A High-Throughput Dependency Parser

by

Andrei Vacariu

B.Sc., Simon Fraser University, 2016

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Science

Approval

Name:	**Andrei Vacariu**
Degree:	**Master of Science (Computing Science)**
Title:	**A High-Throughput Dependency Parser**
Examining Committee:	**Chair:** Parmit Chilana Assistant Professor
	Anoop Sarkar Senior Supervisor Professor
	William Sumner Supervisor Assistant Professor
	Fred Popowich Internal Examiner Professor
Date Defended:	**October 17, 2017**

Abstract

Dependency parsing is an important task in NLP, and it is used in many downstream tasks for analyzing the semantic structure of sentences. Analyzing very large corpora in a reasonable amount of time, however, requires a fast parser. In this thesis we develop a transition-based dependency parser with a neural-network decision function which outperforms spaCy, Stanford CoreNLP, and MALTParser in terms of speed while having a comparable, and in some cases better, accuracy. We also develop several variations of our model to investigate the trade-off between accuracy and speed. This leads to a model with a greatly reduced feature set which is much faster but less accurate, as well as a more complex model involving a BiLSTM simultaneously trained to produce POS tags which is more accurate, but much slower. We compare the accuracy and speed of our different parser models against the three mentioned parsers on the Penn Treebank, Universal Dependencies English, and Ontonotes datasets using two different dependency tree representations to show how our parser competes on data from very different domains. Our experimental results reveal that our main model is much faster than the 3 external parsers while also being more accurate in some cases; our reduced feature set model is significantly faster while remaining competitive in terms of accuracy; and our BiLSTM-using model is somewhat faster than CoreNLP and is significantly more accurate.

Keywords: natural language processing, dependency parsing, transition parsing system, neural network

Acknowledgements

I would like to thank my senior supervisor, Dr. Anoop Sarkar, for introducing me to the task of dependency parsing and to the problem of parsing all of Wikipedia for use in the LensingWikipedia project. He gave me the opportunity to work on this task, as well as several others in NLP, from which I learned an immense amount, and I am sincerely grateful for his support and guidance along the way.

I would also like to thank Dr. Nick Sumner for his advice on analysing the performance of software which allowed me to greatly increase the speed of the parser presented in this thesis.

Finally, I would like to thank my parents for their love and support throughout my studies, and for helping me get this far.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In many natural language processing tasks, it is useful, or even necessary, to know the semantic relations between certain words in a sentence. In the task of dependency parsing, these relations are known as *dependencies*. For example, in the phrase “a red car”, “a” and “red” both depend on the word “car”; “a” being a determiner, and “red” being an adjective. If we are interested in determining the colour of the car in the phrase, for example, it is sufficient to find all adjectives depending on the word “car” and picking out the one which is a colour.

Another, more interesting example occurs when the main word is a verb: “John kicked the ball.” In this case, we are interested in word “kicked”, which has two dependents: “John” and “ball” which are subject and object of the word “kicked”, respectively. These dependents help us understand the action being performed in this sentence. To complete the example, the word “the” is the dependent of the word “ball” (as a determiner).

One thing to note in the examples above is that each word depends on at most one other word, but can be be depended on by any number of words. Although this property is controversial [23], it is widely accepted within dependency parsing literature and all commonly available dependency parsers make this assumption. Therefore, we also assume that each word in a sentence can only depend on one other word.

In most dependency parsing literature, the dependency relation is a binary relation with the word on which the *dependent* depends being referred to as the *head*, *regent*, or *governor*. In this thesis, we will use the *head* and *dependent* terminology.

The task of determining the head word for each word of a sentence is called *dependency parsing*, and the output produced by a dependency parser is a *dependency graph*.

Figure 1.1 shows an example of a dependency graph for a sample sentence from one of our training sets, Universal Dependencies English: “This chef knows what he is doing.” The ROOT token is an artificial token which simply allows us to have an arc to the word ‘knows’ to indicate that it is the root of the sentence and it has no head word.

Going from the beginning of the example sentence, we see that ‘this’ has one head word ‘chef’ and the arc is labeled with ‘det’, indicating that ‘this’ acts as a determiner for the word ‘chef’. The word ‘chef’ itself has the head word ‘knows’, with the label ‘nsubj’, meaning that it is the nominal subject and the agent of the clause headed by ‘knows’; more simply, it is the subject of the verb ‘knows’ and is the one who does the knowing. The other argument to ‘knows’ is the thing the chef knows, which is the clause “what he is doing”. This is encoded as an arc from ‘knows’ to the head of the clause, the word ‘doing’, with the ‘clausal component’ label (meaning it is a dependent clause which is a core argument); this indicates that the clause functions as the object of the verb. In the remaining clause, we again have a verb, ‘doing’, with two arguments: ‘he’ which is the nominal subject (the do-er), and ‘what’, labeled with ‘dobj’, which is the direct object (the thing being done). The word ‘is’ is a dependent of ‘doing’ with the label ‘aux’ (meaning auxiliary), indicating that it is a function word and may express the tense of the phrase (in this case, present tense). Finally, the punctuation is added to the root of the sentence, ‘knows’, with a label ‘punct’.

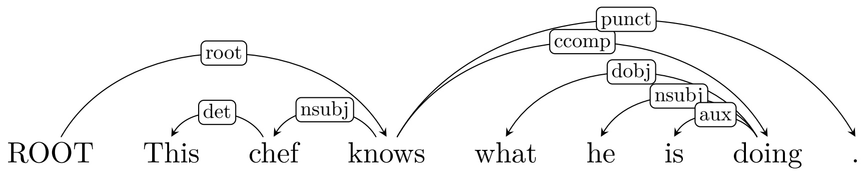


Figure 1.1: A dependency parse of the sentence “This chef knows what he is doing.” which was extracted from the UD English dataset [43] (described in Section 6.1).

In the rest of this chapter, we will explore some applications of dependency parsing, followed by an explanation of the sort of dependents that may exist, and finally we will formally describe a dependency graph.

In Chapter 2 we will discuss the major classes of parsing algorithms, with a focus on the Arc-Standard algorithm we used in our parser. In Chapter 3 we will provide an overview of different neural network structures and how they are trained, to lay the foundation for the subsequent chapters. In Chapter 4 we will describe some prior work in terms of high performance parsing tools. In Chapter 5 we will describe our parsing algorithm, as well as our implementation of the algorithm. In Chapter 6 we describe the experiments we ran in terms of accuracy and speed, and present our results. And finally in Chapter 7 we will conclude.

1.1 Applications

Dependency parsing can have many applications, both for providing features of a word, as well as being used directly to extract information out of a sentence. Although similar information can be provided by constituency parsers—with a conversion to a dependency graph afterward [10, 40]—Cer, et al. [5] showed that dependency parsers are typically significantly faster than constituency parsers (to parse the Penn Treebank development set the fastest constituency parser they tested took 10:14 minutes, whereas the fastest dependency parser took 15 seconds).

Dependency parsing can be used in tasks such as

- semantic role labelling [16];
- question answering [9, 41];
- translation [35]; and
- relationship extraction [17].

Semantic role labelling is the task of assigning semantic roles to words of a sentence. Question answering is the task of providing an answer given a question. In translation, dependency parsing can be used to help provide the correct forms of words which change based on the word on which they depend. For example, if translating "beautiful <noun>" to French, the translation system can pick 'beau/bel' or 'belle' as the translation of 'beautiful' depending on the gender of '<noun>'; however, this is only possible if the translation system is aware that 'beautiful' is an adjective which modifies '<noun>'.

Relationship extraction is the task of finding relations between entities in documents. For example, given a phrase such as "Jack is married to Jill", a relationship extraction system needs to determine that the entities 'Jack' and 'Jill' are related by 'marriage'. A dependency graph will mark both 'Jack' and 'Jill' as dependents of the word 'married', and the arcs will be labeled such that it is easy to determine the relation between the two entities.

1.1.1 LensingWikipedia

LensingWikipedia is a search and summarization interface for Wikipedia [50, 42]. It provides tools and visualizations for filtering and exploring connections between people, organizations, and locations. It is currently focused on only history-related articles, and parses the content of the articles themselves to piece together what occurred, where it occurred, and who was involved. Processing this data is a slow process and takes several hours.

An interesting extension to LensingWikipedia would be to have it analyze all of Wikipedia's data, not just the history-related articles. English Wikipedia has almost 5.5MM articles[1] [52], however, and it is not very feasible to currently do this in a reasonable amount of time. Given a fast parser, though, would help speed up the process as dependency parses are crucial to understanding what occurs in articles and the parser must be run on a large number of sentences.

[1]Specifically, as of October 1, 2017 it contains 5,486,551 articles.

1.2 Dependency graphs

Dependency graphs are the objects produced by dependency parsers. They encode the head-dependency relation between words in the sentence. A word is said to be a dependent of another word (also known as its head word) if it semantically depends on its head word. As this relation clearly makes a distinction between head word and dependent, dependency graphs are directed graphs.

In this section we formally define *dependency graph* and describe some constraints on it.

We define a sentence S of length n as a sequence of words $[w_1, w_2, \dots, w_n]$. Since dependency graphs encode relations between words, each word w_i corresponds to a node v_i in the dependency graph. For each word w_i, we add an arc from v_i to v_j such that w_j is the head word of w_i.

Note that the direction of the arcs in dependency graphs is not consistent in the literature. Some authors [13] direct arcs from dependents to head words, while others [30, 54] direct arcs from head word to dependents. Here we will follow the latter convention and add arcs from head words to dependents.

As we wish our dependency graphs to encode syntactic dependency information for sentences, we need to add some restrictions on them to model grammatical restrictions in natural languages.

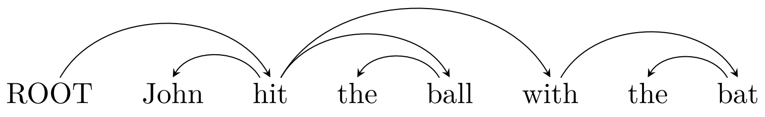
The first such restriction is that dependency graphs must be acyclic. This is clear from how grammatical sentences are structured.

Another restriction is that these graphs must be connected. This constraint arises from the definition of a sentence. Suppose we have a disconnected dependency graph. In this case, the graph represents two sets of words, possibly with one subgraph's words appearing in between some of another subgraph's words. There is no relation between any of the words in one subgraph form any other, meaning the words which form each subgraph are completely independent of the words which form the other subgraphs, but this does not constitute a valid sentence; at most, it is constitutes several independent phrases which may have been mistakenly placed in the same sentence.

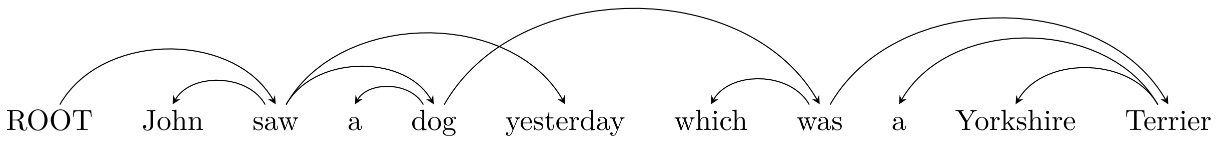
Given our description of a dependency graph, it is clear that dependency graphs are rooted trees (which is why they are also sometimes called *parse trees*). The root of the tree is the word which has no head word. To simplify our algorithms and to make it easier to work with dependency graphs, we follow the convention of adding an artificial ROOT token so that every token in the sentence has a head token, with exactly one token having the ROOT token as the head word. This also means that the ROOT token may only have one dependent, which we'll call the head of the sentence. In the final output produced by parsers, the ROOT token is usually omitted, but the head of the sentence is still labeled with an arc to the root.

An optional restriction that can be placed on dependency graphs is that of projectivity. The formal definition of projectivity is given by Nivre [31], and we repeat it here: A dependency graph is *projective* iff every dependent node is *graph adjacent* to its head. Two nodes n and n' are *graph adjacent* iff every node n'' occurring between n and n' in the surface string is dominated by n or n' in the graph.

Put more simply, if one fixes the position of the nodes in the graph such that they form a line in the same order as the sentence they represent, then none of the dependency arcs cross. Figure 1.2 shows examples of a projective and a non-projective dependency graph.



(a) Projective



(b) Non-projective

Figure 1.2: Examples of dependency graphs from McDonald et al. [30].

The restriction of projectivity is controversial as it does not reflect a grammatical restriction, as can be seen in Figure 1.2b. Thus, if we add this restriction, then we will not be able to produce correct parse trees for some sentences. For example, Figure 1.3 shows how Stanford CoreNLP (which only produces projective dependency graphs) parses the sentence from Figure 1.2b. One significant problem with this parse is that it is impossible to determine from it that the dog John saw was a terrier.

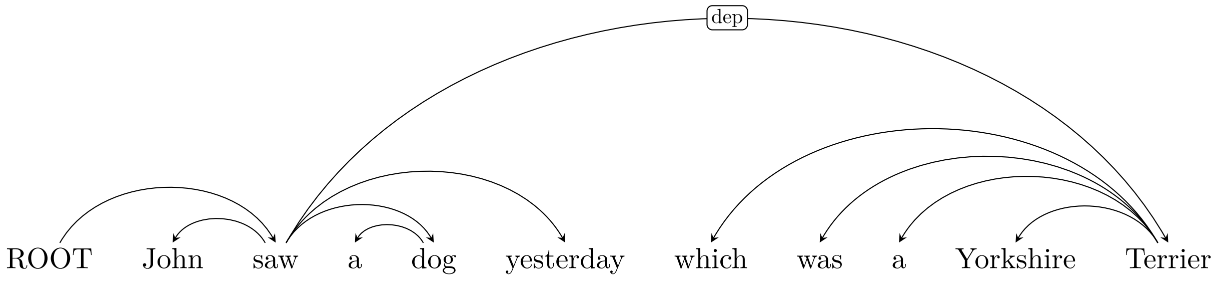


Figure 1.3: Stanford CoreNLP produces a projective dependency graph for a sentence which is non-projective. The ‘dep’ label represents an unspecified dependency, and is present because the parser is not sure how to connect the two subgraphs in a projective manner.

Although parsers which only produce projective dependency graphs are not able to parse all English sentences, assuming all sentences are projective is still a reasonable approximation in practice [31], since in English there are few sentences which are non-projective, as shown in Table 1.1.

Dataset	**Sentences**	**Projective**
Penn Treebank	43,947	99.70%
Universal Dependencies English	16,621	95.05%
Ontonotes 5	143,707	92.04%

Table 1.1: Projectivity statistics for the datasets used in this thesis; these include training, development, and testing sets. More details about these datasets can be found in Section 6.1.

There are parsers which can produce non-projective dependency graphs by moving some words in the sentence while parsing. For example, in Figure 1.2b, if the word "yesterday" were moved after "saw", this would lead to a projective dependency parse. As dependency graphs are trees, there always exists an ordering of the words such that the graph is projective.

1.3 Evaluating a dependency graph

Given a gold dependency graph for a sentence (gold tree), there are two metrics commonly used in literature to evaluate the dependency graph produced by a parser (system tree). They are known as the unlabeled attachment score (UAS) and the labeled attachment score (LAS). A third metric which is sometimes produced by evaluation tools but not commonly reported in literature is the label accuracy score.

To compute the unlabeled attachment score, we compute

$$\text{UAS} = \frac{\text{number of correct unlabeled arcs in system tree}}{\text{number of arcs in gold tree}}$$

where the correct arcs are those which are also present gold tree. These must have the same head and dependents, but the label is omitted in the computation.

Similarly, the labeled attachment score is computed as

$$\text{LAS} = \frac{\text{number of correct arcs in system tree}}{\text{number of arcs in gold tree}}$$

where the correct arcs are those which are present in the gold tree, and have the same label. As this is necessarily a subset of the correct unlabeled arcs, the LAS is always smaller than UAS.

The label accuracy score can be used to evaluate how good the system is at labeling the type of dependent a token is, even though it may have been attached to an incorrect head word. Note that as each token has exactly one head word, we can treat the label as being either attached to the token or to the arc. The label accuracy score is then computed as

$$\text{label accuracy score} = \frac{\text{number of correct labels for tokens in system tree}}{\text{number of tokens in gold tree}}$$

The number of arcs in the gold tree is equal to the number of tokens in the gold tree (as the ROOT token is commonly omitted from the list of tokens in the final tree), so the label accuracy score and LAS are comparable, and so the LAS will always be smaller than the label accuracy score.

Computing UAS and LAS on a complete dataset is done by summing up all correct arcs across the entire dataset and dividing by the total number of arcs in the dataset.

1.4 Annotation scheme for encoding dependency graphs

For different parsers to be trained and evaluated on standard datasets, there must be a standardized annotation scheme for the datasets. There are two commonly used schemes known as Stanford Dependencies (SD) [12, 11] and Universal Dependencies (UD)[2]. The UD scheme is the most recent one and is designed to encode the dependency structures of all the world's languages (hence being "universal"). The two schemes not only define different ways for storing arcs and labels, but also have different hierarchies of labels and POS tags for the tokens within them.

Here we will only describe the UD version 2 annotation scheme. In Section 6.1.1 we will describe how to convert from the more commonly provided treebank format (used for phrase structure information) into the UD scheme.

The UD annotation scheme is designed to provide a generic scheme for annotating dependencies in all languages, such that there is consistency across languages, with language-specific extensions where necessary. It provides both morphological and syntactical information about words in a sentence.

The file format for storing UD annotations is known as CoNLL-U [49], named after the SIGNLL Conference on Computational Natural Language Learning. It is a plaintext format, where each line can be any of the following types:

- Word lines containing annotations for a token.
- Blank line delimiting sentences
- Comment lines (prepended by a #).

Sentences are split in tokens (which may contain one or more words), with a token per line. Each token line provides morphological and syntactical information about the given

[2]`http://universaldependencies.org`

token, as well as its position in the sentence. Figure 1.4 shows the fields provided in each token line.

ID	Word index, integer starting at 1 for each new sentence; may be a range for multiword tokens; may be a decimal number for empty nodes.
FORM	Word form or punctuation symbol.
LEMMA	Lemma or stem of word form.
UPOSTAG	Universal part-of-speech tag.
XPOSTAG	Language-specific part-of-speech tag; underscore if not available.
FEATS	List of morphological features from the universal feature inventory or from a defined language-specific extension; underscore if not available.
HEAD	Head of the current word, which is either a value of ID or zero (0).
DEPREL	Universal dependency relation to the HEAD (root iff HEAD = 0) or a defined language-specific subtype of one.
DEPS	Enhanced dependency graph in the form of a list of head-deprel pairs.
MISC	Any other annotation.

Figure 1.4: Fields in each token line of a CoNLL-U file (extracted from [49])

For this thesis we only looked at the ID, FORM, UPOSTAG, HEAD, and DEPREL fields, as they are the only fields guaranteed to be available.

Chapter 2

Transition-based parsing algorithms

Dependency parsing algorithms can be split into two main categories: graph-based and transition-based. Graph-based algorithms aim to build up the complete parse tree from smaller subgraphs (either spans of the sentence or subtrees covering a span of the sentence); common examples of this are the Eisner algorithm [13], and Easy-First algorithm [18]. Transition-based algorithms, on the other hand, build the parse tree by modeling the building of the tree as a sequence of actions which can be chosen greedily to move the parser from one state to another.

In general, graph-based parsing algorithms are slower than their transition-based equivalents due to their global nature. The most commonly used style (Eisner) has $O(n^3)$ complexity, while modifications of it (to include more context) can have higher complexity.

On the other hand, transition-based parsers are able to run in linear time because the number of transitions they make is constant in the number of tokens they're parsing, and they make relatively greedy decisions. In the next section, we will describe some of the basics of transition parsing systems, including a specific algorithm which is used in the experiments of this thesis.

Transition parsing systems are related to shift-reduce parsers and so have two main components: a parsing state, and a set of transitions. Transitions cause the system to go from one parsing state to another. Parsing happens by initializing the parsing state, and then repeatedly applying a transition until the system arrives in an end-state.

Transition systems differ both in how they define their state (i.e. what they store in their states), and in the set of transitions they use. In this thesis, we use the arc-standard system defined by Nivre [32] which we describe in Section 2.2, however the arc-eager and the arc-hybrid systems are also very commonly used; we will briefly mention their sets of transitions in the same section.

2.1 Common data structures

Almost all transition-based parsers share three key data structures: a stack Σ, a buffer β, and a set of arcs A. All three structures are stored in a configuration $C = (\Sigma, \beta, A)$. We will use 'configuration' and 'parser state' interchangeably.

The buffer β stores all tokens which have not been processed yet. At any point in time, the buffer's elements are denoted as $[b_1, b_2, b_3, \dots]$ where b_1 is the first element in the buffer, b_2 the second, and so on. The tokens are stored in the same order that they appear in the sentence.

The stack Σ contains all tokens which have been shifted from the buffer, and which have no head word yet. Some of these may have gathered dependents already, but some have no dependents. The stack also contains a special ROOT token which is at the bottom of the stack, and which remains there for the entire parsing process. The elements of the stack are denoted as $[ROOT, \dots, s_2, s_1]$, where s_1 is at the top of the stack, s_2 is the second item on the stack, and so on, with the ROOT token being at the bottom. This notation for the stack differs from the more standard notation where the order of the items in the stack is reversed (i.e. $[s_1, s_2, \dots]$), however the standard notation makes it more difficult to visualize the direction of arcs as left-arcs (which go from $s_1 \rightarrow s_2$) would appear to point towards the right if drawn out.

For simplicity in notation, the tokens in the sentence are 1-indexed, however, we say that the ROOT token has index 0.

The last data structure is the set of arcs A. This is simply an unordered set of arcs; it has no special properties other than as a place to store the arcs as the dependency tree is getting built. Arcs within it need to be queried (e.g. to find all dependents of a certain token), but otherwise it may be any data structure which is convenient.

The similarity between most transition-based parsers continues into how they are initialized. The stack Σ is initialized to $\Sigma = [ROOT]$, the buffer is initialized to $\beta = [w_1, w_2, \dots, w_n]$, and $A = \emptyset$. Some transition systems do not require the ROOT token, however.

2.2 Arc-Standard transition algorithm

One of the most common transition parsing systems is arc-standard [32]. The arc-standard system uses the data structures described in Section 2.1, so it is only necessary to describe the transitions available for moving from one configuration to another.

2.2.1 Transitions

Where the transition-based parsing algorithms differ is in the sets of transitions which move the parsing state from one configuration to another. Naturally, each transition must modify at least one of Σ, β, and A in some way.

The arc-standard algorithm provides three transitions which may be applied to a given configuration:

LEFT-ARC(ℓ) Add an arc $s_1 \rightarrow s_2$ to A with label ℓ and remove s_2 from the stack.

RIGHT-ARC(ℓ) Add an arc $s_2 \rightarrow s_1$ to A with label ℓ and remove s_1 from the stack.

SHIFT Pop b_1 from the buffer and push it onto the stack.

The paper which introduced this algorithm [32] used the names LEFT-REDUCE and RIGHT-REDUCE instead of LEFT-ARC and RIGHT-ARC, however the latter terminology is more common now.

In terms of notation, it is important to note with the above transitions that when a certain item s_i, or b_i is removed from the stack, or buffer, respectively, all other items in that data structure is relabeled, such that if (e.g.) b_1 is popped from β, there is still an item known as b_1 in β (the first item in the buffer).

Given the initialization of $C = ([ROOT], [w_1, \dots, w_n], \emptyset)$, then repeated applications of the above transitions can produce any non-projective dependency tree. Thus it is possible with only these three transitions to produce the correct dependency parse of any sentence. Most importantly, there are at most $2n$ transitions necessary to go from initialization state to the end-state.

The LEFT-ARC and RIGHT-ARC transitions remove the dependents from the stack. Dependents of arcs must be removed from the stack at some point to avoid cycles. For example, if the two *-ARC transitions above did not remove the dependent of the created arc, and there was a separate REMOVE transition which removed tokens, then a cycle may be created using a sequence of transitions such as the one in algorithm 1.

Algorithm 1 Causing a cycle in an augmented Arc-Standard system

```
SHIFT
SHIFT
RIGHT-ARC
SHIFT
RIGHT-ARC
REMOVE $s_2$
LEFT-ARC
```

Thus in this situation, extra checks would be necessary to avoid creating the cycle with the LEFT-ARC, and these extra checks would slow down the parser.

From the definition of the transitions in the arc-standard system it is clear that there are situations where certain transitions are invalid. The preconditions for the transitions are shown in table 2.1.

The precondition on LEFT-ARC and RIGHT-ARC ensure that the ROOT token can not be a dependent, and the ROOT token can only be added as a head to an arc when it

Transition	**Precondition**
LEFT-ARC(ℓ)	$\lvert\Sigma\rvert > 2$
RIGHT-ARC(ℓ)	$\lvert\Sigma\rvert > 2$ or $(\lvert\Sigma\rvert = 2$ and $\lvert\beta\rvert = 0)$
SHIFT	$\lvert\beta\rvert > 0$

Table 2.1: The preconditions for the transitions in an Arc-Standard system.

is the last transition to be performed; they also ensure that there are enough tokens in the stack to add an arc between two of them. The precondition on the SHIFT is self-evident: if there is no item in the buffer, there is no item which can be removed from the buffer.

The parsing process ends when we have $\beta = \emptyset$ and $\Sigma = [ROOT]$.

Figure 2.1 summarizes the states and transitions of the arc-standard system.

Initialization	$C = ([ROOT], [w_1, \ldots, w_n], \emptyset)$
End state	$C = ([ROOT], [], A)$
LEFT-ARC	$(\Sigma \vert s_2 s_1, \beta, A) \rightarrow (\Sigma \vert s_1, \beta, A \cup (s_1, s_2))$
RIGHT-ARC	$(\Sigma \vert s_2 s_1, \beta, A) \rightarrow (\Sigma \vert s_2, \beta, A \cup (s_2, s_1))$
SHIFT	$(\Sigma, b_1 \vert \beta, A) \rightarrow (\Sigma \vert b_1, \beta, A)$

Figure 2.1: Summary of the arc-standard system.

Figure 2.2 shows an example of an arc-standard system parsing the sentence “The cat hid in the box.”, and Figure 2.3 shows the final dependency tree that is built.

Transition	**Stack**	**Buffer**	A
	[ROOT]	[the cat hid in the box .]	$\emptyset$
SHIFT	[ROOT the]	[cat hid in the box .]	
SHIFT	[ROOT the cat]	[hid in the box .]	
LEFT-ARC(det)	[ROOT cat]	[hid in the box .]	$A\cup$ det(cat, the)
SHIFT	[ROOT cat hid]	[in the box .]	
LEFT-ARC(nsubj)	[ROOT hid]	[in the box .]	$A\cup$ nsubj(hid, cat)
SHIFT	[ROOT hid in]	[the box .]	
SHIFT	[ROOT hid in the]	[box .]	
SHIFT	[ROOT hid in the box]	[.]	
LEFT-ARC(det)	[ROOT hid in box]	[.]	$A\cup$ det(box, the)
LEFT-ARC(case)	[ROOT hid box]	[.]	$A\cup$ case(box, in)
RIGHT-ARC(nmod)	[ROOT hid]	[.]	$A\cup$ nmod(hid, box)
SHIFT	[ROOT hid .]	[]	
RIGHT-ARC(punct)	[ROOT hid]	[]	$A\cup$ punct(hid, .)
RIGHT-ARC(root)	[ROOT]	[]	$A\cup$ root(ROOT, hid)

Figure 2.2: An example of an arc-standard system parsing a sentence. The format of the arcs is label(head, dependent).

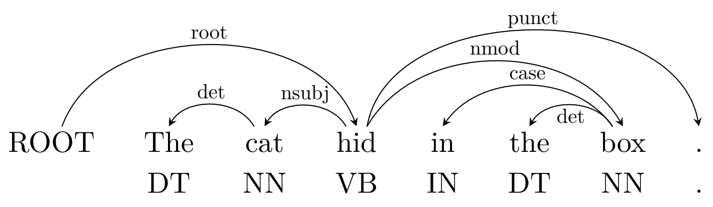


Figure 2.3: A gold dependency tree of the sentence, with the POS tags underneath each token.

2.2.2 Related parsing systems

There are two parsing systems which are closely related to the arc-standard system: arc-eager and arc-hybrid.

Arc-eager The arc-eager system was introduced by Nivre [32] with the aim of increasing incrementality in dependency parsing. He defines incrementality to mean that "at any point during the parsing process, there is a single connected structure representing the analysis of the input consumed so far." To do this, the arc-eager system processes left-dependents bottom-up and right-dependents top-down. It provides the following 4 transitions:

LEFT-ARC(ℓ) Add an arc $b_1 \to s_1$ with label ℓ, and remove s_1.

RIGHT-ARC(ℓ) Add an arc $s_1 \to b_1$ with label ℓ and push b_1 onto the stack.

REDUCE Pop s_1 from stack.

SHIFT Push b_1 onto the stack.

We say that the left-dependents are processed bottom-up because adding a left-arc removes the dependent s_1, implicitly assuming that s_1 has already collected all of its dependents. The right-dependents are said to be processed top-down because adding a right arc does not remove the dependent, so the parser can later move down to that dependent and attach more dependents.

Arc-hybrid The arc-hybrid system was introduced by Kuhlmann et al. [27]. They develop a framework for developing dynamic programming algorithms for transition-based dependency parsing, and in the context of their framework, they create a novel transition system with the following three transitions:

SHIFT Move b_1 from the buffer onto the stack

RIGHT-ARC(ℓ) Add an arc $s_2 \to s_1$ with label ℓ and remove s_1 from the stack.

LEFT-ARC(ℓ) Add an arc $b_1 \to s_1$ with label ℓ and remove s_1 from the stack.

This model combines the LEFT-ARC action from the arc-eager model, and the RIGHT-ARC action from the arc-standard model.

2.3 Making decisions

The Arc-Standard parsing algorithm has $O(n)$ complexity as long as choosing which transition to take is done in constant time. This is due to the fact that there can be at most n SHIFT transitions, and there are exactly n *-ARC transitions as each such transition removes an item from the stack.

To make transition decisions in constant time, it is necessary to consider at most a constant number of features—and to ensure that each feature computation takes constant time. Note that it is possible to analyze the entire sentence a constant number of times while still maintaining the same $O(n)$ bound, as long as this analysis is not performed by the decision function for every decision it makes; for example, it can be performed instead as a pre-processing step. We describe the decision function we used in Chapter 5.

2.4 Training

As the vanilla arc-standard system makes a decision on which transition to apply given the current state, to train it, it requires examples of (state, best transition) tuples. Our training data is in the form of complete dependency graphs, however, so we need a method for generating a sequence of transitions which create a given graph.

The algorithms which create the training examples given a gold dependency graph are known as *oracles*. There are two types of oracles: static and dynamic [19].

Static oracles are ones which, given a gold dependency graph, produce a deterministic sequence of transitions which produce the graph. They are known as static because they are only able to produce a single sequence of transitions even though in some transition systems there can be multiple sequences which produce the same graph. Despite this, the static oracle is used to generate (state, best transition) tuples which are used for training. This becomes an issue at test time because the parser may make a mistake, and then arrive in a state that was not seen during training. This restricts the parser's ability to make the best of the situation and still produce a good dependency graph.

Dynamic oracles, on the other hand, are given a state and a gold dependency graph and asked to produce the optimal transition for reaching the gold dependency graph. This means that any state can be given, and so training examples can be generated which allow the parser to minimize its number of mistakes.

Despite these benefits, however, we used a "shortest stack" static oracle as described by Chen and Manning [6] simply because we wished to compare our performance against their work.

The shortest stack oracle prefers to keep a short stack, and so will prefer to produce a LEFT-ARC, RIGHT-ARC, SHIFT in that order. It picks the next transition in such a way that words are removed from the stack (using an arc transition) only when they no longer need to gather dependents.

The algorithm for producing a transition given a state is described in Algorithm 2 (it is based on the description in [6] and the implementation of the oracle in Stanford CoreNLP). The entire sequence of transitions is produced by starting with the initial state and repeatedly applying the transitions produced by the oracle until the end state is reached.

Assuming each function called by the oracle algorithm runs in constant time, each call to the oracle runs in $O(1)$. Since the oracle is effectively used as a decision function (which produces the gold transitions as a side effect), the time complexity of using it to produce training examples is the same as the time complexity of the parsing system itself (which is $O(n)$ in the case of arc-standard [32], where n is the sentence length).

Algorithm 2 Shortest Stack Oracle algorithm for producing the best transition given the current state C.

```
function ORACLE($C = (\Sigma, \beta, A)$)
    if $|\Sigma| \geq 2$ then
        $i \leftarrow s_2$
        $j \leftarrow s_1$
    else
        $i \leftarrow 0$
        $j \leftarrow s_1$
    end if
    if $i > 0$ and HEAD-OF($i$) is $j$ then
        $\ell \leftarrow$ LABEL-OF($i$)
        return (LEFT-ARC, $\ell$)
    else if $i \geq 0$ and HEAD-OF($j$) is $i$ and HAS-ALL-DEPENDENTS($j$) then
        $\ell \leftarrow$ LABEL-OF($j$)
        return (RIGHT-ARC, $\ell$)
    else
        return (SHIFT, nil)
    end if
end function
```

Chapter 3

Neural networks

In this chapter, we explain some of the basics of neural networks, focusing mainly on the concepts necessary to understand the models presented in this thesis. These include feed-forward networks, recurrent networks, and embedding matrices. We finish with some notes on training these sorts of networks.

Neural networks are simply functions with many parameters which take one or more inputs and produce one or more outputs. They are inspired by the structure of neural connections in human brains, however mathematically they are computation graphs involving matrices and vectors, with various functions applied to them. The relation to the brain is most evident when the term ‘neurons’ is used to describe the dimensionality of certain vectors, although the term ‘hidden units’ is more prevalent.

3.1 The basics

To more easily understand the neural network components in this chapter it helps to understand some basics. We will start with a very simple type of neural network and explain how it works and how to train it. We place the training section here because it helps explain why some neural network components such as the long short-term memory (LSTM) are necessary.

3.1.1 Feed-forward network

The simplest neural networks are known as “feed-forward networks”, as they simply take a fixed input and feed it forward through the network to produce an output. Figure 3.1 shows a very simple example of a feed-forward neural network. For this simple example, consider that each hidden unit, and each output unit computes the following:

$$h = \sigma\left(\sum_{\text{input } x_i} w_i x_i\right)$$

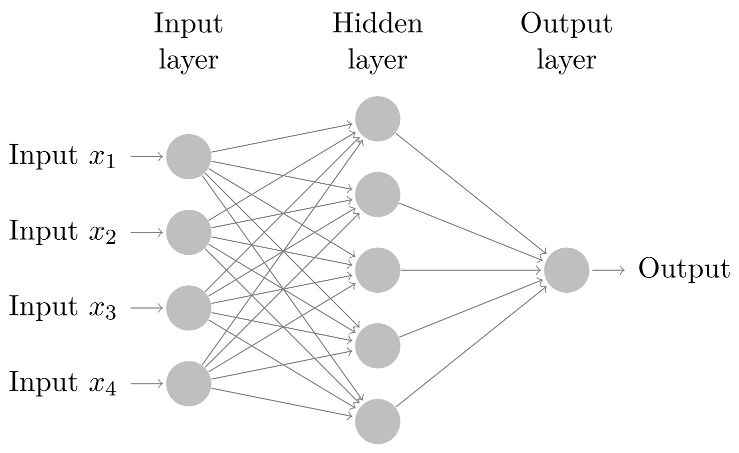


Figure 3.1: A diagram showing a simple feed-forward neural network with 4 input units, 5 hidden units, and 1 output unit. These units may also be called 'neurons', and the connections between them 'synapses'.

where each input to each unit has an associated weight (this is usually drawn on the synapse connecting the two units). Thus, each unit computes the weighted sum of all its inputs, and applies the sigmoid function to it. The function applied is known as the *activation function*.

When each unit in a certain layer is connected to every unit in the previous layer, this is known as a *fully connected layer*. In this case, the network in the figure can be described using the following mathematical notation:

$$\mathbf{h} = \sigma(\mathbf{W}_h\mathbf{x})$$
$$\mathbf{o} = \sigma(\mathbf{W}_o\mathbf{h})$$

where x is a vector of dimension 4 containing the 4 inputs, $\mathbf{W}_h$ is a weight matrix of dimension 4×5, and $\mathbf{W}_o$ is a weight matrix of dimension 5×1. Thus, o is a scalar.

In these networks, it is common to use a *bias* term. Graphically, these may be denoted as an extra neuron in a layer which always produces value 1; thus, for example in the figure, there may be a fifth input (x_5) that always has value 1 such that in the hidden layer, each hidden unit computes

$$h_i = \sigma\left(\sum_{j=1}^{5} w_j x_j\right)$$
$$h_i = \sigma\left(w_5 + \sum_{j=1}^{4} w_j x_j)\right)$$

where w_5 is known as the bias term, and usually denoted as b. In a fully-connected layer, this bias term can be considered a vector, and so it is usually written as:

$$\mathbf{h} = \sigma(\mathbf{W}_h\mathbf{x} + \mathbf{b}_h)$$
$$\mathbf{o} = \sigma(\mathbf{W}_o\mathbf{h} + \mathbf{b}_o)$$

3.1.2 Training

Training a neural network means learning the parameters (or weights) within it so that given an input, it produces the output we desire (or something close to it). When we design a neural network, we only design its structure; we then use the training data to compute good parameters. For simplicity, we will denote by θ all parameters in the network.

For our training data, we need examples of inputs and desired outputs. We will say that example i is composed of input $x^{(i)}$ and output $y^{(i)}$. If we denote our neural network by the function $\mathrm{NN}(\cdot)$, we will say that its predicted output is $\hat{y}^{(i)} = \mathrm{NN}(x^{(i)})$.

To train our network, we need to update the parameters such that $\hat{y}$ is similar to y (for some definition of similarity). To do this, we first need to determine how incorrect the produced output is; we will define a function $E(y, \hat{y}; \mathrm{NN}, \theta, x)$ which we will call the *error function*. This function computes the *error* (also known as *loss* or *cost*).

To decrease the error, we want to change the parameters θ, and to do this, we can compute how a change to θ affects the error. This computation can be done by computing the derivative of $E(\cdot)$ with respect to θ. The derivative then tells us how a change in θ affects $E(\cdot)$; if the derivative is positive in a certain direction, it means that moving θ in that direction will increase the error, so we must change θ in the opposite direction. This is known as *gradient descent*.

Thus, for each parameter w, we compute $\partial E(\cdot)/\partial w$, and then change the value of w a little bit in the opposite direction of the gradient:

$$w := w - \epsilon \frac{\partial}{\partial w} E(\cdot)$$

where ϵ is known as the *learning rate* and is used to (attempt to) keep the descent update from overshooting the minimum.

Notationally, we can denote the update across all parameters by

$$\theta := \theta - \epsilon \nabla_\theta E(\cdot)$$

So for gradient descent, θ needs to be initialized to some value, and then small steps are taken to minimize the error. However, this method will only lead to a local minimum, which is not necessarily the global minimum. There are various techniques for combatting this problem, but they are out of the scope of this quick summary.

In practice, the derivative of the error with respect to each of the parameters is computed when the graph is built and is done automatically. This is possible because the chain rule from calculus allows us to write the derivative of the error as a product of the derivatives of the errors of each of the functions in the graph.

For a simple example, let $x \in \mathbb{R}$ be a scalar, and $f, g : \mathbb{R} \to \mathbb{R}$ be functions. We define our computation graph as

$$\begin{aligned} z &= f(y) \\ y &= g(x) \end{aligned}$$

Then, using the chain rule, we compute the derivative of z with respect to x as

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$$

or, in different notation

$$\begin{aligned} \frac{\partial z}{\partial x} &= f'(y) \cdot g'(x) \\ &= f'(g(x)) \cdot g'(x) \end{aligned}$$

Thus, to compute the gradient of z with respect to x, we need to (i) define the derivative of each of the functions in the computation graph; and (ii) compute the outputs of each of the functions. The first step is usually done when building the computation graph. Most neural network toolkits will have derivatives defined for each of the functions you use in the graph. The second step, however, is performed for each example you feed into the network. The error is often computed as just another function in the graph, and so the expected value is also considered an input to the graph. Feeding inputs to the graph, and computing the values in the graph is known as *forward propagation*; it is often useful to cache the computed values along the way (such as the $g(x)$ in the example above) for efficiency.

Suppose in the example above that our error is z, and the parameter we're allowed to modify is x (example input values are fixed and so are considered constants; the parameters are the 'variables'). To use gradient descent, we do forward propagation, computing y and z. Then we compute $f'(y)$ and $g'(x)$. And finally we update the value of x by

$$x := x - \epsilon \Big(f'(y) \cdot g'(x) \Big)$$

Computing the gradients by going backwards through the computation graph is known as *back-propagation* or *backprop*.

To make the need for back-propagation more obvious, a more complex example is needed. Suppose we have three variables $a, b, c \in R$, and we want to minimize

$$\ell = \sigma(ab + c)$$

To help describe the gradients, we will split up our function in the following way:

$$\begin{aligned}
\ell &= \sigma(y) \\
y &= x + z \\
x &= ab \\
z &= c
\end{aligned}$$

To minimize ℓ, we need to compute $\frac{\partial \ell}{\partial a}$, $\frac{\partial \ell}{\partial b}$, and $\frac{\partial \ell}{\partial c}$; then we will apply the gradient descent update rule:

$$\begin{aligned}
a &:= a - \epsilon \frac{\partial \ell}{\partial a} \\
b &:= b - \epsilon \frac{\partial \ell}{\partial b} \\
c &:= c - \epsilon \frac{\partial \ell}{\partial c}
\end{aligned}$$

Using the chain rule, we compute the gradients as

$$\begin{aligned}
\frac{\partial \ell}{\partial a} &= \frac{\partial \ell}{\partial y} \frac{\partial y}{\partial x} \frac{\partial x}{\partial a} \\
\frac{\partial \ell}{\partial b} &= \frac{\partial \ell}{\partial y} \frac{\partial y}{\partial x} \frac{\partial x}{\partial b} \\
\frac{\partial \ell}{\partial c} &= \frac{\partial \ell}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial c}
\end{aligned}$$

Noting the repetition, we cache the values of gradients such as $\frac{\partial y}{\partial x}$ across the different update rules. The partial gradients are then

$$\begin{aligned}
\frac{\partial \ell}{\partial y} &= \sigma(y)(1-\sigma(y)) \\
\frac{\partial y}{\partial x} &= 1 \\
\frac{\partial y}{\partial z} &= 1 \\
\frac{\partial x}{\partial a} &= b \\
\frac{\partial x}{\partial b} &= a \\
\frac{\partial z}{\partial c} &= 1
\end{aligned}$$

Thus, we have that

$$\begin{aligned}
\frac{\partial \ell}{\partial a} &= \sigma(y)(1-\sigma(y)) \cdot 1 \cdot b \\
\frac{\partial \ell}{\partial b} &= \sigma(y)(1-\sigma(y)) \cdot 1 \cdot a \\
\frac{\partial \ell}{\partial c} &= \sigma(y)(1-\sigma(y)) \cdot 1 \cdot 1 \\
y &= ab + c
\end{aligned}$$

where the value of y is computed as part of the forward propagation, and the values of a, b, and c depend on their current values (when the computation graph is first constructed they must be initialized to some value).

Again, to avoid repetition, we compute the partial derivatives starting from the end of the computation graph (or left-most partial derivative in the expressions above), and back-propagate the gradients backwards toward the beginning. This means we first compute $\sigma(y)(1-\sigma(y))$, and then using that value, we move back one layer through the graph, and multiply that value to the local gradients at that point, repeat this process until reaching the parameters. Each node in the computation graph is aware of the values of its inputs and the value of its output, and during back-propagation will only need to receive the gradient of the global error with respect to its output, and then multiply it with the local gradients of its output with respect to each of its inputs; the back-propagation algorithm will then take these values and continue to the next nodes. If a node's output is used multiple times, then it will simply sum up their corresponding gradients before multiplying the sum by the local gradients.

An important problem caused by the multiplicative nature of the chain rule is that if the gradients are all smaller than 1, or all greater than 1, and there are many of them being multiplied together, the overall gradient of the error with respect to each of the parameters tends towards 0 or towards infinity, respectively. This is known as either *vanishing gradient* in the case of tendency towards 0, and *exploding gradient* in the case of tendency towards infinity.

3.2 Recurrent neural networks

Not all tasks are easily modeled using a feed-forward network as these networks require an input of fixed dimensionality. Some tasks, however, have inputs which can vary in size. For example, tasks involving sentences have inputs whose dimensionality depends on the length of the sentence; one can, of course, set a maximum sentence length to consider, or only look at a fixed number of words in the sentence, but this is not always ideal.

In tasks dealing with sequences where it is required to inspect every element of the sequence, and the sequence can be of any length, it is more useful to use a *recurrent neural network* or RNN.

An RNN is often built dynamically, by reading each element of the input sequence, and building a neural network whose depth depends on the sequence length.

RNNs read a sequence of inputs by

1. starting in an initial state;
2. reading an input;
3. computing a function given the input and the state;
4. producing a new state from the output of the function; then
5. if more inputs are available, repeating from step 2

The recurring sequence of steps for reading and processing each input is where the RNN gets its name. Each time the sequence of steps occurs is known as a *time step*.

The function being applied at each time step is undefined above because it may be anything, and may produce any number of outputs, given any number of inputs; however, for the RNN to function, it must produce some output which can be fed back into the function at the next step and this is known as the 'hidden state'; it must also allow the input at the time step to be fed in. This function is known as the RNN cell, and its structure is usually fixed (although its parameters may change).

The most basic RNN cell is the following:

$$
\begin{aligned}
\mathbf{a}_t &= \mathbf{W}\mathbf{h}_{(t-1)} + \mathbf{U}\mathbf{x}_t + \mathbf{b} \\
\mathbf{h}_t &= \tanh(\mathbf{a}_t) \\
\mathbf{o}_t &= \mathbf{V}\mathbf{h}_t \\
\hat{\mathbf{y}}_t &= \operatorname{softmax}(\mathbf{o}_t)
\end{aligned}
$$

where the hidden state at time step t is $\mathbf{h}^t$, and the cell also produces at output $\hat{\mathbf{y}}^t$. It is often the case that the last hidden state is used to represent an 'encoding' of the sequence.

To train an RNN, we first *unroll* it: we build a graph where we repeat the RNN cell as many times as needed and feed in each x_i to each copy i of the cell. However, we want the parameters to be shared among all the copies (since it's really only one RNN cell), and so after computing the gradients on the unrolled graph, we take the average of all the gradients and use this average to update the parameters.

An issue with the basic RNN cell is that for longer sequences, it is not able to 'remember' what it has read in the past, and thus not able to model long-term dependencies between elements in the sequence. This issue is caused by the fact that the derivative of the tanh function is smaller than 1, and so if the sequence is long, the gradient vanishes.

A commonly used RNN cell which was designed to deal with the vanishing gradient problem and to learn long-term dependencies between elements of a sequence is long short-term memory (LSTM), introduced by Hochreiter and Schmidhuber [21].

LSTM uses the concept of *gates*, which are values that determine how much another value should be reduced or emphasized; for example in an equation such as $\mathbf{y} = \mathbf{f} \odot \mathbf{x}$, $\mathbf{f}$ determines how much of $\mathbf{x}$ remains in the output $\mathbf{y}$, and so is called a 'gate'. Thus, using gates, an LSTM is designed to remember part of the history of its input sequence, while also deciding how much of the new input it will insert into its history.

Formally, the LSTM cell is defined using the following equations:

$$\begin{aligned}
\mathbf{f_t} &= \sigma(\mathbf{W}_f \mathbf{x}_t + \mathbf{W}_f \mathbf{h}_{t-1} + \mathbf{b}_f) \\
\mathbf{i_t} &= \sigma(\mathbf{W}_i \mathbf{x}_t + \mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{b}_i) \\
\mathbf{o_t} &= \sigma(\mathbf{W}_o \mathbf{x}_t + \mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{b}_o) \\
\mathbf{c_t} &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tanh(\mathbf{W}_c \mathbf{x}_t + \mathbf{U}_c \mathbf{h}_{t-1} + \mathbf{b}_c) \\
\mathbf{h_t} &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t)
\end{aligned}$$

where x_t is the input vector; h_t is the output vector; c_t is the cell state vector; $\mathbf{W}$, $\mathbf{U}$ and b are parameters; and f_t, i_t, and o_t are gate vectors: f_t is the forget gate for remembering old information, i_t is the input gate for acquiring new information, and o_t is the output gate.

In the equations above, the cell state vector c_t contains the long-term history of the RNN; the previous output vector and current input are used to determine how much of the old cell state to forget (using element-wise multiplication), and then a new potential cell state is computed and i_t is used to determine how much of it should be added to the cell state. Finally, to produce an output, another gate o_t is used to determine how much of the current cell state should be output (with the tanh being used to keep the value between -1 and 1).

3.3 Bidirectional RNN

As a regular RNN reads sequences in only one direction, it is good at encoding a complete sequence, but if one is interested in using the hidden states at each time step as a way of representing sequence elements, then the representation will only represent the token and the previous tokens. However, if one wants a representation of a token in context, it is useful to also include the subsequent tokens.

To accomplish this, a bidirectional RNN (BiRNN) can be used. A BiRNN actually consists of two RNNs: one that is run over the sequence in the forward direction, and one that is run in the backward direction. Then for each time step, the hidden states from the forward and backward RNNs are concatenated together to produce a single vector. This vector then contains information about the tokens preceding the specific time step, as well as the tokens following it.

3.4 Embedding matrices

When processing text, we receive the input as a sequence of characters or words. However, neural networks require numbers. Thus, we need a way of converting characters or words (henceforth called 'tokens') to numbers. The way to do this is to have a fixed mapping from tokens to consecutive numbers (also known as IDs). For certain types of tokens such as characters, or POS tags this is simple because the total number of tokens can be known ahead of time, but for tokens such as words it is infeasible to have an exhaustive list; for these, we need to determine a fixed-size subset we will be able to handle.

Given a mapping from tokens to numbers, we can transform a sequence of tokens into a sequence of numbers, and given the new sequence we can apply neural networks. However, it is much more useful to represent the tokens as vectors of numbers, rather than as scalars, since this would allow our model to learn useful representations of them.

To do this, we use special matrices known as *embedding matrices*. These matrices have dimension $N \times d$ where N is the number of tokens for which we need vectors, and d is the dimension of the vectors. Given the consecutive IDs we have assigned our tokens, we can extract the corresponding vectors by using the IDs to index into the matrix.

3.5 Training

When training a neural network, we usually have a set of training examples, and when we feed in an example to our neural network, it produces a value for the error. If we have labeled data, we still feed in the label as an input to the network, since it's needed to compute the loss; at test time, we don't compute a loss, so we only need to compute the part of the neural network which produces the output we're interested in.

To train a neural network, there are three main stages: 1. running forward-propagation, 2. computing gradients using back-propagations, and 3. applying the gradients to the variables. The first two parts were described in Section 3.1.2, and despite differences in implementations which aim to provide efficiency and computational accuracy, they are relatively standard. However, there is much work in terms of how to apply the gradients to the variables.

Chapter 4

Other parsers

There are currently several popular parsing tools available, with spaCy[1] [22] being advertised as the “fastest in the world” [14]. The Stanford CoreNLP parser[2] [28], and MALTParser [33, 34] are another two commonly used parsers. In this thesis, we compare our performance and accuracy against these three parsers. Since our model is a derivative of Chen and Manning’s model, we will describe that model in more detail than the other parsers.

In this chapter will describe spaCy’s parsing model, along with CoreNLP’s neural network based dependency parsing model, and MALTParser’s nivrestandard model. Although CoreNLP contains two dependency parsers — a traditional one, and a neural network based one — we will focus on the neural network based one due to it being an implementation of Chen and Manning’s [6] parsing system which reported a very high number of sentences parsed per second.

4.1 spaCy

According to its documentation[3], spaCy implements an arc-eager transition system [32], with feature values generated using a convolutional neural network which is shared by the parser, POS tagger, and named entity recognizer. The features used by spaCy are

1. s_0, s_1, s_2,
2. b_0, b_1,
3. $lc_1(x)$ and $lc_2(x)$ for $x \in \{b_0, b_1, s_0, s_1, s_2\}$
4. $rc_1(x)$ and $rc_2(x)$ for $x \in \{b_0, b_1, s_0, s_1, s_2\}$

[1] `https://spacy.io`

[2] `https://stanfordnlp.github.io/CoreNLP/index.html`

[3] `https://spacy.io/api/#nn-model`

where $lc_i(x)$ is the i-th leftmost child of x and $rc_i(x)$ is the i-th rightmost child of x.

SpaCy is implemented in Cython [3] using Thinc[4].

4.2 Stanford CoreNLP

In this section we describe the neural-network based decision function introduced by Chen and Manning [6], and look at the Stanford CoreNLP implementation of this algorithm.

Chen and Manning use a regular arc-standard system as described in Section 2.2, with their main contribution being their decision function. For the decision function, they use a simple feed-forward neural network that takes vectors of features of the current state and produces a probability distribution over all transitions.

Chen and Manning used the shortest stack oracle to generate training examples.

4.2.1 Features used

As mentioned in Section 2.3, it is necessary for the decision function to be constant-time to maintain the $O(n)$ bound on the overall parsing algorithm. It is also important for speed that we consider a small number of features, and do not analyze the entire sentence at any point in time.

Chen and Manning define the following features (where s_i refer to items in the stack, and b_i refer to items in the buffer).

1. word form and POS tag for each of $[s_1, s_2, s_3, b_1, b_2, b_3]$;
2. word form, POS tag, and label for each of $[lc_1(s_i), rc_1(s_i), lc_2(s_i), rc_2(s_i)]$ for $i = 1, 2$; and
3. word form, POS tag, and label for each of $[lc_1(lc_1(s_i)), rc_1(rc_1(s_i))]$ for $i = 1, 2$.

In the above description, $lc_1(s_i)$ and $lc_2(s_i)$ is the left-most and second left-most dependent of s_i; the case is symmetric for $rc_1(s_i)$ and $rc_2(s_i)$ but for right dependents. Label of s_i is the label of the arc in which s_i is a dependent.

Chen and Manning group these features in three sets: S^w is the set of all word form features, S^t is the set of all POS tag features, and S^ℓ is the set of all label features. The sizes of these sets are denoted as

$$\begin{aligned} n_w &= \left|S^w\right| \\ n_t &= \left|S^t\right| \\ n_\ell &= \left|S^\ell\right| \end{aligned}$$

[4]`https://github.com/explosion/thinc`

4.2.2 Representing features

Chen and Manning use three embedding matrices: one for word vectors, one for part-of-speech (POS) tags, and one for arc labels. The dimension of each of these is fixed, as the vocabulary size is fixed, and the complete list of available POS tags and arc labels is known ahead of time. We denote the embedding dimension as d, and it is the same for all embedding matrices.

We denote the vocabulary size as N_w, the number of POS tags as N_t, and the number of arc labels as N_ℓ. Thus, the dimension of the word embedding matrix $\mathbf{E}^w$ is $d \times N_w$, the dimension of the POS tag embedding matrix $\mathbf{E}^t$ is $d \times N_t$, and the dimension of the arc label embedding matrix $\mathbf{E}^\ell$ is $d \times N_\ell$.

To represent the features, Chen and Manning extract the word, tag, or label embeddings required, and define

$$\begin{aligned}
\mathbf{x}^w &= [\mathbf{E}^w[w_1]; \mathbf{E}^w[w_2]; \ldots; \mathbf{E}^w[w_{n_w}]] \\
\mathbf{x}^t &= [\mathbf{E}^t[t_1]; \mathbf{E}^t[t_2]; \ldots; \mathbf{E}^t[t_{n_t}]] \\
\mathbf{x}^\ell &= [\mathbf{E}^\ell[l_1]; \mathbf{E}^\ell[l_2]; \ldots; \mathbf{E}^\ell[l_{n_\ell}]]
\end{aligned}$$

where w_i, t_i, and l_i are the ith elements in sets S^w, S^t, and S^ℓ, respectively.

4.2.3 Architecture

Chen and Manning use a simple feed-forward neural network architecture, with the inputs being x^w, x^t, and x^ℓ and the output being a vector representing the probability distribution over all transitions.

Given the inputs, they compute the hidden layer of dimension d_h using a cube activation function:

$$h = \left(\mathbf{W}_1^w \mathbf{x}^w + \mathbf{W}_1^t \mathbf{x}^t + \mathbf{W}_1^\ell \mathbf{x}^\ell + b_1\right)^3$$

where $\mathbf{W}_1^w \in \mathbb{R}^{d_h \times (d \cdot n_w)}$, $\mathbf{W}_1^t \in \mathbb{R}^{d_h \times (d \cdot n_t)}$, $\mathbf{W}_1^\ell \in \mathbb{R}^{d_h \times (d \cdot n_\ell)}$, and $b_1 \in \mathbb{R}_h^d$.

A softmax layer is added on top to compute the probabilities:

$$\mathbf{p} = \text{softmax}(\mathbf{W}_2 \mathbf{h})$$

where $\mathbf{W}_2 \in \mathbb{R}^{|\mathcal{T}| \times d_h}$, where $\mathcal{T}$ is the set of transitions. As the set of transitions includes all arc labels for left arcs and for right arcs, $|\mathcal{T}| = 2N_\ell + 1$.

4.2.4 Decision function

The decision function used by Chen and Manning simply extracts the sets of features S^w, S^t, and S^ℓ, executes the feed forward network, and returns the best valid transition. The best valid transition is not always the best transition as predicted by the neural network since the neural network may predict a transition which may not be applicable; for example, it may predict a SHIFT as the most likely transition when the buffer is empty.

Thus, the decision function must sort the transitions by likelihood and then find the first one which is applicable to the current parser state. There are several things that need to be checked:

- If the transition is an arc, it is necessary that there are at least two items in the stack.
- If the transition is a shift, it is necessary that the buffer is not empty.
- A left arc with connecting any work to the ROOT token is invalid as the ROOT token cannot be a dependent.
- A right arc which adds a dependent to the ROOT token is only valid if the label of the arc is 'root'.

4.2.5 Cube activation function

Chen and Manning introduced a novel activation function in their neural network: the cube activation function $g(x) = x^3$. The sigmoid and tanh functions are more commonly used in neural networks, however, they do not model feature combinations.

The intuition for the cube activation function is that the hidden layer can model products of three elements from anywhere in the input layer, thus capturing the interaction between three elements from three different embeddings, thus effectively modeling combinations of three features. This can be seen in the following equation:

$$\begin{aligned} g(w_1x_1 + \cdots + w_mx_m + b) = &\sum_{i,j,k}(w_iw_jw_k)x_ix_jx_k \\ &+ \sum_{i,j} b(w_iw_j)x_ix_j + \sum_{j,k} b(w_jw_k)x_jx_k + \sum_{i,k} b(w_iw_k)x_ix_k \\ &+ \sum_{i} b^2w_ix_i + \sum_{j} b^2w_jx_j + \sum_{k} b^2w_kx_k \\ &+ b^3 \end{aligned}$$

To see what is happening more concretely, let us take a very simple example, where we have one feature for two tokens, and we have already retrieved the relevant two-dimensional embeddings and concatenated them. Thus, our feature vector is

$$\mathbf{v} = [v_1^{(1)}, v_2^{(1)}, v_1^{(2)}, v_2^{(2)}]$$

where the superscript refers to the token. Then our hidden layer is computed as

$$\mathbf{h} = (\mathbf{W}_v\mathbf{v} + \mathbf{b})^3$$

where $\mathbf{W}_v, \mathbf{W}_u \in \mathbb{R}^{2\times 4}$ and $\mathbf{b} \in R^2$, meaning that $\mathbf{h}$ is a two-dimensional vector. However, for simplicity, let us omit the bias term and define our hidden layer as

$$\mathbf{h} = (\mathbf{W}_v \mathbf{v})^3$$

Now let us show the matrix multiplication element by element:

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{bmatrix} \cdot \begin{bmatrix} v_1^{(1)} \\ v_2^{(1)} \\ v_1^{(2)} \\ v_2^{(2)} \end{bmatrix} = \begin{bmatrix} w_{11}v_1^{(1)} + w_{12}v_2^{(1)} + w_{13}v_1^{(2)} + w_{14}v_2^{(2)} \\ w_{21}v_1^{(1)} + w_{22}v_2^{(1)} + w_{23}v_1^{(2)} + w_{24}v_2^{(2)} \end{bmatrix}$$

Since the cubing is element-wise, we'll only look at the top element of the resulting vector, and to make the equation more clear, we will replace the weights by a, b, c, d, and the feature elements by w, x, y, z. Thus, we have

$$\begin{aligned} (aw + bx + cy + dz)^3 =& 6abcwxy + 6abdwxz + 6acdwyz + 6bcdxyz + \\ & 3a^2bw^2x + 3a^2cw^2y + 3a^2dw^2z + 3ab^2wx^2 + 3ac^2wy^2 + 3ad^2wz^2 + \\ & 3b^2cx^2y + 3b^2dx^2z + 3bc^2xy^2 + 3bd^2xz^2 + 3c^2dy^2z + 3cd^2yz^2 + \\ & a^3w^3 + b^3x^3 + c^3y^3 + d^3z^3 \end{aligned}$$

and it is clear that we have interactions between all elements of the feature vector for token 1 with all the elements of the feature vector for token 2. And if we added more features, we would have new elements added to the same sum we have above, so there would be interactions between different elements between different features of different tokens. This effectively models the interaction between any three features among the set of features of the tokens we are considering.

4.3 MALTParser

MALTParser using the LIBLINEAR [15] learner fits a linear classifier to predict the best transition given features of the current configuration. The 'nivrestandard' model implements the arc-standard algorithm. Figure 4.2 shows the features used by this model. Figure 4.1 describes the notation used.

pos(x)	POS tag of x
deprel(x)	Dependency relation of x with its head
head(x)	The head word of x
w(x)	Word form of x
ldep(x)	left-most dependent of x
rdep(x)	right-most dependent of x

Figure 4.1: Explanation of functions used in Figure 4.2. This notation is dissimilar to the other notation used in this thesis (e.g. ldep(x) is $lc_1(x)$ elsewhere), but it is much closer to MALTParser's documentation.

Independent features	pos(x) for x in $\{s_1, s_1, b_1, b_2, b_3, b_4\}$
	deprel(ldep(s_1))
	deprel(rdep(s_1))
	deprel(ldep(b_1)
	deprel(rdep(b_1)
	w(x) for x in $\{s_1, b_1, b_2\}$
	w(head(s_1))

Combined features	pos(s_1) and pos(b_1)
	pos(b_1) and pos(b_2) and pos(b_3)
	pos(b_1) and deprel(ldep(b_1)) and deprel(rdep(b_1))
	pos(b_2) and pos(b_3) and pos(b_4)
	pos(s_1) and pos(b_1) and pos(b_2)
	pos(s_1) and deprel(ldep(s_1)) and deprel(rdep(s_1))
	pos(s_2) and pos(s_1) and pos(b_1)

Figure 4.2: Features used for 'nivrestandard' model.

Chapter 5

Our approach

Our model is similar to the Chen and Manning model [6] described in Section 4.2. To improve speed, though, we made modifications in terms of features used, and the neural network architecture. However, we continued using the arc-standard system since it will only use at most $2n$ transitions to parse a sentence of length n [32].

In this thesis we aim to improve parsing speed while minimizing accuracy loss relative to other parsers, as opposed to increasing both parsing speed and accuracy. However, as some techniques used to improve speed do cause a reduction in parsing accuracy, we also investigate methods for increasing accuracy within the neural network component which don't cause a decrease in speed.

As mentioned in Chapter 2, once we've picked the parsing system, one of the most important components to design is the decision function which outputs a transition given a configuration and transition history. For speed, however, we do not consider the transition history and only take into account the current configuration.

In this chapter we will describe the decision function we used, as well as how we implemented the entire parser. As the decision function takes certain inputs and produces certain outputs, we will first describe the features we extract from the current configuration, then how we represent them, followed by how we compute the transition probabilities, and finish with how we apply the transitions to the configurations.

5.1 Features used

Initially, we used the same features from [6], however when we profiled our code, it appeared the $lc_1(lc_1(s_i))$ and $rc_1(rc_1(s_i))$ features were the slowest features to compute as they required extracting all the dependents for 8 nodes in our tree, only to read the features of 4 of them. When we removed them, we noticed no decrease in accuracy, while gaining a speed increase. The comparison is shown in our results.

Thus, our final set of features is

1. word form and POS tag for each of $[s_1, s_2, s_3, b_1, b_2, b_3]$; and

2. word form, POS tag, and label for each of $[lc_1(s_i), rc_1(s_i), lc_2(s_i), rc_2(s_i)]$ for $i = 1, 2$.

Similarly to [6], we define 3 sets of features: S^w denotes the set of word features, S^t denotes the set of POS tag features, and S^ℓ denotes the set of arc label features. The POS tag features are optional in our parser, allowing the user to increase parsing speed at the expense of a drop in accuracy.

The sizes of the sets of features are denoted as

$$\begin{aligned} n_w &= \left|S^w\right| \\ n_t &= \left|S^t\right| \\ n_\ell &= \left|S^\ell\right| \end{aligned}$$

The features we extract are simply integers which can later be mapped to descriptive strings. To produce useful inputs to the neural network, we use dense vector representations of each of the features. As this requires matrices of fixed sizes to store the representations, we use a fixed size vocabulary which we learn when training. The POS tag set and arc label set are both known in their entirety ahead of time as they are both fully defined in the Universal Dependencies specification.

Thus, we have three embedding matrices (where d is the dimension of the embeddings):

1. $\mathbf{E}^w$ is the word embedding matrix of dimension $N_w \times d$ where N_w is the vocabulary size.

2. $\mathbf{E}^t$ is the POS tag embedding matrix of dimension $N_t \times d$ where N_t is the size of the POS tag set.

3. $\mathbf{E}^\ell$ is the arc label embedding matrix of dimension $N_\ell \times d$ where N_ℓ is the size of the arc label set.

To produce feature values which can be fed into the neural network, we use the integers representing the features as indices into the corresponding embedding matrices; thus, if a feature has value j, we extract row j of the corresponding matrix and use that in the neural network.

In Section 5.6, we discuss how we compute features, and the optimizations we perform.

5.2 Architecture

We use a similar architecture to Chen and Manning [6], with the most significant difference being that we predict the transition and arc labels separately, and we use a different activation function. The separated prediction allows us to perform argsort[1] on the resulting

[1]The argsort function returns “the indices that would sort an array” [46].

vectors much more quickly as there are almost half the number of values in the two smaller vectors than in the single large vector (the combined vector would have size $2|\mathcal{T}|+1$, while the split vectors have a total of $|\mathcal{T}|+3$ elements, where $\mathcal{T}$ is the set of labels).

Once we've extracted our sets S^w, S^t, and S^ℓ of features, we define the inputs to our neural network as

$$\begin{aligned}
\mathbf{x}^w &= [\mathbf{E}^w[w_1]; \mathbf{E}^w[w_2]; \ldots; \mathbf{E}^w[w_{n_w}]] \\
\mathbf{x}^t &= [\mathbf{E}^t[t_1]; \mathbf{E}^t[t_2]; \ldots; \mathbf{E}^t[t_{n_t}]] \\
\mathbf{x}^\ell &= [\mathbf{E}^\ell[l_1]; \mathbf{E}^\ell[l_2]; \ldots; \mathbf{E}^\ell[l_{n_\ell}]]
\end{aligned}$$

where w_i, t_i, and l_i are the ith elements in sets S^w, S^t, and S^ℓ, respectively, and $\mathbf{E}^w[w_i]$, $\mathbf{E}^t[t_i]$, and $\mathbf{E}^\ell[l_i]$ are their respective embeddings.

As our hidden layer, we use a very similar one to [6], except we use the quartic activation function instead of the cube activation. We compared different activation functions and found that the quartic performed a little bit better than the cube activation, with no decrease in speed (see Table 6.9 for the comparison of different activation functions).

$$\mathbf{h} = \left(\mathbf{W}_1^w \mathbf{x}^w + \mathbf{W}_1^t \mathbf{x}^t + \mathbf{W}_1^\ell \mathbf{x}^\ell + \mathbf{b}_1\right)^4$$

Then to produce our outputs we define

$$\begin{aligned}
\mathbf{p}_a &= \text{softmax}(\mathbf{W}_2^a \mathbf{h}) \\
\mathbf{p}_\ell &= \text{softmax}(\mathbf{W}_2^\ell \mathbf{h})
\end{aligned}$$

where $\mathbf{p}_a$ is the probability distribution over transitions, and $\mathbf{p}_\ell$ is the probability distribution over arc labels. Thus, the dimension of $\mathbf{p}_a$ is 3, and the dimension of $\mathbf{p}_\ell$ is $|\mathcal{T}|$ where $\mathcal{T}$ is the set of all labels.

In our parser, we retained the ability to use either two output vectors, or a single one. This allows us test all possible combinations of features and architectures.

We also use the GloVe [37] pretrained word embeddings trained on Wikipedia and Gigaword 5 with 6 billion tokens and 300 dimensions[2]. Since inside our model we may use a different dimensionality for the word embeddings, we apply a linear layer to map the word embeddings extracted from the pretrained matrix into the new vector space; thus if we need the vector for word w_i, instead of using $\mathbf{E}^w[w_i]$, we instead compute

$$\mathbf{W}_G \cdot \mathbf{G}[w_i]$$

[2]`https://nlp.stanford.edu/projects/glove/`

where $\mathbf{G}$ is the GloVe embedding matrix, and $\mathbf{W}_G$ is a matrix of dimension $d \times 300$ where d is the word embedding dimension we use elsewhere in the network.

5.3 Training

To generate training examples, we used the shortest stack oracle (as described in Section 2.4) on each sentence in our training set to generate example tuples of the form (configuration features, best action). We then collected all such examples into one large list L, and shuffled it. We then split this list into batches of size k (with a possibly smaller final batch), and feed these into our neural network one by one. After all batches have been consumed, we re-shuffle the list of examples, and repeat the process. Each batch being fed into the neural network counts as one step, and going through all the examples counts as one epoch. However, we only explicitly count the number of steps.

The training objective is to minimize the cross-entropy loss:

$$L(\theta) = -\sum_i \log p_{t_i} + \frac{\lambda}{2}\|\theta\|^2$$

where p_{t_i} is the predicted probability of the correct transition $t_i \in \mathcal{T}$ for training example i, and θ is the set of all parameters in the network.

To train our network, we used the Adadelta optimizer [53] implementation in TensorFlow, with the initial learning rate set to 1.0.

For initialization of the parameters, we used uniformly random initialization within $(-0.1, 0.1)$ for $\mathbf{E}^w$, $\mathbf{E}^t$ and $\mathbf{E}^\ell$. During training, we also applied a dropout [20, 45] with a 0.5 rate on each of the hidden layers h and h_i. Although Chen and Manning used uniformly random initialization within $(-0.01, 0.01)$, we found that training loss did not decrease when using these values.

We used the following hyper-parameters: embedding size $d = 50$; hidden layer size 200; regularization parameter $\lambda = 0.0001$ for our model, and $\lambda = 10^{-8}$ for Chen and Manning model; and learning rate for Adadelta of 1.0 (ρ and ϵ were left at their defaults of 0.95 and 10^{-8}).

5.4 Extensions to the architecture

In addition to our basic architecture, we also experimented with some extensions in an attempt to improve accuracy without hurting parsing speed. The attempts were in using recurrent neural networks to encode word vectors in context, as well as using different number of hidden layers.

5.4.1 Recurrent neural network

In a simple attempt to retain the small number of features, while also viewing them in a larger context, we experimented with using a bidirectional recurrent neural network (BiRNN) based on an LSTM cell to encode the words of the sentence in context. We fed the word embeddings into the BiRNN, and produced a new sequence of vectors (one for each token in the sentence). We then used these new vectors in place of each $\mathbf{E}^w[w_i]$ in x^w.

We experimented with two methods of training this BiRNN. In the first version, we simply used its outputs in the rest of the network, as described above. In the second method, however, we used POS tagging as a scaffold task. Thus, while using the BiRNN as before, we also added a linear layer on top of each of the outputs of the BiRNN to predict the POS tags of each token. We added the loss of the POS tag prediction to our total loss at training time, while at test time we left out the extra linear layer and used the BiRNN as before.

More formally, given a sentence $S = [w_1, \ldots, w_n]$, we produce

$$[h_1, \ldots, h_n] = \text{BiRNN}([\mathbf{E}^w[w_1], \ldots, \mathbf{E}^w[w_n]])$$

where h_i is the hidden state of the BiRNN at time step i. Then instead of $\mathbf{E}^w[w_j]$ in x^w as defined previously, we replace each such element with the respective BiRNN output

$$x^w = [h_{w_1}; h_{w_2}; \ldots; h_{w_{n_w}}]$$

where the subscripts refer to the word indices in S^w (not in S).

BiRNN with POS tagging as scaffold task

As the simple usage of the BiRNN did not lead to much improvement in accuracy, we decided to also train the BiRNN to produce POS tags. As we already have POS tags for all tokens in our training data, we were able to easily add this new task.

At each time step i, we take h_i and multiply it by a projection matrix W_p such that the output is a vector with the same number of elements as the total number of POS tags available. Then we computed the average cross-entropy loss across all time steps. We then added this loss to our overall loss.

This is similar to stack-propagation [55], and the low-level task supervision of Søgaard and Goldberg [44].

We call this model “Main + POS-BiLSTM”.

5.4.2 Different activation functions

Although Chen and Manning showed that the cube activation function is a good activation function for this structure of neural network, we decided to also experiment with more activation functions to see if there is any benefit in increasing the exponent past 3. As the cube activation function considers combinations of three elements from all the input vectors, using a quartic function would consider combinations of four elements. Thus, we decided to check the limits of this approach by also using the quadratic, quartic, quintic, and sextic activation functions.

Thus, our full list of activation functions is

$$\begin{aligned}
a(x) &= x \\
a(x) &= x^2 \\
a(x) &= x^3 \\
a(x) &= x^4 \\
a(x) &= x^5 \\
a(x) &= x^6 \\
a(x) &= \text{relu}(x) = \max(0, x) \\
a(x) &= \tanh(x) \\
a(x) &= \sigma(x) \\
a(x) &= \tanh(x^3 + x)
\end{aligned}$$

where $\tanh(x^3+x)$ is known as the tanh-cube function and was introduced by Pei, et al. [36].

5.4.3 Different hidden layers

In our parser there is at least one hidden layer (as defined previously), but the activation function may be changed to any of the ones defined in Section 5.4.2.

Our parser also allows extra hidden layers. Each layer i (greater than 1) is defined as

$$h_i = a(\mathbf{W}^{(i)} h_{i-1} + b^{(i)})$$

where $a(\cdot)$ is any of the activation functions mentioned above. This activation function may be different than the first hidden layer, but must be the same for all the extra layers. Each $\mathbf{W}^{(i)}$ is a square matrix to keep the dimension of all the hidden layers the same.

The reasoning is that each entry in the first hidden layer contains all combinations of 3 elements from all feature vectors, but with its own set of weights (as each element i corresponds to row i of each of the weight matrices). However, it may be beneficial to have a linear combination between the rows. As we want the network to only include other rows if it deems it beneficial, we initialize each $\mathbf{W}^{(i)}$ to the identity matrix.

5.4.4 Scaling labels based on arc direction

Since the probability of a left-arc, right-arc, and shift is computed separately than the label probabilities of the potential arc, we considered allowing the transition prediction to affect the label probabilities, such that for certain arc directions, some of the label probabilities may increase or decrease.

We decided to either scale the label predictions by a value between 0 and 1, or to shift the predictions by adding a vector.

The "scale" version is defined as follows:

$$\begin{aligned}
\mathbf{a}_{\text{in}} &= \mathbf{W}_2^a \mathbf{h} \\
\mathbf{p}_a &= \text{softmax}(\mathbf{W}_2^a \mathbf{h}) \\
\mathbf{p}_\ell &= \text{softmax}(\mathbf{W}_2^\ell \mathbf{h} \odot \sigma(\mathbf{W}_2^s \mathbf{a}_{\text{in}}))
\end{aligned}$$

And the "shift" version is defined as follows:

$$\begin{aligned}
\mathbf{a}_{\text{in}} &= \mathbf{W}_2^a \mathbf{h} \\
\mathbf{p}_a &= \text{softmax}(\mathbf{W}_2^a \mathbf{h}) \\
\mathbf{p}_\ell &= \text{softmax}(\mathbf{W}_2^\ell \mathbf{h} + \mathbf{W}_2^b \mathbf{a}_{\text{in}})
\end{aligned}$$

5.5 Decision function

Given that not all transitions are valid, we need to find the most highest-scoring valid transition to apply. To do this, we use two nested loops: we first iterate over p_a in decreasing order, and then over p_ℓ also in decreasing order. For each possible transition we find, we check that

- the preconditions are met;
- the ROOT token is not added as a dependent; and
- the ROOT token is only added as a head at the end.

Once we find a transition that is valid, we apply it, computing the features for the new configuration as described in Section 5.6.2.

5.6 Implementation

We implemented our parser in Cython, with only the main callable being in pure Python. The neural network component was implemented using TensorFlow. To take advantage of TensorFlow and the GPU, we used mini-batches to compute the predicted transitions for multiple configurations at once. This component, however, was the only one which runs in parallel. The rest of the parsing process is sequential. In this section we explain our (pseudo) mini-batching technique, along with optimizations we perform in computing features.

5.6.1 Parsing using mini-batches

To take full advantage of TensorFlow and the GPU, we used mini-batches of configurations. The mini-batch is represented as a list of size k of configurations not in their terminal state. We initialize this list with the initial configurations of the first k sentences to be parsed. Then, for each configuration in the mini-batch, we read the features, and feed them into the neural network as a mini-batch, which then produces two matrices representing p_a and p_ℓ for all the configurations in the batch. We then run argsort[3] on these matrices to produce a sorted list of transitions and labels for all configurations in the mini-batch.

We then iterate over the list of configurations, finding the best valid transition, and applying it. The configurations that are in their terminal state after the transition remain in the list, while the terminal configurations are added to a new "finished" list. The arcs in the finished list are then printed out, and new initial configurations of new sentences are added to the mini-batch to fill out all k entries.

As some sentences require fewer transitions than others to parse, the configurations which are ready to be printed out are usually out of order. Thus, we store the sentence number along with the configuration, and only print out a configuration when all previous sentences' configurations have been printed. To do this efficiently, we maintain a priority queue of terminal configurations, and when a new configuration is added to the queue, we check whether the head of the queue should be printed, and if so, we print out all consecutive configurations starting with the head.

5.6.2 Computing features

As feature computation is performed before choosing a transition to apply, and there are most $2n$ transitions that can be applied per sentence before parsing is completed, it is crucial to have a fast feature computation. Even though we have a small number of features, the large number of times they must be computed ensures that they can still become a bottleneck if not optimized.

For each sentence, we create one configuration object which fully defines its current parsing state. In this configuration object, we also store three arrays, one for each set of features. At any point in time, the arrays store the correct values for the features defined, thus when retrieving the features to feed into the neural network, it is sufficient to simply read these arrays; no extra computation is necessary.

Feature values clearly need to change as the configuration changes, though, but as they only change when a transition is applied, that is when we re-compute the values in the three arrays. Knowing the transition being applied allows us to perform extra optimizations which would not be possible had we computed the features later.

[3]The argsort function returns "the indices that would sort an array" [46].

Optimizations

The optimizations that are possible are due to the fact that transitions do not cause all features to change. Thus, when applying a transition, we only need to re-compute a subset of features, and of that subset some computations become much simpler. More specifically, we note the following properties of the transitions which allow for optimization:

- When a **shift** occurs, the old s_1 moves into the position of the old s_2, and the old b_1 moves into position s_1. Thus, for a shift, it is not necessary to re-compute the features of the dependents of the new s_2; it is sufficient to copy them from the old s_1. The only new features to be computed are the ones for the new s_1 and the top 3 words in the buffer. Some copying can be done for the buffer features as well, but the code computing these features is fast enough that it is not necessary.

- When a **left-arc** transition is applied, the right-dependents of s_1 do not change, but as it now has a new left-dependent, its left-dependent features need to be re-computed, as well as the features for the new s_2.

- When a **right-arc** transition is applied, the left-dependents of the new s_1 do not change; only the right-dependents change. Thus, it is sufficient to re-compute the new right-dependent features, as well as the ones for the new s_2. The other features only need to be copied to their new positions in the arrays.

It is important to note that in this case, when we are "computing" features, we are simply looking them up in the current parse tree, or in the objects representing the words. Thus, no speedup would occur if we considered an optimization where the features for s_3 would be stored somewhere to be copied in when a left-arc transition causes s_3 to become the new s_2. The optimizations are only beneficial when the copying occurs within the arrays representing the features for the current configuration as no extra Python function calls are necessary; the Cython (and thus C) compiler is then able to generate fast memory-copying code.

Chapter 6

Experiments

All experiments were performed on a single machine with an Intel i7-5820K processor and an NVIDIA GeForce GTX 1080; this processor has 6 physical cores, and 12 logical cores. We used the latest versions of all libraries and compilers, except for cuDNN for which we used the latest version supported by the TensorFlow 1.3. The relevant details for the hardware and software versions are listed in Table 6.1.

Hardware component	**Details**
CPU	Intel i7-5820K (6 cores / 12 logical cores)
CPU Frequency	3.30GHz
GPU	NVIDIA GeForce GTX 1080
GPU Memory	8105MiB
Main memory	8 $\times$ 8192MB / DDR4 / 2133 MHz

Software component	**Version**
Ubuntu	16.04.3
Python	3.6.2
Cython	0.26
Nvidia driver	367.57
CUDA	8.0.61
cuDNN	6.0.21
Numpy	1.11
TensorFlow	1.3

Table 6.1: Hardware and software versions on our machine.

We experimented with different sets of features, and with several model structures. To effectively test for speed, we used compile-time flags to include or exclude code rather than runtime conditional statements. Table 6.2 contains the general hyper-parameters used throughout the network unless specified differently in the description of the model.

Parameter	**Value**
Word-dropout α	0.25
Adadelta learning rate	1.0
Embedding dimension d	50
Hidden layer size	200
LSTM encoder dimension	300
Pretrained GloVe embedding dimension	300

Table 6.2: Hyper-parameters used in the models.

6.1 Datasets

We experimented with three datasets. The first was the Universal Dependencies UD English dataset which is based on the English Web Treebank[1] [43]. This contains sentences from weblogs, newsgroups, emails, reviews, and Yahoo! answers. This dataset will be referred to as UD English.

The second dataset was constructed from the Penn Treebank dataset [29] by converting it to Universal Dependencies version 2 format using the Stanford CoreNLP tool. The sentences in this dataset are from the Wall Street Journal, and we used the standard split of sections 2–21 for training, section 22 for development and section 23 for testing. This dataset will be referred to as PTB.

The third dataset is Ontonotes 5.0. Similarly to the Penn Treebank dataset, this one does not contain dependency information but does contain constituency parses which were converted using the NLP4j DDR[2] tool to CoNLL-U format. This tool does not use the universal dependencies labels for relations, however; it instead uses a deep dependency representation[3] which contains both primary and secondary dependencies. Secondary dependencies are used to encode one word referring to another. As these were not relevant for us, we removed secondary dependencies from the converted dataset. This format and conversion algorithm is further described in [8, 7].

As the Ontonotes 5.0 dataset is not split into training, development, and testing sets, we used the same split of the data as the CoNLL-Formatted Ontonotes 5.0 repository[4] which contains the skeleton files used in the CoNLL 2012 Shared Task [38]; the repository does not contain parse information, so we instead used the file names to map the split back to the original data files.

[1]`https://catalog.ldc.upenn.edu/LDC2012T13`

[2]`https://github.com/emorynlp/ddr`

[3]`https://emorynlp.github.io/ddr/pages/overview.html`

[4]`https://github.com/ontonotes/conll-formatted-ontonotes-5.0`

Table 6.3 shows the statistics for these three datasets. As can be seen, the Ontonotes dataset is by far the largest of the three in terms of number of sentences, significantly more distinct tokens than the other two.

Dataset	**Sentences**	**Avg. length**	**Distinct tokens**
PTB	39,831 / 1,699 / 2,415	24	39,547 / 6,264 / 7,726
UD	12,542 / 2,001 / 2,076	15	16,632 / 4,809 / 4946
Ontonotes	115,803 / 15,679 / 12,216	19	54,476 / 20,651 / 16,506

Table 6.3: Dataset statistics, split into training/development/test sets. Distinct tokens are defined as the set of lower-cased word forms present in the FORM column of the CoNLL-U format.

To handle unknown words in the testing data, we preprocessed the training data using a variant of word-dropout [24], as proposed by Kiperwasser and Goldberg [25]: we replace each word with the unknown-word token with a probability inversely proportional to the word's frequency in the dataset. We define the probability of a replacement of word w as

$$p_{\text{unk}}(w) = \frac{\alpha}{\text{count}(w) + \alpha}$$

Unlike Kiperwasser and Goldberg, we do not apply this variant of word-dropout for each training example independently while training the network; we instead apply it on the training dataset as a pre-processing step to produce a new dataset with some words dropped out. This allows us to create dense mini-batches of training examples (i.e. large matrices of inputs and outputs to our neural network) ahead of time, and thus speed up training time significantly. Table 6.4 shows some statistics of performing this form of dropout.

Dataset	**Num. sentences**	≥ 1 **UNK**
PTB	39,831	8,279 (20.79%)
UD	12,542	3,308 (26.38%)
Ontonotes	115,803	12,040 (10.40%)

Table 6.4: Statistics of the word dropout performed on each training dataset.

We re-tagged our testing sets using the Stanford CoreNLP POS Tagger [48, 47], using the pretrained 'english-bidirectional-distsim' model; we chose this model because the documentation stated that it is the most accurate one. Accuracies of the re-tagged test sets are shown in Table 6.5.

Dataset	**UPOSTAG (%)**	**XPOSTAG (%)**
PTB	89.35	97.25
UD	82.11	87.60
Ontonotes	88.65	87.11

Table 6.5: Accuracies of re-tagging the test sets. UPOSTAG and XPOSTAG are referring to the field names of the CoNLL-U format, where XPOSTAGs are more fine-grained than UPOSTAGs.

6.1.1 Converting treebanks to Universal Dependencies

To convert the Penn Treebank dataset into Universal Dependencies, we used the Stanford CoreNLP tools[5], whose method for converting treebanks to Universal Dependencies is described in [40]. The converter uses a set of rules to determine the head of each constituent, and in a depth-first manner builds a dependency graph; afterwards, regular expressions on the tree are defined which are used for labeling the arcs.

6.2 Other parsers

We compared the accuracy and speed of our parser against three other commonly-available parsers: Stanford CoreNLP, MALTParser, and spaCy (versions listed in Table 6.6). We retrained the parsers on each dataset, using the same CoNLL-U formatted files for each.

Parser	**Version**
spaCy	2.0.2
Stanford CoreNLP	3.8.0 (2017-06-09)
MALTParser	1.9.1

Table 6.6: Version numbers of the parsers we compared against.

We trained CoreNLP's `nndep.DependencyParser` tool using the default parameters, and only specified that it must use the coarse-grained UPOSTAG field instead of the default fine-grained XPOSTAG. We will call this model "CoreNLP".

We trained MALTParser using the 'nivrestandard' model using the LIBLINEAR learner, with default parameters. When given a CoNLL-U format file, it will use UPOSTAG field by default for the POS features. We call this model "MALT:ns".

The spaCy parser was trained using its built-in 'train' command using the default parameters; it also uses the Universal Dependencies UPOSTAGs as its POS tag set.

[5]Instructions for converting Penn Treebank to UD are found here: `https://nlp.stanford.edu/software/stanford-dependencies.shtml`

6.3 Results

From our experiments we found that the $lc_1(lc_1(s_i))$ and $rc_1(rc_1(s_i))$ features used by Chen and Manning were of negligible importance to the overall performance of the parser, and due to the complexity of querying the set of arcs, they were among the slowest features to compute; thus, we eliminated these from our parser.

We also found that part-of-speech tags were very important, and accuracy dropped significantly when they were omitted. Interestingly, the accuracy drop was smaller on the UD English dataset than the PTB dataset. We suspect this is due to the lack of professional editing of online content, which may lead to more ambiguous statements, and the increased importance of POS tags to disambiguate meaning.

In the following results, the 'Main' model refers to our model which we described in Section 5.2, without any extensions.

6.3.1 Reduced feature sets

The fewer number of features which must be computed, the faster the parser is, at the expense of accuracy. Some features, however, do not appear important: the accuracy remains the same with or without the grandchild features. However, other features related to existing arcs do seem to be important. These remain much slower to compute than the simple unigram features, so they slow down the parser. They are slower to compute because they require queries to the object storing arcs to return two new lists of (at most) two items for each of s_1 and s_2, whereas the s_1 and s_2 objects store the word and tag features directly as integers (since they do not change during the parsing process). Results for the different feature sets are shown in Table 6.7.

Model	**UAS**	**LAS**	**Speed (w/s)**
Main	80.69	74.25	44,864 (1,912)
Main + grandchildren	80.80	73.99	43,834 (1,545)
Main − arc features	74.89	64.79	60,760 (3,976)

Table 6.7: Results on PTB comparing the main model to the same model including grandchildren features (i.e. $lc_1(lc_1(s_i))$ and $rc_1(rc_1(s_i))$, and to the same model excluding the arc-related features (i.e. $lc_1(s_i)$, $lc_2(s_i)$, $rc_1(s_i)$, $rc_2(s_i)$.). Speed reported is average of 10 runs with standard deviation in brackets.

6.3.2 Scaling labels based on arc direction

We attempted to adjust the label prediction using the transition prediction, such that the probabilities of certain labels change based on the arc direction. However, given the results in Table 6.8, we found no benefit to doing this, as the differences between the different scaling methods are negligible. To simplify the network, it seems best to produce independent output vectors. This result may be due to the fact that all information necessary to predict transition and labels is already present in the hidden layer h, and so there is no benefit to including transition information in the label prediction. This is likely related to the polynomial activation function which already combines many disparate elements in the input vectors in such a way that another linear layer adds no extra information.

Model	**UAS**	**LAS**	**Speed (w/s)**
Main (no scaling)	**80.57**	73.80	**44,294** (2,403)
Main + multiplicative	80.54	73.95	43,665 (1,086)
Main + additive	80.51	73.95	43,757 (1,615)
Main + combined output	80.41	**74.67**	39,190 (1,670)

Table 6.8: Results on PTB comparing the main model with the extensions which modify the label prediction based on the transition prediction using $\odot$ or $+$, respectively, as well as the extension with a combined output vector. Speed reported is average of 10 runs with standard deviation in brackets.

6.3.3 Different activation functions

To confirm Chen and Manning's findings of the cube activation function being better than tanh or sigmoid, we ran experiments with many activation functions, including many different exponents; the results are presented in Table 6.9. We found that the tanh function performs worst, while the quartic function performs best in terms of LAS, although not by much, compared to the other polynomial functions. The quintic function performs better in terms of UAS than quartic (0.34 percentage points better), but it also does worse on LAS by 0.38 percentage points. This leads us to speculate that although combinations of elements from the input vectors are beneficial, there is a limited number of combinations which are useful. This may simply be due to the limited number of features, or due to the size of the model.

6.3.4 Number of hidden layers

Despite numerous experiments with different number of extra hidden layers, and different activation functions, we found no benefit to having more than one hidden layer. Indeed, we found cases in which increased number of hidden layers decrease accuracy. Mathematically, adding an extra hidden layer

$$h_2 = \mathbf{W}h + \mathbf{b}$$

computes a linear combination between the elements of h, which, as can be seen from the equations in Section 4.2.5, already contains many combinations of elements from different parts of the input vectors. Thus, it appears that extra combinations of elements of h do not add any extra value. Table 6.10 shows there results of having multiple hidden layers using the sigmoid activation function.

Activation function	**UAS**	**LAS**
identity (x)	73.99	65.51
quadratic (x^2)	79.70	72.75
cubic (x^3)	80.26	72.80
quartic (x^4)	80.34	**73.40**
quintic (x^5)	**80.71**	73.02
sextic (x^6)	80.55	72.57
relu ($\max(0, x)$)	80.00	71.96
tanh	77.19	69.76
sigmoid	76.88	68.03
tanh-cube ($\tanh(x^3 + x)$)	77.80	70.06

Table 6.9: Accuracy results for the different activation functions as evaluated on the PTB dataset. Different activation functions do not affect parsing speed, thus it is not reported here.

Model hidden layers	**UAS**	**LAS**	**Speed (w/s)**
Main	80.55	73.99	42,961 (1,146)
Main + 1	80.73	73.23	43,450 (1,276)
Main + 2	80.71	73.18	43,314 (761)
Main + 3	80.69	73.31	42,787 (1,043)

Table 6.10: Results on PTB of using multiple hidden layers with all hidden layers after the first using a sigmoid activation. Speeds reported is average of 10 runs with standard deviation in brackets.

6.3.5 Pretrained word embeddings

Model	**UAS**	**LAS**	**Speed (w/s)**
Learned	**80.41**	73.70	**44,102** (1,275)
Pretrained	80.32	**73.90**	43,911 (1,852)

Table 6.11: Results on PTB of learning the word embeddings directly, versus using pretrained GloVe word embeddings and only learning the dimensionality reduction matrix from GloVe vectors to d dimensional vectors. Speed reported is average of 10 runs with standard deviation in brackets.

Table 6.11 shows the results of learning word embeddings, compared to using the pretrained GloVe word embeddings and only learning a linear map into a lower dimension vector space. Despite having a significantly larger vocabulary when using pretrained embeddings, it appears that this does not lead to a significant improvement in accuracy. Neither does it lead to a significant decline in accuracy, so for real-world data where there will be significantly more unknown words, it would be a good idea to use the pretrained word embeddings. The decrease in parsing speed caused by the pretrained embeddings is likely due to the fact that at parsing time, we continue using the original 300-dimensional vectors, and applying the linear map to reduce them to 50 dimensions. Applying the map ahead of time, and storing only the new smaller vectors should close the gap.

6.3.6 Use of BiLSTMs to encode sentence

Model	**UAS**	**LAS**	**Speed (w/s)**	
Main	80.41	73.70	**44,102**	(1,275)
Main + BiLSTM	82.85	76.23	18,515	(195)
Main + POS-BiLSTM	**85.53**	**80.91**	18,493	(241)

Table 6.12: Results on PTB for using an BiLSTM, along with having the BiLSTM be simultaneously trained to produce POS tags. Speed reported is average of 10 runs with standard deviation in brackets.

As can be seen from the results in Table 6.12, using a BiLSTM improves accuracy in terms of both UAS and LAS, however it leads to a significantly reduced parsing speed. Using POS tag prediction as a scaffold task during training leads to a significant accuracy improvement; the decrease in parsing speed is most likely a measurement error as the parsing code is identical to the regular Main + BiLSTM model. This accuracy improvement exists despite the fact that the POS tagging extension was fairly naive (in terms of the loss function used and the use of a simple linear layer to predict tags); a more carefully designed scaffold task may amplify these results. Although at test time the POS tags in the dataset were used for the POS features, the BiLSTM component was trained to produce POS tags and so the embeddings used for the word features include POS information. The network thus has the opportunity to learn which POS tags are more useful.

6.3.7 Compared to other parsers

In Table 6.13 we see a comparison of our best models with spaCy, CoreNLP, and MALTParser. Our main model has a somewhat higher UAS than CoreNLP, but it also has a lower LAS by 1.6 percentage points. It is, however, about 4 times faster than CoreNLP, and our more complex model (with BiLSTM and POS tag prediction) has higher UAS and LAS while having a somewhat faster speed. MALTParser is more than three times faster than spaCy, but is less accurate, and our models are several times faster than spaCy but only more accurate than it on PTB. As spaCy's model effectively contains a POS tagging component within it, it performs equally well regardless of the given POS tags in the test set, while the other parsers' parsing accuracies decrease as POS tag accuracy decreases.

As a comparison to more complex and more accurate models, Table 6.14 shows the results of the SyntaxNet [2] and DRAGNN [26] models with on the PTB and UD datasets. The results are extracted from the papers cited in the table, and as the POS tagger, POS tag set, and label set differs from the ones used in this thesis, the results are not directly comparable.

Dataset	**Parser**	**UAS**		**LAS**		**Speed (w/s)**		**Scale**
		Retag	Gold	Retag	Gold			
PTB	Main	80.41	90.58	73.70	88.48	**44,294**	(2,403)	×1
	Main + POS-BiLSTM	**85.53**	91.67	**80.91**	89.79	18,493	(241)	×2.40
	spaCy 2	84.68	84.68	80.04	80.04	11,446	(189)	×3.87
	CoreNLP nndep	69.68	78.56	63.20	75.84	11,246	(153)	×3.94
	MALT:ns	74.04	86.25	67.35	82.78	36,433	(639)	×1.22
UD	Main	72.21	87.01	62.92	84.04	**39,584**	(1,410)	×1
	Main + POS-BiLSTM	77.46	85.85	69.45	82.43	16,375	(3,058)	×2.42
	spaCy 2	**85.67**	85.67	**80.96**	80.96	7,312	(82)	×5.41
	CoreNLP nndep	66.89	81.42	58.54	78.51	11,242	(181)	×3.52
	MALT:ns	68.03	85.64	59.94	81.64	31,713	(392)	×1.25
Onto-notes	Main	78.73	81.39	73.63	79.40	**44,283**	(1,074)	×1
	Main + POS-BiLSTM	81.23	82.93	76.99	81.31	16,385	(539)	×2.70
	spaCy 2	**89.34**	89.34	**87.39**	87.39	10,012	(83)	×4.42
	CoreNLP nndep	76.36	79.61	72.42	78.28	14,683	(71)	×3.02
	MALT:ns	76.24	86.28	71.53	84.34	31,312	(370)	×1.41

Table 6.13: Comparison of our main model, our most accurate model, and the other parsers. Results are reported on both the re-tagged test sets and the original test sets with the gold POS tags. Speed reported is average of 10 runs with standard deviation in brackets.

As spaCy 1.9.0 uses the fine-grained POS tags (XPOSTAG) in the CoNLL-U files, it is not directly comparable to the other parsers which all use coarse-grained POS tags. However, we trained and tested this version of spaCy against 2.0.2 and present the results in Table 6.15. The newer version of spaCy is much more accurate, but also significantly slower than the previous version.

To better visualize the trade-off between accuracy and speed, Figure 6.1 shows the results from Table 6.13 and Table 6.7 on the PTB dataset. The line connecting our fastest model and our most accurate one illustrates a linear interpolation between these two points; from this it clear that our main feed-forward model performs better than a direct linear interpolation of the other two would.

6.3.8 Training times

Table 6.16 shows approximate training times for the models presented above, and only includes time spent training the neural network, and not any time spent generating training examples (which may be done once as a pre-processing step). The feed-forward models all

Model	**Dataset**	**UAS**	**LAS**
SyntaxNet [2]	PTB	94.61	92.79
Arc-Swift [39]	PTB	94.3	92.2
SyntaxNet[6] [39]	UD English	84.79	80.38
DRAGNN [1]	UD English	87.86	84.45
Arc-Swift [39]	UD English	86.1	82.2

Table 6.14: Results of SyntaxNet, DRAGNN and a newer transition-based system Arc-Swift. All the PTB results used Stanford dependencies, and the UD English results used Universal Dependencies. Results are extracted from the cited papers.

Dataset	**Parser**	**UAS**		**LAS**		**Speed (w/s)**	
		Retag	Gold	Retag	Gold		
PTB	spaCy 1	73.77	74.57	69.57	70.87	32,945	(673)
	spaCy 2	84.68	–	80.04	80.04	11,446	(189)
	Main	80.41	90.58	73.70	88.48	44,294	(2,403)
	Main + POS-BiLSTM	85.53	91.67	80.91	89.79	18,493	(241)
UD	spaCy 1	73.03	77.13	67.58	73.15	25,484	(396)
	spaCy 2	85.67	–	80.96	80.96	7,312	(82)
	Main	72.21	87.01	62.92	84.04	39,584	(1,410)
	Main + POS-BiLSTM	77.46	85.85	69.45	82.43	16,375	(3,058)
Ontonotes	spaCy 1	74.14	77.08	71.64	75.11	22,318	(436)
	spaCy 2	89.34	–	87.39	87.39	10,012	(83)
	Main	78.73	81.39	73.63	79.40	44,283	(1,074)
	Main + POS-BiLSTM	81.23	82.93	76.99	81.31	16,385	(539)

Table 6.15: Comparison of spaCy 1.9.0 and spaCy 2.0.2 against our results from Table 6.13. The accuracy numbers are not directly comparable since spaCy 1 used the fine-grained XPOSTAG features, while spaCy 2 uses the coarse-grained UPOSTAG features. The number in parentheses after the speed is the standard deviation across 10 runs.

took roughly 30 minutes to train on PTB, but the BiLSTM models took over 3 hours. This is due to the fact that the BiLSTM is run for each training example. As the focus of this thesis was on parsing speed, no attempt was made to optimize the training times.

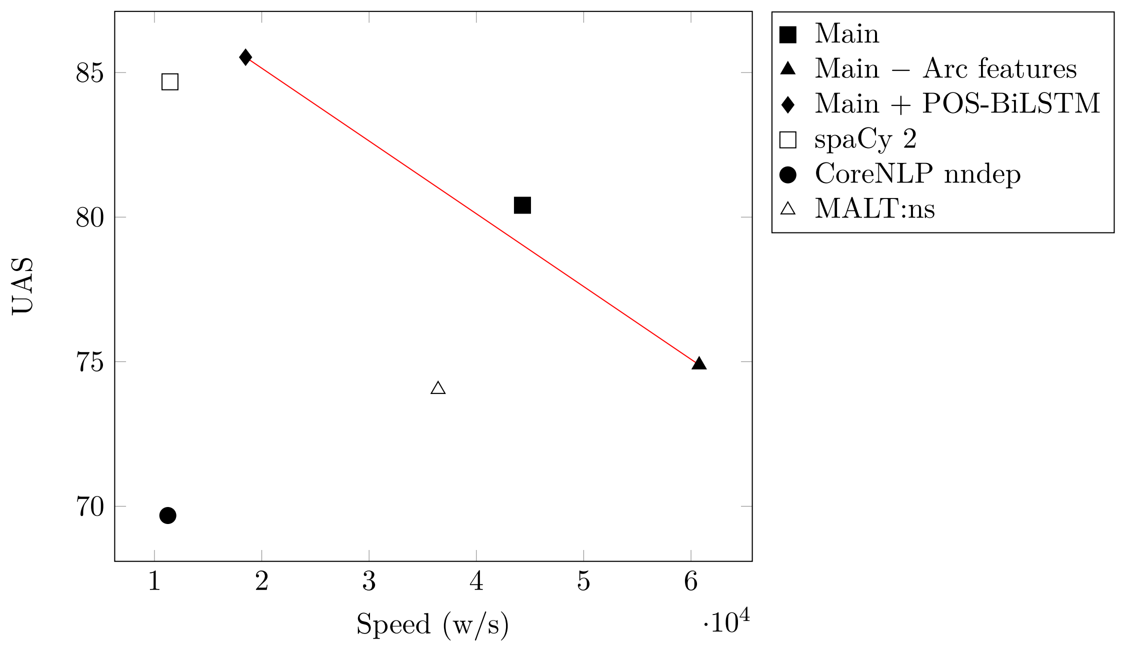


Figure 6.1: A plot combining the results of our main model, our fastest model, our most accurate model, as well as the three other parsers we tested against. These results are on the re-tagged PTB test set.

Model	**PTB training time**
Main	27 mins
Main + grandchildren	33 mins
Main − arc features	33 mins
Main + combined output	37 mins
Main + 1 hidden layer	31 mins
Main + 2 hidden layers	32 mins
Main + 3 hidden layers	32 mins
Main + pretrained embeddings	20 mins
Main + BiLSTM	3 hours 14 mins
Main + POS-BiLSTM	3 hours 16 mins

Table 6.16: Approximate training times of each neural network model on PTB. The training time is defined as the time it took to reach the training step which maximized accuracy on the development set.

Chapter 7

Conclusion

This thesis presents a model and implementation of a dependency parser, along with several extensions which show a trade-off between accuracy and speed. The results show that our main model is significantly faster than other parsers which are advertised as being fast, while having no decrease in accuracy on the Penn Treebank dataset when gold POS tags are provided. Our more complex model which includes a BiLSTM trained to also produce POS tags is more accurate than spaCy on PTB and UD with gold POS tags and much more accurate than the other parsers on the retagged test sets; however, it is much slower than MALTParser. However, across all datasets, it is faster than CoreNLP's nndep parser while simultaneously being more accurate.

We used a profiling-driven approach to investigate and improve bottlenecks in the parser, leading to findings that removing arc-related features would lead to a significant improvement in speed, as would splitting the transition and label predictions into two separate output vectors.

We also showed that the quartic and quintic activation functions perform a little bit better than the cube activation function used by Chen and Manning [6], although all polynomial functions of degree greater than 2 perform similarly well.

In finding a small set of features which are fast to compute, we discovered that the $lc_1(lc_1(w))$ and $rc_1(rc_1(w))$ features included in [6] are unnecessary, but the other dependent-related features are very important and their removal results in an approximately 6% and 10% absolute drop in UAS and LAS, respectively. However, the arc features are relatively slow to compute, and their removal leads to a 33% increase in parsing speed, and the UAS is competitive to spaCy and MALTParser, while being 89% and 68% faster than them, respectively.

Although our main model with a combined output has a very similar structure to CoreNLP's nndep model and has a similar accuracy, our parser is still three times faster, and more importantly, we showed that predicting the next transition and label separately results in a 25% increase in parsing speed while retaining a very similar accuracy.

In terms of using a BiLSTM, it is clear that it is beneficial for accuracy as it is effectively providing a much larger set of features, although its use does lead to a significant decrease in speed. In our experiments, we showed that using POS tagging as a scaffold task when training the BiLSTM results in a significant improvement in accuracy, and this is a task which can be easily implemented in other models which employ a BiLSTM.

In terms of what this means for parsing English Wikipedia (3.35B words [51]), Table 7.1, shows the expected parsing times of 3.35B tokens based on parsing speeds from Table 6.7 and Table 6.13.

Model	**Speed (w/s)**	**Time to parse**
Main	44,294	21 hrs
Main − Arc features	60,760	15 hrs
Main + POS-BiLSTM	18,493	50 hrs

Table 7.1: Expected parsing times for 3.35B words using 3 of the models presented in this thesis. Calculation of the expected time to parse was done by dividing 3.35B by the speed.

7.1 Future work

An area of future work we would like to explore is different methods for training our model to be more resilient to tagging errors and colloquial grammar. This includes performing word-frequency sensitive word-dropout (the same method we used in this thesis) on a per-training-example basis instead of performing it as a pre-processing step. This would lead to a better distribution of dropped words, as well as a higher number of examples with missing words.

To make the model more resilient to incorrect POS tags, we would like to explore introducing errors into the training data, potentially similar to word-dropout. These can be either replacing tags with an UNK tag or random other tag, or based on tags that the POS tagger used would produce. Ideally, we would re-tag the training dataset with the intended POS tagger and then with a certain probability either pick the gold POS tag, or the tagger's POS tag for each token for each training example independently.

As our model (as well as all parsers tested) rely on POS tags as features, they naturally require that the input data is tagged before it is parsed; this simply shifts the bottleneck from the parser to the tagger, and so there is no benefit to making the parser faster than the tagger unless the tagger can also be sped up or removed entirely. Bohnet and Nivre [4] extended the arc-standard set of actions to do joint parsing and tagging by predicting a POS tag as part of the shift action. Our model can be extended with that method by producing a third output vector which generates POS tags, and this should lead to a minimal decrease in speed (as only shift actions would be slightly slowed down) but the complete system would not require a dedicated POS tagger.

Implementation-wise, we would like to allow our parser to use multiple threads to process the outputs of the decision function, thus allowing our parser to truly parse multiple sentences in parallel. Currently, when the single thread is applying the outputs of the neural network to the configuration objects, it does so sequentially, leading to very low GPU utilization. With multiple threads, the parser can apply the decisions to the sentences more quickly and thus lower the percentage of time that the GPU sits unused. An extension of this would be to have a much larger number of sentences on which decisions are applied than the mini-batch size k sent to the neural network; this would mean that when k configuration objects' features are sent to the neural network, other threads are getting another mini-batch of size k ready, and will be able to send the new mini-batch as soon as the neural network produces results for the previous one. This would greatly increase GPU utilization and would allow for more complex neural network models which are slower to compute, as the whole pipeline would spend less time waiting for other components to finish.

An alternative method for parallelizing the application of transitions may be to use different data structures such that SIMD (single instruction, multiple data) or SIMT (single instruction, multiple threads) may be used; currently, it is impossible to apply the same instructions across all the configurations in a mini-batch since different configurations have different best transitions, and so different code executes for each configuration. If the configurations are implemented such that the same transition can be applied across multiple configurations at once, this may allow for the use of SIMD instructions.

Bibliography

[1] Chris Alberti, Daniel Andor, Ivan Bogatyy, Michael Collins, Dan Gillick, Lingpeng Kong, Terry Koo, Ji Ma, Mark Omernick, Slav Petrov, et al. SyntaxNet models for the CoNLL 2017 shared task. *arXiv preprint arXiv:1703.04929*, 2017.

[2] Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. Globally normalized transition-based neural networks. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 2442–2452. Association for Computational Linguistics, 2016.

[3] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, 2011.

[4] Bernd Bohnet and Joakim Nivre. A transition-based system for joint part-of-speech tagging and labeled non-projective dependency parsing. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 1455–1465. Association for Computational Linguistics, 2012.

[5] Daniel Cer, Marie-Catherine de Marneffe, Dan Jurafsky, and Chris Manning. Parsing to stanford dependencies: Trade-offs between speed and accuracy. In Nicoletta Calzolari (Conference Chair), Khalid Choukri, Bente Maegaard, Joseph Mariani, Jan Odijk, Stelios Piperidis, Mike Rosner, and Daniel Tapias, editors, *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, Valletta, Malta, may 2010. European Language Resources Association (ELRA).

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

[7] Jinho Choi. Deep dependency graph conversion in english. In *TLT*, pages 35–62, 2017.

[8] Jinho D Choi and Martha Palmer. Guidelines for the clear style constituent to dependency conversion. *Technical Report 01–12*, 2012.

[9] Hang Cui, Renxu Sun, Keya Li, Min-Yen Kan, and Tat-Seng Chua. Question answering passage retrieval using dependency relations. In *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 400–407. ACM, 2005.

[10] Marie-Catherine De Marneffe, Bill MacCartney, Christopher D Manning, et al. Generating typed dependency parses from phrase structure parses. In *Proceedings of LREC*, volume 6, pages 449–454. Genoa Italy, 2006.

[11] Marie-Catherine De Marneffe and Christopher D Manning. Stanford typed dependencies manual. Technical report, Technical report, Stanford University, 2008.

[12] Marie-Catherine De Marneffe and Christopher D Manning. The stanford typed dependencies representation. In *Coling 2008: proceedings of the workshop on cross-framework and cross-domain parser evaluation*, pages 1–8. Association for Computational Linguistics, 2008.

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

[14] Explosion AI. Facts & figures | spaCy usage documentation. `https://spacy.io/usage/facts-figures#benchmarks`. Accessed: 2017-09-27.

[15] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of machine learning research*, 9(Aug):1871–1874, 2008.

[16] William Foland and James H Martin. Dependency-based semantic role labeling using convolutional neural networks. In ** SEM@ NAACL-HLT*, pages 279–288, 2015.

[17] Katrin Fundel, Robert Küffner, and Ralf Zimmer. RelEx — relation extraction using dependency parse trees. *Bioinformatics*, 23(3):365–371, 2006.

[18] Yoav Goldberg and Michael Elhadad. An efficient algorithm for easy-first non-directional dependency parsing. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 742–750. Association for Computational Linguistics, 2010.

[19] Yoav Goldberg and Joakim Nivre. A dynamic oracle for arc-eager dependency parsing. In *COLING*, pages 959–976, 2012.

[20] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

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

[22] Matthew Honnibal and Ines Montani. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. *To appear*, 2017.

[23] Richard A Hudson. *English word grammar*. B. Blackwell, 1991.

[24] Mohit Iyyer, Varun Manjunatha, Jordan Boyd-Graber, and Hal Daumé III. Deep unordered composition rivals syntactic methods for text classification. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, volume 1, pages 1681–1691, 2015.

[25] Eliyahu Kiperwasser and Yoav Goldberg. Easy-first dependency parsing with hierarchical tree lstms. *TACL*, 4:445–461, 2016.

[26] Lingpeng Kong, Chris Alberti, Daniel Andor, Ivan Bogatyy, and David Weiss. DRAGNN: A transition-based framework for dynamically connected neural networks. *arXiv preprint arXiv:1703.04474*, 2017.

[27] Marco Kuhlmann, Carlos Gómez-Rodríguez, and Giorgio Satta. Dynamic programming algorithms for transition-based dependency parsers. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pages 673–682. Association for Computational Linguistics, 2011.

[28] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

[29] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The Penn Treebank. *Computational linguistics*, 19(2):313–330, 1993.

[30] Ryan McDonald, Koby Crammer, and Fernando Pereira. Online large-margin training of dependency parsers. In *Proceedings of the 43rd annual meeting on association for computational linguistics*, pages 91–98. Association for Computational Linguistics, 2005.

[31] Joakim Nivre. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*. Citeseer, 2003.

[32] Joakim Nivre. Incrementality in deterministic dependency parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*, pages 50–57. Association for Computational Linguistics, 2004.

[33] Joakim Nivre, Johan Hall, and Jens Nilsson. MaltParser: A data-driven parser-generator for dependency parsing. In *Proceedings of LREC*, volume 6, pages 2216–2219, 2006.

[34] Joakim Nivre, Johan Hall, Jens Nilsson, Atanas Chanev, Gülşen Eryigit, Sandra Kübler, Svetoslav Marinov, and Erwin Marsi. MaltParser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13(2):95–135, 2007.

[35] Franz Josef Och, Daniel Gildea, Sanjeev Khudanpur, Anoop Sarkar, Kenji Yamada, Alex Fraser, Shankar Kumar, Libin Shen, David Smith, Katherine Eng, et al. A smorgasbord of features for statistical machine translation. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics: HLT-NAACL 2004*, 2004.

[36] Wenzhe Pei, Tao Ge, and Baobao Chang. An effective neural network model for graph-based dependency parsing. In *ACL (1)*, pages 313–322, 2015.

[37] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. GloVe: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[38] Sameer Pradhan, Alessandro Moschitti, Nianwen Xue, Olga Uryupina, and Yuchen Zhang. CoNLL-2012 shared task: Modeling multilingual unrestricted coreference in OntoNotes. In *Proceedings of the Sixteenth Conference on Computational Natural Language Learning (CoNLL 2012)*, Jeju, Korea, 2012.

[39] Peng Qi and Christopher D Manning. Arc-swift: A novel transition system for dependency parsing. *arXiv preprint arXiv:1705.04434*, 2017.

[40] Sebastian Schuster and Christopher D Manning. Enhanced English Universal Dependencies: An improved representation for natural language understanding tasks. In *LREC*, 2016.

[41] Dan Shen and Mirella Lapata. Using semantic roles to improve question answering. In *Emnlp-conll*, pages 12–21, 2007.

[42] Maryam Siahbani, Ravikiran Vadlapudi, Max Whitney, and Anoop Sarkar. Knowledge base population and visualization using an ontology based on semantic roles. In *Proceedings of the 2013 workshop on Automated knowledge base construction*, pages 85–90. ACM, 2013.

[43] Natalia Silveira, Timothy Dozat, Marie-Catherine de Marneffe, Samuel Bowman, Miriam Connor, John Bauer, and Christopher D. Manning. A gold standard dependency corpus for English. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC-2014)*, 2014.

[44] Anders Søgaard and Yoav Goldberg. Deep multi-task learning with low level tasks supervised at lower layers. In *ACL*, 2016.

[45] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958, 2014.

[46] The Scipy community. numpy.argsort — NumPy v1.13 Manual. `https://docs.scipy.org/doc/numpy/reference/generated/numpy.argsort.html`.

[47] Kristina Toutanova, Dan Klein, Christopher D Manning, and Yoram Singer. 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*, pages 173–180. Association for Computational Linguistics, 2003.

[48] Kristina Toutanova and Christopher D Manning. Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In *Proceedings of the 2000 Joint SIGDAT conference on Empirical methods in natural language processing and very large corpora: held in conjunction with the 38th Annual Meeting of the Association for Computational Linguistics-Volume 13*, pages 63–70. Association for Computational Linguistics, 2000.

[49] Universal Dependencies contributors. Universal Dependencies - CoNLL-U Format. `http://universaldependencies.org/format.html`.

[50] Ravikiran Vadlapudi, Maryam Siahbani, Anoop Sarkar, and John Dill. Lensing-wikipedia: Parsing text for the interactive visualization of human history. In *Visual Analytics Science and Technology (VAST), 2012 IEEE Conference on*, pages 247–248. IEEE, 2012.

[51] Wikipedia. Wikipedia:size comparisons — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Wikipedia:Size_comparisons&oldid=806014528`, 2017. [Online; accessed 20-Oct-2017].

[52] Wikipedia. Wikipedia:size of wikipedia — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Wikipedia:Size_of_Wikipedia&oldid=803182643`, 2017. [Online; accessed 02-Oct-2017].

[53] Matthew D Zeiler. ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[54] Hao Zhang and Ryan McDonald. Generalized higher-order dependency parsing with cube pruning. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 320–331. Association for Computational Linguistics, 2012.

[55] Yuan Zhang and David Weiss. Stack-propagation: Improved representation learning for syntax. *CoRR*, abs/1603.06598, 2016.