# Simple and Accurate Dependency Parsing Using Bidirectional LSTM Feature Representations

**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 present a simple and effective scheme for dependency parsing which is based on bidirectional-LSTMs (BiLSTMs). Each sentence token is associated with a BiLSTM vector representing the token in its sentential context, and feature vectors are constructed by concatenating a few BiLSTM vectors. The BiLSTM is trained jointly with the parser objective, resulting in very effective feature extractors for parsing. We demonstrate the effectiveness of the approach by applying it to a greedy transition based parser as well as to a globally optimized graph-based parser. The resulting parsers have very simple architectures, and match or surpass the state-of-the-art accuracies on English and Chinese.

## 1 Introduction

The focus of this paper is on feature representation for dependency parsing, using recent techniques from the neural-networks ("deep learning") literature. Modern approaches to dependency parsing can be broadly categorized into graph-based and transition-based parsers (Kübler et al., 2008). Graph-based parsers (McDonald, 2006) treat parsing as a search-based structured prediction problem in which the goal is learning a scoring function over dependency trees such that the correct tree is scored above all other trees. Transition-based parsers (Nivre, 2004; Nivre, 2008) treat parsing as a sequence of actions that produce a parse tree, and a classifier is trained to score the possible actions at each stage of the process and guide the parsing process. Perhaps the simplest graph-based parsers are arc-factored (first order) models (McDonald, 2006), in which the scoring function for a tree decomposes over the individual arcs of the tree. More elaborate models look at larger (overlapping) parts, requiring more sophisticated inference and training algorithms (Martins et al., 2009; Koo and Collins, 2010). The basic transition-based parsers work in a greedy manner, performing a series of locally-optimal decisions, and boast very fast parsing speeds. More advanced transition-based parsers introduce some search into the process using a beam (Zhang and Clark, 2008) or dynamic programming (Huang and Sagae, 2010).

Regardless of the details of the parsing framework being used, a crucial step in parser design is choosing the right *feature function* for the underlying statistical model. Recent work (see Section 2.2 for an overview) attempt to alleviate parts of the feature function design problem by moving from linear to non-linear models, enabling the modeler to focus on a small set of "core" features and leaving it up to the machine-learning machinery to come up with good feature combinations (Chen and Manning, 2014; Pei et al., 2015; Lei et al., 2014; Taub-Tabib et al., 2015). However, the need to carefully define a set of core features remains. For example, the work of (Chen and Manning, 2014) uses 18 different elements in its feature function, while the work of (Pei et al., 2015) uses 21 different elements. Other works, notably (Dyer et al., 2015; Le and Zuidema, 2014), propose more sophisticated feature representations, in which the feature engineering is replaced with architecture engineering.

In this work, we suggest an approach which is much simpler in terms of both feature engineering and architecture engineering. Our proposal (Section 3) is centered around BiRNNs (Irsoy and Cardie, 2014; Schuster and Paliwal, 1997), and more specifically BiLSTMs (Graves, 2008), which are strong and trainable sequence models (see Section 2.3). The BiLSTM excels at representing elements in a sequence (i.e., words) together with their contexts, capturing the element and an "infinite" window around it. We represent each word by its BiLSTM encoding, and use a concatenation of a minimal set of such BiLSTM encodings as our feature function, which is then passed to a non-linear scoring function (multi-layer perceptron). Crucially, the BiLSTM is trained with the rest of the parser in order to learn a good feature representation for the parsing problem. If we set aside the inherent complexity of the BiLSTM itself and treat it as a black box, our proposal results in a frustratingly simple feature extractor.

We demonstrate the effectiveness of the approach by using the BiLSTM feature extractor in two parsing architectures, transition-based (Section 4) as well as a graph-based (Section 5). In the graph-based parser, we jointly train a structured-prediction model on top of a BiLSTM, propagating errors from the structured objective all the way back to the BiLSTM feature-encoder. To the best of our knowledge, we are the first to perform such end-to-end training of a structured prediction model and a recurrent feature extractor.

Aside from the novelty of the BiLSTM feature extractor and the end-to-end structured training, we rely on existing models and techniques from the parsing and structured prediction literature. We stick to the simplest parsers in each category – greedy inference for the transition-based architecture, and a first-order, arc-factored model for the graph-based architecture. Despite the simplicity of the parsing architectures and the feature functions, we achieve state-of-the-art parsing accuracies in both English (93.2 UAS) and Chinese (86.4 UAS), using a first-order parser with two features and while training solely on Treebank data, without relying on semi-supervised signals such as pre-trained word embeddings (Chen and Manning, 2014), word-clusters (Koo et al., 2008), or techniques such as tri-training (Weiss et al., 2015). When including also pre-trained word embeddings, we obtain further improvements, with accuracies of 93.6 UAS (English) and 87.3 UAS (Chinese) for a greedy transition-based parser with 11 features, and 93.3 UAS (En) / 86.6 (Ch) for a greedy transition-based parser with 4 features.

## 2 Background and Notation

**Notation** We use $x_{1:n}$ to denote a sequence of $n$ vectors $x_1, \cdots, x_n$. $F_\theta(\cdot)$ is a function parameterized with parameters $\theta$. We write $F_L(\cdot)$ as a shortcut to $F_{\theta_L}$ – an instantiation of $F$ with a specific set of parameters $\theta_L$. We use $\circ$ to denote a vector concatenation operation, and $v[i]$ to denote an indexing operation taking the $i$th element of a vector $v$.

### 2.1 Feature Functions in Dependency Parsing

Traditionally, state-of-the-art parsers rely on linear models over hand-crafted feature functions. The feature functions look at core components (e.g. "word on top of stack", "leftmost child of the second-to-top word on the stack", "distance between the head and the modifier words"), and are comprised of several templates, where each template instantiates a binary indicator function over a conjunction of core elements (resulting in features of the form "word on top of stack is X and leftmost child is Y and ..."). The design of the feature function – which components to consider and which combinations of components to include – is a major challenge in parser design. Once a good feature function is proposed in a paper it is usually adopted in later works, and sometimes tweaked to improve performance. Examples of good feature functions are the feature-set proposed by Zhang and Nivre (2011) for transition-based parsing (including roughly 20 core components and 72 feature templates), and the feature-set proposed by McDonald et al (2005) for graph-based parsing, with the paper listing 18 templates for a first-order parser, and the MSTParser's first-order feature-extractor code containing roughly a hundred feature templates.

The core features in a transition-based parser usually look at information such as the word-identity and part-of-speech (POS) tags of a fixed number of words on top of the stack, a fixed number of words on the top of the buffer, the modifiers (usually left-most and right-most) of items on the stack and on the buffer, the number of modifiers of these elements, parents of words on the stack, and the length of the spans spanned by the words on the stack. The core features of a first-order graph-based parser usually take into account the word and POS of the head and the modifier items, as well as the POS-tags of the items around to the head and the modifier, the POS tags of items between the head and the modifier, and the distance and direction between the head and the modifier.

### 2.2 Related Research Efforts

Coming up with a good feature-set for a parser is a hard and time consuming task, and many researchers attempt to reduce the required manual efforts. The work of Lei et al (2014) suggest a low-rank tensor representation to automatically find good feature combinations. Taub-Tabib et al (2015) suggest a kernel-based approach to implicitly consider all possible feature combinations over sets of core-features. The recent popularity of neural-networks prompted a move from templates of sparse, binary indicator features to dense core feature encodings fed into non-linear classifiers. Chen and Manning (2014) encode each core feature of a greedy transition-based parser as a dense low-dimensional vector, and the vectors are then concatenated and fed into a non-linear classifier (multi-layer perceptron) which can potentially capture arbitrary feature combinations. Weiss et al (2015) showed further gains using the same approach coupled with a somewhat improved set of core features, a more involved network architecture with skip-layers, beam search-decoding, and careful hyper-parameter tuning. Pei et al (2015) apply a similar methodology to graph-based parsing. While the move to neural-network classifiers alleviates the need for hand-crafting feature-combinations, the need to carefully define a set of core features remain. For example, the feature representation in (Chen and Manning, 2014) is a concatenation of 18 word vectors, 18 POS vectors and 12 dependency-label vectors.[1]

The above works tackle the effort in hand-crafting effective feature combinations. A different line of work attacks the feature-engineering problem by suggesting novel neural-network architectures for encoding the parser state, including intermediately-built subtrees, as vectors which are then fed to non-linear classifiers. In the work of Dyer et al (2015), the entire stack and buffer of a transition-based parser are encoded as a stack-LSTMs, where each stack element is itself based on a compositional representation of parse trees. Le and Zuidema (2014) encode each tree node as two compositional representations capturing the inside and outside structures around the node, and feed the representations into a reranker. A similar reranking approach, this time based on convolutional neural networks, is taken by Zhu et al (2015). Finally, in Kiperwasser and Goldberg (2016) we present an Easy-First parser based on a novel hierarchical-LSTM tree encoding.

In contrast to these, the approach we present in this work results in much simpler feature functions, without resorting to elaborate network architectures or compositional tree representations.

### 2.3 Bidirectional Recurrent Neural Networks

Recurrent neural networks (RNNs) are statistical learners for modeling sequential data. An RNN allows to model the $i$th element in the sequence based on the past – the elements $x_{1:i}$ up to and including it. The RNN model provides a framework for conditioning on the entire history $x_{1:i}$ without resorting to the Markov assumption which is traditionally used for modeling sequences. RNNs were shown to be capable of learning to count, as well as to model line lengths and complex phenomena such as bracketing and code indentation (Karpathy et al., 2015). Our proposed feature extractors are based on a bidirectional recurrent neural network (BiRNN), an extension of RNNs that take into account both the past $x_{1:i}$ and the future $x_{i:n}$. We use a specific flavor of RNN called a long short-term memory network (LSTM). For brevity, we treat RNN as an abstraction, without getting into the mathematical details of the implementation of the RNNs and LSTMs. For further details on RNNs and LSTMs, the reader is referred to (Goldberg, 2015; Cho, 2015).

[1]In all of these neural-network based approaches, the vector representations of words were initialized using pre-trained word-embeddings derived from a large corpus external to the training data. This puts the approaches in the semi-supervised category, making it hard to tease apart the contribution of the automatic feature-combination component from that of the semi-supervised component.

The recurrent neural network (RNN) abstraction is a parameterized function $\textsc{Rnn}_\theta(x_{1:n})$ mapping a sequence of $n$ input vectors $x_{1:n}$, $x_i \in \mathbb{R}^{d_{in}}$ to a sequence of $n$ output vectors $h_{1:n}, h_i \in \mathbb{R}^{d_{out}}$. Each output vector $h_i$ is conditioned on all the input vectors $x_{1:i}$, and can be thought of as a *summary* of the prefix $x_{1:i}$ of $x_{1:n}$. In our notation, we ignore the intermediate vectors $h_{1:n-1}$ and take the output of $\textsc{Rnn}_\theta(x_{1:n})$ to be the vector $h_n$.

A *bidirectional RNN* is composed of two RNNs, $\textsc{Rnn}_F$ and $\textsc{Rnn}_R$, one reading the sequence in its regular order, and the other reading it in reverse. Concretely, given a sequence of vectors $x_{1:n}$ and a desired index $i$, the function $\textsc{BiRnn}_\theta(x_{1:n}, i)$ is defined as:

$$\textsc{BiRnn}_\theta(x_{1:n}, i) = \textsc{Rnn}_F(x_{1:i}) \circ \textsc{Rnn}_R(x_{n:i})$$

The vector $v_i = \textsc{BiRnn}(x_{1:n}, i)$ is then a representation of the $i$th item in $x_{1:n}$, taking into account both the entire history $x_{1:i}$ and the entire future $x_{i:n}$. We can view the BiRNN encoding of an item $i$ as representing the item $i$ together with a context of an infinite window around it.

**Computational Complexity** Computing the BiRNN vectors encoding of the $i$th element of a sequence $x_{1:n}$ requires $O(n)$ time for computing the two RNNs and concatenating their outputs. A naive approach of computing the bidirectional representation of all $n$ elements result in $O(n^2)$ computation. However, it is trivial to compute the BiRNN encoding of all sequence items in linear time by pre-computing $\textsc{Rnn}_F(x_{1:n})$ and $\textsc{Rnn}_R(x_{n:1})$, keeping the intermediate representations, and concatenating the required elements as needed.

**BiRNN Training** Initially, the BiRNN encodings $v_i$ do not capture any particular information. During training, the encoded vectors $v_i$ are fed into further network layers, until at some point a prediction is made, and a loss is incurred. The back-propagation algorithm is used to compute the gradients of all the parameters in the network (including the BiRNN parameters) with respect to the loss, and an optimizer is used to update the parameters according to the gradients. The training procedure causes the BiRNN function to extract from the input sequence $x_{1:n}$ the relevant information for the task task at hand.

**Going deeper** A *deep RNN* (or $k$-layer RNN) is composed of $k$ RNN functions $\textsc{Rnn}_1, \cdots, \textsc{Rnn}_k$ that feed into each other: the output $h^\ell_{1:n}$ of $\textsc{Rnn}_\ell$ becomes the input of $\textsc{Rnn}_{\ell+1}$. Stacking RNNs in this way was empirically shown to be effective. Finally, in a *deep bidirectional RNN*, both $\textsc{Rnn}_F$ and $\textsc{Rnn}_R$ are $k$-layer RNNs, and $\textsc{BiRnn}^\ell(x_{1:n}, i) = v^\ell_i = h^\ell_{F,i} \circ h^\ell_{R,i}$. In this work, we use BiRNNs and deep-BiRNNs interchangeably, specifying the number of layers when needed.

**Historical Notes** RNNs were introduced by Elamn (Elman, 1990), and extended to BiRNNs by (Schuster and Paliwal, 1997). The LSTM variant of RNNs is due to (Hochreiter and Schmidhuber, 1997). BiLSTMs were recently popularized by Graves (2008), and deep BiRNNs were introduced to NLP by Irsoy and Cardie (2014), who used them for sequence tagging.

## 3 Our Approach

We propose to replace the hand-crafted feature functions in favor of minimally-defined feature functions which make use of automatically learned Bidirectional LSTM representations.

Given an $n$ words input sentence $s$ with words $w_1, \ldots, w_n$ together with the corresponding POS tags $t_1, \ldots, t_n$,[2] we associate each word $w_i$ and POS $t_i$ with embedding vectors $e(w_i)$ and $e(t_i)$, and create a sequence of input vectors $x_{1:n}$ in which each $x_i$ is a concatenation of the corresponding word and POS vectors:

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

The embeddings are trained together with the model. This encodes each word in isolation, disregarding its context. We introduce context by representing each input element as its (deep) BiLSTM vector, $v_i$:

$$v_i = \textsc{BiLstm}(x_{1:n}, i)$$

Our feature function $\phi$ is then a concatenation of a small number of BiLSTM vectors. The exact feature function is parser dependent and will be discussed when discussing the corresponding parsers.

[2]In this work the tag sequence is assumed to be given, and in practice is predicted by an external model. Future work will address relaxing this assumption.

The resulting feature vectors are then scored using a non-linear function, namely a multi-layer perceptron with one hidden layer (MLP):

$$MLP_\theta(x) = W^2 \cdot tanh(W^1 \cdot x + b^1) + b^2$$

where $\theta = \{W^1, W^2, b^1, b^2\}$ are the model parameters.

Beside using the BiLSTM-based feature functions, we make use of standard parsing techniques. Crucially, the BiLSTM is trained jointly with the rest of the parsing objective. This allows it to learn representations which are suitable for the parsing task.

Consider a concatenation of two BiLSTM vectors ($v_i \circ v_j$) scored using an MLP. The scoring function has access to the words and POS-tags of $v_i$ and $v_j$, as well as the words and POS-tags of the words in an infinite window surrounding them. As LSTMs are known to capture length and sequence position information, it is very plausible that the scoring function can be sensitive also to the distance between $i$ and $j$, their ordering, and the sequential material between them.

**Parsing-time Complexity** Once the BiLSTM is trained, parsing is performed by first computing the BiLSTM encoding $v_i$ for each word in the sentence (a linear time operation).[3] Then, parsing proceeds as usual, where the feature extraction involves a concatenation of a small number of the pre-computed $v_i$ vectors.

## 4 Transition-based Parser

We begin by integrating the feature extractor in a transition-based parser (Nivre, 2008). We follow the notation in (Goldberg and Nivre, 2013). The transition-based parsing framework assumes a transition system, an abstract machine that processes sentences and produces parse trees. The transition system has a set of configurations and a set of transitions which are applied to configurations. When parsing a sentence, the system is initialized to an initial configuration based on the input sentence, and transitions are repeatedly applied to this configuration. After a finite number of transitions, the system arrives at a terminal configuration, and a parse tree is read off the terminal configuration. In a greedy parser, a classifier is used to choose the transition to take in each configuration, based on features extracted from the configuration itself. The parsing algorithm is presented in algorithm 1 below:

**Algorithm 1** Greedy transition-based parsing

```
1: **Input:** sentence $s = w_1, \dots, x_w,\ t_1, \dots, t_n$, parameterized function $\text{SCORE}_\theta(\cdot)$ with parameters $\theta$.
2: $c \leftarrow \text{INITIAL}(s)$
3: **while not** $\text{TERMINAL}(c)$ **do**
4:     $\hat{t} \leftarrow \arg\max_{t \in \text{LEGAL}(c)} \text{SCORE}_\theta\big(\phi(c), t\big)$
5:     $c \leftarrow \hat{t}(c)$
6: **return** $tree(c)$
```

Given a sentence $s$, the parser is initialized with the configuration $c$ (line 2). Then, a feature function $\phi(c)$ represents the configuration $c$ as a vector, which is fed to a scoring function SCORE assigning scores to (configuration,transition) pairs. SCORE scores the possible transitions $t$, and the highest scoring transition $\hat{t}$ is chosen (line 4). The transition $\hat{t}$ is applied to the configuration, resulting in a new parser configuration. The process ends when reaching a final configuration, from which the resulting parse tree is read and returned (line 6).

Transition systems differ by the way they define configurations, and by the particular set of transitions available to them. A parser is determined by the choice of a transition system, a feature function $\phi$ and a scoring function SCORE. Our choices are detailed below.

**The Arc-Hybrid System** Many transitions systems exist in the literature. In this work, we use the arc-hybrid transition system (Kuhlmann et al., 2011), which is similar to the more popular arc-standard system (Nivre, 2004), but for which an efficient dynamic oracle is available (Goldberg and Nivre, 2012; Goldberg and Nivre, 2013). In the arc-hybrid system, a configuration $c = (\sigma, \beta, T)$ consists of a stack $\sigma$, a buffer $\beta$, and a set $T$ of dependency arcs. Both the stack and the buffer hold integer indices to sentence elements. Given a sentence $s = w_1, \dots, w_n,\ t_1, \dots, t_n$, the system is initialized with an empty stack, an empty arc set, and

[3]While the BiLSTM computation is quite efficient as it is, if using a GPU the BiLSTM encoding can be performed over many of sentences in parallel, making its computation cost almost negligible.

$\beta = 1, \ldots, n, \texttt{ROOT}$, where ROOT is the special root index. Any configuration $c$ with an empty stack and a buffer containing only ROOT is terminal, and the parse tree is given by the arc set $T_c$ of $c$. Arc-hybrid system allows 3 possible transitions, SHIFT, LEFT$_\ell$ and RIGHT$_\ell$, defined as:

$$\begin{aligned}
\textsc{Shift}[(\sigma,\ b_0|\beta,\ T)] &= (\sigma|b_0,\ \beta,\ T)\\
\textsc{Left}_\ell[(\sigma|s_1|s_0,\ b_0|\beta,\ T)] &= (\sigma,\ b_0|\beta,\ T \cup \{(b_0, s_0, \ell)\})\\
\textsc{Right}_\ell[(\sigma|s_1|s_0,\ \beta,\ T)] &= (\sigma|s_1,\ \beta,\ T \cup \{(s_1, s_0, \ell)\})
\end{aligned}$$

The SHIFT transition moves the first item of the buffer ($b_0$) to the stack. The LEFT$_\ell$ transition removes the first item on top of the stack ($s_0$) and attaches it as a modifier to $b_0$ with label $\ell$, adding the arc $(b_0, s_0, \ell)$. The RIGHT$_\ell$ transition removes $s_0$ from the stack and attaches it as a modifier to the next item on the stack ($s_1$), adding the arc $(s_1, s_0, \ell)$.

**Scoring Function** Traditionally, the scoring function SCORE$_\theta(x, t)$ is a discriminative linear model of the form SCORE$_W(x, t) = (W \cdot x)[t]$. The linearity of SCORE required the feature function $\phi(\cdot)$ to encode non-linearities in the form of combination features. We follow Chen and Manning (2014) and replace the linear scoring model with an MLP.

$$\textsc{Score}_\theta(x, t) = MLP_\theta(x)[t]$$

**Simple Feature Function** The feature function $\phi(c)$ typically complex (see Section 2.1). Our feature function is the concatenated BiLSTM vectors of the top 3 items on the stack and the first item on the buffer. I.e., for a configuration $c = (\ldots|s_2|s_1|s_0,\quad b_0|\ldots,\quad T)$ the feature extractor is defined as:

$$\begin{aligned}
\phi(c) &= v_{s_2} \circ v_{s_1} \circ v_{s_0} \circ v_{b_0}\\
v_i &= \textsc{BiLstm}(x_{1:n}, i)
\end{aligned}$$

This feature function is rather minimal: it takes into account the BiLSTM representations of $s_1, s_0$ and $b_0$, which are the items affected by the possible transitions being scored, as well as one extra stack context $s_2$.[4] Note that, unlike previous work, this feature function *does not* take into account $T$, the already built structure. The high parsing accuracies in the experimental sections suggest that the BiLSTM encoding is capable of estimating a lot of the missing information based on the provided stack and buffer elements and the sequential content between them.

**Extended Feature Function** One of the benefits of the greedy transition-based parsing framework is precisely its ability to look at arbitrary features from the already built tree. If we allow somewhat less minimal feature function, we could add the BiLSTM vectors corresponding to the right-most and left-most modifiers of $s_0$, $s_1$ and $s_2$, as well as the left-most modifier of $b_0$, reaching a total of 11 BiLSTM vectors. We refer to this as the *extended feature set*. As we'll see in Section 6, using the extended set does indeed improve parsing accuracies when using pre-trained word embeddings, but has a minimal effect in the fully-supervised case.

## 4.1 Details of the Training Algorithm

The training objective is to set the score of correct transitions above the scores of incorrect transitions. We use a margin-based objective, aiming to maximize the margin between the highest scoring correct action and the highest scoring incorrect action. The *hinge loss* at each parsing configuration $c$ is defined as:

$$max\Big(0, 1 - \max_{t_o \in G} MLP\big(\phi(c)\big)[t_o] + \max_{t_p \in A \setminus G} MLP\big(\phi(c)\big)[t_p]\Big)$$

where $A$ is the set of possible transitions and $G$ is the set of correct (gold) transitions at the current stage. At each stage of the training process the parser scores the possible transitions $A$, incurs a loss, selects a transition to follow, and moves to the next configuration based on it. The local losses are summed throughout the parsing process of a sentence, and the parameters are updated with respect to the sum of the losses at sentence boundaries.[5]

[4]An additional buffer context is not needed, as $b_1$ is by definition adjecent to $b_0$, a fact that we expect the BiLSTM encoding of $b_0$ to capture. In contrast, $b_0$, $s_0$, $s_1$ and $s_2$ are not necessarily adjacent to each other.

[5]To increase gradient stability and training speed, we simulate mini-batch updates by only updating the parameters when the sum of local losses contains at least 50 non-zero elements. Sums of fewer elements are carried across sentences. This assures us a sufficient number of gradient samples for every update thus minimizing the effect of gradient instability.

The gradients of the entire network (including the MLP and the BiLSTM) with respect to the sum of the losses are calculated using the backpropagation algorithm. As usual, we perform several training iterations over the training corpus, shuffling the order of sentences in each iteration.

**Error-Exploration and Dynamic Oracle Training** We follow Goldberg and Nivre (2013; 2012) in using error exploration training with a dynamic-oracle, which we briefly describe below.

At each stage in the training process, the parser assigns scores to all the possible transitions $t \in A$. It then selects a transition, applies it, and moves to the next step. Which transition should be followed? A common approach follows the highest scoring transition that can lead to the gold tree. However, when training in this way the parser sees only configurations that result from following correct actions, and as a result tends to suffer from error propagation at test time. Instead, in error-exploration training the parser follows the highest scoring action in $A$ during training even if this action is incorrect, exposing it to configurations 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*. We perform error-exploration training using the dynamic-oracle defined in (Goldberg and Nivre, 2013).

**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 follow an aggressive-exploration scheme: we sometimes follow incorrect transitions also if they score below correct transitions. Specifically, when the score of the correct transition is greater than that of the wrong transition but the difference is smaller than a margin constant, we chose to follow the incorrect action with probability $p_{agg}$ (we use $p_{agg} = 0.1$ in our experiments).

**Summary** The greedy transition based parser follows standard techniques from the literature (margin-based objective, dynamic oracle training, error exploration, MLP-based non-linear scoring function). We depart from the literature by replacing the hand-crafted feature function over carefully selected components of the configuration with a concatenation of BiLSTM representations of few prominent items on the stack and the buffer, and training the BiLSTM encoder jointly with the rest of the network.

## 5 Graph-based Parser

Graph-based parsing follows the common structured prediction paradigm (Taskar et al., 2005; McDonald et al., 2005):

$$predict(s) = \arg\max_{y \in \mathcal{Y}(s)} score_{global}(s, y)$$
$$score_{global}(s, y) = \sum_{part \in y} score_{local}(s, part)$$

Given an input sentence $s$ (and the corresponding sequence of vectors $x_{1:n}$) we look for the highest-scoring parse tree $y$ in the space $\mathcal{Y}(s)$ of valid dependency trees over $s$. In order to make the search tractable, the scoring function is decomposed to the sum of local scores for each part independently.

In this work, we focus on arc-factored graph based approach presented in (McDonald et al., 2005). Arc-factored parsing decomposes the score of tree to the sum of the score of its head-modifier arcs $(h, m)$:

$$parse(s) = \arg\max_{y \in \mathcal{Y}(s)} \sum_{(h,m) \in y} score\big(\phi(s, h, m)\big)$$

Given the scores of the arcs the highest scoring projective tree can be efficiently found using Eisner's decoding algorithm (1996). McDonald et al and most subsequent work estimate the local score of an arc by a linear model parameterized by a weight vector $w$, and a feature function $\phi(s, h, m)$ assigning sparse feature vector for an arc linking modifier $m$ to head $h$. We follow Pei et al (2015) and replace the linear scoring function with an MLP.

The feature extractor $\phi(s, h, m)$ is usually complex, involving many elements (see section 2.1). In contrast, our feature extractor uses merely the BiLSTM encoding of the head word and the modifier word:

$$\phi(s, h, m) = \textsc{BiRnn}(x_{1:n}, h) \circ \textsc{BiRnn}(x_{1:n}, m)$$

The final model is:

$$
\begin{aligned}
parse(s) &= \arg\max_{y\in\mathcal{Y}(s)} score_{global}(s,y) \\
&= \arg\max_{y\in\mathcal{Y}(s)} \sum_{(h,m)\in y} score\big(\phi(s,h,m)\big) \\
&= \arg\max_{y\in\mathcal{Y}(s)} \sum_{(h,m)\in y} MLP(v_h \circ v_m) \\
v_i &= \textsc{BiRnn}(x_{1:n}, i)
\end{aligned}
$$

**Training** The training objective is to set the Score function such that correct tree $y$ is scored above incorrect ones. We use a margin-based objective (McDonald et al., 2005; LeCun et al., 2006), aiming to maximize the margin between the score of the gold tree $y$ and highest scoring incorrect tree $y'$. We define a hinge loss with respect to a gold tree $y$ as:

$$
\begin{aligned}
max\Big(0, 1 - \max_{y'\neq y} \sum_{(h,m)\in y'} MLP(v_h \circ v_m) \\
+ \sum_{(h,m)\in y} MLP(v_h \circ v_m)\Big)
\end{aligned}
$$

Each of the tree scores is the calculated by activating the MLP on the arc representations. The entire loss can viewed as the sum of multiple neural networks, which is sub-differentiable. We calculate the gradients of the entire thing (including to the BiLSTM encoder and word embeddings).

**Labeled Parsing** Up to now, we described unlabeled parsing. A possible approach of adding labels is to score the combination of an unlabeled arc $(h, m)$ and its label $\ell$ by considering the label as part of the arc $(h, m, \ell)$. This results in $|Labels| \times |Arcs|$ parts that need to be scored, leading to slow parsing speeds and arguably a harder learning problem.

Instead, we chose to first predict the unlabeled structure using the model given above, and then predict the label of each resulting arc. Using this approach, the number of parts stays small, enabling fast parsing.

The labeling of an arc $(h, m)$ is performed using the same feature representation $\phi(s, h, m)$ fed into a different MLP predictor:

$$
label(h,m) = \arg\max_{\ell\in labels} MLP_{LBL}(v_h \circ v_m)[\ell]
$$

As before we use a margin based hinge loss. The labeler is trained on the gold trees.[6] The BiLSTM encoder responsible for producing $v_h$ and $v_m$ is shared with the arc-factored parser: the same BiLSTM encoder is used in the parer and the labeler. This sharing of parameters can be seen as an instance of multi-task learning (Caruana, 1997). As we show in Section 6, the sharing is effective: training the BiLSTM feature encoder to be good at predicting arc-labels significantly improves the parser's unlabeled accuracy.

**Loss augmented inference** In initial experiments, the network learned quickly and overfit the data. In order to remedy this, we found it useful to use *loss augmented inference* (Taskar et al., 2005). The intuition behind loss augmented inference is to update against trees which have high model scores and are also very wrong. This is done by augmenting the score of each part not belonging to the gold tree by adding a constant to its score. Formally, the loss transforms as follows:

$$
\begin{aligned}
max(0{,}1 + score(x, y) - \\
\max_{y'\neq y} \sum_{part\in y'} (score_{local}(x, part) + I_{part\notin y}))
\end{aligned}
$$

**Speed improvements** The arc-factored model requires the scoring of $n^2$ arcs. Scoring is performed using an MLP with one hidden layer, resulting in $n^2$ matrix-vector multiplications from the input to the hidden layer, and $n^2$ multiplications from the hidden to the output layer. The first $n^2$ multiplications involve larger dimensional input and output vectors, and are the most time consuming. Fortunately, these can be reduced to $2n$ multiplications and $n^2$ vector additions, by observing that the multiplication $W \cdot (v_h \circ v_m)$ can be written as $W^1 \cdot v_h + W^2 \cdot v_m$ where $W^1$ and $W^1$ are are the first and second half of the matrix $W$ and reusing the products across different pairs.

**Summary** The graph-based parser is straightforward first-order parser, trained with a margin-based hinge-loss and loss-augmented inference. We depart from the literature by replacing the hand-crafted feature function with a concatenation of

[6]When training the labeled parser, we calculate the structure loss and the labeling loss for each training sentence, and sum the losses prior to computing the gradients.

BiLSTM representations of the head and modifier words, and training the BiLSTM encoder jointly with the structured objective. We also introduce a novel MTL-based approach for labeled parsing by training a second-stage arc-labeler sharing the same BiLSTM encoder with the unlabeled parser.

## 6 Experiments and Results

We evaluated our parsing model on English and Chinese data. For comparison purposes we follow 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).[7]

**Implementation Details** The parsers are implemented in python, using the PyCNN toolkit[8] for neural network training. The code will be made available on the first author's website. We use the LSTM variant implemented in PyCNN, and optimize using the Adam optimizer (Kingma and Ba, 2014). Unless otherwise noted, we use the default values provided by PyCNN (e.g. for random initialization, learning rates etc).

The word and POS embeddings $e(w_i)$ and $e(p_i)$ are initialized to random values and trained together with the rest of parsers' networks. In some experiments, we introduce also pre-trained word embeddings. In those cases, the vector representation of a word is a concatenation of its randomly-initialized vector embedding with its pre-trained word vector. Both are tuned during training. We use the same word vectors as in Dyer et al (2015).

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

[8]`http://www.github.com/clab/cnn/pycnn`

During training, we employ a variant of *word dropout* (Iyyer et al., 2015), and replace a word with the unknown-word symbol with probability that is inversely proportional to frequency of the word. A word $w$ appearing $\#(w)$ times in the training corpus is replaced with the unknown symbol with a probability $p_{unk}(w) = \frac{\alpha}{\#(w)+\alpha}$. If a word was dropped the external embedding of the word is also dropped with probability of half.

We train the parsers for up to 30 iterations, and choose the best model according to the UAS accuracy on the development set.

**Hyperparameter Tuning** We performed a very minimal hyper-parameter search with the graph-based parser, and use the same hyper-parameters for both parsers. 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 |
| Hidden units in $MLP$ | 100 |
| Hidden units in $MLP_{LBL}$ | 100 |
| BI-LSTM Layers | 2 |
| BI-LSTM Dimensions (hidden/output) | 125 / 125 |
| $\alpha$ (for word dropout) | 0.25 |
| $p_{aug}$ (for exploration training) | 0.1 |

Table 1: Hyper-parameter values used in experiments

**Main Results** Table 2 lists the test-set accuracies of our best parsing models, compared to other state-of-the art parsers from the literature.[9]

It is clear that our parsers are very competitive, despite using very simple parsing architectures and minimal feature extractors. When not using external embeddings, the first-order graph-based parser with 2 features outperforms all other systems that are not using external resources, including the third-order TurboParser. The greedy transition based parser with 4 features also matches or outperforms most other parsers, including the beam-based transition parser with heavily engineered features of Zhang and Nivre (2011) and the Stack-LSTM parser of

[9]Unfortunately, many papers still report English parsing results on the deficient Yamada and Matsumoto head rules (PTB-YM) rather than the more modern Stanford-dependencies (PTB-SD). We note that the PTB-YM and PTB-SD results are not strictly comparable, and in our experience the PTB-YM results are usually about half a UAS point higher.

| System | Method | Representation | Emb | PTB-YM | PTB-SD | | CTB | |
|---|---|---|---|---|---|---|---|---|
| | | | | UAS | UAS | LAS | UAS | LAS |
| This work | graph, 1st order | 2 BiLSTM vectors | – | – | **93.2** | **91.0** | **86.4** | **84.8** |
| This work | transition (greedy) | 4 BiLSTM vectors | – | – | 92.8 | 90.7 | 86.0 | 84.3 |
| This work | transition (greedy) | 11 BiLSTM vectors | – | – | 92.9 | 90.7 | 85.6 | 83.9 |
| ZhangNivre11 | transition (beam) | large feature set (sparse) | – | 92.9 | – | – | 86.0 | 84.4 |
| Martins13 (TurboParser) | graph, 3rd order+ | large feature set (sparse) | – | 92.8 | 93.1 | – | – | – |
| Pei15 | graph, 2nd order | large feature set (dense) | – | 93.0 | – | – | – | – |
| Dyer15 | transition (greedy) | Stack-LSTM + composition | – | – | 92.4 | 90.0 | 85.7 | 84.1 |
| This work | graph, 1st order | 2 BiLSTM vectors | YES | – | 92.7 | 90.5 | 86.1 | 84.5 |
| This work | transition (greedy) | 4 BiLSTM vectors | YES | – | 93.3 | 91.1 | 86.6 | 85.0 |
| This work | transition (greedy) | 11 BiLSTM vectors | YES | – | 93.6 | 91.5 | **87.3** | **85.7** |
| Weiss15 | transition (greedy) | large feature set (dense) | YES | – | 93.2 | 91.2 | – | – |
| Weiss15 | transition (beam) | large feature set (dense) | YES | – | **94.0** | **92.0** | – | – |
| Pei15 | graph, 2nd order | large feature set (dense) | YES | 93.3 | – | – | – | – |
| Dyer15 | transition (greedy) | Stack-LSTM + composition | YES | – | 93.1 | 90.9 | 87.1 | 85.5 |
| LeZuidema14 | reranking /blend | inside-outside recursive net | YES | 93.1 | 93.8 | 91.5 | – | – |
| Zhu15 | reranking /blend | recursive conv-net | YES | 93.8 | – | – | 85.7 | – |

Table 2: Test-set parsing results of various state-of-the-art parsing systems on the English (PTB) and Chinese (CTB) datasets. The systems that use embeddings may 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 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); Dyer15 (Dyer et al., 2015); LeZuidema14 (Le and Zuidema, 2014); Zhu15: (Zhu et al., 2015).

Dyer et al (2015). Moving from the simple (4 features) to the extended (11 features) feature set leads to very small improvements for English and small accuracy losses in Chinese.

Interestingly, when adding external word embeddings the accuracy of the graph-based parser *degrades*. We are not sure why this happens, and leave the exploration of effective semi-supervised parsing with the graph-based model for future work. The greedy parser does manage to benefit from the external embeddings, and with using them we also see gains from moving from the simple to the extended feature set. Both feature sets result in very competitive results, with the extended feature set yielding the best reported results for Chinese, and the is ranked third for English, after the heavily-tuned beam-based parser of Weiss et al (2015) and the much more elaborate reranker-based neural parser of Le and Zuidema (2014).

**Additional Results** We perform some ablation experiments in order to quantify the effect of the different components on our best models (Table 3).

| | PTB | | CTB | |
|---|---|---|---|---|
| | UAS | LAS | UAS | LAS |
| Graph (no ext. emb) | 93.2 | 90.8 | 86.7 | 85.0 |
| –ArcLabeler | 92.3 | – | 85.5 | – |
| –Loss Aug. | 81.8 | 79.8 | 56.3 | 55.3 |
| Greedy (ext. emb) | 93.5 | 91.1 | 87.2 | 85.7 |
| –DynOracle | 93.2 | 91.0 | 87.3 | 85.8 |

Table 3: Ablation experiments results (dev set) for the graph-based parser without external embeddings and the greedy parser with external embeddings and extended feature set.

Loss augmented inference is crucial for the success of the graph-based parser, and the MTL arc-labeler contributes nicely to the *unlabeled* scores. Dynamic-oracle training has mixed results on the greedy parser, yielding small gains for English and small drops for Chinese.

## 7 Conclusion

We presented a frustratingly effective approach for feature extraction for dependency parsing based on a BiLSTM encoder that is trained jointly with the parer, and demonstrated its effectiveness by integrating it into two simple parsing models: a greedy transition based parser and a globally optimized first-order graph-based parser, yielding very competitive parsing accuracies in both cases.

**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

Rich Caruana. 1997. Multitask learning. *Mach. Learn.*, 28(1):41–75, July.

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.

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

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.

Jason Eisner. 1996. Three new probabilistic models for dependency parsing: An exploration. In *Proc. of COLING*.

Jeffrey L. Elman. 1990. Finding Structure in Time. *Cognitive Science*, 14(2):179–211, March.

Yoav Goldberg and Joakim Nivre. 2012. A dynamic oracle for the arc-eager system. In *Proc. of COLING 2012*.

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

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

A. Graves. 2008. *Supervised sequence labelling with recurrent neural networks*. Ph.D. thesis, Technische Universität München.

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

Liang Huang and Kenji Sagae. 2010. Dynamic programming for linear-time incremental parsing. In *Proc. of ACL*, July.

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.

Andrej Karpathy, Justin Johnson, and Fei-Fei Li. 2015. Visualizing and Understanding Recurrent Networks. *arXiv:1506.02078 [cs]*, June.

Diederik Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*, December.

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

Terry Koo and Michael Collins. 2010. Efficient third-order dependency parsers. In *Proc. of ACL*.

Terry Koo, Xavier Carreras, and Michael Collins. 2008. Simple semi-supervised dependency parsing. In *Proc. of ACL*.

Sandra Kübler, Ryan McDonald, and Joakim Nivre. 2008. Dependency Parsing. *Synthesis Lectures on Human Language Technologies*, 2(1):1–127, December.

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

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.

Yann LeCun, Sumit Chopra, Raia Hadsell, M. Ranzato, and F. Huang. 2006. A tutorial on energy-based learning. *Predicting structured data*, 1:0.

Tao Lei, Yu Xin, Yuan Zhang, Regina Barzilay, and Tommi Jaakkola. 2014. Low-rank tensors for scoring dependency structures. In *Proceedings of the*

*52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1381–1391, Baltimore, Maryland, June. Association for Computational Linguistics.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marchinkiewicz. 1993. Building a large annotated corpus of English: The penn Treebank. *Computational Linguistics*, 19.

Andre Martins, Noah Smith, and Eric Xing. 2009. Concise integer linear programming formulations for dependency parsing. In *Proc. ACL/AFNLP*.

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, Koby Crammer, and Fernando Pereira. 2005. Online large-margin training of dependency parsers. In *Proc. of ACL*.

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

Joakim Nivre. 2004. Incrementality in deterministic dependency parsing. In *Incremental Parsing: Bringing Engineering and Cognition Together, ACL-Workshop*.

Joakim Nivre. 2008. Algorithms for Deterministic Incremental Dependency Parsing. *Computational Linguistics*, 34(4):513–553, December.

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.

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

Ben Taskar, Vassil Chatalbashev, Daphne Koller, and Carlos Guestrin. 2005. Learning structured prediction models: a large margin approach. In *Proceedings of the 22nd international conference on Machine learning*, ICML '05, pages 896–903, New York, NY, USA. ACM.

Hillel Taub-Tabib, Yoav Goldberg, and Amir Globerson. 2015. Template kernels for dependency parsing. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1422–1427, Denver, Colorado, May–June. 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 using beam-search. In *Proc. of EMNLP*.

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.

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.