LSTM Feature Representation in Projective and Non-Projective Transition-Based Dependency Parsers

Lena Reisinger

MInf Project Part 2
Master of Informatics
School of Informatics
University of Edinburgh

2017

Abstract

The objective of the first part of my project was to investigate how allowing for non-projectivity influences the attachment accuracy of transition-based dependency parsers. I implemented four parsers, two of which were based on novel transition systems. Since the novel non-projective parser achieved the highest attachment scores, I concluded that allowing for all kinds of dependency arcs gives better results than restricting the output to only projective ones, for languages with a large number of non-projective sentences. However, the parsing accuracies achieved were still significantly lower than state-of-the-art results, which motivated me to explore how they could be raised. In the course of the second part of my project I have evaluated two possible ways of improving the binary indicator feature model I was using. I first conducted experiments leading me to believe that extending that feature model with additional features would not allow me to achieve this goal. Inspired by the recent trend of using recurrent neural networks in dependency parsing, I investigated how I could apply such a model to the four parsers. I decided to implement a model where each word in the sentence is represented by a BiLSTM capturing its whole context and thereby allowing the multi-layer perceptron used as scoring function to use information about the whole sentence when making parsing decisions. Experiments showed that this led to significant improvements, both in attachment accuracies and training and parsing times. They also agreed with last year's results in the respect that the highest attachment accuracies could be achieved using the non-projective parsers. Matching state-of-the-art results allowed me to argue about the validity of my novel transition systems.

Table of Contents

Chapter 1

Introduction

During the first part of my project I have implemented four different transition-based dependency parsers, two of which were projective and two non-projective. The aim was to explore how allowing for dependency trees with crossing arcs influences the attachment accuracies of these parsers. I have shown that when evaluating the parsers on languages with a large number of non-projective sentences, the highest results can be achieved when using a non-projective parser. A novel parser based on the well-known non-projective transition system by Covington led to the highest results. This parser differs from the original one in the sense that instead of allowing us to add arcs to any previously encountered word whenever we consider the next word of the sentence, it lets us to restrict the words we can still add arcs to and from by adding reduce transitions.

These results were, however, still lower than results of state-of-the-art parsers, making it hard to argue about whether adding reduce transitions to Covington's parser really is an improvement. Therefore my goal this year is to get better attachment accuracies from the dependency parsers I have implemented last year. This will be done by improving the feature model I am using. Instead of a large and sparse vector indicating whether a specific feature is present in the current configuration, I will use bidirectional long short-term memory networks (BiLSTMs) to automatically learn relevant feature combinations, which has been shown to lead to better dependency parsing accuracies. BiLSTMs are neural networks that are good at modelling sequential data, such as sentences, where long-term dependencies are important.

Chapter 2 of this report will give an introduction to dependency parsing and neural networks. I will summarise my contributions to the project in section 2.3. In the following chapter, Chapter 3, I will describe the work carried out during the first part of my project, going into detail about the feature model I was using. The next chapter, Chapter 4, will evaluate two different ways this feature model could be improved by first summarising experiment results from extending the binary feature model I implemented last year and then analysing the successful applications of neural networks to natural language processing tasks. I will conclude by explaining why I decided to im-

plement the BiLSTM feature model introduced in (Kiperwasser and Goldberg, 2016). This is followed by a chapter describing my implementation of the new feature model. After that, in Chapter 6, I have presented the results of my experiments which consisted of evaluating all parsers on six different languages with a significant number of non-projective sentences. This is followed by Chapter 7, describing the conclusions that can be drawn from my results.

Chapter 2

Background

This chapter includes a section on dependency parsing and one on neural networks. I will also summarise what I have contributed in the course of my project. Part of the material in this chapter on dependency parsing has been drawn from (Kübler et al., 2009), while the section on neural networks makes use of (Goldberg, 2015).

2.1 Dependency

There are two main ways the syntactic structure of language can be viewed. The first one is constituency structure, where sentences are built up from nested constituents. To do this we need rules defining how to make up constituents from other constituents and terminals, which are words. The constituent *S* (which stands for sentence) can for example be created by combining a *VP* (verb phrase) with a *NP* (noun phrase) using the rule $S \rightarrow VP\ NP$.

The other way to view sentence structure is by specifying the words that depend on each word. Dependency structure refers to syntactic relations between words in a sentence, where one is the head and the other the dependent. The dependent can be referred to as the syntactic subordinate of its head. Every word has exactly one head, which means that dependencies can be represented as a rooted tree. Each node in a dependency tree corresponds to a terminal (a word), whereas nodes in a constituency tree can correspond to either constituents or terminals. Advantages of dependency structure are that dependency trees are often simpler than corresponding constituency trees and

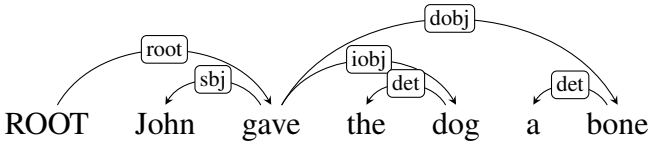

Figure 2.1: Dependency relations of an example sentence.

they are usually better suited to represent sentences of languages with flexible word order.

Dependents are either:

- Arguments: In the example sentence in Figure 2.1, *John*, *dog* and *bone* are arguments of *gave*.
- Modifiers (sometimes called adjuncts): In Figure 2.1, *the* is a modifier of *dog* and *a* is a modifier of *bone*.

As Figure 2.1 shows, a dependency relation is represented as an arrow pointing from the head to the dependent. An artificial word *ROOT* is inserted at the beginning of the sentence, which has an arc to the syntactic head word of the sentence, in this case *gave*. Each dependency arc has a label associated with it, describing the kind of relation between the two words. In the example in Figure 2.1, we have the relation that *John* is the subject of *gave*, therefore the label of the arc connecting the two is *sbj*. *Dog* is the indirect object of *gave*, indicated by label *iobj*, *bone* is its direct object with label *dobj* and *the* and *a* are determiners, which means their dependency label is *det*. Formally we define an arc as (i, l, j), which means that there is a dependency relation from the ith to the jth word in the sentence which is of the kind l.

2.1.1 Dependency Parsing

The problem of dependency parsing consists of automatically finding the dependency tree of a given input sentence. Dependency parsers can either be data-driven, meaning that machine learning approaches are used to learn a model from annotated data, or grammar-based, where formal grammars describe the kind of dependency trees that are possible.

Data-driven dependency parsing can be split into two sub-problems:

- Learning problem: This entails learning a model from a text corpus where each word is annotated with its head and the type of dependency to its head, that can then be used to find the dependency tree of an input sentence.
- Parsing problem: Given an input sentence, use the model to output its dependency tree.

Data-driven models can be categorised into graph-based parsers and transition-based parsers. For graph-based parsers the learning problem consists of learning a model that can assign likelihood scores to all possible dependency trees for a sentence. Parsing then consists of finding the dependency tree with the highest score, which can be achieved with dynamic programming.

Transition-based parsers build a dependency tree for a sentence by applying a sequence of transitions to it. The learning problem entails computing a model that predicts the next most likely transition for a given partially parsed sentence and the parsing problem

means using this model to predict a sequence of transitions that builds a dependency tree for a given input sentence.

A main difference between the two approaches is that transition-based dependency parsers are greedy and only choose locally optimal next transitions, whereas graph-based ones build the globally most likely parse tree. On the other hand, transition-based parsers can use much richer features and take much more information about the parsing history into account when making parsing decisions than the graph-based parsers.

2.1.2 Non-Projectivity

Many dependency parsers allow only for projective dependency trees, meaning none of the arcs are crossing each other. An example for a non-projective sentence can be seen in Figure 2.2 where the arcs *saw* $\rightarrow$ *yesterday* and *dog* $\rightarrow$ *was* are crossing each other. Formally, non-projectivity can be defined in the following way:

Definition 2.1.1. Non-projectivity: Let $w = w_1, w_2, .., w_n$ be a sentence and w_0 the artificial root node. It is non-projective if its dependency tree contains arcs (i, l, j) and (i', l', j') (we are assuming w.l.o.g. that $i < j$ and $i' < j'$) where the integers $i, j, i', j' \in [0, n]$ and $i < i' < j < j'$.

Restricting the parser to non-projective structures is a reasonable assumption for languages such as English where the number of crossing arcs is very small, but too constrained for several other languages, especially those permitting free word order to some degree.

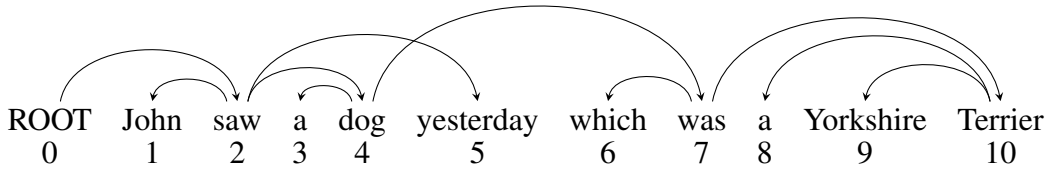

Figure 2.2: Dependency relations of an example non-projective sentence.

2.1.3 Transition-Based Dependency Parsing

Transition-based dependency parsing describes the process of transforming a sentence into its corresponding dependency tree through a sequence of transitions. Such a transition system is defined by an initial configuration, a set of transitions that transform one configuration into the next one, and a set of final configurations which indicate the end of a parse (Nivre, 2008). A configuration typically includes a buffer β with tokens that were not processed yet, a stack σ containing tokens that have been processed

Transition	$(\sigma,$	$\beta,$	$A)$
	$[ROOT],$	$[John, gave, the, dog, a, bone],$	$A = \emptyset$
SHIFT $\Longrightarrow$	$[ROOT, John],$	$[gave, the, dog, a, bone],$	A
LEFT-ARC $\Longrightarrow$	$[ROOT],$	$[gave, the, dog, a, bone],$	$A \cup John \leftarrow gave$
RIGHT-ARC $\Longrightarrow$	$[ROOT, gave],$	$[the, dog, a, bone],$	$A \cup ROOT \rightarrow gave$
SHIFT $\Longrightarrow$	$[ROOT, gave, the],$	$[dog, a, bone],$	A
LEFT-ARC $\Longrightarrow$	$[ROOT, gave],$	$[dog, a, bone],$	$A \cup the \leftarrow dog$
RIGHT-ARC $\Longrightarrow$	$[ROOT, gave, dog],$	$[a, bone],$	$A \cup gave \rightarrow dog$
SHIFT $\Longrightarrow$	$[ROOT, gave, dog, a],$	$[bone],$	A
LEFT-ARC $\Longrightarrow$	$[ROOT, gave, dog],$	$[bone],$	$A \cup a \leftarrow bone$
REDUCE $\Longrightarrow$	$[ROOT, gave],$	$[bone]$	A
RIGHT-ARC $\Longrightarrow$	$[ROOT, gave, bone],$	$[],$	$A \cup gave \rightarrow bone$

Figure 2.3: Trace of the arc-eager parser for the sentence "John gave the dog a bone", whose dependency structure can be seen in Figure 2.2.

already and can still be involved in future arcs, and a set of arcs A containing all dependencies (i, l, j) between tokens i and j with dependency label l that have been added so far. Typical transitions to go from one configuration to the next are *shift*, which moves the item in the beginning of the buffer onto the stack, *reduce*, which removes the item on top of the stack, *left-arc*, which adds an arc from the word in the beginning of the buffer to the word on top of the stack, and *right-arc*, which adds an arc from the word on top of the stack to the word in the beginning of the buffer.

An example of a sequence of transitions of a well-known parser called the arc-eager parser that turn the sentence "John gave the dog a bone" from Figure 2.2 into a dependency tree can be seen in Figure 2.3.

There are two stages to transition-based dependency parsing:

- Learning: You need training data in the form of sentences, potentially annotated with extra information such as part-of-speech tags, with their corresponding dependency tree. The goal of the learning problem is to train a model that predicts the optimal transition for each configuration, where a configuration is a partial analysis of the sentence. The input to the classifier has to be a sequence of configurations with the optimal transition to apply at each of them. Since there is an infinite number of possible configurations, we define a function f that transforms a configuration into a feature vector representing certain properties of the configuration. Let C be the set of all possible configurations, T the set of all possible transitions and Y the set of all feature vectors. Then we define f as:

$$f(c) : C \rightarrow Y$$

  Our classifier g will be of the form:

$$g(y) : Y \rightarrow T$$

Now the question is how to turn a training example into a sequence of transitions, needed as input to the classifier. For a given configuration there is often more than one legal transition. Therefore we need a mechanism, called oracle, to decide what transition to apply next. During training this is an algorithm that looks at the items on the stack and in the buffer and the dependency tree of the sentence and determines which transition is optimal.

- Parsing: This problem consists of finding the dependency tree for a given input sentence. Our oracle for predicting the next transition during parsing is the model g that we trained during the learning stage. A sentence's parse is finished when a final configuration is reached, which is often defined as an empty buffer. Note that for many transition systems it is possible that the dependency graph found during parsing is not connected and that not all words in a sentence are assigned a head. One extreme such case is when the only transition performed is the *shift* transition.

2.2 Neural Networks

Natural language processing tasks that involve training a model and then using it to make predictions used to mainly use linear models that were given large and sparse feature vectors as input. In recent years a new trend emerged of using non-linear neural networks with a more dense feature representation. The non-linearity of the model allows the machine learning framework to come up with useful feature combinations, meaning they do not have to be explicitly represented in the feature vector. In this section I will introduce two main types of neural networks, feed-forward neural networks that often can directly replace previous linear models, and recurrent neural networks, that are especially well suited for modelling sequences. I will discuss the possible feature representations that can be used as input to these models and how the neural networks are trained and then used to make predictions.

2.2.1 Feature Embeddings

The standard way to represent features in NLP tasks used to be in the form of large and sparse indicator vectors, where each dimension corresponds to a specific feature and is set to 1 if it is present and set to 0 if it is not. In the context of dependency parsing, such a feature could be "the word on top of the stack is a noun". Recently and in the context of neural networks, a different way of representing features has become popular. Instead of indicator vectors where each dimension corresponds to a specific feature, embedding vectors are used that are of a fixed dimension and represent core features such as words or part-of-speech tags. The vector representation for each feature is then learnt alongside the other model parameters during the training stage of

the neural network. An advantage of this way of encoding features is that similar features often have similar embeddings. This mechanism is based on context, meaning that words that appear in similar contexts, such as maybe ”son” and ”daughter”, get assigned similar embedding vectors during the training stage and if we only have very limited training examples containing the word ”daughter” but many more containing the word ”son”, we could still expect to perform well on test data with many occurrences of ”daughter”. It is often the case that multiple of these embedding vectors are concatenated to form the input into the neural network.

2.2.2 Feed-Forward Neural Networks

Neural networks are, as the name suggests, inspired by the connections between neurons in our brain. Each of the neurons receives inputs that are multiplied with a weight, summed up and then passed to a nonlinear function, whose result is the neuron's output and serves as input into other neurons. A neural network is split into layers: an input layer, an unlimited number of hidden layers and an output layer. Each neuron's output can feed into an arbitrary number of neurons in the next layer. If it holds that every neuron's output is fed as input to every neuron in the following layer, it is called a fully-connected network. Figure 2.4 is an example of a fully-connected neural network with one hidden layer that is a simplified model of a network that might be used for transition-based dependency parsing. Let us assume we have a transition system with three transitions: *left-arc*, *right-arc* and *shift*. The feature vector that serves as input into the neural network we are using as oracle to predict the next parsing step is a concatenation of the embedding vector of the word on top of the stack, which we assume is $\sigma[0] = the$, and the first word in the buffer, $\beta[0] = dog$. We then have $x = emb(the) \circ emb(dog)$, where *emb* is the function that returns the embedding vector of a feature. In our simplified example the dimension of this embedding vector is 2. For a realistic example, an embedding vector for words might have a dimension of around 100. We now give x as input to the network, where it is fed to the hidden layer at which point a non-linearity is applied to a weighted sum of the inputs. The output of the network is 3-dimensional, where each dimension of the output vector corresponds to a score for one of the transitions. For this example we assume that there is a gold arc from ”dog” to ”the”, in which case the *left-arc* transition should be assigned the highest score.

Let us now formally define what a neural network is. First we look at the simplest neural network, the perceptron, which only consists of an input and an output layer. This model is simply a weighted sum (with W as weight matrix) of its inputs to which we add a scalar bias value (b):

$$P(x) = Wx + b$$

If we add hidden layers whose input is first passed to a non-linear *activation function*, this becomes a Multi-Layer Perceptron. The formula for a 1-layer MLP is as follows,

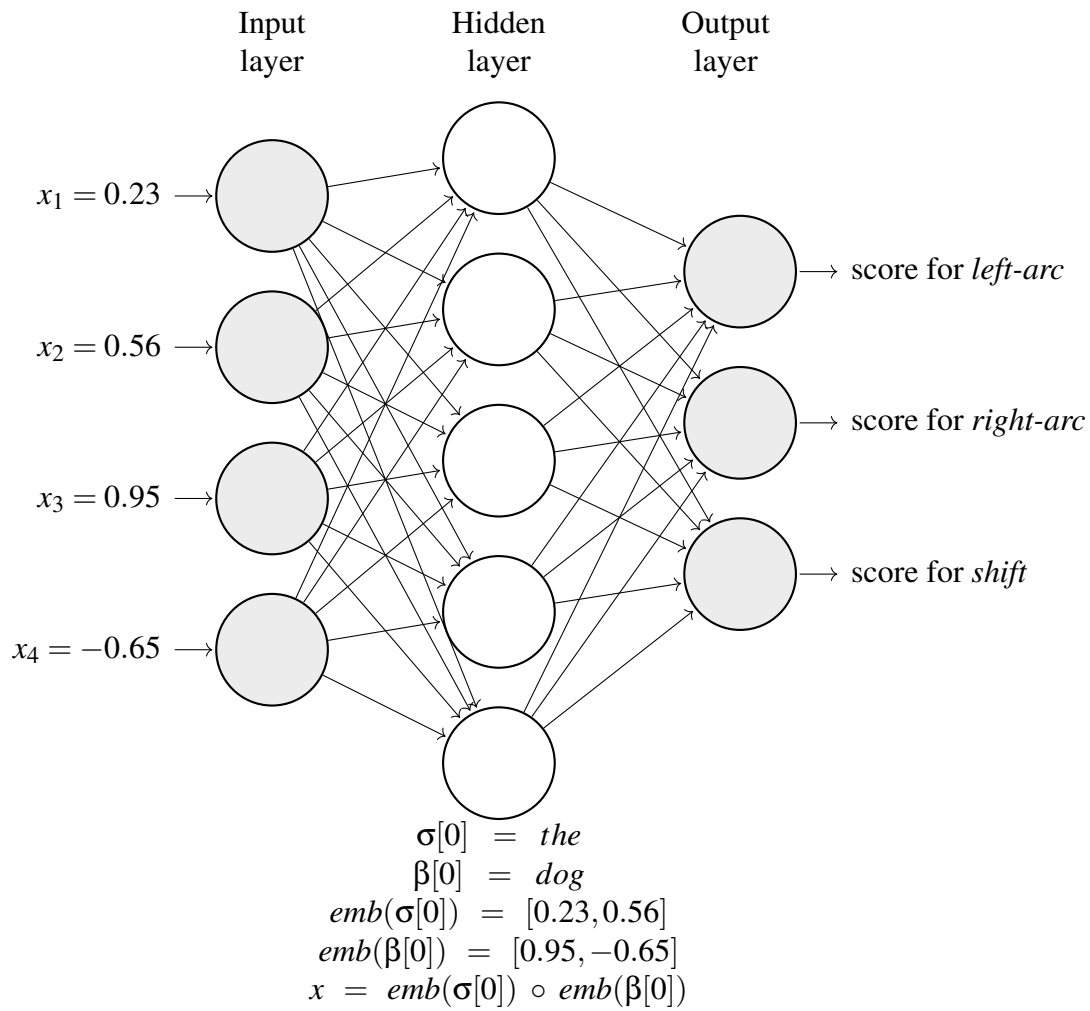

Figure 2.4: An example of a feed-forward neural network with one hidden layer that takes as input a concatenation of the embedding of the word on top of the stack and the word in the beginning of the buffer and outputs the scores for each possible transition.

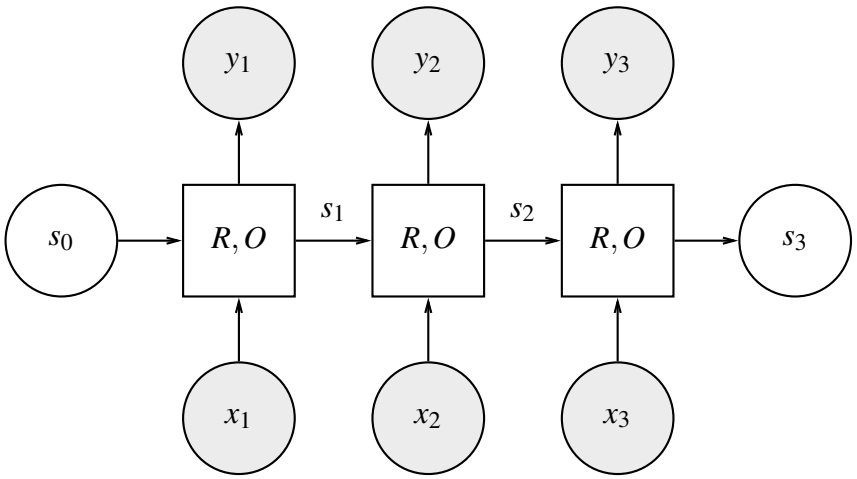

Figure 2.5: Example of an RNN with input sequence x_1, x_2, x_3, state vectors s_0, s_1, s_2, s_3 and output vectors y_1, y_2, y_3.

where g is the activation function, W^1 and b^1 the weight matrix and bias term for the linear transformation of the input x and W^2 and b^2 the weight matrix and bias term for the linear combination of the output of the hidden layer:

$$MLP_1(x) = W^2 g(W^1 x + b^1) + b^2$$

The non-linear activation function is necessary if we want the network to be able to model complex, non-linear functions since applying an arbitrarily long sequence of linear transformations to some input, will just remain a linear combination of that input.

2.2.3 Recurrent Neural Networks

Recurrent neural networks (RNNs) are suitable for modelling sequences, for example of letters or words, making them especially suited for many NLP tasks. While we can use feed-forward networks to represent sequences of features, that involves concatenation or addition of individual feature vectors which both limit the suitability of those models to many tasks. If we use concatenations of feature vectors we face the problem that the input vector always needs to have the same size, so we have to define a fixed number of features we want to concatenate and therefore cannot represent whole sentences of different length. Adding the feature vectors loses structural information and we would get the same feature vector for ”I love peace and hate war” and ”I hate peace and love war”. RNNs fix these problems by transforming an arbitrary number of feature embeddings into a fixed size feature vector, combining information from all of them while taking structural information into account.

As we can see in Figure 2.5, a recurrent neural network transforms a sequence of input vectors $x_1, x_2, ..., x_n$ into a sequence of output vectors $y_1, y_2, ..., y_n$ while keeping track

of the network state with state vectors $s_0, s_1, ..., s_n$, where the initial state vector s_0 has to be given as input to the system. Every output y_i is a function of the current state s_i, which depends on the previous state s_{i-1} and the current input x_i. Since the previous state depends on the previous input, we can apply this principle recursively to see that output y_i depends on all previous inputs $x_1, x_2, ..., x_i$. We define a function O that takes the current state as input and produces the output and a function R of the previous state and the current input, that outputs the current state. Formally, the following holds:

$$s_i = R(s_{i-1}, x_i)$$
$$y_i = O(s_i)$$

In the following I will discuss two popular forms of recurrent neural networks.

2.2.3.1 Simple Recurrent Neural Networks

In this section I will discuss the simplest form of recurrent neural networks. The output vector y_i is simply set to the state vector s_i, which is a linear combination of the input x_i and the previous state s_{i-1} that is then passed to a non-linear activation function g. Formally we have:

$$s_i = g(W^x x_i + W^s s_{i-1} + b)$$
$$y_i = s_i$$

2.2.3.2 Long Short-Term Memory Neural Networks

While simple RNNs have been shown to perform well for a number of applications, such as language modelling and sequence tagging, they are not very good at capturing long-range dependencies. To alleviate this, a different recurrent neural network architecture has been introduced in (Hochreiter and Schmidhuber, 1997) that has a powerful mechanism for deciding what information to keep and forget, which makes it capable of "remembering" information over long periods. The main idea behind these long short-term memory networks is that the state component now keeps track of a "memory cell" c_j which uses so-called "gates" to decide what fraction of the new input information to remember and what fraction of the previous state's memory cell to forget. There is also an output gate that controls what content of c_j should be included in output y_j. The network is composed of following elements:

- A current state s_j that consists of a memory cell c_j and an output, or state, component h_j (it holds that the output $y_j = h_j$)
- An input gate i, controlling how much information of the current input x_j and the previous state h_j should be let into the memory cell c_j. It is a linear combination of the input and the previous state where each component is then passed to a sigmoid function that ensures it takes on a value between 0 and 1 and in most cases

a value close to either 0 or 1. The higher the value, the more of that particular information will be kept. This results in a vector $i \in [0,1]^n$. Mathematically it is defined as:

$$i = \sigma(W^{xi}x_j + W^{hi}h_{j-1})$$

- A forget gate f, controlling how much information of the previous memory c_{j-1} should be forgotten and not included in c_j:

$$f = \sigma(W^{xf}x_j + W^{hf}h_{j-1})$$

- An output gate o that that decides how much of the memory cell c_j's content should be passed as output y_j:

$$o = \sigma(W^{xo}x_j + W^{ho}h_{j-1})$$

- An update function g that proposes a possible update to the memory cell c_j. It is defined as a linear combination of the input vector x_j and the previous state h_{j-1} that is then passed to a non-linear activation function:

$$g = tanh(W^{xg}x_j + W^{hg}h_{j-1})$$

- An update mechanism for the memory cell c_j that consists of applying the forget gate to the previous content of the memory cell, c_{j-1}, and applying the input gate to the proposed update g. Formally we define this in the following way, where $\odot$ stands for component-wise multiplication:

$$c_j = c_{j-1} \odot f + g \odot i$$

- The output h_j (that is equal to y_j) that is defined as the output gate applied to the current content of the memory cell c_j passed through a non-linear function:

$$h_j = tanh(c_j) \odot o$$

LSTMs model an element in a sequence in terms of the elements before it and the element itself. If we are interested in representing an element's full context, meaning the elements before and the elements after it, we should use a BiLSTM representation. Such a bidirectional LSTM is a combination of a LSTM of the sequence and a LSTM of the sequence in reverse order.

2.2.4 Training a Neural Network

In this section I will discuss supervised training of neural networks, where supervised means that we need training data annotated with the desired outcome. Training a neural network involves tuning the parameters (e.g. the weight matrices and bias terms) so that the network exhibits good performance on the task you want it to perform (e.g. part-of-speech tagging or translating text). The main concept behind neural network training is trying to minimise a defined loss function. This is done by computing the gradients of the error function with respect to the parameters of the network and then updating the parameters in the direction of this gradient.

I will introduce two algorithms for accomplishing this, first the basic and common *stochastic gradient decent* and then a variation called *Adam* (Kingma and Ba, 2014) that is used by Kiperwasser and Goldberg (2016) to train their transition-based dependency parser.

Stochastic gradient descent: Let us define the parameters of our neural network as θ. We also need to define a loss function L. The training data consists of inputs $x_1, ..., x_n$ and corresponding outputs $y_1, ..., y_n$. The objective of the gradient descent algorithm is now to minimise the loss function over the training data by adjusting θ. This is an iterative process where the neural network with the current parameters θ is used to predict output $\hat{y}_i$ when given input x_i. Then a loss value is computed that reflects how "wrong" $\hat{y}_i$ is with respect to the desired output y_i.The gradients of the loss value with respect to the parameters are calculated, and the parameters updated in the direction of this gradient. The magnitude of the update depends on a variable called the learning rate, which is usually decided on through trying out different values. There is a variation of this called "mini-batch", in which the error is not evaluated for every individual training example, but over a collection of training examples which makes the parameter updates more robust to noisy training examples.

Adam: This algorithm also takes estimates of the first and second moment of the gradient into account when updating the parameters, which integrates information from previous updates and can lead to a faster convergence. Another advantage it has over the basic stochastic gradient descent algorithm is that it adaptively chooses the learning rate which takes away the necessity of having to empirically find one that works well for a task.

2.3 Contributions

In this section I will summarise my contributions to this project and explain what implementing the BiLSTM feature model for the four transition parsers entailed:

- After researching ways to improve the attachment accuracies of the four transition-based dependency parsers I implemented during the first part of my project, I

decided to explore how extending the binary feature model according to (Zhang and Nivre, 2011) would affect performance. I implemented this extended feature model, more closely described in Section 4.1, and used it to train Covington's parser with reduce transitions on German, Hungarian and Dutch. I then used the only slightly improved attachment accuracies resulting from parsing the test sets of the respective languages and the significantly increased training and parsing times to argue about how this might not be the most effective way of improving the accuracy of my parsers.

- Reading up on several papers using neural networks instead of linear classifiers to predict transitions and comparing their reported performances made me decide to implement a BiLSTM feature representation that is given as input to a multi-layer perceptron and used as an oracle when making parsing decisions, as described in (Kiperwasser and Goldberg, 2016). They also made the Python code of their system available, which I used to guide my own implementation. Following steps had to be carried out:
  - Learn how to use the neural network toolkit DyNet to model the new oracle I decided to implement, which was made more difficult by the fact that documentation was only published mid-December 2016.
  - Figure out how the system by Kiperwasser and Goldberg would have to be adapted to use it for the four transition-based parsers I have implemented last year and then integrate the new neural network feature model with those parsers.
  - For each transition system, write a function that given the current configuration returns the set of transitions leading to a gold dependency tree. This function is then used in the loss function to decide whether the predicted transition would incur a loss or not.
- I identified and implemented several adjustments to this model that I believed would influence the attachment accuracies and evaluated the parsers on these new models. I then also reported results on models with following adjustments:
  - Instead of only taking word forms and POS tags into account, I also implemented a version that included word lemmas and additional grammatical word features such as tense, person, verb form and case.
  - I varied the number of tokens on the stack whose BiLSTM representation is given as input to the oracle and evaluated how that influenced attachment accuracies.
  - For the two non-projective parsers that keep track of a temporary list of processed items that can still be involved in dependency arcs and are added back to the stack at the next shift transition, I implemented a variation of the model that also looked at the first few elements on this list when making parsing decisions.
- The paper by Kiperwasser and Goldberg (2016) reported results of using their parser on the Stanford Dependency conversion of the Penn Treebank data for

English. Since I want to compare my parser's performance with their parser I needed to first ensure I could use it correctly by reproducing their results. This entailed the following:

  - Learn how to use the Stanford POS tagger to perform "10-way jackknifing".
  - Use the Stanford parser to convert the resulting data files into the Stanford Dependency format.
  - Carefully go over their paper to find out what exact parameters they used in the model whose parsing accuracies they reported and train their parser on the files I created by converting the English Penn Treebank with predicted POS tags into the Stanford Dependency format.

- I evaluated the performance of each of the four parsers I implemented on the complete training and test sets from the Universal Dependencies treebank on German, Dutch, Hungarian, Portuguese and Danish and on part of the training set (40,000 of 68,525 sentences) and the complete test set for Czech. Then I used these results to argue about how using a BiLSTM feature model and a non-linear neural network to predict transitions could improve performance compared to a binary indicator feature model using a linear classifier as oracle.

Chapter 3

Previous Work Carried Out

The goal of last year's work was to compare performance of non-projective and projective parsers on a variety of languages. Therefore I implemented a well-known projective transition-based dependency parser, the arc-eager parser, and Covington's non-projective parser, which can build every possible dependency tree with an arbitrary number of crossing arcs. After observing that for the arc-eager parser more than one transition sequence could lead to the same dependency tree, I developed a novel projective transition system, the attach-complete parser. During initial experiments, Covington's parser seemed to perform much worse than the arc-eager one, even on languages with a very high number of crossing arcs, and to take a long time to train. Therefore I developed a transition system that behaves like the arc-eager one on projective sentences, without losing the ability to build every possible non-projective dependency tree. In the following I will briefly describe each transition system, look at my results and show how last year's work leads into the second part of my project.

3.1 Arc-Eager Parser

The first transition system I implemented last year is the arc-eager transition system by Nivre, which only builds projective dependency trees. The arc-eager parser works by keeping track of a buffer, which initially contains all input tokens, and a stack. There are transitions to push tokens from the buffer onto the stack, to add arcs between the token on top of the stack and in the beginning of the buffer, and to pop items off the stack.

3.2 Attach-Complete Parser

The arc-eager dependency parser exhibits spurious ambiguity, meaning more than one transition sequence can lead to the same dependency tree. I developed the attach-complete parser as a way of removing spurious ambiguity, by making use of the fact that dependency grammars can be transformed into context-free grammars, where a

sentence corresponds to exactly one parse tree. A transition system can then be developed where each transition corresponds to a production rule of the CFG. The transition system works by adding arcs with the ATTACH operations and using the COMPLETE operations as soon as a token that already has a head has no more dependents in the buffer. To accomplish this, we add two copies of each word to the buffer, a left copy and a right copy. The ATTACH transitions delete one copy of the dependent token and the COMPLETE transitions delete the second one. The left copy of the word is used for adding arcs to and from words to the left of it and the right copy for adding arcs to and from words to its right.

3.3 Covington's Parser

Covington's non-projective algorithm works by going through the tokens in a sentence and keeping track of the ones that have been processed so far and the ones that do not have a head yet. When processing the next token t in the sentence, it first looks at the list of tokens without head and checks if any of them are dependents of t and then checks if any of the tokens that have already been processed are t's head. The transition system, with configuration $(\sigma, \lambda, \beta, A)$, works by adding arcs between the top of the stack and the first element in the buffer, moving elements from the buffer to the stack, and pushing tokens from the top of the stack onto the list λ if there are no arcs to add between them and the first element of the buffer. When the next token in the sentence is processed, all elements of λ are added to the stack again to allow for all possible arcs.

3.4 Covington's Parser with Reduce Transitions

The development of this transition system was motivated by my observations that during some initial experiments, Covington's parser took much longer to train and had a significantly lower number of correctly added projective dependency arcs than the arc-eager parser. I theorised that allowing Covington's parser to behave like the arc-eager one on projective sentences and to only use the transitions that enable it to add crossing arcs when necessary, would improve both its training time and its attachment accuracy. To do this I have implemented a version of Covington's parser that allows for REDUCE operations. Two versions of the LEFT-ARC transition are needed, one that allows for further non-projective links to and from the child in the dependency relation, LEFT-ARC-KEEP, and one that removes it from consideration, LEFT-ARC-REDUCE.

3.5 Results

During my experiments on the German, Portuguese, Dutch, Danish, Hungarian and Czech Universal Dependencies data sets, Covington's parser with reduce transitions had the best average attachment scores, while the arc-eager parser had the best average attachment scores on projective arcs. Covington's parser on the other hand had relatively poor performance and the attach-complete parser led to the best exact match scores.

My best results were however about 5% lower than results shown in (Tiedemann, 2015) on the same data. In the next section I will talk about what feature model I used and how using a different one might lead to vastly improved attachment scores.

3.6 Feature Model

The feature model I used to describe the current configuration of the transition-based parsers consists of a large vector of indicator features with value 0 if they do not occur in the current configuration and value 1 if they do. Most features consist of an attribute of a word at a specific position in the configuration. We can for example say that one of the features we look at is the part-of-speech of the word on top of the stack. When specifying the position of the word for the *(position, attribute)* pair that represents a feature, we can for example specify "left-most dependent of the word in the beginning of the buffer" or "head of the word on top of the stack". It is of course possible that such a word does not exist (yet), in which case no feature will be added.

It is infeasible to add features for every attribute of every word in the configuration since our feature vector will get extremely big, leading to very long training times. Therefore the features I have extracted last year for my implementation of the parsers are based on suggestions from Nivre (2008). Table 3.1 shows the features extracted for each configuration. The first column shows the words we are looking at when extracting features, which are the first four words in the buffer, the left-most dependent of the word in the beginning of the buffer of the current configuration, the two top words on the stack, the head of the word on top of the stack if it has one already, the left- and right-most dependent of the word on top of the stack, the first and last word in list λ if it is part of the configuration and the number of words between the word in the beginning of the buffer and the word on top of the stack. There are several features we can extract for each of the words we are looking at. In Table 3.1, *Form* refers to the word itself, *Lemma* to the stem of the word, *UPOS* to the universal part-of-speech tag, *XPOS* to certain language specific POS tags, *Features* are additional grammatical and structural properties of the word, such as tense and case and *Dependency* describes the kind of dependency label the dependency arc leading to the word has, if one has been assigned already.

	Form	Lemma	UPOS	XPOS	Features	Dependency
β[0]	X	X	X	X	X	X
β[1]	X		X			
β[2]			X			
β[3]			X			
ld(β[0])						X
σ[0]	X	X	X	X	X	X
σ[1]			X			
hd(σ[0])	X					
ld(σ[0])						X
rd(σ[0])						X
λ[0]			X			
λ[n]			X			

Table 3.1: Features for training the classifier; β = buffer; ld = left-most dependent; rd = right-most dependent; σ = stack; hd = head; λ = list for Covington's parsers

For the *(position of word, attribute)* pair *(top of the stack, universal part-of-speech)*, you could have following features associated with it:

s0_UPOS_VERB
s0_UPOS_NOUN
s0_UPOS_DET
s0_UPOS_ADP
.
.
.

Of the positions in the feature vector corresponding to those features only one can be set to 1, whereas all others have to be 0. This leads to a very large and sparse feature vector.

During the training stage of the parser we associate every feature we encounter with a dimension in the feature vector. I have conducted experiments to evaluate how large this feature vector gets in practice. Let us call the set of all possible features F. The size of F can get very large since the features I implemented include the form of the word on top of the stack and in the beginning of the buffer for every configuration. While training the arc-eager parser on Hungarian the maximum size of F was 16,271 and on Dutch it was 36,431.

Chapter 4

Motivation for Using a LSTM Feature Model

In this section I will explore two different directions that can be taken in improving last year's feature model. I will conduct experiments showing how extending the binary indicator feature model I was using in a number of ways leads to significantly worsened training and parsing time without large improvements to performance. Then I will show motivation for using LSTMs for feature representation and discuss evidence from the dependency parsing literature that suggests that non-linear neural networks are the way forward.

4.1 Extending the Indicator Feature Model

One way to improve performance of the transition-based parsers I have implemented would be to extend this feature model, e.g. by adding third-order features or using conjunctions of features, as is discussed in (Zhang and Nivre, 2011). An example for a third order feature is the head of the head of the word on top of the stack. Since I used a linear classifier to decide what transition to apply it would improve performance to use feature combinations, such as a feature for "the word in the beginning of the buffer is a verb and the word on top of the stack a noun". It is, however, not easy to decide what feature combinations to use and feature engineering like this often takes an extensive amount of time. Having a very large, sparse feature vector also negatively impacts training and parsing time. In the remainder of this section I will explore how extending the indicator feature model I was using last year can improve performance of the parsers and look at how much longer it takes to train the model and parse sentences.

The extended feature model I have implemented is heavily based on the one described in (Zhang and Nivre, 2011). Last year, I have only included features consisting of a single position in the configuration (e.g. beginning of buffer) and a single characteristic of that token (e.g. part of speech tag, lemma, part of speech tag of its head, dependency relation to its left-most dependent). Now I will look at additional features

as shown in Table 4.1. Let us assume t_r is a token in the current configuration, with properties $p_{r,1}, p_{r,2}, ..., p_{r,i}, ..., p_{r,n}$. We denote a feature that is a combination of tokens and properties in the following way:

$$t_r.p_{r,i}.p_{r,j}... \circ t_s.p_{s,k}.p_{s,l}... \circ ...$$

The newly added features fall into following categories:

- **Multiple properties**: It is beneficial to look at combinations of properties for a given word, since there are words that are spelt the same but may mean different things depending on their part of speech tag (e.g. fly as a verb or a noun). Therefore we include features such as $\sigma[0].Form.UPOS$, which represents the form and universal part-of-speech tag of the word on top of the stack.

- **Multiple tokens**: Since a linear classifier is used as the scoring function that takes a feature vector representing the current configuration as input and outputs which transition to apply, we cannot automatically learn combinations of features and have to include them explicitly as separate features. We include for example a feature representing the form of the token on top of the stack and the form of the token in the beginning of the buffer, which we denote as $\sigma[0].Form \circ \beta[0].Form$.

- **Third-order features**: My feature model from last year already includes second-order features, such as the head of the word on top of the stack. The paper (Zhang and Nivre, 2011) showed that parsing results could be improved by also looking at third-order features. The only third-order token I included in my extended feature model was the head of the head of the word on top of the stack.

- **Valency**: I also look at the number of left and right dependents that have already been added to a token, combined with the form or part-of-speech tag of a token. A reason why we expect this to increase parsing performance is that specific words often have a relatively fixed number of left and right dependents. The verb "give" for example is likely to have a left dependent that is its subject, and two right dependents, its indirect and direct object.

- **Distance**: The extended feature model also includes features representing the distance between the word on top of the stack and the word in the beginning of the buffer, where we define distance as the number of words in the sentence between the two tokens.

Training Covington's parser with reduce transition using the extended feature model on German, Hungarian and Dutch, showed that on average we can expect a performance increase compared to using the simple feature model. As Table 4.2 shows, only the UAS for German is higher using last year's model and in all other cases both the UAS and LAS are higher using the new feature model. However, we can see that the improvements are quite small and remain lower than 1%. Part of the reason for this might be that there is not enough training data to meaningfully learn from features that are composed from combining multiple word forms. It also makes sense that we cannot meaningfully argue about the best transition to apply while only looking at such a limited number of tokens in the configuration. One possible improvement therefore

One-word features, multiple properties
$\sigma[0]$.Form.UPOS
$\beta[0]$.Form.UPOS
$\beta[1]$.Form.UPOS
Two-word features
$\sigma[0]$.Form.UPOS $\circ$ $\beta[0]$.Form.UPOS
$\sigma[0]$.Form $\circ$ $\beta[0]$.Form
$\sigma[0]$.UPOS $\circ$ $\beta[0]$.UPOS
$\sigma[0]$.Form.UPOS $\circ$ $\beta[0]$.Form
$\sigma[0]$.Form.UPOS $\circ$ $\beta[0]$.UPOS
$\sigma[0]$.Form $\circ$ $\beta[0]$.Form.UPOS
$\sigma[0]$.UPOS $\circ$ $\beta[0]$.Form.UPOS
$\sigma[0]$.UPOS $\circ$ $ld(\sigma[0])$.Dependency
$\sigma[0]$.UPOS $\circ$ $rd(\sigma[0])$.Dependency
$\beta[0]$.UPOS $\circ$ $\beta[1]$.UPOS
Three-word features
$\sigma[0]$.UPOS $\circ$ $\beta[0]$.UPOS $\circ$ $hd(\sigma[0])$.UPOS
$\sigma[0]$.UPOS $\circ$ $\beta[0]$.UPOS $\circ$ $ld(\sigma[0])$.UPOS
$\sigma[0]$.UPOS $\circ$ $\beta[0]$.UPOS $\circ$ $rd(\sigma[0])$.UPOS
$\sigma[0]$.UPOS $\circ$ $\beta[0]$.UPOS $\circ$ $\beta[1]$.UPOS
$\beta[0]$.UPOS $\circ$ $\beta[1]$.UPOS $\circ$ $\beta[2]$.UPOS
Valency features
$\sigma[0].RightValency(\sigma[0])$.Form
$\sigma[0].RightValency(\sigma[0])$.UPOS
$\sigma[0].LeftValency(\sigma[0])$.Form
$\sigma[0].LeftValency(\sigma[0])$.UPOS
Third-order features
$hd(hd(\sigma[0]))$.FORM
$hd(hd(\sigma[0]))$.Dependency
$hd(hd(\sigma[0]))$.UPOS
Distance features
$\sigma[0]$.UPOS $\circ$ $\beta[0]$.UPOS $\circ$ $Distance(\sigma[0], \beta[0])$
$\sigma[0]$.Form $\circ$ $\beta[0]$.Form $\circ$ $Distance(\sigma[0], \beta[0])$

Table 4.1: Additional features used in the extended feature model, where β = buffer; ld = left-most dependent; rd = right-most dependent; σ = stack; hd = head.

	UAS (LAS)	
Language	Simple feature model	Extended feature model
German	**79.90** (74.69)	79.73 (**74.88**)
Hungarian	78.09 (74.86)	**78.72** (**75.05**)
Dutch	73.89 (71.01)	**74.66** (**71.89**)

Table 4.2: The attachment scores of Covington's parser with reduce transitions using the simple feature model I implemented last year, compared to the same parser using the extended feature model described in Table 4.1.

	Training (Parsing) Time	
Language	Simple feature model	Extended feature model
German	21,979 (1,420)	62,470 (2,641)
Hungarian	120 (30)	334 (55)
Dutch	10,706 (353)	24.465 (1,216)

Table 4.3: The training and parsing time using the simple feature representation I implemented last year, compared to using the extended feature representation described in this section.

would be to add more features using different combinations of tokens, for example including $\sigma[1]$ or $\beta[3]$.

Looking at Table 4.3 however shows that the additional features already lead to a large increase in training and parsing times, with the training time always more than double that of the previous model. This lets me believe that blindly adding more and more features, which will probably only lead to small increases in performance, is not the best way to improve my parsers. More work could be done on carefully selecting feature combinations that lead to the best results, but this form of feature engineering takes a long time and is not very robust across different languages.

4.2 LSTM Feature Representation

Another way to improve the feature model for transition-based dependency parsers, that has become increasingly popular over recent years and months, is to use neural networks to automatically learn useful feature combinations. Several successful applications of RNNs to natural language processing tasks involving sequences of words or characters have been reported. To show the benefits of using neural networks, I will first mention several successful applications across the field of natural language processing and then focus on how they have been shown to improve results for transition-based dependency parsers. I will discuss several papers dealing with different ways of applying neural network approaches to dependency parsing and then explain that using a specific type of neural network, Long Short-Term Memory neural networks, seems to be especially suitable when dealing with sequences of words. The section will conclude by giving my motivation for choosing the model introduced in (Kiperwasser and Goldberg, 2016) over other neural network models for dependency parsing.

4.2.1 Neural Networks for a Variety of NLP Tasks

Using simple RNNs for **language modelling** was discussed in (Mikolov et al., 2010). They observed that tasks involving the prediction of words given the context of previous words could benefit from recurrent neural network frameworks that have the

ability to capture an arbitrarily large context. Previous models often only looked at fixed size context windows, which were not able to capture long-range dependencies and connections between words. Mikilov et al. evaluated their RNN language model on speech recognition tasks and could show reduced error rates compared to several previous models, even though they used significantly less training data due to the time complexity of training recurrent neural networks.

Successful applications of LSTMs have been shown for **handwriting generation and synthesis** (Graves, 2013). The resulting generation model managed to learn letters, pen strokes and several short words, and even the words it invented often could pass as English words. For the handwriting synthesis task, which means transforming a text into handwriting, Graves showed that using his LSTM model, samples could be synthesised that were indistinguishable from real handwriting.

Promising results could also be shown by applying LSTM architecture to a **machine translation** task in which a LSTM was used to map a sentence in the input language to a vector and this vector was then used as input into a second LSTM, that estimates the probability of target sequences, conditioned on the input sequence (Sutskever et al., 2014). They showed that even with limited vocabulary and little work done to optimise the system, they could outperform a standard statistical machine translation system.

A similar method as the one in the above mentioned machine learning task was applied to **image caption generation** (Vinyals et al., 2015). Generating a caption for a given image is a challenging and involved task, for which it is first necessary to recognise the objects in the image and understand how they interact and relate, and then a properly phrased description has to be produced. Previously, these tasks have often used solutions to each of these individual parts and then combined them, but Vinyals et al. propose a single joint model that treats image captioning like a translation task where an image has to be translated into its description. Again, a neural network (in this case a convolutional neural network) is used to first transform the input image into a vector and then this vector is used as input into a LSTM that finds the most probable sequence of words, given the image. The model manages to produce reasonable captions in many cases and could be shown to match state-of-the-art results on a recent data set.

4.2.2 Neural Networks for Dependency Parsing

Neural networks were first used for transition-based dependency parsing in (Chen and Manning, 2014). They showed how instead of using a very large and sparse indicator feature vector, it was possible to use dense feature representations consisting of word, part-of-speech and dependency label embeddings, that are learnt by a neural network alongside its other parameters. The feed-forward neural network they used is non-linear, which enables them to define a set of core features without having to explicitly

include feature that are combinations of other features. While they still use a relatively large set of core features (e.g. of the form ”part-of-speech tag of the left-most dependent of the word on top of the stack”) it is now left to the machine learning framework to decide on how to combine these features to get optimal parsing accuracies with respect to the training data. Using this model, Chen and Manning manage to show improved results in both speed and accuracy compared to popular transition-based dependency parsers, such as the MaltParser.

The parser by Chen and Manning still only uses a limited view of the parsing state when constructing the feature vector. A recent model by Dyer et al. (2015) on the other hand encodes the complete state of the stack, the buffer, the previously taken actions and the partially built dependency tree as its feature representation. It introduces **stack LSTMs** that support the push and pop operations necessary for an efficient implementation of a constantly changing stack and buffer of tokens. Three such stack LSTMs are used, to represent the state of the stack (including partially built dependency structures), the buffer and the history of parsing actions. A stack pointer is defined that points to the current head of the stack LSTM and is moved to point to the previous item on the stack by every pop operation. A push operation adds a new token to the stack LSTM, which includes a pointer back to the word previously on top of the stack. This creates a LSTM that is structured like a tree and encodes the complete state and history of the configuration, meaning the model can learn to use any property of the configuration to make parsing decisions. The model manages to show improved results compared to (Chen and Manning, 2014).

The paper (Kiperwasser and Goldberg, 2016) suggests using BiLSTMs instead of hand-crafted feature functions. Each word i is represented by x_i, which is a concatenation of two embedding vectors, one for the word form and one for its part of speech. These embedding vectors are learnt alongside the other parameters of the model during the training stage. In order to capture the context around each word i in the sentence, a BiLSTM vector v_i is associated with it that takes the sequence $x_{1:n}$ and i as input. For every configuration of the parser, a small number of such BiLSTM vectors (e.g. from the top two words on the stack and the word in the beginning of the buffer) is used as feature representation and assigned a score using a multi-layer perceptron. This architecture is simpler and it shows results that are slightly higher than those reported in (Dyer et al., 2015), which was my motivation for deciding to implement Kiperwasser and Goldberg's neural network model for transition-based dependency parsing in an effort to improve the accuracies of the transition systems I implemented last year. The advantage of this feature model over the binary indicator feature model is that no explicit feature engineering is necessary since each word vector has access to the whole context of the word, meaning the whole sentence, and also the part-of-speech tag of each word in the sentence. It is up to the neural network to learn what of this information is relevant to the parsing problem. Even though we only look at a very limited number of items on the stack we are still able to get good parsing accuracies, as can be seen in the results chapter.

Chapter 5

Methodology

Alongside the paper on using LSTMs for dependency parsing, Kiperwasser also released the code for the parser they developed. They are using the arc-hybrid transition system, which differs from the arc-eager one in the sense that it only has three transitions: SHIFT, LEFT-ARC and RIGHT-ARC. It only adds a head to a word after all its dependents have been added. I have used their system as guidance when implementing the feature model for my own parsers. Following their example, I used the neural network toolkit DyNet (Neubig et al., 2017) to model the LSTMs.

5.1 Data

The data set I used for evaluating the dependency parsers this year is the same as last year: the the Universal Dependencies treebank, version 1.2, released on November 15, 2015 (Nivre et al., 2015). Each word in a sentence is annotated with its lemma, part-of-speech tags, features detailing additional lexical or grammatical properties, its head and the dependency relation to its head.

	training				testing			
	sentences		arcs		sentences		arcs	
Language	#	% NP	#	% NP	#	% NP	#	% NP
Danish	4,868	22.70	88,979	4.92	322	22.98	5,884	4.47
German	14,118	11.21	269,626	2.42	1,000	22.10	16,609	5.11
Hungarian	1,032	24.81	20,764	5.61	138	24.81	2,725	4.84
Dutch	13,000	30.83	188,882	10.10	386	27.46	5,585	9.67
Portuguese	8,800	18.61	201,845	3.39	288	17.01	5,867	3.22
Czech	68,495	12.49	1,173,282	2.94	10,148	12.82	173,737	2.97

Table 5.1: Number of sentences, number of dependency arcs and the percentage of non-projective sentences and non-projective arcs for both training and testing data in the Universal Dependencies (UD) treebanks for several languages.

Table 5.1 shows properties of the sentences in the UD treebanks for Danish, German, Hungarian, Dutch, Portuguese and Czech. The Dutch data set has with 27.46% an especially high number of non-projective sentences in the test set, which is why we might expect that a dependency parser allowing for crossing dependencies will obtain better results than a solely projective one.

When training the projective parsers, the non-projective training sentences are first turned into projective ones using an algorithm described by Nivre (2005) that lifts non-projective links until the sentence is projective. We recursively call the `Lift` function on the shortest crossed arc until the sentence is projective. If this shortest crossed arc is between i and j then it attaches j to i's head. This procedure is designed to preserve as much of the original sentence's structure as possible.

5.2 DyNet

In this section I will briefly describe how DyNet works and what advantages it has over other neural network toolkits using material from (Neubig et al., 2017).

The main concept behind neural network tools is a computation graph that keeps track of the parameters of the model and performs operations on them. It supports forward computations, meaning that it computes values of the expressions that build up the network according to the current parameters. During training, backwards computations are performed that compute the value of a loss function and then update the network parameters in the direction of the gradient of that loss function. The main difference between DyNet and many other similar toolkits is that it uses dynamic declarations instead of static ones. This means that it uses a separate computation graph for each training example (or collection of training examples in the case of mini-batch training) instead of creating one computation graph in the beginning of the training process and then using it to execute all the computations over the training examples. An advantage of DyNet's implementation is that it supports having networks with different architectures for each training example. This is helpful when modelling RNNs for a varying number of inputs which is necessary if you want to use them to represent sentences.

In the following I will describe the main components of a neural network in the DyNet framework:

- **Parameters** are used to represent the vectors and matrices that have to be tuned during the training process, such as the weight matrices and the bias values.
- **Lookup parameters** refer to sets of vectors corresponding to embeddings, such as POS embeddings. If we have a set P of part-of-speech tags, then a lookup parameter is defined by a matrix of dimension $|P| \times d$ which corresponds to an embedding vector of size d for each of the tags. Both kinds of parameters

are preserved across training examples, even when a new computation graph is created.

- A **model** is created that is defined by a set of parameters and lookup parameters and keeps track of them and their gradients throughout the training process.
- **Expressions** describe the computations that need to be performed by the neural network. The most simple expressions are just representing input values or parameters, but they can be combined to form larger expressions through operations such as concatenation, addition and multiplication, and transformed through activation functions.
- A **trainer** needs to be created that is responsible for performing gradient descent and thereby updating the model parameters.
- **Forward computations** are applied to the network to get the values of the defined expressions given the current parameters.
- **Backwards computations** perform back propagation to compute the gradients of the network. These gradients are then used by the trainer to update the parameters in order to minimise future errors on similar data.

5.3 Evaluation Metrics

I will evaluate the performance of the parsers using following metrics:

- Unlabelled attachment score (UAS): The percentage of words that have been assigned the correct head by the parser.
- Labelled attachment score (LAS): The percentage of words that have been assigned the correct head and the correct dependency label.

5.4 Training the Parsers

In this section I will describe how the neural network will be trained to predict transitions. First, I will give an overview of the main steps taken and then I will give a detailed description of how this was achieved in the DyNet framework.

5.4.1 Conceptual Overview

When training the model we take a sentence $w_0, w_1, ..., w_n$, where w_i has universal part-of-speech tag p_i, and create a word vector for word i, x_i, by concatenating the embedding of w_i and the embedding of p_i :

$$x_i = emb(w_i) \circ emb(p_i)$$

Euclidean distance	
	son
daughter	0.77
man	0.69
house	1.58
to be	2.26
nice	1.19

Table 5.2: Euclidean distances between the embedding vectors for several words after training a neural network model on all the Universal Dependencies training data for German. They have been translated into English for this table.

The advantage of such word and POS embeddings that are learnt during the training stage of the parser alongside the other parameters of the model, is that they are supposed to be able to capture similarities between words that appear in related contexts. I have evaluated this assumption by reducing the word form embedding dimension to 5 (instead of 100, as it is for all my other experiments) and training the arc-eager parser on the complete Universal Dependency training set for German. I then looked at the learnt embedding vector for ”Sohn”, the German word for son, and compared its Euclidean distance to several other vectors. The results are summarised in Table 5.2 and we can see that the distance between ”son” and ”daughter” and between ”son” and ”man” is significantly shorter than to any of the other words. This shows evidence for how the embedding vectors actually manage to represent similar words by similar vectors.

Using this concatenation of the word embedding and the POS embedding as the feature representation for a word in the sentence, I am only taking information about the word form and universal part of speech tag into account. Since I am mainly interested in comparing the performance of my parsers using LSTMs to represent the features of a configuration to last year's results using binary indicator features, I will also consider how my results improve when taking additional information into account. Last year I also included information about the tokens' lemmas, features, and XPOS tags, which are language specific part-of-speech tags.

Therefore I will implement a variation of my BiLSTM feature model where each word vector is a composition of the embeddings of its form, its part-of-speech tag, its lemma and 7 different features. The features I have chosen for this are Case, Gender, Number, Tense, PronType, Person and VerbForm. If any of these features do not apply to the current token, a padding vector is used instead. The new word vector x_i will be of following form, where we let l_i be the lemma of token i and f_i the set of features, where $f_i(CASE)$ denotes the case of token i:

$$x_i = emb(w_i) \circ emb(p_i) \circ emb(l_i) \circ emb(f_i(CASE)) \circ emb(f_i(GENDER)) \circ emb(f_i(NUMBER)) \circ emb(f_i(TENSE)) \circ emb(f_i(PRONTYPE)) \circ emb(f_i(PERSON)) \circ emb(f_i(VERBFORM))$$

The embeddings of the words, part-of-speech tags, lemmas and features are initially set to arbitrary values and then learnt alongside the rest of the parameters of the neural network. For each token we now construct a BiLSTM vector representing its whole context. To do so for x_i we concatenate a forward LSTM $LSTM(x_{0:i})$ with a backwards LSTM $LSTM(x_{n:i})$:

$$BiLSTM(x_i) = LSTM(x_{0:i}) \circ LSTM(x_{n:i})$$

I am in fact using a *2-layer BiLSTM* to represent a token in the sentence. This variation refers to constructing a BiLSTM as described above and then using its output sequence as input into another BiLSTM. So for token x_i in sentence $x = x_0 x_1 ... x_n$ we build its BiLSTM representation in the following way:

$$BiLSTM^1(x_i) = LSTM(x_{0:i}) \circ LSTM(x_{n:i})$$

Let us denote $BiLSTM^1(x_i)$ by $BiLSTM_i^1$ and a sequence of such outputs as $BiLSTM_{0:i}^1$. Then the second layer is formed the following way:

$$BiLSTM^2(BiLSTM_i^1) = LSTM(BiLSTM_{0:i}^1) \circ LSTM(BiLSTM_{n:i}^1)$$

Our feature vector x is then a concatenation of the BiLSTM representation of the first 3 elements on the stack and of the token in the beginning of the buffer:

$$x = BiLSTM^2(\sigma[0]) \circ BiLSTM^2(\sigma[1]) \circ BiLSTM^2(\sigma[2]) \circ BiLSTM^2(\beta[0])$$

The reason why we are only looking at the first element in the buffer is that the remaining sequence of tokens on the buffer is encoded in the BiLSTM representation of the tokens. When word i is in the beginning of the buffer then $\beta = [w_i, w_{i+1}, w_{i+2}, ..., w_n]$. The stack on the other hand might only contain a subset of already processed tokens, meaning we arrive at better results looking at more than one token. In theory I would assume that Covington's parser is an exception to this, since whenever a new token is in the beginning of the buffer, all tokens of the sentence preceding it are pushed onto the stack. Therefore I theorise that Covington's parser's performance should not be impacted as significantly as that of the other parsers if we reduce the number of tokens on the stack that we include in out feature vector. I will further explore this idea during my experiments and summarise what I discover in the results section.

The feature vector is then passed as input to two multi-layer perceptrons with one hidden layer each, which are of following form:

$$MLP(x) = W^{output} \cdot tanh(W^{hidden} \cdot x + b^{hidden}) + b^{output}$$

Let r be the number of dependency relations found in the training data (can be up to 40). Let a be the number of transitions of the parser that add dependency arcs ($a = 3$ for Covington's parser with reduce transitions and $a = 2$ for the other three parsers). Let t be the number of transitions that do not add any dependency arcs. Then one of the MLPs, $MLP_1(x)$, has an output dimension of $a+t$ and predicts which transition to apply, without considering arc labels. The other MLP, $MLP_2(x)$, has an output dimension of $a*r+t$ and assigns scores to all *(transition, relation)* pairs.

We assign scores to each possible transition, where the score is the sum of the respective entries of $MLP_1(x)$ and $MLP_2(x)$. We now try to maximise the difference in score between the correct transition and the highest-scoring incorrect transition. To do that, we define the loss of the model in the following way:

$$loss = max(0, 1 - highestValidScore + highestInvalidScore)$$

So as long as the highest valid score is at least 1 more than the highest invalid score, the loss is zero.

We then apply the highest scoring gold transition to the current configuration. The losses are stored in a list and once the end of a configuration has been reached, we count how many have been accumulated. A version of minibatch training is used, meaning the parameters are only updated after a certain number of losses have been encountered. If the number of losses at the end of a training example is less than 50, we go to the next sentence. Otherwise we use the back-propagation algorithm to calculate the gradients of the network. The parameters of the neural network are then optimised in the direction of the gradients using DyNet's `AdamTrainer`.

Each parser is trained for 30 iterations, each resulting in a model. The training sentences are shuffled randomly for each new iteration. These models are then evaluated on the development set and the one with the highest UAS is chosen.

5.4.2 Concrete Implementation

When training the neural network that functions as oracle to predict the next transition using the DyNet library we follow these steps:

1. Create the `Model` and `Trainer` objects. My implementation, following Kiperwasser and Goldberg's lead, is using the Adam gradient descent algorithm, which means an `AdamTrainer` object is created.
2. Add all parameters to the model, such as the weight matrices and bias terms for the two multi-layer perceptrons.

3. Add all lookup parameters to the model. In the implementation where I am only taking word forms and part-of-speech tags into account there is a lookup table containing vectors for all the words and one for all the POS tags encountered in the training data.

4. Use DyNet's `LSTMBuilder` to create LSTMs for both layers of the BiLSTM representation we are using. Altogether that means 4 LSTMs are created: a forward and a backward one for each of the layers. For the first layer we denote these as `forward1` and `backward1` and for the second layer as `forward2` and `backward2`.

5. For each training batch do the following:

   (a) Create a `ComputationGraph` object and populate it by building expressions from the parameters.

   (b) For the next sentence in the training data, x, initialise the configuration of the transition system to $(\sigma = [0], \beta = x, A = [])$.

   (c) Create a 2-layer BiLSTM representation for each token x_i in x in the following way:

   - Look up each token x_i's form and POS tag in the corresponding lookup tables and set their embedding vector e_i to be the concatenation of them.
   - Loop over the tokens in x and add each e_i to `forward1` using the `add_input` function.
   - Loop over the tokens of the sentence in reverse order and add their embedding vectors to `backward1`.
   - For each token x_i in the sentence set its intermediary BiLSTM vector representation v'_i to a concatenation of the output of `forward1` representing sequence $x_0, ..., x_i$ and the output of `backward1` representing $x_n, ..., x_i$.
   - Add each v'_i in order to `forward2`.
   - Add each v'_i in reverse order to `backward2`.
   - For each token x_i in the sentence set its BiLSTM vector representation v_i to a concatenation of the output of `forward2` representing sequence $v'_0, ..., v'_i$ and the output of `backward2` representing $v'_n, ..., v'_i$.

   (d) The next step is using the current model parameters to predict what transition to apply. To do so we concatenate the BiLSTM vector representation of the first k items on the stack and the item in the beginning of the buffer and give it as input to two expressions representing multi-layer perceptrons. Figure 5.1 shows an example computation graph for this expression. The embedding vectors are obtained by looking up the words and POS tags, concatenated and their BiLSTM representation is fetched. We then concatenate the BiLSTM vectors for the item on top of the stack and the item in the beginning of the buffer, multiply the result with the weight matrix of the hidden

layer, W^{hidden}, add a bias term b^{hidden}, pass the result to the non-linear activation function $tanh$, multiply it with the weight matrix for the output layer, W^{output} and add a bias term b^{output}. Through this procedure we have now computed a score for each possible transition.

(e) Next we compute a list of all possible transitions that can be applied to the current configuration so that the gold parser tree can still be reached.

(f) We loop over each possible transition, in descending order according to the score they have been assigned. The first time we encounter a transition that is in our list of gold transitions we set it to be the best valid transition and apply it. The first time we encounter a transition that is not in our list of gold transitions we set it to be the best invalid transition.

(g) If the highest valid transition has a score that is less than the best invalid transition plus 1 then we have incurred a loss which we add to a list of loss values.

(h) Once the final configuration has been reached we check how many losses have been incurred. If this value is less than 50, we go to step 5(b).

(i) If 50 loss values have been accumulated, calculate the result of the forward computation through the graph by calling the `value` function on the final loss function that is defined as the sum of accumulated loss values. Then we use its `backward` function to perform back-propagation and use the `AdamTrainer` to update the model parameters and lookup parameters.

5.5 Parsing

When parsing a sentence using a the neural network model to predict the next transition we follow these steps:

1. Load the `Model` we have trained to predict what transition to apply given a current configuration.
2. Create a new `ComputationGraph`.
3. Compute the BiLSTM representation of the sentence. If a word or POS tag cannot be found in the corresponding lookup table, a padding vector is used.
4. Initialise the configuration of the transition system to $(\sigma = [0], \beta = x, A = [])$.
5. Concatenate BiLSTM vectors for the first k items on top of the stack and the first item in the beginning of the buffer and use this vector as input to the multi-layer perceptrons used to compute a score for each transition.
6. Check if the preconditions of the highest scoring transition are fulfilled. If not check the second highest scoring one and so on.
7. Apply the highest scoring legal transition to the current configuration.

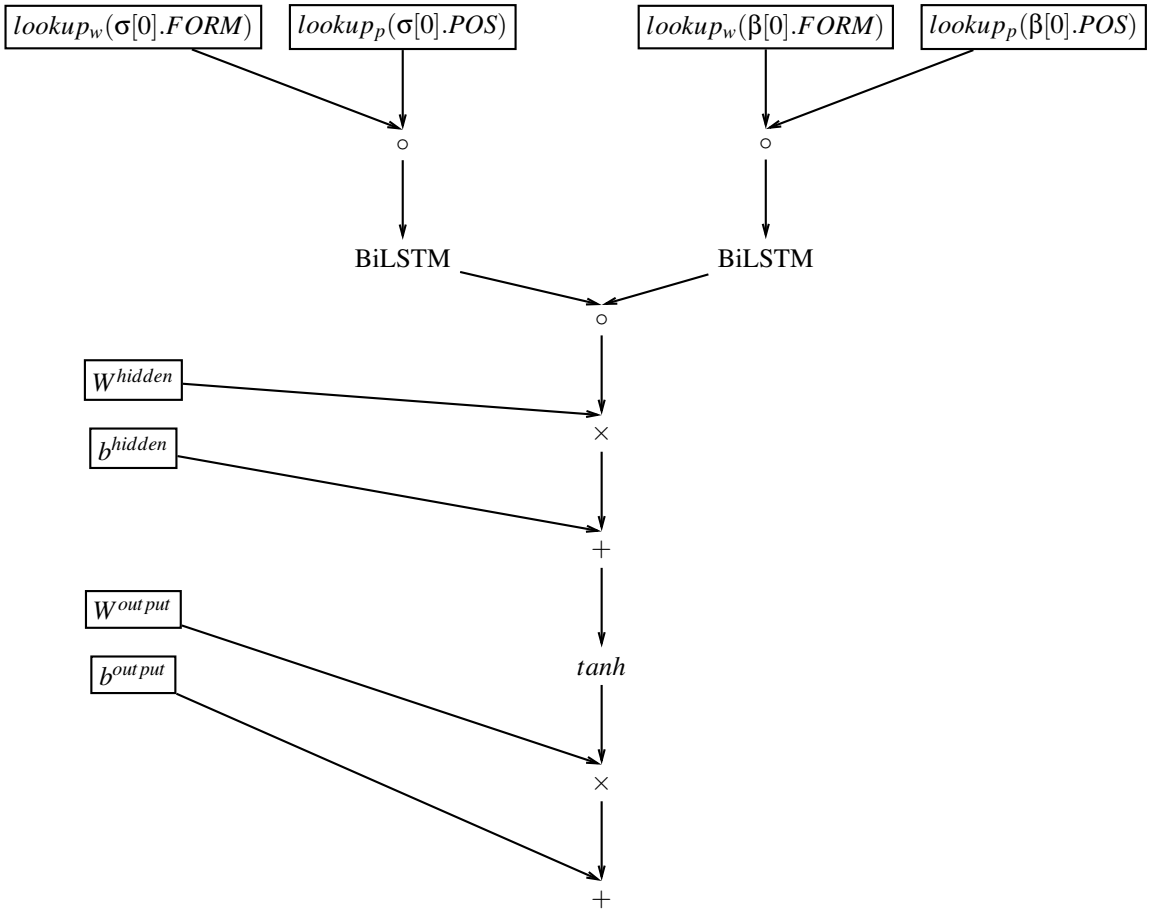

Figure 5.1: A computation graph for the multi-layer perceptron that takes the concatenation of the BiLSTM representation of two embedding vectors as input (that are each a concatenation of a word embedding vector and a POS embedding vector) and outputs scores for each of the transitions. The exact workings of computing the BiLSTM representation for a token given its word and POS embedding vectors is abstracted away here.

8. If the final configuration has not been reached, go to step 5.

5.6 Reproducing Kiperwasser and Goldberg (2016)

In the course of my experiments, I will also compare the performance of my parsers with Kiperwasser and Goldberg's arc-hybrid BiLSTM parser, whose Python code they have made available online and which I will refer to as `barchybrid` parser. To do so, I will train and evaluate it on the same data I am using. In order to ensure I am using their system correctly, I will first start by reproducing the results they presented in their paper. They published parsing accuracies for both English and Chinese, but I will focus only on the English data set here, since reproducing those results would be sufficient in showing that I am using their parser correctly. The exact model I am trying to reproduce is the transition-based parser that uses 4 BiLSTM vectors, no pre-trained word embeddings, and a dynamic oracle, meaning that during training also non-optimal decisions are made in order to improve the system's ability to recover from errors.

They report that they use the Stanford Dependencies conversion of the English Penn Treebank with standard splits into a training, a development and a test set and that they use the same data as Dyer et al. (2015) and Chen and Manning (2014) in order to better compare results. Consulting the latter paper reveals that the Wall Street Journal (WSJ) portion of the Penn Treebank is used, where sections 2-21 are used as training data, section 22 as development data and section 23 as test data.

Even though the treebank data is annotated with gold POS tags, predicted POS tags are used. They are obtained using the Stanford POS tagger (Toutanova, 2003) using a method referred to as "10-way jackknifing". This means the data is split into 10 parts and POS tags are predicted for each of these parts using the other 9 as training data. I am using version 3.6.0 of the Stanford POS tagger, which was the most recent version at the time (Kiperwasser and Goldberg, 2016) was published. It was reported by Chen and Manning (2014) that they achieved an accuracy of around 97.3% when using this method to perform part-of-speech tagging on the WSJ data. I managed to get an accuracy of 97.29% on the training set, 97.11% on the development set and 97.57% on the test set, which seems to reproduce Chen and Manning's results reasonably well.

The training, development and test files with the predicted POS tags are then converted to the Stanford Basic Dependencies format using the Stanford parser (Marneffe et al., 2006), version 3.3.0. I used the resulting dependency data files to train the `barchhybrid` parser using the same hyper-parameters used by Kiperwasser and Goldberg to report their results (e.g. word embedding dimension = 100, POS tag embedding dimension = 25, hidden units = 100).

Looking at Table 5.3 we can see that I managed to reproduce Kiperwasser and Goldberg's results almost exactly. The unlabelled attachment score is 93.1% in both cases,

	UAS	LAS
K&G16	93.1	91.0
Reproduction	93.1	91.7

Table 5.3: Comparison between the results published in (Kiperwasser and Goldberg, 2016) and my reproduction using the Python code of their parser.

while the labelled attachment score I got when using their system was with 91.7% slightly higher (+0.7%) than the result they reported in their paper. It is not unexpected that the results might have some variation since there is a random element when training the parser because the training sentences are randomly shuffled. I would say that the results I report are close enough to the published results that I can claim to have successfully reproduced them.

Chapter 6

Evaluation

6.1 Experiment Results

In the following subsections I will report the performance of the four parsers on a variety of languages with a reasonably large fraction of non-projective sentences. I will compare the results to those I have achieved last year using a binary indicator feature vector and a linear classifier to predict the next transition. Then I will also look at the results of Kiperwasser and Goldberg's parser, which I have trained and evaluated on the Universal Dependency data I am using for my experiments. During the course of implementing the neural network model I have identified several possible improvements which I will explore by looking at how they influence the attachment accuracies of the parsers.

6.1.1 Comparison with Binary Indicator Feature Representation

I have evaluated the four parsers on the same languages I have used last year: Hungarian, Dutch, Portuguese, Danish, German and Czech. First, I evaluated them on the version of the BiLSTM feature representation only looking at the form and universal part-of-speech tags of each word. The results are summarised in Table 6.2. The table shows both the UAS and LAS scores the parsers achieved and how they compare to last year's results that can be found in Table 6.1. When looking at the unlabelled attachment scores we see that the only case in which using a BiLSTM feature model and a neural network as oracle decreases the accuracy of a parser is when I evaluated the arc-eager system on Hungarian and observed a UAS that was 0.3% lower than last year's result. In all other cases we see improved performance, often significantly so.

It is interesting to note that the performance of Convington's parser is significantly improved compared to last year's result. For Dutch for example the UAS increases from 71.26% to 80.49%, which is the highest result of all the parsers. In fact, it beats Covington's parser with reduce transitions on the three languages with the highest fraction of non-projective sentences in the training and test data, Hungarian, Dutch

Binary Indicator Feature Model

	UAS (LAS)			
Language	arc-eager	attach-complete	cov+reduce	cov
Danish	80.76 (77.62)	**81.00** (**78.20**)	80.83 (77.94)	77.09 (74.80)
German	79.63 (74.27)	78.81 (73.85)	**79.90** (**74.69**)	74.83 (70.70)
Hungarian	**79.38** (**75.67**)	78.83 (75.56)	78.09 (74.86)	75.45 (72.51)
Dutch	72.05 (69.10)	71.10 (68.40)	**73.89** (**71.01**)	71.26 (69.11)
Portuguese	81.56 (78.95)	81.98 (79.39)	**83.02** (**80.55**)	79.82 (77.72)
Czech	84.10 (81.30)	82.24 (79.49)	**84.62** (**81.84**)	80.82 (78.55)
Average	79.58 (76.15)	78.99 (75.82)	**80.06** (**76.82**)	76.55 (73.90)

Table 6.1: The Unlabeled Attachment Score and the Labeled Attachment Score for the four parsers I implemented last year on six different languages. A linear classifier that is given a large and sparse vector of indicator features is used to predict the next transition.

and Danish. On the other three languages, which have fewer non-projective structures, adding reduce transitions improves performance. For Portuguese this improvement of unlabelled attachment scores is around half a percent, while it is over one percent for German.

When looking at the labelled attachment scores for Hungarian we see that they decrease by quite a large margin when using the LSTM feature model compared to the linear classifier, which is most likely due to the fact that Hungarian has the fewest training examples (only 1,032 sentences) and that is probably not enough to learn a MLP scoring function that accurately decides between around 80 different transitions (there are up to 40 different dependency labels and usually 2 transitions adding arcs). The only other case where the unlabelled attachment score is lower than last year's is Covington's parser with reduce transitions evaluated on Czech. Since its UAS scores for that language are over 2% higher than last year I would still argue that using a neural network scoring function is beneficial.

Overall, we can observe that the highest parsing accuracy was always achieved by one of the non-projective parsers. While in most cases the attachment scores are so close together that variations might be treated as noise, the language with the most non-projective sentences, Dutch, has unlabelled attachment scores of 74.05% and 76.40% for the projective parsers and 79.45% and 80.49% for the non-projective ones. Taking these significantly improved scores and the fact that there is not a single occasion in which a projective parser outperforms both non-projective ones, I would argue that improved performance can be expected from using parsers allowing for crossing dependency arcs on languages with a considerable amount on non-projectivity.

Table 6.2 also shows that the attach-complete parser outperforms the arc-eager one on

BiLSTM Feature Model

UAS (improvement)

Language	arc-eager	attach-complete	cov+reduce	cov
Hungarian	79.08 (-0.30)	79.45 (+0.62)	**79.71** (+1.62)	**79.71** (+4.26)
Dutch	74.05 (+2.00)	76.40 (+5.30)	79.45 (+5.56)	**80.49** (+9.23)
Portuguese	84.24 (+2.68)	84.70 (+2.72)	**85.03** (+2.01)	84.46 (+4.64)
Danish	82.80 (+2.04)	81.92 (+0.92)	82.09 (+1.26)	**83.45** (+6.36)
German	82.30 (+2.67)	82.36 (+3.55)	**83.95** (+4.05)	82.75 (+7.92)
Czech	86.14 (+2.04)	86.58 (+4.34)	**86.78** (+2.16)	86.43 (+5.61)
Average	81.44 (+1.86)	81.90 (+2.91)	82.84 (+2.78)	**82.88** (+6.34)

LAS (improvement)

Language	arc-eager	attach-complete	cov+reduce	cov
Hungarian	68.66 (-7.01)	69.36 (-6.50)	65.94 (-8.92)	**69.06** (-3.75)
Dutch	70.36 (+1.26)	72.87 (+4.47)	74.70 (+3.69)	**77.03** (+7.92)
Portuguese	81.09 (+2.15)	81.65 (+2.26)	**81.73** (+1.18)	81.23 (+3.51)
Danish	79.62 (+2.00)	78.93 (+0.73)	79.18 (+1.24)	**80.56** (+5.76)
German	77.78 (+3.51)	77.99 (+4.14)	**78.71** (+4.02)	78.32 (+7.62)
Czech	81.88 (+0.58)	82.24 (+2.75)	81.35 (-0.49)	**82.27** (+3.72)
Average	76.57 (+0.42)	77.17 (+1.36)	76.94 (+0.12)	**78.08** (+4.18)

Table 6.2: The UAS and LAS scores of the four parsers with BiLSTM feature representation. The values in bracket show the improvement over the scores I achieved last year using a binary indicator feature model and a linear classifier.

5 of the 6 languages evaluated. This is different to last year's results, where the attach-complete parser only had higher attachment scores on two out of the 6 languages.

It is interesting to note, that for three of the six languages the parsing accuracies achieved were significantly higher on the development set than on the test set, while they were approximately the same for the other three. Even though we are using the development set to choose between the 30 models we train, we never use it directly to update parameters of the models and improvements can be observed consistently over all 30 models (except maybe the first two). For Portuguese and Dutch these increases in unlabelled attachment scores were around 5% and for German around 3%. This shows how dependent such parsing results are on the data you are using to evaluate them. A possible theory would be that the data in the development sets is much more similar to the training data than the data in the test sets.

In order to explore how using a BiLSTM feature model and a neural network scoring function influences dependency parsing performance, I have also looked at how training and parsing times have changed compared to using a binary indicator feature model with a linear classifier. Table 6.3 shows that the neural network model only takes

Training and Parsing Times

Last year

	arc-eager	attach-complete	cov	cov+reduce
Training	200,066	725,968	722,860	304,197
Parsing	39,345	79,268	96,425	53,036

This year

	arc-eager	attach-complete	cov	cov+reduce
Training	60,106	76,356	280,681	80,119
Parsing	103	174	368	121

Table 6.3: A comparison of the parsing and learning time of last year's parsers and this year's parsers. Training data is 40,000 sentences of the Czech training set and testing data the complete Czech test set which contains 10,148 sentences.

a fraction of the time to train and parses over 10,000 sentences in minutes instead of hours. This shows another advantage of the neural network oracle.

When comparing the training and parsing time of this year's parsers we observe that Covington's parser takes about three times as long in both cases as Covington's parser with reduce transitions. That would be an argument for why it would still be beneficial to use this parser even though the attachment performance of the two is very similar.

6.1.2 Comparison with Kiperwasser and Goldberg (2016)

Since the source code of Kiperwasser and Goldberg's arc-hybrid parser with BiLSTM feature representation (`barchhybrid` parser) was made available online, I have been able to train and evaluate it on the same data I am using to evaluate my own parser implementations. I have used the same setup as in my implementation, meaning for example no dynamic oracle, looking at the top three items on the stack when making parsing decisions and using a 2-layer BiLSTM feature representation.

The results are summarised in Table 6.4, where we can see that the `barchybrid` parser has the highest unlabelled attachment scores on two out of the five languages, while my implementation of Covington's parser or Covington's parser with reduce transitions achieves superior performance on the other three. The average UAS scores of the `barchybrid` parser are slightly higher than those of my two highest performing ones, while Covington's parser has higher average labelled attachment scores than the `barchhybrid` parser.

	UAS (LAS)				
Language	arc-eager	attach-complete	cov+reduce	cov	barchybrid
Hungarian	79.08 (68.66)	79.45 (**69.36**)	**79.71** (65.94)	**79.71** (69.06)	79.48 (65.96)
Dutch	74.05 (70.36)	76.40 (72.87)	79.45 (74.70)	**80.49** (**77.03**)	77.03 (72.72)
Portuguese	84.24 (81.09)	84.70 (81.65)	85.03 (81.73)	84.46 (81.23)	**86.62** (**83.07**)
Danish	82.80 (79.62)	81.92 (78.93)	82.09 (79.18)	**83.45** (**80.56**)	83.40 (79.45)
German	82.30 (77.78)	82.36 (77.99)	83.95 (78.71)	82.75 (78.32)	**85.43** (**80.70**)
Average	80.49 (75.50)	80.97 (76.16)	82.05 (76.05)	82.17 (**77.24**)	**82.39** (76.38)

Table 6.4: The performance of the parsers I implemented compared to accuracies achieved with a model I trained on the same data using Kiperwasser and Goldberg's arc-hybrid parser.

6.1.3 Including Lemmas and Features

During the course of my project I explored several changes or extensions to the neural network model that might improve performance. Results above have been reported on parsers using only embeddings of word forms and POS tags as input to the BiLSTM feature vector. In this section I will show how parsing accuracies change when also taking lemma and feature (e.g. case, tense, person,...) embeddings into account. Table 6.5 displays the results of these experiments and shows that in most cases improvements can be observed when representing these additional characteristics. It is interesting to note that while average UAS performance increases for the arc-eager parser, the attach-complete parser and Covington's parser with reduce transitions, we observe a slight decrease of average unlabelled attachment score for Covington's parser.

When looking at the LAS scores we see that for Hungarian, where the previous model using only the word form and universal part-of-speech tags led to low LAS scores under 70%, we can see a vast improvement. The LAS scores for each of the parsers have gone up by around 10%.

6.1.4 Looking at λ

Two of the parsers I have implemented, Covington's parser and Covington's parser with reduce transition, have in addition to a buffer β and a stack σ also a list λ for temporarily storing tokens that can still be involved in further arcs. With the model I am using, I do not look at λ at all when making parsing decisions. I theorised that for Covington's parser with reduce transitions an improvement of attachment scores can be achieved when also looking at the first few elements on λ. For Covington's parser on the other hand I would not expect this to lead to improvements, since λ simply contains all already processed tokens that are not on the stack, which is information that is encoded in the BiLSTM representation of the tokens. I have implemented a variant of both those parsers that, for its feature vector, concatenates the top three elements on the stack, the top three elements on λ and the first element of the buffer. The results can be seen in Table 6.6.

BiLSTM with Lemmas and Features

UAS (improvement)

Language	arc-eager	attach-complete	cov+reduce	cov
Hungarian	82.02 (+2.94)	81.76 (+2.31)	**82.13** (+2.42)	81.72 (+2.01)
Dutch	77.70 (+3.65)	76.67 (+0.27)	**80.16** (+0.71)	79.48 (-1.01)
Portuguese	86.84 (+2.60)	86.88 (+2.18)	86.96 (+1.93)	**87.08** (+2.62)
Danish	83.57 (+0.77)	**83.91** (+1.99)	83.63 (+1.54)	81.10 (-2.35)
German	84.24 (+1.94)	**84.38** (+2.02)	83.37 (-0.58)	81.24 (-1.51)
Average	82.87 (+2.38)	82.72 (+1.72)	**83.25** (+1.20)	82.12 (-0.05)

LAS (improvement)

Language	arc-eager	attach-complete	cov+reduce	cov
Hungarian	**78.75** (+10.09)	**78.75** (+9.39)	78.06 (+12.12)	78.53 (+9.47)
Dutch	74.04 (+3.68)	73.52 (+2.51)	**75.92** (+1.22)	75.85 (-1.18)
Portuguese	84.57 (+3.48)	**84.73** (+3.08)	84.25 (+2.52)	84.22 (+2.99)
Danish	80.91 (+1.29)	**81.14** (+2.21)	80.27 (+1.09)	77.96 (-2.60)
German	81.30 (+3.52)	**81.65** (+3.66)	79.99 (+1.28)	77.56 (-0.76)
Average	79.91 (+4.57)	**79.96** (+4.17)	79.70 (+3.65)	78.82 (+1.58)

Table 6.5: The attachment scores of the parsers when taking additional features of the words into account. The values in bracket show the improvement over the model only looking at word forms and POS tags.

Looking at λ

UAS (improvement)

Language	cov+reduce	cov
Hungarian	79.12 (-0.59)	78.61 (-1.10)
Dutch	81.14 (+1.69)	81.00 (+0.51)
Portuguese	85.05 (+0.02)	84.33 (-0.13)
Danish	83.74 (+1.65)	82.73 (-0.72)
German	82.84 (-1.11)	82.78 (+0.03)
Average	82.38 (+0.33)	81.89 (-0.28)

Table 6.6: The attachment scores of the non-projective parsers when also looking at the first three tokens in λ.

BiLSTM with Limited Look on Stack

	UAS (improvement)			
Language	arc-eager	attach-complete	cov+reduce	cov
Hungarian	76.22 (-2.86)	77.61 (-1.84)	81.50 (+1.79)	**81.54** (+1.83)
Dutch	75.78 (+1.73)	75.78 (-0.62)	80.40 (+0.95)	**80.97** (+0.48)
Portuguese	84.40 (+0.16)	84.60 (-0.10)	**85.46** (+0.43)	85.08 (+0.62)
Danish	81.11 (-1.69)	81.78 (-1.14)	82.56 (+0.47)	**83.11** (-0.34)
German	82.23 (-0.07)	82.91 (+0.55)	**83.56** (-0.39)	81.67 (-1.08)
Average	79.95 (-0.55)	80.54 (-0.63)	82.70 (+0.65)	82.47 (+0.30)

Table 6.7: The attachment scores of the two non-projective parsers when also looking at the first three elements on the list λ when making parsing decisions.

We can see that the average UAS scores slightly increase for Covington's parser with reduce transitions, while they slightly decrease for Covington's parser. Though I have expected increased results for Covington's parser with reduce transitions and a slight decrease for Covington's parser is not unexpected due to the fact that additional redundant information might actually decrease performance, both these changes are so small that they seem more like noise. I would therefore argue that looking at the elements on the temporary list λ also does not benefit Covington's parser with reduce transitions.

6.1.5 Limiting Look on Stack

Kiperwasser and Goldberg (2016) reported good results from concatenating the BiLSTM feature representation of the top 3 words on the stack and the word in the beginning of the buffer and that is how I have also implemented my models. I theorised that for Covington's parser only looking at the first element on the stack should be sufficient and was also interested in finding out how limiting the look on the stack would influence the parsing accuracies of the other parsers. Therefore I implemented a version of the neural network oracle that only takes the BiLSTM representation of the item on top of the stack (or the top two items in the case of the attach-complete parser since arcs are added between the first and second item on the stack) and the item in the beginning of the buffer into account when making predictions.

The results of those experiments can be seen in Table 6.7. We see that on average the unlabelled attachment score of both projective parsers decreases slightly while an increase can be observed for the non-projective parsers. Since we however have both increases and decreases for all parsers except for Covington's with reduce transitions depending on the language and the values are quite small in almost all cases, I cannot argue with certainty what effect looking at three items on the stack has. It seems the BiLSTM representation of the sentence structure is expressive enough to make correct parsing decisions when only looking at one token on the stack.

6.2 Critical Evaluation of Results

One of the motivating factors in deciding to implement a neural network oracle for the four dependency parsers was that the parsing accuracies achieved using a binary indicator feature model were significantly lower than state-of-the-art accuracies reported on the Universal Dependency data for some of the languages I used in my experiments. Tiedemann (2015) showed that a dependency parser using Mate tools which ”combine the advantages of graph-based and transition-based dependency parsers” (Bohnet and Kuhn, 2012), on the Universal Dependencies data for German and Hungarian achieved better performance than the parsers I implemented, with unlabelled attachment scores over 5% higher. Comparing this years results to the parsing accuracies published in (Tiedemann, 2015) shows that while they still have about 3% higher UAS scores for Hungarian, the highest UAS score for German I obtained during my experiments matches theirs exactly while my LAS score is over 2% higher. The reason for my lower performance on Hungarian might be that a neural network feature model needs more training data before performance converges.

Most of the dependency parsing results published in papers are using different data sets than the one I am using, making direct comparison difficult. However, a few other results of dependency parsers on the languages and data I am using have been published in recent months. Six different parsers (MaltParser, MaltOptimizer, MATE, RB61, RB63, TurboParser) have been evaluated on the German Universal Dependency data with a reported average LAS score of 80.17% (Kirilin and Versley, 2015). The highest result of any of my parsers was with a LAS of 81.65% slightly above that. A neural network transition-based dependency parser similar to that of Chen and Manning (2014) has been introduced and evaluated on all languages in the Universal Dependency data set (Straka et al., 2016). They have also used additional lexical and morphological features for their implementation and when comparing their best results to my best results using also lemma and feature embeddings, I observe that I could show higher scores for Danish, Dutch and Hungarian, higher labelled attachment scores for German, but slightly lower unlabelled attachment scores for German and lower scores for Portuguese.

During the course of the second part of my project I have shown evidence of how using a BiLSTM feature model in combination with a non-linear MLP oracle to predict parsing decisions improves performance of dependency parsers over a feature model with binary indicator features using a limited set of feature templates. I have however not managed to show conclusive evidence for how such a model is superior to other kinds of dependency parsers, such as the MATE parser. If we are interested in improving parsing results even further, it would be necessary to explore additional improvements such as using predefined word embeddings alongside those obtained from the training data. Such embeddings could be learnt from large corpora of unannotated text and would be especially helpful for languages with a limited amount of Universal Dependencies training data.

Chapter 7

Conclusion

During the first part of my project I have developed two novel transition systems and implemented them alongside two well-known dependency parsers. The main objective was to reason about how allowing for non-projective arcs during dependency parsing would influence parser performance on a number of languages with a significant number of non-projective structures. While I could show that the highest attachment accuracies could be achieved by a non-projective parser, I was still getting considerably lower scores than state-of-the-art dependency parsers. A recent trend in the field of natural language processing that involved using neural network models to predict actions motivated me to explore how using such models could improve performance of my parsers. Previously I had used a binary indicator feature vector to describe the current configuration during parsing that was then fed to a linear classifier that predicted the next transition. After exploring problems with such a representation, which included the fact we only looked at a very limited number of tokens when making parsing decisions and that deciding on the optimal features and combination of features of those tokens is difficult, I decided to implement the BiLSTM feature representation introduced in (Kiperwasser and Goldberg, 2016).

The main idea of this model was to represent each word in the sentence in terms of a BiLSTM that could capture the whole context. Such a BiLSTM is a powerful tool that is capable of learning what of that information is useful when predicting transitions. Parsing decisions were made by concatenating several of these BiLSTM representations and giving the resulting vector as input to a non-linear scoring function. Implementing this feature model for the four parsers significantly improved results compared to last year with considerable reductions in training and parsing times.

In addition to showing the benefits of such a neural network feature model, I could also provide evidence of how using a parser allowing for non-projective dependency trees could improve attachment accuracies. The novel non-projective transition system I introduced last year which behaves like the arc-eager parser on projective trees, while using the setup of Covington's parser to deal with non-projective arcs, was shown to achieve accuracies matching Covington's parser while being significantly faster. I

have also shown how a transition system that does not exhibit spurious ambiguity, the novel attach-complete parser, could lead to slightly improved parsing accuracies compared to the widely used arc-eager parser that suffers from such ambiguity. While the differences are so small they might just be noise, it could still suggest that there are advantages to using a transition system where each different sequence of transitions leads to a unique dependency tree.

Overall, I could report results matching state-of-the-art parsing accuracies on the same data. I would therefore claim I have managed to show that my novel transition systems, the attach-complete system and Covington's transition system with reduce transitions, are interesting and valuable variations of the systems they were motivated by.

Bibliography

Bohnet, B. and Kuhn, J. (2012). The best of both worlds: A graph-based completion model for transition-based parsers. In *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics*, EACL '12, pages 77–87, Stroudsburg, PA, USA. Association for Computational Linguistics.

Chen, D. and Manning, C. D. (2014). A fast and accurate dependency parser using neural networks. In *Empirical Methods in Natural Language Processing (EMNLP)*.

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.

Goldberg, Y. (2015). A primer on neural network models for natural language processing. *CoRR*, abs/1510.00726.

Graves, A. (2013). Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850.

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

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

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

Kirilin, A. and Versley, Y. (2015). What is hard in Universal Dependency Parsing? In *6th Workshop on Statistical Parsing of Morphologically Rich Languages (SPMRL 2015)*, pages 31–38.

Kübler, S., McDonald, R., and Nivre, J. (2009). Dependency parsing. *Synthesis Lectures on Human Language Technologies*, 2(1):1–127.

Marneffe, M., Maccartney, B., and Manning, C. (2006). Generating typed dependency parses from phrase structure parses. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC-2006)*, Genoa, Italy. European Language Resources Association (ELRA). ACL Anthology Identifier: L06-1260.

Mikolov, T., Karafiát, M., Burget, L., Cernocký, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010*, pages 1045–1048.

Neubig, G., Dyer, C., Goldberg, Y., Matthews, A., Ammar, W., Anastasopoulos, A., Ballesteros, M., Chiang, D., Clothiaux, D., Cohn, T., Duh, K., Faruqui, M., Gan, C., Garrette, D., Ji, Y., Kong, L., Kuncoro, A., Kumar, G., Malaviya, C., Michel, P., Oda, Y., Richardson, M., Saphra, N., Swayamdipta, S., and Yin, P. (2017). Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*.

Nivre, J. (2008). Algorithms for deterministic incremental dependency parsing. *Comput. Linguist.*, 34(4):513–553.

Nivre, J., Agić, Ž., Aranzabe, M. J., Asahara, M., Atutxa, A., Ballesteros, M., Bauer, J., Bengoetxea, K., Bhat, R. A., Bosco, C., Bowman, S., Celano, G. G. A., Connor, M., de Marneffe, M.-C., Diaz de Ilarraza, A., Dobrovoljc, K., Dozat, T., Erjavec, T., Farkas, R., Foster, J., Galbraith, D., Ginter, F., Goenaga, I., Gojenola, K., Goldberg, Y., Gonzales, B., Guillaume, B., Hajič, J., Haug, D., Ion, R., Irimia, E., Johannsen, A., Kanayama, H., Kanerva, J., Krek, S., Laippala, V., Lenci, A., Ljubešić, N., Lynn, T., Manning, C., Mărănduc, C., Mareček, D., Martínez Alonso, H., Mašek, J., Matsumoto, Y., McDonald, R., Missilä, A., Mititelu, V., Miyao, Y., Montemagni, S., Mori, S., Nurmi, H., Osenova, P., Øvrelid, L., Pascual, E., Passarotti, M., Perez, C.-A., Petrov, S., Piitulainen, J., Plank, B., Popel, M., Prokopidis, P., Pyysalo, S., Ramasamy, L., Rosa, R., Saleh, S., Schuster, S., Seeker, W., Seraji, M., Silveira, N., Simi, M., Simionescu, R., Simkó, K., Simov, K., Smith, A., Štěpánek, J., Suhr, A., Szántó, Z., Tanaka, T., Tsarfaty, R., Uematsu, S., Uria, L., Varga, V., Vincze, V., Žabokrtský, Z., Zeman, D., and Zhu, H. (2015). Universal dependencies 1.2. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics, Charles University in Prague.

Nivre, J. and Nilsson, J. (2005). Pseudo-projective dependency parsing. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, ACL '05, pages 99–106, Stroudsburg, PA, USA. Association for Computational Linguistics.

Straka, M., Hajič, J., and Straková, J. (2016). Parsing universal dependency treebanks using neural networks and search-based oracle.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215.

Tiedemann, J. (2015). Cross-lingual dependency parsing with universal dependencies and predicted pos labels. In *Proceedings of the Third International Conference on Dependency Linguistics (Depling 2015)*, pages 340–349.

Toutanova, K., Klein, D., Manning, C. D., and Singer, Y. (2003). Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*, NAACL '03, pages 173–180, Stroudsburg, PA, USA. Association for Computational Linguistics.

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*, HLT '11, pages 188–193, Stroudsburg, PA, USA. Association for Computational Linguistics.