Easy-First Dependency Parsing with Hierarchical Tree LSTMs

Eliyahu Kiperwasser
Computer Science Department
Bar-Ilan University
Ramat-Gan, Israel
elikip@gmail.com

Yoav Goldberg
Computer Science Department
Bar-Ilan University
Ramat-Gan, Israel
yoav.goldberg@gmail.com

Abstract

We suggest a compositional vector representation of parse trees that relies on a recursive combination of recurrent-neural network encoders. To demonstrate its effectiveness, we use the representation as the backbone of a greedy, bottom-up dependency parser, achieving very strong accuracies for English and Chinese, without relying on external word embeddings. The parser's implementation is available for download at the first author's webpage.

1 Introduction

Dependency-based syntactic representations of sentences are central to many language processing tasks (Kübler et al., 2009). Dependency parse-trees encode not only the syntactic structure of a sentence but also many aspects of its semantics.

A recent trend in NLP is concerned with encoding sentences as vectors ("sentence embeddings"), which can then be used for further prediction tasks. Recurrent neural networks (RNNs) (Elman, 1990), and in particular methods based on the LSTM architecture (Hochreiter and Schmidhuber, 1997), work very well for modeling sequences, and constantly obtain state-of-the-art results on both language-modeling and prediction tasks (see, e.g. (Mikolov et al., 2010)).

Several works attempt to extend recurrent neural networks to work on trees (see Section 8 for a brief overview), giving rise to the so-called recursive neural networks (Goller and Kuchler, 1996; Socher et al., 2010). However, recursive neural networks do not cope well with trees with arbitrary branching factors – most work require the encoded trees to be binary-branching, or have a fixed maximum arity. Other attempts allow arbitrary branching factors, at the expense of ignoring the order of the modifiers.

In contrast, we propose a tree-encoding that naturally supports trees with arbitrary branching factors, making it particularly appealing for dependency trees. Our tree encoder uses recurrent neural networks as a building block: we model the left and right sequences of modifiers using RNNs, which are composed in a recursive manner to form a tree (Section 3). We use our tree representation for encoding the partially-built parse trees in a greedy, bottom-up dependency parser which is based on the easy-first transition-system of Goldberg and Elhadad (2010).

Using the Hierarchical Tree LSTM representation, and without using any external embeddings, our parser achieves parsing accuracies of 92.6 UAS and 90.2 LAS on the PTB (Stanford dependencies) and 86.1 UAS and 84.4 LAS on the Chinese treebank, while relying on greedy decoding.

To the best of our knowledge, this is the first work to demonstrate competitive parsing accuracies for full-scale parsing while relying solely on recursive, compositional tree representations, and without using a reranking framework. We discuss related work in Section 8.

While the parsing experiments demonstrate the suitability of our representation for capturing the structural elements in the parse tree that are useful for predicting parsing decisions, we are interested in exploring the use of the RNN-based compositional vector representation of parse trees also for seman-

Transactions of the Association for Computational Linguistics, vol. 4, pp. 445–461, 2016. Action Editor: Noah Smith.
Submission batch: 11/2015; Revision batch: 2/2016; Published 8/2016.

tic tasks such as sentiment analysis (Socher et al., 2013b; Tai et al., 2015), sentence similarity judgements (Marelli et al., 2014) and textual entailment (Bowman et al., 2015).

2 Background and Notation

2.1 Dependency-based Representation

A dependency-based syntactic representation is centered around syntactic modification relations between head words and modifier words. The result are trees in which each node is a word in the sentence, and every node except for one designated root node has a parent node. A dependency tree over a sentence with n words $w_1, \ldots, w_n$ can be represented as a list of n pairs of the form (h, m), where $0 \leq h \leq n$ and $1 \leq m \leq n$. Each such pair represents an edge in the tree in which h is the index of a head word (including the special ROOT node 0), and m is the index of a modifier word. In order for the dependency trees to be useful for actual downstream language processing tasks, each edge is labeled with a syntactic relation. The tree representation then becomes a list of triplets (h, m, ℓ), where $1 \leq \ell \leq L$ is the index of a dependency relation out of a designated set of L syntactic relations.

Dependency trees tend to be relatively shallow, with some nodes having many children. Looking at trees in the PTB training set we find that 94% of the trees have a height of at most 10, and 49% of the trees a height of at most 6. In terms of width, 93% of the trees have at least one node with an arity of 4 or more, and 56% of the trees have at least one node with an arity of 6 or more.

2.2 Recurrent Networks and LSTMs

Recurrent neural networks (RNNs), first proposed by Elman (1990) are statistical learners for modeling sequential data. In this work, we use the RNN abstraction as a building block, and recursively combine several RNNs to obtain our tree representation. We briefly describe the RNN abstraction below. For further detail on RNNs, the reader is referred to sources such as (Goldberg, 2015; Bengio and Courville, 2016; Cho, 2015).

The RNN abstraction is a function RNN that takes in a sequence of inputs vectors $x_1, \ldots, x_n$ ($x_i \in \mathbb{R}^{d_{in}}$), and produces a sequence of state vectors (also called output vectors) $y_1, \ldots, y_n$ ($y_i \in \mathbb{R}^{d_{out}}$). Each y_i is conditioned on all the inputs $x_1, \ldots, x_i$ preceding it. Ignoring the intermediate outputs $y_1, \ldots, y_{n-1}$, the RNN can be thought of as encoding the sequence $x_1, \ldots, x_n$ into a final state y_n. Our notation in this paper follows this view.

The RNN is defined recursively using two functions:[1]

$$RNN(s_0, x_1, \ldots, x_n) = y_n = O(s_n)$$
$$s_i = N(s_{i-1}, x_i)$$

Here, a function N takes as input a vector x_i and a state vector s_{i-1} and returns as output a new state s_i. One can then extract an output vector y_i from s_i using the function O (the function O is usually the identity function, or a function that returns a subset of the elements in s_i).

Taking an algorithmic perspective, one can view the RNN as a `state` object with three operations: `s = RNN.initial()` returns a new initial state, `s.advance(x)` takes an input vector and returns a new state, and `s.output()` returns the output vector for the current state. When clear from the context, we abbreviate and use the state's name (`s`) instead of `s.output()` to refer to the output vector at the state.

The functions N and O defining the RNN are parameterized by parameters θ (matrices and vectors), which are trained from data. Specifically, one is usually interested in using some of the outputs y_i for making predictions. The RNN is trained such that the encoding y_i is good for the prediction task. That is, the RNN learns which aspects of the sequence $x_1, \ldots, x_i$ are informative for the prediction.

We use subscripts (i.e. RNN_L, RNN_R) to indicate different RNNs, that is, RNNs that have different sets of parameters.

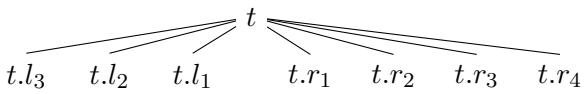
Specific instantiations of N and O yield different recurrent network mechanisms. In this work we use the Long Short Term Memory (LSTM) variant (Hochreiter and Schmidhuber, 1997) which is shown to be a very capable sequence learner. However, our algorithm and encoding method do not rely on any specific property of the LSTM architecture, and the

[1] We follow the notation of Goldberg (2015), with the exception of taking the output of the RNN to be a single vector rather than a sequence, and renaming R to N.

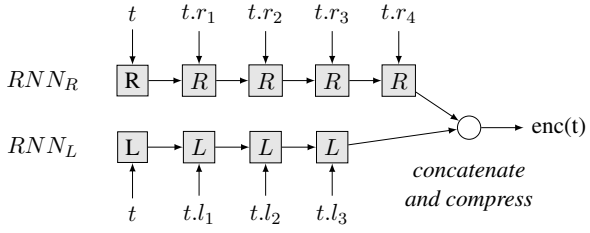
LSTM can be transparently switched for any other RNN variant.

3 Tree Representation

We now describe our method for representing a tree as a d-dimensional vector. We assume trees in which the children are ordered and there are $k_l \geq 0$ children before the parent node (left children) and $k_r \geq 0$ children after it (right children). Such trees correspond well to dependency tree structures. We refer to the parent node as a head, and to its children as modifiers. For a node t, we refer to its left modifiers as $t.l_1, t.l_2, \ldots, t.l_{k_l}$ and its right modifiers as $t.r_1, t.r_2, \ldots, t.r_{k_r}$ The indices of the modifier are always from the parent outward, that is $t.l_1$ is the left modifier closest to the head t:



The gist of the idea is to treat the modifiers of a node as a sequence, and encode this sequence using an RNN. We separate left-modifiers from right-modifiers, and use two RNNs: the first RNN encodes the sequence of left-modifiers from the head outwards, and the second RNN the sequence of right-modifiers from the head outwards. The first input to each RNN is the vector representation of the head word, and the last input is the vector representation of the left-most or the right-most modifier. The node's representation is then a concatenation of the RNN encoding of the left-modifiers with the RNN encoding of the right-modifiers. The encoding is recursive: the representation for each of the modifier nodes is computed in a similar fashion.



More formally, consider a node t. Let $i(t)$ be the sentence index of the word corresponding to the head node t, and let v_i be a vector corresponding to the ith word in the sentence (this vector captures information such as the word form and its part of speech tag, and will be discussed shortly). The vector encoding of a node $enc(t) \in \mathbb{R}^{d_{enc}}$ is then defined as follows:

$$\begin{aligned} enc(t) =& g(W^e \cdot (e_l(t) \circ e_r(t)) + b^e) \\ e_l(t) =& RNN_L(v_{i(t)}, enc(t.l_1), \ldots, enc(t.l_{k_l})) \\ e_r(t) =& RNN_R(v_{i(t)}, enc(t.r_1), \ldots, enc(t.r_{k_r})) \end{aligned}$$

First, the sequences consisting of the head-vector $v_{i(t)}$ followed by left-modifiers and the head-vector followed by right-modifiers are encoded using two RNNs, RNN_L and RNN_R, resulting in RNN states $e_l(t) \in \mathbb{R}^{d_{out}}$ and $e_r(t) \in \mathbb{R}^{d_{out}}$. Then, the RNN states are concatenated, resulting in a $2d_{out}$-dimensional vector $(e_l(t) \circ e_r(t))$, which is reduced back to d-dimensions using a linear transformation followed by a non-linear activation function g. The recursion stops at leaf nodes, for which:

$$\begin{aligned} enc(\text{leaf}) =& g(W^e \cdot (e_l(\text{leaf}) \circ e_r(\text{leaf})) + b^e) \\ e_l(\text{leaf}) =& RNN_L(v_{i(\text{leaf})}) \\ e_r(\text{leaf}) =& RNN_R(v_{i(\text{leaf})}) \end{aligned}$$

Figure 1 shows the network used for encoding the sentence "the black fox who really likes apples did not jump over a lazy dog yesterday".

3.1 Representing words

In the discussion above we assume a vector representation v_i associated with the ith sentence word. What does v_i look like? A sensible approach would be to take v_i to be a function of the word-form and the part-of-speech (POS) tag of the ith word, that is:

$$v_i = g(W^v \cdot (w_i \circ p_i) + b^v)$$

where w_i and p_i are the embedded vectors of the word-form and POS-tag of the ith word.

This encodes each word in isolation, disregarding its context. The context of a word can be very informative regarding its meaning. One way of incorporating context is the Bidirectional RNN (Schuster and Paliwal, 1997). Bidirectional RNNs are shown to be an effective representation for sequence tagging (Irsoy and Cardie, 2014). Bidirectional RNNs represent a word in the sentence using a concatenation of the end-states of two RNNs, one running

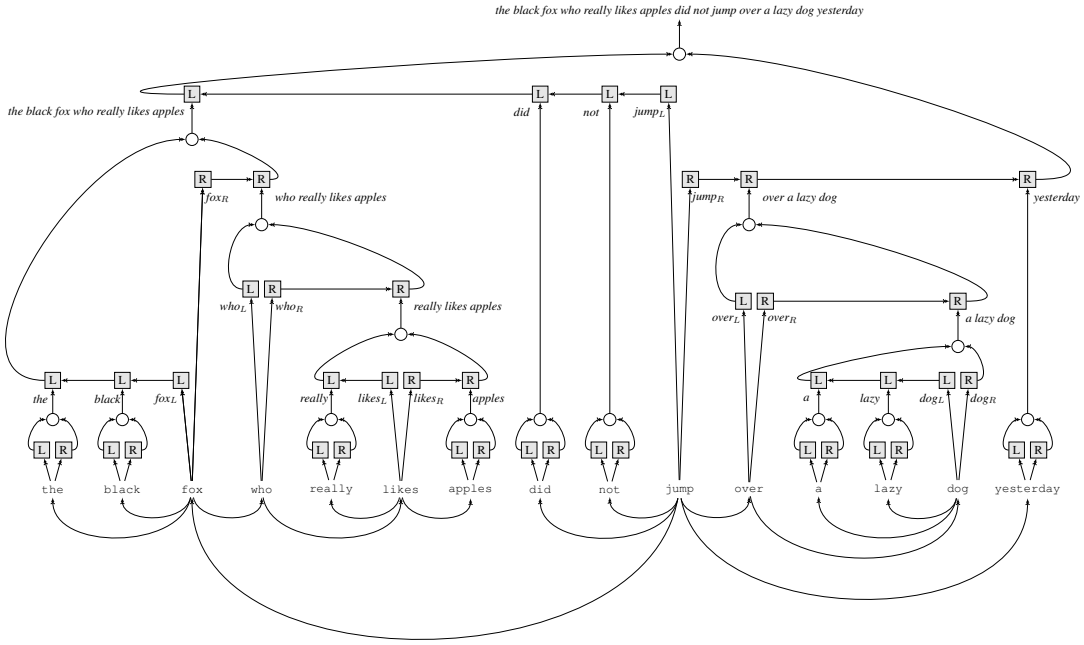


Figure 1: Network for encoding the sentence "the black fox who really likes apples did not jump over a lazy dog yesterday". **Top**: the network structure: boxed nodes represent LSTM cells, where L are cells belonging to the left-modifiers sequence model RNN_L, and R to the right-modifiers sequence model RNN_R. Circle nodes represent a concatenation followed by a linear transformation and a non-linearity. **Bottom**: the dependency parse of the sentence.

from the beginning of the sentence to the word and the other running from the end to the word. The result is a vector representation for each word which captures not only the word but also its context.

We adopt the Bidirectional LSTM scheme to enrich our node vector representation, and for an n-words sentence compute the vector representations v_i as follows:

$$
\begin{aligned}
v'_i &= g(W^v \cdot (w_i \circ p_i) + b^v) \\
f_i &= LSTM_F(v'_1, v'_2, \ldots, v'_i) \\
b_i &= LSTM_B(v'_n, v'_{n-1}, \ldots, v'_i) \\
v_i &= (f_i \circ b_i)
\end{aligned}
$$

We plug this word representation as word vectors, allowing each word vector v_i to capture information regarding the word form and POS-tag, as well as the sentential context it appears in. The BI-LSTM encoder is trained jointly with the rest of the network towards the parsing objective, using back-propagation.

Embedding vectors The word and POS embeddings w_i and p_i are also trained together with the network. For the word embeddings, we experiment with random initialization, as well as with initialization using pre-trained word embeddings. Our main goal in this work is not to provide top parsing accuracies, but rather to evaluate the ability of the proposed compositional architecture to learn and capture the structural cues that are needed for accurate parsing. Thus, we are most interested in the random initialization setup: what can the network learn from the training corpus alone, *without* relying on external resources.

However, the ability to perform semi-supervised learning by initializing the word-embeddings with vectors that are pre-trained on large amount of unannotated data is an appealing property of the neural-network approaches, and we evaluate our parser also in this semi-supervised setup. When using pre-trained word embeddings, we follow (Dyer et al., 2015) and use embedding vectors which are trained using *positional context* (Ling et al., 2015), as these

were shown to work better than traditional skip-gram vectors for syntactic tasks such as part-of-speech tagging and parsing.

3.2 A note on the head-outward generation

Why did we choose to encode the children from the head outward, and not the other way around? The head outward generation order is needed to facilitate incremental tree construction and allow for efficient parsing, as we show in section 4 below. Besides the efficiency considerations, using the head-outward encoding puts more emphasis on the outermost dependants, which are known to be the most informative for predicting parse structure.[2] We rely on the RNN capability of extracting information from arbitrary positions in the sequence to incorporate information about the head word itself, which appears in the beginning of the sequence. This seems to work well, which is expected considering that the average maximal number of siblings in one direction in the PTB is 4.1, and LSTMs were demonstrated to capture much longer-range interactions. Still, when using the tree encoding in a situation where the tree is fully specified in advance, i.e. for sentence classification, sentence similarity or translation tasks, using a head-inward generation order (or even a bi-directional RNN) may prove to work better. We leave this line of inquiry to future work.

The head-outward modifier generation approach has a long history in the parsing literature, and goes back to at least Eisner (1996) and Collins (1997). In contrast to previous work in which each modifier could condition only on a fixed small number of modifiers preceding it, and in which the left- and right- sequences of modifiers were treated as independent from one another for computational efficiency reasons, our approach allows the model to access information from the entirety of both the left and the right sequences jointly.

[2]Features in transition-based dependency parsers often look at the current left-most and right-most dependents of a given node, and almost never look further than the second left-most or second right-most dependents. Second-order graph based dependency parsers (McDonald, 2006; Eisner, 2000) also condition on the current outermost dependent when generating its sibling.

4 Parsing Algorithm

We now turn to explain how to parse using the tree encoder defined above. We begin by describing our bottom-up parsing algorithm, and then show how the encoded vector representation can be built and maintained throughout the parsing process.

4.1 Bottom-up Parsing

We follow a (projective) bottom-up parsing strategy, similar to the easy-first parsing algorithm of Goldberg and Elhadad (2010).

The main data-structure in the parser is a list of partially-built parse trees we call *pending*. For a sentence with words $w_1, \ldots, w_n$, the *pending* list is initialized with n nodes, where $pending[i]$ corresponds to word w_i. The algorithm then chooses two neighbouring trees in the *pending* list $pending[i]$ and $pending[i+1]$ and either attaches the root of $pending[i+1]$ as the right-most modifier of the root of $pending[i]$, or attaches the root of $pending[i]$ as the left-most modifier of the root of $pending[i+1]$. The tree which was treated as modifier is then removed from the $pending$ list, shortening it by one. The process ends after $n-1$ steps, when a single tree remains in the pending list, which is taken to be the output parse tree. The parsing process is described in Algorithm 1.

Algorithm 1 Parsing

Input: Sentence $w = w_1, \ldots, w_n$
for $i \in 1, \ldots, n$ **do**
  pend$[i].id \leftarrow i$
arcs $\leftarrow$ []
while |pend| > 1 **do**
  $A \leftarrow \{(i,d) \mid 1 \leq i < |\text{pend}|, d \in \{l, r\}\}$
  $i, d \leftarrow select(A)$
  if $d = l$ **then**
    $m, h \leftarrow$ pend$[i]$, pend$[i+1]$
    pend.remove(i)
  else
    $h, m \leftarrow$ pend$[i]$, pend$[i+1]$
    pend.remove($i+1$)
  arcs.append($h.id, m.id$)
return arcs

This parsing algorithm is both sound and complete with respect to the class of projective depen-

dency trees (Goldberg and Elhadad, 2010). The algorithm depends on non-deterministic choices of an index in the pending list and an attachment direction (line 7). When parsing in practice, the non-deterministic choice will be replaced by using a trained classifier to assign a score to each index-direction pair, and selecting the highest scoring pair. We discuss the scoring function in Section 4.4, and the training algorithm in Section 5.

4.2 Bottom-up Tree-Encoding

We would like the scoring function to condition on the vector encodings of the subtrees it aims to connect. Algorithm 2 shows how to maintain the vector encodings together with the parsing algorithm, so that at every stage in the parsing process each item $pending[i]$ is associated with a vector encoding of the corresponding tree.

Algorithm 2 Parsing while maintaining tree representations

1: **Input:** Sentence $w = w_1, \ldots, w_n$
2: **Input:** Vectors v_i corresponding to words w_i
3: arcs $\leftarrow$ []
4: **for** $i \in 1, \ldots, n$ **do**
5: pend$[i].id \leftarrow i$
6: pend$[i].e_l \leftarrow RNN_L.init().append(v_i)$
7: pend$[i].e_r \leftarrow RNN_R.init().append(v_i)$
8: **while** |pend| > 1 **do**
9: $A \leftarrow \{(i, d) \mid 1 \leq i < |\text{pend}|, d \in \{l, r\}\}$
10: $i, d \leftarrow select(A)$
11: **if** $d = l$ **then**
12: $m, h \leftarrow$ pend$[i]$, pend$[i+1]$
13: $m.c = m.e_l \circ m.e_r$
14: $m.enc = g(W(m.c) + b)$
15: $h.e_l.append(m.enc)$
16: pend.remove(i)
17: **else**
18: $h, m \leftarrow$ pend$[i]$, pend$[i+1]$
19: $m.c = m.e_l \circ m.e_r$
20: $m.enc = g(W(m.c) + b)$
21: $h.e_r.append(m.enc)$
22: pend.remove($i+1$)
23: arcs.add($h.id, m.id$)
24: **return** arcs

4.3 Labeled Tree Representation

The tree representation described above does not account for the relation labels ℓ the parsing algorithm assigns each edge. In cases the tree is fully specified in advance, the relation of each word to its head can be added to the word representations v_i. However, in the context of parsing, the labels become known only when the modifier is attached to its parent. We thus extend the tree representation by concatenating the node vector representation with a vector representation assigned to the label connecting the subtree to its parent. Formally, only the final $enc(t)$ equation changes:

$$enc(t) = g(W^e \cdot (e_l \circ e_r \circ \ell) + b^e)$$

where ℓ is a learned embedding vector associated with the given label.

4.4 Scoring Function

The parsing algorithm relies on a function $select(A)$ for choosing the action to take at each stage. We model this function as:

$$select(A) = argmax_{(i,d,\ell) \in A} Score(pend, i, d, \ell)$$

where $Score(.)$ is a learned function whose job is to assign scores to possible actions to reflect their quality. Ideally, it will not only score correct actions above incorrect ones, but also more confident (easier) actions above less confident ones, in order to minimize error propagation in the greedy parsing process.

When scoring a possible attachment between a head h and a modifier m with relation ℓ, the scoring function should attempt to reflect the following pieces of information:

- Are the head words of h and m compatible under relation l?
- Is the modifier m compatible with the already existing modifiers of h? In other words, is m a good subtree to connect as an outer-most modifier in the subtree h?
- Is m complete, in the sense that it already acquired all of its own modifiers?

to this end, the scoring function looks at a window of k subtrees to each side of the head-modifier pair $(pend[i-k], \ldots, pend[i+1+k])$ where the neighbouring subtrees are used for providing hints regarding possible additional modifiers of m and h that are

yet to be acquired. We use $k = 2$ in our experiments, for a total of 6 subtrees in total. This window approach is also used in the Easy-First parser of Goldberg and Elhadad (Goldberg and Elhadad, 2010) and works that extend it (Tratz and Hovy, 2011; Ma et al., 2012; Ma et al., 2013). However, unlike the previous work, which made use of extensive feature engineering and rich feature functions aiming at extracting the many relevant linguistic sub-structures from the 6 subtrees and their interactions, we provide the scoring function solely with the vector-encoding of the 6 subtrees in the window.

Modeling the labeled attachment score is more difficult than modeling the unlabeled score and is prone to more errors. Moreover, picking the label for an attachment will cause less cascading error in contrast to picking the wrong attachment, which will necessarily preclude the parser from reaching the correct tree structure. In order to partially overcome this issue, our scoring function is a sum of two auxiliary scoring function, one scoring unlabeled and the other scoring labeled attachments. The unlabeled attachment score term in the sum functions as a fallback which makes it easier for a parser to predict the attachment direction even when there is no sufficient certainty as to the label.

$$\begin{aligned} Score(pend, i, d, \ell) = Score_U(pend, i, d) \\ + Score_L(pend, i, d, \ell) \end{aligned}$$

Each of $Score_U$ and $Score_L$ are modeled as multilayer perceptrons:

$$\begin{aligned} &Score_U(pend, i, d) = MLP_U(x_i)[d] \\ &Score_L(pend, i, d, \ell) = MLP_L(x_i)[(d, \ell)] \\ &x_i = pend[i-2].c \circ \cdots \circ pend[i+3].c \end{aligned}$$

where MLP_U and MLP_L are standard multilayer perceptron classifiers with one hidden layer ($MLP_X(x) = W^2 g(W^1 x + b^1) + b^2$) and have output layers with size 2 and $2L$ respectively, [.] is an indexing operation, and we assume the values of d and (d, ℓ) are mapped to integer values.

4.5 Computational Complexity

The Easy-First parsing algorithm works in $O(n \log n)$ time (Goldberg and Elhadad, 2010). The parser in this works differ by three aspects: running a BI-LSTM encoder prior to parsing ($O(n)$); maintaining the tree representation during parsing (lines 11–22 in Algorithm 2) which take a constant time at each parsing step; and local scoring using an MLP rather than a linear classifier (again, a constant-time operation). Thus, the parser maintains the $O(n \log n)$ complexity of the Easy-First parser.

5 Training Algorithm

5.1 Loss and Parameter Updates

At each step of the parsing process we select the highest scoring action (i, d, ℓ). The goal of training is to set the Score function such that correct actions are scored above incorrect ones. We use a margin-based objective, aiming to maximize the margin between the highest scoring correct action and the set of incorrect actions. Formally, we define a hinge loss for each parsing step as follows:

$$\begin{aligned} max\{0, 1 - max_{(i,d,\ell) \in G} Score(pend, i, d, \ell) \\ + max_{(i',d',\ell') \in A \setminus G} Score(pend, i, d, \ell)\} \end{aligned}$$

where A is the set of all possible actions and G is the set of correct actions at the current stage.

As the scoring function depends on vector-encodings of all trees in the window, and each tree-encoding depends on the network's parameters, each parameter update will invalidate all the vector encodings, requiring a re-computation of the entire network. We thus sum the local losses throughout the parsing process, and update the parameter with respect to the sum of the losses at sentence boundaries. Since we are using hinge loss the gradients will become sparser as the training progresses. Fewer non-zero gradients could translate to unreliable updates. In order to increase gradient stability and training speed, we use a variation of mini-batch in which we update the parameters only after 50 errors were made. This assures us a sufficient number of gradients for every update thus minimizing the effect of gradient instability. The gradients of the entire network with respect to the sum of the losses are calculated using the backpropagation algorithm. Initial experiments with an SGD optimizer showed very instable results. We settled instead on using the ADAM optimizer (Kingma and Ba, 2015) which

worked well without requiring fiddling with learning rates.

5.2 Error-Exploration and Dynamic Oracle Training

At each stage in the training process, the parser assigns scores to all the possible actions $(i, d, \ell) \in A$. It then selects an action, applies it, and moves to the next step. Which action should be chosen? A sensible option is to define G as the set of actions that can lead to the gold tree, and following the highest scoring actions in this set. However, using training in this manner tends to suffer from error propagation at test time. The parser sees only states that result from following correct actions. The lack of examples containing errors in the training phase makes it hard for the parser to infer the best action given partly erroneous trees. In order to cope with this, we follow the *error exploration* training strategy, in which we let the parser follow the highest scoring action in A during training even if this action is incorrect, exposing it to states that result from erroneous decisions. This strategy requires defining the set G such that the correct actions to take are well-defined also for states that cannot lead to the gold tree. Such a set G is called a *dynamic oracle*. Error-exploration and dynamic-oracles were introduced by Goldberg and Nivre (2012).

The Dynamic Oracle A dynamic-oracle for the easy-first parsing system we use is presented in (Goldberg and Nivre, 2013). Briefly, the dynamic-oracle version of G defines the set of gold actions as the set of actions which does not increase the number of erroneous attachments more than the minimum possible (given previous erroneous actions). The number of erroneous attachments is increased in three cases: (1) connecting a modifier to its head prematurely. Once the modifier is attached it is removed from the pending list and therefore can no longer acquire any of its own modifiers; (2) connecting a modifier to an erroneous head, when the correct head is still on the pending list; (3) connecting a modifier to a correct head, but an incorrect label.

Dealing with cases (2) and (3) is trivial. To deal with (1), we consider as correct only actions in which the modifier is *complete*. To efficiently identify complete modifiers we hold a counter for each word which is initialized to the number of modifiers the word has in the gold tree. When applying an attachment the counter of the modifier's gold head word is decreased. When the counter reaches 0, the sub-tree rooted at that word has no pending modifiers, and is considered complete.

Aggressive Exploration We found that even when using error-exploration, after one iteration the model remembers the training set quite well, and does not make enough errors to make error-exploration effective. In order to expose the parser to more errors, we employ a cost augmentation scheme: we sometimes follow incorrect actions also if they score below correct actions. Specifically, when the score of the correct action is greater than that of the wrong action but the difference is smaller than the margin constant, we chose to follow the wrong action with probability p_{aug} (we use $p_{aug} = 0.1$ in our experiments). Pseudocode for the entire training algorithm is given in the supplementary material.

5.3 Out-of-vocabulary items and word-dropout

Due to the sparsity of natural language, we are likely to encounter at test time a substantial number of the words that did not appear in the training data (OOV words). OOV words are likely even when pre-training the word representations on a large unannotated corpora. A common approach is to designate a special "unknown-word" symbol, whose associated vector will be used as the word representation whenever an OOV word is encountered at test time. In order to train the unknown-word vector, a possible approach is to replace all the words appearing in the training corpus less than a certain number of times with the unknown-word symbol. This approach gives a good vector representation for unknown words but at the expense of ignoring many of the words from the training corpus.

We instead propose a variant of the word-dropout approach (Iyyer et al., 2015). During training, we replace a word with the unknown-word symbol with probability that is inversely proportional to frequency of the word. Formally, we replace a word w appearing $\#(w)$ times in the training corpus with the unknown symbol with a probability:

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

Using this approach we learn a vector representation for unknown words with minimal impact on the training of sparse words.

6 Implementation Details

Our Python implementation will be made available at the first author's website. We use the PyCNN wrapper of the CNN library[3] for building the computation graph of the network, computing the gradients using automatic differentiation, and performing parameter updates. We noticed the error on the development set does not improve after 20 iterations over the training set, therefore, we ran the training for 20 iterations. The sentences where shuffled between iterations. Non-projective sentences were skipped during training. We use the default parameters initialization, step sizes and regularization values provided by the PyCNN toolkit. The hyper-parameters of the final networks used for all the reported experiments are detailed in Table 1.

Word embedding dimension	100
POS tag embedding dimension	25
Relation embedding dimension	25
Hidden units in $Score_U$	100
Hidden units in $Score_L$	100
LSTM Dimensions (tree)	200
LSTM Layers (tree)	2
BI-LSTM Dimensions	100+100
BI-LSTM Layers	2
Mini-batch size	50
α (for word dropout)	0.25
p_{aug} (for exploration training)	0.1
g	$tanh$

Table 1: Hyper-parameter values used in experiments

Weiss et al (2015) stress the importance of careful hyperparameter tuning for achieving top accuracy in neural network based parser. We did not follow this advice and made very few attempts at hyper-parameter tuning, using manual hill climbing until something seemed to work with reasonable accuracy, and then sticking with it for the rest of the experiments.

[3] https://github.com/clab/cnn/tree/master/pycnn

7 Experiments and Results

We evaluated our parsing model to English and Chinese data. For comparison purposes we followed the setup of (Dyer et al., 2015).

Data For English, we used the Stanford Dependency (SD) (de Marneffe and Manning, 2008) conversion of the Penn Treebank (Marcus et al., 1993), using the standard train/dev/test splitswith the same predicted POS-tags as used in (Dyer et al., 2015; Chen and Manning, 2014). This dataset contains a few non-projective trees. Punctuation symbols are excluded from the evaluation.

For Chinese, we use the Penn Chinese Treebank 5.1 (CTB5), using the train/test/dev splits of (Zhang and Clark, 2008; Dyer et al., 2015) with gold part-of-speech tags, also following (Dyer et al., 2015; Chen and Manning, 2014).

When using external word embeddings, we also use the same data as (Dyer et al., 2015).[4]

Experimental configurations We evaluated the parser in several configurations BOTTOMUPPARSER is the baseline parser, not using the tree-encoding, and instead representing each item in $pending$ solely by the vector-representation (word and POS) of its head word. BOTTOMUPPARSER+HTLSTM is using our Hierarchical Tree LSTM representation. BOTTOMUPPARSER+HTLSTM+BI-LSTM is the Hierarchical Tree LSTM where we additionally use a BI-LSTM encoding for the head words. Finally, we added external, pre-trained word embeddings to the BOTTOMUPPARSER+HTLSTM+BI-LSTM setup. We also evaluated the final parsers in a –POS setup, in which we did not feed the parser with any POS-tags.

Results Results for English and Chinese are presented in Tables 2 and 3 respectively. For comparison, we also show the results of the Stack-LSTM transition-based parser model of Dyer et al (2015), which we consider to be a state-of-the-art greedy model which is also very competitive with search-based models, with and without pre-trained embeddings, and with and without POS-tags.

[4] We thank Dyer et al for sharing their data with us.

	Dev		Test	
	UAS	LAS	UAS	LAS
BOTTOMUPPARSER	83.3	79.0	82.7	78.6
+HTLSTM	92.4	90.1	92.0	89.8
+BI-LSTM input	93.0	90.5	92.6	90.2
+external embeddings	93.3	90.8	93.0	90.9
Dyer et al (2015) no external	92.7	90.4	92.4	90.0
Dyer et al (2015) w/ external	93.2	90.9	93.1	90.9
C&M (2014) w/ external	92.2	89.7	91.8	89.6
BOTTOMUP+ALL–POS	92.9	90.5	92.9	90.6
Dyer et al (2015) –POS	93.1	90.4	92.7	90.3

Table 2: English parsing results (SD)

	Dev		Test	
	UAS	LAS	UAS	LAS
BOTTOMUPPARSER	79.3	77.1	78.8	76.3
+HTLSTM	86.2	84.5	86.2	84.7
+BI-LSTM	86.2	84.5	86.1	84.4
+external embeddings	87.2	85.7	87.1	85.5
Dyer et al (2015) no external	86.3	84.7	85.7	84.1
Dyer et al (2015) w/ external	87.2	85.9	87.2	85.7
C&M (2014) no external	84.0	82.4	83.9	82.4
BOTTOMUP+ALL –POS	82.9	80.0	82.6	79.5
Dyer et al (2015) –POS	82.8	79.8	82.2	79.1

Table 3: Chinese parsing results (CTB5)

The trends are consistent across the two languages. The baseline Bottom-Up parser performs very poorly. This is expected, as only the headword of each subtree is used for prediction. When adding the tree-encoding, results jump to near state-of-the-art accuracy, suggesting that the composed vector representation is indeed successful in capturing predictive structural information. Replacing the head-words with their BI-LSTM encodings results in another increase in accuracy for English, outperforming the Dyer et al (S-LSTM no external) models on the test-set. Adding the external pre-trained embeddings further improves the results for both our parser and Dyer et al's model, closing the gap between them. When POS-tags are not provided as input, the numbers for both parsers drop. The drop is small for English and large for Chinese, and our parser seem to suffer a little less than the Dyer et al model.

Importance of the dynamic oracle We also evaluate the importance of using the dynamic oracle and error-exploration training, and find that they are indeed important for achieving high parsing accuracies with our model (Table 4).

	English		Chinese	
	UAS	LAS	UAS	LAS
RAND	93.0	90.5	86.2	84.5
RAND-NODYN	92.2	89.8	85.7	84.1
EXT	93.3	90.8	87.2	85.7
EXT-NODYN	92.7	90.4	86.6	85.1

Table 4: Effect of the error-exploration training (dynamic-oracle) on dev set accuracy in English and Chinese. RAND: random initialization. EXT: pre-trained external embeddings.

When training without error-exploration (that is, the parser follows only correct actions during training and not using the dynamic aspect of the oracle), accuracies of unseen sentences drop by between 0.4 and 0.8 accuracy points (average 0.58). This is consistent with previous work on training with error-exploration and dynamic oracles (Goldberg and Nivre, 2013), showing that the technique is not restricted to models trained with sparse linear models.

Comparison to other state-of-the-art parsers Our main point of comparison is the model of Dyer et al, which was chosen because it is (a) a very strong parsing model; and (b) is the closest to ours in the literature: a greedy parsing model making heavy use of LSTMs. To this end, we tried to make the comparison to Dyer et al as controlled as possible, using the same dependency annotation schemes, as well as the same predicted POS-tags and the pre-trained embeddings (when applicable).

It is also informative to position our results with respect to other state-of-the-art parsing results reported in the literature, as we do in Table 5. Here, some of the comparisons are less direct: some of the results use different dependency annotation schemes[5], as well as different predicted POS-tags, and different pre-trained word embeddings. While the numbers are not directly comparable, they do give a good reference as to the expected range of

[5]Our English parsing experiments use the Stanford Dependencies scheme, while other work use less informative dependency relations which are based on the Penn2Malt converter, using the Yamada and Matsumoto head rules. From our experience, this conversion is somewhat easier to parse, resulting in numbers which are about 0.3-0.4 points higher than Stanford Dependencies.

state-of-the-art parsing results. Our system's English parsing results are in range of state-of-the-art and the Chinese parsing results surpass it. These numbers are achieved while using a greedy, bottom up parsing method without any search, and while relying solely on the compositional tree representations.

8 Related Work

We survey two lines of related work: methods for encoding trees as vectors, and methods for parsing with vector representations.

The popular approach for encoding trees as vectors is using recursive neural networks (Goller and Kuchler, 1996; Socher et al., 2010; Tai et al., 2015). Recursive neural networks represent the vector of a parent node in a tree as a function of its children nodes. However, the functions are usually restricted to having a fixed maximum arity (usually two) (Socher et al., 2010; Tai et al., 2015; Socher, 2014). While trees can be binarized to cope with the arity restriction, doing so results in deep trees which in turn leads to the vanishing gradient problem when training. To cope with the vanishing gradients, (Tai et al., 2015) enrich the composition function with a gating mechanism similar to that of the LSTM, resulting in the so-called Tree-LSTM model. Another approach is to allow arbitrary arities but ignoring the sequential nature of the modifiers, e.g. by using a bag-of-modifiers representation or a convolutional layer (Tai et al., 2015; Zhu et al., 2015). In contrast, our tree encoding method naturally allows for arbitrary branching trees by relying on the well established LSTM sequence model, and using it as a black box. Very recently, Zhang et al. (2015) proposed an RNN-based tree encoding which is similar to ours in encoding the sequence of modifiers as an RNN. Unlike our bottom-up encoder, their method works top-down, and is therefore not readily applicable for parsing. On the other hand the top-down approach is well suited for generation. In future work, it could be interesting to combine the bottom-up and top-down approaches in an encoder-decoder framework (Sutskever et al., 2014; Kiros et al., 2015). Work by Dyer et al (2016), that was submitted in parallel to ours, introduces a similar LSTM-based representation of syntactic constituents in the context of phrase-grammar parsing.

In terms of parsing with vector representations, there are four dominant approaches: search based parsers that use local features that are fed to a neural-network classifier (Pei et al., 2015; Durrett and Klein, 2015); greedy transition based parsers that use local features that are fed into a neural-network classifier (Chen and Manning, 2014; Weiss et al., 2015), sometimes coupled with a node composition function (Dyer et al., 2015; Watanabe and Sumita, 2015); bottom up parsers that rely solely on recursively combined vector encodings of subtrees (Socher et al., 2010; Stenetorp, 2013; Socher et al., 2013a); and parse-reranking approaches that first produce a k-best list of parses using a traditional parsing technique, and then score the trees based on a recursive vector encoding of each node (Le and Zuidema, 2014; Le and Zuidema, 2015; Zhu et al., 2015).

Our parser is a greedy, bottom up parser that relies on compositional vector encodings of subtrees as its sole set of features. Unlike the re-ranking approaches, we do not rely on an external parser to provide k-best lists. Unlike the bottom-up parser in (Socher et al., 2010) that only parses sentences of up to 15 words and the parser of (Stenetorp, 2013) that achieves very low parsing accuracies, we parse arbitrary sentences with near state-of-the-art accuracy. Unlike the bottom up parser in (Socher et al., 2013a) we do not make use of a grammar. The parser of (Weiss et al., 2015) obtains exceptionally high results using local features and no composition function. The greedy version of their parser uses extensive tuning of hyper-parameters and network depth in order to squeeze every possible bit of accuracy. Adding beam search on top of that further improves results. Due to our much more limited resources, we did not perform a methodological search over hyper-parameters, and explored only a tiny space of the possible hyper-parameters, and our parser does not perform search. Finally, perhaps closest to our approach is the greedy, transition-based parser of (Dyer et al., 2015) that also works in a bottom-up fashion, and incorporates an LSTM encoding of the input tokens and hierarchical vector composition into its scoring mechanism. Indeed, that parser obtains similar scores to ours, although we obtain somewhat better results when not using pre-trained embeddings. We differ from the parser of Dyer et

System	Method	Representation	Emb	PTB-YM	PTB-SD	CTB	Runtime
ZhangNivre11	transition (beam)	large feature set (sparse)	–	92.9	–	86.0	$O(n)+$
Martins13	graph, 3rd order+	large feature set (sparse)	–	92.8	93.07	–	$O(n^4)$
Pei15	graph, 2nd order	large feature set (dense)	–	92.99	–	–	$O(n^3)$
This Work	EasyFirst (greedy)	Rec-LSTM encoding	–	–	92.6	86.1	$O(n \log n)$
Weiss15	transition (greedy)	large feature set (dense)	YES	–	93.19	–	$O(n)$
Weiss15	transition (beam)	large feature set (dense)	YES	–	93.99	–	$O(n)+$
Pei15	graph, 2nd order	large feature set (dense)	YES	93.29	–	–	$O(n^3)$
LeZuidema14	reranking /blend	inside-outside recursive net	YES	93.12	93.84	–	$O(n^3)$
Zhu15	reranking /blend	recursive conv-net	YES	93.83	–	85.71	$O(n)+$
This Work	EasyFirst (greedy)	Rec-LSTM encoding	YES	–	93.0	87.1	$O(n \log n)$

Table 5: Parsing results (UAS) of various state-of-the-art parsing systems on the English and Chinese datasets. The systems that use embeddings use different pre-trained embeddings. English results use predicted POS tags (different systems use different taggers), while Chinese results use gold POS tags. **PTB-YM**: English PTB, Yamada and Matsumoto head rules. **PTB-SD**: English PTB, Stanford Dependencies (different systems may use different versions of the Stanford converter. **CTB**: Chinese Treebank. *reranking /blend* in method column indicates a reranking system where the reranker score is interpolated with the base-parser's score. The reranking systems' runtimes are those of the base parsers they use. $O(n)+$ indicates a linear-time system with a large multiplicative constant. The different systems and the numbers reported from them are taken from: ZhangNivre11: (Zhang and Nivre, 2011); Martins13: (Martins et al., 2013); Weiss15 (Weiss et al., 2015); Pei15: (Pei et al., 2015); LeZuidema14 (Le and Zuidema, 2014); Zhu15: (Zhu et al., 2015).

al by having a more elaborate vector-composition function, relying solely on the compositional representations, and performing fully bottom-up parsing without being guided by a stack-and-buffer control structure.

9 Conclusions and Future Work

We suggest a compositional vector representation of parse trees that relies on a recursive combination of recurrent-neural network encoders, and demonstrate its effectiveness by integrating it in a bottom-up easy-first parser. Future extensions in terms of parsing include the addition of beam search, handling of unknown-words using character-embeddings, and adapting the algorithm to constituency trees. We also plan to establish the effectiveness of our Hierarchical Tree-LSTM encoder by applying it to more semantic vector representation tasks, i.e. training tree representation for capturing sentiment (Socher et al., 2013b; Tai et al., 2015), semantic sentence similarity (Marelli et al., 2014) or textual inference (Bowman et al., 2015).

Acknowledgements This research is supported by the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI) and the Israeli Science Foundation (grant number 1555/15).

References

Ian Goodfellow Yoshua Bengio and Aaron Courville. 2016. Deep learning. Book in preparation for MIT Press, `http://www.deeplearningbook.org`.

Samuel R. Bowman, Gabor Angeli, Christopher Potts, and Christopher D. Manning. 2015. A large annotated corpus for learning natural language inference. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 632–642, Lisbon, Portugal, September. Association for Computational Linguistics.

Danqi Chen and Christopher Manning. 2014. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 740–750, Doha, Qatar, October. Association for Computational Linguistics.

Kyunghyun Cho. 2015. Natural language understanding with distributed representation. *CoRR*, abs/1511.07916.

Michael Collins. 1997. Three generative, lexicalised models for statistical parsing. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*, pages 16–23, Madrid, Spain, July. Association for Computational Linguistics.

Marie-Catherine de Marneffe and Christopher D. Manning. 2008. Stanford dependencies manual. Technical report, Stanford University.

Greg Durrett and Dan Klein. 2015. Neural crf parsing. 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 302–312,

Beijing, China, July. Association for Computational Linguistics.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. 2015. Transition-based dependency parsing with stack long short-term memory. 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 334–343, Beijing, China, July. Association for Computational Linguistics.

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A. Smith. 2016. Recurrent neural network grammars. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 199–209, San Diego, California, June. Association for Computational Linguistics.

Jason Eisner. 1996. Three new probabilistic models for dependency parsing: An exploration. In *16th International Conference on Computational Linguistics, Proceedings of the Conference, COLING 1996, Center for Sprogteknologi, Copenhagen, Denmark, August 5-9, 1996*, pages 340–345.

Jason Eisner. 2000. Bilexical grammars and their cubic-time parsing algorithms. *Advances in Probabilistic and Other Parsing Technologies*.

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

Yoav Goldberg and Michael Elhadad. 2010. 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, Los Angeles, California, June. Association for Computational Linguistics.

Yoav Goldberg and Joakim Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. In *Proceedings of COLING 2012*, pages 959–976, Mumbai, India, December. The COLING 2012 Organizing Committee.

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

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

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

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

Ozan Irsoy and Claire Cardie. 2014. Opinion mining with deep recurrent neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 720–728, Doha, Qatar, October. Association for Computational Linguistics.

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

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference for Learning Representations*, San Diego, California.

Ryan Kiros, Yukun Zhu, Ruslan Salakhutdinov, Richard S. Zemel, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. 2015. Skip-thought vectors. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 3294–3302.

Sandra Kübler, Ryan T. McDonald, and Joakim Nivre. 2009. *Dependency Parsing*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers.

Phong Le and Willem Zuidema. 2014. 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. Association for Computational Linguistics.

Phong Le and Willem Zuidema. 2015. The forest convolutional network: Compositional distributional semantics with a neural chart and without binarization. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1155–1164, Lisbon, Portugal, September. Association for Computational Linguistics.

Wang Ling, Chris Dyer, Alan W. Black, and Isabel Trancoso. 2015. Two/too simple adaptations of word2vec for syntax problems. In *NAACL HLT 2015, The 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Denver, Colorado, USA, May 31 - June 5, 2015*, pages 1299–1304.

Ji Ma, Tong Xiao, Jingbo Zhu, and Feiliang Ren. 2012. Easy-first Chinese POS tagging and dependency parsing. In *Proceedings of COLING 2012*, pages 1731–1746, Mumbai, India, December. The COLING 2012 Organizing Committee.

Ji Ma, Jingbo Zhu, Tong Xiao, and Nan Yang. 2013. Easy-first pos tagging and dependency parsing with beam search. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 110–114, Sofia, Bulgaria, August. Association for Computational Linguistics.

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

Marco Marelli, Luisa Bentivogli, Marco Baroni, Raffaella Bernardi, Stefano Menini, and Roberto Zamparelli. 2014. Semeval-2014 task 1: Evaluation of compositional distributional semantic models on full sentences through semantic relatedness and textual entailment. In *Proceedings of the 8th International Workshop on Semantic Evaluation (SemEval 2014)*, pages 1–8, Dublin, Ireland, August. Association for Computational Linguistics and Dublin City University.

Andre Martins, Miguel Almeida, and Noah A. Smith. 2013. Turning on the turbo: Fast third-order non-projective turbo parsers. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 617–622, Sofia, Bulgaria, August. Association for Computational Linguistics.

Ryan McDonald. 2006. *Discriminative Training and Spanning Tree Algorithms for Dependency Parsing*. Ph.D. thesis, University of Pennsylvania.

Tomas Mikolov, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010*, pages 1045–1048.

Wenzhe Pei, Tao Ge, and Baobao Chang. 2015. An effective neural network model for graph-based dependency parsing. 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 313–322, Beijing, China, July. Association for Computational Linguistics.

Mike Schuster and Kuldip K. Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE Trans. Signal Processing*, 45(11):2673–2681.

Richard Socher, Christopher Manning, and Andrew Ng. 2010. Learning Continuous Phrase Representations and Syntactic Parsing with Recursive Neural Networks. In *Proceedings of the Deep Learning and Unsupervised Feature Learning Workshop of (NIPS) 2010*, pages 1–9.

Richard Socher, John Bauer, Christopher D. Manning, and Andrew Y. Ng. 2013a. Parsing with compositional vector grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, ACL 2013, 4-9 August 2013, Sofia, Bulgaria, Volume 1: Long Papers*, pages 455–465.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. 2013b. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642, Seattle, Washington, USA, October. Association for Computational Linguistics.

Richard Socher. 2014. *Recursive Deep Learning For Natural Language Processing and Computer Vision*. Ph.D. thesis, Stanford University, August.

Pontus Stenetorp. 2013. Transition-based Dependency Parsing Using Recursive Neural Networks. In *Deep Learning Workshop at the 2013 Conference on Neural Information Processing Systems (NIPS)*, Lake Tahoe, Nevada, USA, December.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3104–3112.

Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. 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. Association for Computational Linguistics.

Stephen Tratz and Eduard Hovy. 2011. A fast, accurate, non-projective, semantically-enriched parser. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 1257–1268, Edinburgh, Scotland, UK., July. Association for Computational Linguistics.

Taro Watanabe and Eiichiro Sumita. 2015. Transition-based neural constituent parsing. 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 1169–1179, Beijing, China, July. Association for Computational Linguistics.

David Weiss, Chris Alberti, Michael Collins, and Slav Petrov. 2015. Structured training for neural network transition-based parsing. 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 323–333, Beijing, China, July. Association for Computational Linguistics.

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

Yue Zhang and Joakim Nivre. 2011. Transition-based dependency parsing with rich non-local features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 188–193, Portland, Oregon, USA, June. Association for Computational Linguistics.

Xingxing Zhang, Liang Lu, and Mirella Lapata. 2015. Tree recurrent neural networks with application to language modeling. *CoRR*, abs/1511.00060.

Chenxi Zhu, Xipeng Qiu, Xinchi Chen, and Xuanjing Huang. 2015. A re-ranking model for dependency parser with recursive convolutional neural network. 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 1159–1168, Beijing, China, July. Association for Computational Linguistics.

Appendix: Training Algorithm Pseudocode

Algorithm 3 Training on annotated corpus

Input: Sentences $w^1, \ldots, w^m$
Input: Tree annotations $T^1, \ldots, T^m$
Input: Number of epochs to train

$V \leftarrow InitializeVectors()$
$Loss \leftarrow []$

for $epoch \in \{1, \ldots, Epochs\}$ **do**
  for $S, T \in \{(w^1, T^1), \ldots, (w^m, T^m)\}$ **do**
    $Loss \leftarrow TrainSentence\,(S, V[w_1, \ldots, w_n], T, Loss)$

    if $|Loss| > 50$ **then**
      $SumLoss \leftarrow sum(Loss)$
      $Call\ ADAM\ to\ minimize\ SumLoss$
      $Loss \leftarrow []$

(See Algorithm 4, training of a single sentence, on next page.)

Algorithm 4 Training on a single sentence with dynamic oracle algorithm

function TRAINSENTENCE($w, v, T, Loss$)
  Input: Sentence $w = w_1, \ldots, w_n$
  Input: Vectors v_i corresponding to inputs w_i
  Input: Annotated tree T in the form of (h, m, rel) triplets
  Input: List $Loss$ to which loss expressions are added

  for $i \in 1, \ldots, n$ **do**
    $unassigned[i] \leftarrow |Children(w_i)|$
    $pend[i].id \leftarrow i$
    $pend[i].e_l \leftarrow RNN_L.init().append(v_i)$
    $pend[i].e_r \leftarrow RNN_R.init().append(v_i)$

  while $|pend| > 1$ **do**
    $G, W \leftarrow \{\}, \{\}$

    for $(i, d, rel) \in \{1 \leq i < |pend|, d \in \{l, r\}, rel \in Relations\}$ **do**
      if $d = l$ **then** $m, h \leftarrow$ pend$[i]$, pend$[i+1]$
      else $m, h \leftarrow$ pend$[i+1]$, pend$[i]$

      if $unassigned[m.id] \neq 0 \vee \exists_{\ell \neq rel}(h, m, \ell) \in T$ **then**
        $W.append((h, m, rel))$
      else $G.append((h, m, rel))$

    $h_G, m_G, rel_G \leftarrow argmax_{(i,d,\ell) \in G} Score(pend, i, d, \ell)$
    $h_W, m_W, rel_W \leftarrow argmax_{(i,d,\ell) \in W} Score(pend, i, d, \ell)$
    $score_G \leftarrow Score(h_G, m_G, rel_G)$
    $score_W \leftarrow Score(h_W, m_W, rel_W)$

    if $score_G - score_W < 0$ **then**
      $h, m, rel, score \leftarrow h_W, m_W, rel_W, score_W$
    else if $score_G - score_W > 1 \vee random() < p_{aug}$ **then**
      $h, m, rel, score \leftarrow h_G, m_G, rel_G, score_G$
    else
      $h, m, rel, score \leftarrow h_W, m_W, rel_W, score_W$
    if $score_G - score < 1$ **then**
      $Loss.append(1 - score_G + score)$

    $m.c = m.e_l \circ m.e_r$
    $m.enc = g(W(m.c \circ rel) + b)$
    if $h.id < m.id$ **then** $h.e_l.append(m.enc)$
    else $h.e_r.append(m.enc)$

    $unassigned[T_{Parent}(m).id] \leftarrow unassigned[T_{Parent}(m).id] - 1$
    pend.remove(m)
  return Loss