequence of input vectors $x_{1:n}$ is created where x_i is the concatenation of the corresponding word and POS tag embeddings:

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

The embeddings are trained together with the model. This encodes each word and POS tag in isolation disregarding its context. The word and POS tag embeddings concatenations are fed into the BiLSTM network (figure 2.1) in order to acquire a deep context-aware encoding for each word in the sentence.

The BiLSTM network is a stack of a forward and a backward LSTM. The only difference between the forward and the backward LSTM is the order in which the input vectors are fed. The forward LSTM consumes sequentially the input vectors. In contrast, the backward LSTM consumes the same input vectors but in reversed order. At each time step, the LSTM is fed with one input vector that corresponds to a word in the sentence and produces a hidden vector for the next time step and an output vector. Also, at each time step the LSTM's memory cell is updated. When the entire sentence is consumed from the LSTMs the output vectors are concatenated in order to get a deep context-aware encoding for each word in the sentence.

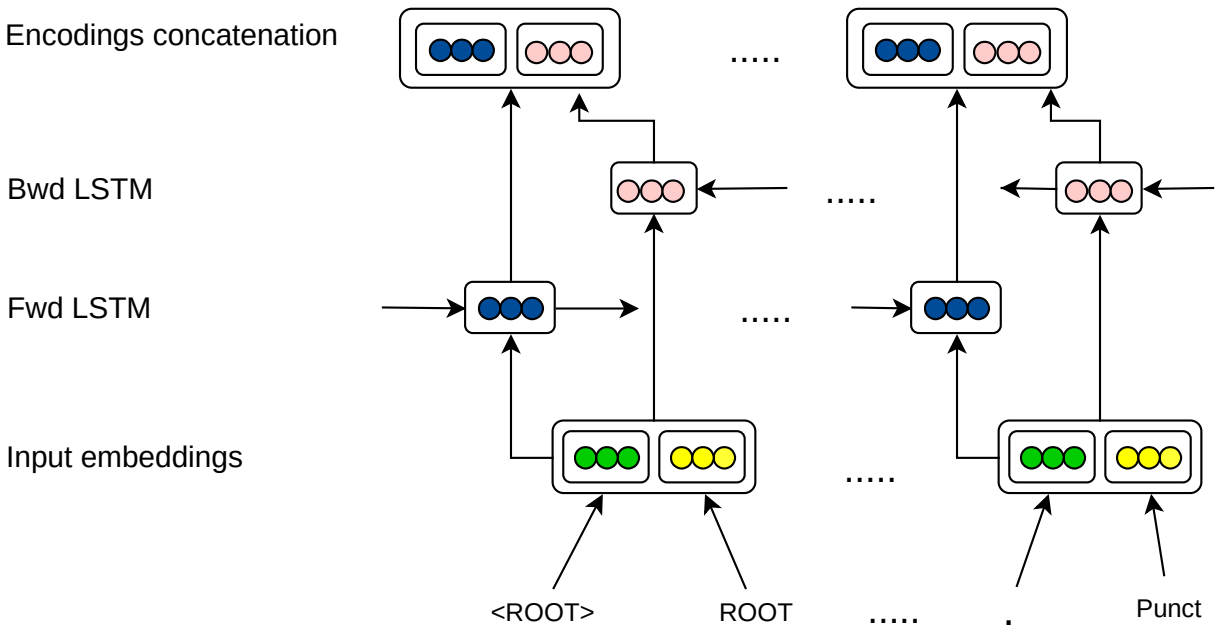

Figure 2.1: The input BiLSTM encoder of Kiperwasser et al. (2016). The special <ROOT> token is treated as the first word of the sentence.

2.1.2 MLP score function, parsing and hinge loss objective

Given a sentence of length n there are n^2 possible head-modifier pairs. Each possible head-modifier pair is scored (figure 2.2) via a multilayer perceptron (MLP) with one hidden layer using the following formula:

$$score(h,m) = W_2 \tanh(W_1[v(h) \oplus v(m)] + b_1) + b_2$$

The input to the MLP is the concatenation of the BiLSTM's encodings that correspond to the head and modifier words. Head-modifier relation labels are scored in a similar manner using the same encodings but a different MLP. The MLP produces a score for every label in the set of labels (figure 2.2).

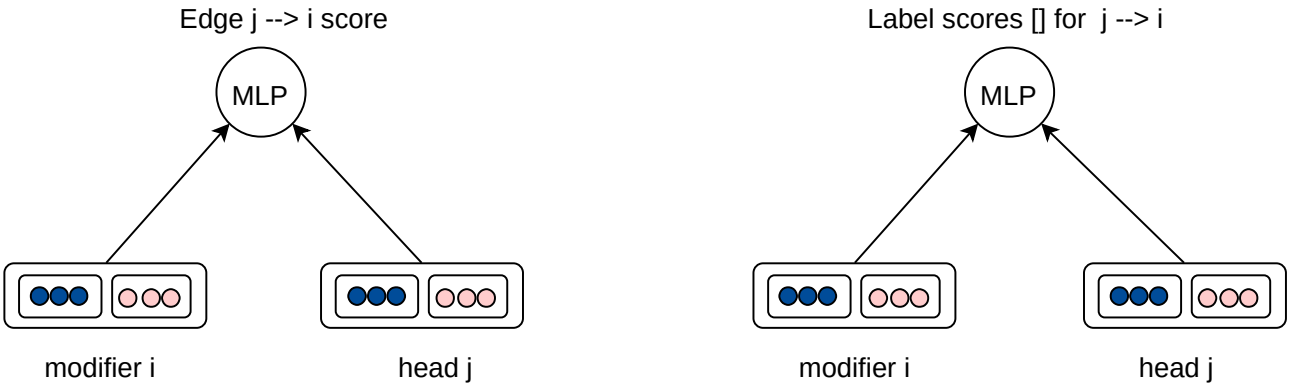

Figure 2.2: Scoring arcs and labels in the system of Kiperwasser et al. (2016).

Given the produced by the model arc scores for all the possible head-modifier pairs of a sentence s, the highest-scoring dependency tree y in the space $\mathcal{Y}(s)$ of valid dependency trees of S should be selected during parsing. The parsing process decomposes the score of a tree to the sum of the score of its head-modifier arcs (h,m) using the formula:

$$parse(s) = argmax_{y \in Y(s)} \sum_{(h,m) \in y} score(h,m)$$

The model utilizes Eisner's decoding algorithm to extract the highest-scoring projective tree during test time.

During training time a margin-based objective is defined. The goal is to separate the score of each gold (correct) tree y from the score of the highest-scoring incorrect tree y',

such that the former score will be higher than the latter and there will be at least a margin (set to 1) between them. The following hinge loss with respect to a gold tree y is used:

$$loss = max(0, 1 - \sum_{(h,m) \in y} MLP(v_h \oplus v_m) + max_{y' \neq y} \sum_{(h,m) \in y'} MLP(v_h \oplus v_m))$$

The BiLSTM-produced encodings for the head (v_h) and the modifier (v_m) are used as input.

A similar margin-based hinge loss is used for the labels. A different scoring function (MLP) but the same BiLSTM-produced encodings for the head (v_h) and the modifier (v_m) are used. The label loss is computed using the gold tree arcs rather than the predicted-ones. Both arc and label losses are optimized together. This can be seen as an instance of multitask learning. Training the BiLSTM feature encoder to be also good at predicting arc-labels significantly improves the parser's unlabeled accuracy.

2.2 The parser of Dozat et al.

To the best of our knowledge the graph-based parser of Dozat et al. (2017) holds the state-of-the art performance (May 2018) for the majority of universal dependency treebanks. This parser utilizes a deep BiLSTM feature encoder and two biaffine classifiers for the arcs and the labels respectively. As input the model uses word and POS tag embeddings. During test time the model utilizes a state-of-the-art POS tagger also developed by Dozat et al. (2017) to obtain predictions for POS and XPOS tags. Dozat et al. showed that a better POS tagger improves the performance of a dependency parser.

2.2.1 Character-level embeddings

For most of the languages, especially those with rich morphology, adding a representation made from sequence of characters improves the performance of POS tagging and dependency parsing tasks (Dozat et al., 2017). A word is represented as a sequence of its characters including a special start and end symbol. For example, the word dog is represented as the sequence of the following characters: <w>, d, o, g, </w>. Each character is associated with a trainable vector embedding and a sequence of character embeddings that represents a word is fed to a unidirectional LSTM. A common practice in the literature is to use the last hidden state of the LSTM as the character-based representation of the word. Dozat et al. use a different approach (figure 2.3) that combines

all the recurrent states produced by the LSTM using an attention mechanism and the final LSTM's cell state as well. More specifically, a linear attention over the stack H of the recurrent/hidden states (viewed as columns of H) produced by the LSTM is computed and concatenated to the final LSTM's cell state C_n. The produced vector is then projected to the desired dimension using a linear projection layer. The following formulas are used:

$$a = softmax\left(H\, w^{(attn)}\right)$$

$$\widetilde{h} = H^T a$$

$$\hat{v} = W\left(\widetilde{h} \oplus C_n\right) + b$$

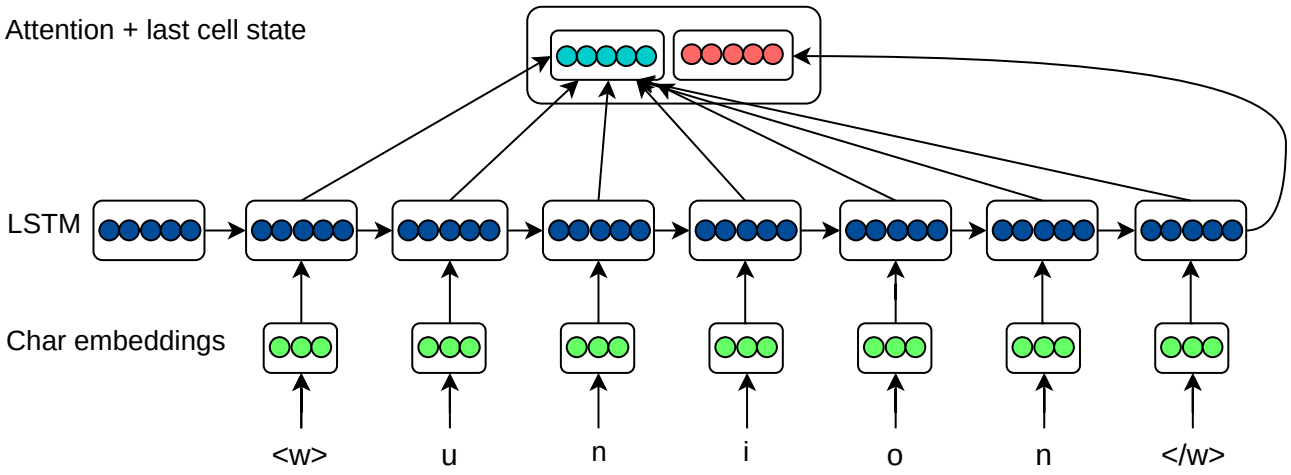

Figure 2.3: Producing character-level word embeddings in the model of Dozat et al. (2017).

2.2.2 Input encoder and head/modifier ReLu layers

The model of Dozat et al. uses as input the concatenation of word and POS tag embeddings (figure 2.4) for each word in a sentence. The word embeddings are constructed using the element-wise summation of trainable word embeddings, pretrained word embeddings and trainable character-level word embeddings (Section 2.2.1). The POS tag embeddings are constructed using the element-wise summation of the trainable POS (universal) and XPOS (language-specific) tag embeddngs. A sequence of input vectors $x_{1:n}$ is created where x_i is the concatenation of the word and POS tag embeddings.

$$x_i = v_i^{(word)} \oplus v_i^{(tag)}$$

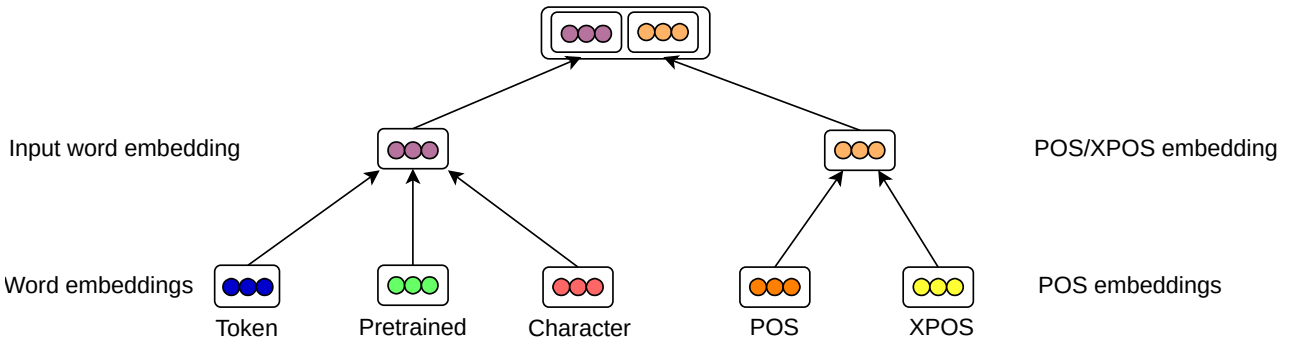

Figure 2.4: BiLSTM input vector of Dozat et al.'s parser (2017).

The use of pretrained word embeddings is a form of transfer learning that is commonly used in order to boost the performance in various NLP tasks. Pretrained word vectors are usually trained in large corpora like Wikipedia or/and Common Crawl. Dozat et al. use 100D word2vec (Mikolov et al., 2013) pretrained (on Wikipedia and Common Crawl) word embeddings[1].

Dozat et al.'s parser utilizes a deep BiLSTM feature encoder (figure 2.5) to learn non-linear features for dependency parsing. More concretely, given an n-words input sentence s with words $w_1, w_2, ..., w_n$ a sequence of input vectors $x_1, x_2, ..., x_n$ is derived from the concatenation of the respective words and tags embeddings. The input vectors are fed to a forward and a backward LSTM of depth three. Constructing deeper LSTMs is pretty straightforward since the output of a layer is fed as input to the next layer. The produced encodings of the forward and the backward LSTM are concatenated for every word in the sentence: $v_{1:n} = \vec{v}_{1:n} \oplus \overleftarrow{v}_{1:n}$.

The concatenated encodings are fed through four separate fully connected ReLU layers (leaky ReLu activation with a=0.1), producing four specialized vector representations: one for the word as a dependent seeking its head, one for the word as a head seeking all its dependents another for the word as a dependent deciding on its label and a fourth for the word as head deciding on the labels of its dependents. The Leaky ReLu is one of the most commonly used activation functions. Its definition is given by the formula: $f(x) = max(ax, x)$. The formulas below are used for the ReLu layers:

$$h_i^{(arc-head)} = LReLu(W_1 v_i + b_1)$$

$$h_i^{(arc-dep)} = LReLu(W_2 v_i + b_2)$$

1 Pretrained w2v embeddings are provided by the CoNLL 2017 Shared Task: https://lindat.mff.cuni.cz/repository/xmlui/handle/11234/1-1989

$$h_i^{(rel-head)} = LReLu\left(W_3 v_i + b_3\right)$$

$$h_i^{(rel-dep)} = LReLu\left(W_4 v_i + b_4\right)$$

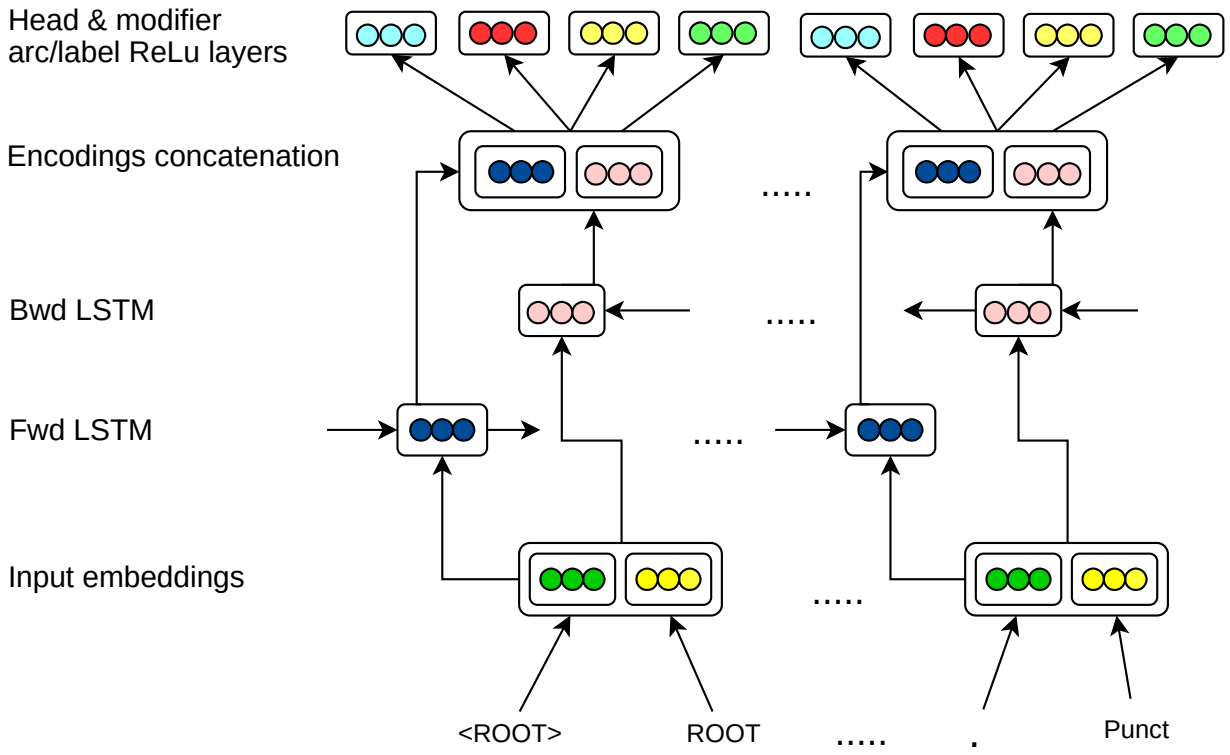

Figure 2.5: The input BiLSTM encoder of Dozat et al.'s parser (2017).

2.2.3 Biaffine arc/label classifiers

In order to predict an arc from a modifier i to a head j, all arcs from the modifier i to every possible head should be scored. A biaffine classifier is used based on the following formula:

$$s_i^{(arc)} = H^{(arc-head)} W^{(arc)} h_i^{(arc-dep)} + H^{(arc-head)} b^{T(arc)}$$

$H^{(arc\text{-}head)}$ is the stack of all vectors of the sentence that are produced by the arc-head ReLu layer. The biaffine formula produces a score for each possible head j for the modifier i. Then, the head j with the highest score is predicted for the modifier i forming the edge i→j: $y_i^{(arc)} = argmax_j\, s_{ij}^{(arc)}$. Note that both biaffine formula's terms have an intuitive interpretation.

The first relates to the probability of word j being the head of word i given the information in both the vector of the head and the vector of the dependent (modifier). The second term relates to the probability of word j being the head of word i given only the information in the head's vector.

After predicting the head j for the word i, another biaffine transformation is used to predict the relation label of the arc. This time the (rel) hidden vectors are used to predict a label. Given that we have already predicted the head (as $y_i^{(arc)}$), we now consider only the vector of that particular head (and the vector of the dependent), but we output a score for each possible label of the edge. The following formula is used:

$$s_i^{(rel)} = h_{y_i^{(arc)}}^{T(rel-head)} U^{(rel)} h_i^{(rel-dep)}$$
$$+ W^{(rel)} \left(h_i^{(rel-dep)} \oplus h_{y_i^{(arc)}}^{(rel-head)} \right) + b^{(rel)}$$

The biaffine formula produces a score for every possible relation label regarding the edge i→j. The relation with the highest score is then predicted: $y_i^{(rel)} = argmax_j s_{ij}^{(rel)}$. Again, each term of the biaffine formula has an intuitive interpretation. The first term relates to the probability of observing a label given the information in both the vector of the head and the vector of the dependent. The second term relates to the probability of observing a label given each one of the two vectors (for the head and dependent) independently, and the last bias term relates to the prior probability of observing a label.

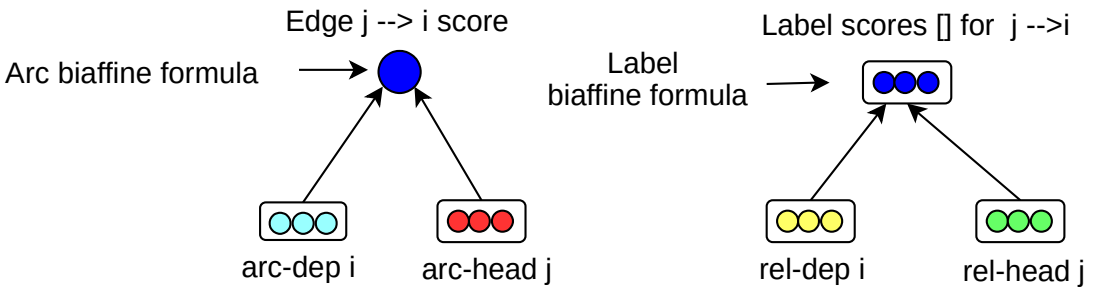

Figure 2.6: Scoring arcs and labels using biaffine classifiers in the system of Dozat et al. (2017).

2.2.4 Training details

The biaffine classifiers are trained jointly using cross-entropy losses that are summed together during optimization. Note, that in contrast to Kiperwasser et al.'s parser that uses the gold tree to train the label classifier, Dozat et al.'s. parser uses the predicted arcs to train the biaffine label classifier. The model uses Adam (Kinigma et al., 2015) as optimization algorithm. Adam (Adaptive Moment Estimation) computes adaptive learning rates for each parameter incorporating exponentially decaying average of past squared gradients and gradients, similar to momentum.

During training variational dropout (Gal and Grahami, 2016) of 33% is used through the entire network. Regarding the LSTM's regularization both input and recurrent units are dropped. In variational dropout the same dropout mask is used across the sequence/sentence. Furthermore, the (entire) word and tag embeddings are dropped independently with probability 33%. When one is dropped the other is scaled up to compensate. When both embeddings are dropped, the entire input vector is replaced with zeros. This additional dropout prevents the model from heavily relying on either word or tag embeddings.

The model is trained for up to 30000 steps, where one step/iteration is a single mini-batch with approximately 5000 tokens. First the model is saved every 100 steps if fewer than 1000 iterations have passed, and afterwards is saved only if validation accuracy increases. When 5000 training steps pass without improving on validation accuracy, the training terminates. The best saved state of the model is eventually retained.

2.2.5 The POS/XPOS tagger of Dozat et al.

Dozat et al. also proposed a POS tagger with similar architecture that achieves state of the art (SOTA) performance in the majority of the universal dependencies treebanks. Greek is one of the languages that they report SOTA performance in both universal part-of-speech tags (POS tags) and language-specific part-of-speech tags (XPOS tags). XPOS tags contain language-specific part-of-speech tags, normally from a traditional, more fine-grained tagset. Universal POS tagset is shared across all the languages. In contrast, XPOS tagsets are unique for each language and may contain additional information regarding gender, case etc.

The POS/XPOS tagger includes a BiLSTM feature encoder similar to their dependency parser. The BiLSTM-produced encodings are fed to two different MLPs that predict POS and XPOS tags respectively. Given an n-words input sentence s with words

$w_1, w_2, ..., w_n$ each word w_i is associated with a trainable word embedding $e(w_i)$, a pretrained word embedding $p(w_i)$ and a trainable character-level word embedding $c(w_i)$ (Section 2.2.1). A sequence of input vectors $x_{1:n}$ is created where x_i is the element-wise summation of the aforementioned embeddings (figure 2.7):

$$x_i = e(w_i) + p(w_i) + c(w_i)$$

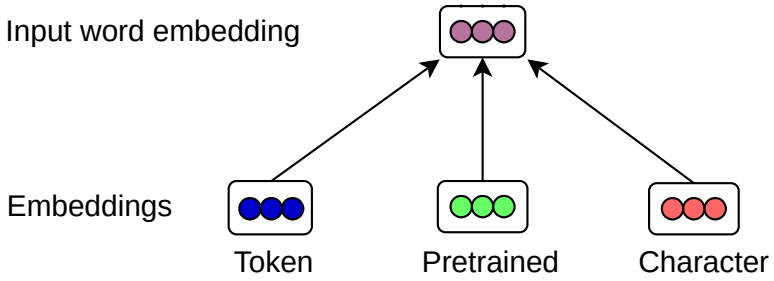

Figure 2.7: BiLSTM input vector in the Dozat et al.'s POS/XPOS tagger (2017).

The sequence of constructed input vectors $x_{1:n}$ is fed to the BiLSTM encoder. The BiLSTM is made of a forward and a backward LSTM of depth two. The produced encodings of the forward and the backward LSTMs are concatenated and every word of the sentence is represented by a deep context-aware representation: $v_{1:n} = \overrightarrow{v}_{1:n} \oplus \overleftarrow{v}_{1:}$. Two different MLPs with one hidden layer, activated using the leaky ReLu function (a=0.1), are used to predict the POS and XPOS tags scores respectively.

$$score_i^{(POS)}[\,] = W_2(LRelu(W_1 v_i + b_1) + b_2)$$

$$score_i^{(XPOS)}[\,] = W_4(LRelu(W_3 v_i + b_3) + b_4)$$

The model predicts the POS and XPOS tags with the highest score.

Both POS and XPOS classifiers are trained jointly using cross-entropy losses that are summed together during optimization. POS and XPOS classifiers share the same BiLSTM and embeddings parameters which enables multitask learning. During training variational dropout of 33% is applied throughout the whole network. On the recurrent connections of the LSTMs the dropout is increased to 50%. The figure 2.8 demonstrates the overall model's architecture.

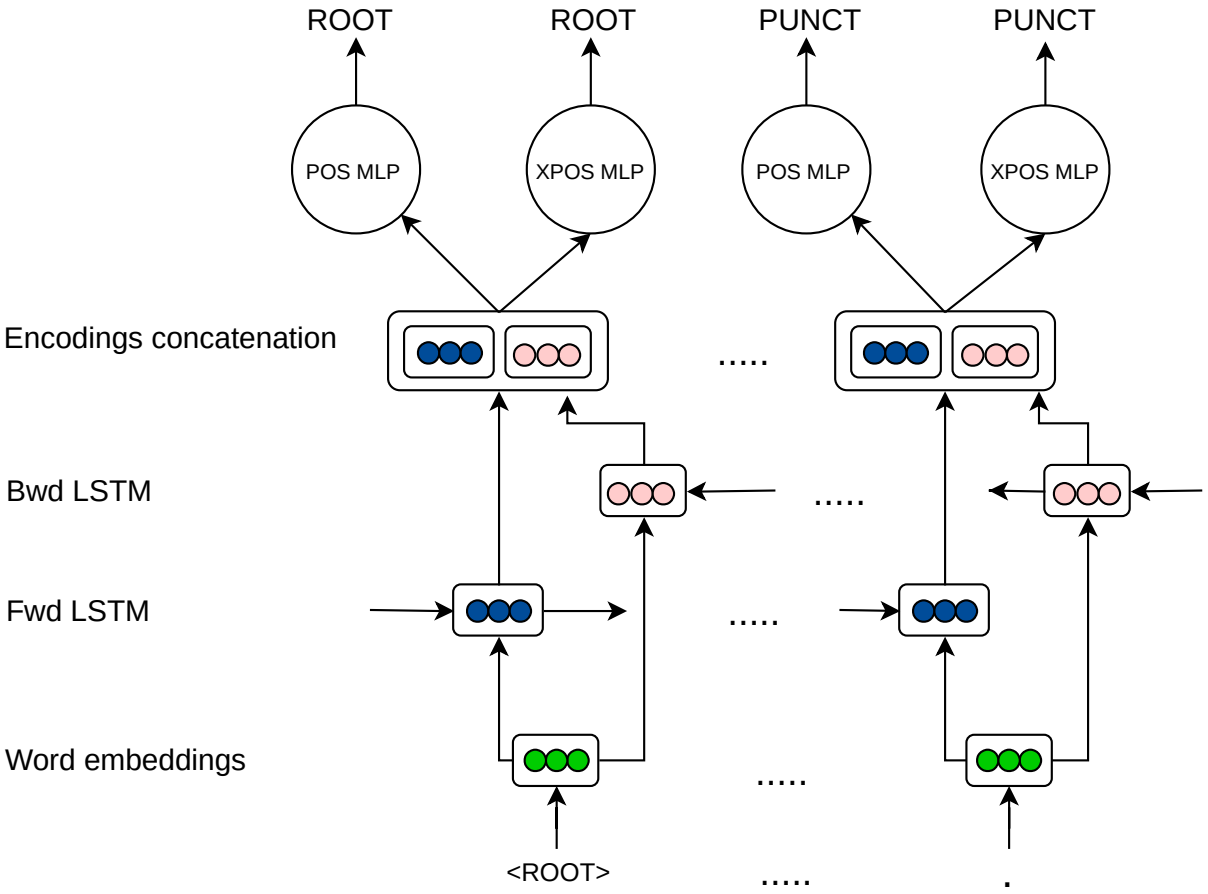

Figure 2.8: The input BiLSTM encoder of Dozat et al.'s POS/XPOS tagger (2017).

3. Our work

Our systems are implemented in Python using the DyNet[2] (Neubig et al., 2017) framework. DyNet is a deep learning framework that builds the computational graph on the fly. In contrast, static graph deep learning frameworks like Tensorflow[3] and Theano[4] first define a computation graph, and then examples are fed into the graph. Building a computational graph on the fly facilitates the implementation of more complicated network architectures and is also very helpful for NLP tasks where sentences have different lengths. The source code of our POS tagger, dependency parser and DyNet re-implementation of Dozat et al.'s parser can be found on the following repositories https://bitbucket.org/makyr90/tagger, https://bitbucket.org/makyr90/simple-parser, https://bitbucket.org/makyr90/biaffine_parser.

At the heart of our systems lies our LSTM cell implementation. We use standard LSTMs with uncoupled input, forget gates and no forget bias, following the equations below:

$$i_t = sigmoid\left(W_{ix} x_t + W_{ih} h_{t-1} + b_i\right) \quad (1)$$

$$f_t = sigmoid\left(W_{fx} x_t + W_{fh} h_{t-1} + b_f\right) \quad (2)$$

$$o_t = sigmoid\left(W_{ox} x_t + W_{oh} h_{t-1} + b_o\right) \quad (3)$$

$$\tilde{c} = \tanh\left(W_{cx} x_t + W_{ch} h_{t-1} + b_c\right) \quad (4)$$

$$c_t = c_{t-1} \odot f_t + \tilde{c} \odot i_t \quad (5)$$

$$h_t = \tanh\left(c_t\right) \odot o_t \quad (6)$$

Equations 1-3 compute the input, forget and output gates respectively. Equation 4 computes the candidate cell state. Finally, equations 5 and 6 compute the new cell state and the hidden state respectively. This LSTM version is very efficient since input, forget, output gates and candidate cell state can be computed in one affine transformation. Then, sigmoid activation is applied to the gates and tanh activation to the candidate cell state.

2 http://dynet.io/
3 https://www.tensorflow.org/
4 http://deeplearning.net/software/theano/

3.2 POS/XPOS tagger

Our POS tagger is actually a re-implementation of Dozat et al.'s POS/XPOS tagger in the DyNet framework. Although we have tried different architectures and hyper-parameter setups we could not surpass the performance of Dozat et al.'s tagger. The fact that it achieves excellent results, with F1-score of approximately 97.7% for both POS and XPOS tags in the Greek test treebank made it extremely difficult for us to beat its performance. Our tagger re-implementation utilizes fasttext pretrained word embeddings (Bojanowski et al., 2016) instead of the word2vec pretrained word embeddings used by Dozat et al.

3.2.1 Input and Model architecture

Similar to Dozat et al.'s tagger our model takes as input trainable, pretrained (fasttext) and character-level word embeddings. Fasttext embeddings are trained similarly to word2vec embeddings using the skipgram model. In contrast to the word2vec model where every word is represented solely by the word itself, fasttext learns representations for character n-grams, and represents words as the sum of their n-gram vectors. The aforementioned technique is an extension of the skipgram model that takes into account sub-word information. For example the word vector "apple" is a sum of the vectors of the n-grams "<ap", "app", "appl", "apple", "apple>", "ppl", "pple", "pple>", "ple", "ple>", "le>", assuming the hyper-parameters for the smallest and largest n-grams are set to 3 and 6, respectively. One advantage of fasttext embeddings is their ability to generate better word embeddings for rare words. Even if words are rare their character n-grams are still shared with other words and hence the produced embeddings can still be good.

Given an n-words input sentence s with words $w_1, w_2, ..., w_n$ each word w_i is associated with a trainable word embedding $e(w_i)$, a pretrained (fasttext) word embedding $f(w_i)$ and a trainable character-level word embedding $c(w_i)$ (Section 2.2.1). A sequence of input vectors $x_{1:n}$ is created where x_i is the element-wise summation of the aforementioned embeddings:

$$x_i = e(w_i) + f(w_i) + c(w_i)$$

The input vectors are fed to a BiLSTM encoder of depth 2 and the produced encodings are used by two different MLPs with one hidden layer in order to predict POS and XPOS tags as described in the Section 2.2.5.

3.2.2 Training details

Our POS tagger is trained for up to 30000 training steps. Each training step is a mini-batch of approximately 5000 tokens. The Adam optimizer with initial learning rate = 0.002 and β1 = β2 = 0.9 is used. Like Dozat et al., the initial learning rate is decayed every 5000 training steps using the following equation:

$$new\ learning\ rate = 0.75 * current\ learning\ rate$$

The model is saved every 100 training steps during the first 1000 training steps. Then, it is saved only if the accuracy on the validation data increases. If validation accuracy is not improved for 5000 consecutive training steps the training process terminates. The best saved state of the model is eventually retained. Variational dropout of 33% is used throughout the whole network including the LSTM's input and recurrent states. For the LSTM's recurrent connections the dropout is increased to 50%.

Trainable word embeddings are defined for words that occur at least twice in the training dataset. Rare words (that occur only once in the entire train dataset) are replaced with a special <UNK> embedding. Also, if a word does not exist in the external embeddings matrix its pretrainned embedding vector is set to zeros. A special <ROOT> embedding is also trained since an artificial root word is added to the start of each sentence. Trainable word embeddings, bias terms and final linear layers are initialized to zeros. Character embeddings are initialized using a Gaussian distribution with mean = 0 and variance = 1. All the other parameters, including the LSTM's and MLP's hidden layers weights, are initialized using orthonormal initialization (Saxe et al., 2014).

Frequent word and character embeddings sizes are set to 100D. Fasttext pretrained embeddings have an original size of 300D and are squeezed to 100D via a linear projection layer (Wx + b). The weights of the fasttext embeddings projection are regularized using L2 regularization. The BiLSTM has recurrent states of 200D while the LSTM of character-level word embeddings has recurrent states of 400D. Finally, the POS and XPOS MLPs have hidden states of 200D.

3.3 Dependency parser

For our dependency parser we investigated various architectures with a focus on the Greek language. We have re-implemented Dozat et al.'s parser in the DyNet framework and experimented with various architectures, including a stack of two shallow Bi-LSTMS instead of one deep BiLSTM, a joint arc-label score function and the addition of distance/direction embeddings as an extra input to the joint arc-label score function. Our target was to develop a parser capable to produce state-of-the art performance and as much simple as possible.

3.3.1 Neural architectures investigation

In our re-implementation of Dozat et al.'s parser we have compared (in the Greek development treebank) the performance between a three layers deep BiLSTM encoder (Dozat et al.) and a stack of two shallow (depth=1) BiLSTMs. Both UAS and LAS F1-scores were better using the stack of two BiLSTMs. Another advantage of the aforementioned modification was that the stacked BiLSTMs encoder requires ~16% less parameters and thus it is faster and less memory intensive during training. The main difference between a stack of n shallow BiLSTMs and a single n-layers deep BiLSTM is the way that forward and backward LSTM states are concatenated. In a single n-layers deep BiLSTM the forward and backward LSTM states are concatenated only in the last n^{th} layer. In contrast, each of the stacked BiLSTMs receives as input the concatenation of the forward and backward LSTM states of the previous BiLSTM.

Our next step was to eliminate the biaffine classifiers. We kept only two specialized representations produced by the BiLSTM encodings, instead of the four in the original Dozat et al. model. One specialized representation for the word as head seeking all of its dependents and another for the word as dependent seeking its head. Both arcs and labels are predicted using the same (two) types of vectors. We have tried both concatenation and element-wise summation of the head-modifier representations before feeding them to the MLP score function. Both options produced similar results in the Greek development treebank but, the element-wise summation was faster and required less resources. Instead of biaffine classifiers, we used a single MLP with two hidden layers in order to produce labeled arc scores (joint arc-label predictions) for the head-modifier pairs. A single cross-entropy loss of joint arcs-labels was optimized during training.

Since both the parsers of Dozat et al. and Kiperwasser et al. do not use anything like traditional linear features and rely on the BiLSTM encoders exclusively as features, we decided to add distance/direction information as an extra feature to our MLP score

function. Distance and direction between the head and the modifier were core features (McDonald et al., 2005) for the linear graph-based parsers. Distance defines how far a modifier is from its head in a sentence. We can define the distance as the number of words that lies between the head and the modifier. Direction declares if a modifier lies before (right arc) or after (left arc) its head in the sentence. Each head-modifier pair in a sentence has a distance/direction score. The sentence in the figure 3.1 is annotated with the distance/direction scores of its head-modifier pairs.

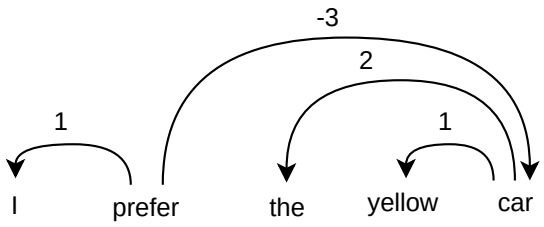

Figure 3.1: Distance/direction scores.

Our parser utilizes distance and direction information regarding the head-modifier pairs using embeddings for buckets (particular ranges) of distance/direction scores. The vast majority (~94%) of the head-modifier distance/direction scores in the Greek universal dependencies train treebank lies in the range of [-10,10]. Thus, we have decided to include distance/direction embedding buckets in the aforementioned range. When a distance/direction score of a head-modifier pair lies in the range of [-10,10] the embedding (of the corresponding bucket) is retrieved from the distance embeddings lookup matrix. If the distance/direction score does not lie in the aforementioned range, the -10 and 10 embeddings (of the corresponding buckets) are selected for the negative and positive distance/direction scores respectively.

3.3.2 Input encoder and head/modifier ReLu layers

Similar to Dozat et al. parser our model takes as input word and POS tag embeddings (figure 3.2). The word embeddings are constructed using the element-wise summation of trainable embedding vectors, pretrained word vectors (fasttext) and trainable character-level word embeddings (Section 2.2.1). The POS tag embeddings are constructed using the element-wise summation of the trainable POS and XPOS embeddngs. A sequence of input vectors $x_{1:n}$ is created where x_i is the concatenation of the word and POS tag embeddings.

$$x_i = [e(w_i) + f(w_i) + c(w_i)] \oplus [e(t_i^{POS}) + e(t_i^{XPOS})]$$

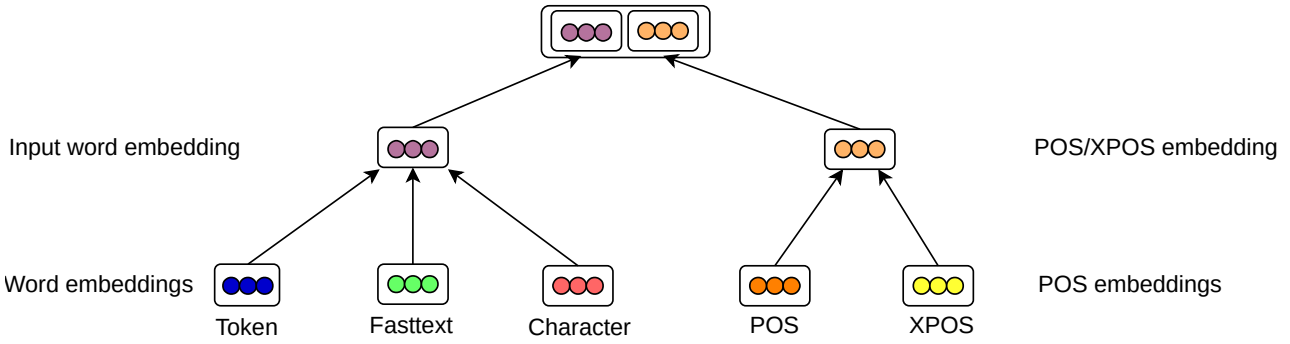

Figure 3.2: BiLSTM input vector of our parser

Our parser utilizes a stack of two BiLSTMs as encoder (figure 3.3). Each BiLSTM is composed by a forward and backward LSTM, both of depth 1. The encodings (concatenated forward and backward states) of the first BiLSTM are fed to the second BiLSTM that produces the final encodings. More concretely, given an n-words input sentence s with words $w_1, w_2, ..., w_n$ a sequence of input vectors $x_1, x_2, ..., x_n$ is derived from the concatenation of the respective words and tags embeddings. The input vectors are fed to the forward and the backward LSTM of the first BiLSTM. The produced encodings of the forward and the backward LSTM are concatenated for every word in the sentence: $v^1_{1:n} = \overrightarrow{v}^1_{1:n} \oplus \overleftarrow{v}^1_{1:n}$. The produced sequence of encodings $v^1_1, v^1_2, ..., v^1_n$ is fed to the forward and the backward LSTM of the second BiLSTM. The produced forward and backward LSTM states of the second BiLSTM are concatenated again for every word in the sentence: $v^2_{1:n} = \overrightarrow{v}^2_{1:n} \oplus \overleftarrow{v}^2_{1:n}$. By contrast, in the original depth-3 BiLSTM of Dozat et al. the forward and the backward LSTM states are concatenated only in the last (3^d) layer.

The BiLSTM-produced encodings are fed through two separate ReLU layers (leaky ReLu with a=0.1), producing two specialized vector representations: one for the word as a head seeking all its dependents and another for the word as a dependent seeking its head. The following formulas are used to compute the ReLu layers:

$$h_i^{(arc)} = LReLu(W_1 v_i^2 + b_1)$$

$$h_i^{(dep)} = LReLu(W_2 v_i^2 + b_2)$$

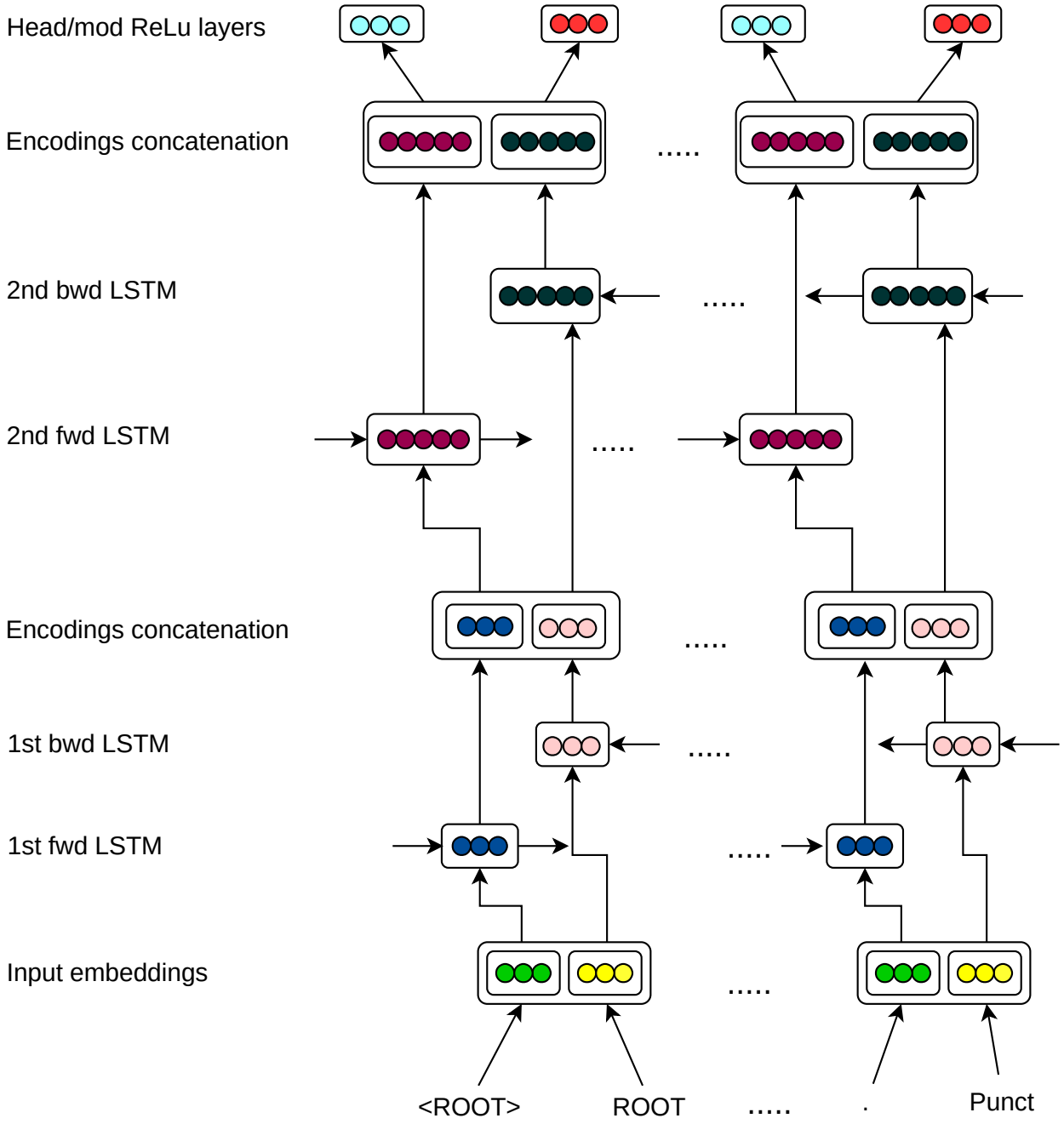

Figure 3.3: The input BiLSTM encoder of our parser.

3.3.3 MLP score function

Given a head-modifier pair (j,i), the MLP score function (figure 3.4) takes as input a vector $V_{j,i}$ that is the concatenation of the element-wise summation of the head and modifier specialized representations and the distance/direction embedding that corresponds to that pair ($d(j,i)$). The formula that is used is:

$$V(j,i) = (h_j^{(arc)} + h_i^{(dep)}) \oplus d(j,i)$$

The MLP has two hidden layers with leaky ReLu as activation function and produces as output the scores of all the candidate labeled arcs between a head-modifier pair. The formula that is used is:

$$scores(j,i)[\,] = LReLu(W_2(LReLu(W_1 V_{(j,i)} + b_1)) + b_2)$$

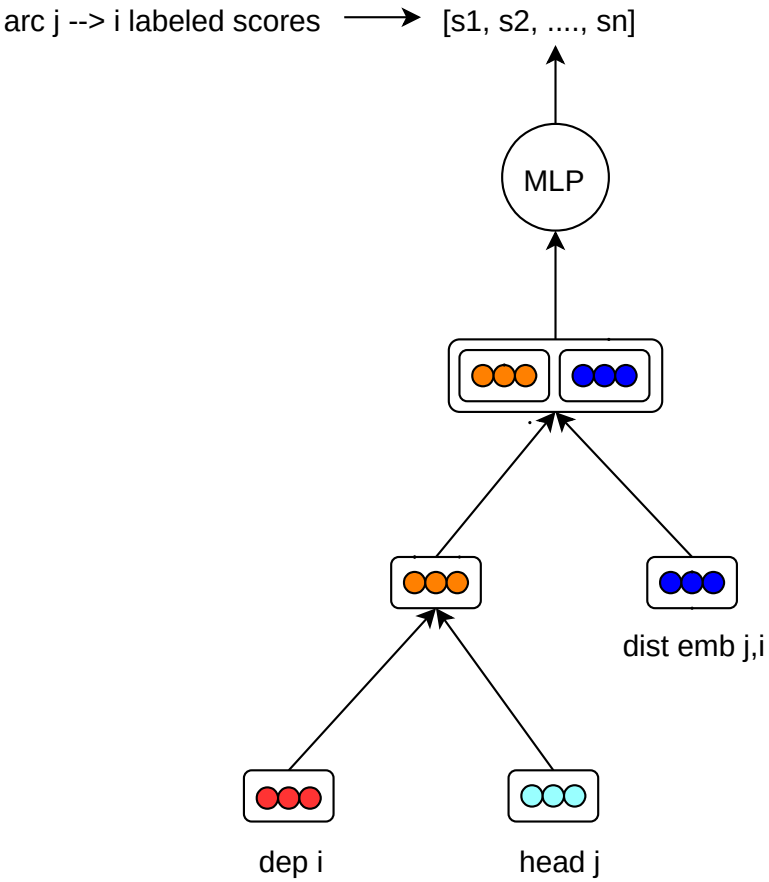

Figure 3.4: Scoring labeled arcs (joint arc-label predictions) in our parser.

The produced labeled arc scores of a modifier with all of its possible heads are concatenated creating a large flat vector of joint heads/labels scores. The size of that

vector is equal to the number of possible heads times the number of dependency relation labels. During training, a single cross-entropy loss over the joint heads/labels scores is optimized.

3.3.4 Parsing using Edmond's decoder

During training we validate our parser's performance on a development dataset. For this purpose, the highest scoring head is predicted for each modifier in a sentence. The above method is very efficient but does not guarantee that a valid dependency structure is produced. When parsing new sentences with our model we want to ensure that correct dependency structures are produced. A valid dependency structure is a single rooted directed maximum spanning tree. Hence, we have implemented a parsing decoder using the Chu-Liu-Edmonds algorithm. Given a sentence of n words, our decoder takes as input a (n+1) X (n+1) scores matrix for all the possible head-modifier pairs in the sentence. The +1 corresponds to the artificial root token that is placed at the start of each sentence. The scores matrix is calculated using our parser's predictions. Each column of the matrix corresponds to the scores of all the possible heads for the modifier that is indexed in the column. The scores matrix forms a dense (fully-connected) directed graph.

The first step of Edmond's decoder is to select the highest-scored head (incoming arc) for each modifier in order to form an initial dependency graph. Then, using Kosaraju's algorithm for strongly connected components (SCC) the decoder search for possible cycles in the graph. If there is no cycles we have already found the maximum spanning tree of the dependency graph. Otherwise, if one or more cycles are found the decoder recursively contract all the cycles until a solution with no cycles is derived. During cycle contraction two or more graph's vertices are contracted into one cycle. Incoming and outcoming arcs are recalculated for the newly contacted vertex and pointers are kept in order to backtrack the original arcs of the graph.

When the decoder finds the maximum spanning tree of the dependency graph it proceeds to the check of the single rooted property. If there is no multiple roots the decoder returns the heads derived from the extracted maximum spanning tree. In case of multiple roots, the root with the highest score is greedily selected. This is done by setting all the other outcoming arcs of the root vertex to $-inf$ in the initial dense directed graph. Then, using the updated dense directed graph graph as input the decoder re-runs. Since only one outcoming arc of the root has score different to $-inf$ the maximum spanning tree that will be extracted on the 2nd run mandatory satisfies the single rooted property. The root of the new MST will be the greedily selected one.

Since our parser produce scores for labeled arcs, we reduce the labeled arcs parsing problem (directed multi-graph) to unlabeled (directed graph) in order to utilize our

decoder implementation. This is done by picking the highest labeled score as arc score for each head-modifier pair.

3.3.5 Training details

Our dependency parser is trained for up to 30000 training steps. Each training step is a mini-batch of approximately 1500 tokens. The Adam optimizer with initial learning rate = 0.002 and β1 = β2 = 0.9 is used. Like Dozat et al., the initial learning rate is decayed every 5000 training steps using the following equation:

$$new\ learning\ rate = 0.75 * current\ learning\ rate$$

The model is saved every 100 training steps during the first 5000 training steps. Then, it is saved only if the LAS (Labeled Attachment Score) accuracy on the validation data increases. If validation LAS accuracy is not improved for 2500 consecutive training steps the training process terminates. The best saved state of the model is eventually retained.

Regular dropout (Srivastava et al., 2014) of 33% is applied throughout the entire network, but variational dropout of 33% and 50% is used for the LSTM's input and recurrent connections respectively. The unidirectional LSTM of character-level embeddings has 33% variational dropout for both input and recurrent connections. Furthermore, (entire) word and tag embeddings are dropped independently with 33% probability. If one is dropped the other is scaled to compensate. If both are dropped then the whole input vector is replaced with zeros.

Trainable word embeddings are defined for words that occur at least twice in the training dataset. Rare words (that occur only once in the entire train dataset) are replaced with a special <UNK> embedding. Also, if a word does not exist in the external embeddings matrix its pretrainned embedding vector is set to zeros. A special <ROOT> embedding is also trained since an artificial root word is added to the start of each sentence. Trainable word embeddings, POS and XPOS tag embeddings, bias terms and final linear layers are initialized to zeros. Character embeddings are initialized using a Gaussian distribution with mean = 0 and variance = 1. All the other parameters, including the LSTM's and MLP's hidden layers weights, are initialized using orthonormal initialization.

Frequent word, character, POS and XPOS tags embeddings sizes are set to 100D. Distance/direction embeddings are set to 32D. Fasttext pretrained embeddings have original size of 300D and are squeezed to 100D via a linear projection layer (Wx + b). The 1st BiLSTM has recurrent states of 200D and the 2nd BiLSTM has recurrent states of 400D. The unidirectional LSTM of character embeddings has recurrent states of 400D. Finally, the ReLU layers of the specialized head/dependent representations and the hidden state of the scoring MLP all have size of 400D.

4. Experiments

In this section we evaluate our systems and compare their performance with the currently state-of-the art systems of Dozat et al. (2017). In both systems we first use the baseline sentence splitter and tokenizer provided by the CoNLL 2017 shared task[5]. Dozat et al. use the same sentence splitter and tokenizer for their systems. Thus, we can made a fair comparison between their systems and ours. The Universal dependencies treebanks (version 2.1) are used for training, development and testing. Dozat et al. also use the same version (2.1) of universal dependencies for the development and evaluation of their systems. Universal Dependencies[6] (UD) is a project that is developing cross-linguistically consistent treebank annotation for many languages, with the goal of facilitating multilingual parser development, cross-lingual learning, and parsing research from a language typology perspective. The annotation scheme is based on an evolution of (universal) Stanford dependencies (de Marneffe et al., 2006, 2008, 2014), Google universal POS tags (Petrov et al., 2012), and the Interset interlingua for morphosyntactic tagsets (Zeman, 2008). The general philosophy is to provide a universal inventory of categories and guidelines to facilitate consistent annotation of similar constructions across languages, while allowing language-specific extensions when necessary. Most of the UD languages have train, development ant test treebanks.

4.1 CoNLL-U format

Treebanks are stored using the CoNLL-U format (figure 4.1). Annotations are encoded in plain text files with three types of lines:

1. Word lines containing the annotation of a word/token in 10 fields separated by single tab characters.
2. Blank lines marking sentence boundaries.
3. Comment lines starting with hash (#).

Sentences consist of one or more word lines, and word lines contain the following fields:

1. **ID:** Word index, integer starting at 1 for each new sentence.
2. **FORM:** Word form or punctuation symbol.
3. **LEMMA:** Lemma or stem of word form.

5 http://universaldependencies.org/conll17/
6 http://universaldependencies.org/

4. **UPOS:** Universal part-of-speech tag.
5. **XPOS:** Language-specific part-of-speech tag, underscore if not available.
6. **FEATS:** List of morphological features from the universal feature inventory or from a defined language-specific extension, underscore if not available.
7. **HEAD:** Head of the current word, which is either a value of ID or zero (0).
8. **DEPREL:** Universal dependency relation to the HEAD (root iff HEAD = 0).
9. **DEPS:** Enhanced dependency graph[7] in the form of a list of head-deprel pairs.
10. **MISC:** Any other annotation.

The following figure is an example of a CoNLL-U formatted text file (source: http://universaldependencies.org)

```
# sent_id = 1
# text = They buy and sell books.
1   They     they    PRON    PRP    Case=Nom|Number=Plur              2   nsubj   2:nsubj|4:nsubj   _
2   buy      buy     VERB    VBP    Number=Plur|Person=3|Tense=Pres   0   root    0:root            _
3   and      and     CONJ    CC     _                                 4   cc      4:cc              _
4   sell     sell    VERB    VBP    Number=Plur|Person=3|Tense=Pres   2   conj    0:root|2:conj     _
5   books    book    NOUN    NNS    Number=Plur                       2   obj     2:obj|4:obj       SpaceAfter=No
6   .        .       PUNCT   .      _                                 2   punct   2:punct           _

# sent_id = 2
# text = I have no clue.
1   I       I       PRON    PRP   Case=Nom|Number=Sing|Person=1     2   nsubj   _   _
2   have    have    VERB    VBP   Number=Sing|Person=1|Tense=Pres   0   root    _   _
3   no      no      DET     DT    PronType=Neg                      4   det     _   _
4   clue    clue    NOUN    NN    Number=Sing                       2   obj     _   SpaceAfter=No
5   .       .       PUNCT   .     _                                 2   punct   _   _
```

Figure 4.1: Example of a CoNLL-U formatted text file.

7 http://universaldependencies.org/u/overview/enhanced-syntax.html

4.2 POS/XPOS tagger results

The following table compares our re-implementation of Dozat et al.'s POS/XPOS tagger with the original system. As we can see our re-implementation is slightly inferior to the original implementation. Such small differences can be attributed to different random initialization of the model's parameters.

	Dozat et al. tagger		**Our re-implementation**	
Language	POS	XPOS	POS	XPOS
Greek	**97.74**	**97.76**	97.66	97.63

Table 1: Comparison between Dozat et al.'s tagger and our re-implementation.

4.3 Dependency parser results

The table below summarizes the incremental improvement on Greek development treebank after applying the modifications described in the Section 3.3.1.

Model	**UAS F1-score**	**LAS F1-score**
Dozat et al. re-implementation	89.75	87.81
Dozat et al. re-implementation (stacked BiLSTMs)	90.13	88.17
Stacked BiLSTMs + joint arc-labels predictions	90.20	88.54
Stacked BiLSTMs + joint arc-labels predictions + distance/direction embeddings	**90.66***	**89.16***

Table 2: Results on the Greek development treebank.

Our final model that includes all the modifications we have described in the Section 3.3.1 has superior performance compared to our Dozat et al. re-implementation. This

superiority is statistically significant since both UAS and LAS p-values are equal to 0.001. The significance tests were performed using a web tool[8] for dependency parsing evaluation developed by Choi et al. (2015), which uses McNemar's test to detect statistically significant differences.

The following table compares the performance of our parser with the Dozat et al. parser. To be fair in comparisons we used the predicted POS/XPOS tags from the Dozat et al. POS tagger as input to our parser. Both parsers make predctions based on non-gold sentences and tokens. The test dataset is first preprocessed using the baseline sentence splitter and tokenizer provided by the CoNLL 2017 shared task.

	Dozat et al. parser		**Our parser**	
Language	UAS	LAS	UAS	LAS
Greek	89.73	87.38	**90.44**	**88.32**
Bulgarian	**92.89**	**89.81**	92.79	89.76
English	**84.74**	**82.23**	84.59	82.16
Dutch	**85.17**	**80.48**	83.63	79.13

Table 3: Comparison between Dozat et al.'s parser and our parser.

As we observe our model is superior compared to Dozat et al.'s in the Greek language, which is probably due to the fact that we focused mostly on Greek (e.g., model selection was performed on the Greek development treebank). Furthermore, we achieve near state-of-the art performance for English and Bulgarian which means that our model performs really well across other languages as well. Thus, we can claim that the complex biaffine transformation of Dozat et al.'s parser may not be necessary and a simpler model that uses a joint arc-labels loss suffices.

8 https://emorynlp.github.io/dependable/evaluate.html

In order to determine whether our parser is superior compared to Dozat et. al we performed significance tests using Choi et al.'s (2015) tool for dependency parsing evaluation. In the following table we report the performance of both parsers using gold POS/XPOS tags, sentences and tokens.

	Dozat et al. parser		**Our parser**		**P-values**	
Language	UAS	LAS	UAS	LAS	UAS	LAS
Greek	90.8	88.87	**91.53***	**89.69***	0.001	0.001
Bulgarian	94.55	91.19	**95.23***	**91.89***	0.001	0.001
English	91.17	89.36	**91.34**	**89.49**	0.5	0.5
Dutch	**89.70***	**86.54***	89.19	85.92	0.05	0.02

Table 4: Significance tests between Dozat et al.'s parser and our parser.

We observe that our parser has significantly superior performance on Greek (p-value = 0.001) and Bulgarian (p-value = 0.001). On English, our parser achieves slightly superior performance but no statistically significant difference was detected (p-value = 0.5). Finally, on Dutch Dozat et al.'s parser has significantly superior performance (p-value = 0.05, 0.02 for UAS and LAS, respectively) compared to our parser.

5. Conclusions and future work

5.1 Conclusions

The use of bi-directional LSTMs as feature encoders is a very effective method to produce state-of-the art results for the tasks of POS tagging and dependency parsing. Actually, bi-directional LSTM networks have replaced the tedious task of feature-engineering that requires a lot of expertise. Through bi-directional LSTMs it is possible for a model to learn complex non-linear future functions that can significantly enhance the model's performance.

POS tag embeddings significantly increase the performance of a dependency parsing model, as also reported by Dozat et al. (2017). Thus, we can claim that POS tags remain a core future for dependency parsing models.

Including character-level word embeddings increases the performance in both POS tagging and dependency parsing (Dozat et al., 2017), especially for morphological rich languages. Also, aggressive dropout helps to reduce overfitting and improves generalization. LSTMs are very powerful models that are prone to overfit and thus dropout, which is a well-established ensembling technique, helps to improve the generalization in new unseen data.

In our work we have seen that even simpler dependency parsing models without complex biaffine classifiers on top of the BiLSTM feature encoder can yield state-of-the art or near state-of-the art performance. Also, including distance/directions embeddings as an extra input to our parser's MLP score function boosts its performance. Back to the "linear world", distance and direction between a head and a modifier were fundamental features for dependency parsing models. Although LSTMs were shown to be capable of learning complex features (Karpathy et al., 2015), it seems that by adding distance and direction information regarding the head-modifier pairs to our parser's MLP score function improves its performance.

5.2 Future work

In this section we describe future work that may improve the performance of our POS tagging and dependency parsing systems. First, we plan to include ELMo embeddings (Embeddings from Language Models) as input to both of our models as described by Peters et al. (2018). ELMo embeddings are produced by deep bi-directional language models (biLMs) that are trained in large corpora. ELMo word representations are computed on top of two-layer biLMs with character convolutions, as a linear function of the internal network states. Peters et al. report that the addition of ELMo representations alone significantly improves the state of the art performance in various NLP tasks. Thus, the addition of deep contextualized word embeddings may improve the accuracy of our systems.

Utilizing the FEATS tags that are included in the universal dependencies treebanks as an extra future for our dependency parser. The FEATS tags include additional information regarding morphological features such as gender, number, case etc. Adding FEATS embeddings as input to our parser's BiLSTM may boost its performance since the additional information regarding morphological features seems useful for the task of dependency parsing. For example, in a correct head-modifier arc of a noun (head) and a determiner (modifier) the gender of both head and modifier should be matched. For this purpose we will also extend and retrain our POS tagging system to be able to predict FEATS tags as well.

Extending our model from the arc-factored (1st order) paradigm to higher order non-projective parsing, may enhance its performance. The fact that our model is relatively simple, with a single cross-entropy loss over the joint heads/labels scores, facilitates its extension to a higher-order parser. Since higher-order non-projective dependency parsing is NP-hard an approximation algorithm will be utilized (Zhang et al. 2012) for parsing.

References

Sandra Kübler, Ryan McDonald, Joakim Nivre, *Dependency Parsing*, Morgan & Claypool Publishers, 2009.

Dan Jurafsky, James H. Martin, *Speech and Language Processing (3rd ed. Draft),* 2017.

Yoav Goldberg, *Neural Network Methods for Natural Language Processing*, Morgan & Claypool Publishers, 2017.

Eliyahu Kiperwasser, Yoav Goldberg, *"*Simple and Accurate Dependency Parsing Using Bidirectional LSTM Feature Representations*". Transactions of the Association for Computational Linguistics*, 4:313-327, 2016.

Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean, "Efficient Estimation of Word Representations in Vector Space". arXiv: 1301.3781, 2013.

Timothy Dozat, Christopher D. Manning, *"*Deep Biaffine Attention for Neural Dependency Parsing*". arXiv: 1611.01734,* 2016.

Timothy Dozat, Peng Qi, Christopher D. Manning, "Stanford's Graph-based Neural Dependency Parser at the CoNLL 2017 Shared Task"*. Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, Vancouver, Canada, pp. 20-30, 2017.

Danqi Chen, Christopher D. Manning, *"*A Fast and Accurate Dependency Parser using Neural Networks*". Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP),* Doha, Qatar, pp. 740-750, 2014.

Yarin Gal, Zoubin Ghahramani, "A Theoretically Grounded Application of Dropout in Recurrent Neural Networks"*. Proceedings of the 30th International Conference on Neural Information Processing Systems,* Barcelona, Spain, pp. 1027-1035, 2016.

Joakim Nivre, Željko Agić, Lars Ahrenberg, et al., "Universal Dependencies 2.1". *LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University,* http://hdl.handle.net/11234/1-2515, 2017.

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

Piotr Bojanowski, Edouard Grave, Armand Joulin, Tomas Mikolov, *"*Enriching Word Vectors with Subword Information*". Transactions of the Association for Computational Linguistics,* 5(1): 135-146, 2017.

Jeffrey Pennington, Richard Socher, Christopher D. Manning, *"*GloVe: Global Vectors for Word Representation*". Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP),* Doha, Qatar, pp. 1532-1543, 2014.

Xuezhe Ma, Zecong Hu, Jingzhou Liu, Nanyun Peng, Graham Neubig, Eduard Hovy, *"*Stack-Pointer Networks for Dependency Parsing". arXiv: 1805.01087, 2018.

Kazuma Hashimoto, Caiming Xiong, Yoshimasa Tsuruoka, Richard Socher, *"*A Joint Many-Task Model: Growing a Neural Network for Multiple NLP Tasks*". Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Copenhagen, Denmark, pp. 1923-1933, 2017.

Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, Luke Zettlemoyer, "Deep contextualized word representations". *Proceedings of NAACL-HLT 2018*, New Orleans, Louisiana, pp. 2227-2237, 2018.

Diederik P. Kingma, Jimmy Ba, "Adam: a Method for Stochastic Optimization". *International Conference on Learning Representations*", San Diego, 2014.

Andrej Karpathy, Justin Johnson, Li Fei-Fei, "Visualizing and Understanding Recurrent Networks", arXiv: 1506.02078, 2015.

Andrew M. Saxe, James L. McClelland, Surya Ganguli, "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks", arXiv: 1312.6120, 2013.

Sepp Hochreiter, Jürgen Schmidhuber, "Long Short-term Memory". *Neural computation*, 9(3): 1735-1780, 1997.

Alex Graves, Santiago Fernández, Jürgen Schmidhuber, "Bidirectional LSTM Networks for Improved Phoneme Classification and Recognition". *Proceedings of the 15th International Conference on Artificial Neural Networks: Formal Models and Their Applications-Volume Part II*, Warsaw, Poland, pp. 799-804, 2005.

John Lafferty, Andrew Mccallum, Fernando Pereira, "Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data". *Proceedings of the Eighteenth International Conference on Machine Learning*, San Francisco, CA, USA, pp. 282-289, 2001.

Jason Eisner, "Three New Probabilistic Models for Dependency Parsing: An Exploration". *Proceedings of the 16th International Conference on Computational Linguistics*, Copenhagen, Denmark, pp: 340-345, 1996.

Ming Sun, Jerome R. Bellegarda, “Improved pos tagging for text-to-speech synthesis”. *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Prague, Czech Republic, pp. 5384-5387, 2011.

Chris Quirk, Simon Corston-Oliver, “The impact of parse quality on syntactically-informed statistical machine translation”. *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Sydney, Australia, pp. 62-69, 2006.

Abdur Chowdhury, Catherine McCabe, “Improving Information Retrieval Systems using Part of Speech Tagging”. *Digital Repository at the University of Maryland (DRUM)*, http://hdl.handle.net/1903/5958, 1998.

Joakim Nivre, 2008, “Algorithms for deterministic incremental dependency parsing”. *Computational Linguistics*, 34(4), pp. 513-553, 2008.

Stuart M. Shieber, “Sentence disambiguation by a shift-reduce parsing technique”. *Proceedings of the 21st Conference on Association for Computational Linguistics (ACL),* Cambridge, Massachusetts, pp. 113-118, 1983.

Richard Hudson, “Word grammar”, *Oxford: Basil Blackwell*, 1984.

Igor A. Mel'čuk, “A formal lexicon in the Meaning-Text Theory: (or how to do lexica with words)”. *Computational Linguistics-Special issue of the lexicon archive*, 13(3-4), pp. 261-275, 1987.

Yoeng-Jin Chu, Tseng-Hong Liu, “On shortest arborescence of a directed graph”. *Scientia Sinica,* 14(10):1396, 1965.

Jack Edmonds, “Optimum branchings”. *Journal of Research of the national Bureau of Standards.* B71(4):233-240, 1967.

Ryan McDonald, Koby Crammer, Fernando Pereira, “Online Large-Margin Training of Dependency Parsers”. *Association for Computational Linguistics (ACL)*, Stroudsburg, PA, USA, pp. 91-98, 2005.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. *Journal of Machine Learning Research*, 15, pp. 1929-1958, 2014.

Kristina Toutanova, Dan Klein, Christopher Manning, Yoram Singer, “Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network”. *Proceedings of HLT-NAACL*, Edmonton, Canada, pp. 252-259, 2003.

Adwait Ratnaparkhi, “A Maximum Entropy Model for Part-Of-Speech Tagging”. *Conference on Empirical Methods in Natural Language Processing (EMNLP),* Philadelphia, PA, USA, pp. 133-142, 1996.

Thorsten Brants, “TnT: A Statistical Part-of-Speech Tagger”. *Proceedings of the Sixth Conference on Applied Natural Language Processing,* Seattle, Washington, pp. 224-231, 2000.

Eric Brill, “A simple rule-based part of speech tagger”. *Proceedings of the third conference on Applied natural language processing*, Trento, Italy, pp. 152-155, 1992.

Mitchell P. Marcus, Mary Ann Marcinkiewicz, Beatrice Santorini, “Building a large annotated corpus of English: the penn treebank”. *Computational Linguistics-Special issue on using large corpora: II*, 19(2), pp. 313-330, 1993.

Terry Koo, Michael Collins, “Efficient third-order dependency parsers”, *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, Uppsala, Sweden, pp. 1-11, 2010.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, Jan Hajič, “Non-Projective Dependency Parsing using Spanning Tree Algorithms”. *Human Language Technologies and Empirical Methods in Natural Language Processing (HLT-EMNLP)*, Vancouver, British Columbia, Canada, pp. 523-530, 2005.

Ryan McDonald, Giorgio Satta, “On the Complexity of Non-Projective Data-Driven Dependency Parsing”. *Proceedings of the 10th International Conference on Parsing Technologies*, Prague, Czech Republic, pp. 121-132, 2007.

Ryan McDonald, Fernando Pereira, “Online Learning of Approximate Dependency Parsing Algorithms”. *11th Conference of the European Chapter of the Association for Computational Linguistics*, Trento, Italy, pp. 81-88, 2006.

Hao Zhang, Ryan McDonald, “Generalized Higher-Order Dependency Parsing with Cube Pruning”. *Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, Jeju Island, Korea, pp. 320-331, 2012.

Marie-Catherine de Marneffe, Timothy Dozat, Natalia Silveira, Katri Haverinen, Filip Ginter, Joakim Nivre, Christopher D. Manning, “Universal Stanford Dependencies: A cross-linguistic typology”. *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC),* Reykjavik, Iceland, 2014.

Marie-Catherine de Marneffe, Bill MacCartney, Christopher D. Manning, “Generating typed dependency parses from phrase structure parses”. *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC)*, Genoa, Italy, 2006.

Marie-Catherine de Marneffe, Christopher D. Manning, “The Stanford typed dependencies representation”. *Coling 2008: Proceedings of the Workshop on Cross-Framework and Cross-Domain Parser Evaluation*, Manchester, United Kingdom, pp. 1-8, 2008.

Slav Petrov, Dipanjan Das, Ryan McDonald, “A universal part-of-speech tagset”. *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC)*, Istanbul, Turkey, 2012.

Daniel Zeman, “Reusable Tagset Conversion Using Tagset Drivers”. *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC)*, Marrakech, Morocco, 2008.

Jinho D. Choi, Joel Tetreault, Amanda Stent, “It Depends: Dependency Parser Comparison Using A Web-based Evaluation Tool”. *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, Beijing, China, pp. 387-396, 2015.