DEEP LEARNING APPLICATIONS FOR TRANSITION-BASED DEPENDENCY PARSING

by

MOHAB ELKAREF

A thesis submitted to
University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
University of Birmingham
January 2018

Abstract

Dependency Parsing is a method that builds dependency trees consisting of binary relations that describe the syntactic role of words in sentences. Recently, dependency parsing has seen large improvements due to deep learning, which enabled richer feature representations and flexible architectures. In this thesis we focus on the application of these methods to Transition-based parsing, which is a faster variant. We explore current architectures and examine ways to improve their representation capabilities and final accuracies. Our first contribution is an improvement on the basic architecture at the heart of many current parsers. We show that using Recurrent Neural Network hidden layers, initialised with pre-trained weights from a feed forward network, provides significant accuracy improvements. Second, we examine the best parser architecture. We show that separate classifiers for dependency parsing and labelling, with a shared input layer provides the best accuracy. We also show that a parser and labeller can be successfully trained separately. Finally, we propose Recursive LSTM Trees, which can represent an entire tree as a single dense vector, and achieve competitive accuracy with minimal features. The parsers that we develop in this thesis cover many aspects of this task, and are easy to integrate with current methods.

ACKNOWLEDGEMENTS

I would like to sincerely thank my supervisor Bernd Bohnet, for his guidance, unwavering support, and endless patience. Were it not for him, I would not have been able to make the jump to Natural Language Processing, and from there to go onto deep learning. I have learnt much from him over the past four years, and have come to see him not only as my mentor, but as my friend. I would also like to thank my co-supervisors Mark Lee and John Barnden, for their support, countless interesting discussions, and invaluable feedback on my work and general research.

I have been lucky to come across so many wonderful people in Birmingham. I would especially like to thank Tomáš Jakl and Lenka Mudrová, for their friendship, encouragement, and constant support. I would like to thank Harish Tayyar Madabushi for always making time for interesting discussions and feedback on my ideas, and Andrew Gargett for always taking interest in my research and encouraging me to explore further applications. My friends both within the School of Computer Science and without are what helped make Birmingham feel like home, and I would like to thank Eva Reindl, Bram Geron, Mohamed Samra, Cory Knapp, Khulood AlYahya, Elisa Lauricell and Ruşen Aktaş. My time here so far away in Birmingham has not stopped Mohamed Hatem and Hassan Selim from making home seem no further than an instant away.

And finally, nothing I do is possible without my parents. There is nothing that I could ever write that can describe how much I owe them, how much I have learnt from them, or begin to thank them for it all. I can only hope to make them proud, and that this thesis is one step closer towards that goal.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER 1

INTRODUCTION

Dependency parsing is a central task in Natural Language Processing (NLP), which aims to represent the syntactic structure of sentences. It builds representations of sentences by assigning asymmetric parent/child annotations, known as dependency relations, to pairs of words that reflect the syntactic and, to some extent, semantic relationship between any given pair of words. Each word in a given sentence is assigned exactly one dependency relation for which it is the child, in addition to all that word's dependency relations with its own child words, culminating in a directed dependency tree that preserves word order and also describes the syntactic structure within the sentence. In addition, dependency parsing has the ability to attach words freely to nodes within a tree, which makes it easier to learn more flexible formulations in languages with free word order (Melčuk, 1988). An example of a sentence represented as a dependency tree is shown in Figure 1.1.

Moreover, dependency parsing is a useful component in other major NLP applications such as language modelling (Gubbins and Vlachos, 2013), relation extraction (Mausam et al., 2012; Angeli et al., 2015), and semantic parsing (Surdeanu et al., 2008; Hajič et al., 2009; Parikh et al., 2015). Because of the usefulness of dependency grammar, much effort has been devoted to further analysing its strengths and limitations, as well as exploring possible options for optimising and improving the accuracy of dependency parsing systems.

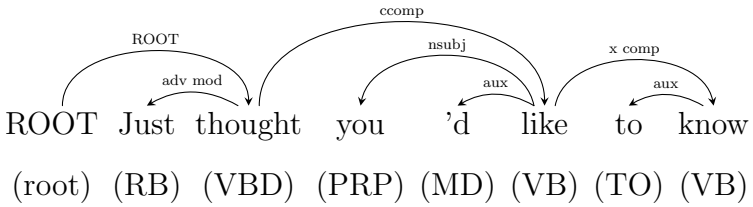


Figure 1.1: An example of a dependency tree representation of a sentence. ROOT is an additional word conventionally added to all sentences. It acts as a placeholder for the head of the actual root word of a sentence (in this case the root word is "thought"). The Part-of-Speech (pos) tag of each word is shown between parentheses below each word. Each arrow is a dependency arc, a directed dependency relation, going from the parent word to the child word. The type of relationship is indicated by the dependency labels on the arcs[1]. Note how all words have exactly one parent word, and how the whole arrangement culminates in a directed tree with the top of the tree having the extra ROOT token as its parent.

The pace and performance of dependency parsing has been improving over the years, largely due to a wave of competing statistical parsers using a variety of machine learning techniques (Yamada and Matsumoto, 2003; Zhang and Clark, 2008; Huang and Sagae, 2010b; Zhang and Nivre, 2012; Bohnet and Nivre, 2012) and vast amounts of annotated corpora as training data. These parsers were able to provide flexible language independent methods to model relationships between words based on their occurrence, their surrounding words, and their Part-of-Speech tags, and correctly identify these relationships.

The features of the sentences and words used by machine learning algorithms to achieve this were usually hand-crafted and could number in the hundreds (Bohnet, 2010). This challenge of feature crafting complements a core strength of Deep Learning, and so has led to a wave of investigation into it as a possible alternative, and achieving impressive results (Chen and Manning, 2014; Weiss et al., 2015; Andor et al., 2016; Dozat and Manning, 2016; Kuncoro et al., 2016a). A deep neural network consisting of hidden layers of neurons determines the most useful combinations of raw features, thus shifting the burden from feature crafting towards network architecture crafting for classification tasks.

Regardless of the learning method used, the most common statistical approach to dependency parsing relies on supervised machine learning techniques coupled with parsing

[1]For simplicity we will not show the dependency labels and pos tags in any of the examples in the rest of this thesis.

algorithms. This avoids the problem of needing to express our complex use of language in a rule-based manner, with new rules needing to be developed for each language. The machine learning model sees a sentence as a set of possible dependency trees that are scored based on their correctness, the highest scoring of which is presented as the correct dependency tree. This approach requires a large number of correctly labelled sentences as training data, and much effort has been dedicated to building large treebanks for various languages (Buchholz and Marsi, 2006; Nivre et al., 2007; Mcdonald et al., 2013).

Of the various parsing algorithms to have emerged, two families remain dominant and successful. The **Graph-based** parser (Eisner, 1996; McDonald et al., 2005a), represents a graph of all the possible arcs in the input sentence, weighing all the arcs, and choosing the highest scoring tree. This approach explores an exhaustive list of possible trees making it very slow, but capable of producing state of the art results (Cheng et al., 2016; Hashimoto et al., 2016; Kiperwasser and Goldberg, 2016b; Dozat and Manning, 2016).

The other main approach is the **Transition-based** parser (Nivre, 2003b; Yamada and Matsumoto, 2003), which treats the parsing problem as a state-machine, with the words of a sentence being input, and a stack (or a variation on the structure) being used to manipulate input and attach arcs. This method attempts to parse the entire sentence in one pass making it faster and more efficient, while still achieving competitive accuracies compared with other approaches (Weiss et al., 2015; Andor et al., 2016; Kuncoro et al., 2016a; Shi et al., 2017).

The application of deep learning to the dependency parsing task has evolved rapidly and increased in complexity. Simple, but effective parsers such as (Chen and Manning, 2014) used a feed forward neural network for feature representation, hidden state encoding, and classification. This was soon improved upon by increasing the size and number of hidden layers in the network, and combining their output with previously successful linear methods (Weiss et al., 2015), or by applying global training methods (Andor et al., 2016). Other works approached the task differently, instead focusing on creating more flexible and richer feature representation using recursive or recurrent representation (Dyer et al.,

2015; Kiperwasser and Goldberg, 2016a,b). Yet another direction involves improving or reexamining the classification layer itself (Cross and Huang, 2016; Dozat and Manning, 2016).

1.1 Goals & Contributions

Our goal in this thesis is to explore the application of deep learning specifically to the transition-based parsing task. We approach this topic from multiple angles, as has been the case with current literature in general. We attempt to improve already established methods by extending their hidden state modelling capabilities. We additionally scrutinise the basic formulation of the transition-based parsing task. And finally, we consider feature modelling, and deep learning's promise of no feature-crafting.

Thus we cover some of the different aspects of the Transition-based parsing task, and produce some useful insights that can be extended to improve existing architectures. Our contributions are also capable of integrating with other important state-of-the-art techniques, and in some cases are even potentially applicable to other tasks in Natural Language Processing.

1.2 Thesis Questions

To address our stated goals, the work in this thesis focuses on answering the following questions:

1. Does the configuration of a parser at one point in the parsing process hold information that is useful to making transitions later in the parsing sequence?

2. What is the best structure for the classification task? How does this influence the architecture of the neural networks used?

3. Given deep learning's ability to learn important features and combinations from context, how can this ability be used to increase the expressiveness of features? And if this is possible, does this expressivness hold with fewer features?

1.3 Structure of this Thesis

Chapter 2 discusses background topics. We describe the key concepts of Transition-based dependency parsing, notable transition systems, and our choice of Arc-Standard, and the motivation to do so.

We additionally provide a basic overview of neural networks and how they are trained, in addition to architecture varieties and their current application to transition-based dependency parsing.

In **Chapter 3** we focus more closely on the various approaches in the literature towards using neural networks for transition-based parsing. We consider the relations between these different works and how they solve different aspects of the dependency parsing task, as well as how they relate to our work in this thesis.

In **Chapter 4** we explore an extension of the simple feed forward neural network parser presented by Chen and Manning (2014) using Long Short-Term Memory networks. We additionally propose a simple alternative method to initialise Recurrent Neural Networks in this architecture, and explore its effects on different RNN variants.

In **Chapter 5** we examine the basic classification task underlying current Transition-based parsers, with a specific focus on the effects of Hierarchical classification as observed by Cross and Huang (2016).

We also propose a number of amended architectures to further determine the extent of the relation between dependency parsing and arc labelling, including training separate networks for each sub-task.

In **Chapter 6** we shift our attention to feature modelling. We propose a Recursive LSTM Tree model that is capable of representing whole dependency trees, and is capable of

integrating with, and retaining the information from, other successful methods of feature representation. We finally explore the minimal feature set necessary for such a robust model to train successfully.

The thesis ends with concluding remarks in **chapter 7**, where we discuss the implications of our contributions and how they could be built upon in future work. We also revisit the questions raised in Section 1.2 and discuss to what extent they have been answered.

CHAPTER 2

BACKGROUND

2.1 Dependency Grammar

Binary dependency relations are a feature used in many languages and is a broad field in itself. Similar ideas can be traced as far back as the middle ages with an example the *primium-secundum* relation (Covington, 1984). However, in its current form dependency grammar is largely attributed to the work of Tesnière (1959) and was then used extensively to model languages with free word order (Melčuk, 1988). In this Section we will only state the basic formalisms needed for the dependency parsing task, and do not aim to examine the linguistic background of dependency grammars themselves.

2.1.1 Components of a Dependency Structure

Dependency grammars focus on representing the relationships between the words in a sentence. These relationships, or dependencies, are organised in such a way as to express the role words in a sentence with respect to each other. Formally, the dependencies within a sentence S consisting of words $w_1, ..., w_n$ can be represented as a **directed graph** $G = (V_S, A)$, such that

- $V_S = \{0, 1, ..., n\}$ is a set of nodes.
- $A \subseteq V_S \times V_S$ is a set of directed arcs representing dependencies.

- A dependency arc from $i \rightarrow j$ represents a **head** w_i to its **dependent** w_j.
- A dependency arc from $i \rightarrow j$ has a label $l_{i,j} \in L$, where L is the set of possible dependency labels.

Node 0 in V_S is the root node. The sentence does not require this additional node in order for its dependencies to be fully represented, and as such it is absent from many formal definitions. The addition of this extra node, however, is useful for parsing systems, especially Transition-based parsers which will be introduced later. Nodes 1 through n correspond to $w_1 ... w_n$ in the order in which they appear in S.

The set of arcs contains pairs of nodes (i, j). The relationship between them is given a variety of names in the literature, head/governor/parent and modifier/dependent/child. We generally use head/dependent, however both sets are used interchangeably throughout the literature.

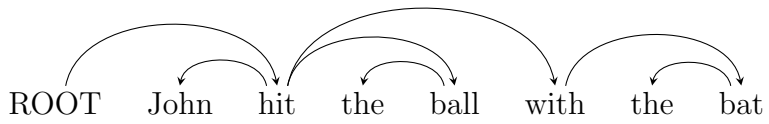


Figure 2.1: An example of a dependency tree from (McDonald et al., 2005b)

2.1.2 Constraints on a Dependency Structure

A dependency graph must satisfy a number of constraints in order to be considered valid. These are not always necessary to represent the structure of a sentence fully, but they remain important widely held assumptions necessary for parsing systems. These constraints are:

Acyclicity There graph cannot include a directed cycle.

Connectedness There can be no isolated part within the graph.

Single governance Every word must be a dependent of exactly one head.

Root governance Every graph must have a word act as root, with no head.

As stated earlier, the *root governance* constraint is modified by some systems to have the word acting as root become a dependent of an additional *root* node. This approach is adopted here as well. In addition, the *acyclicity* constraint limits valid dependency graphs to trees while *connectedness* guarantees each sentence be represented as a single tree and not a forest. For this reason, they will be referred to here as **dependency trees**, trees, and graphs interchangeably. An example of a tree fulfilling all the characteristics and constraints described is show in Figure 2.1.

2.1.3 Projectivity

An additional complication that arises is **projectivity**. Generally, a tree is considered projective if it has no crossing dependency arcs. Non-projective sentences are more common in languages with free-word-order, but they also occur in English, especially in longer sentences. Many dependency parsers cannot successfully parse non-projective sentences and so place as an additional constraint that sentences be projective.

The example shown in Figure 2.2 illustrates this phenomenon. The arc from *dog* $\rightarrow$ *was* crosses the arc *saw* $\rightarrow$ *yesterday*. This is not the case for the tree in Figure 2.1.

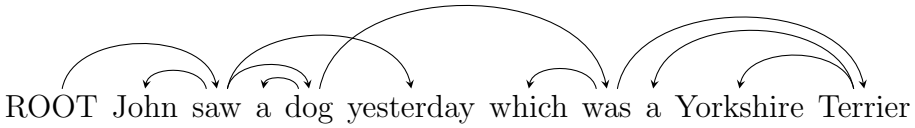


Figure 2.2: An example of a non-projective tree from (McDonald et al., 2005b)

2.1.4 The Parsing Process

Now that we have defined the representation of dependencies within a sentence, the task of parsing is simply to correctly predict these dependencies between words, and decide which type of relation each dependency is (the label on a dependency arc). The possible

combinations of $V_S \times V_S$ can be rather large, but are narrowed down when constraints such as acyclicity are taken into consideration.

2.2 Transition-based Dependency Parsing

A transition-based system processes an input sentence one word at a time in order of appearance, with the goal of building a dependency tree describing its syntactic structure as discussed in Section 2.1. It represents a state machine that attaches a dependency arc to the current word or stores it until a word that can be attached is reached. The decision to attach or store a word, to move from one state to the next, is made based on features of the input word, its part-of-speech tag, and other features of the sentence and transition system itself.

When training, the parsing system is presented with a correctly annotated set of sentences (often referred to as the gold set) that the system can use to learn the correct transitions to make, and to model the relation of the transitions with the various features. The underlying machine-learning algorithm then produces a statistical model that is used by the parser to build dependencies for blank sentences. When parsing an input sentence, this model is used to score each possible transition and usually the option receiving the highest score is chosen. This is often referred to as a greedy parser, and is the approach we will be using throughout our work.

2.2.1 Structure of a Transition System

The approach as described above may be seen as a more flexible form of shift-reduce parsing (Nivre, 2008), with the added complexity of natural language as opposed to programming languages, which are perhaps the usual use cases for such an approach. A defining advantage of this system is that it builds dependencies for the whole sentence in a single pass, without the need for backtracking, resulting in an efficient runtime and minimal use of memory. A pitfall, however, is the issue of error-propagation. This is

caused by incorrect transitions made early on in the sentence, which then affect decisions made later in the process as well (McDonald and Nivre, 2007).

In general the state of a transition system is represented as a configuration $c \in C$, where C is the set of all possible configurations. The state changes from one configuration to the next with the application of a transition $t \in T$, where T is the set of all possible transitions and $t : C \rightarrow C$. The whole transition system is thus defined as $S = (C, T, c_s, C_t)$ where:

- C is the set of possible configurations in the system.
- T is the set of all possible transitions defined by the parsing strategy.
- $c_s \in C$ is the initial configuration of the system.
- $C_t \subseteq C$ is the set of all terminal nodes.

And so given a transition system S, an input sentence $x = (w_1, ..., w_n)$, and a transition prediction function[1] $o : C \rightarrow T$, the complete dependency graph G_c for x can be built using algorithm 1.

Algorithm 1 Basic Parsing Algorithm

```
1: function PARSE(x)
2:     c ← c_s(x)
3:     while c ∉ C_t do
4:         t ← o(c)
5:         c ← t(c)
6:     return G_c
```

2.2.2 Examples of Transition Systems

The data structures constituting a configuration depend on the transitions specified by the parsing system. A parsing system specifies possible transitions, their effects on a

[1] usually referred to as the **oracle**

configuration, and preconditions limiting the choice of transition. We list below examples of popular parsing strategies.

Arc-standard parsing

Arc-Standard dependency parsing (Nivre, 2004) is a system that builds projective dependency trees in a bottom-up approach. The structures defined by this strategy are:

- **Stack(σ):** (A first-in-last-out structure) acts as storage for words that are not yet attached, with the head on the right of the structure. For example $(\sigma|i)$ represents a stack with the i^{th} word of the sentence as its top item.

- **Buffer(β):** (A first-in-first-out structure) consists of the words in x in order of appearance with the head on the left of the structure. For example $(i|\beta)$ represents a buffer with the i^{th} word of the sentence as its front item.

- **Arcs(A):** is a set of arcs from i to j with label l, (i, l, j).

This parsing algorithm produces dependency trees by building dependency arcs between the top two items on its stack. It does not necessarily build arcs as soon as they become possible, but instead only builds an arc when the dependent word has had all its dependents attached throughout the sentence. And so if the top two items on the stack are not related by a dependency relation, or the potential dependent still has children in the buffer, the next word in the buffer is pushed onto the stack instead. The Transitions are defined as:

$Left - Arc_l$**:** $(\sigma|i|j, \beta, A) \Rightarrow (\sigma|j, \beta, A \cup (j, l, i))$

$Right - Arc_l^s$**:** $(\sigma|i|j, \beta, A) \Rightarrow (\sigma|i, \beta, A \cup (i, l, j))$

$Shift$**:** $(\sigma, i|\beta, A) \Rightarrow (\sigma|i, \beta, A)$

Preconditions of building an arc are:

$Left - Arc_l$**:**

$\neg[i = 0]$

The Left/Right-Arc transitions operate solely on the top two items in the stack, with the dependent being popped from the stack. This means that after an arc is built, no more children can be attached to the dependent, hence the constraint to only attach a head to a word once all its dependents are attached. The only precondition to building an arc is for the Left-Arc transition, which cannot have the ROOT token as the dependent.

Arc-eager parsing

Arc-Eager parsing (Nivre, 2003a) has a lot in common with the Arc-Standard system. It uses the same data-structures as Arc-Standard, but instead aims to attach the heads of a token as soon as possible. Arc-Eager adds a new transition, *Reduce*, which pops the top item on the stack, for when a word has all its children and its head attached. The transitions are:

$Left - Arc_l$**:** $(\sigma|i, j|\beta, A) \Rightarrow (\sigma, j|\beta, A \cup (j, l, i))$

$Right - Arc_l^e$**:** $(\sigma|i, j|\beta, A) \Rightarrow (\sigma|i|j, \beta, A \cup (i, l, j))$

$Shift$**:** $(\sigma, i|\beta, A) \Rightarrow (\sigma|i, \beta, A)$

*Reduce***:** $(\sigma|i, \beta, A) \Rightarrow (\sigma, \beta, A)$

Preconditions of building an arc are:

$Left - Arc_l$**:**

$\neg[i = 0]$

$\neg\exists k \exists l'[(k, l', i) \in A]$

*Reduce***:**

$\neg\exists k \exists l[(k, l, i) \in A]$

Arc-Eager Left/Right-Arc transitions build arcs between the top item on the stack and the front item on the buffer. The Left-Arc transition pops the top item on the stack after the arc is built, since Arc-Eager attaches arcs as soon as possible, and so the popped item would already have all its children attached in a projective sentence. The Right-Arc transition, on the other hand, moves the front item of the buffer onto the stack after the arc is built.

The preconditions for Left-Arc are that the top item on the stack is not the ROOT token and does not have a head already attached. The Reduce transition, on the other hand, requires the top item on the stack to already have a head token attached.

List-based parsing

The list based approach introduced by (Covington, 2001) was a basis adapted later in the development of Nivre's algorithms (Nivre, 2008). It follows a familiar structure with the difference being two list based memory structures instead of the stack. This enabled both backtracking and in a subsequent version of this algorithm was used to deal with non-projectivity. This, however, led to the algorithm having a complexity of $O(n^3)$ as a worst case.

Nivre's arc-standard and arc-eager strategies left out backtracking by design, and so were much faster, having a worst case complexity of $O(n)$. The two approaches in the form stated above still required the projectivity constraint, but later dealt with the issue using *pseudo-projective parsing*, where non-projective trees were transformed into projective equivalents, then transformed back afterwards. A subsequent version of these algorithms introduced the *Swap* transition, which could exchange the position of the top two elements on the stack, allowing Nivre's Algorithms to deal with non-projectivity without pseudo-projective transformation.

Other systems and approaches

There are other widely used transition systems, with varying levels of complexity, and different features. Notable examples include Arc-Hybrid (Yamada and Matsumoto, 2003; Gómez-Rodrıguez et al., 2008; Kuhlmann et al., 2011b), which shares much in common with Arc-Standard, but with the Left-Arc transition being built with the front of the buffer b_0 as head, and the top of the stack s_0 as the dependent, forcing all left dependents to be added before right dependents. Another example is Easy-first parsing (Goldberg and Elhadad, 2010), which maintains a list of remaining words in a sentence, but is not constrained to access these words in order, yet still builds dependency trees in a bottom up fashion.

2.2.3 Our choice of Transition-system

For our investigations in this thesis we chose to use the **Arc-Standard** system as described in Section 2.2.2. This choice was made both for simplicity and due to certain properties that Arc-Standard has that combine well with our work on a whole range of neural network architectures, from the simple to the increasingly complex.

The primary feature that motivates our decision is the determinism that can be imposed on Arc-Standard without harming the final performance. Given that the rules of Arc-Standard already restrict parsing to a bottom-up strategy, that only allows the attachment of heads if all children are attached, the only remaining uncertainty is whether to apply a Left-Arc transition first, or to apply a Shift followed by a Right-Arc. This scenario assumes that the word at the top of the stack s_0 still has an unattached child as the second element of the stack s_1 and a right child at the front of the buffer b_1, and potentially more children beyond them. In this case there appears to be multiple paths that the parser can take towards the correct outcome, complicating the learning task.

To simplify the learning task, we implement a *static oracle* that imposes a strict order of transition application, where all left children are added first when possible. This

translates to giving precedence to the Left-Arc transition over the Shift transition when generating the gold sequence of decisions for a tree. The result is a single correct sequence for any given sentence. This set-up is particularly well suited for neural networks, on which we provide a brief introduction in Section 2.3, as they require pairs of input/expected output as training data, and are more suited for atomic decisions by default.

The Arc-Standard system has a lot in common with other competing approaches, such as the Arc-Eager and Arc-Hybrid methods. These latter methods, however, are usually trained with the help of *dynamic oracles* (Goldberg and Nivre, 2013), which explore possible future choices during train time. This adds an extra layer of complication, both regarding implementation and final analysis. We note, however, that these are commonly used approaches in the literature, even among some of the leading neural network-based parsers with which we compare our results. Their performance, on the other hand, remains comparable to our Arc-Standard based models, or can sometimes even fall behind. This can be seen in the comparisons of our results with competing systems throughout our work, or in more direct empirical comparisons such as in Shi et al. (2017).

There are more fine-grained aspects where our choice of transition system also provides an advantage, but we explain these in subsequent chapters as they become relevant in context.

2.2.4 Application & Evaluation

In order to paint a clearer picture of just how this transition system is used, and what it is that we need a machine learning system to model, we give here an example of building a dependency tree using our chosen system, Arc-Standard.

The example parse shown in Table 2.1 highlights a few aspects of the parsing process that were mentioned before. First, the initial configuration of the parser has a stack with only ROOT in it, and a buffer with all the words in the sentence in order. Parsing then terminates when the buffer is empty and ROOT is the only remaining element in the stack once more.

Second, is the role that our static oracle plays. In Section 2.2.3 we explain the ambiguity that can arise if a given word has both right and left dependents that are not yet attached, and we resolved this ambiguity by giving priority to the Left-Arc transition over a Shift transition whenever both are possible.

Consider row 3 and rows 4-7 in Table 2.1. Both "John" and "ball" are dependents of the word "hit", and at the third step the parser has the choice of whether to immediately attach "John" with a Left-Arc transition, or to pursue "ball" first. Had we not put the additional constraint of preferring Left-Arc over Shift, the parser could have also produced the correct dependency tree by perfoming steps 4-7, and indeed steps 8-13, before step 3.

Another thing to note, in the same ranges mentioned, is that children of hit are not attached until their children are attached first. This is especially apparent from step 9 onwards, where the stack consists of [ROOT, hit, with], each word being a dependent of the one beneath it on the stack. And yet the head of "with" is not attached until after all its dependents have been added in step 13, and "hit" is not attached to ROOT until step 14.

There are many statistics that can be calculated to assess the accuracy of a predicted dependency tree after it has been built. But in this work (and in much of the current literature) we rely on two metrics, Unlabelled Attachment Score (UAS), and Labelled Attachment Score (LAS). UAS is a measure of the percentage of arcs that have been correctly attached from head to dependent. LAS is a measure of the percentage of arcs that are both correctly attached *and* have the correct dependency label assigned. It has become a de facto standard to report these two metrics excluding punctuation, and we follow this convention in this thesis as well.

#	Stack	Buffer	Transition	Tree
1	[ROOT]	[John, hit, the, ball, with, the, bat]	Shift	
2	[ROOT, John]	[hit, the, ball, with, the, bat]	Shift	
3	[ROOT, John, hit]	[the, ball, with, the, bat]	Left-Arc	
4	[ROOT, hit]	[the, ball, with, the, bat]	Shift	
5	[ROOT, hit, the]	[ball, with, the, bat]	Shift	
6	[ROOT, hit, the, ball]	[with, the, bat]	Left-Arc	
7	[ROOT, hit, ball]	[with, the, bat]	Right-Arc	
8	[ROOT, hit]	[with, the, bat]	Shift	
9	[ROOT, hit, with]	[the, bat]	Shift	
10	[ROOT, hit, with, the]	[bat]	Shift	
11	[ROOT, hit, with, the, bat]	ϕ	Left-Arc	
12	[ROOT, hit, with, bat]	ϕ	Right-Arc	
13	[ROOT, hit, with]	ϕ	Right-Arc	
14	[ROOT, hit]	ϕ	Right-Arc	
15	[ROOT]	ϕ	Terminate	

Table 2.1: An example of parsing a whole sentence using Arc-Standard. For each row, the Stack & Buffer columns represent the configuration of the parser *before* the transition is applied, while the dependency tree shown in the Tree column shows the effect of the transition (if any).

2.3 Neural Networks

Neural networks have seen wide adaptation in NLP as a consequence of successes in areas such as computer vision, handwriting recognition, and speech recognition. Neural networks learn to perform classification tasks by creating statistical models defining higher level features as weighted combinations of lower level features from raw input. The problem is simplified, somewhat, in that it becomes about learning an optimal configuration of weights over a number of levels of varying size between the raw input and the desired output.

Outside of neural networks, learning such deep structures required hand-crafting many features. This process is both time-consuming and often inadequate, needing hundreds of features that may not properly represent the parameters of the problem completely or over-specify them leading to over-fitting. Neural networks offer the option of automatically inferring such features leading to more generalisable solutions and models that are easier to train.

In addition, a development that has invited attention to this technique in NLP is the introduction of dense word vector representations (Collobert et al., 2011; Mikolov et al., 2013). This meant that the need to represent individual words or word counts discretely was no longer an issue, and that the problem of encountering words not in the original training set was alleviated to an extent. This problem of sparse data, or what is called the curse of dimensionality, was now solved and large input layers were no longer required for many NLP applications.

Neural networks have managed to obtain state-of-the-art results in subfields of NLP such as part-of-speech-tagging and named entity recognition (Collobert and Weston, 2008), and have recently matched or surpassed the performance of handcrafted systems that rely on SVM and MaxEnt models for dependency parsing. This will be discussed further, later in Section 2.5.

In this chapter will only explain the structure of a basic neural network, with additional information for evolved structures being explained along with their application in

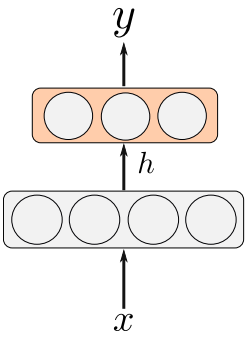


Figure 2.3: A Feed Forward Neural Network. The network has a single hidden layer with an output h, and takes input x, with the final output layer producing y.

subsequent chapters.

2.3.1 Basics of a Neural Network

The structure of a neural network rests on the definition of a neuron. A neuron acts as a gateway between a set of inputs, an n-dimensional vector $\in \mathbb{R}$, and the output signal. It can be modelled as ...

$$s = f(\sum_{n=1}^{i} w_n x_n + b)$$

... where w is a set of weights corresponding to each input, b is an overall bias controlling the output of the neuron, and f is the activation function, mapping the input signal to a range. A typical activation function is the sigmoid function ...

$$f(x) = \frac{1}{1 + e^{-x}}$$

... which maps the input signals to a value in the range of [0,1]. With this definition, the output of a neuron can be seen as the probability of a value on a certain input.

With this building block, neurons can be stacked horizontally to form a layer, and layers can be stacked vertically to form a deeper structure of **hidden layers**, allowing for more complicated structures to be represented before the final **output layer**.

The Feedforward pass

In a fully connected neural network, where each neuron in a layer receives the output signal from all neurons in the layer below it, the output signal s of a neuron i in layer m is ...

$$s_i^m = f(W_i^m s^{m-1} + b)$$

In this way, a signal propagates from the input to higher layers, where the weight matrix W needs to be tuned in order to affect the result. In a neural network with m layers and n possible outputs the result would be ...

$$y = max(s_0^m, ..., s_n^m)$$

For the input layer, neurons receive the raw input signals directly. The weight vector W_i^m for each neuron is initialised with random values at the start of training and is adjusted based on the performance on training data.

Backpropagation

With the structure outlined so far it becomes clear that achieving a functioning neural net is a matter of optimising the weight vectors of each layer so as to find a local or global minima of error, represented by a loss function.

The backpropagation of error was developed to address this in the case of neural networks with multiple hidden layers. It is, as the name implies, a reversal of feedforward propagation in that it uses the results of the output layer together with the expected result to calculate the error and subsequently a rate of change (delta) with which to update the weigh vectors of the output layer.

This delta is calculated based on the gradient of a loss function and is propagated backwards to lower layers in the network where it is included in the calculation of delta

for that layer. Simply put, the backpropagation algorithm is essentially an application of gradient descent. A consequence of this is that the activation function used by the artificial neurons must be differentiable.

A typical cost function is the squared error function. For a neuron in the output layer with output s, the cost function C is ...

$$C = \frac{1}{2}(t - s)^2$$

... where t is the expected target output. Further breaking down the make up of s, as stated in 2.3.1, the output of a neuron is the result of the weighted sum h of inputs with an applied bias run through an activation function. In this example the activation function used is the Sigmoid function σ. Thus s becomes ...

$$s = \sigma(h)$$

where ...

$$h = \sum_{n=1}^{i} w_n x_n + b$$

As for an inner neuron s_i, the cost function can be seen as the sum of errors of all neurons $s_o : o \in O$, that take the output of s_i as input, where O is the number of neurons that take s_i as input.

$$C_i = \sum_{o=1}^{O} C(s_o)$$

And so an optimisation problem involving finding the minimum value for C through the adjustment of the weight vector w becomes a gradient descent problem to compute the partial derivative $\frac{\partial C}{\partial w}$. Through applying the chain rule this becomes ...

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial s}\frac{\partial s}{\partial h}\frac{\partial h}{\partial w}$$

Computing the components of this is relatively straight forward.

$$\frac{\partial C}{\partial s} = \frac{1}{2}\frac{\partial (t-s)^2}{\partial w} = s - t$$

$$\frac{\partial s}{\partial h} = \frac{\partial \sigma(h)}{\partial h} = \sigma(h)(1-\sigma(h))$$

$$\frac{\partial h}{\partial w} = \frac{\partial (\sum_{n=1}^{i} w_n x_n + b)}{\partial w} = x$$

It becomes apparent from the calculation of $\frac{\partial s}{\partial h}$ why the activation function used by a neuron must be differentiable. The activation function used here, the Sigmoid function, is represented as σ and is stated in 2.3.1. Combining the previous results ...

$$\frac{\partial C}{\partial w} = (s-t)\sigma(h)(1-\sigma(h))x$$

or as commonly stated ...

$$\frac{\partial C}{\partial w} = \delta x$$

$$\delta = (s-t)\sigma(h)(1-\sigma(h))$$

This formulation however only applies to neurons of the output layer since, as stated previously, inner layer neurons use a cost function that relies on that of the layer above them. So calculating $\frac{\partial C_i}{\partial w_i}$ requires the calculation of $\frac{\partial C_i}{\partial s_i}$ ...

$$\frac{\partial C_i}{\partial s_i} = \sum_{o=1}^{O}(\frac{\partial C(s_o)}{\partial s_i}) = \sum_{o=1}^{O}(\frac{\partial C}{\partial s_o}\frac{\partial s_o}{\partial h}\frac{\partial h}{\partial s_i})$$

Note that s_i for the inner layer is x in the outer layer ...

$$\frac{\partial h}{\partial s_i} = \frac{\partial w_{io} s_i + b}{\partial s_i} = w_{io}$$

$$\frac{\partial C_i}{\partial s_i} = \sum_{o=1}^{O}(\frac{\partial C}{\partial s_o}\frac{\partial s_o}{\partial h}w_{io}) = \sum_{o=1}^{O}\delta_o w_{io}$$

Putting the result of $\frac{\partial C_i}{\partial s_i}$ in place of $\frac{\partial C}{\partial s}$ we get ...

$$\frac{\partial C_i}{\partial w_i} = (\sum_{o=1}^{O}\delta_o w_{io})\sigma(h_i)(1-\sigma(h_i))x_i$$

$$\delta_i = (\sum_{o=1}^{O}\delta_o w_{io})\sigma(h_i)(1-\sigma(h_i))$$

Finally the change in weight Δw by which the weight vector is updated is ...

$$\Delta w = -\alpha\frac{\partial C}{\partial w}$$

... where α is the *learning rate*. Two important things to note here. The delta receives a negative update since we are trying to minimise error. Second, the learning rate controls to what degree the gradient of the cost function affects the weight update. The reason behind this is to stabilise the change from one training example to the next. If the learning rate is too large the weights will change too rapidly around the optimum and may never converge, while if the learning rate is too low the system may converge very slowly or may get trapped in a local minimum.

Summary

In this Section we covered two important concepts used to train a neural network. **Feed-forward** propagation, for calculating an output based on layered and weighted combinations of raw input features, and **Backpropagation** which updates the weights in the various layers after comparing the output of the network.

These approaches are essential because of their relative straightforwardness and ease of implementation. It is important to note that the fact that they are so often coupled together does not mean that they are not compatible with other approaches or network

structures. They are however the basis for all the networks implemented in this thesis.

A simple outline of the way a neural network functions is presented in Algorithm 2. It can be seen in this algorithm how simple it would be to exchange one update strategy for another. In addition, the termination condition is a matter of judgement since perfect performance is rare in practice. Commonly used termination conditions include setting a maximum number of update passes in total and aborting after a set number of passes with no improvement in prediction accuracy.

Algorithm 2 Illustration of a functioning Neural Network

```
1: function TrainNN(Examples,ExpectedOutputs,LearningRate)
2:     w ← random() \\ Initialise all weights with random values
3:     α ← LearningRate
4:     repeat
5:         for all t ∈ Examples, e ∈ ExpectedOutputs do
6:             predictedOutput ← FeedForward(t, w)
7:             Δw ← Backpropagation(e, predictedOutput)
8:             w ← UpdateWeights(w, Δw, α)
9:     until Set number of passes or all predicted outputs are correct
10:    return w
```

2.3.2 Recurrent Neural Networks

The neural network described so far is capable of learning to assign a particular classification to a given input. It does not, however, model sequences. This can be seen in algorithm 2, line 5, where examples and expected outputs are consumed as pairs, with no information passed between one loop and the next.

A Recurrent Neural Network (RNN), on the other hand, is structured in a way that incorporates information from previous time-steps. This allows the network to pass along information from the start of a sequence to future time-steps.

The most basic version of this family of networks is the Simple Recurrent Neural Network (SRNN), of which the most prominent example is the Elman network (Elman, 1990). The structure of this network closely resembles that of the feed forward network discussed before, but its hidden layers pass on their output to both the deeper layer in

the network, and the same hidden layer in the next time-step. An illustration of this is shown in Figure 2.4.

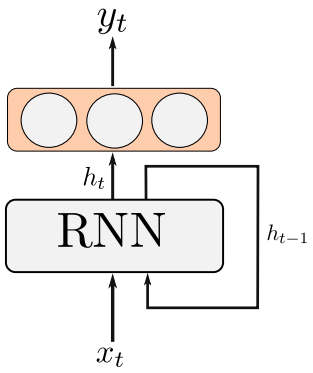


Figure 2.4: A Recurrent Neural Network (RNN)

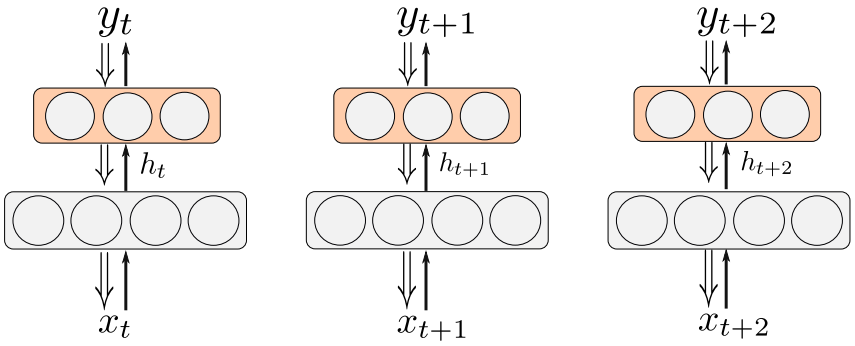
Since we are now considering input/output pairs over a sequence of time-steps, each signal must be qualified with the subscript t, referring to the time-step in which it is occurring. Notice how at time t in Figure 2.4 the input to the hidden layer is both the input for that time-step x_t, and the output of the hidden layer from the previous time-step h_{t-1}. And so the hidden layer in an Elman network can be written as ...

$$h_t = f(W_x x_t + W_h h_{t-1} + b)$$

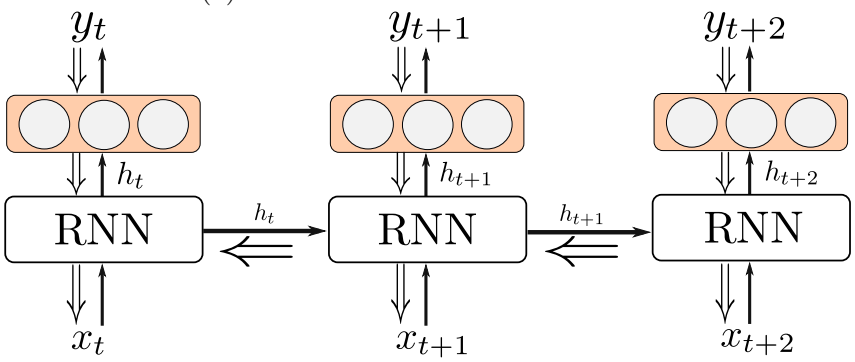
... where W_x is the weight matrix corresponding to the input at the current time-step x_t, and W_h is the weight matrix for the hidden layer output from the previous time-step h_{t-1}, and b remains the bias term.

RNNs in general are capable of arbitrarily expanding the context being considered by a network by propagating information between layers in different time-steps (Boden, 2002). The cost and subsequent weight updates are calculated and backpropagated through time (BPTT), to all incarnations of the network across the whole sequence.

To illustrate this we show an unrolled RNN across three time-steps compared to a feed forward network over the same sequence in Figure 4.1. The figure shows how information is passed between time-steps. Consider both the RNN and FNN at time t, for the FNN it will receive weight updates as a consequence of the error calculation for y_t. For the



(a) Unrolled Feed Forward Network



(b) Unrolled Recurrent Neural Network

Figure 2.5: Forward and back-propagation of information for RNNs and FNNs over three time-steps. Forward passing of information is represented by ↑ and backpropagation is represented by ⇓ & ⇐.

RNN at time t, on the other hand, it will receive weight updates based on the error calculation for all three time-steps, filtered through the calculations of the hidden layer error calculation for each time-step. In a sense, the hidden layer in a future time-step is correcting based on how useful past information was, while the hidden layer in a past time-step is correcting in order to pass along more useful information.

Additionally, SRNNs are known to suffer from a vanishing gradient problem (Pascanu et al., 2013), which leads to very little information backpropagating to past time-steps, an effect that increases the longer the sequence is. This makes training on sequences beyond a certain length unhelpful. For this reason, SRNNs are often trained with a *truncated backpropagation through time* strategy, where a set limit τ determines how many time-steps back the error is propagated. In the example shown in Figure 2.5b, the truncation limit $\tau = 3$. In our experiments in chapter 4 we apply this, among other tools, to deal with the gradient vanishing problem.

Moreover, there are other more sophisticated architectures of RNNs that are better equipped to learn which information in a sequence to retain, and can better deal with the gradient vanishing problem, such as Long Short-Term Memory Networks (Hochreiter and Schmidhuber, 1997) and Gated Recurrent Units (Cho et al., 2014). While their structure is very different to the Elman network discussed so far, they are still used in the same way. One key difference to consider however is that more complex RNNs do not necessarily pass along the same output to both the deeper layer in the current time-step and the next instances of themselves in the next time-step. We explore these structures in more detail in chapter 4.

2.4 Alternative Components

So far we have laid out a basic example of how a neural network is structured, how it computes its outputs, and how it adjusts its weights depending on its training error. In addition, we have described a basic improvement on that simple structure, in the form of the Simple Recurrent Neural Network. In practice, however, there are some finer-grained changes that we use in this thesis, and that are widespread in current literature as well. We describe some of these differences here.

In our example we present the Sigmoid function as the **activation function** for our hidden layer. While this remains widely used, it is also joined by others such as the Hyperbolic Tangent function, tanh, and Rectified Linear Units (Nair and Hinton, 2010)...

$$f(x) = \begin{cases} x : x > 0 \\ 0 : x \leq 0 \end{cases}$$

...or ReLUs, which provide a simpler to implement, and faster to calculate alternative. This function also helps with the gradient vanishing problem for both deeper networks and for RNNs, since its gradient is always either 1 or 0. In our work, we generally use

ReLUs for feed forward layers, while other activation functions are components of the RNN architectures that we use.

Additionally, we showed in our basic network that the output layer was the same as other feed forward layers, but with a number of neurons equivalent to the number of classes in the given task. In practice we additionally apply a Softmax function on each output neuron...

$$y_i = softmax(s_i) = \frac{e^{s_i}}{\sum_{j=1}^{n} e^{s_j}}$$

...where s_i is the signal from output neuron i, and a total of n output neurons/classes. This function thus produces a vector of scores that sum up to 1, and can thus be interpreted as a probability score for each class.

In conjunction with the softmax output layer, we also use a negative log likelihood loss function, instead of the Squared loss used before...

$$L(\theta) = -\sum_{i} log(y_i)$$

...where y_i is the probability of class i, which translates to the output of the i^{th} neuron *after* the softmax function is applied.

The next stage of traning was updating the weights in order to minimise the train error. This optimisation step was done in our example using stochastic gradient descent (SGD) (Bottou, 2012; LeCun et al., 1998), which remains the central method around which most of the common optimisation strategies are built. In our work we mainly use two strategies, SGD with Momentum, which accumulates previous gradients and applies them to current weight updates at a set rate, and Adam (Kingma and Ba, 2014), which adjusts the learning rate itself for each parameter.

2.5 Neural Networks in Dependency Parsing

One of the first works that attempted to use neural networks for parsing was that of Titov and Henderson (2007) which used an Incremental Sigmoid Belief Network for constituency parsing and was later adapted for use with dependency parsing in (Garg and Henderson, 2011). The latter adaptation employed Temporal Restricted Boltzman Machines and had the drawback of restricting vocabulary for a tractable approximation.

A simple but robust parser based on a simple feed forward network was introduced by Chen and Manning (2014), and remains the basis for some of the best performing transition-based dependency parsers, such as the work of Andor et al. (2016).

The more sophisticated variants of RNNs have enabled the modelling of entire components of the parsing process, going beyond even the selection of raw features. For example, Dyer et al. (2015) uses stack-LSTMs to model the stack and buffer, in addition to a recursive function that models partially built trees.

RNNs were also used to model whole input sentences, such as in the works of Kiperwasser and Goldberg (2016b), Shi et al. (2017), and Cross and Huang (2016), who used a bi-directional LSTM sentence representation to produce feature vectors for their neural networks. This approach was also used for graph-based parsing by Dozat and Manning (2016) who used the bi-directional LSTM layer as input to a biaffine attention classifier, producing the most accurate dependency parsing results to date.

Neural networks have also been used to model the dependency trees themselves. LSTMs were extended to be able to model tree structures by Tai et al. (2015), in so creating Tree-LSTMs. Another more common approach involved applying neural network elements recursively to model tree structures. This approach first gained ground in constituency parsing where recursive networks were used in (Goller and Kuchler, 1996; Socher et al., 2010) to model constituency trees.

Stenetorp (2013) trained Recursive Neural Networks for dependency parsing, however the final performance was not competitive. More successful approaches include the recursive compositional function used by Dyer et al. (2015), as well as the recursively applied

LSTM encoding of dependency trees done by Kiperwasser and Goldberg (2016a).

Our work in this thesis touches on a number of these approaches, and is deeply tied to an exploration of the basic neural network architecture presented by Chen and Manning (2014), as well as a more expressive representation of a sentence and its corresponding dependency tree.

CHAPTER 3

ARCHITECTURE OF NEURAL NETWORK-BASED TRANSITION-BASED DEPENDENCY PARSERS

The general structure of a dependency parser has remained largely unchanged since neural network based methods became the popular approach for this task. The parser first encodes relevant features from an input sentence using some sort of feature representation method, which is then passed to a hidden state that in turn encodes useful combinations for the final classification layer.

In the case of transition-based dependency parsing, this classification layer is almost always a scoring of all the shift actions together with a joint score for arc-building transitions and dependency labelling, as shown in Figure 3.1.

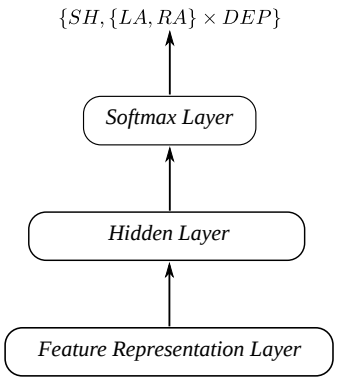


Figure 3.1: The basic neural network architecture for a transition-based dependency parser.

This architecture appears in Chen and Manning (2014), but also appears at the centre of a variety of techniques that have expanded upon this approach in various ways to produce competitive and state-of-the-art parsers. This approach is also the core around which all of the parsers in this thesis have been devloped.

In this chapter we will outline the Chen and Manning (2014) architecture, and discuss various subsequent works that have expanded upon it, as well as how these relate to the work in this thesis.

3.1 The Basic Architecture

The general operation of a transition-based parser, as previously described by Algorithm 1 in Section 2.2.1, can be summarised as: 1. extract features from the parser state, 2. input features into a prediction system, 3. apply the prediction to the parser state, and 4. repeat until parsing is done. It is step 2 that is the focus of most of the work, and where neural networks are used to make a prediction.

The Chen and Manning (2014) architecture, which we will also refer to interchangeably as the basic or core architecture, uses a simple feed-forward neural network with a single hidden layer for this task.

The hidden layer consists of 200 neurons, and uses the cubed activation function $f(x) = x^3$. The output layer is a softmax layer (described in Section 2.4) and produces a score for the Shift transition, and separate Left-Arc and Right-Arc transitions for every possible dependency label as shown in Figure 3.1.

The input layer, or feature representation layer, is a concatenation of vectors of features. Each feature is represented by a token and a certain aspect of that token such as its word form, its part-of-speech, or the dependency label connecting that token to its head token in the sentence. The vectors representing these features are stored in dictionaries relating each word/part-of-speech/dependency label to a corresponding vector. The values of the word vectors are often initialised with pre-trained values, and in the case of

Chen and Manning (2014) they are set to the word vectors produced by Collobert et al. (2011) for English and Mikolov et al. (2013) for Chinese. This architecture, and the rest of the models described here, can still train successfully without these pre-trained vectors, albeit with a drop in accuracy the extent of which varies depending on the architecture. The vectors for part-of-speech tags, dependency labels, and word forms (in the absence of pre-trained vectors) are randomly initialised, and their values are trained along with the weights of other neruons in the network using the same backpropagation of error described in Section 2.3.1.

3.2 Where Do Features Come From?

Feature modelling and selection are core challenges of any machine learning task. Before the rise in popularity of deep learning techniques, it was necessary to hand-craft feature combinations, which required extensive in-depth knowledge of the task and the range of information required to model. For tasks involving natural language, the flexibility and variety of human language makes this a daunting problem.

A key feature of deep learning is its ability to work out the necessary feature combinations during training. This has made for a very attractive alternative to existing techniques, and indeed neural networks have become an important component in many of the state-of-the-art dependency parsers in current literature. Moreover, the move from one-hot feature representation to dense word vectors (Collobert et al., 2011; Mikolov et al., 2013) facilitated the adoption of deep learning, since parsers could now have more informative, dense features, that represent a broad vocabulary.

This advance, however, still leaves the task of providing enough raw features that a neural network can then use to learn useful combinations. In the case of a transition-based parser this means providing enough features that adequately describe the configuration c of the parser, as well as relevant features of the partially built trees in the given sentence.

The features used in the Chen and Manning (2014) architecture, shown in Table 3.1,

Source	Features
Stack	$s_0^{w,t}, s_1^{w,t}, s_2^{w,t}$
Buffer	$b_0^{w,t}, b_1^{w,t}, b_2^{w,t}$
Dependency Tree	$rc_1(S_0)^{w,t,l}, rc_2(S_0)^{w,t,l}$
	$rc_1(S_1)^{w,t,l}, rc_2(S_1)^{w,t,l}$
	$lc_1(S_0)^{w,t,l}, lc_2(S_0)^{w,t,l}$
	$lc_1(S_1)^{w,t,l}, lc_2(S_1)^{w,t,l}$
	$rc_1(rc_1(S_0))^{w,t,l}, lc_1(lc_1(S_0))^{w,t,l}$
	$rc_1(rc_1(S_1))^{w,t,l}, lc_1(lc_1(S_1))^{w,t,l}$

Table 3.1: Features extracted from a configuration. w, t, and l are words, pos tags, and dependency labels respectively. rc_n & lc_n refer to the n^{th} rightmost/leftmost child.

are intuitive in that they encompass the words affected by the transitions that the parser can make, in addition to some structural context around these words. This set of features has remained with little or no change in works that expand on this architecture. Even the more elaborate representation methods that will be discussed in Section 3.4 attempt to keep the main features while producing more informative vectors to describe them, instead of relying on a concatenation of word, label, or part-os-speech vectors. Noteable exceptions to this trend are the works of Cross and Huang (2016) and Shi et al. (2017) who used very small subsets of this set of features, while making them more expressive. The work presented in Chapter 6 pushes in this direction even further.

As an example of how the features in Table 3.1 would be used, we present an example in Table 3.2, which is a step from the sequence of transitions discussed earlier in Table 2.1 (Section 2.2.4). Note that some words, such as "ball" in this example, can act as two features, and that many of the features in Table 3.1 may not be applicable at a point in the parsing sequence, such as $rc_1(s_0)$ in this case, since "ball" does not have any right dependents.

#	Stack	Buffer	Tree
7	[ROOT, hit, ball] (s_2, s_1, $s_0/rc_1(s_1)$)	[with, the, bat] (b_0, b_1, b_2)	ROOT (s_2) the ($lc_1(s_1)$) hit (s_1) the ($lc_1(s_0)$) ball ($s_0/rc_1(s_1)$) with (b_0) the (b_1) bat (b_2)

Table 3.2: Mapping the features in Table 3.1 to a configuration of a parser.

3.3 Enhancements around the Basic Architecture

Some approaches expand the architecture of Chen and Manning (2014) by addressing some main limitations of the basic architecture: 1. the inability of a feed-forward network to model sequences of parsing decisions, and 2. the small size of the network, which helps reduce train time, but is potentially less capable of modelling complex interactions.

To address the issue of sequence modelling Zhou et al. (2015) use global structured learning (also known as structured perceptron) with early updates (Collins and Roark, 2004). The architecture and dimensions used remain the same as those in Chen and Manning (2014), but this global training method acts as an extra layer above the final output layer, learning to increase the likelihood of the transition sequences that appear in the training data (usually referred to as gold sequences/transitions).

Complementing this train-time enhancement, beam-search (Zhang and Nivre, 2012) is used during test and run-time. This search method keeps a list (beam) of the n highest scoring transition sequences, where the score for each sequence is the sum of the probabilities for each transition in the sequence. The items in the beam are ordered by highest score, and each sequence is expanded by obtaining the next set of probabilities for it from the parser. Once all items in the beam are expanded and all the new scores are calculated, the n highest scoring sequences are kept in the beam and the rest are discarded. Once parsing is complete, the highest scoring sequence in the beam is taken to be the final predicted parse for the sentence.

Weiss et al. (2015) used a similar approach, but they also experimented with a range of hidden layer sizes from 256 to 2048 neurons wide, with depths of 1 and 2 hidden layers. In addition, they replaced the cubed activation function used by Chen and Manning (2014) with the Rectified Liner Units (ReLUs)(Nair and Hinton, 2010) discussed in Section 2.4. Their results showed a substantial improvement in accuracy, but with dimishing returns for wider layers. Moreover, Weiss et al. (2015) used the structured perceptron layer on top of the neural network's output layer differently to Zhou et al. (2015), passing as input the results of the output layer and the outputs of the the hidden layers. This combination

of differences resulted in an additional improvement to the final accuracy despite using a much smaller beam size.

Following this same line of work, Andor et al. (2016) use the refined architecture from Weiss et al. (2015), with hidden layers of dimension 1024. A key difference, however, is that they allow the structured perceptron layer to also tune the weights of the core network itself. This is also the case for the work of Zhou et al. (2015), but in their case they lack the size and improvements of the Weiss et al. (2015) model. The Andor et al. (2016) model achieves state-of-the-art results and remains the highest scoring transition-based dependency parser to date.

In contrast to these approaches, Kuncoro et al. (2016b) extended the basic model of Chen and Manning (2014) by replacing the hidden layer with Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) units, with peephole connections[1] (Gers et al., 2002). As discussed in Section 2.3.2, LSTMs and Recurrent Neural Networks in general are capable of passing hidden layer information to later time-steps in a sequence of decisions, thus allowing later decisions to take previous contextual information into account. The performance of this model was not competitive, however Kuncoro et al. (2016b) showed in their analysis that the LSTM achieved up to 3% improvement in accuracy for long-range dependencies[2].

In Chapter 4 we build on Kuncoro et al. (2016b)'s approach, exploring the use of multiple types of RNNs. We find that where LSTM-based models are concerned, they can produce strong results if peephole connections are **not** used, with accuracies that approach those of Zhou et al. (2015), despite not using global training or beam-search.

[1]Peephole connections (Gers et al., 2002) are a variant on LSTMs that give the input (i), output (o), and forget (f) gates access to the internal state c_{t-1} of an LSTM. The architecture and function of these gates and LSTMs is further discussed in Chapter 4.

[2]In their work Kuncoro et al. (2016b) define a long range dependency as being of length 7 words or more.

3.4 Enhancements to the Feature Representation Layer

The neural network-based parsers discussed so far use a combination of raw features, as represented by their dense vector embeddings to represent features of a sentence and the parser configuration. On the other hand, there has been substantial work on using innovative architectures to build more informative feature representations instead of simply concatenating feature embeddings.

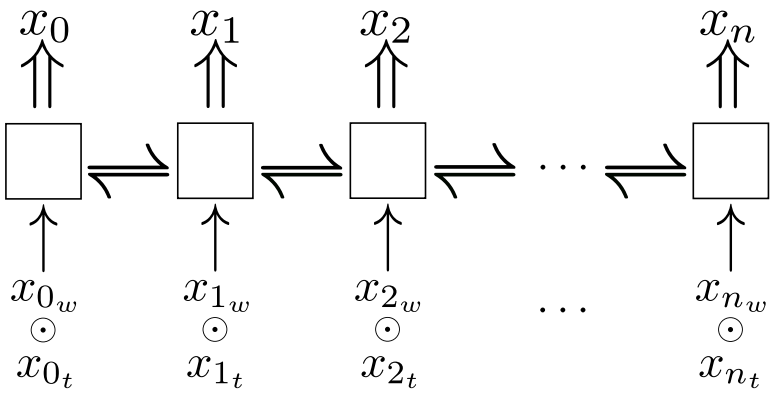


Figure 3.2: Bi-LSTM Feature Representation. The bi-LSTM layer produces the vectors x_{0-n} which represent the words of a sentence in order. The input to this layer is x_{*_w}, which is the word vector representation for a word, and x_{*_w}, which is the part-of-speech vector for that same word.

One approach has been to model the entire input sentence using bi-directional Long Short-Term Memory Networks (bi-LSTMs) (Cross and Huang, 2016; Kiperwasser and Goldberg, 2016b). The inputs to these bi-LSTMs are usually a concatenation of the word vector representation, as show in Figure 3.2, and its corresponding part-of-speech (pos) tag vector. The result is a vector for each word that encodes both its information, and relevant information from other words in the sentence, regardless of their position. The output of the bi-LSTMs was again passed onto a feed-forward layer to compute a hidden state before the final output layer. This approach has produced competitive parsers, and has even enabled fewer and more expressive features resulting in smaller feature sets than was possible before (Cross and Huang, 2016; Shi et al., 2017).

A limitation of this representation method, however, is that bi-LSTMs only model se-

quential information, and do not encode any hierarchical information, such as dependency relation within a sentence. The result is that parsers that rely on this method do not use any label features as input. In Chapter 5 we show that incorporating these label features where relevant in addition to the vectors produced by bi-LSTMs improves the accuracy of this method even further.

Another approach has been to represent the dependency tree itself with some form of recursive network, either bottom-up as in (Dyer et al., 2015; Kiperwasser and Goldberg, 2016a; Stenetorp, 2013), or top-down as in (Le and Zuidema, 2014).

Vector tree representation has a long history, primarily used to model constituency trees using Recursive neural networks (Goller and Kuchler, 1996; Socher et al., 2010). Such networks relied on the repeat application of a feed forward layer to encode a fixed maximum number of relations. Adapting this approach to an arbitrary number of dependents results in deep narrow trees and the gradient vanishing problem. One approach to deal with this has been the TreeLSTM model, an amended gating mechanism proposed by Zhang et al. (2015) based on LSTMs, which estimates the probability that a certain dependency tree is generated given a sentence.

For transition-based parsing, earlier work with recursive representation includes Stenetorp (2013), who uses a recursive layer to model dependency trees in a manner similar to that used in constituency parsers, but does not produce a high accuracy. Dyer et al. (2015) use a similar method of a recursively applied feed forward layer to represent subtrees as part of larger parsing architecture. Chen et al. (2015) use two Gated Recurrent Unit (GRUs) networks to represent the partially built dependency trees in the stack and to model potential dependencies that have not been built.

Kiperwasser and Goldberg (2016a) used LSTMs recursively to represent trees, and bi-LSTM vectors to represent the basic input words. Their work splits the sequence of a node's children into left and right children, with the head node itself as the first element in both sequences, before concatenating the last output of both directions to represent the subtree.

Finally, the Inside-Outside Recursive model of Le and Zuidema (2014) uses a top-down recursive representation of tree representations to re-rank a k-best list of trees produced by the MST parser (McDonald et al., 2006), which is not a neural network-based parser. This model achieves the highest accuracy of all the tree-modelling transition-based parsing methods in current literature.

We explore a similar concept to Kiperwasser and Goldberg (2016a) in Chapter 6, where LSTMs are used recursively to represent partial and full dependency trees, and also incorporate information from a bi-LSTM layer as discussed earlier to produce a competitive parser that relies on a minimal feature set.

3.5 Enhancements to the Classification Task

All of these works still keep the basic classification task intact, working to maximise a neural network's ability to both parse and label the dependencies of a sentence.

Recently Cross and Huang (2016) showed that restructuring the parser to have separate transition and dependency label classifiers, each with its own hidden state representation, but a shared bi-LSTM positional vector representation, (a set up which they called Hierarchical classification) produced better accuracies. This architecture, illustrated in Figure 3.3, was also used by Kiperwasser and Goldberg (2016b) and Shi et al. (2017), but neither examined the effect of this on parser performance.

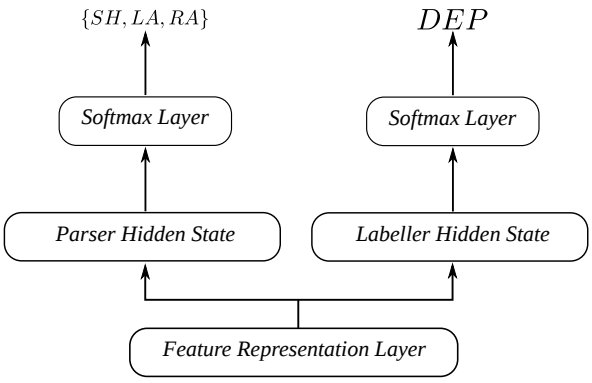


Figure 3.3: Hierarchical classification.

This change meant that the parser effectively optimised a network for two separate tasks, with one half of the network providing scores for the next parser transition $\{SH, LA, RA\}$, and the other half only scoring the dependency labels $\{DEP\}$ where applicable. All the examples of this approach also used a bi-LSTM feature representation layer.

In Chapter 5 we investigate the usefulness of this approach versus a joint classification architecture like that of Chen and Manning (2014), in addition to examining whether or not its benefits are tied to the use of bi-LSTM feature represention.

CHAPTER 4

RNN INITIALISATION WITH PRE-TRAINED FEED-FORWARD LAYERS

4.1 Introduction

The process of transition-based parsing involves a sequence of transitions used to shift through words in a sentence and build dependency arcs between them. Any incorrect transitions made could lead to more mistakes further down the sequence, and ultimately an incorrect dependency tree. Chen and Manning (2014)'s use of a feed forward network to decide these transitions meant that there was no way to consider the wider sequence around this transition. As discussed in Section 3.3, this problem was addressed by either using structured global training (Zhou et al., 2015; Weiss et al., 2015; Andor et al., 2016), or by replacing the hidden layer with a recurrent one (Kuncoro et al., 2016b).

In this chapter we build on Kuncoro et al. (2016b)'s approach by initialising the weights of an LSTM-based dependency parser with weights of a pre-trained Feed-Forward network. We show that this method produces a substantial improvement in accuracy scores, and is also applicable to different kinds of RNNs. An additional contribution is a refinement of the basic training model of Chen and Manning (2014) producing a more accurate Feed Forward model as a baseline for our experiments.

We begin with an explanation of our baseline models; the basic FFN and LSTM-

based models that are the centre of this work. We then explain our proposed method for the alternative initialization of the LSTM weights, and then present the results of our experiments with a comparison with other state-of-the-art parsers. Finally we explore the use of GRUs and Elman networks in place of LSTMs, and show the effect of initializing individual gates using our proposed method on the overall performance.

4.2 Baseline Models

Our proposed approach makes use of a simple feed-forward model to improve the performance of an LSTM-based model. We show that the final network surpasses both of our baselines, which are the original feed-forward network, and an LSTM model trained with randomly initialized weights. In this section we will describe the structure of both baselines.

4.2.1 Input Layer, Selected Features, & Output Layer

The Embeddings layer is a concatenation of the embedding vectors of select raw features of the parser configuration. The resulting layer is a dense feature representation of x. The features used in our implementation are the same as those shown in Table 3.1 in Section 3.2.

We represent the configuration of the parser at a particular timestep as a number of raw features extracted from the data structures of x. We use vector embeddings to represent each of the raw features.

Each word (w), part of speech tag (t), and arc label (l) is represented as a d-dimensional vector $e_w \in \mathbb{R}^{d_w}$, $e_t \in \mathbb{R}^{d_t}$, and $e_l \in \mathbb{R}^{d_l}$ respectively. And so the embedding matrices for the different types of features are $E^w \in \mathbb{R}^{d_w \times V_w}$, $E^t \in \mathbb{R}^{d_t \times V_t}$, and $E^l \in \mathbb{R}^{d_l \times V_l}$, where d_* is the dimensionality of the embedding vector for a feature type, and V_* is the vocabulary size. We add additional vectors for *"ROOT"* and *"NULL"* for all feature types, as well as *"UNK"* (unknown), for unknown/infrequent words.

This embeddings layer is used as the input layer in all models described in this work. For all models we use dropout (Hinton et al., 2012) on the input layer. We find that this improves the final accuracy of all the networks trained.

The **output layer** y consists of nodes representing every possible transition, with one node representing Shift, and a node for every possible pair of arc transitions (Left/Right-Arc) and dependency labels. This makes the size of the output layer constant at $2V_l + 1$, regardless of the structure of the network.

4.2.2 Feed-Forward Model

For our FFN model we use the same basic structure of Chen and Manning (2014) with a single hidden layer and a final softmax output layer. We however follow Weiss et al. (2015) in using rectified linear units (ReLUs) (Nair and Hinton, 2010) as hidden neuron activation functions. Finally, we use dropout on the hidden layer similar to the input layer. The structure of the FFN is specified below.

$$h = max\{0, Wx + b_h\}$$
$$y = softmax(W_h h)$$

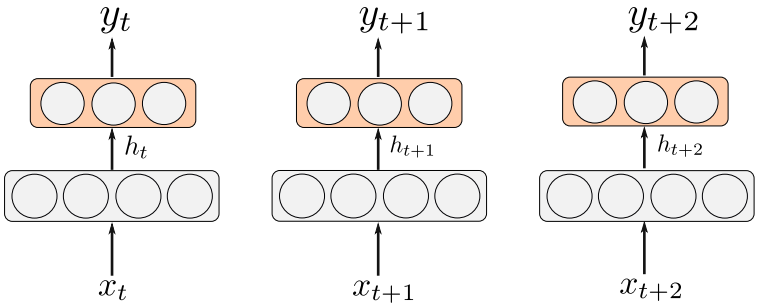
Following Weiss et al. (2015) we set the initial bias of the hidden layer to 0.02 in order to avoid having any dead ReLUs at the start of training.

4.2.3 RNN-based Model

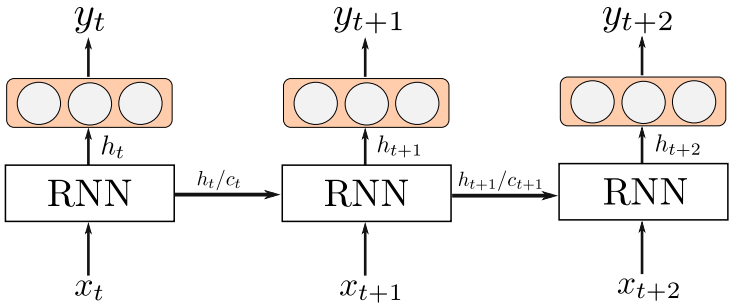
Our RNN-based model is an extension of the basic feed forward model, with Long Short-Term Memory (LSTM) units (Hochreiter and Schmidhuber, 1997) standing in for the traditional feed forward hidden layers.

The change allows for the information in the parser configuration to be shared as needed with future time-steps. This lets the network at any point in the sequence of

transitions make a decision based on a more informative context, that is not only based on the current configuration, or the present state of the dependency tree, but also on the changes made to them.



(a) Feed-Forward Network (FFN)



(b) Unrolled RNN

Figure 4.1: An FNN and an RNN over 3 time-steps. The FFN shown in 4.1a only has access to information from the current configuration as represented in x. RNNs on the otherhand also receive information about previous configurations as encoded in the hidden states from previous time-steps. The h_t/c_t refers to the external and internal hidden states produced by an LSTM, however other types of RNN units do not necessarily maintain a c_t.

In their standard forms, RNNs are affected by both exploding and vanishing gradients (Bengio et al., 1994), making them notoriously hard to train despite their expressive ability. LSTMs are a variety of RNNs that maintain an internal state c_t that forms the basis for the recurrence, and is passed from time-step to the next. This direct connection is not interrupted by any weight matrices, as would be the case in simpler RNN architectures such as Elman networks (Elman, 1990), but is instead scaled and added to by a number of gates that handle extracting and scaling information from the input data, and computing a final hidden state h_t at each time step to pass on to deeper layers. This uninterrupted connection of internal states throughout the sequence is an important part of how LSTMs

address the shortcomings of RNNs.

There have been a variety of architectures in literature referred to as LSTMs, all bearing slight differences to the basic LSTM unit. The definition of the LSTM we use in this work is shown below.

$$
\begin{aligned}
i_t &= \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \\
j_t &= tanh(W_{xj}x_t + W_{hj}h_{t-1} + b_j) \\
f_t &= \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \\
o_t &= \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \\
c_t &= c_{t-1} \odot f_t + i_t \odot j_t \\
h_t &= tanh(c_t) \odot o_t
\end{aligned}
$$

With the final softmax output layer, just as with the FNN model.

$$y = softmax(W_h h_t)$$

Unlike Kuncoro et al. (2016b), we do not use peephole connections like those suggested by Graves (2013). Additionally, we add a bias of 1 to the LSTM's forget gate following Gers et al. (2000). Finally, we also apply a dropout similar to that in Zaremba et al. (2014).

As shown in this definition, the LSTM cell maintains an internal state c_t, where the previous internal state c_{t-1} is modulated at each time-step by the forget gate f_t, and then added to by a scaled selection of the current input x_t by the input gates i_t and j_t. This new c_t is then used for the external state h_t and passed on to the next time-step. All gates rely on weighted activations of the current input x_t and the previous external state h_{t-1}.

This pair of hidden states allows the LSTM to contribute to long-term decisions with c_t, while still being able to make immediate or short-term decisions with h_t, and it is this

final calculation of h_t, along with o_t, that is the focus of our contribution in this work.

4.3 Initializing LSTM gates

Much has been written about the need for careful initialization of weights, often done to complement certain optimisation methods such as gradient descent with momentum in Sutskever et al. (2013). For deep networks, Hinton et al. (2006) and later Bengio et al. (2007) approached initialization differently by using a greedy layer-wise unsupervised learning algorithm, which trains each layer sequentially, before fine-tuning the entire network as a whole.

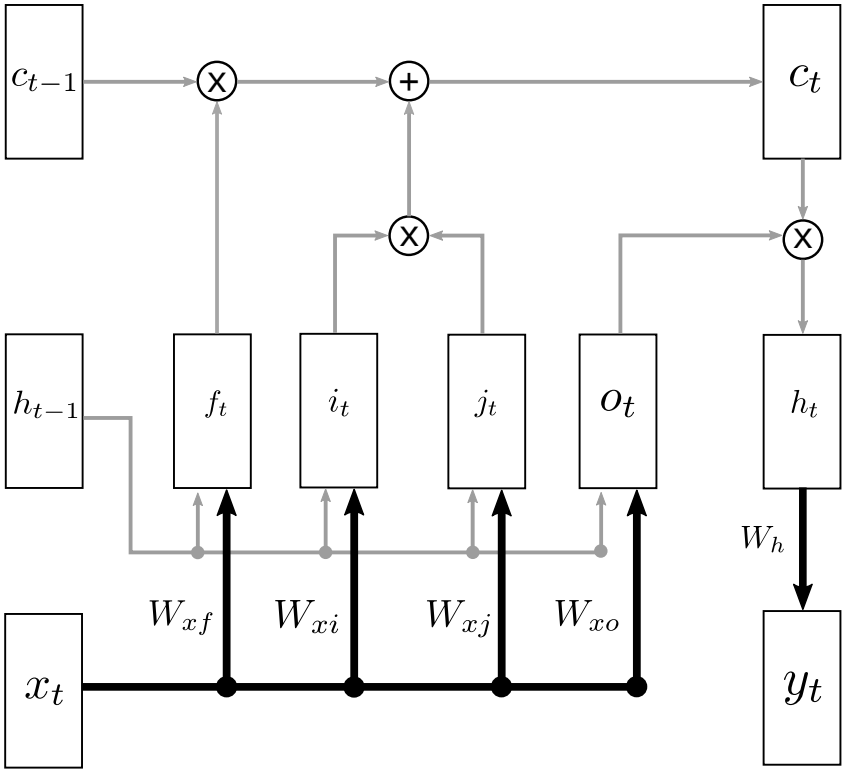
Le et al. (2015) suggested replacing traditional tanh units in a simple RNN with ReLUs, in addition to initializing the weights with an identity matrix.

As previously mentioned, Gers et al. (2000) suggested initializing the bias of the forget gate b_f of an LSTM to 1. This allowed the LSTM unit to learn which information it needed to forget as opposed to detecting the opposite. This was later shown by Jozefowicz et al. (2015) to improve performance of an LSTM on a variety of tasks.

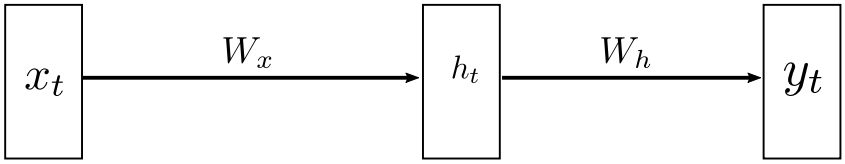
Alternatively, it has become increasingly common to use tuned outputs from one network as initialization for another. For example the use of pre-trained embeddings as initialization for word vectors has become de facto standard procedure for tasks such as dependency parsing, language modelling and question answering.

Following this approach, we propose initializing the LSTM weights, specifically the W_{x*} and bias b_* of all LSTM gates, with the weight matrix W_x and hidden bias b_h of a pre-trained, similarly structured feed-forward neural network. We also initialize the embedding matrices used $E^{w,t,l}$ and the weights of the final softmax layer W_{hy} with those of the pre-trained feed-forward network.

To illustrate this idea we reproduce a modified version of the LSTM architecture diagram appearing in Jozefowicz et al. (2015) in Figure 4.2, with the addition of the final softmax layer y_t. The flow of information from the current input x_t to y_t (as shown by



(a) LSTM-based Model



(b) FFN-based Model

Figure 4.2: A comparison of the architecture of an FFN and an LSTM-based model. The bold arrows represent the weight matrices that are roughly equivalent to those in an FFN, and y_t is the final softmax layer that scores each possible transition. We only show labels for the matrices that we initialize with their FFN counterparts, $W_x \rightarrow W_{x*}$ and $W_h \rightarrow W_h$, where $* \in \{i, j, f, o\}$. Additionally we replace the biases of the LSTM gates with the bias of the hidden layer of the FFN, $b_h \rightarrow b_*$, and all the FFN trained embeddings for all feature types.

the bold arrows) is almost identical to that in an FFN, except for the addition of h_{t-1} as input to o, and the "interference" of information from c_t to produce h_t.

This approach rests on the 2 hidden states of the LSTM requiring different information from the same input data. Since h_t is more concerned with immediate decisions, it would strongly benefit from the trained weights of a feed-forward network, which are tuned to extract the maximum relevant information from the input of the current time-step, since it has no access to prior information.

The various LSTM gates would still be able to learn to use information from h_{t-1} but would be in a better position to do so with the biases and input weights closer to an optimum configuration.

Moreover, the internal state c_t would receive less severe errors early on in the training process, owing to a better contribution from o_t in the calculation of h_t, and a less disruptive result from c_t due to the input and forget gates initially behaving more similarly to the regular hidden layer of the original FFN.

This would mean less pressure on the weights of the input and forget gates to adapt to immediate decisions while the internal state would be more capable of gradually learning longer term patterns.

We will henceforth differentiate networks initialized in the manner described in this section by referring to them as **bootstrapped models**, while we refer to the usual randomly initialized networks as **baselines models**.

4.4 Experiments

We begin by comparing the performance of our FFN and LSTM baseline networks with our bootstrapped model. For all networks we ran a model with a single hidden layer 256 neurons/LSTM units wide. The embeddings dimensions used were $d_w = d_t = d_l = 100$. We use the GloVe pre-trained embeddings produced by Pennington et al. (2014) to initialize the word vectors.

Learning is done with mini-batch stochastic gradient descent (SGD) with momentum to minimise negative log likelihood loss with the learning rate $\alpha = 0.05$ and momentum $\mu = 0.9$. We also use an additional l_2 regularization cost ($\lambda = 10^{-8}$).

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

Where θ represents all weight, biases, and embeddings matrices. We also set the dropout rate to 0.3 for the embeddings layers and hidden layer for both the baselines and bootstrapped model, and initialise all baseline weights randomly in the range $[-0.01, 0.01]$.

For LSTM-based models we used truncated backpropagation through time (BPTT), with a truncation limit $\tau = 5$. This means that errors are propagated backwards to layers in previous time steps until a limit τ is reached. In our experiments varying τ between 5 and full back propagation had a negligible effect on the final accuracy of the networks, while using a truncation limit produced a significant speed up in training. We stress that this insignificant difference is most likely a task and architecture specific issue, and would probably be much more pronounced in other tasks and neural network set-ups.

For our experiments we use the Wall Street Journal (WSJ) section from the Penn Treebank (Marcus et al., 1993). We use §2-21 for training, §22 for development, and §23 for testing. We use Stanford Dependencies (SD) De Marneffe et al. (2006) converted from constituency trees using version 3.3.0 of the converter. As is standard we use predicted POS tags for the train, dev, and test sets. We report unlabelled attachment score (UAS) and labelled attachment score (LAS), with punctuation excluded.

Network Type	Dev		Test	
	UAS	LAS	UAS	LAS
Feed-Forward Network				
Chen and Manning (2014)	92.00	89.70	91.80	89.60
Andor et al. (2016)	92.85	90.59	92.95	91.02
Weiss et al. (2015)	N/A	N/A	93.19	91.18
Our FFN baseline	92.76	90.47	92.10	89.95
LSTM Network				
Zhang et al. (2015)	92.66	89.14	91.99	88.69
Dyer et al. (2015)	93.2	90.9	93.1	90.9
Kuncoro et al. (2016b)	N/A	87.8	N/A	87.5
Kiperwasser and Goldberg (2016a)	93.3	90.8	93.0	90.9
Kiperwasser and Goldberg (2016b)	N/A	N/A	93.9	91.9
Our LSTM baseline	93.23	90.94	92.77	90.64
Our bootstrapped model	93.41	91.20	93.06	91.01

Table 4.1: Final dev and test set scores on WSJ (SD). Zhang et al. (2015) do not use pre-trained word vectors for their final result. The values given for Andor et al. (2016) and Weiss et al. (2015) reflect only the performance of the greedy FFN models produced in their work, with other improvements made explained briefly in section 4.1.

The results in Table 4.1 show the effect of applying dropout on the input layer for our FFN baseline, when compared to the similarly sized Chen and Manning (2014) model which has 200 neurons in its hidden layer. This is in addition to achieving very close dev score accuracy results with only a single 256 neuron hidden layer when compared to the significantly larger models of Weiss et al. (2015) with 2 layers of size 2048, and Andor et al. (2016) with 2 layers of size 1024 layers.

Comparing our 2 baseline models shows that the LSTM-based model performs much better than the FFN model, with an almost 0.5% gain in dev score accuracy. Our main result is our bootstrapped model, which not only surpassed the original FFN baseline, but also the LSTM baseline.

We note that our LSTM-baseline achieves a substantial improvement over the similar architecture of Kuncoro et al. (2016b). The main differences in this case are a slightly larger model and using LSTMs *without peephole connections*.

In addition, our bootstrapped model produces better results than all the mentioned feed forward models in addition to most of the LSTM-based approaches in Table 4.1, with

the exception of Kiperwasser and Goldberg (2016b), despite only having a single hidden layer of LSTM units and making no use of bi-LSTMs, TreeLSTMs, or Stack LSTMs.

4.5 Discussion

The results of our experiments seem to lend credence to the idea that learning short and long-term patterns separately is useful to the performance of an LSTM. To generalize this further, one could say that a sequence modelling task where a 1-to-1 relation between input/output pairs can be learned should first attempt this with an FFN, and then transfer that knowledge to an LSTM as described in Section 4.3, so sequence specific information can be further modelled.

An additional benefit of this approach is that it can be applied to previously trained FFNs and can improve any of the models that we have compared our results with in Table 4.1. This is also true of the LSTM-based models, where the strength of their contributions lies in their innovative approaches to feature extraction while keeping the rest of the network essentially the same.

For example, we can merge our work with that of Kiperwasser and Goldberg (2016b), by first training their model; a biLSTM input layer going to a feed-forward hidden layer followed by an output layer, and then replacing the hidden layer with an LSTM initialized with the weights of that hidden layer.

Finally, our addition of applying dropout to the input layer can also be used here to further strengthen the performance of this example.

4.6 Alternative Recurrent Units

So far we have shown how to improve the performance of LSTMs by drawing parallels between the functions of certain gates and the traditional feed-forward network. In this section we attempted to do the same for 2 other popular forms of RNNs, the Simple Recurrent Network, otherwise known as the Elman network (Elman, 1990), and the Gated Recurrent Unit (GRU) (Cho et al., 2014).

4.6.1 Elman networks

The Elman network is one of the earliest and simplest RNNs found in literature. It was the subject of much study and suffered from all the original problems of vanishing and exploding gradients mentioned before, which later motivated the development and adoption of more sophisticated units such as LSTMs and GRUs.

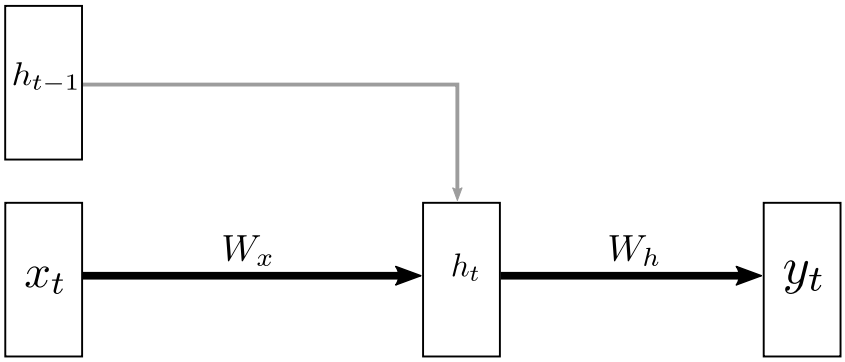
Nevertheless there have been examples where Elman networks were capable of performing relatively well, notably the work of Mikolov et al. (2010) on language modelling and an extended memory version of Elman networks in Mikolov et al. (2014).

Elman networks themselves are only a simple addition to the architecture of the traditional feed-forward network. Whereas an FFN has a hidden layer, and Elman network has an additional context layer, that represents the output of the hidden layer in the previous time-step. In a way, it can be compared to output gate of an LSTM, without any additional tools to model the sequence.

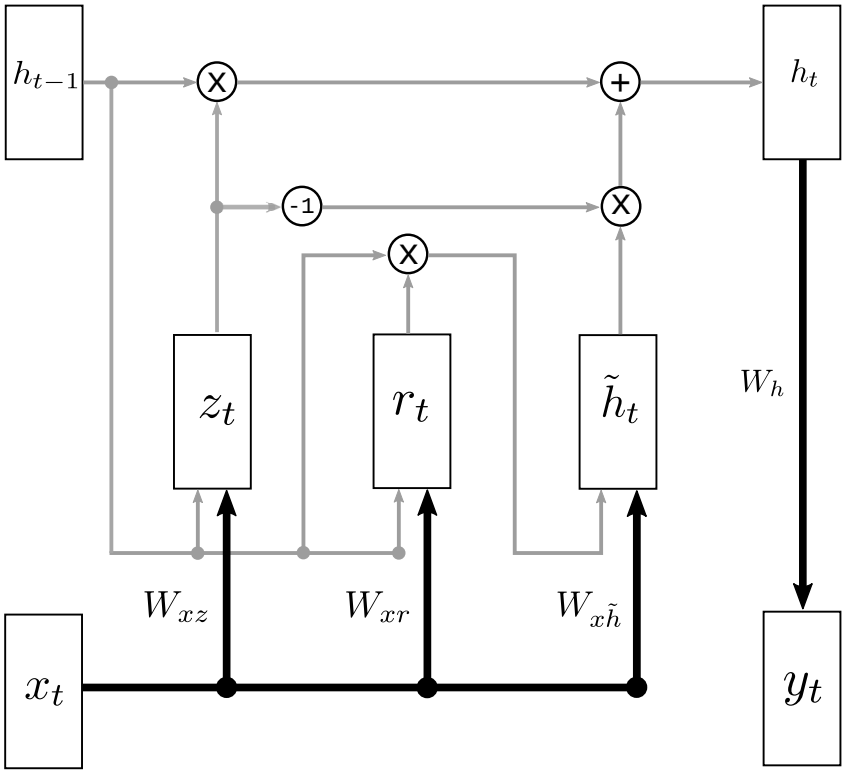
In our experiment we use the ReLU activation function once more for the hidden layer similar to Le et al. (2015), but without their initialization strategy. The precise definition of the Elman network that we use is shown below.

$$h = max\{0, W_x x + W_h h_{t-1} + b_h\}$$

$$y = softmax(W_h h)$$



(a) Elman-based Model



(b) GRU-based Model

Figure 4.3: The architectures of an Elman and a GRU-based model. As in 4.2, the bold arrows represent the path of information roughly equivalent to that in an FFN. The replaced matices in the Elman-based model are $W_x \rightarrow W_x$, and $W_h \rightarrow W_h$. For the GRU-based model the replaced matrices are $W_x \rightarrow W_{x*}$, where $* \in \{z, r, \tilde{h}\}$, and $W_h \rightarrow W_h$. For both RNNs this is in addition to initializing the embeddings vectors with those trained by the baseline FFN for all feature types.

In Figure 4.3a we illustrate the structure of this network. The simplicity of the addition here makes it far easier to draw parallels between the function of the weight matrices in the Elman network and in the FFN as shown in Figure 4.2b.

4.6.2 Gated Recurrent Units

Introduced by Cho et al. (2014), GRUs are an architecture often compared to LSTMs. It also attempts to solve the gradient vanishing problem in a similar way, by keeping the modulation and addition of information in separate gates, and avoiding any weighted obstructions between the hidden states of one time-step and the next. A notable difference however is the lack of an internal state. All modifications are done directly to the external hidden state h_t, potentially complicating the learning process with conflicting information about short and long-term dependencies.

Despite this apparently simpler structure, Chung et al. (2014) found GRUs to outperform LSTMs on a number of tasks, and Jozefowicz et al. (2015) also found that GRUs can beat LSTMs except in language modelling. However, Jozefowicz et al. (2015) also found that initializing the LSTM forget gate bias b_f to 1 allowed the LSTM to almost match the performance of the GRU on other tasks. The structure of a GRU is shown below.

$$
\begin{aligned}
r_t &= \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r) \\
z_t &= \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z) \\
\tilde{h_t} &= \tanh(W_{x\tilde{h}}x_t + W_{h\tilde{h}}(r_t \odot h_{t-1}) + b_h) \\
h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h_t} \\
y &= softmax(W_h h_t)
\end{aligned}
$$

The internal architecture of a GRU consists of a reset gate r_t modulating the previous state h_{t-1}, a candidate gate $\tilde{h}$ computing the next addition to h_t, and an update gate z_t controlling how much of the candidate $\tilde{h_t}$ is added to h_t.

In this case the candidate gate $\tilde{h_t}$ is the most analogous to the hidden layer in an FFN. As shown by the bold lines in Figure 4.3b, this flow of information appears similar to that of the output gate o_t in an LSTM, except that the additional input here is modulated by the r_t instead of receiving h_{t-1}, in addition to dealing with further interference from the update gate.

4.6.3 Comparison & Results

For the experiments in this section we used the same network dimensions as in Section 4.4, as well as the same training parameters and procedure.

For each RNN type we trained 2 FFN and RNN baselines, one with GloVe pre-trained word embeddings Pennington et al. (2014) and another with randomly initialized embeddings. We then trained bootstrapped models initialized with the FFN baselines. The results are shown in Table 4.2.

As in Section 4.4, this initialization method shows a positive effect on an LSTM-based model, again surpassing both its baselines. The Elman network is stronger than expected and benefits greatly from this approach. Indeed, the bootstrapped Elman model is comparable in accuracy to some of the results in Table 4.1.

This cannot be said of GRUs, however, where its baselines perform significantly worse than other RNNs. Moreover, bootstrapped GRU models perform even worse than their baselines, even failing to match the accuracy of the FFNs used to initialize them. This disparity in accuracy compared to LSTMs seems to lend credence to our earlier hypothesis that learning long-term sequences can interfere with learning to make immediate decisions based on the input from the current time step. The architecture of an LSTM which maintains a long-term internal state c_t separate from a short-term external state h_t, and the additional improvement gained from learning these separately, as opposed to the single common hidden state h_t in GRUs, appears to provide a distinct advantage here.

The improvement achieved by a bootstrapped Elman model can thus be explained by the fact that it suffers from gradient vanishing Bengio et al. (1994), and so sequence

Embeddings Type	UAS	LAS
Random Embeddings		
FFN baseline	92.21	89.85
LSTM baseline	92.16	89.87
bootstrapped LSTM	**92.43**	**90.06**
Elman baseline	91.97	89.62
bootstrapped Elman	92.40	**90.06**
GRU baseline	91.62	89.18
bootstrapped GRU	91.67	89.31
Pre-trained Embeddings		
FFN baseline	92.76	90.47
LSTM baseline	93.23	90.94
bootstrapped LSTM	**93.41**	**91.20**
Elman baseline	92.01	89.47
bootstrapped Elman	92.87	90.52
GRU baseline	92.21	89.77
bootstrapped GRU	92.14	89.78

Table 4.2: Dev set scores on WSJ (SD) for different RNN types. The Random/Pre-trained embedding only refers to the initial word vectors of the FFN/RNN baseline. All other RNNs in these categories use the final trained embeddings of their respective FFN baseline.

specific information does not affect training to the extent that it does in GRUs.

4.7 Initializing Individual Gates

Our final set of experiments is to investigate whether or not individual gates of LSTMs and GRUs can benefit from this initialization technique. We follow the same initialization and training procedures described previously, and for every gate we also initialize its corresponding bias vectors. We keep the same size and parameters as in Section 4.6.3, and also train baselines with and without pre-trained embeddings.

Bootstrapping individual LSTM gates produces mixed results, as show in Table 4.3, especially when considering the difference in performance between the random and pre-trained embeddings experiments.

Full bootstrapping, bootstrapping the j gate or bootstrapping the o gate seem to be

Embeddings Type	UAS	LAS
Random Embeddings		
FFN baseline	92.21	89.85
LSTM baseline	92.16	89.87
bootstrapped i gate	92.29	90.00
bootstrapped j gate	92.38	89.96
bootstrapped f gate	92.25	89.81
bootstrapped o gate	92.43	90.06
bootstrapped all gates	92.38	90.01
Pre-trained Embeddings		
FFN baseline	92.76	90.47
LSTM baseline	93.23	90.94
bootstrapped i gate	93.20	90.96
bootstrapped j gate	93.30	91.02
bootstrapped f gate	93.42	91.22
bootstrapped o gate	93.35	91.11
bootstrapped all gates	93.41	91.20

Table 4.3: Dev set scores on WSJ (SD) for individually bootstrapped LSTM gates

the most reliable options based on these results.

Results for bootstrapping individual GRU gates, as show in Table 4.4, vary drastically, with individual gates performing very differently in their random and pre-trained embedding experiments.

Surprisingly, bootstrapping all GRU gates achieves better results than the GRU baseline for random embeddings, while severely hurting accuracy with pre-trained embeddings. All GRU experiments, bootstrapped or not, still do not perform better than the FFN baseline.

Embeddings Type	UAS	LAS
Random Embeddings		
FFN baseline	92.21	89.85
GRU baseline	91.62	89.18
bootstrapped r gate	91.70	89.15
bootstrapped z gate	90.59	87.90
bootstrapped $\tilde{h}$ gate	91.67	89.31
bootstrapped all gates	91.73	89.20
Pre-trained Embeddings		
FFN baseline	92.76	90.47
GRU baseline	92.21	89.77
bootstrapped r gate	92.22	89.79
bootstrapped z gate	92.62	90.37
bootstrapped $\tilde{h}$ gate	92.14	89.78
bootstrapped all gates	89.30	86.09

Table 4.4: Dev set scores on WSJ (SD) for individually bootstrapped GRU gates

4.8 Conclusion

In this chapter we have presented a simple and effective LSTM transition-based dependency parser. Its performance rivals that of far more complicated approaches, while still being capable of integrating with minimal changes to their architecture.

Additionally, we showed that the application of dropout to the input layer can improve the performance of a network. Like our other contributions here this is simple to apply to other models and is not only limited to the architectures presented in this work.

Finally, we proposed a method of using pre-trained FFNs as initializations for an RNN-based model. We showed that this approach can produce gains in accuracy for both LSTMs and Elman networks, with the final LSTM model surpassing or matching most state-of-the-art LSTM-based models.

This initialization method can potentially be applied to any LSTM-based task, where a 1-to-1 relation between inputs can first be modelled using an FFN. Exploring the effects of this method on other tasks is left for future work.

CHAPTER 5

EXPLORING THE BEST STRUCTURE FOR A TRANSITION-BASED DEPENDENCY PARSER

5.1 Introduction

For a majority of neural network-based transition-based parsers building dependency arcs and assigning a dependency label are done together. This can be seen in the structure of the output layer which assigns a separate neuron/class to each combination of dependency arc and dependency label. An exception to this is the work of Cross and Huang (2016) who noted an increase in accuracy when separating parsing and labelling into two different classification layers, while maintaining a join input layer.

This observation by Cross and Huang (2016) seems to indicate that dependency parsing and dependency labelling are not as interdependent as a survey of the current parser architectures might lead one to believe, at least as far as neural network-based parsers are concerned.

This seems to go against a direction towards more joint learning of different tasks, such as work showing improvements in parser accuracy when trained as a joint parser and part-of-speech tagger, both for neural network-based models (Alberti et al., 2015), and non-neural network-based models (Bohnet and Nivre, 2012), in addition to work on joint syntactic-semantinc parsing for neural network-based models (Swayamdipta et al., 2016;

Henderson et al., 2013) and non-neural network-based models (Björkelund et al., 2010).

In this chapter we explore whether dependency parsing and dependency labelling are altogether independent tasks, **specifically for neural network-transition-based parsers**, and if they should therefore be approached separately. To investigate this we ask the following questions ...

- Can the hierarchical architecture be broken down into finer grained classification tasks?
- Is the parser transition information useful for assigning a dependency label?
- Does having a joint feature representation layer help the goals of parsing and labelling?
- Can a dependency parser and a dependency labeller be successfully trained without sharing any encoding or classification layer?

To answer these questions we experiment with three neural networks that are modified versions of the hierarchical architecture, in addition to our own implementations of a joint state network like (Chen and Manning, 2014) and a hierarchical network like Cross and Huang (2016). Our networks are an extended hierarchical network, that has separate classifiers for right and left dependencies, a successive network, that passes the output of the parsing component to the labeling component, and finally we experiment with training the parsing and labelling components separately. In addition, we perform our experiments using two different feature representation methods, an embeddings concatenation based method, and a more robust bi-LSTM positional vector representation.

Our experiments do show that a dependency parser and dependency labeller can successfully be trained independently, and the results are comparable with those in current literature and state-of-the-art systems. We do however confirm the improvement that using hierarchical classification has over other architectures for neural network-based parsers, and also produce better results than separately training a parser and labeller.

Our best separately trained parser/labeller pairs achieve accuracies of (92.21/89.92) on WSJ test set when using concatenated word embeddings, and (94.04/91.95) when using bi-LSTM positional encoding. For our hierarchical models, we achieve a best accuracy of (92.27/90.09) when using concatenated word embeddings, and (94.30/92.23) with bi-LSTM positional encoding. We explore the implications of our experiments and how they relate to the questions that we have asked, as well as offer an idea on how parsing and labelling could be pursued further in future work in Sections 5.6 & 5.7.

5.2 Feature Representation

In order to fully understand the effects of the architectures that we explore in Section 5.3, we will be testing each architecture using 2 popular methods of feature representation. In our experiments we attempt to make the features modelled by each method of representation as similar as possible, without impacting performance.

Embeddings vectors This method has been used in some of the earliest, as well as the highest-scoring, neural-network based parsers in current literature (Chen and Manning, 2014; Weiss et al., 2015; Andor et al., 2016). A dictionary of n-dimensional vectors for possible *words, part-of-speech tags, and dependency labels* is built and used to represent features of a sentence. These vectors are typically randomly initialised, although in the case of word vectors, pre-trained vector embeddings are frequently used.

We follow the feature scheme used by Chen and Manning (2014), which uses the following features:

- the word and pos tags of the first 3 items from the stack and buffer, $\{s_{0-2}, b_{0-2}\}$
- the word, pos tags, and dependency labels of the first 2 left/right-most children of the top 2 items on the stack, $\{lc_0(s_{0,1}), lc_1(s_{0,1}), rc_0(s_{0,1}), rc_1(s_{0,1})\}$

- the word, pos tags, and dependency labels of the leftmost child of the leftmost child, and the rightmost child of the rightmost child of the top 2 items on the stack, $\{lc_0(lc_0(s_{0,1})), rc_0(rc_0(s_{0,1}))\}$

The final output layer is a concatenation of the vectors of all these features. The values of these vectors are simultaneously trained with the rest of the network. For the rest of this chapter we will refer to this method simply as the *Embeddings* method/representation.

Positional vectors A bi-directional LSTM (Graves and Schmidhuber, 2005) is a method used to encode a sequence using Long Short-term Memory networks (Hochreiter and Schmidhuber, 1997) or LSTMs, in such a way that each output vector represents both relevant information about the input at that position, but also information about other parts of the sequence as it relates to. This is accomplished by running a *forward*-LSTM on the sequence in order, and a second *backward*-LSTM on the sequence in reverse. The forward and backward vectors at corresponding positions are then concatenated to produce the final bi-directional LSTM (bi-LSTM) vector for each position in the sequence.

For transition-based parsing, modelling an input sentence using bi-LSTMs was proposed by both Cross and Huang (2016) and Kiperwasser and Goldberg (2016b), and this approach was also used to great effect by Dozat and Manning (2016) in an attention based parser. Each word in the sentence would be represented as a concatenation of the word's word and pos tags, drawn from embeddings dictionaries created in the same way discussed for the Embeddings method. Cross and Huang (2016) also realised that the minimal feature set required when representing the sentence this way becomes as small as the top 2 items on the stack and the front item on the buffer $\{s_{0,1}, b_0\}$, an observation that was later confirmed empirically by Shi et al. (2017).

In our experiments however, we found that while this minimal feature set does produce a competitive parser, adding more structural features can improve the final accuracy further. This is in line with the findings of Kiperwasser and Goldberg (2016b) who also used an extended feature set to improve the final results. In our case we not only use the

bi-LSTM vectors, but also the dependency label embeddings for structural features, i.e. features representing children or children of children. These dependency label embeddings are concatenated as above and then used as input to the hidden state along with bi-LSTM feature vectors. We will refer to this method in this chapter as the *Positional* method/representation. This has the added benefit of bringing the feature set used more in line with that used for the Embeddings method. The features used are as follows:

- the word and pos tags of the first 2 items on the stack and the front item of the buffer, $\{s_{0,1}, b_0\}$

- the word, pos tags, and dependency labels of the 2 leftmost children of the top of the stack and the 2 left/right-most children of the second item on the stack, $\{lc_0(s_{0,1}), lc_1(s_{0,1}), rc_0(s_1), rc_1(s_1)\}$

- the word, pos tags, and dependency labels of the leftmost child of the leftmost child of the top 2 items on the stack, and the rightmost child of the rightmost child of the 2nd item on the stack, $\{lc_0(lc_0(s_{0,1})), rc_0(rc_0(s_1))\}$

5.3 Architectures

To investigate the limits of the observation made by Cross and Huang (2016), that parsing and labelling benefit from having separate classification layers and separate hidden layers, we designed a number of parsers that approach the classification problem in ways that are slightly different to both the popular method and that of Cross and Huang (2016). We compare the performance of these proposed architectures to our own implementation of published methods in Section 5.5. In this section we describe all the architectures we use.

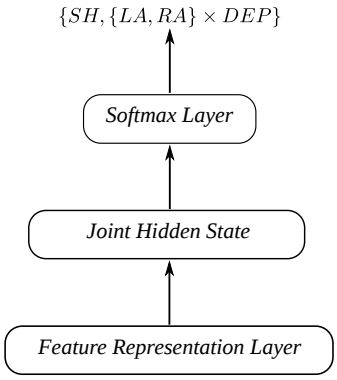


Figure 5.1: The Joint state architecture.

5.3.1 Joint state

This architecture is the simplest possible approach to transition-based dependency parsing, and perhaps the easiest to implement. It is made up of a single hidden state and a joint classification layer of the form $\{SH, \{LA, RA\} \times DEP\}$, where $\{SH, LA, RA\}$ are the Shift, Left-Arc, and Right-Arc transitions as defined in Section 2.2.2, and DEP is the set of all possible dependency labels. This is the approach used initially by Chen and Manning (2014) and on which a number of state-of-the-art parsers are based.

5.3.2 Hierarchical

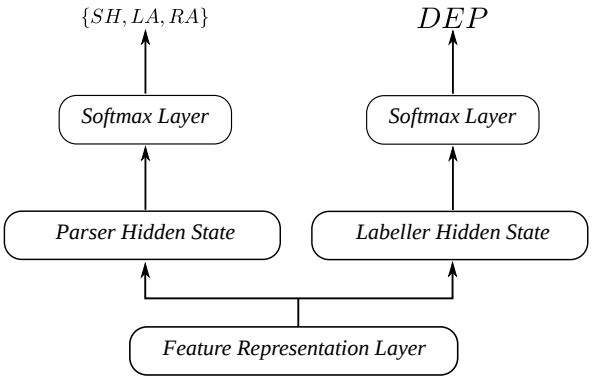


Figure 5.2: The Hierarchical architecture.

Hierarchical classification was independently done by both Cross and Huang (2016) and Kiperwasser and Goldberg (2016b), however only the former explicity mentioned this particular modification is having a notable effect on parser accuracy. This architecture

splits the classification into two, one responsible for producing the next transition, the other produces the dependency label. Each classification layer has its own separate hidden state, but both still share the same feature representation layer. This means that during training that the weights of the feature representation layer receives the backpropagated errors from both classifiers, and ultimately encodes information to the benefit of both.

This shared layer has the potential to either improve the performance of both tasks, since information learnt from one task could be useful to the other, or it could the learning processes of both could be conflicting, resulting in a lower final accuracy. We already know, given the results of the comparison done by Cross and Huang (2016), that Hierarchical classification does produce better results than a Joint state parser, but that could possibly be only due to the separation of the hidden states and classification layers.

5.3.3 Extended Hierarchical

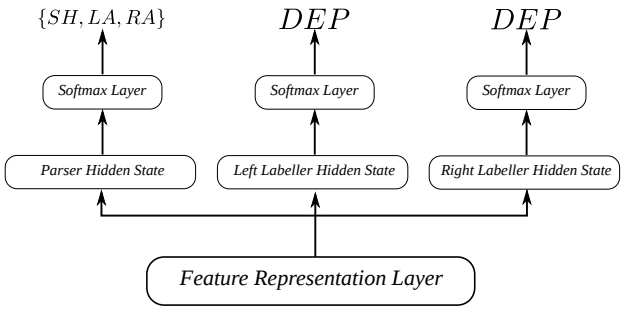


Figure 5.3: The Extended hierarchical architecture.

We designed this architecture in order to investigate whether or not a similar improvement could be gained from separating Left-Arc labelling and Right-Arc labelling. To do so we split the dependency labelling layer into 2, bringing the total number of output neurons to be the same as for the Joint state architecture, but with a separate hidden state for each classification layer. As with Hierarchical classification, the feature representation layer is shared between all three classifiers.

The transition made would depend on the scores produced by the parsing output layer, as usual, but the label given to that arc will depend on whether a right arc or a left arc

was performed, and the final scores will be used from the appropriate labeller.

5.3.4 Successive

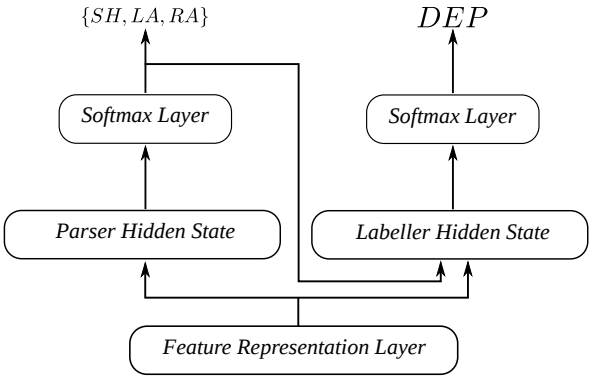


Figure 5.4: The Successive architecture.

The use of only a single output layer to assign dependency labels raises the question, how does the labeller distinguish between a left-arc and a right-arc? If it does not, does that mean that the distinction is unnecessary? To answer this question we designed a new architecture that is built the same as the hierarchical architecture, i.e. with a single parsing component and a single labelling component with a shared representation layer. However, the output of the parsing output layer is then passed to the hidden state of the labelling component.

If this modification produces an accuracy better than that of a hierarchical parser, then this would indicate that the labelling component is indeed dependent on the parsing component, or at the very least can benefit further from the information that it provides.

5.3.5 Separate

Our main question in this chapter is whether Dependency Parsing and Dependency Labelling are two separate, independent tasks. To test for this we also train two separate networks in parallel, a parser and a labeller. This is essentially a further separation of the two components of the hierarchical architecture, where the parsing and labelling compo-

nents no longer share the feature representation, and so get to tune this layer to present information only useful to them.

While both components are separate, they are still trained together. At train time this means that the final cost function is identical to that used in the hierarchical architecture, which incorporates the scores of both components to calculate the error. While using separate cost functions for each component is more intuitive, using a combined cost function makes the architectures more comparable, and limits the interpretation of our results strictly to the usefulness of encoding each task's information separately.

During parsing time, this split means that after the parser makes a new attachment, a dependency label is assigned to it by the labeller. Given the feature set defined in Section 5.2, there remains one small area of dependence, which is that both the parser and labeller use structural features. These features dictate the order in which both components are used, since the parser uses the dependency labels of some structured features as input, which means that the next attachment cannot be decided until a dependency label has been assigned to the current one. On the other hand, the labeller's use of structural features also means that it must wait for the parser to decide on a transition for the previous step.

In order to remove the dependence of the parsing component on the labelling one, we later experiment with a version of the parser that does not use dependency labels as features, but keeps the rest of the feature set the same. Chen and Manning (2014) showed in ablation studies that removing the labelling features resulted in a slight drop in accuracy, but their model used a joint state architecture. Conversely, there are a number of state-of-the-art parsers that do not use dependency labels as features, but depend on a more robust feature representation layer (Shi et al., 2017; Kiperwasser and Goldberg, 2016b).

We do not attempt to remove the dependence of the labeller on structural features, however, since the final outcome would still require either a partial or full tree, and would need a more far reaching modification to the architecture, and so is left for future work.

This set up can have a number of outcomes, each having separate but useful implications. We examine the broadest here and analyse them further with the final results in Section 5.5.

If this method fails to match the accuracy of the Joint state architecture, then parsing and labelling are indeed dependent on each other, and encode information that is beneficial to the performance of both. If it matches or exceeds it, however, then this would mean that parsing and labelling are in fact independent, but the implications of this would be depend on how it compares with Hierarchical classification.

If training the two components separate networks results in accuracies that match or exceed that of the Hierarchical architecture, then this would further support the case for separate training, in addition to indicating that the joint feature representation layer might hinder performance, as we speculated earlier. If this is not the case, however, then the reverse is definitely true, that while parsing and labelling can be trained independently, a joint feature representation layer, but not a joint hidden state, is ultimately capable of encoding useful information for both tasks.

5.4 Implementation & Training

In our implementation we used the same dimensions of embeddings for both types of feature representation, with both part-of-speech tags and dependency labels having dimensions of size 50, and word vector embeddings being of size 100. For word embeddings we used pretrained GloVe vectors (Pennington et al., 2014), while the vectors for dependency labels and part-of-speech tags were randomly initialised. For the positional representation, we use two bi-LSTM layers, with 256 LSTM units representing each direction.

All hidden states, regardless of architecture, use a single hidden layer of size 256. The neurons in this layer use rectified linear units (ReLUs) (Nair and Hinton, 2010) as activation functions. We set a dropout rate of 0.3 on all LSTMs (Gal, 2015) and

the hidden layer (Hinton et al., 2012). All weights and pos tag vectors were initialised uniformly (Glorot and Bengio, 2010).

For training we use a negative log likelihood loss function where y_i is the combined transition/dependency score for the joint state model, t_i is the transition score, and d_i is the dependency score. θ represents all model parameters, which includes all weights and vector embbeddings used. We use mini-batch updates of 10 sentences, and stop training after 40 epochs for models using the embeddings representation, and 30 epochs for models using positional representation. We optimise the model parameters using Adam (Xu et al., 2015) with a learning rate $\alpha = 1 \times 10^{-}3$. The loss function for the joint state model is ...

$$L(\theta) = -\sum_{i} log(y_i)$$

and for all other models is ...

$$L(\theta) = -(\sum_{i} log(t_i) + \sum_{i} log(d_i))$$

For hierarchical, successive, and separate architectures, we do not calculate error when a shift transition occurs. Similarly, for the extended hierarchical architecture, we only calculate cost for left/right-arc labels only when their corresponding transitions are made.

In addition, due to its architecture, the parser component in successive architecture models also receives back-propagated error updates from the hidden state of the labeller component.

We train our models using the Wall Street Journal (WSJ) section from the Penn Treebank (Marcus et al., 1993). We use §2-21 for training, §22 for development, and §23 for testing. We use Stanford Dependencies (SD) (De Marneffe et al., 2006) converted from constituency trees using version 3.3.0 of the converter. As is standard we use predicted POS tags for the train, dev, and test sets. We report unlabelled attachment score (UAS) and labelled attachment score (LAS), with punctuation excluded. The models are tuned on the development set, with the tuning that produced the highest UAS used to obtain the

final scores on the test set. All our models were implemented using the DyNet framework (Neubig et al., 2017) in python.

5.5 Experiments & Results

As mentioned in Section 5.2, we use two feature representation methods in order to have a clearer understanding of the effects of each architecture on the overall performance of the parser/labeller. Additionally we will initially choose works from current literature as an external baseline of comparison for our work. A more complete comparison with the state-of-the-art is present later in Table 5.4.

For experiments using the embeddings representation we chose Chen and Manning (2014) as our baseline, which has served as the basis for much of the work done on feed-forward networks since (Weiss et al., 2015; Andor et al., 2016; Zhou et al., 2015). The results of our experiments for all our architectures using the embeddings representation are shown in Table 5.1.

	Dev		**Test**	
Architecture	UAS	LAS	UAS	LAS
Joint state	92.23	89.74	91.73	89.50
Hierarchical	**92.55**	**89.92**	**92.27**	**90.09**
Ext. Hierarchical	91.51	88.86	90.73	88.38
Successive	91.80	89.25	91.51	89.04
Separate	92.53	89.81	92.08	89.82
Chen and Manning (2014)	92.00	89.70	91.80	89.60

Table 5.1: Dev and test set scores on WSJ (SD) using embeddings feature representation.

The results for our basic joint state model are very close to the baseline Chen and Manning (2014) model, and with the exception of the UAS score on the development set, the differences are negligible. They both use the same architecture and our model uses a slightly larger hidden layer size of 256 versus 200 for Chen and Manning (2014).

As was expected, our hierarchical model outperforms the joint state and our chosen baseline achieving a (UAS/LAS) development score of (92.55/89.92) compared to

(92.23/89.74) for our joint state model, and a more notable difference in performance on the test set with scores of (92.27/90.09) compared to (91.73/89.50), making for roughly 0.5% gain in the final score.

The extended hierarchical model, on the other hand, produced substantially worse results than both the hierarchical and joint state models. The development scores of the extended hierarchical model fell to (91.51/88.86), while the test scores fell even further to (90.73/99.39). The result is an approximate drop of 1% in accuracy on the development set and an almost 2% drop on the test set when compared to the hierarchical model, meaning that its ability to generalise to data outside the development set was also impacted.

Our successive model fared slightly better, but still performed worse than both the joint state and hierarchical models. The drop in LAS is slightly more than for UAS but, unlike the extended hierarchical model, its generalising ability was not harmed with a development score of (91.80/89.25) and a test set score of (91.51/89.04).

Separately training the parser and labeller seems to have been a success, surpassing the baseline and the joint state, as well as approaching the performance of the hierarchical model on the development set scoring (92.53/89.81), but remaining behind on the test set scores (92.08/89.82), especially on the UAS score. This model still surpassed all the other architectures.

For experiments using the positional feature representation, we chose the results of Kiperwasser and Goldberg (2016b) as our baseline for comparison. We noted that Cross and Huang (2016) explicity tested for the effects of hierarchical classification and, like our work here, use Arc-Standard. They do, however, use a minimal number of features, unlike Kiperwasser and Goldberg (2016b) who use an extended feature set similar to ours, in addition to also using a hierarchical set up. We do differ, however, in our use of the extra dependency label features, and in that we use Arc-Standard, where Kiperwasser and Goldberg (2016b) use Arc-Hybrid (Yamada and Matsumoto, 2003; Gómez-Rodrıguez et al., 2008; Kuhlmann et al., 2011b) with a dynamic oracle.

The results of the joint state model for this feature representation are surprisingly high

	Dev		Test	
Architecture	UAS	LAS	UAS	LAS
Joint state	94.04	91.95	94.13	92.17
Hierarchical	**94.27**	**92.17**	**94.30**	**92.23**
Ext. Hierarchical	94.17	92.02	94.20	92.17
Successive	94.23	92.10	94.16	92.05
Separate	94.03	91.67	94.04	91.95
Kiperwasser and Goldberg (2016b)	93.8	91.5	93.9	91.9

Table 5.2: Dev and test set scores on WSJ (SD) using positional feature representation.

when compared with the baseline, where the development scores are (94.04/91.95) compared to (93.8/91.5), and the test scores are (94.13/92.17) compared to (93.9/91.9). The joint state model scores higher than our baseline despite the use of a hierarchical architecture in Kiperwasser and Goldberg (2016b). This could be a combination of two factors, first is our use of a larger positional vector (256 LSTM-units in each direction vs 125), and second is the difference in the parsing systems used, where Shi et al. (2017) showed that Arc-Standard can slightly outperform Arc-Hybrid using this feature representation, when using the same training methods.

As was the case when using the embeddings representation, our hierarchical model outperforms both the joint state model and the baseline. It does so, however, by a smaller margin for positional representation, achieving development scores of (94.27/92.17) and test scores of (94.30/92.23), with difference for the test LAS being negligible.

The extended hierarchical architecture still suffered a drop in accuracy compared to the hierarchical model, but surprisingly not by much, considering the dramatic difference produced in the embeddings representation experiments. The positional representation even allows the extended hierarchical architecture to slightly beat the joint state model with (94.17/92.02) for the development set, and (94.20/92.17) for the test set. The difference in scores with the joint state model is not substantial, and so can optimistically be seen as the positional representation providing a rich enough encoding of the sentence, that the performance of the parser was not degraded.

For the successive architecture, the positional representation continues to prove re-

markably resilient, achieving a development set score of (94.23/92.10) which is slightly below that of the hierarchical model, but negligibly so. The model, however, scores (94.16/92.05) on the test set which is less than the extended hierarchical model.

Surprisingly, separate training produced worse results than all our other architectures, when using positional features. The model achieved a UAS score similar to the joint state, but still fell behind on labelled accuracy with (94.03/91.67) on the development set, and (94.04/91.95). This translates to approximately a 0.3% drop in accuracy when compared to the hierarchical model, although this model still managed to surpass the scores of the baseline.

As mentioned in Section 5.3.5, we also conducted experiments for the separate architecture where the dependency label features were removed. If the performance of the parsers is not impacted, then this would completely remove any dependence the parsing component has on the labelling component. The results are shown in Table 5.3.

	Dev		**Test**	
Feature Type	UAS	LAS	UAS	LAS
Embeddings				
+ dep. labels	**92.53**	89.81	92.08	89.82
− dep. labels	**92.53**	**89.93**	**92.21**	**89.92**
Positional				
+ dep. labels	94.03	91.67	**94.04**	**91.95**
− dep. labels	**94.28**	**91.86**	94.03	**91.95**

Table 5.3: Dev and test set scores on WSJ (SD) comparing parsers using a separate architecture, with and without dependency label features, for both feature representations.

The effects of removing the dependency label features paint an interesting picture. For the embeddings representation-based model the development scores resembled that of the model that uses dependency label features with (92.53/89.93). It did however generalise better to the test set scoring (92.21/89.92), which brings it closer to the hierarchical model.

For the models using positional feature vectors, removing dependency label features improved the accuracy on the development set to (94.28/91.86), which is a UAS score matching that of the hierarchical model, but still lagging behind on LAS. As for test set

scores, however, the results were almost identical to the model that uses dependency label features, scoring (94.03/91.95), which remains below all other architectures trained that use positional feature vectors.

	Dev		**Test**	
	UAS	LAS	UAS	LAS
This work				
Hierarchical + Embeddings + Dependency label features	92.55	89.92	92.27	90.09
Separate + Embeddings + Dependency label features	92.53	89.81	92.08	89.82
Separate + Embeddings − Dependency label features	92.53	89.93	92.21	89.92
Hierarchical + Positional + Dependency label features	94.27	**92.17**	94.30	92.23
Separate + Positional + Dependency label features	94.03	91.67	94.04	91.95
Separate + Positional − Dependency label features	94.28	91.86	94.03	91.95
Chen and Manning (2014)	92.00	89.70	91.80	89.60
Weiss et al. (2015)	N/A	N/A	93.99	92.05
Dyer et al. (2015)	93.2	90.9	93.1	90.9
Andor et al. (2016)	**94.38**	**92.17**	**94.61**	**92.79**
Kiperwasser and Goldberg (2016b)	N/A	N/A	93.9	91.9
Cross and Huang (2016)	93.67	91.48	93.42	91.36
Shi et al. (2017)	93.92	N/A	94.53	N/A

Table 5.4: A comparison of dev and test set scores on WSJ (SD) with our external baselines and some of the highest scoring transition-based dependency parsers in the current literature.

5.6 Discussion

We can observe the same positive effect of the hierarchical structure over the joint state structure for both of the feature representation methods used, confirming the results of (Cross and Huang, 2016). The performance of our proposed architectures is also illuminating in a number of ways.

The large drop in the result of the extended hierarchical model, even below that of the joint state when using the embeddings representation, is an interesting result in itself. The simplest explanation could be that the split labellers receive less updates than a single labeller would, meaning that they would in effect have less training and updates. On the other hand, the split labellers have a theoretical advantage in that the position of the head and dependent are fixed in their input, while their position would alternate for the single labeller in the hierarchical architecture.

This raises the question, is the dependency label relation undirected, notwithstanding the specification of the head and dependent? Or could it be that the range of possibilities given a pair of words is already limited? Consider words such as “a” & “the”. These words would always be modifiers of a parent, and would never have dependents themselves. More general patterns along these lines could conceivably be the case, making split labelling unnecessary.

The results of the successive architecture seem to indicate that parsing and labelling might be separate. Using the transition scores as input to the labeller did not help improve labelling, but did not severely impact it either. In fact, its effect is almost nullified by the use of positional feature representation.

Separately training the parser and labeller, while producing successful results, did provide contradictory outcomes. For both feature representation methods there was a drop compared to the corresponding hierarchical models, but it was much more substantial for the positional representation experiment. We speculate that this is a consequence of a combination of the denser encoding of the sentence and the use of dependency label features.

The fact that models with a shared positional representation layer performed better in this instance seems to point to the dependency features being useful to one of the components. A further investigation with different features for each component would shed more light on this discrepancy, and is left for future work.

Retraining the separate models without dependency label features produced mixed results. While it definitely improved the accuracy of separate models on the development set, it only improved the test set accuracy for embeddings-based models, with no improvement at all for models using positional vectors. Both models still produce strong results that beat their respective baselines, however they do not beat their hierarchical counterparts.

This seems to point in the direction of a joint feature encoding layer, as used by hierarchical models, being able to encode useful information for both sides, and that helps the final model generalise better, as evidenced by how hierarchical model accuracies on the test set outperform the separate models despite close development accuracies.

In Table 5.4 we compare our results with some of the strongest works in current literature. We, however, limit our comparison to other transition-based dependency parsers. Despite the fluctuations of the various architectures, our positional models are generally quite competitive, and are only beaten by Andor et al. (2016) and Shi et al. (2017). Both of these high performing parsers use global models for training, as opposed to our greedy models trained with only local features. Despite this, our hierarchical model comes close to matching Andor et al. (2016) on the development set, with scores of (94.27/92.17) compared to (94.38/92.17).

5.7 Conclusion & Future work

In this chapter we have shown that dependency parsing and dependency labelling can be successfully trained separately. And while training a parser and labeller separately as we have presented here does not offer any clear benefits over hierarchical classification

in terms of accuracy scores, laying out the two tasks in this way is illuminating and has potentially useful implications that we plan to explore in future work.

The effect of removing the dependency label features is very interesting, especially when contrasted with the reported drop in performance when removing these features in Chen and Manning (2014). This, together with the performance of our hierarchical models, means that perhaps more gains can be made by changing the features extracted from the feature representation layer for each task.

One interesting result of this split is that labelling can now be done on fully built dependency trees, and does not need to take into account the specifics of transition parsing itself. While the way of calculating labelled accuracy scores ensures that it cannot surpass the unlabelled attachment score, since a label cannot be correct if the arc itself is incorrect, training a labeller without the interference of the parser can help close the considerable gap between the two, which can be readily observed in the scores of the state of the art systems shown in Table 5.4.

Our separate parsers still shared one link, and that is the joint cost function that was common between them and the other architectures used. We believe that this could have an interesting impact since the results of the extended hierarchy, and the gain in performance from removing dependency label features seem to point towards removing any influence of either side on the other.

Finally our experiments were all conducted on English. Investigation into the use of these architectures on other languages would provide a more complete picture.

CHAPTER 6

RECURSIVE LSTM TREE REPRESENTATION

6.1 Introduction

In chapter 5 we took advantage of bi-LSTM representation to produce positional vectors, but found that it was naturally limited to modelling sequential features of the sentence, and was incapable of incorporating structural information relating to the dependency tree itself. For this reason we augmented this representation method with the relevant label embeddings that complemented the selected positional features, and achieved competitive results.

Another approach has been to represent the dependency tree itself with some form of recursive network, either bottom-up as in (Dyer et al., 2015; Kiperwasser and Goldberg, 2016a; Stenetorp, 2013), or top-down as in (Le and Zuidema, 2014), with the latter achieving strong results.

In this chapter we approach feature representation in a very general manner, attempting to incorporate features from all previous work. We propose a new method of recursively modelling dependency trees using LSTMs, which we call Recursive Tree LSTMs. Our experiments show that this method of representation is very powerful, and can even be used as an additional layer of encoding over bi-LSTM feature representation, which results in a more informative model.

The resulting parsers show an ability to model both positional and structural infor-

mation without the need for external augmentations, and as a consequence require even fewer features than both existing tree representation methods and bi-LSTM representation. The final parser is capable of achieving competitive results with a feature set consisting of only the top two items on the stack, which is the smallest feature set for an Arc-Standard dependency parser used successfully to date.

6.2 Recursive LSTM Trees

We propose a method of representing a dependency tree as a single dense vector that results from the repeat application of an encoding mechanism to sequences of head-dependent pairs. The architecture of the encoding mechanism can vary and we discuss some options in detail in Section 6.3.

Each node in the tree represents a head token's interaction with the representations of all of its immediate dependents. Similarly, the representations of each of these dependents are themselves the result of the interaction between their token and the representations of their corresponding dependents.

Each token has 2 representations, a vector representation v and a tree representation τ. The vector representation is the raw description of a token in its sentence, which in the most basic form can simply be the concatenation of the word and part-of-speech vectors of that token. We do however show later in Section 6.4 that there is room to enrich v further by using a positional vector representation.

The tree representation of a token, on the other hand, encodes the dependency information of a token and its dependents. Consider a simple subtree consisting of a head token h and its dependents a, b, and c as illustrated in Figure 6.1. The subtree is represented as a sequence of pairs of head vectors (h_v) and child trees (a_τ, b_τ, c_τ). These pairs are then input to the encoding mechanism with the final output being the head tree vector h_τ.

The first pair in the sequence representation is always $(h_v, < S >)$, where $< S >$ has the same size as the output size of the encoding mechanism and represents the start tag

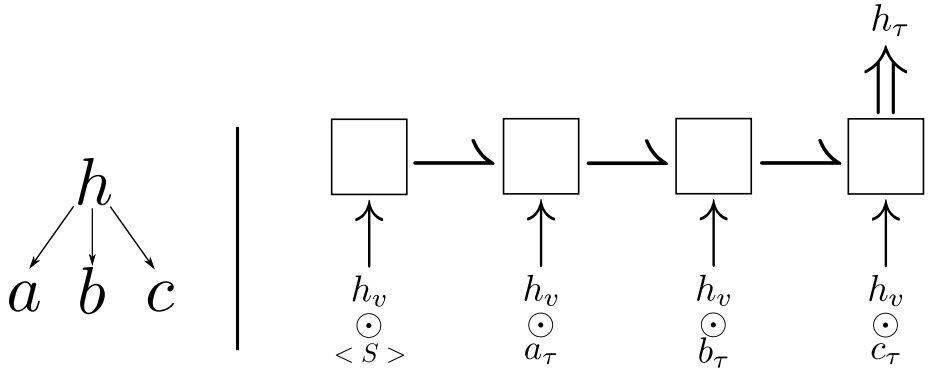


Figure 6.1: A compact representation showing how a subtree (left) is arranged as a sequence to produce a tree vector for the head token h_τ (right). The encoding mechanism here is a single forward LSTM. The operation $\odot$ is concatenation, $\uparrow$ is input, $\rightharpoonup$ is the passing of the internal state from one LSTM step to another, and $\Uparrow$ is the output of the LSTM.

of the sequence. This also serves as the base case for leaf nodes in the dependency tree as well as for tokens without dependents in partially built trees while parsing, where they would otherwise be represented with a zeroed out vector. The encoding mechanism in this example is represented as a single forward Long Short-term Memory network (LSTM) (Hochreiter and Schmidhuber, 1997).

Each input pair uses the same h_v which is then concatenated with the tree representation of the dependent. The dependents are presented in their order of appearance in the sentence, and the encoding mechanism output at each step can be taken to represent the subtree of h including the dependents introduced up to that step. The recursive element of this formulation is the repeat application of the encoding mechanism, in a bottom-up approach, in order to produce tree representations for tokens that are then used in turn to produce the tree representations of their corresponding heads.

An illustration of this idea is shown in Figure 6.2. Here the head token h has one dependent who also has dependents and another which has none. b_τ is represented by the base case with $(b_v \odot < S >)$, while a_τ requires 2 additional steps to incorporate information from x_τ and y_τ.

As is apparent in Figure 6.2, a_τ and b_τ must be calculated first before h_τ can be produced, and by extension x_τ and y_τ are required first in order to calculate a_τ. In this

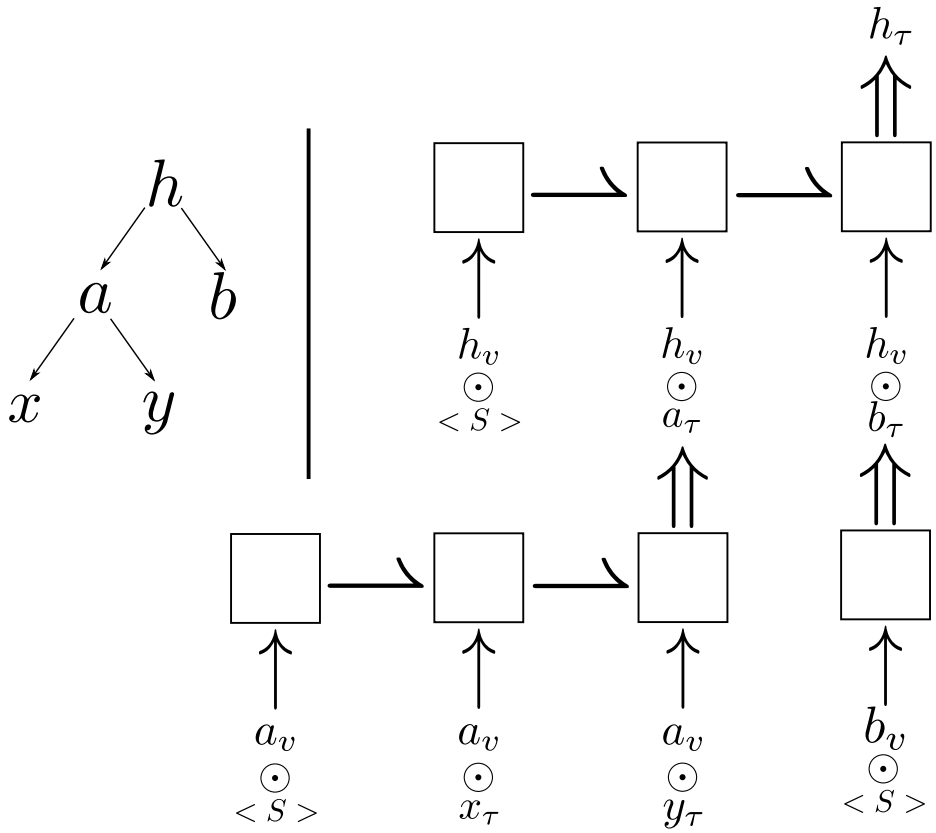


Figure 6.2: A deeper subtree that shows the recursive application of the encoding mechanism across different depths in the subtree.

way the final dependency tree representation is built recursively, bottom-up, with the final representation being the tree representation $ROOT_\tau$, whose only dependent would be the main verb of the sentence. An example of a full dependency tree encoding is presented in Figure 6.3.

6.3 Sub-tree Encoding

As stated in Section 6.2, the encoding mechanism is the architecture used to convert a head vector h_v and a set of dependent subtrees $\{d_{0\tau}, d_{1\tau}, \cdots, d_{n\tau}\}$ into a tree representation h_τ. The structure of RLTs makes them largely independent of the method of encoding used, so long as they can accept an arbitrary number of dependents.

Put simply an encoding mechanism should accept the sequence in Figure 6.4b and produce an h_τ which will be used to represent Figure 6.4a.

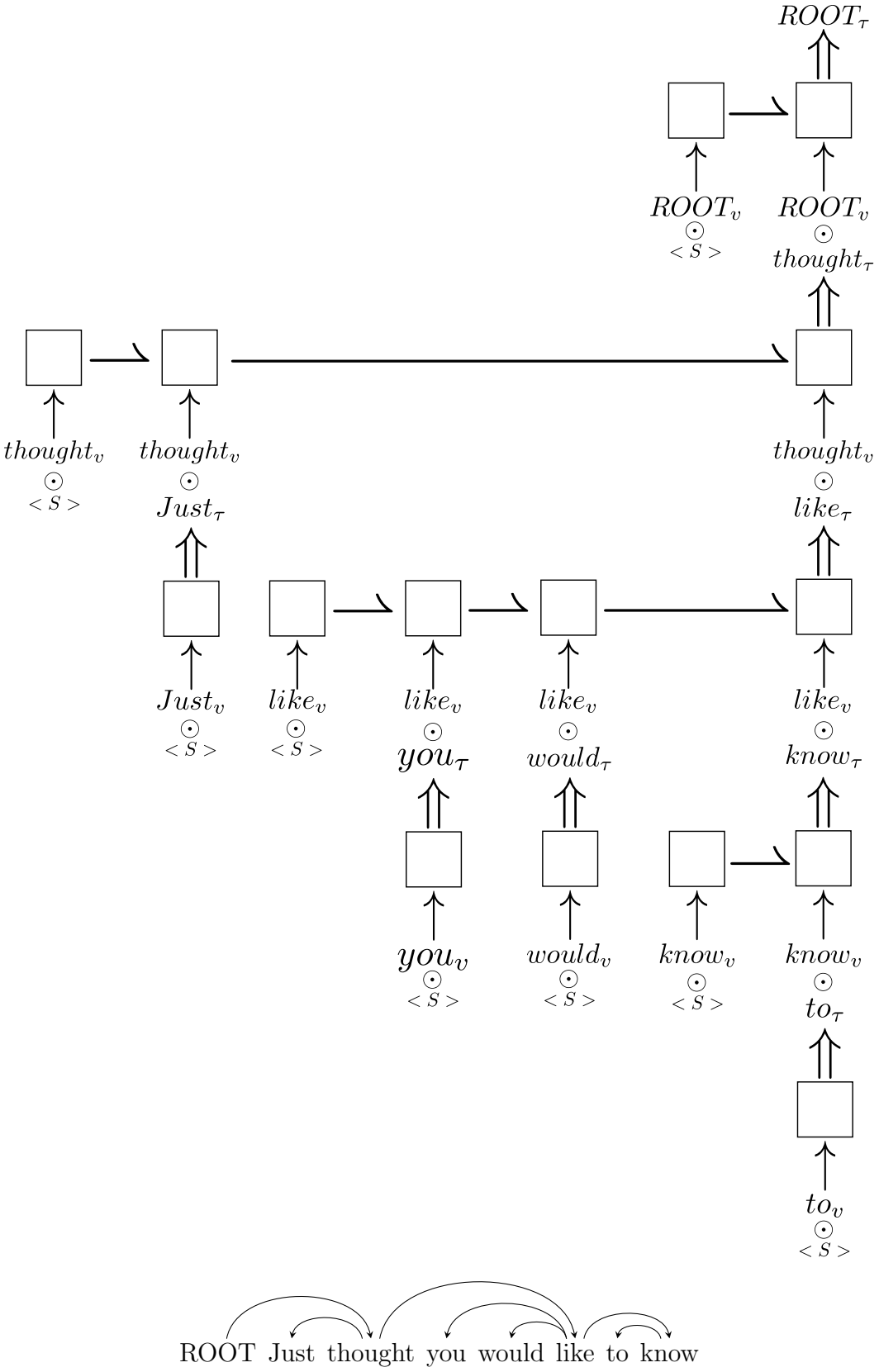
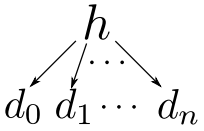


Figure 6.3: An example of an RLT encoding (top) of a dependency tree (bottom).



(a) Source subtree

$\{h_v \odot < S >, h_v \odot d_{0\tau}, h_v \odot d_{1\tau}, \cdots, h_v \odot d_{n\tau}\}$

(b) Input sequence representation

Figure 6.4: The Encoding Task: A simple subtree represented as a sequence. v and τ are the vector and tree representations respectively. $< S >$ is the start tag, and $\odot$ is the concatenation operation.

The encoding task stated in this way allows for great flexibility in constructing an encoding mechanism, so we have explored the use of some of the more intuitive possibilities.

6.3.1 Forward Encoding

The examples given thus far have all used a forward LSTM to encode subtrees. The final tree representation would be the output of the LSTM at the step where the last element in the sequence was entered.

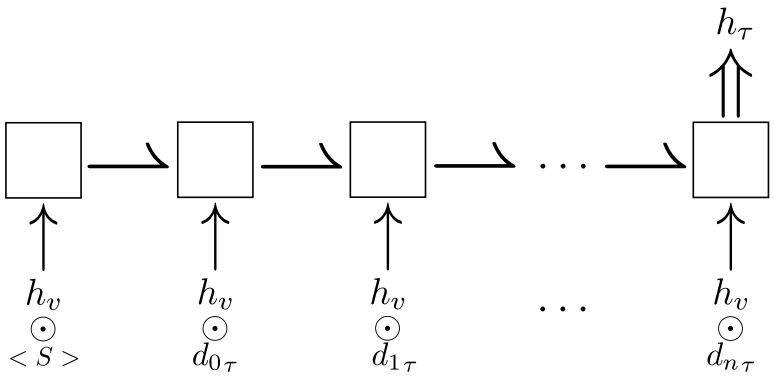


Figure 6.5: Forward Encoding

This method is particularly computationally cheap and quite effective when compared with the next 2 encoding methods. If it could be assumed that dependents are added in order, then this mechanism need only store the last state of the LSTM and h_v. In practice, however, this only holds true for RA transitions whose dependents are guaranteed to be in order for all parsing systems discussed in Section **??**. On the other hand, LA transitions add a new left-most dependent, which translates to a new entry at the start of the sequence

representing a subtree. This means that the whole subtree needs to be recalculated in the event of an LA, and so the mechanism effectively must store the entire sequence.

One solution to this is to not worry about the order of the dependents in the subtree, which would negate the need for any recalculation and we would only need to store the last state of the LSTM. We found, however, that the order of the dependents does appear to hold information that impacts the overall final accuracy of the RLT.

Another possible compromise is the use of the Arc-Hybrid system (Yamada and Matsumoto, 2003; Gómez-Rodrıguez et al., 2008; Kuhlmann et al., 2011b) as described in Section 2.2.2. The restriction of performing all LA transitions first would indeed ensure that recalculation is done for the shortest possible sequences. We did find that training an Arc-Standard parser to prioritise LA transitions was enough to approach this behaviour without impacting its final accuracy.

6.3.2 Bi-directional Encoding

Representing a word in a sentence as the output of a bi-directional LSTM (Graves and Schmidhuber, 2005) was shown by Cross and Huang (2016) and Kiperwasser and Goldberg (2016b) to be a better descriptor since it took into account relevant information from other words in the sentence. This had the added bonus of reducing the number of features required to encode the same information. This encoding mechanism is inspired by this same idea, except that we need to produce an encoding that represents the entire sequence at once. To this end we run a bi-directional LSTM (bi-LSTM) over the sequence and concatenate the output vectors at both ends to represent h_τ.

6.3.3 Compositional Encoding

For this mechanism we adapt the method used by Kuncoro et al. (2016a) to encode parts of a constituency tree. Kuncoro et al. (2016a)'s composition function was originally designed to encode a constituency relation, in this case, however, we use the same strategy with

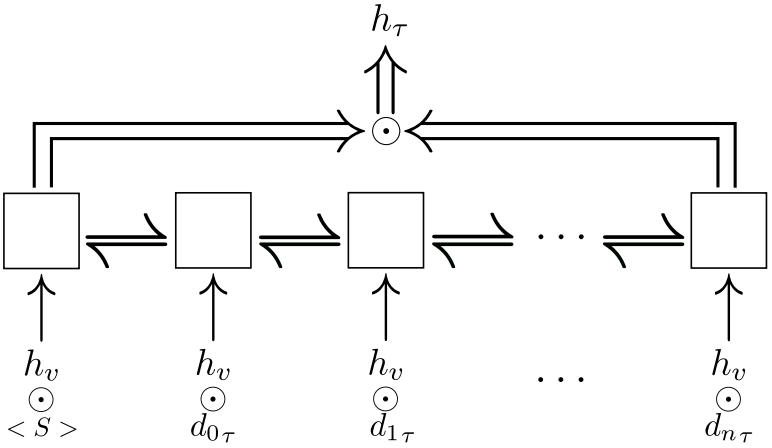


Figure 6.6: Bi-directional Encoding

the start tags taking the place of the constituency relation.

This method uses forward and backward encoding of the sequence, but does not assume that the position of the start tag at the front of the sequence is most informative. To achieve this the forward and backward sequences are encoded separately and their outputs are concatenated to give h_τ.

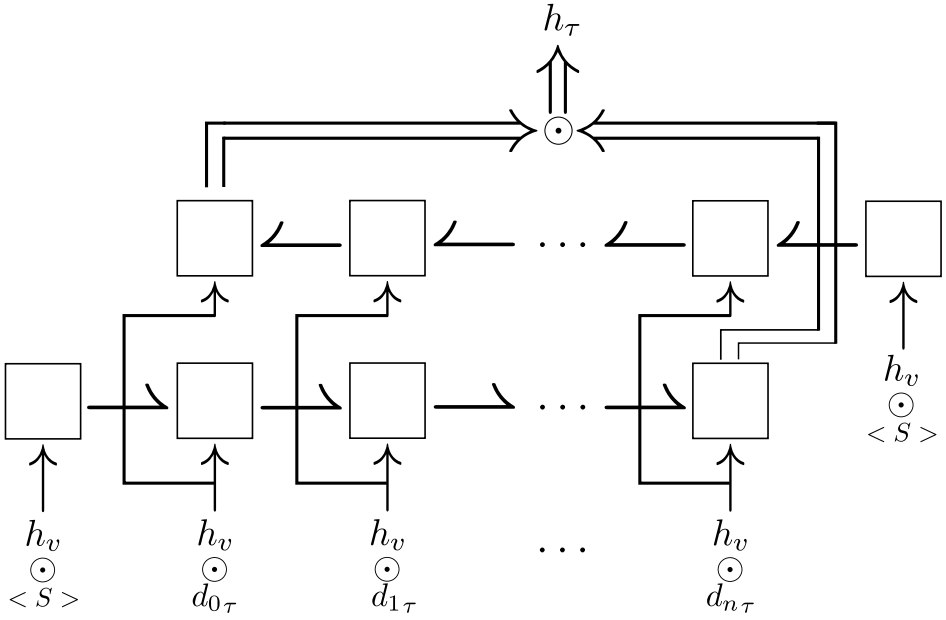


Figure 6.7: Compositional Encoding

The use of a backward pass for both the bi-directional and compositional encoding mechanisms has a few implications. The first is the doubling of the computation cost compared to that of Forward encoding, since the later only requires a single forward LSTM.

The second is that the backward pass makes both bi-directional and compositional encoding incapable of producing an intermediate representation that can be carried forward to future time-steps. This is because every addition to the tree is the first input (or second in the case of compositional encoding) to the backward pass, which changes the outputs for all other steps.

Given this computationaly expensive setup, the number of times that a sequence is calculated must be reduced. Since the latter 2 mechanisms use a backward pass, the order in which children are added does not matter since an entire recalculation of the sequence would be needed anyway. Hence there is nothing to gain from the strict order of addition imposed by the Arc-Hybrid system, discussed in Section 2.2.2.

Another change that can trigger a recalculation is adding dependents at a deep level in the subtree. Such an addition not only invalidates the sequence that it has been added to, but also invalidates every sequence that its head is dependent on. This cascading effect can require the recalculation of significant parts of an RLT, making an expensive operation far costlier. It is for this reason that we prefer a bottom-up parsing system, and so have ruled out using Arc-Eager.

6.3.4 Vector Representation (v)

The encoding mechanism relies on a base representation for the head, the vector representation v. The simplest version of this would be to simply concatenate the word vector and part-of-speech tag vector.

$$h_v = h_w \odot h_t$$

However, positional representation has been shown to be a richer, more informative feature about a token and its position in a sentence (Cross and Huang, 2016; Kiperwasser and Goldberg, 2016b). This approach rests on using a bi-LSTM to encode the whole sentence, with the concatenated outputs of the forward and backward LSTMs at the token's position

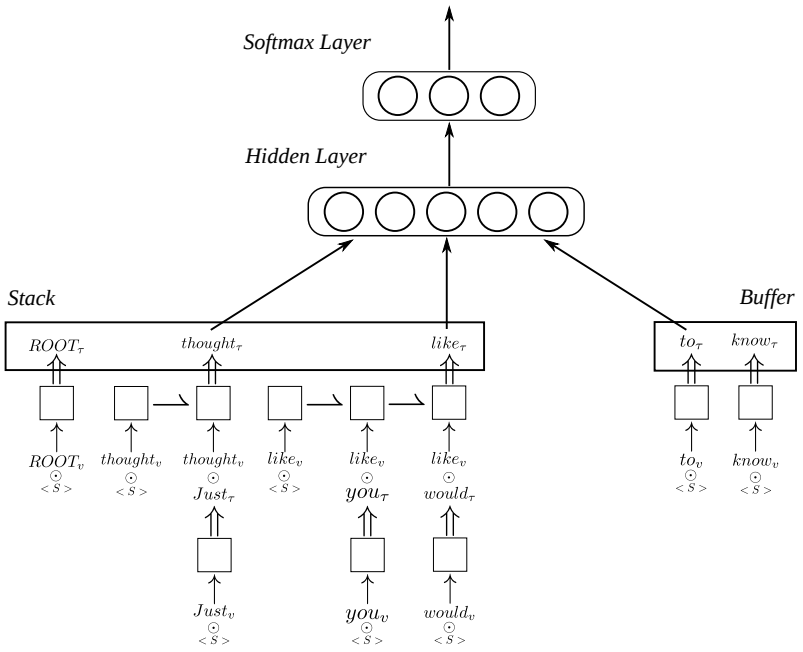


Figure 6.8: Example of a parser configuration with features using RLTs.

taken to be its positional vector.

We experiment with both approaches and confirm that a positional vector does improve the performance of RLTs, in addition to its properties being carried over to the RLTs themselves, meaning that parsing can be done with minimal features.

6.4 Implementation & Training Details

We implemented our model in python using the DyNet framework (Neubig et al., 2017). The encoding mechanisms used by the RLTs in our experiments used 2 layers of LSTMS/bi-LSTMs, depending on the mechanism. For RLTs using Forward encoding we used LSTMs of size 512, while for Bi-directional and Compositional encoding we used LSTMs of size 256 in each direction. For experiments using a bi-LSTM positional representation, we also used 2 layers of bi-LSTMs of size 256 in each direction. For the basic vector representations we used randomly initialised part-of-speech tag vectors of size 50, and for word embeddings we used vectors of size 100 initialised using the pretrained GloVe vectors (Pennington et al., 2014).

The tree vectors of relevant RLTs are then concatenated and passed as input to a feed-forward hidden layer of size 256, with rectified linear units (ReLUs) (Nair and Hinton, 2010) as activation functions. We set a dropout rate of 0.3 on all LSTMs (Gal, 2015) and the hidden layer (Hinton et al., 2012). In our experiments we tried different dropout rates, but the differences were too small to experiment with separate dropout rates for different layers. The final output layer is a softmax output layer with the same structure as in the setup of Chen and Manning (2014), in which the scores correspond to $(SH, \{LA, RA\} \times DEP)$, where DEP is the set of all possible dependency labels. All weights and pos tag vectors were initialised uniformly (Glorot and Bengio, 2010).

For training we use a negative log likelihood loss function where y_i is the transition score. We use mini-batch updates of 10 sentences, and stop training after 30 epochs. We optimise the model parameters using Adam (Xu et al., 2015) with a learning rate $\alpha = 1 \times 10^{-}3$.

$$L(\theta) = -\sum_{i} log(y_i)$$

We train our models using the Wall Street Journal (WSJ) section from the Penn Treebank (Marcus et al., 1993). We use §2-21 for training, §22 for development, and §23 for testing. We use Stanford Dependencies (SD) (De Marneffe et al., 2006) converted from constituency trees using version 3.3.0 of the converter. As is standard we use predicted POS tags for the train, dev, and test sets. We report unlabelled attachment score (UAS) and labelled attachment score (LAS), with punctuation excluded. The models are tuned on the development set, with the tuning that produced the highest UAS used to obtain the final scores on the test set.

6.5 Experiments & Results

We have described 3 encoding mechanisms to produce h_τ and 2 sources for the basic vector representation h_v in Section 6.3. In this section we present the results of our

experiments with these combinations, in addition to experiments with varying feature sets. We compare our results to initially to those of Dyer et al. (2015) and Kiperwasser and Goldberg (2016a), since they are the closest in the literature to our approach. We make a more complete comparison with state of the art Transition-based parsers in Table 6.4.

For our initial set of experiments we trained models that used the top 4 RLTs on the stack, and the front 4 on the buffer as input features to the feed forward hidden layer. This setup is influenced by work on the simple feed forward network, where successful networks used the top 3 from each data structure as in (Chen and Manning, 2014), or the top 4 from each as in (Weiss et al., 2015). These examples used an input layer that was a concatenation of the word and pos vectors of these features in addition to structural features describing dependents. This is not necessary with RLTs that model the entire subtree and so their corresponding h_τ represents all of this extended information.

	Dev		**Test**	
Encoding Type	UAS	LAS	UAS	LAS
Forward	**93.45**	**91.09**	93.06	90.93
Bi-directional	93.28	**91.09**	93.04	**91.01**
Compositional	93.30	90.94	92.96	90.86
Kiperwasser and Goldberg (2016a)	93.3	90.8	93.0	90.9
Dyer et al. (2015)	93.2	90.9	**93.1**	90.9

Table 6.1: Dev and test set scores on WSJ (SD) using an h_v that is a concatenation of the tokens word vector and pos tag vector.

When setting h_v **to be the concatenation of the word and pos vectors**, the resulting accuracies of the encoding mechanisms, shown in Table 6.1, are very similar. The Forward encoding mechanism achieves only a slightly better accuracy with dev scores of (93.45/01.09) and test scores of (93.06/90.93), but with negligible differences between it and other mechanisms. Nevertheless all three mechanisms largely match the performance of Dyer et al. (2015) and Kiperwasser and Goldberg (2016a), with Dyer et al. (2015) having a slightly better test accuracy of (93.1/90.9). It is interesting to note that neither the backward pass, in Bi-directional and Compositional encoding, nor the alternating

	Dev		Test	
Encoding Type	UAS	LAS	UAS	LAS
Forward	93.85	91.67	93.61	91.65
Bi-directional	**94.20**	**91.94**	**93.79**	**91.86**
Kiperwasser and Goldberg (2016a)	93.3	90.8	93.0	90.9
Dyer et al. (2015)	93.2	90.9	93.1	90.9

Table 6.2: Dev and test set scores on WSJ (SD) using a bi-LSTM positional vector as h_v.

start tag in Compositional encoding provided any benefit to modelling a sequence using this kind of h_τ. For the rest of our experiments we limit our examination to Forward and Bi-directional encoding.

For the second set of experiments **we use the bi-LSTM positional representation as** h_v. The results are shown in Table 6.2. The Bi-directional encoding model clearly comes out ahead here, with dev scores of (94.2/91.94) and (93.79/91.86). Forward encoding also benefited substantially from the richer representation reaching dev scores of (93.85/91.67) and test scores of (93.61/91.65). Both mechanisms now well outperform the results of Dyer et al. (2015) and Kiperwasser and Goldberg (2016a), the latter of whom also used bi-LSTM representations as inputs to their tree.

The final set of experiments were to investigate whether or not RLTs managed to retain the properties of the bi-LSTM representation in addition to its own, i.e., produce an h_τ that can represent a token's special position in a sentence *in addition to* representing it as the head of its own subtree.

An important property of the bi-LSTM positional representation is its ability to encode relevant information from other parts of a sentence into a particular word's representation. This means that fewer features are required to do parsing, and indeed Cross and Huang (2016) achieved successful results with Arc-Standard using only the top 2 items on the stack and the front item on the buffer, $\{s_{0,1}, b_0\}$. (Kiperwasser and Goldberg, 2016b) also showed that this was the case for Arc-Hybrid, but they also showed that more structural features (up to 11 features) improved performance. (Shi et al., 2017) showed that the minimal feature set of just the first items of the stack and buffer $\{s_0, b_0\}$ were needed for

	Dev		Test	
	UAS	LAS	UAS	LAS
Forward				
$\{s_{0-3}, b_{0-3}\}$	93.85	91.67	93.61	91.65
$\{s_{0,1}, b_0\}$	93.84	91.58	93.64	91.70
$\{s_{0,1}\}$	93.85	91.69	93.48	91.51
Bi-directional				
$\{s_{0-3}, b_{0-3}\}$	**94.20**	91.94	93.79	91.86
$\{s_{0,1}, b_0\}$	94.16	**92.09**	**93.91**	**92.03**
$\{s_{0,1}\}$	93.93	92.01	93.81	91.94

Table 6.3: Dev and test set scores for different feature sets, using a bi-LSTM positional vector as h_v, for Forward and Bi-directional encoding.

a successful parser for both Arc-Hybrid and Arc-Eager.

The results shown thus far are the results of a wide feature set, the first 4 items on both structures $\{s_{0-3}, b_{0-3}\}$, which was comparable to earlier features sets used by (Weiss et al., 2015) and (Chen and Manning, 2014). The results in Table 6.3 show the performance of our RLT models on increasingly small feature sets. Interestingly the drop in the accuracy of RLTs with the complete removal of buffer features is negligible, with the main notable drop being that of the dev set UAS. Our minimal feature set here consists of only the top 2 items on the stack $\{s_{0,1}\}$, while the minimal feature set explored by Shi et al. (2017) was the first item on the stack and buffer $\{s_0, b_0\}$, which worked for Arc-Hybrid/Eager but did not work for Arc-Standard.

The accuracy scores in Table 6.3 are not impacted significantly by the change in feature sets. The main effect appears to be on the UAS scores of the Bi-directional encoding models on the development set, which fall from 94.20 for $\{s_{0-3}, b_{0-3}\}$ to 93.93 for $\{s_{0,1}\}$. The set $\{s_{0,1}, b_0\}$ achieves slightly a better test accuracy of (93.91/92.03), compared to (93.79/91.86) for the largest set, and (93.81/91.94) for the smallest. Results for Forward encoding largely mimic the same pattern, with the minimal set slightly underperforming on the test set, and with $\{s_{0,1}, b_0\}$ producing the best accuracy of (93.64/91.70).

	Dev		Test	
	UAS	LAS	UAS	LAS
This work				
Forward Encoding + 8 feats. + word/pos embeddings	93.45	91.09	93.06	90.93
Forward Encoding + 4 feats. + positional vectors	93.84	91.58	93.64	91.70
Forward Encoding + 2 feats. + positional vectors	93.85	91.69	93.48	91.51
Bi-directional Encoding + 8 feats. + word/pos embeddings	93.28	91.09	93.04	91.01
Bi-directional Encoding + 4 feats. + positional vectors	94.16	92.09	93.91	92.03
Bi-directional Encoding + 2 feats. + positional vectors	93.93	92.01	93.81	91.94
Recursive Tree				
Le and Zuidema (2014)	N/A	N/A	93.84	91.51
Dyer et al. (2015)	93.2	90.9	93.1	90.9
Kiperwasser and Goldberg (2016a)	93.3	90.8	93.0	90.9
Ballesteros et al. (2016)	N/A	N/A	93.56	91.42
Feed Forward				
Chen and Manning (2014)	92.00	89.70	91.80	89.60
Weiss et al. (2015)	N/A	N/A	93.99	92.05
Andor et al. (2016)	**94.38**	**92.17**	**94.61**	**92.79**
Bi-LSTM positional representation				
Cross and Huang (2016)	93.67	91.48	93.42	91.36
Kiperwasser and Goldberg (2016b)	93.8	91.5	93.9	91.9
Shi et al. (2017)	93.92	N/A	94.53	N/A

Table 6.4: Dev and test set scores on WSJ (SD) for some of the highest scoring Transition-based Dependency Parsers in current literature. Positional vectors refer to the bi-LSTM vector representation used for h_v, and word/pos embeddings refers to the concatenation of these vectors to represent h_v. 8 feats. refers to the use of the top 4 items on the stack and buffer, 2 feats. refers to the use of the top 2 items on the stack.

Other experiments An intuitive addition to the representation of a subtree as a sequence is to include the dependency label. This surprisingly harmed results by up to 1% across all encoding mechanisms. Additionally, we experimented with adding a separate ending tag at the end of the sequence, which did not have any statistically significant impact on the final accuracy.

6.6 Discussion

Our main comparisons have been with the work of Kiperwasser and Goldberg (2016a). The former uses a bottom up recursive approach to build a tree representation as well,

but separates the sequence of children into a left and a right sequence, with the head itself being the start of both sequences, and the final representation of the subtree being a concatenation of the output of both sequences. As in our work, Kiperwasser and Goldberg (2016a) use bi-LSTM vectors to represent words being input to the encoding LSTM. We note that in the case of dependents that are leaf nodes in the dependency tree, the representation of Kiperwasser and Goldberg (2016a) models the left sequence backwards and the right sequence forwards, and leaves no way to model information considering the entire set of dependents.

We also compare our results with Dyer et al. (2015), who uses a bottom encoding to represent words on the stack, and then uses a stack-LSTM to represent the stack and buffer. The main point of interest here is a recursive composition function (not to be confused with that of Kuncoro et al. (2016a), which was the basis for our Compositional encoding in Section 6.3.3). This method encodes a (head, relation, dependent) tuple, and represents heads with multiple dependents by reapplying the composition function with the previous output as the head. The dependents are encoded into this representation as they are added to the tree, which again means an unordered representation of dependents. Our models suffered a drop in accuracy when we used an unordered sequence of dependents, which could be an explanation for the 1% difference in accuracy scores.

Finally in the Recursive tree encoding category, our results are comparable with those of Le and Zuidema (2014), who use a tree representation for parser re-ranking by scoring a k-best list of parses.

The performance of RLTs show a considerable ability to encode structural information into a single dense vector. This ability is highlighted when comparing with Weiss et al. (2015), where the resulting accuracy scores are comparable but only with the use of a structured perceptron. Similarly, the scores of Kiperwasser and Goldberg (2016b) improve by using structural features in addition to the initial set of $\{s_{0-2}, b_0\}$, with the left and right-most modifiers of the first 3 and the left-most modifier of the last, for a total of 11 positional features. In both of these cases the stack and buffer features are similar, with

RLTs showing an ability to implicitly encode useful structural features in the final tree vector τ.

Additionally RLTs gain much from the use of positional vectors as the base representation v. The structure of RLTs predictably is not capable of modelling the sequential position of a word in its sentence, but it can retain the information modelled by the bi-LSTM representation fed into it. This can be seen in the very similar accuracies of the different feature sets for both Forward and Bi-directional encoding. We note that Bi-directional encoding offers no benefit over Forward encoding when embedding vectors are used for v, but are seemingly capable of extracting more information from positional vectors.

We observe that the accuracy of the RLTs remains largely stable despite the removal of all buffer features. We speculate that this is a consequence of 2 properties of our model. The first is our use of the bi-LSTM feature representation, which allows the top of the stack to also encode information from the following words in the sentence. This alone would possibly not be enough, since the front of the buffer might not be the word right after the top of the stack in the sentence. This could indicate the success of our prioritisation of LA decisions during train time, allowing the Arc-Standard system to approximate the strict behaviour of Arc-Hybrid.

Finally, our model produces competitive results with a minimal feature set that, to the best of our knowledge, has not yet been achieved for Arc-Standard, but has been achieved for Arc-Eager and Arc-Hybrid by Shi et al. (2017). A key difference is that our minimal features set consisted of the top 2 items on the stack, while Shi et al. (2017) used the first items from the stack and buffer, which did not work for Arc-Standard. This difference could be due to the different definitions of the LA transition in particular which use the front of the buffer as head, while Arc-Standard limits all transition effects to the stack.

Previously Shi et al. (2017) used the minimal features achieved for other parsing systems in the dynamic programming decoders of Huang and Sagae (2010a) and Kuhlmann et al. (2011a). This approach was too expensive to perform for the smallest Arc-Standard

feature set published by Cross and Huang (2016), but would now be possible.

Our results remain behind those of Andor et al. (2016) and Shi et al. (2017), both of whom used global loss function, in addition to the latter's exact decoding. This change easily integrates with our approach and is left for future work.

6.7 Conclusion & Future Work

In this chapter we proposed a recursive tree architecture capable of modeling both subtrees and whole dependency trees. This method exploits the ability of deep learning to model combinations of features as needed in dense vectors, moving further away from feature selection to more expressive architectures.

We have shown the extent to which this approach is capable of encoding wide ranging relevant features, managing to produce competitive results with a minimal feature set for Arc-Standard.

Furthermore, we believe this architecture is potentially useful for other applications as well, including question answering, sentence similarity, and natural language generation. This in addition to being applicable to other techniques that improve dependency parsing, such as the reranking approach demonstrated by Le and Zuidema (2014). This is left for future work.

CHAPTER 7

CONCLUSION

In this thesis we have explored different ways in which Deep Learning is applied to the task of Transition-based dependency parsing. Our work has included both simple architectures that produce strong results, as well as more complex feature representation methods.

In chapter 4 we showed how using LSTMs instead of the feed forward layer in the basic Chen and Manning (2014) model can increase its accuracy beyond that of much larger models. In addition we proposed a method of initialisation for RNNs that was capable of combining the immediate relationship between a configuration and a transition, as captured by a pre-trained feed forward network, with the useful sequence modelling abilities of an RNN.

Our results showed an improvement for both LSTMs and Elman networks, and was successful both with the use of external embeddings and without. This two-stage training method is potentially also applicable to other learning tasks, where both a feed forward network and an RNN-based network can be trained.

Our investigation into the best structure for a dependency parsing model provided some interesting insights. Based on the results of our experiments in chapter 5, Hierarchical classification (Cross and Huang, 2016) appears to be the best structure for a neural network-based parser so far. Our success in training a dependency parser and a dependency labeller separately opens up possibilities for alternative training methods, especially where improving the accuracy of labelling is concerned.

We would like to emphasise that both the observation of the usefulness of Hierarchical classification and our separately trained parser and labeller contrast with a notable direction in literature towards more joint learning for both neural network-based models and non-neural network-based models. Examples of this include joint dependency parsing and part-of-speech tagging (Alberti et al., 2015; Bohnet and Nivre, 2012), and joint syntactic-semantic parsing (Swayamdipta et al., 2016; Henderson et al., 2013; Björkelund et al., 2010). Yet Hierarchical classification features in some of the highest accuracy neural network-based dependency parsers in current literature (Kiperwasser and Goldberg, 2016b; Shi et al., 2017).

Moreover, both our hierarchical and separately trained models achieve competitive results, that are only surpassed by globally trained models. Exluding the latter method, the parsers built in chapter 5 are the most accurate transition-based dependency parsers to date.

We explored feature representation for a dependency parser in chapter 6. We proposed a novel method for representing both subtrees and whole dependency trees with a Recursive LSTM Tree. This method of representation also proved capable of incorporating the representation capabilities of a sequential bi-LSTM layer, meaning that at any node in the tree, the resulting dense vector represents both the structural features of the word, but also the positional features of that word in its sentence.

Our proposed Recursive LSTM Trees outperformed other bottom up tree representation techniques by a substantial margin, and also managed to surpass other methods of tree representation, making it, to our knowledge, the highest performing tree representation mechanism in current literature.

Finally, we found that the representational abilities of Recursive LSTM Trees enabled our Arc-Standard parser to train successfully with a minimal feature set of only the top two items on the stack. This feature set is smaller than that asserted by Cross and Huang (2016) and Shi et al. (2017) in their work for Arc-Standard, and is the same size as the minimal feature sets required for other parsing strategies as found by Shi et al. (2017).

7.1 Thesis Question Revisited

Here we consider how the work in this thesis addresses the Thesis Questions set out in Section 1.2.

1. Does the configuration of a parser at one point in the parsing process hold information that is useful to making transitions later in the parsing sequence?

   The experiments in Chapter 4 indicate that the answer is yes, but only to a limited extent. Our experiments show that if all other factors are kept unchanged, then LSTMs are the only type of RNNs that are capable of extracting information from the parser configuration in a way that benefits both the decision being made for that configuration and for future decisions as well.

   Our 2-stage initialisation method shows that learning to optimise for both current and future decisions simultaneously can lead to worse results, but learning them separately improves both an already successfull LSTM-based network, and propels a previously unsuccessful Elman network to perform even better than the baseline architecture.

2. What is the best structure for the classification task? How does this influence the architecture of the neural networks used?

   We confirmed that Hierarchical classification, independently proposed by Cross and Huang (2016) and Kiperwasser and Goldberg (2016b), produced the best results despite the trend towards more joint task learning in other areas of literature. We applied this structure to different neural network architectures in Chapter 5, and attempted to devise finer grained versions of this approach, which ultimately did not produced better results than Hierarchical classification.

   Surprisingly, our experiments also showed that dependency parsing can be learnt

independent of the dependency labelling task, and still produce accuracies matching the original joint model of Chen and Manning (2014), and enhanced versions of it. While this approach does not produce better results than Hierarchical classification, it has interesting implications for dependency labelling, and is a potential starting point towards closing the ever present gap between Unlabelled Attachment accuracy, and Labelled Attachment accuracy. Exploring this aspect further is left for future work.

3. Given deep learning's ability to learn important features and combinations from context, how can this ability be used to increase the expressiveness of features? And if this is possible, does this expressivness hold with fewer features?

   The Recursive LSTM Trees proposed in Chapter 6 show an ability to model hierarchies in a contextually aware manner. This can be seen in their ability to provide rich features to parsers relying solely on representations from the stack and buffer, eliminating the need for structural features, such as left-most and right-most child, since they are already encoded by our approach into the same dense vector that represents the root of the tree.

   The principle behind our solution stated simply is to reorganise a structure as a hierarchy of sequences. The sequences are modelled by a *Recurrent* element, and the hierarchy is modelled by the *Recursive* application of this element. In the case of dependency trees this translates as sequences of head-dependent pairs being arranged according to their position in the dependency tree, with LSTMs being used to model these sequences.

   Finally, the performance of this solution also relies on the expressiveness of the basic word vector used to represent tokens in the sentence/dependency tree. We showed that incorporating the output of positional representation increased the expressiveness of the resulting dense vectors to the point where a competitive parser

could be trained with only 2 input features, further demonstrating deep learning's ability to learn from context given the right architecture.

7.2 Summary

In summary, this thesis has covered three key areas of the classification task for transition-based dependency parsing, the hidden state modelled by a neural network for this task in chapter 4, the structure of the output layer and by extension the definition of the task itself in chapter 5, and the input layer where feature representation is the key challenge in chapter 6.

Our contributions are also easy to integrate with each other, in addition to being able to benefit from the key strategies on which other state-of-the-art transition-based parsers depend, namely global training and exact decoding.

Finally, both the two-stage alternative initialisation method (Chapter 4) and Recursive LSTM Trees (Chapter 6) are applicable to areas outside of parsing, and potentially outside of NLP altogether. Our Recursive LSTM Trees in particular have the potential to contribute to areas such as question answering, sentence similarity, and natural language generation.

LIST OF REFERENCES

Chris Alberti, David Weiss, Greg Coppola, and Slav Petrov. Improved transition-based parsing and tagging with neural networks. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1354–1359, 2015.

Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. Globally normalized transition-based neural networks. *arXiv preprint arXiv:1603.06042*, 2016.

Gabor Angeli, Melvin Johnson Premkumar, and Christopher D Manning. Leveraging linguistic structure for open domain information extraction. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL 2015)*, 2015.

Miguel Ballesteros, Yoav Goldberg, Chris Dyer, and Noah A. Smith. Training with exploration improves a greedy stack lstm parser. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2005–2010. Association for Computational Linguistics, 2016. doi: 10.18653/v1/D16-1211. URL `http://www.aclweb.org/anthology/D16-1211`.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

Yoshua Bengio, Pascal Lamblin, Dan Popovici, Hugo Larochelle, et al. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19:153, 2007.

Anders Björkelund, Bernd Bohnet, Love Hafdell, and Pierre Nugues. A high-performance syntactic and semantic dependency parser. In *Proceedings of the 23rd International Conference on Computational Linguistics: Demonstrations*, COLING '10, pages 33–36, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics. URL `http://dl.acm.org/citation.cfm?id=1944284.1944293`.

Mikael Boden. A guide to recurrent neural networks and backpropagation. *The Dallas project, SICS technical report*, 2002.

Bernd Bohnet. Very high accuracy and fast dependency parsing is not a contradiction. In *Proceedings of the 23rd International Conference on Computational Linguistics*, pages 89–97. Association for Computational Linguistics, 2010.

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*, EMNLP-CoNLL '12, pages 1455–1465, Stroudsburg, PA, USA, 2012. Association for Computational Linguistics. URL `http://dl.acm.org/citation.cfm?id=2390948.2391114`.

Léon Bottou. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436. Springer, 2012.

Sabine Buchholz and Erwin Marsi. Conll-x shared task on multilingual dependency parsing. In *Proceedings of the Tenth Conference on Computational Natural Language Learning*, pages 149–164. Association for Computational Linguistics, 2006.

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

Xinchi Chen, Yaqian Zhou, Chenxi Zhu, Xipeng Qiu, and Xuanjing Huang. Transition-based dependency parsing using two heterogeneous gated recursive neural networks. In *EMNLP*, pages 1879–1889, 2015.

Hao Cheng, Hao Fang, Xiaodong He, Jianfeng Gao, and Li Deng. Bi-directional attention with agreement for dependency parsing. *arXiv preprint arXiv:1608.02076*, 2016.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

Michael Collins and Brian Roark. Incremental parsing with the perceptron algorithm. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, page 111. Association for Computational Linguistics, 2004.

Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 160–167, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-205-4.

Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *J. Mach. Learn. Res.*, 12:2493–2537, November 2011. ISSN 1532-4435.

Michael A Covington. Syntactic theory in the high middle ages. modistic models of sentence structure. *Cambridge Studies in Linguistics London*, (39):1–163, 1984.

Michael A Covington. A fundamental algorithm for dependency parsing. Citeseer, 2001.

James Cross and Liang Huang. Incremental parsing with minimal features using bi-directional LSTM. *CoRR*, abs/1606.06406, 2016.

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, 2006.

Timothy Dozat and Christopher D. Manning. Deep biaffine attention for neural dependency parsing. *CoRR*, abs/1611.01734, 2016.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A Smith. Transition-based dependency parsing with stack long short-term memory. *arXiv preprint arXiv:1505.08075*, 2015.

Jason M. Eisner. Three new probabilistic models for dependency parsing: An exploration. In *Proceedings of the 16th Conference on Computational Linguistics - Volume 1*, COLING '96, pages 340–345, Stroudsburg, PA, USA, 1996. Association for Computational Linguistics.

Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

Yarin Gal. A theoretically grounded application of dropout in recurrent neural networks. *arXiv:1512.05287*, 2015.

Nikhil Garg and James Henderson. Temporal restricted boltzmann machines for dependency parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers-Volume 2*, pages 11–17. Association for Computational Linguistics, 2011.

Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

Felix A Gers, Nicol N Schraudolph, and Jürgen Schmidhuber. Learning precise timing with lstm recurrent networks. *Journal of machine learning research*, 3(Aug):115–143, 2002.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR. URL `http://proceedings.mlr.press/v9/glorot10a.html`.

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.

Yoav Goldberg and Joakim Nivre. Training deterministic parsers with non-deterministic oracles. *Transactions of the association for Computational Linguistics*, 1:403–414, 2013.

Christoph Goller and Andreas Kuchler. Learning task-dependent distributed representations by backpropagation through structure. In *Neural Networks, 1996., IEEE International Conference on*, volume 1, pages 347–352. IEEE, 1996.

Carlos Gómez-Rodrıguez, John Carroll, and David Weir. A deductive approach to dependency parsing. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL08: HLT)*, pages 968–976, 2008.

Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5):602 – 610, 2005. IJCNN 2005.

Joseph Gubbins and Andreas Vlachos. Dependency language models for sentence completion. In *EMNLP*, volume 13, pages 1405–1410, 2013.

Jan Hajič, Massimiliano Ciaramita, Richard Johansson, Daisuke Kawahara, Maria Antònia Martí, Lluís Màrquez, Adam Meyers, Joakim Nivre, Sebastian Padó, Jan Štepánek, Pavel Straňák, Mihai Surdeanu, Nianwen Xue, and Yi Zhang. The conll-2009 shared task: Syntactic and semantic dependencies in multiple languages, 2009. URL http://www.aclweb.org/anthology/W09-1201.

Kazuma Hashimoto, Caiming Xiong, Yoshimasa Tsuruoka, and Richard Socher. A joint many-task model: Growing a neural network for multiple nlp tasks. *arXiv preprint arXiv:1611.01587*, 2016.

James Henderson, Paola Merlo, Ivan Titov, and Gabriele Musillo. Multilingual joint parsing of syntactic and semantic dependencies with a latent variable model. *Computational Linguistics*, 39(4):949–998, 2013. doi: 10.1162/COLI_a_00158. URL https://doi.org/10.1162/COLI_a_00158.

Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

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.

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

Liang Huang and Kenji Sagae. Dynamic programming for linear-time incremental parsing. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL '10, pages 1077–1086, Stroudsburg, PA, USA, 2010a. Association for Computational Linguistics. URL http://dl.acm.org/citation.cfm?id=1858681.1858791.

Liang Huang and Kenji Sagae. Dynamic programming for linear-time incremental pars-

ing. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1077–1086. Association for Computational Linguistics, 2010b.

Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. *Journal of Machine Learning Research*, 2015.

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

Eliyahu Kiperwasser and Yoav Goldberg. Easy-first dependency parsing with hierarchical tree lstms. *arXiv preprint arXiv:1603.00375*, 2016a.

Eliyahu Kiperwasser and Yoav Goldberg. Simple and accurate dependency parsing using bidirectional lstm feature representations. *arXiv preprint arXiv:1603.04351*, 2016b.

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*, HLT '11, pages 673–682, Stroudsburg, PA, USA, 2011a. Association for Computational Linguistics. ISBN 978-1-932432-87-9. URL `http://dl.acm.org/citation.cfm?id=2002472.2002558`.

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, 2011b.

Adhiguna Kuncoro, Miguel Ballesteros, Lingpeng Kong, Chris Dyer, Graham Neubig, and Noah A. Smith. What do recurrent neural network grammars learn about syntax? *CoRR*, abs/1611.05774, 2016a.

Adhiguna Kuncoro, Yuichiro Sawai, Kevin Duh, and Yuji Matsumoto. Dependency parsing with lstms: An empirical evaluation. *arXiv preprint arXiv:1604.06529*, 2016b.

Phong Le and Willem Zuidema. The inside-outside recursive neural network model for dependency parsing. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 729–739, Doha, Qatar, October 2014. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/D14-1081`.

Quoc V Le, Navdeep Jaitly, and Geoffrey E Hinton. A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*, 2015.

Yann LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–50. Springer, 1998.

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.

Mausam, Michael Schmitz, Robert Bart, Stephen Soderland, and Oren Etzioni. Open language learning for information extraction. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, EMNLP-CoNLL '12, pages 523–534, Stroudsburg, PA, USA, 2012. Association for Computational Linguistics. URL `http://dl.acm.org/citation.cfm?id=2390948.2391009`.

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, 2005a.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of the Conference on*

Human Language Technology and Empirical Methods in Natural Language Processing, HLT '05, pages 523–530, Stroudsburg, PA, USA, 2005b. Association for Computational Linguistics.

Ryan McDonald, Kevin Lerman, and Fernando Pereira. Multilingual dependency analysis with a two-stage discriminative parser. In *Proceedings of the Conference on Computational Natural Language Learning (CONLL*, pages 216–220, 2006.

Ryan Mcdonald, Joakim Nivre, Yvonne Quirmbach-brundage, Yoav Goldberg, Dipanjan Das, Kuzman Ganchev, Keith Hall, Slav Petrov, Hao Zhang, Oscar Tckstrm, Claudia Bedini, Nria Bertomeu, and Castell Jungmee Lee. Universal dependency annotation for multilingual parsing. In *In Proc. of ACL 13*, 2013.

Ryan T McDonald and Joakim Nivre. Characterizing the errors of data-driven dependency parsing models. In *EMNLP-CoNLL*, pages 122–131, 2007.

Igor Aleksandrovič Melčuk. *Dependency syntax: theory and practice*. SUNY Press, 1988.

Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Interspeech*, volume 2, page 3, 2010.

Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *HLT-NAACL*, pages 746–751, 2013.

Tomas Mikolov, Armand Joulin, Sumit Chopra, Michael Mathieu, and Marc'Aurelio Ranzato. Learning longer memory in recurrent neural networks. *arXiv preprint arXiv:1412.7753*, 2014.

Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

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

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

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

Joakim Nivre. Incrementality in deterministic dependency parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*, IncrementParsing '04, pages 50–57, Stroudsburg, PA, USA, 2004. Association for Computational Linguistics. URL `http://dl.acm.org/citation.cfm?id=1613148.1613156`.

Joakim Nivre. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34(4):513–553, 2008.

Joakim Nivre, Johan Hall, Sandra Kübler, Ryan McDonald, Jens Nilsson, Sebastian Riedel, and Deniz Yuret. The conll 2007 shared task on dependency parsing. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, 2007.

Ankur P. Parikh, Hoifung Poon, and Kristina Toutanova. Grounded semantic parsing for complex knowledge extraction. In *HLT-NAACL*, pages 756–766. The Association for Computational Linguistics, 2015.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. *ICML (3)*, 28:1310–1318, 2013.

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. URL `http://www.aclweb.org/anthology/D14-1162`.

Tianze Shi, Liang Huang, and Lillian Lee. Fast(er) exact decoding and global training for transition-based dependency parsing via a minimal feature set. *CoRR*, abs/1708.09403, 2017. URL `http://arxiv.org/abs/1708.09403`.

Richard Socher, Christopher D Manning, and Andrew Y Ng. Learning continuous phrase representations and syntactic parsing with recursive neural networks. In *Proceedings of the NIPS-2010 Deep Learning and Unsupervised Feature Learning Workshop*, pages 1–9, 2010.

Pontus Stenetorp. Transition-based dependency parsing using recursive neural networks. In *NIPS Workshop on Deep Learning*. Citeseer, 2013.

Mihai Surdeanu, Richard Johansson, Adam Meyers, Lluís Màrquez, and Joakim Nivre. The conll-2008 shared task on joint parsing of syntactic and semantic dependencies. In *Proceedings of the Twelfth Conference on Computational Natural Language Learning*, pages 159–177. Association for Computational Linguistics, 2008.

Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

Swabha Swayamdipta, Miguel Ballesteros, Chris Dyer, and Noah A Smith. Greedy, joint syntactic-semantic parsing with stack lstms. *CoNLL 2016*, page 187, 2016.

Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. 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), pages 1556–1566, Beijing, China, July 2015. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P15-1150`.

Lucien Tesnière. Eléments de syntaxe structurale. *Paris, Klincksieck*, 1959.

Ivan Titov and James Henderson. Fast and robust multilingual dependency parsing with a generative latent variable model. In *Proceedings of CONLL-2007 shared task. EMNLP-CONLL*, 2007.

David Weiss, Chris Alberti, Michael Collins, and Slav Petrov. Structured training for neural network transition-based parsing. *arXiv preprint arXiv:1506.06158*, 2015.

Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*, pages 2048–2057, 2015.

Hiroyasu Yamada and Yuji Matsumoto. Statistical dependency analysis with support vector machines. In *Proceedings of IWPT*, volume 3, pages 195–206, 2003.

Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

Xingxing Zhang, Liang Lu, and Mirella Lapata. Top-down tree long short-term memory networks. *arXiv preprint arXiv:1511.00060*, 2015.

Yue Zhang and Stephen Clark. A tale of two parsers: investigating and combining graph-based and transition-based dependency parsing using beam-search. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 562–571. Association for Computational Linguistics, 2008.

Yue Zhang and Joakim Nivre. Analyzing the effect of global learning and beam-search on transition-based dependency parsing. In *COLING (Posters)*, pages 1391–1400, 2012.

Hao Zhou, Yue Zhang, Shujian Huang, and Jiajun Chen. A neural probabilistic structured-prediction model for transition-based dependency parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics*, pages 1213–1222, 2015.