Transition Based Dependency Parsing With Deep Learning

by

Ömer Kırnap

A Dissertation Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Computer Science and Engineering

KOÇ
UNIVERSITY

September 27, 2018

Transition Based Dependency Parsing With Deep Learning

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Ömer Kırnap

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

_____

Assoc. Prof. Dr. Deniz Yuret

_____

Assist. Prof. Dr. Gülşen Cebiroğlu Eryiğit

_____

Assoc. Prof. Dr. Engin Erzin

Date: _____

Dedicated to my dad and mom and for those who never give up...

ABSTRACT

I introduce word and context embeddings derived from a language model representing left/right context of a word instance and demonstrate that context embeddings significantly improve the accuracy of transition based parser. Our multi-layer perceptron (MLP) parser making use of these embeddings was ranked 7th out of 33 participants (ranked 1st among transition based parsers) in CoNLL 2017 UD Shared Task. However MLP parser relies on additional hand-crafted features which are used to summarize sequential information. I exploit recurrent neural networks to remove these features by implementing tree-stack LSTM, and develop new set of continuous embeddings called morphological feature embeddings. According to official comparison results in CoNLL 2018 UD Shared Task, our tree-stack LSTM outperforms MLP in transition based dependency parsing.

ÖZETÇE

Bu tezde bir dil modelinden türetilen bir kelimenin sol ve sağ bağlamını temsil eden vektöleri tanıtıyorum. Bu vektörlerin faydasını çok katmanlı algılayıcı (MLP) bağlılık ayrıştırıcı model ile test ettim. 2017 Evrensel Bağlılık Analizi yarışmasında 33 katılımcı arasından 7. olan bu model, sağ/sol bağlam vektörlerinin bağlılık ayrıştırmasında kullanıldığında ayrıştırmanın doğruluğunu önemli ölçüde arttırdığını gösteriyor. Bu tezde daha detaylı anlatacağm çok katmanlı ayrıştırıcı model bir takım el-yapımı özelliklerden de faylandığında en iyi performansı vermektedir. Derin öğrenmedeki son gelişmelerden faydalanarak el-yapımı özellik belirlemek yerine bunu otomatikleştiren bir model geliştirdim. Bu model yine bu tezde tanıttığım biçimsel özellik vektörleri ile birleştiğinde çok katmanlı ayrıştırıcı modelden daha yüksek başarım elde etti. Bu model ile 2018 Evrensel Bağlılık Analizi yarışmasına katıldım. Modelim 30 farklı sistem içerisinden 16. oldu.

ACKNOWLEDGMENTS

I would like to express my utmost gratitude to my advisor, Deniz Yuret, for his genuine help, excellent advising, and critical insights. Working with Deniz Yuret is one of the most inspiring and unique experiences in my life. I am also grateful to my thesis committee Engin Erzin and Gülşen Cebiroğlu Eryiğit for their valuable comments and feedbacks.

I consider myself very lucky to become a member of Artificial Intelligence Laboratory. I would like to especially thank Erenay Dayanık, İlker Kesen, and Ozan Arkan Can who always help and provide their elegant guidance through my journey. I think studying with them helps me to cover fascinating and interesting areas of Artificial Intelligence (AI).

I owe a a debt of gratitude to all my friends making Koç irreplaceable: Kemal Emrecan Şahin, Fazıl Emre Uslu, Recep Gül, Can Emre Ayar, Houman Bahmani Jalali, Itır Bakış Doğru, Çağan Ürküp, Göksu İlbaş, Ozan Can Altıok. I am thankful to Umut Bozkurt for his support to start my AI journey. I also would like to thank TÜBİTAK for providing financial support to my research.

I would like to beholden to my brothers Alper Köse, Atabey Oğulcan Özpirinç, Kaan Telciler, Feraz Kian, Berk Eker, Seçkin Yaşar, Cem Tunç, Egem Eraslan for their invaluable support and patience through my both social and academical life.

I thank Alper Köse, and Atabey Oğulcan Özpirinç also for proofreading the thesis.

My biggest “thank you” goes to my dad Süleyman, my mom Hülya, and my brother Taha. I would not be able to achieve any of my goals without their infinite support, and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

Chapter 1

INTRODUCTION

In this thesis, I have introduced context embeddings and tree-stack LSTM. Context embeddings model the natural language with deep neural networks and map words into continuous dense vectors. Tree-stack LSTM reduces hand-crafted feature engineering in neural transition based dependency parsing. I tested the context embeddings in Conll17 UD Shared Task [Zeman et al., 2017], and tree-stack LSTM in Conll18 UD Shared Task [Zeman et al., 2018].

Numeric vectors, which are trained to model natural language, helps to improve performance in crucial downstream NLP tasks, i.e. dependency parsing, machine translation, question answering. Language Modeling (LM) enables us to find these functional vectors. There are different solutions for LM in the literature. Statistical Language Modeling with maximum entropy [Rosenfeld, 1996] replaced with neural language modeling [Mikolov et al., 2010] over years. In addition, context embeddings were previously shown to improve tasks such as part of speech [Yatbaz et al., 2012] and word sense induction [Baskaya et al., 2013]. In light of these facts, I implemented character level bi-directional LSTM (BiLSTM) to model LM. BiLSTM reads a sentence from both left-to-right and right-to-left and gives a concatenated vector at each reading step called a context vector. I compared my language model's vectors with others [Bojanowski et al., 2017] and showed that it is able to produce useful numeric vectors once trained with a sufficient corpus.

Dependency parsing is a task of automatically finding the tree structure of a sentence. The main purpose is to analyze the syntactic structure of a sentence inspired by a dependency grammar. Dependency structure is based on the idea of having binary, asymmetrical relations in between words called dependency relations (or dependencies for short). A

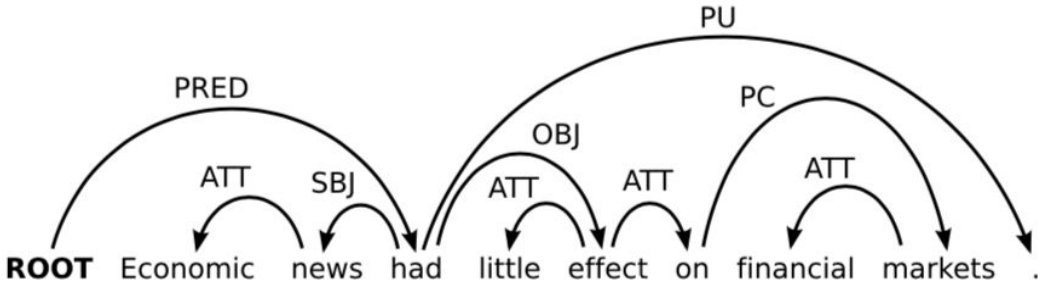

Figure 1.1: Dependency annotations for a sentence " Economic news had little effect on financial markets."

dependency relation connects two words; one of which is subordinate to the other. Specifically, there is a link in between a head and a dependent. Over years, it has shown to its help in downstream NLP tasks, such as machine translation and information retrieval. Figure 1.1 illustrates the dependency structure for the given sentence " Economic news had little effect on financial markets." in which arrows represent the dependency by pointing from head to dependent. For instance, the noun *markets* is a dependent of the preposition *on* with the dependency type by prepositional complement (PC). By contrast, *on* is a dependent of *effect* with a dependency type by attribute (ATT). Dependency grammar is able to deal with languages morphologically rich and relatively free word order. For example, Turkish sentences' word-order are generally more variable than English equivalents. One may change the verb's location to alter the meaning of a sentence. Dependency links in between head and dependent makes flexible word-order information for parsing. Head-dependent relation also provides an information for semantic relations between different phrases.

Data driven dependency parsing is divided into two main branches: graph or transition. Graph based dependency parsing benefits from maximum spanning tree [McDonald et al., 2005] in graph theory [Eisner, 1996]. Data-driven transition based dependency parsing capitalizes on greedy stack based algorithms introduced by [Kudo and Matsumoto, 2002] and [Yamada and Matsumoto, 2003]. Transition based dependency parsing has an advantage of speed over graph based systems. I explained transition based dependency parsing in chapter 2.

In this thesis, I have elaborated my research on transition based framework. My main motivation was two fold. Firstly, building useful context embeddings and showing their advantages in transition based dependency parsing. Next, eliminating hand-crafted features by utilizing recent advancements in deep neural networks.

The Universal Dependencies (UD) [Nivre et al., 2016] supply cross-linguistically applicable and linguistically motivated dependency relations. Thanks to this study, I was able to experiment with my methods for 83 different datasets and 57 different languages annotated within UD project.

The structure of the thesis is organized as follows:

Chapter 2 explains the foundations of Transition Based Dependency Parsing and summarizes the related work done in Neural Transition Based Dependency Parsing.

Chapter 3 details my context embedding model.

Chapter 4 describes my MLP model used in Conll17 Shared Task.

Chapter 5 describes my tree-stack model used in Conll18 Shared Task.

Chapter 6 analyzes the results obtained in shared tasks.

Chapter 7 concludes my work.

Chapter 8 presents possible future research directions.

Chapter 2

RELATED WORK

In this chapter, I summarized the transition based dependency parsing, and, clarify the recent enhancements in neural transition based dependency parsing.

2.1 Transition Based Dependency Parsing

Transition based dependency parsing (TBDP) is based on an abstract machine named transition system [Nivre, 2003] to build a dependency tree. A transition system consists of states and transitions. States (configurations) are defined as triplets $c = (\sigma, \beta, A)$. Namely, a buffer β, a stack σ, and set of dependency arcs A. Although there are different transition systems proposed in the literature, I used ArcHybrid [Kuhlmann et al., 2011] to extract dependency tree. There are three types of transitions in ArcHybrid:

- shift$(\sigma, b|\beta, A) = (\sigma|b, \beta, A)$
- left$_d(\sigma|s, b|\beta, A) = (\sigma, b|\beta, A \cup \{(b, d, s)\})$
- right$_d(\sigma|s|t, \beta, A) = (\sigma|s, \beta, A \cup \{(s, d, t)\})$

where $|$ denotes concatenation and (b, d, s) is a dependency arc between b (head) and s (modifier) with label d. The parser terminates when there is a single word in stack σ while buffer β is empty. Stack σ and buffer β are simple lists, and actions are used to transfer in between states.

Modeling a transition based dependency parser with data requires a mechanism to extract best action from the current parser state. Recent enhancements in deep learning provide different mechanisms for neural TBDP. The components of neural TBDP are divided into two: Feature extraction (state representation) and the decision module.

2.2 Feature Extraction

Recently, there is a shift from hand-engineered sparse features, e.g. [Zhang and Nivre, 2011] to dense continuous feature vectors, e.g. [Chen and Manning, 2014]. Shallow linear models cannot represent feature conjunctions which may be useful to choose a transition in TBDP. To that extent, [Zhang and Nivre, 2011] designed 72 hand-designed conjunctive combinations of 39 primitive features. Deep models, on the other hand, can represent and learn automatic feature combinations. Therefore, a designer should only provide the primitive features. Two questions still remain critical in feature selection: How to represent these (typically discrete) features with continuous vectors, and what part of the parser (which words from the stack σ and the buffer β) should represent the state (c)?

2.2.1 Embedding features and words

In neural TBDP, words, part of speech (POS) tags and other discrete features are represented with numeric vectors. These vectors can be initialized and optimized in number of ways. There are two alternatives for dense vectors to be used, either randomly initialize or transfer from a model for a related task such as language modeling. After that, these vectors can be fixed or fine-tuned during the training of neural TBDP.

A neural language model (NLM) assigns probabilities to word sequences [Bengio et al., 2003, Mikolov et al., 2013b]. Relevant words, e.g. cat and dog, have closer word vectors in transformed space [Kim et al., 2015]. Using these meaningful vectors in down-stream NLP tasks is an advantage [Mikolov et al., 2010]. Since I tested my word vectors in dependency parsing, I investigated dependency parsing works employing pre-trained word vectors from an NLM in the following paragraph.

[Chen and Manning, 2014] initialize with pre-trained word vectors from [Collobert et al., 2011] in English and [Mikolov et al., 2013b] in Chinese. They use randomly initialized POS tag vectors. Similarly, [Dyer et al., 2015] get pre-trained word embeddings from [Bansal et al., 2014] and use randomly initialized POS tag vectors. Both studies fine-tune the vectors during neural parser training. Moreover, [Kiperwasser and Goldberg, 2016a] start with random POS embeddings and fine-tuned word embedding during parser training. They also report that

random initialization in word vectors gives inferior performance.
[Alberti et al., 2017] use a character-level LSTM [Hochreiter and Schmidhuber, 1997] to read each word where the final hidden state creates a word representation.They use that vector as an input to the word-level LSTM whose hidden states constitute what they called *lookahead representation* of each word. Finally, the *lookahead* is used by a tagger LSTM trained to predict POS tags. Concatenation of the *lookahead* and the tagger is used as a word representation in feature vector.

2.2.2 State Representation

Neural TBDP uses various continuous embeddings to represent the parser's state. In [Chen and Manning, 2014 they used first and second words' POS and word embeddings from stack σ, and the leftmost and the rightmost children's word vectors were from the top two words of the stack (σ) and word and POS embeddings of the first word from the buffer (β). They proposed *word-pair* and *three-word* features. These features were the concatenation combination of stack's first and second words. For instance, concatenation of stack's second word's word vector and first word's POS vector and leftmost child's vectors was denoted as one of the three-word features. Please see Table 1 from [Chen and Manning, 2014] for more details. [Kiperwasser and Goldberg, 2016a] used concatenation of all words and POS embeddings as a feature vector. [Andor et al., 2016a] introduced previously predicted tags (other's use gold tags) as a feature into their system. [Alberti et al., 2017] used concatenation of *lookahead* (explained in the previous section) and POS tag vectors as a feature vector. They also use the last two transitions executed by the parser (shift and reduce operations) as binary encoded features. [Dozat et al., 2017]'s model took the concatenation of word vectors from [Mikolov et al., 2013a] and randomly initialized POS embeddings.

Most of the works done in neural dependency parsing attempt to find the best set of features from continuous embeddings and this requires a lot of human effort. [Dyer et al., 2015] proposed a model called stack-LSTM addressing this problem. After generating the feature vector, next step is to find the best neural decision module.

2.3 Decision Module

Decision module is defined as a neural network module which takes state (c) representation vector and returns labeled actions. I applied ArcHybrid [Kuhlmann et al., 2011] system which has 73 possible decisions for 37 different dependency labels. These are defined for labeled-right, labeled-left and shift actions. In the remaining part of current section, I am going the explain fundamental architectures used in neural dependency parsers.

2.3.1 Multi-Layer Perceptron (MLP)

Here are the defining equations for MLP used in related works:

Let **X** be the input matrix containing all the extracted features and W_1 be the matrix of the size hidden by input and W_2 be the matrix of the size output by hidden

$$h = relu(W_1 * X) \tag{2.1}$$

$$o = relu(W_2 * h) \tag{2.2}$$

$$L = -\sum_{n=1}^{c} o_{gold} log(o_{pred}) \tag{2.3}$$

Therefore, the loss is defined as a cross entropy loss over the possible labels.

[Chen and Manning, 2014] uses MLP with different activation functions other than relu.

2.3.2 Recurrent Neural Networks

Recurrent Neural Networks (RNN) introduced during the 1990s [Medsker and Jain, 2001]. The main purpose of the design was to learn time (varying) dependent sequences. Although the architecture and details change, foundation is always the same. Basically, RNNs have an internal hidden states which gets updated based on the current input and the current hidden. Below the basic RNN equations are given:

$$h_t = f(W_{hh} * h_{t-1} + W_{hx} * x_t + b) \; where f \in (tanh, relu, sigm) \tag{2.4}$$

$$o_t = f(W_{ho} * h_t) \tag{2.5}$$

All the related works that I study for this thesis, uses a specific modification of RNNS called Long short term memory (LSTM).

2.3.3 Long Short Term Memory Networks

RNNs suffer from from exploding and vanishing gradients. [Hochreiter and Schmidhuber, 1997] proposed long short term memory (LSTM) architecture to solve exploding and vanishing gradient problems. They basically introduced gated connections which decide what to store and what to delete from LSTM's hidden state. Below the defining equations are given:

$$f_t = sigmoid(W_f * [h_{t-1}, x_t] + b_f) \tag{2.6}$$

$$i_t = sigmoid(W_i * [h_{t-1}, x_t] + b_i) \tag{2.7}$$

$$\tilde{C_t} = tanh(W_c * [h_{t-1}, x_t] + b_c) \tag{2.8}$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C_t} \tag{2.9}$$

$$o_t = sigmoid(W_o * [h_{t-1}, x_t] + b_o) \tag{2.10}$$

$$h_t = o_t * tanh(C_t) \tag{2.11}$$

It a design decision to either use final h_t or h_t s at each time step in Neural TBDP.

2.3.4 Bi-directional LSTM

Bi-directional LSTM (BiLSTM) is designed to a model a sequence by not only considering forward direction but also backward direction [Graves and Schmidhuber, 2005]. In other words, BiLSTM reads a sentence both from left-to-right and right-to-left so that it holds information from both the previous and upcoming part of a sentence for each word position. [Kiperwasser and Goldberg, 2016b] benefited from BiLSTM to implement neural dependency parser. [Dozat et al., 2017] modified this BiLSTM by putting MLPs on top of it and reported performance improvement. [Shi et al., 2017, Che et al., 2017] employed BiLSTMs to provide character based word vectors. They concatenated the final hidden layers of forward and backward LSTMs final hidden layers to obtain word vector. These vectors became input for their parser model. They trained the whole system end-to-end.

2.3.5 Stack-Long Short Term Memory Networks

[Dyer et al., 2015] employs LSTMs, for each part of the state in TBDP and names this model Stack-LSTM. One LSTM reads the words in buffer β, the other LSTM reads the words in stack σ. The third LSTM updates its hidden based on previous transitions. The concatenation of these hidden layers become a representation of state (see Figure 2.3.5). However, they use constant embeddings during parsing. The input to buffer β and stack σ LSTM is the concatenation of pre-trained word embeddings and randomly initialized POS tag embeddings. They fine-tune word embeddings and learn POS embeddings during training.

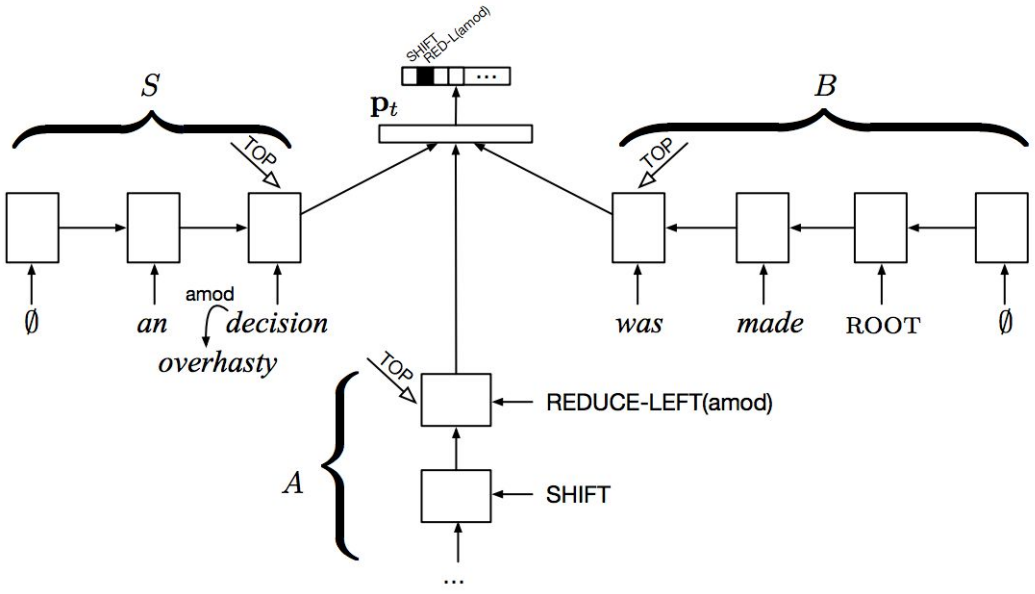

Figure 2.1: Stack-LSTM taken from [Dyer et al., 2015]

Chapter 3

WORD AND CONTEXT EMBEDDINGS

In this chapter, I described the model that I used to extract word and context embeddings.

The main purpose behind word embeddings is to represent meanings of words with continuous vectors such that the related meanings are closer in a space. The better the meaning represented, the more performance is obtained in a down-stream task, dependency parsing in my case. I designed a model to extract two types of language based embeddings: *word* and *context* vectors. To this end, the model consists of two parts to acquire these embeddings: A character based unidirectional LSTM to produce *word* embeddings, and a word based bi-directional LSTM to predict words and produce *context* embeddings.

3.1 Word Vectors

I used a single layer unidirectional LSTM to read each word of a sentence character by character. Each word is padded with a specific start-of-word and end-of-word character. I fed the LSTM from left to right and the final hidden layer becomes *word embedding*. This step is repeated until all the words of a sentence is mapped to *word embeddings*.

3.2 Context Vectors

These *word embeddings*, obtained from character LSTM in Figure 3.1, become inputs to bi-directional LSTM being trained to predict each word based on the left and right contexts. As a result, a *context embedding* for a word is obtained by concatenating the hidden vectors of the forward and backward LSTMs used in predicting that word. The reasons why I trained to predict words other than directly in dependency parsing were to exploit huge

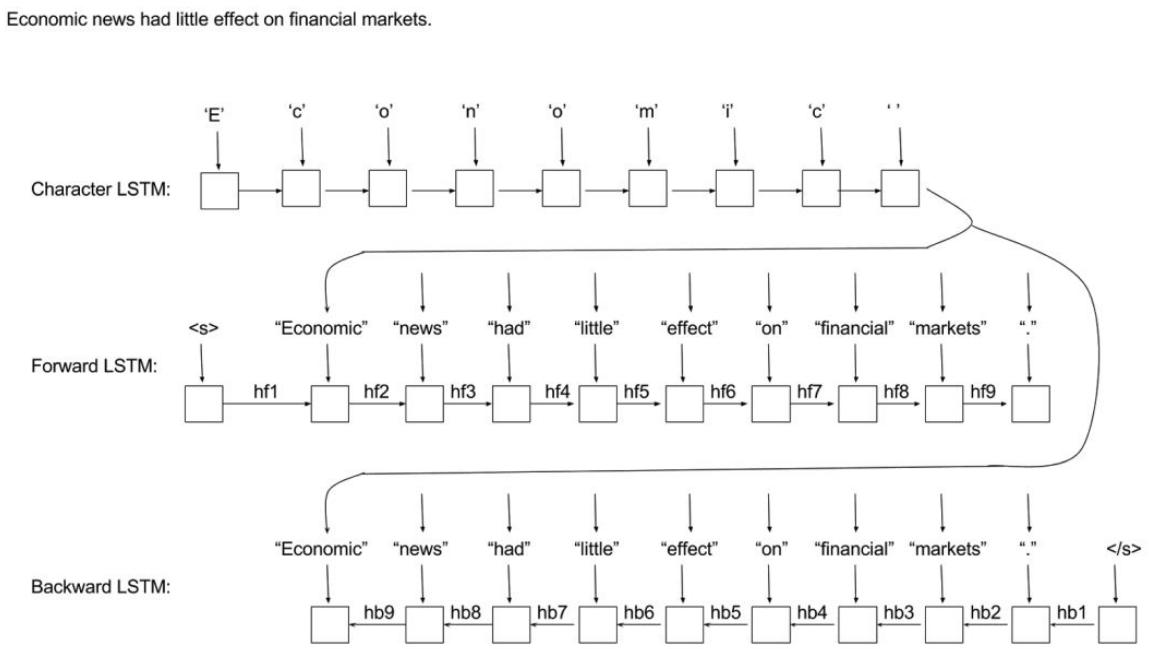

Figure 3.1: Processing of the sentence "Economic news had little effect on financial markets" by the bi-directional LSTM language model. Word embeddings are generated by the character LSTM. Each word is predicted (e.g. "news) by feeding the adjacent hidden states (e.g. "hf2" and "hb8") to a softmax layer.

corpus sizes, and to apply transfer learning. It took approximately 24 hours to convergence on Wikipedia dataset [Zeman et al., 2017] with Tesla K80 GPU for a language model. If I would train it with a parser, I would miss the pre-training advantage and suffer from training times. Moreover, there were no language datasets for some languages in UD Shared Tasks. Thanks to exclusive training I was able to use word and context vectors for these languages by transferring from the same language family. (see Chapter 6 for performance)

Figure 3.1 depicts the language model processing of an example sentence. The unidirectional character LSTM produces the word embeddings (shown for the word "Economic" in the Figure) which are fed as input to the bi-directional word LSTM. The bi-directional LSTM predicts a given word using the adjacent forward and backward hidden states at that position (e.g. the word "news" is predicted using "hf2" and "hb8").

3.3 Training

I trained bi-directional models on Wikipedia dataset provided by Conll17 UD Shared task organizers [Ginter et al., 2017]. The character and word LSTMs were trained end-to-end using backpropagation through time [Werbos, 1990] and Adam [Kingma and Ba, 2014] with default parameters and gradients clipped at 5.0. Sentences which are longer than 28 words were omitted during LM training. In addition, if a word is longer than 65 characters, only the first 65 characters were used and the rest was ignored. The output vocabulary was restricted to the most frequent 20K words of each language.The training was stopped if there was no significant improvement in out-of-sample perplexity during the last 1M words. The perplexity for each language can be found in Chapter 6 Tables 6.2 and 6.3.

I compared these embeddings' performance with current state-of-the-art embeddings in dependency parsing. You may find the results in Chapter 6.

The code is publicly available under [https://github.com/kirnap/ku-dependency-parser/lm].

Chapter 4

MULTI-LAYER PERCEPTRON (MLP) PARSER

In this chapter, I outlined the model which is used to participate Conll17 UD shared task [Zeman et al., 2017] for dependency parsing. Thanks to shared task, I tested the contributions of word and context embeddings. The main objective of an MLP model was to find best dependency parser as well as showing the contribution of word and context embeddings.

MLP parser is based on the idea of having feature intensive neural network parser. Therefore, choosing the proper set of features still remains critical as of [Chen and Manning, 2014].

4.1 Features

In order to find the correct set of features, I and my teammates, experimented with different set of features. We tried to find optimal set of features without loosing too much speed performance.

Abbrev	**Feature**
c	context embedding
v	word embedding
p	universal POS tag
d	distance to the next word
a	number of left children
b	number of right children
A	set of left dependency labels
B	set of right dependency labels
L	dependency label of current word

Table 4.1: Possible features for each word

Table 4.1 lists potential features the model were able to extract for each word. *Context* and *word* embeddings were coming from the language model. The 17 universal POS tags were mapped into 128 dimensional embedding vectors and the 37 universal dependency labels were mapped in 32 dimensional embedding vectors. These were initialized randomly and trained with a parser. To represent sets of dependency labels we simply added the embeddings of each element in the set. Each distinct left/right child count and distance was represented using a randomly initialized 16 dimensional embedding vector trained with parser. Counts and distances larger than 10 were truncated to 10.

This leaved which words to use and which of their features to extract. The transition system informed feature selection: ArcHybrid transitions (see 2.1 for details) directly affect the top word in the buffer and the top two words in the stack.

Figure 4.1 lists the features that were actually extracted by the model to represent a state. $s0, s1, \ldots$ were stack words, $n0, n1, \ldots$ were buffer words, $s1r$ and $s0r$ were the rightmost children of the top two stack words, $n0l$ was the leftmost child of the top buffer word. The letters above each word were the features extracted for that word (using the notation in Table 4.1). Nonexistent features (e.g. the dependency label of $n0l$ when $n0$ did

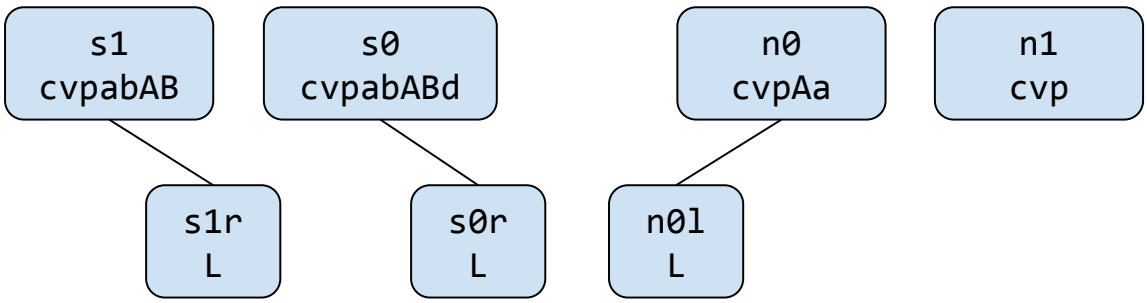

Figure 4.1: Features used by the model. Please see the text and Table 4.1 for an explanation of the notations.

not have any left children) were represented with vectors of zeros.

4.2 Decision Module

We implemented MLP with a single hidden layer of 2048 units to pick parser actions. Embeddings of each feature were concatenated to provide the input to the decision module, which resulted in a 4664 dimensional input vector. Note that word and context embeddings came from the language model and were fixed, whereas the other embeddings were randomly initialized and trained with the MLP.

The output of the MLP was a 73 dimensional softmax layer which represents the shift, 36 left and 36 right (labeled) actions of the parser. Artificial "root" label had no head, so a parser was terminated when the buffer(β) was empty and stack (σ) remained only with "root".

To train the MLP, we used Adam with a dropout rate of 0.5. We trained 5 to 30 epochs, quitting with the best model when the dev score did not improve for 5 epochs.

4.3 Training

Training procedure was different for languages having training and development data, than languages having no development data. We had a different training strategy for languages having too few training data (a.k.a surprise languages).

Language	Parent Language	LAS
North Sami	Estonian	60.48
Buryat	Turkish	47.68
Kurmanji	Bulgarian	46.87
Upper Sorbian	Croatian	65.98

Table 4.2: Parent models used for parsing surprise languages and LAS obtained after pre-training and finetuning.

4.3.1 Languages with training and development data

For most languages, a substantial amount of training and development data with gold parses were supplied. In this case, the parser was trained as described in previous section. The development data was used to decide when to stop training.

4.3.2 Languages without development data

For languages with no development data, we used 5 fold cross validation on the training data to determine the number of epochs for training. The MLP model was trained on each fold for up to 30 epochs during 5 fold cross validation. If the LAS on test split did not improve for 5 epochs, training was stopped and number of epochs to reach the best score was recorded. In final step, the MLP model was trained using whole training data for a number of epochs determined by the average of 5 splits.

4.3.3 Surprise Languages

Surprise languages did not come with a raw data to train a language model, so we decided to implement unlexicalized parsers for them. An unlexicalized parser in our case was simply one that does not use the *"c"* and *"v"* features in Figure 4.1, i.e. no word and context embeddings. The surprise languages also did not have enough training data to train a parser. We hypothesized that an unlexicalized parser trained on a related language might perform better than the one trained on the small amount of sample data. Therefore, we trained

this version for most of the languages provided in the task and tried as “parent” languages for each surprise language. An unlexicalized model trained on the parent language was fine-tuned for the surprise language with its small amount of sample data. Table 4.3.3 lists the parent language used for each surprise language and the LAS achieved on sample data provided using 5-fold cross validation. We submitted the best performing language as a parent.

I explained the results under the Section 6.1. The code is publicly available under [https://github.com/kirnap/ku-dependency-parser/parser].

Chapter 5

TREE-STACK LSTM PARSER

In this chapter, I explained the details of tree-stack LSTM. I used this model to reduce hand-crafted feature engineering without loosing any performance, even improving it if possible. We, as a KParse team, participated in Conll18 shared task with this model.

MLP parser (see Chapter 4 for details) opened a question on how to optimize feature engineering for the state representation of parser. However, thanks to sequential memory of RNNs, tree-stack LSTM could summarize the information from previous transitions as well as word sequences in buffer (β) and stack (σ). Therefore, I did not need to extract any features as in Figure 4.1. I only took advantage of LM (word and context), and morphological feature embeddings.

Tree-stack LSTM differed from [Dyer et al., 2015] by updating the buffer's word embedding and recalculation of the final hidden layer. I used 73 different dependency relation embedding explained in 5.1.4. In contrast, [Dyer et al., 2015] employed the same embeddings both for *"actions"* and *"dependency relations"*.

In the remaining parts of this chapter, I first explained the features used to represent words, then the 4 main components of tree-stack LSTM, namely, buffer's β-LSTM, stack's σ-LSTM, action-LSTM and tree's t-RNN.

5.1 Features

I practiced with only limited number of features as a word feature for tree-stack LSTM, namely, POS, language, and morphological feature embeddings. Language embeddings are the same with Section 4, which are not fine-tuned during training. However, POS and morphological feature embeddings are randomly initialized and learned with parser.

Abbrev	**Feature**
c	context embedding
v	word embedding
p	universal POS tag
f	morphological features

Table 5.1: Possible features extracted for each word

Table 5.1 depicts the possible features that are useful for the model.

5.1.1 Word and context embeddings

I employed the same embeddings explained in Chapter 3. Moreover, I found that reducing the total LM vector's dimension from 950 to 512 did not reduce the parser's dimension but made training faster, whereas there is a trainable parameter W fulfilling this objective. I realized that lower dimensions than 512 resulted in a performance loss. The main reason behind that may be the information loss in lower dimensions.

5.1.2 Part-of-Speech embeddings

I initialized 17 distinct 128 dimensional vectors for 17 distinct Part-of-Speech (POS) tags. These vectors were learned during training.

5.1.3 Morphological feature embeddings

I initiated morphological feature embeddings for some languages as an additional feature, and changed initial word representation by concatenating this vector to the others. I verified that morphological features reduced parsing performance for languages having less than 50k tokens in training set. You may find the details of results in Chapter 6.

Suppose you are given a word *"it"* with the following morphological features:

- *Case=Nom*

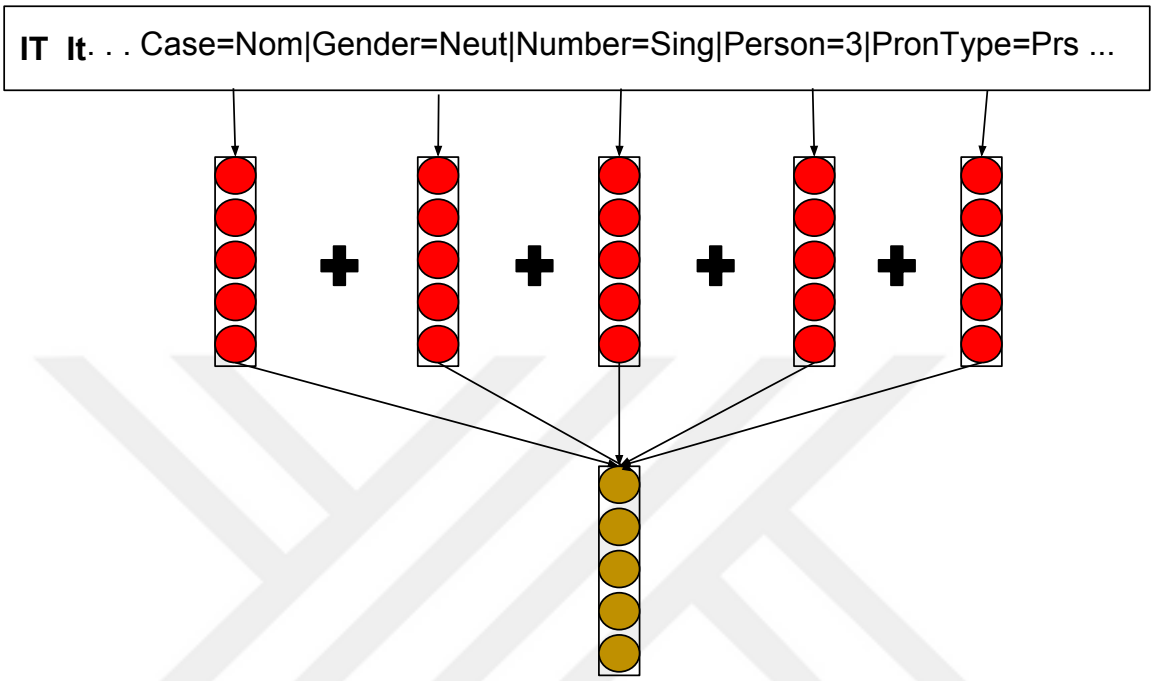

Figure 5.1: Processing of a word ”it” which has 5 unique morphological feature vector. Addition of these vectors corresponds to morph-feat embedding of that word.

- *Gender=Neut*
- *Number=Sing*
- *Person=3*
- *PronType=Prs*

Every $X = Y$ entry, each bullet point in above, corresponds to a unique 128 dimensional continuous vector. I added these vectors to obtain morphological feature vector of a word, and named it as morph-feat vector.

I experimented with dimension range from 32 to 256 for morph-feat vector. Parsing performance improved until dimension of 128. However, I could not see any further enhancement after that point. All the vectors are initialized from normal distribution, and learned during training. I am detailing on components taking morph-feat vector as an input in the remaining part of this chapter.

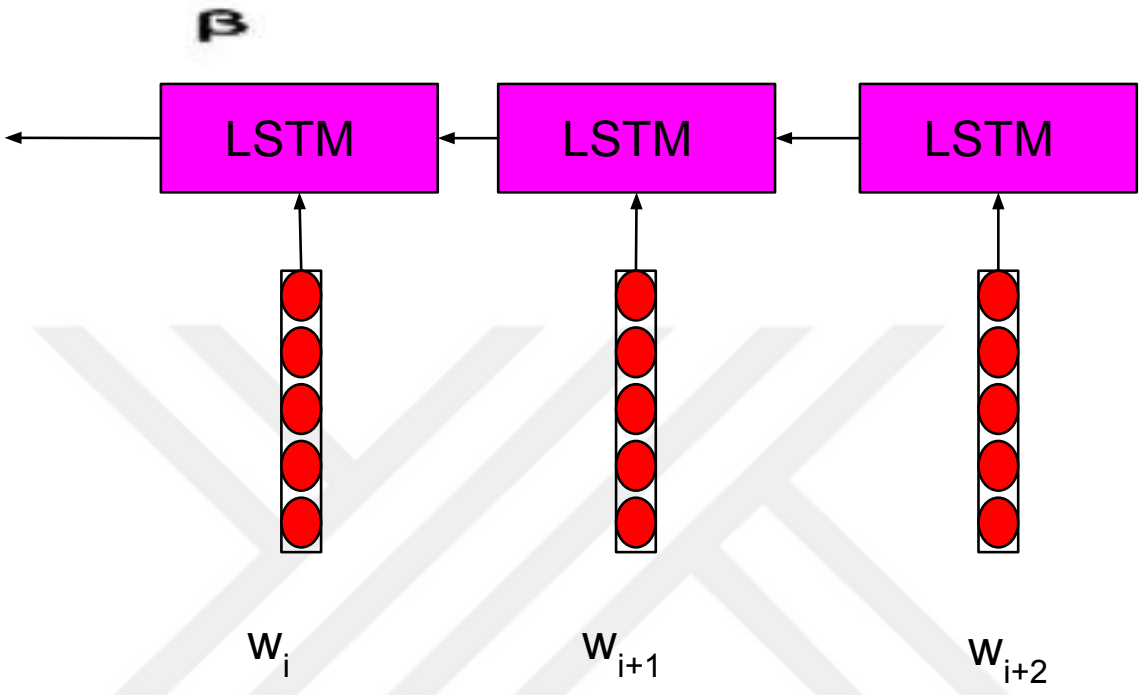

Figure 5.2: $\beta - LSTM$ processing a sentence. It starts to read from right to left. Each vector represents the concatenation of POS, language and morph-feat embeddings.

5.1.4 Dependency Label embeddings

There are 37 universal dependency labels. I initialized the embeddings for each of these labels with 2 random vectors, one for right the other for left transition. I took these vectors either from left or right based on the move taken in previous transition. Dependency label embeddings also became an input to t-RNN part of tree-stack LSTM. You may find the details for t-RNN in 5.1.7

5.1.5 Buffer's β-LSTM

The words of a sentence start their life in a buffer. Thus, each word of a sentence becomes an input to β-LSTM initially. The concatenation of POS, language, and morph-feat embeddings constitutes the input for β-LSTM. β-LSTM reads these words from right to left and its final hidden layer becomes buffer representation. Figure 5.2 depicts this flow.

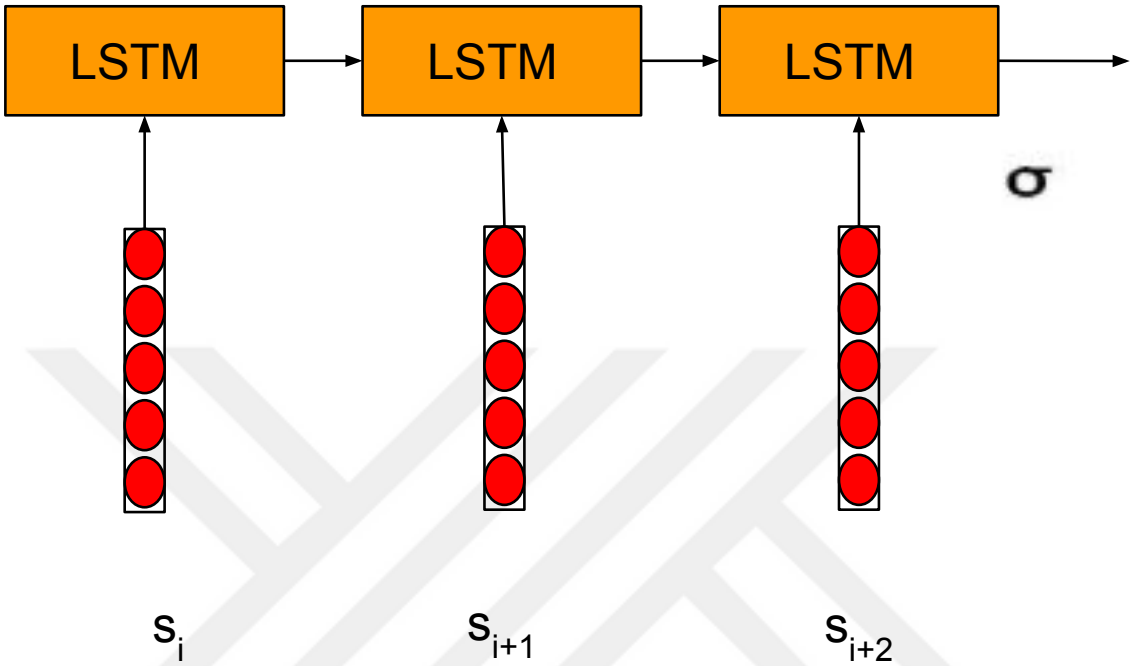

Figure 5.3: $\sigma - LSTM$ processing a sentence. It starts to read from left to right. Each vector represents word embeddings.

Although I experimented with both left-to-right and right-to-left directions, I found scanning from right to left performs better. I hypothesized that the information closer to word to be used is more important than the words waiting to be processed. I explained how the input embeddings of the top word in buffer β was updated based on tree-fragments in 5.1.7.

5.1.6 Stack's σ-LSTM

A stack may contain either a tree-fragment or a single word. σ-LSTM encodes this information for all words in stack and the final hidden layer is used to summarize this information. Similar to the β-LSTM, σ-LSTM uses POS, language and morph-feat embeddings for initial input-word embeddings. However, if a right transition taken stack's second last word becomes *"head"* of stack's top word, and that creates a tree-fragment in the stack. In order to represent a tree-fragment, I modified initial word representation by tree-RNN. I

explained the details of embedding modification in 5.1.7. Figure 5.3 demonstrates working principle of σ-LSTM which basically reads an embedding for each time step of an LSTM and the final hidden layer supposed to be summarize words and tree-fragments in stack.

5.1.7 Tree-RNN

As I previously mentioned, concatenation of POS, language and morph-feat embeddings initiated word's representation. Transition based parser builds parse tree step by step which creates tree-fragments that would be connected after some transitions. Therefore, we need to encode a word's children information respectively. To do that I employed an RNN modifying the initial word representation. Following equation defines that tree-RNN (t-RNN):

$$w_{new} = tanh([\mathbf{w_{head}}; \mathbf{dl}; \mathbf{w_{dep}}]) \tag{5.1}$$

t-RNN updated the representation of *"head"* word based on its *"dependent"* word and *"dependency relation"* between the *"head"* and the *"dependent"*. For instance, t-RNN amended buffer's top word embedding when the left transition taken. In other words, t-RNN calculated new *"head"* embedding, based on *"dependency relation"*, *"dependent"*, and previous *head* embedding. In Figure 5.4 head word embedding which needs to be updated is circled. Similarly, dependent word which effects head embedding is in dashed square. The output of t-RNN becomes new input for β-LSTM. After that, final hidden layer of β-LSTM was recalculated with new embedding. In other words, the output of t-RNN in Figure 5.4 became an input to a final stage of β-LSTM.

Similarly to left transition, t-RNN recalculated new *"head"* embedding when right transition taken. The update was based on the *"dependency relation"* and previous *"dependent"* embedding. In Figure 5.5 stack's second top word embedding, which was circled, need to be updated by t-RNN. The update was based on dependent word, which was pointed with a dashed square, and the dependency relation. The output of t-RNN became an input for σ-LSTM to update its hidden layers from the second stage. This showed that σ-LSTM's hidden layers were affected by tree-fragments.

The reason why I called it t-RNN is because the embeddings had memory. For instance, if a word had two children during parsing, the naive embedding was updated two times

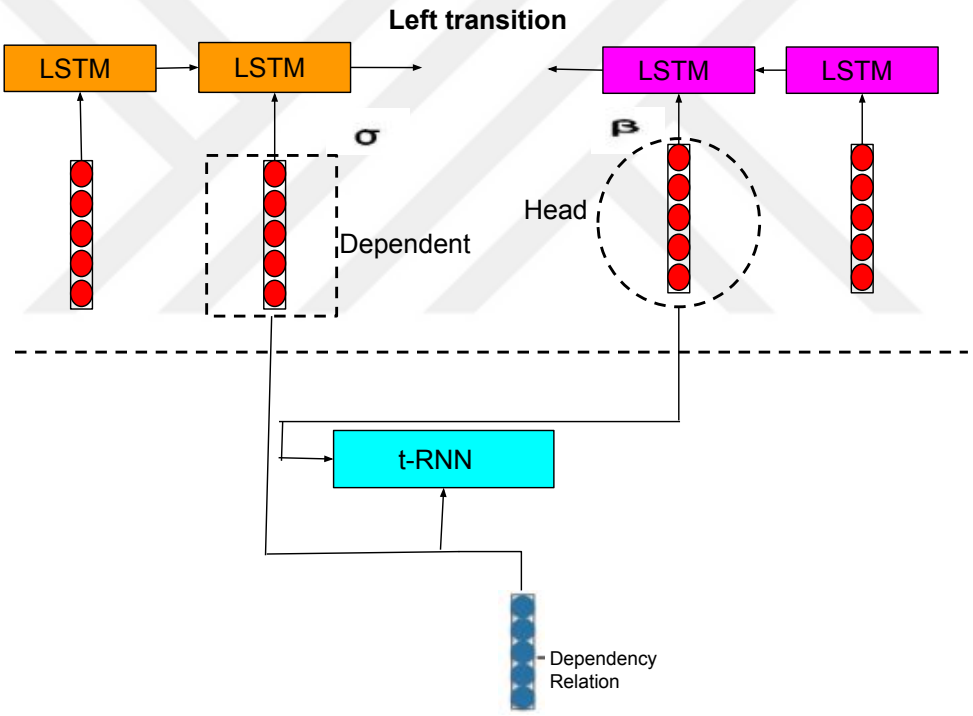

Figure 5.4: t-RNN left update flow.The output of t-RNN becomes new representation for buffer's top word.

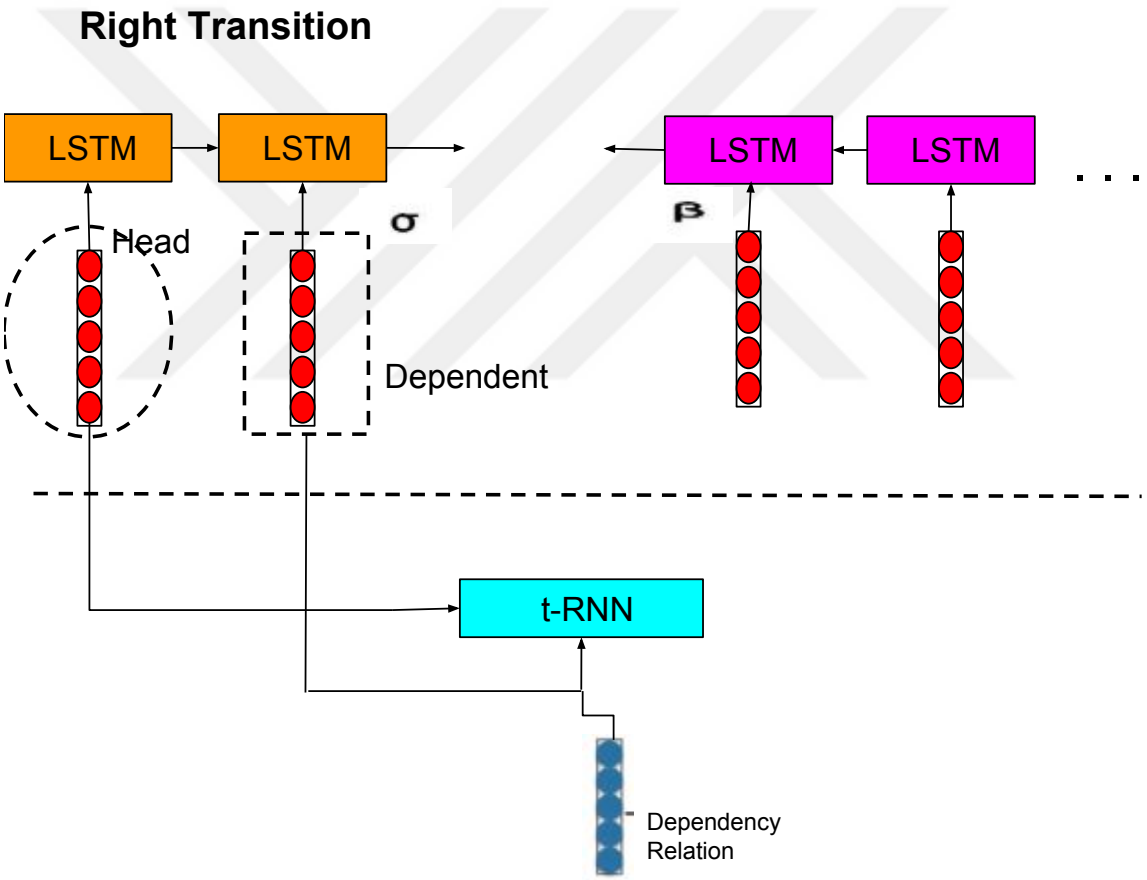

Figure 5.5: t-RNN right update flow.The output of t-RNN becomes new representation for stack's second top word.

sequentially. Thus, the previous embeddings of a *head* word may be considered as previous hidden state of an RNN. Last but not least, I experimented with different non-linearities for t-RNN, but I couldn't see any significant performance difference in between these non-linearities.

5.1.8 Action LSTM

There are 73 distinct actions for shift, labeled right and left actions in ArcHybrid transition system. I randomly initialized 128 dimensional vector for each labeled action and shift. This part is different from [Dyer et al., 2015]. They only initialized for labels but not for actions and used these embeddings in both action's LSTM and t-RNN.

After calculating the final hidden's from β, σ and Action LSTMs, I concatenated these hidden layers and applied it as an input to MLP which outputs probabilities for each transition.

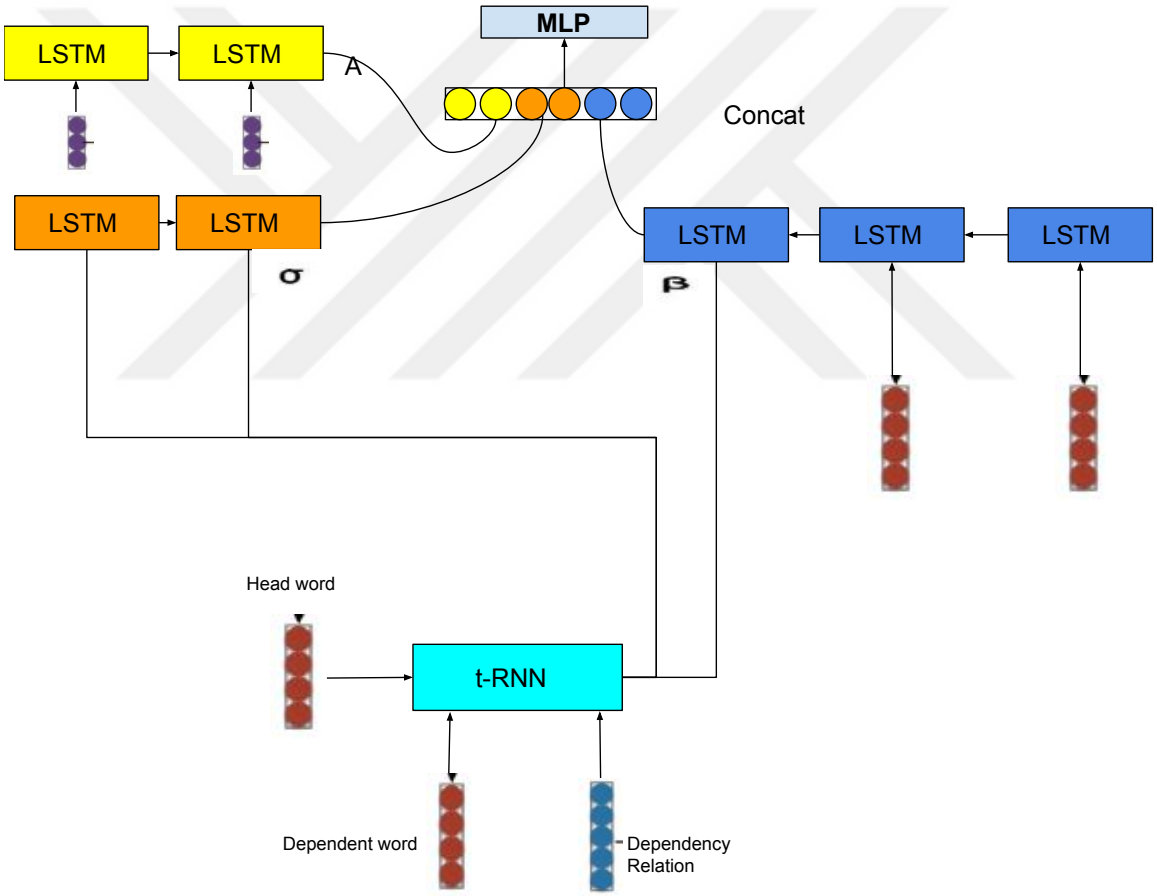

Figure 5.6: End-to-end tree-stack LSTM

Chapter 6

EXPERIMENTS AND RESULTS

We participated Conll17 and Conll18 UD Shared tasks to test MLP and tree-stack LSTM respectively. Furthermore, I designed different experiments to compare performances of MLP and tree-stack LSTM. In the remaining part of this chapter, I detailed these experiments and the results.

6.1 MLP Parser

In this section, I discussed our best/worst results relative to other task participants, and analyzed the benefit of using context vectors evaluated in Conll17 UD Shared Task [Zeman et al., 2017]. We submitted MLP parser (see chapter 4 for details.) to Conll17 UD Shared Task. Tables 6.2 and 6.3 presents our official results.

6.1.1 Best and worst results

Looking at the best/worst results may give insights into the strengths and weaknesses of the approach. Relative to other participants, Finnish, Hungarian, and Turkish are among our best languages: all agglutinative languages with complex morphology. This may be due to our character based language model which can capture morphological features when constructing word vectors. We ranked the first among other participants who only used transition based algorithms [Garcia and Gamallo, 2017]. Our worst results are in ancient languages: Ancient Greek, Gothic, Old Church Slavonic. We believe this is due to lack of raw text to construct high quality language models. Finally, the results for languages with large treebanks (Syntagrus and Czech) are also relatively worse than languages with smaller treebanks. A large treebank may offset the advantage of extra information we capture from a language model trained on raw text.

6.1.2 Impact of context vectors

Feats	**Hungarian**	**En-ParTUT**	**Latvian**
p	63.6	76.6	55.9
v	73.5	75.9	63
c	72.2	76	63.5
v-c	76	79	67.6
p-c	78	82.5	70.6
p-v	76.6	80.8	67.7
p-fb	74.7	79.7	66.3
p-v-c	79.3	83.2	74.2

Table 6.1: Feature comparison results on three languages. p=postag, v=word-vector, c=context-vector, fb=Facebook-vector.

To analyze the impact of context vectors and other embeddings on parsing performance, we performed experiments on three corpora (Hungarian, English-ParTUT, Latvian) with different feature combinations. These corpora were chosen for their relatively small sizes to allow quick experimentation. We tried eight different feature combinations on each language. In each setting, we used a different subset of context, word, and postag embeddings. The p-fb setting uses postag embeddings and Facebooks pre-trained word embeddings instead of the ones from our language model. We can make some observations consistent across all three languages based on the results in Table 6.1:

- Word vectors from our language model perform slightly better than Facebook vectors [Bojanowski et al., 2017] (p-v vs p-fb).

- Both part-of-speech tags and context vectors have significant contributions (comparing v with p-v or v-c).

- Context vectors seem to provide independent information on top of part-of-speech tags that significantly boosts parser accuracy (p-v vs p-v-c).

6.2 Tree-stack LSTM Parser

In this section, I described the details of experiments to compare previous MLP and tree-stack LSTM and the results. Finally, I gave the full results of our system in Conll18 UD Shared Task.

First, I divided Conll18 UD dataset 2.2 into 4 parts based on number of training tokens for each language:

- Languages having 100k tokens or more
- Languages having more than 50k less than 100k tokens
- Languages having more than 20k less than 50k tokens
- Languages having less than 20k tokens

I had different experimental strategies for each category above to obtain best performance. I described each strategy and how different languages performed in these strategies.

6.2.1 Training with Morphological Features

I changed initial word representation by concatenating morph-feat embedding to the other embeddings (see section 5.1.3 for details). Next, I had experiments with/without morph-feat embeddings and realized that in some cases they were useful. Morph-feat embedding improved performance for languages having training tokens in between 50k and 100k. However, they did not bring any enhancement for languages having more than 100k tokens. They, on the other hand, reduced the performance with languages having less than 50k tokens. Tables 6.4, 6.5, and 6.6 demonstrate the results that I obtained for each category. Finally, I submitted our system with morphological features for only languages having more than 50k training tokens.

The reason behind these experiments may be the model's capacity. The model's representational capacity increases when we add additional dimensional like morph-feat embeddings. However, learning those parameters becomes a challenge when we do not have

Language	**LM Perp.**	**Rank**	**LAS**	**Language**	**LM Perp.**	**Rank**	**LAS**
ar	99.21	13	66.14	hsb	Not used	17	50.25
ar_pud	99.21	12	44.97	hu	27.83	4	69.55
bg	25.60	9	84.95	id	52.64	9	75.54
bxr	Not used	14	24.96	it	27.97	10	86.45
ca	18.49	10	86.09	it_pud	27.97	10	84.52
cs	37.65	20	81.55	ja	29.14	18	72.67
cs_cac	44.87	15	82.91	ja_pud	29.14	15	76.27
cs_pud	37.65	20	78.57	kmr	Not used	4	42.11
cu	Not used	26	58.63	ko	34.60	8	71.70
de	33.98	11	72.44	la_ittb	59.28	16	76.15
de_pud	33.98	6	70.96	la_proiel	130.01	13	59.36
el	20.14	7	81.35	lv	37.81	6	63.63
en	44.50	15	75.96	nl	32.43	11	70.24
en_ParTUT	51.57	11	75.71	no_bokmaal	34.38	12	83.73
en_pud	44.50	11	79.51	no_nynorsk	31.03	9	82.72
es	26.33	7	83.34	pl	27.97	9	80.84
et	45.77	6	62.04	pt_pud	24.11	6	75.02
eu	39.92	8	71.47	ro	21.02	7	81.48
fa	63.29	12	79.56	ru	26.99	7	77.11

Table 6.2: Our official results in Conll17 Shared Task-1

Language	**LM Perp.**	**Rank**	**LAS**	**Language**	**LM Perp.**	**Rank**	**LAS**
fi_ftb	41.03	11	75.37	ru_syntagrus	29.36	20	85.24
es_ancora	26.33	9	85.63	pt	24.11	9	82.92
da	30.28	7	76.39	la	111.51	10	47.08
fi	29.36	5	77.72	ru_pud	26.99	3	71.2
cs_cltt	52.64	10	73.88	kk	715.23	17	22.34
es_pud	26.33	8	78.74	pt_br	33.6	10	86.7
fi_pud	29.36	4	82.37	sk	21.99	7	76.46
fr	18.76	9	81.30	sl_sst	194.75	8	49.56
fr_ParTUT	14.60	7	80.22	sme	Not used	4	37.93
fr_pud	18.76	6	76.04	sv	40.42	9	78.31
fr_sequoia	16.75	7	81.97	sv_lines	34.21	7	75.71
ga	56.32	8	63.22	sv_pud	40.42	6	72.36
en_lines	40.79	10	74.39	nl_lassysmall	35.62	8	80.85
gl	28.70	5	80.27	tr	57.31	6	56.8
gl_treegal	32.32	4	69.13	tr_pud	57.31	6	34.65
got	Not used	24	56.81	ug	866.74	21	31.59
grc	116.72	23	49.31	uk	36.16	6	63.76
grc_proiel	227.78	22	61.70	ur	105.38	11	77.64
he	78.75	10	58.98	vi	91.67	13	38.3
hi	37.36	10	87.23	zh	92.01	19	57.15
hi_pud	37.36	9	51.49	hr	33.29	7	79.22

Table 6.3: Our official results in Conll17 Shared Task-2

Lang code	**Morph-Feats**	**no Morph-Feats**	**#of tokens**
ko_gsd	73.74	72.54	56,687
got_proiel	54.33	53.24	35,024
id_gsd	75.76	73.97	97,531
nl_lassymal	76.7	75.8	75,134
fr_sequoia	84.36	82.17	50,543
gl_ctg	79.02	79.018	79,327
en_gum	76.44	75.34	53,686
lv_lvtb	72.33	72.24	80,666
eu_bdt	74.55	73.32	72,974
hu_szeged	66.23	68.18	20,166
sv_lines	72.18	74.81	48,325

Table 6.4: Morphological feature embeddings for some languages having tokens more than 50k and less than 100k training data

enough training data. Simple and complex models converge to the same place for languages having high training resource. Therefore, I could not see the performance improve for these languages.

6.2.2 Static vs. Dynamic Oracle Training

There was a single correct transition at each parsing step due to ArcHybrid transition's nature [Kuhlmann et al., 2011]. Correct sequence of transitions defined correct parsing path. The loss function is defined to maximize correct path's probability. However, applying the correct action or network's most probable action to the parser is an open question. This question was crucial because it determined the next step's state, LSTM's hidden in my case. Applying the correct action at each step is named as static oracle training, but applying the most probable action is named as dynamic oracle training. The model itself may learn to recover as many arcs as possible even if it took one wrong transition in dynamic oracle.

Lang code	**Morph-Feats**	**no Morph-Feats**	**#of tokens**
ar_padt	68.02	68.14	223,881
bg_btb	84.53	84.55	124,336
bg_btb	84.53	84.55	124,336
ca_ancora	85.89	85.874	417,587
cs_cac	83.57	83.63	472,608
cs_pdt	81.43	82.12	1,173,282
de_gsd	71.59	71.32	263,804
en_ewt	75.77	75.682	204,585
es_ancora	84.99	84.78	444,617
fa_seraji	81.18	81.12	121,064

Table 6.5: Morphological feature embeddings for some languages having tokens more than 100k training data

Lang code	**Morph-Feats**	**no Morph-Feats**	**#of tokens**
la_perseus	49.93	51.6	18,184
no_nynorsklia	51.13	53.33	3,583
ru_taiga	58.32	60.55	10,479
sl_sst	46.72	48.77	19,473
sme_giella	52.78	53.39	16,385
ug_udt	52.78	53.39	19,262
hu_szeged	66.23	68.18	20,166

Table 6.6: Morphological feature embeddings for some languages having tokens less than 20k in training data

Lang code	**Static No-Morp**	**Dyn No-Morp**	**Static Morp**	**Dyn Morp**
hu_szeged (20k)	67.12	**68.18**	66.23	67.12
tr_imst (40k)	**58.75**	57.49	56.67	56.32
id_gsd (98k)	73.97	74.54	**75.76**	75.48
ar_padt (220k)	**68.14**	66.44	68.02	67.89
ug_udt (20k)	52.99	**53.39**	50.13	51.67
ko_gsd (57k)	72.54	71.45	**73.74**	72.99
cs_pdt (1M)	**82.12**	81.70	81.43	82.07
es_ancora (450k)	84.78	83.49	**84.99**	84.42
sl_sst (20k)	**48.77**	48.5	46.72	45.97
nl_lassysmall (17k)	**76.7**	76.1	75.4	75.2

Table 6.7: The effect of static vs. dynamic oracle training together with morph-feats, parenthesis in lang code denotes number of training tokens

However, training a model with dynamic oracle might become a challenge. In order to find these difficulties with dynamic oracle and compare static-dynamic oracle training, I was required to design set of experiments.

Table 6.7 depicts that for languages having more than 50k tokens suffers from dynamic oracle training. For languages having training tokens in between 20k and 50k the performances are close. Why dynamic oracle training suffers is an open research question. I chose the best performances from the table 6.7 while submitting to Conll18 Shared Task.

6.2.3 Transfer Learning

In this subsection, I explained what is transfer learning, and my experimental designs to test it.

I might divide transfer learning into two categories:

- Transferring from similar tasks
- Transferring with different datasets in the same task

Transferring from similar tasks is based on having same component representation at some part of a model. For instance, I used sequence copying task to initialize action LSTM part of tree-stack LSTM. Although this did not effect the performance, it helped model to converge faster.

The other possibility is transferring a model. I trained a model with one dataset. Then, I transferred that model's parameters to initialize another model. Universal Dependencies (UD) Conll18 dataset contains many different languages with the same annotation style. Thus, I applied this technique for languages having training tokens less than 20k; and selected a parent from the same language family with the low resource language. However, I questioned what happens if I transfer from language without the same family. I provided my experimental results in Table 6.8.

Transfer learning between parser models almost always improved the parsing accuracy. As I inferred from Table 6.8, transferring in between same language family helped to improve results. Moreover, if there is more than one alternatives to transfer, parent language with more training data enhanced to improve parsing accuracy.

Transfer Learning for languages without LM data

There are some languages which contains very limited or no data for training my language model. In such cases, I have four options:

(i) Using very limited data to train LM for word and context vectors and use them to train a parser.

(ii) Using Facebook's word vectors [Bojanowski et al., 2017] to train a parser.

(iii) Using my own word and context vectors trained with different language but from the same language family.

(iv) Applying transfer learning with a pre-trained parser.

I chose Afrikaans-AfriBooms(af_afribooms), Kazakh (kk_ktb), Buryat(bxr_bdt) and Kurmanji (kmr_mg) languages to test (i), (ii), (iii), (iv). I picked English (en_ewt), Turk-

language	**parent-language**	**same family ?**	**las**
kmr_mg	not provided	no	18.12
kmr_mg	tr_imst	no	20.59
kmr_mg	fa_seraji	yes	**21.39**
ug_udt	not provided	no	51.12
ug_udt	bg_btb	yes	55.37
ug_udt	tr_imst	yes	**57.04**
kk_ktb	not provided	no	20.12
kk_ktb	bg_btb	yes	22.14
kk_ktb	tr_imst	yes	**23.86**
hy_armtdp	not provided	no	20.45
hy_armtdp	tr_imst	no	21.23
hy_armtdp	el_gdt	yes	**24.58**
hsb_ufal	not provided	no	24.42
hsb_ufal	tr_imst	no	26.54
hsb_ufal	cs_pdt	yes	29.31
hsb_ufal	bg_btb	yes	**30.81**
bxr_bdt	not provided	no	7.42
bxr_bdt	id_gsd	no	8.12
bxr_bdt	ru_syntagrus	no	**9.93**

Table 6.8: Transfer language experiments with low resource languages. I transferred parser models from parent to child.

Language	(i)	(ii)	(iii)	(iv)
af_afribooms	not provided	75.46	77.43	**78.12**
kk_ktb	20.19	22.31	21.96	**23.86**
bxr_bdt	7.64	9.76	**9.93**	8.98
kmr_mg	20.12	22.57	22.78	**23.39**

Table 6.9: LAS values for (i), (ii), (iii) and (iv)

ish (tr_imst), Russian (ru_syntagrus), and Persian (fa_seraji) languages for af_afribooms, kk_ktb, bxr_bdt and kmr_mg respectively, for transfer learning.

Looking at Table 6.9 gives insight about the experiments. First of all, most of the time my context embeddings performs better than Facebook's word vectors [Bojanowski et al., 2017] which was also valid in MLP parser. Moreover, training my language model with very limited data does not improve the parsing accuracy. However, transfer learning in between parsers performs better than using naive language vectors. Although Kazakh (kk_ktb) has its own limited language data, transfer learning in between parsers brings a significant improvement in parsing accuracy. Due to the character based language model, I was able to do transfer learning experiments for languages with the same alphabet.

6.2.4 Best and Worst Results

My team's average ranking is 16th within 26 teams in Conll18 UD Shared task. In this section, I am going to analyze our best results that are significantly better than 16 and worst results significantly worse than 16.

My team's best results are in two sets. Namely, small treebanks and low-resource languages.

Small treebanks do not contain development data, but very limited training data (10k-20k tokens). These languages are Galician Treegal (gl_treegal), Latin (la_perseus), Irish (ga_idt), North-Sami Giella (sme_giella), Norwegian-NynorskLIA (no_nynorsklia), Russian-Taiga(ru_taiga) and Slovenian-SST (sl_sst). My team ranked 10th in these languages. Low-

Resource languages do not contain development data, and training data is too tiny (0-1k tokens). These languages are Buryat-BDT (bxr_bdt), Upper Sorbian-UFAL(bsh_ufal), Armenian (hy_armtdp), Kazakh(kk_ktb), Breton-Keb (br_keb), Faroese-OFT (fo_oft), Naija-NSC (pcm_nsc), Thai-PUD (th_pud). My team ranked 11th in these languages. Looking at these results, my model's strength lies on the convergence with low resource languages.

My team's worst results are in ancient languages, i.e Ancient-Greek (grc_proiel), Ancient_Greek-Perseus (grc_perseus), and in languages with more than 1 million training tokens, i.e. Czech (cs_pdt). The reason why my model performs worse with ancient languages may be the low quality of word vectors that I obtained from my language model. Although I tried all different transfer learning options, I could not get significant performance boost. A large treebank, on the other hand, may offset the advantage of my tree-stack LSTM model and/or context embeddings.

Finally, I provided full results in Tables 6.10 and 6.11 that my team obtained in Conll18 UD Shared Task.

6.3 Comparison of MLP and Tree-stack LSTM Parser

My main motivation to build tree-stack LSTM parser was to reduce hand-crafted feature engineering. In order to compare these two models, I designed an ablation analysis where the model started from an MLP and evolved to a tree-stack LSTM parser.

6.3.1 Ablation analysis

I selected datasets from each category that I explained in 6.2. Namely, Hungarian (hu_szeged), Swedish-LinES (sv_lines), Turkish (tr_imst), Arabic (ar_padt), Czech-Cac(cs_cac), and English-Ewt (en_ewt).

The first step in ablation analysis was to put LSTM instead of MLP for decision mechanism. I named this model as *"only Action LSTM"*. I extracted the same features explained in Figure 4.1 for that model too. Yet, the decision mechanism became an LSTM. Since the LSTM's hidden is updated via the transitions, that model became very similar to using *"only Action LSTM"* from tree-stack LSTM. As I inferred from the Table 6.12, these two

Language	**LAS**	**MLAS**	**BLEX**	**Ranks**	**Language**	**LAS**	**MLAS**	**BLEX**	**Ranks**
af_afribooms	78.12	65.12	63.93	17-15-19	ar_padt	68.02	58.05	61.21	16-11-13
bg_btb	84.53	75.58	77.63	20-16-9	br_keb	8.91	0.35	1.77	19-15-19
bxr_bdt	9.93	0.49	0.69	18-22-23	ca_ancora	85.89	77.22	77.22	18-17-17
cs_cac	83.57	75.3	77.24	19-12-18	cs_fictree	82.67	71.93	75.31	18-15-16
cs_pdt	81.43	73.77	71.44	21-21-22	cs_pud	78.69	64.14	84.52	18-19-17
cu_proiel	59.42	46.96	81.55	23-22-21	da_ddt	76.4	67.05	63.09	17-15-19
en_ewt	75.77	66.78	68.76	20-20-18	en_gum	76.44	65.19	64.32	16-12-15
de_gsd	71.59	46.87	35.41	17-8-23	el_gdt	83.34	65.74	66.6	16-14-18
en_lines	73.96	64.91	62.76	17-15-19	en_pud	78.41	66.16	69.25	19-19-17
es_ancora	84.99	76.71	77.06	18-17-16	et_edt	74.52	65.7	63.5	20-19-17
eu_bdt	74.55	63.11	61.25	17-13-18	fa_seraji	81.18	74.84	71.65	17-16-14
fi_ftb	75.84	65.53	67.91	17-16-11	fi_pud	81.55	74.18	66.29	13-12-10
fi_tdt	78.42	70.4	65.89	16-15-12	fo_oft	22.5	0.29	5.44	20-19-20
fr_gsd	81.07	72.07	73.96	19-19-17	fr_sequoia	84.36	76.56	71.33	14-10-20
fr_spoken	57.32	15	57.76	10-10-10	fro_srcmf	76.92	67.85	71.35	20-19-20
ga_idt	63.13	34.36	40.76	13-19-16	gl_ctg	79.02	66.13	71.12	14-14-9
gl_treegal	70.45	52.15	56.38	10-9-9	got_proiel	54.33	40.58	40.51	24-24-22
grc_perseus	55.03	34.19	37.0	20-15-16	grc_proiel	62.11	43.92	37.00	22-21-20
he_htb	58.28	45.06	48.09	18-16-16	hi_hdtb	86.86	70.44	79.98	20-17-15
hr_set	81.6	65.23	68.74	17-8-18	hsb_ufal	30.81	6.22	15.39	8-9-7
hu_szeged	68.18	51.13	50.53	14-20-21	hy_armtdp	24.58	7.24	6.96	12-9-21
id_gsd	75.51	65.02	63.92	17-13-17	it_isdt	85.80	75.5	70.16	22-21-22

Table 6.10: Our official results in Conll18 Shared Task

Language	**LAS**	**MLAS**	**BLEX**	**Ranks**	**Language**	**LAS**	**MLAS**	**BLEX**	**Ranks**
it_postwita	70.03	56.04	47.53	13-12-20	ja_gsd	73.30	59.46	59.89	14-14-20
ja_modern	23.35	8.94	10.13	5-4-5	kk_ktb	23.86	5.98	8.02	11-11-13
kmr_mg	23.39	3.97	7.56	15-15-16	ko_gsd	73.74	67.31	60.52	19-18-16
ko_kaist	78.81	71.49	65.29	19-19-16	la_ittb	75.79	71.49	71.66	20-19-19
la_perseus	51.6	33.65	38.04	11-8-8	la_proiel	59.35	46.36	51.13	20-19-19
lv_lvtb	72.33	57.08	60.54	16-13-14	nl_alpino	78.83	64.22	65.79	17-16-15
nl_lasysmall	76.70	63.97	64.58	16-15-15	no_bokmaal	82.32	37.93	73.52	20-19-18
no_nynorsk	80.57	70.78	72.27	20-19-17	no_nynorskla	53.33	41.01	44.46	13-10-11
pcm_nsc	15.84	5.3	13.61	10-1-11	pl_lfg	86.12	71.96	76.71	21-21-20
pl_sz	82.83	63.76	73.05	16-17-14	pt_bosque	82.71	68.01	73.07	18-17-15
ro_rrt	80.90	72.39	72.59	17-14-14	ru_syntagrus	82.89	56.8	75.48	20-19-18
ru_taiga	60.55	39.41	44.05	11-9-9	sk_snk	75.75	53.49	61.0	20-20-16
sl_ssj	78.18	62.94	69.49	16-18-14	sl_sst	48.77	34.66	39.82	10-11-9
sme_giella	53.39	39.13	41.75	19-19-15	sr_set	80.85	67.46	72.09	20-19-16
sv_lines	74.81	59.72	67.35	17-15-15	sv_pud	70.77	43.04	54.67	16-12-15
sv_talbanken	77.91	69.25	69.87	17-16-17	th_pud	0.74	0.04	0.44	7-8-8
tr_imst	58.75	48.28	49.84	15-12-13	ug_udt	57.04	36.63	44.44	16-16-14
uk_iu	76.5	36.63	65.5	16-16-13	ur_udtb	78.12	51.25	64.66	17-15-15
vi_vtb	40.48	33.88	36.03	16-14-15	zh_gsd	59.76	50.87	53.11	19-15-18

Table 6.11: Our official results in Conll18 Shared Task-2

Lang Code	**MLP**	**Only Action**	**Only-β**	**Only-σ**
hu_szeged	66.21	66.87	66.94	67.03
sv_lines	71.12	72.05	72.17	72.45
tr_imst	57.12	56.87	57.02	57.12
ar_padt	67.83	66.67	66.89	66.92
cs_cac	83.89	82.23	83.13	83.17
en_ewt	75.54	75.43	75.56	75.67

Table 6.12: Comparison between MLP and "Only" models

models did not differentiate in parsing performance. I hypothesized that choosing the right features for MLP offsets the memory advantage of an LSTM.

Next, I removed the buffer related features from the MLP parser, instead, I put $\beta - LSTM$ scanning all the words in buffer-β. The decision module is still an MLP, but the buffer related features replaced with $\beta - LSTM$'s final hidden. I named this model *"only-*$\beta - LSTM$*"*. I deduced from the Table 6.12, *"only-*$\beta - LSTM$*"* performs slightly better than the MLP.

Instead of buffer-β related features, I removed stack-σ related features from the MLP. In other words, the decision module is still an MLP, but the stack related features are replaced with $\sigma - LSTM$'s final hidden. I named this model *"only-*$\sigma - LSTM$*"*. I reasoned from the Table 6.12 that, *"only-*$\sigma - LSTM$*"* also brings a performance improvement.

In the next stage, I removed the tree-RNN (t-RNN) from the tree-stack LSTM model. I realized that the performance loss with low-resource languages is better than the high resource languages. As a result, I decided to include more small treebanks to this study.

I concluded from the Table 6.13, t-RNN gives comparable advantage with low resource languages. I supposed that interconnecting the model components, i.e. σ-LSTM and β-LSTM with t-RNN, enables model to learn/convergence better with low resource languages. However, high-resource languages offsets this advantage. This may be due to reaching the models' capacities in both cases. My final observation is that removing

Lang Code	**without t-RNN**	**with t-RNN**
ar_padt	68.04	68.14
cs_cac	82.89	83.57
en_ewt	74.87	75.77
sv_lines	74.04	74.46
tr_imst	58.12	58.75
hu_szeged	66.12	**68.18**
gl_treegal	69.76	**70.45**
ru_taiga	59.13	60.55
no_nynorsklia	51.78	**53.33**

Table 6.13: Comparison of stack-LSTMs with and without t-RNN

t-RNN reduces our model's comparative advantage in low-resource languages in Conll18 UD Shared Task.

Last but not least, my main purpose to implement tree-stack LSTM was not to use hand-crafted features in TBDP. In order to measure the success of my model, I trained both models (MLP and tree-stack LSTM) with the same datasets and compared their performances.

As the Table 6.14 shows that my tree-stack LSTM performs better than the MLP model. Moreover, similarly to the previous results, tree-stack's low-resource languages performance boost is better than high-resource languages.

To sum up, I had two purposes in this thesis:

- Testing the context embedding's performance
- Reducing hand-crafted features

In the first part of this chapter, I showed that the context embeddings performs either better or very close with current state-of-the-art embeddings. The experiments suggested

Lang Code	**MLP**	**tree-stack**
tr_imst	56.78	**58.75**
hu_szeged	66.21	**68.18**
en_ewt	74.87	**75.77**
ar_padt	67.83	**68.14**
cs_cac	83.39	83.57
ru_taiga	58.89	**60.55**

Table 6.14: Comparison of MLP and tree-stack LSTM models

that context embeddings provided additional semantic information to better perform in dependency parsing. In the next set of experiments, I attempted to the question of reducing hand-crafted features with a better model. Tree-stack LSTM was a result of that effort. I improved the parsing accuracy without extracting any state related features. Next, I examined the tree-stack LSTM with an ablation study. I realized that t-RNN was a crucial component to learn in low-resource languages and morph-features were important to learn in mid/high resource languages. Moreover, I hypothesized that interconnecting model components helps model to convergence faster which was t-RNN in my case. The comparison of two models, MLP and tree-stack LSTM, proved that tree-stack LSTM performs better. Applying the ideas such as context embeddings, morph-feats to a graph based dependency parsing is still an open research question.

Chapter 7

CONCLUSION

In this thesis, I proposed deep neural networks based context embeddings. I tested these context embeddings in Conll17 UD Shared Task within a Multi Layer Perceptron (MLP) model extracting additional hand-crafted features. Yet, these experiments suggested that context embeddings boosted performance more than current state-of-the-art word embeddings [Bojanowski et al., 2017]. However, I observed that choosing the correct feature set for the MLP model is crucial. I researched the question of removing hand-crafted features from the model without any performance loss. I modified stack-LSTM [Dyer et al., 2015] and proposed tree-stack LSTM using only morphological features, part-of-speech, and context embedding. I, together with my team, participated in Conll18 Shared Task to test tree-stack LSTM. My results revealed that tree-stack LSTM performs better than MLP without using any hand-crafted features. I, further analyzed the effect of model components as well as context embeddings. Again, my context embeddings performed better than the other embeddings. I also discovered that t-RNN component of tree-stack LSTM was the key part to accomplish comparable results in low-resource languages. Furthermore, I found transfer learning from high-resource to low-resource languages very useful.

Chapter 8

FUTURE WORK

The main problem in transition based algorithms is their locality. This is the major weakness of transition based dependency parsing - having no possibility of undoing a decision. Beam search heuristically solves this problem within a specific beam width. Beam training should help to solve locality problem. However, I encountered with difficulties about convergence, when I implemented beam training. I discovered that removing non-linearity from MLP solves the convergence problem, but the performance was poor. I think using another loss function like CRF might address this problem. In addition, global normalization [Andor et al., 2016b] might also help to overcome locality problem. In light of my experiments, interconnecting model components helps to learn from low-resources. Attention [Bahdanau et al., 2014] is a good example of interconnecting model components, so adding attention to tree-stack LSTM is also a valuable direction. Exercising context embeddings in current state-of-the-art dependency parsers, which are based on graph based algorithms, [Dozat and Manning, 2016] might lead to improvement. Tree-Stack LSTM uses randomly initialized and learned continuous vectors for morphological feature embeddings. These vectors may be extracted from another pre-trained component, such as LSTM, MLP or CNN. Most of the systems in shared tasks used end-to-end trained systems for tokenization, pos-tagging, lemmatization and dependency parsing. Yet, my system is only trained for dependency parsing. End-to-end training tree-stack LSTM with other models might also help to enhance performance.

BIBLIOGRAPHY

[Alberti et al., 2017] Alberti, C., Andor, D., Bogatyy, I., Collins, M., Gillick, D., Kong, L., Koo, T., Ma, J., Omernick, M., Petrov, S., Thanapirom, C., Tung, Z., and Weiss, D. (2017). Syntaxnet models for the conll 2017 shared task. *CoRR*, abs/1703.04929.

[Andor et al., 2016a] Andor, D., Alberti, C., Weiss, D., Severyn, A., Presta, A., Ganchev, K., Petrov, S., and Collins, M. (2016a). Globally normalized transition-based neural networks. *CoRR*, abs/1603.06042.

[Andor et al., 2016b] Andor, D., Alberti, C., Weiss, D., Severyn, A., Presta, A., Ganchev, K., Petrov, S., and Collins, M. (2016b). Globally normalized transition-based neural networks. *arXiv preprint arXiv:1603.06042*.

[Bahdanau et al., 2014] Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

[Bansal et al., 2014] Bansal, M., Gimpel, K., and Livescu, K. (2014). Tailoring continuous word representations for dependency parsing. In *ACL (2)*, pages 809–815.

[Baskaya et al., 2013] Baskaya, O., Sert, E., Cirik, V., and Yuret, D. (2013). Ai-ku: Using substitute vectors and co-occurrence modeling for word sense induction and disambiguation. In *Second Joint Conference on Lexical and Computational Semantics (* SEM), Volume 2: Proceedings of the Seventh International Workshop on Semantic Evaluation (SemEval 2013)*, volume 2, pages 300–306.

[Bengio et al., 2003] Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155.

[Bojanowski et al., 2017] Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. (2017). Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146.

[Che et al., 2017] Che, W., Guo, J., Wang, Y., Zheng, B., Zhao, H., Liu, Y., Teng, D., and Liu, T. (2017). The hit-scir system for end-to-end parsing of universal dependencies. *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 52–62.

[Chen and Manning, 2014] Chen, D. and Manning, C. D. (2014). A fast and accurate dependency parser using neural networks. In *EMNLP*, pages 740–750.

[Collobert et al., 2011] Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537.

[Dozat and Manning, 2016] Dozat, T. and Manning, C. D. (2016). Deep biaffine attention for neural dependency parsing. *arXiv preprint arXiv:1611.01734*.

[Dozat et al., 2017] Dozat, T., Qi, P., and Manning, C. D. (2017). Stanford's graph-based neural dependency parser at the conll 2017 shared task. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 20–30, Vancouver, Canada. Association for Computational Linguistics.

[Dyer et al., 2015] Dyer, C., Ballesteros, M., Ling, W., Matthews, A., and Smith, N. A. (2015). Transition-based dependency parsing with stack long short-term memory. *CoRR*, abs/1505.08075.

[Eisner, 1996] Eisner, J. M. (1996). Three new probabilistic models for dependency parsing: An exploration. In *Proceedings of the 16th conference on Computational linguistics-Volume 1*, pages 340–345. Association for Computational Linguistics.

[Garcia and Gamallo, 2017] Garcia, M. and Gamallo, P. (2017). A rule-based system for cross-lingual parsing of romance languages with universal dependencies. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 274–282, Vancouver, Canada. Association for Computational Linguistics.

[Ginter et al., 2017] Ginter, F., Hajič, J., Luotolahti, J., Straka, M., and Zeman, D. (2017). CoNLL 2017 shared task - automatically annotated raw texts and word embeddings. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics, Charles University.

[Graves and Schmidhuber, 2005] Graves, A. and Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5-6):602–610.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[Kim et al., 2015] Kim, Y., Jernite, Y., Sontag, D., and Rush, A. M. (2015). Character-aware neural language models. *CoRR*, abs/1508.06615.

[Kingma and Ba, 2014] Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[Kiperwasser and Goldberg, 2016a] Kiperwasser, E. and Goldberg, Y. (2016a). Simple and accurate dependency parsing using bidirectional LSTM feature representations. *CoRR*, abs/1603.04351.

[Kiperwasser and Goldberg, 2016b] Kiperwasser, E. and Goldberg, Y. (2016b). Simple and accurate dependency parsing using bidirectional lstm feature representations. *arXiv preprint arXiv:1603.04351*.

[Kudo and Matsumoto, 2002] Kudo, T. and Matsumoto, Y. (2002). Japanese dependency analysis using cascaded chunking. In *proceedings of the 6th conference on Natural language learning-Volume 20*, pages 1–7. Association for Computational Linguistics.

[Kuhlmann et al., 2011] Kuhlmann, M., Gómez-Rodríguez, C., and Satta, G. (2011). Dynamic programming algorithms for transition-based dependency parsers. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pages 673–682. Association for Computational Linguistics.

[McDonald et al., 2005] McDonald, R., Pereira, F., Ribarov, K., and Hajič, J. (2005). Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pages 523–530. Association for Computational Linguistics.

[Medsker and Jain, 2001] Medsker, L. and Jain, L. (2001). Recurrent neural networks. *Design and Applications*, 5.

[Mikolov et al., 2013a] Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

[Mikolov et al., 2010] Mikolov, T., Karafiát, M., Burget, L., Černockỳ, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*.

[Mikolov et al., 2013b] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013b). Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119.

[Nivre, 2003] Nivre, J. (2003). An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT*, pages 149–160.

[Nivre et al., 2016] Nivre, J., De Marneffe, M.-C., Ginter, F., Goldberg, Y., Hajic, J., Manning, C. D., McDonald, R. T., Petrov, S., Pyysalo, S., Silveira, N., et al. (2016). Universal dependencies v1: A multilingual treebank collection. In *LREC*.

[Rosenfeld, 1996] Rosenfeld, R. (1996). A maximum entropy approach to adaptive statistical language modeling.

[Shi et al., 2017] Shi, T., Wu, F. G., Chen, X., and Cheng, Y. (2017). Combining global models for parsing universal dependencies. *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 31–39.

[Werbos, 1990] Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560.

[Yamada and Matsumoto, 2003] Yamada, H. and Matsumoto, Y. (2003). Statistical dependency analysis with support vector machines. In *Proceedings of IWPT*, volume 3, pages 195–206. Nancy, France.

[Yatbaz et al., 2012] Yatbaz, M. A., Sert, E., and Yuret, D. (2012). Learning syntactic categories using paradigmatic representations of word context. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 940–951. Association for Computational Linguistics.

[Zeman et al., 2018] Zeman, D., Hajič, J., Popel, M., Potthast, M., Straka, M., Ginter, F., Nivre, J., and Petrov, S. (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 1–20, Brussels, Belgium. Association for Computational Linguistics.

[Zeman et al., 2017] Zeman, D., Popel, M., Straka, M., Hajic, J., Nivre, J., Ginter, F., Luotolahti, J., Pyysalo, S., Petrov, S., Potthast, M., et al. (2017). Conll 2017 shared

task: multilingual parsing from raw text to universal dependencies. *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 1–19.

[Zhang and Nivre, 2011] Zhang, Y. and Nivre, J. (2011). Transition-based dependency parsing with rich non-local features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers-Volume 2*, pages 188–193. Association for Computational Linguistics.