A Fast and Accurate Dependency Parser using Neural Networks

Danqi Chen
Computer Science Department
Stanford University
danqi@cs.stanford.edu

Christopher D. Manning
Computer Science Department
Stanford University
manning@stanford.edu

Abstract

Almost all current dependency parsers classify based on millions of sparse indicator features. Not only do these features generalize poorly, but the cost of feature computation restricts parsing speed significantly. In this work, we propose a novel way of learning a neural network classifier for use in a greedy, transition-based dependency parser. Because this classifier learns and uses just a small number of dense features, it can work very fast, while achieving an about 2% improvement in unlabeled and labeled attachment scores on both English and Chinese datasets. Concretely, our parser is able to parse more than 1000 sentences per second at 92.2% unlabeled attachment score on the English Penn Treebank.

1 Introduction

In recent years, enormous parsing success has been achieved by the use of feature-based discriminative dependency parsers (Kübler et al., 2009). In particular, for practical applications, the speed of the subclass of transition-based dependency parsers has been very appealing.

However, these parsers are not perfect. First, from a statistical perspective, these parsers suffer from the use of millions of mainly poorly estimated feature weights. While in aggregate both lexicalized features and higher-order interaction term features are very important in improving the performance of these systems, nevertheless, there is insufficient data to correctly weight most such features. For this reason, techniques for introducing higher-support features such as word class features have also been very successful in improving parsing performance (Koo et al., 2008). Second, almost all existing parsers rely on a manually designed set of feature templates, which require a lot of expertise and are usually incomplete. Third, the use of many feature templates cause a less studied problem: in modern dependency parsers, most of the runtime is consumed not by the core parsing algorithm but in the feature extraction step (He et al., 2013). For instance, Bohnet (2010) reports that his baseline parser spends 99% of its time doing feature extraction, despite that being done in standard efficient ways.

In this work, we address all of these problems by using dense features in place of the sparse indicator features. This is inspired by the recent success of distributed word representations in many NLP tasks, e.g., POS tagging (Collobert et al., 2011), machine translation (Devlin et al., 2014), and constituency parsing (Socher et al., 2013). Low-dimensional, dense word embeddings can effectively alleviate sparsity by sharing statistical strength between similar words, and can provide us a good starting point to construct features of words and their interactions.

Nevertheless, there remain challenging problems of how to encode all the available information from the configuration and how to model higher-order features based on the dense representations. In this paper, we train a neural network classifier to make parsing decisions within a transition-based dependency parser. The neural network learns compact dense vector representations of words, part-of-speech (POS) tags, and dependency labels. This results in a fast, compact classifier, which uses only 200 learned dense features while yielding good gains in parsing accuracy and speed on two languages (English and Chinese) and two different dependency representations (CoNLL and Stanford dependencies). The main contributions of this work are: (i) showing the usefulness of dense representations that are learned within the parsing task, (ii) developing a neural network architecture that gives good accuracy and speed, and (iii) introducing a novel acti-

vation function for the neural network that better captures higher-order interaction features.

2 Transition-based Dependency Parsing

Transition-based dependency parsing aims to predict a transition sequence from an initial configuration to some terminal configuration, which derives a target dependency parse tree, as shown in Figure 1. In this paper, we examine only greedy parsing, which uses a classifier to predict the correct transition based on features extracted from the configuration. This class of parsers is of great interest because of their efficiency, although they tend to perform slightly worse than the search-based parsers because of subsequent error propagation. However, our greedy parser can achieve comparable accuracy with a very good speed.[1]

As the basis of our parser, we employ the **arc-standard** system (Nivre, 2004), one of the most popular transition systems. In the arc-standard system, a *configuration* $c = (s, b, A)$ consists of a *stack* s, a *buffer* b, and a set of *dependency arcs* A. The initial configuration for a sentence $w_1, \ldots, w_n$ is $s = [\texttt{ROOT}], b = [w_1, \ldots, w_n], A = \emptyset$. A configuration c is terminal if the buffer is empty and the stack contains the single node `ROOT`, and the parse tree is given by A_c. Denoting s_i $(i = 1, 2, \ldots)$ as the i^{th} top element on the stack, and b_i $(i = 1, 2, \ldots)$ as the i^{th} element on the buffer, the arc-standard system defines three types of transitions:

- `LEFT-ARC`(l): adds an arc $s_1 \rightarrow s_2$ with label l and removes s_2 from the stack. Precondition: $|s| \geq 2$.
- `RIGHT-ARC`(l): adds an arc $s_2 \rightarrow s_1$ with label l and removes s_1 from the stack. Precondition: $|s| \geq 2$.
- `SHIFT`: moves b_1 from the buffer to the stack. Precondition: $|b| \geq 1$.

In the labeled version of parsing, there are in total $|\mathcal{T}| = 2N_l + 1$ transitions, where N_l is number of different arc labels. Figure 1 illustrates an example of one transition sequence from the initial configuration to a terminal one.

The essential goal of a greedy parser is to predict a correct transition from $\mathcal{T}$, based on one given configuration. Information that can be obtained from one configuration includes: (1) all the words and their corresponding POS tags (e.g., `has` / `VBZ`); (2) the head of a word and its label (e.g., `nsubj`, `dobj`) if applicable; (3) the position of a word on the stack/buffer or whether it has already been removed from the stack.

Single-word features (9)	
$s_1.w$; $s_1.t$; $s_1.wt$; $s_2.w$; $s_2.t$;	
$s_2.wt$; $b_1.w$; $b_1.t$; $b_1.wt$	
Word-pair features (8)	
$s_1.wt \circ s_2.wt$; $s_1.wt \circ s_2.w$; $s_1.wts_2.t$;	
$s_1.w \circ s_2.wt$; $s_1.t \circ s_2.wt$; $s_1.w \circ s_2.w$	
$s_1.t \circ s_2.t$; $s_1.t \circ b_1.t$	
Three-word feaures (8)	
$s_2.t \circ s_1.t \circ b_1.t$; $s_2.t \circ s_1.t \circ lc_1(s_1).t$;	
$s_2.t \circ s_1.t \circ rc_1(s_1).t$; $s_2.t \circ s_1.t \circ lc_1(s_2).t$;	
$s_2.t \circ s_1.t \circ rc_1(s_2).t$; $s_2.t \circ s_1.w \circ rc_1(s_2).t$;	
$s_2.t \circ s_1.w \circ lc_1(s_1).t$; $s_2.t \circ s_1.w \circ b_1.t$	

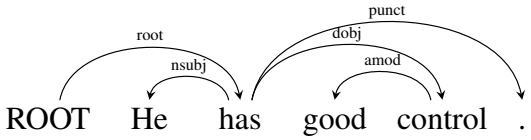
Table 1: The feature templates used for analysis. $lc_1(s_i)$ and $rc_1(s_i)$ denote the leftmost and rightmost children of s_i, w denotes word, t denotes POS tag.

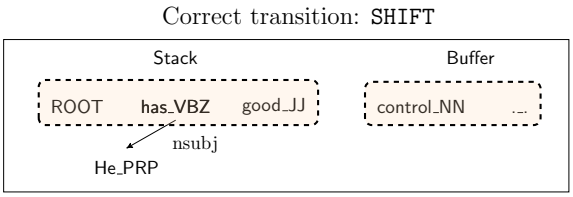
Conventional approaches extract indicator features such as the conjunction of $1 \sim 3$ elements from the stack/buffer using their words, POS tags or arc labels. Table 1 lists a typical set of feature templates chosen from the ones of (Huang et al., 2009; Zhang and Nivre, 2011).[2] These features suffer from the following problems:

- **Sparsity**. The features, especially lexicalized features are highly sparse, and this is a common problem in many NLP tasks. The situation is severe in dependency parsing, because it depends critically on word-to-word interactions and thus the high-order features. To give a better understanding, we perform a feature analysis using the features in Table 1 on the English Penn Treebank (CoNLL representations). The results given in Table 2 demonstrate that: (1) lexicalized features are indispensable; (2) Not only are the word-pair features (especially s_1 and s_2) vital for predictions, the three-word conjunctions (e.g., $\{s_2, s_1, b_1\}$, $\{s_2, lc_1(s_1), s_1\}$) are also very important.

[1] Additionally, our parser can be naturally incorporated with beam search, but we leave this to future work.

[2] We exclude sophisticated features using labels, distance, valency and third-order features in this analysis, but we will include all of them in the final evaluation.





Transition	Stack	Buffer	A
	[ROOT]	[He has good control .]	$\emptyset$
SHIFT	[ROOT He]	[has good control .]	
SHIFT	[ROOT He has]	[good control .]	
LEFT-ARC(nsubj)	[ROOT has]	[good control .]	$A \cup$ nsubj(has,He)
SHIFT	[ROOT has good]	[control .]	
SHIFT	[ROOT has good control]	[.]	
LEFT-ARC(amod)	[ROOT has control]	[.]	$A \cup$amod(control,good)
RIGHT-ARC(dobj)	[ROOT has]	[.]	$A \cup$ dobj(has,control)
...	...	...	...
RIGHT-ARC(root)	[ROOT]	[]	$A \cup$ root(ROOT,has)

Figure 1: An example of transition-based dependency parsing. Above left: a desired dependency tree, above right: an intermediate configuration, bottom: a transition sequence of the arc-standard system.

Features	UAS
All features in Table 1	88.0
single-word & word-pair features	82.7
only single-word features	76.9
excluding all lexicalized features	81.5

Table 2: Performance of different feature sets. UAS: unlabeled attachment score.

- **Incompleteness**. Incompleteness is an unavoidable issue in all existing feature templates. Because even with expertise and manual handling involved, they still do not include the conjunction of every useful word combination. For example, the conjunction of s_1 and b_2 is omitted in almost all commonly used feature templates, however it could indicate that we cannot perform a RIGHT-ARC action if there is an arc from s_1 to b_2.
- **Expensive feature computation**. The feature generation of indicator features is generally expensive — we have to concatenate some words, POS tags, or arc labels for generating feature strings, and look them up in a huge table containing several millions of features. In our experiments, more than 95% of the time is consumed by feature computation during the parsing process.

So far, we have discussed preliminaries of transition-based dependency parsing and existing problems of sparse indicator features. In the following sections, we will elaborate our neural network model for learning dense features along with experimental evaluations that prove its efficiency.

3 Neural Network Based Parser

In this section, we first present our neural network model and its main components. Later, we give details of training and speedup of parsing process.

3.1 Model

Figure 2 describes our neural network architecture. First, as usual word embeddings, we represent each word as a d-dimensional vector $e_i^w \in \mathbb{R}^d$ and the full embedding matrix is $E^w \in \mathbb{R}^{d \times N_w}$ where N_w is the dictionary size. Meanwhile, we also map POS tags and arc labels to a d-dimensional vector space, where $e_i^t, e_j^l \in \mathbb{R}^d$ are the representations of i^{th} POS tag and j^{th} arc label. Correspondingly, the POS and label embedding matrices are $E^t \in \mathbb{R}^{d \times N_t}$ and $E^l \in \mathbb{R}^{d \times N_l}$ where N_t and N_l are the number of distinct POS tags and arc labels.

We choose a set of elements based on the stack / buffer positions for each type of information (word, POS or label), which might be useful for our predictions. We denote the sets as S^w, S^t, S^l respectively. For example, given the configuration in Figure 2 and $S^t =$

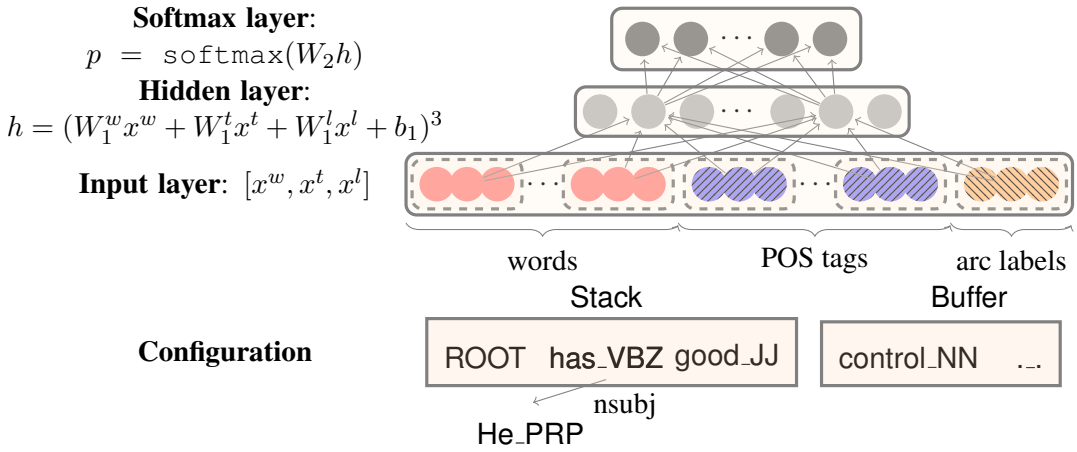


Figure 2: Our neural network architecture.

$\{lc_1(s_2).t, s_2.t, rc_1(s_2).t, s_1.t\}$, we will extract PRP, VBZ, NULL, JJ in order. Here we use a special token NULL to represent a non-existent element.

We build a standard neural network with one hidden layer, where the corresponding embeddings of our chosen elements from S^w, S^t, S^l will be added to the input layer. Denoting n_w, n_t, n_l as the number of chosen elements of each type, we add $x^w = [e^w_{w_1}; e^w_{w_2}; \dots e^w_{w_{n_w}}]$ to the input layer, where $S^w = \{w_1, \dots, w_{n_w}\}$. Similarly, we add the POS tag features x^t and arc label features x^l to the input layer.

We map the input layer to a hidden layer with d_h nodes through a **cube activation function**:

$$h = (W_1^w x^w + W_1^t x^t + W_1^l x^l + b_1)^3$$

where $W_1^w \in \mathbb{R}^{d_h \times (d \cdot n_w)}$, $W_1^t \in \mathbb{R}^{d_h \times (d \cdot n_t)}$, $W_1^l \in \mathbb{R}^{d_h \times (d \cdot n_l)}$, and $b_1 \in \mathbb{R}^{d_h}$ is the bias.

A softmax layer is finally added on the top of the hidden layer for modeling multi-class probabilities $p = \texttt{softmax}(W_2 h)$, where $W_2 \in \mathbb{R}^{|\mathcal{T}| \times d_h}$.

POS and label embeddings

To our best knowledge, this is the first attempt to introduce POS tag and arc label embeddings instead of discrete representations.

Although the POS tags $\mathcal{P}$ = {NN, NNP, NNS, DT, JJ, ...} (for English) and arc labels $\mathcal{L}$ = {amod, tmod, nsubj, csubj, dobj, ...} (for Stanford Dependencies on English) are relatively small discrete sets, they still exhibit many semantical similarities like words. For example, NN (singular noun) should be closer to NNS (plural noun) than DT (determiner), and amod (adjective modifier) should be closer to num (numeric modifier) than nsubj (nominal subject). We expect these semantic meanings to be effectively captured by the dense representations.

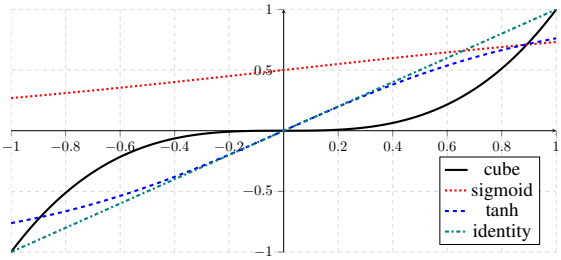


Figure 3: Different activation functions used in neural networks.

Cube activation function

As stated above, we introduce a novel activation function: cube $g(x) = x^3$ in our model instead of the commonly used tanh or sigmoid functions (Figure 3).

Intuitively, every hidden unit is computed by a (non-linear) mapping on a weighted sum of input units plus a bias. Using $g(x) = x^3$ can model the product terms of $x_i x_j x_k$ for any three different elements at the input layer directly:

$$g(w_1 x_1 + \dots + w_m x_m + b) = \sum_{i,j,k} (w_i w_j w_k) x_i x_j x_k + \sum_{i,j} b(w_i w_j) x_i x_j \dots$$

In our case, x_i, x_j, x_k could come from different dimensions of three embeddings. We believe that this better captures the interaction of three ele-

ments, which is a very desired property of dependency parsing.

Experimental results also verify the success of the cube activation function empirically (see more comparisons in Section 4). However, the expressive power of this activation function is still open to investigate theoretically.

The choice of S^w, S^t, S^l

Following (Zhang and Nivre, 2011), we pick a rich set of elements for our final parser. In detail, S^w contains $n_w = 18$ elements: (1) The top 3 words on the stack and buffer: $s_1, s_2, s_3, b_1, b_2, b_3$; (2) The first and second leftmost / rightmost children of the top two words on the stack: $lc_1(s_i), rc_1(s_i), lc_2(s_i), rc_2(s_i),\ i = 1, 2$. (3) The leftmost of leftmost / rightmost of rightmost children of the top two words on the stack: $lc_1(lc_1(s_i)), rc_1(rc_1(s_i)), i = 1, 2$.

We use the corresponding POS tags for S^t ($n_t = 18$), and the corresponding arc labels of words excluding those 6 words on the stack/buffer for S_l ($n_l = 12$). A good advantage of our parser is that we can add a rich set of elements cheaply, instead of hand-crafting many more indicator features.

3.2 Training

We first generate training examples $\{(c_i, t_i)\}_{i=1}^m$ from the training sentences and their gold parse trees using a "shortest stack" oracle which always prefers $\texttt{LEFT-ARC}_l$ over $\texttt{SHIFT}$, where c_i is a configuration, $t_i \in \mathcal{T}$ is the oracle transition.

The final training objective is to minimize the cross-entropy loss, plus a l_2-regularization term:

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

where θ is the set of all parameters $\{W_1^w, W_1^t, W_1^l, b_1, W_2, E^w, E^t, E^l\}$. A slight variation is that we compute the softmax probabilities only among the feasible transitions in practice.

For initialization of parameters, we use pre-trained word embeddings to initialize E^w and use random initialization within $(-0.01, 0.01)$ for E^t and E^l. Concretely, we use the pre-trained word embeddings from (Collobert et al., 2011) for English (#dictionary = 130,000, coverage = 72.7%), and our trained 50-dimensional `word2vec` embeddings (Mikolov et al., 2013) on Wikipedia and Gigaword corpus for Chinese (#dictionary = 285,791, coverage = 79.0%). We will also compare with random initialization of E^w in Section 4. The training error derivatives will be back-propagated to these embeddings during the training process.

We use mini-batched AdaGrad (Duchi et al., 2011) for optimization and also apply a dropout (Hinton et al., 2012) with 0.5 rate. The parameters which achieve the best unlabeled attachment score on the development set will be chosen for final evaluation.

3.3 Parsing

We perform greedy decoding in parsing. At each step, we extract all the corresponding word, POS and label embeddings from the current configuration c, compute the hidden layer $h(c) \in \mathbb{R}^{d_h}$, and pick the transition with the highest score: $t = \arg\max_{t \text{ is feasible}} W_2(t, \cdot)h(c)$, and then execute $c \rightarrow t(c)$.

Comparing with indicator features, our parser does not need to compute conjunction features and look them up in a huge feature table, and thus greatly reduces feature generation time. Instead, it involves many matrix addition and multiplication operations. To further speed up the parsing time, we apply a **pre-computation** trick, similar to (Devlin et al., 2014). For each position chosen from S^w, we pre-compute matrix multiplications for most top frequent $10,000$ words. Thus, computing the hidden layer only requires looking up the table for these frequent words, and adding the d_h-dimensional vector. Similarly, we also pre-compute matrix computations for all positions and all POS tags and arc labels. We only use this optimization in the neural network parser, but it is only feasible for a parser like the neural network parser which uses a small number of features. In practice, this pre-computation step increases the speed of our parser $8 \sim 10$ times.

4 Experiments

4.1 Datasets

We conduct our experiments on the English Penn Treebank (PTB) and the Chinese Penn Treebank (CTB) datasets.

For English, we follow the standard splits of `PTB3`, using sections 2-21 for training, section 22 as development set and 23 as test set. We adopt two different dependency representations: CoNLL Syntactic Dependencies (CD) (Johansson

Dataset	#Train	#Dev	#Test	#words (N_w)	#POS (N_t)	#labels (N_l)	projective (%)
PTB: CD	39,832	1,700	2,416	44,352	45	17	99.4
PTB: SD	39,832	1,700	2,416	44,389	45	45	99.9
CTB	16,091	803	1,910	34,577	35	12	100.0

Table 3: Data Statistics. "Projective" is the percentage of projective trees on the training set.

and Nugues, 2007) using the LTH Constituent-to-Dependency Conversion Tool[3] and Stanford Basic Dependencies (SD) (de Marneffe et al., 2006) using the Stanford parser v3.3.0.[4] The POS tags are assigned using Stanford POS tagger (Toutanova et al., 2003) with ten-way jackknifing of the training data (accuracy $\approx 97.3\%$).

For Chinese, we adopt the same split of `CTB5` as described in (Zhang and Clark, 2008). Dependencies are converted using the Penn2Malt tool[5] with the head-finding rules of (Zhang and Clark, 2008). And following (Zhang and Clark, 2008; Zhang and Nivre, 2011), we use gold segmentation and POS tags for the input.

Table 3 gives statistics of the three datasets.[6] In particular, over 99% of the trees are projective in all datasets.

4.2 Results

The following hyper-parameters are used in all experiments: embedding size $d = 50$, hidden layer size $h = 200$, regularization parameter $\lambda = 10^{-8}$, initial learning rate of Adagrad $\alpha = 0.01$.

To situate the performance of our parser, we first make a comparison with our own implementation of greedy arc-eager and arc-standard parsers. These parsers are trained with structured averaged perceptron using the "early-update" strategy. The feature templates of (Zhang and Nivre, 2011) are used for the arc-eager system, and they are also adapted to the arc-standard system.[7]

Furthermore, we also compare our parser with two popular, off-the-shelf parsers: MaltParser — a greedy transition-based dependency parser (Nivre et al., 2006),[8] and MSTParser — a first-order graph-based parser (McDonald and Pereira, 2006).[9] In this comparison, for MaltParser, we select `stackproj` (arc-standard) and `nivreeager` (arc-eager) as parsing algorithms, and liblinear (Fan et al., 2008) for optimization.[10] For MSTParser, we use default options.

On all datasets, we report unlabeled attachment scores (UAS) and labeled attachment scores (LAS) and punctuation is excluded in all evaluation metrics.[11] Our parser and the baseline arc-standard and arc-eager parsers are all implemented in Java. The parsing speeds are measured on an Intel Core i7 2.7GHz CPU with 16GB RAM and the runtime does not include pre-computation or parameter loading time.

Table 4, Table 5 and Table 6 show the comparison of accuracy and parsing speed on PTB (CoNLL dependencies), PTB (Stanford dependencies) and CTB respectively.

Parser	Dev		Test		Speed
	UAS	LAS	UAS	LAS	(sent/s)
standard	89.9	88.7	89.7	88.3	51
eager	90.3	89.2	89.9	88.6	63
Malt:sp	90.0	88.8	89.9	88.5	560
Malt:eager	90.1	88.9	90.1	88.7	535
MSTParser	92.1	90.8	**92.0**	90.5	12
Our parser	**92.2**	**91.0**	**92.0**	**90.7**	**1013**

Table 4: Accuracy and parsing speed on PTB + CoNLL dependencies.

Clearly, our parser is superior in terms of both accuracy and speed. Comparing with the baselines of arc-eager and arc-standard parsers, our parser achieves around 2% improvement in UAS and LAS on all datasets, while running about 20 times faster.

It is worth noting that the efficiency of our

[3] http://nlp.cs.lth.se/software/treebank_converter/

[4] http://nlp.stanford.edu/software/lex-parser.shtml

[5] http://stp.lingfil.uu.se/ nivre/research/Penn2Malt.html

[6] Pennconverter and Stanford dependencies generate slightly different tokenization, e.g., Pennconverter splits the token `WCRS\/Boston_NNP` into three tokens `WCRS_NNP / _CC Boston_NNP`.

[7] Since arc-standard is bottom-up, we remove all features using the head of stack elements, and also add the right child features of the first stack element.

[8] http://www.maltparser.org/

[9] http://www.seas.upenn.edu/ strctlrn/MSTParser/MSTParser.html

[10] We do not compare with libsvm optimization, which is known to be sightly more accurate, but orders of magnitude slower (Kong and Smith, 2014).

[11] A token is a punctuation if its gold POS tag is {" " : , .} for English and `PU` for Chinese.

Parser	Dev		Test		Speed
	UAS	LAS	UAS	LAS	(sent/s)
standard	90.2	87.8	89.4	87.3	26
eager	89.8	87.4	89.6	87.4	34
Malt:sp	89.8	87.2	89.3	86.9	469
Malt:eager	89.6	86.9	89.4	86.8	448
MSTParser	91.4	88.1	90.7	87.6	10
Our parser	**92.0**	**89.7**	**91.8**	**89.6**	**654**

Table 5: Accuracy and parsing speed on PTB + Stanford dependencies.

Parser	Dev		Test		Speed
	UAS	LAS	UAS	LAS	(sent/s)
standard	82.4	80.9	82.7	81.2	72
eager	81.1	79.7	80.3	78.7	80
Malt:sp	82.4	80.5	82.4	80.6	420
Malt:eager	81.2	79.3	80.2	78.4	393
MSTParser	**84.0**	82.1	83.0	81.2	6
Our parser	**84.0**	**82.4**	**83.9**	**82.4**	**936**

Table 6: Accuracy and parsing speed on CTB.

parser even surpasses MaltParser using liblinear, which is known to be highly optimized, while our parser achieves much better accuracy.

Also, despite the fact that the graph-based MSTParser achieves a similar result to ours on PTB (CoNLL dependencies), our parser is nearly 100 times faster. In particular, our transition-based parser has a great advantage in LAS, especially for the fine-grained label set of Stanford dependencies.

4.3 Effects of Parser Components

Herein, we examine components that account for the performance of our parser.

Cube activation function

We compare our `cube` activation function (x^3) with two widely used non-linear functions: `tanh` ($\frac{e^x-e^{-x}}{e^x+e^{-x}}$), `sigmoid` ($\frac{1}{1+e^{-x}}$), and also the `identity` function (x), as shown in Figure 4 (left).

In short, `cube` outperforms all other activation functions significantly and `identity` works the worst. Concretely, `cube` can achieve $0.8\% \sim 1.2\%$ improvement in UAS over `tanh` and other functions, thus verifying the effectiveness of the cube activation function empirically.

Initialization of pre-trained word embeddings

We further analyze the influence of using pre-trained word embeddings for initialization. Figure 4 (middle) shows that using pre-trained word embeddings can obtain around 0.7% improvement on PTB and 1.7% improvement on CTB, compared with using random initialization within $(-0.01, 0.01)$. On the one hand, the pre-trained word embeddings of Chinese appear more useful than those of English; on the other hand, our model is still able to achieve comparable accuracy without the help of pre-trained word embeddings.

POS tag and arc label embeddings

As shown in Figure 4 (right), POS embeddings yield around 1.7% improvement on PTB and nearly 10% improvement on CTB and the label embeddings yield a much smaller 0.3% and 1.4% improvement respectively.

However, we can obtain little gain from label embeddings when the POS embeddings are present. This may be because the POS tags of two tokens already capture most of the label information between them.

4.4 Model Analysis

Last but not least, we will examine the parameters we have learned, and hope to investigate what these dense features capture. We use the weights learned from the English Penn Treebank using Stanford dependencies for analysis.

What do E^t, E^l capture?

We first introduced E^t and E^l as the dense representations of all POS tags and arc labels, and we wonder whether these embeddings could carry some semantic information.

Figure 5 presents t-SNE visualizations (van der Maaten and Hinton, 2008) of these embeddings. It clearly shows that these embeddings effectively exhibit the similarities between POS tags or arc labels. For instance, the three adjective POS tags `JJ`, `JJR`, `JJS` have very close embeddings, and also the three labels representing clausal complements `acomp`, `ccomp`, `xcomp` are grouped together.

Since these embeddings can effectively encode the semantic regularities, we believe that they can be also used as alternative features of POS tags (or arc labels) in other NLP tasks, and help boost the performance.

What do W_1^w, W_1^t, W_1^l capture?

Knowing that E^t and E^l (as well as the word embeddings E^w) can capture semantic information very well, next we hope to investigate what each feature in the hidden layer has really learned.

Since we currently only have $h = 200$ learned dense features, we wonder if it is sufficient to learn the word conjunctions as sparse indicator features, or even more. We examine the weights $W_1^w(k, \cdot) \in \mathbb{R}^{d \cdot n_w}$, $W_1^t(k, \cdot) \in \mathbb{R}^{d \cdot n_t}$, $W_1^l(k, \cdot) \in \mathbb{R}^{d \cdot n_l}$ for each hidden unit k, and reshape them to $d \times n_t$, $d \times n_w$, $d \times n_l$ matrices, such that the weights of each column corresponds to the embeddings of one specific element (e.g., $s_1.t$).

We pick the weights with absolute value > 0.2, and visualize them for each feature. Figure 6 gives the visualization of three sampled features, and it exhibits many interesting phenomena:

- Different features have varied distributions of the weights. However, most of the discriminative weights come from W_1^t (the middle zone in Figure 6), and this further justifies the importance of POS tags in dependency parsing.
- We carefully examine many of the $h = 200$ features, and find that they actually encode very different views of information. For the three sampled features in Figure 6, the largest weights are dominated by:
  - Feature 1: $s_1.t, s_2.t, lc(s_1).t.$
  - Feautre 2: $rc(s_1).t, s_1.t, b_1.t.$
  - Feature 3: $s_1.t, s_1.w, lc(s_1).t, lc(s_1).l.$

  These features all seem very plausible, as observed in the experiments on indicator feature systems. Thus our model is able to automatically identify the most useful information for predictions, instead of hand-crafting them as indicator features.
- More importantly, we can extract features regarding the conjunctions of more than 3 elements easily, and also those not presented in the indicator feature systems. For example, the 3rd feature above captures the conjunction of words and POS tags of s_1, the tag of its leftmost child, and also the label between them, while this information is not encoded in the original feature templates of (Zhang and Nivre, 2011).

5 Related Work

There have been several lines of earlier work in using neural networks for parsing which have points of overlap but also major differences from our work here. One big difference is that much early work uses localist one-hot word representations rather than the distributed representations of modern work. (Mayberry III and Miikkulainen, 1999) explored a shift reduce constituency parser with one-hot word representations and did subsequent parsing work in (Mayberry III and Miikkulainen, 2005).

(Henderson, 2004) was the first to attempt to use neural networks in a broad-coverage Penn Treebank parser, using a simple synchrony network to predict parse decisions in a constituency parser. More recently, (Titov and Henderson, 2007) applied Incremental Sigmoid Belief Networks to constituency parsing and then (Garg and Henderson, 2011) extended this work to transition-based dependency parsers using a Temporal Restricted Boltzman Machine. These are very different neural network architectures, and are much less scalable and in practice a restricted vocabulary was used to make the architecture practical.

There have been a number of recent uses of deep learning for constituency parsing (Collobert, 2011; Socher et al., 2013). (Socher et al., 2014) has also built models over dependency representations but this work has not attempted to learn neural networks for dependency parsing.

Most recently, (Stenetorp, 2013) attempted to build recursive neural networks for transition-based dependency parsing, however the empirical performance of his model is still unsatisfactory.

6 Conclusion

We have presented a novel dependency parser using neural networks. Experimental evaluations show that our parser outperforms other greedy parsers using sparse indicator features in both accuracy and speed. This is achieved by representing all words, POS tags and arc labels as dense vectors, and modeling their interactions through a novel cube activation function. Our model only relies on dense features, and is able to automatically learn the most useful feature conjunctions for making predictions.

An interesting line of future work is to combine our neural network based classifier with search-based models to further improve accuracy. Also,

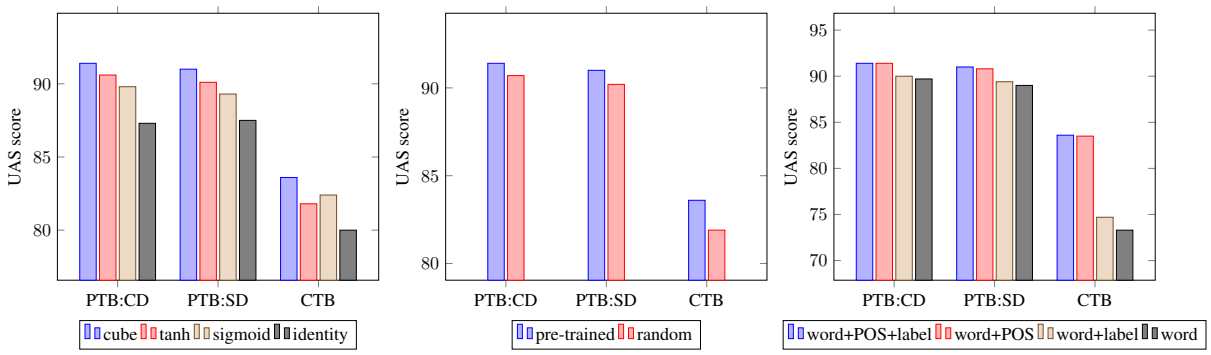


Figure 4: Effects of different parser components. Left: comparison of different activation functions. Middle: comparison of pre-trained word vectors and random initialization. Right: effects of POS and label embeddings.

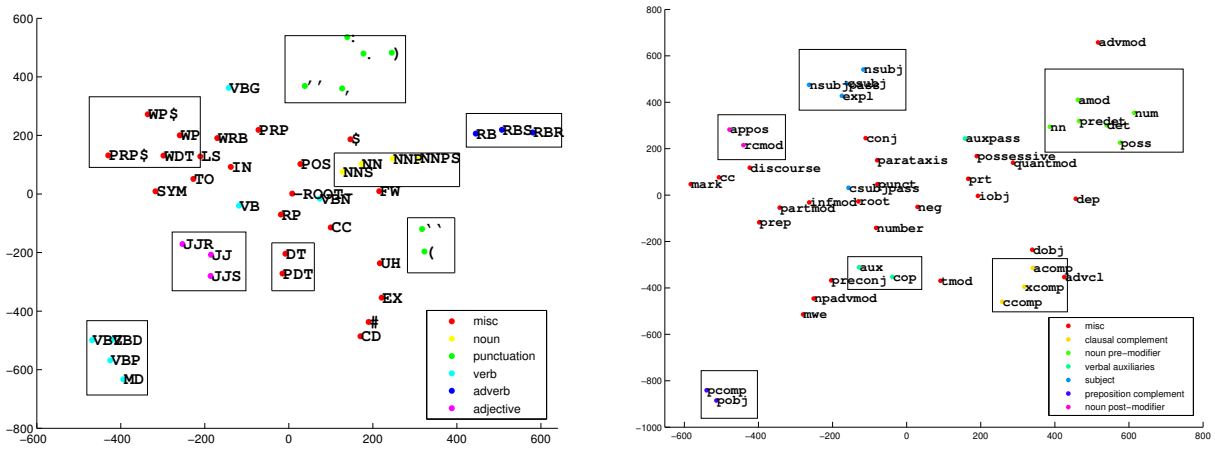


Figure 5: t-SNE visualization of POS and label embeddings.

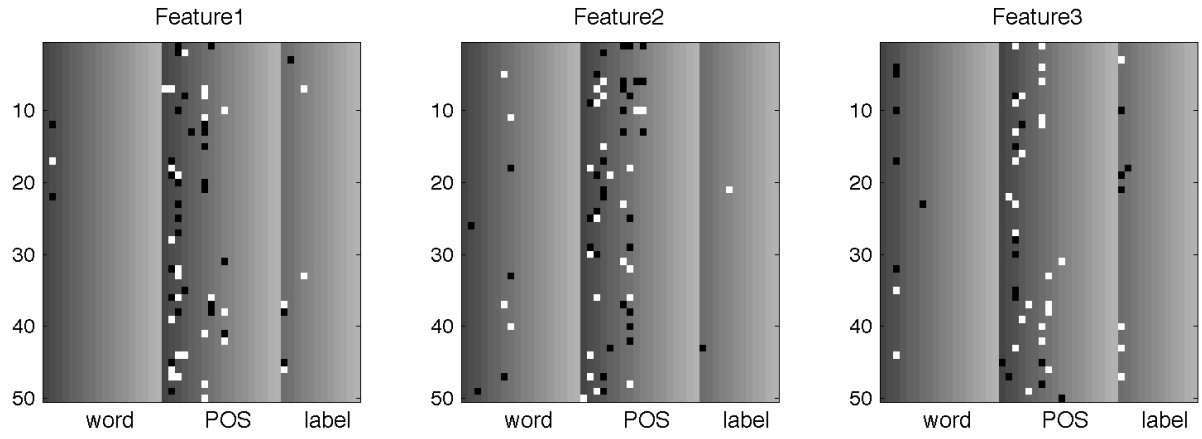


Figure 6: Three sampled features. In each feature, each row denotes a dimension of embeddings and each column denotes a chosen element, e.g., $s_1.t$ or $lc(s_1).w$, and the parameters are divided into 3 zones, corresponding to $W_1^w(k,:)$ (left), $W_1^t(k,:)$ (middle) and $W_1^l(k,:)$ (right). White and black dots denote the most positive weights and most negative weights respectively.

there is still room for improvement in our architecture, such as better capturing word conjunctions, or adding richer features (e.g., distance, valency).

Acknowledgments

Stanford University gratefully acknowledges the support of the Defense Advanced Research Projects Agency (DARPA) Deep Exploration and Filtering of Text (DEFT) Program under Air Force Research Laboratory (AFRL) contract no. FA8750-13-2-0040 and the Defense Threat Reduction Agency (DTRA) under Air Force Research Laboratory (AFRL) contract no. FA8650-10-C-7020. Any opinions, findings, and conclusion or recommendations expressed in this material are those of the authors and do not necessarily reflect the view of the DARPA, AFRL, or the US government.

References

Bernd Bohnet. 2010. Very high accuracy and fast dependency parsing is not a contradiction. In *Coling*.

Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*.

Ronan Collobert. 2011. Deep learning for efficient discriminative parsing. In *AISTATS*.

Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. 2006. Generating typed dependency parses from phrase structure parses. In *LREC*.

Jacob Devlin, Rabih Zbib, Zhongqiang Huang, Thomas Lamar, Richard Schwartz, and John Makhoul. 2014. Fast and robust neural network joint models for statistical machine translation. In *ACL*.

John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*.

Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. Liblinear: A library for large linear classification. *The Journal of Machine Learning Research*.

Nikhil Garg and James Henderson. 2011. Temporal restricted boltzmann machines for dependency parsing. In *ACL-HLT*.

He He, Hal Daumé III, and Jason Eisner. 2013. Dynamic feature selection for dependency parsing. In *EMNLP*.

James Henderson. 2004. Discriminative training of a neural network statistical parser. In *ACL*.

Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580.

Liang Huang, Wenbin Jiang, and Qun Liu. 2009. Bilingually-constrained (monolingual) shift-reduce parsing. In *EMNLP*.

Richard Johansson and Pierre Nugues. 2007. Extended constituent-to-dependency conversion for english. In *Proceedings of NODALIDA*, Tartu, Estonia.

Lingpeng Kong and Noah A. Smith. 2014. An empirical comparison of parsing methods for Stanford dependencies. *CoRR*, abs/1404.4314.

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

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

Marshall R. Mayberry III and Risto Miikkulainen. 1999. Sardsrn: A neural network shift-reduce parser. In *IJCAI*.

Marshall R. Mayberry III and Risto Miikkulainen. 2005. Broad-coverage parsing with neural networks. *Neural Processing Letters*.

Ryan McDonald and Fernando Pereira. 2006. Online learning of approximate dependency parsing algorithms. In *EACL*.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *NIPS*.

Joakim Nivre, Johan Hall, and Jens Nilsson. 2006. Maltparser: A data-driven parser-generator for dependency parsing. In *LREC*.

Richard Socher, John Bauer, Christopher D Manning, and Andrew Y Ng. 2013. Parsing with compositional vector grammars. In *ACL*.

Richard Socher, Andrej Karpathy, Quoc V. Le, Christopher D. Manning, and Andrew Y. Ng. 2014. Grounded compositional semantics for finding and describing images with sentences. *TACL*.

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

Ivan Titov and James Henderson. 2007. Fast and robust multilingual dependency parsing with a generative latent variable model. In *EMNLP-CoNLL*.

Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *NAACL*.

Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *The Journal of Machine Learning Research*.

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 *EMNLP*.

Yue Zhang and Joakim Nivre. 2011. Transition-based dependency parsing with rich non-local features. In *ACL*.