An Effective Neural Network Model for Graph-based Dependency Parsing

Wenzhe Pei **Tao Ge** **Baobao Chang**[*]
Key Laboratory of Computational Linguistics, Ministry of Education,
School of Electronics Engineering and Computer Science, Peking University,
No.5 Yiheyuan Road, Haidian District, Beijing, 100871, China
Collaborative Innovation Center for Language Ability, Xuzhou, 221009, China.
{peiwenzhe,getao,chbb}@pku.edu.cn

Abstract

Most existing graph-based parsing models rely on millions of hand-crafted features, which limits their generalization ability and slows down the parsing speed. In this paper, we propose a general and effective Neural Network model for graph-based dependency parsing. Our model can automatically learn high-order feature combinations using only atomic features by exploiting a novel activation function *tanh-cube*. Moreover, we propose a simple yet effective way to utilize phrase-level information that is expensive to use in conventional graph-based parsers. Experiments on the English Penn Treebank show that parsers based on our model perform better than conventional graph-based parsers.

1 Introduction

Dependency parsing is essential for computers to understand natural languages, whose performance may have a direct effect on many NLP application. Due to its importance, dependency parsing, has been studied for tens of years. Among a variety of dependency parsing approaches (McDonald et al., 2005; McDonald and Pereira, 2006; Carreras, 2007; Koo and Collins, 2010; Zhang and Nivre, 2011), graph-based models seem to be one of the most successful solutions to the challenge due to its ability of scoring the parsing decisions on whole-tree basis. Typical graph-based models factor the dependency tree into subgraphs, ranging from the smallest edge (first-order) to a controllable bigger subgraph consisting of more than one single edge (second-order and third order), and score the whole tree by summing scores of the subgraphs. In these models, subgraphs are usually represented as a high-dimensional feature vectors which are fed into a linear model to learn the feature weight for scoring the subgraphs.

In spite of their advantages, conventional graph-based models rely heavily on an enormous number of hand-crafted features, which brings about serious problems. First, a mass of features could put the models in the risk of overfitting and slow down the parsing speed, especially in the high-order models where combinational features capturing interactions between head, modifier, siblings and (or) grandparent could easily explode the feature space. In addition, feature design requires domain expertise, which means useful features are likely to be neglected due to a lack of domain knowledge. As a matter of fact, these two problems exist in most graph-based models, which have stuck the development of dependency parsing for a few years.

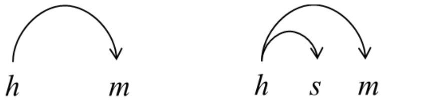
To ease the problem of feature engineering, we propose a general and effective Neural Network model for graph-based dependency parsing in this paper. The main advantages of our model are as follows:

- Instead of using large number of hand-crafted features, our model only uses atomic features (Chen et al., 2014) such as word unigrams and POS-tag unigrams. Feature combinations and high-order features are automatically learned with our novel activation function *tanh-cube*, thus alleviating the heavy burden of feature engineering in conventional graph-based models (McDonald et al., 2005; McDonald and Pereira, 2006; Koo and Collins, 2010). Not only does it avoid the risk of overfitting but also it discovers useful new features that have never been used in conventional parsers.

- We propose to exploit phrase-level information through distributed representation for phrases (phrase embeddings). It not only en-

[*] Corresponding author

Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics
and the 7th International Joint Conference on Natural Language Processing, pages 313–322,
Beijing, China, July 26-31, 2015.



(a) First-order (b) Second-order

Figure 1: First-order and Second-order factorization strategy. Here h stands for head word, m stands for modifier word and s stands for the sibling of m.

ables our model to exploit richer context information that previous work did not consider due to the curse of dimension but also captures inherent correlations between phrases.

- Unlike other neural network based models (Chen et al., 2014; Le and Zuidema, 2014) where an additional parser is needed for either extracting features (Chen et al., 2014) or generating k-best list for reranking (Le and Zuidema, 2014), both training and decoding in our model are performed based on our neural network architecture in an effective way.
- Our model does not impose any change to the decoding process of conventional graph-based parsing model. First-order, second-order and higher order models can be easily implemented using our model.

We implement three effective models with increasing expressive capabilities. The first model is a simple first-order model that uses only atomic features and does not use any combinational features. Despite its simpleness, it outperforms conventional first-order model (McDonald et al., 2005) and has a faster parsing speed. To further strengthen our parsing model, we incorporate phrase embeddings into the model, which significantly improves the parsing accuracy. Finally, we extend our first-order model to a second-order model that exploits interactions between two adjacent dependency edges as in McDonald and Pereira (2006) thus further improves the model performance.

We evaluate our models on the English Penn Treebank. Experiment results show that both our first-order and second-order models outperform the corresponding conventional models.

2 Neural Network Model

A dependency tree is a rooted, directed tree spanning the whole sentence. Given a sentence x, graph-based models formulates the parsing process as a searching problem:

$$y^*(x) = \operatorname*{arg\,max}_{\hat{y} \in Y(x)} Score(x, \hat{y}(x); \theta) \quad (1)$$

where $y^*(x)$ is tree with highest score, $Y(x)$ is the set of all trees compatible with x, θ are model parameters and $Score(x, \hat{y}(x); \theta)$ represents how likely that a particular tree $\hat{y}(x)$ is the correct analysis for x. However, the size of $Y(x)$ is exponential large, which makes it impractical to solve equation (1) directly. Previous work (McDonald et al., 2005; McDonald and Pereira, 2006; Koo and Collins, 2010) assumes that the score of $\hat{y}(x)$ factors through the scores of subgraphs c of $\hat{y}(x)$ so that efficient algorithms can be designed for decoding:

$$Score(x, \hat{y}(x); \theta) = \sum_{c \in \hat{y}(x)} ScoreF(x, c; \theta) \quad (2)$$

Figure 1 gives two examples of commonly used factorization strategy proposed by Mcdonald et.al (2005) and Mcdonald and Pereira (2006). The simplest subgraph uses a first-order factorization (McDonald et al., 2005) which decomposes a dependency tree into single dependency arcs (Figure 1(a)). Based on the first-order model, second-order factorization (McDonald and Pereira, 2006) (Figure 1(b)) brings sibling information into decoding. Specifically, a sibling part consists of a triple of indices (h, m, s) where (h, m) and (h, s) are dependencies and s and m are successive modifiers to the same side of h.

The most common choice for $ScoreF(x, c; \theta)$, which is the score function for subgraph c in the tree, is a simple linear function:

$$ScoreF(x, c; \theta) = \boldsymbol{w} \cdot \boldsymbol{f}(x, c) \quad (3)$$

where $\boldsymbol{f}(x, c)$ is the feature representation of subgraph c and $\boldsymbol{w}$ is the corresponding weight vector. However, the effectiveness of this function relies heavily on the design of feature vector $\boldsymbol{f}(x, c)$. In previous work (McDonald et al., 2005; McDonald and Pereira, 2006), millions of hand-crafted features were used to capture context and structure information in the subgraph which not only limits the model's ability to generalize well but only slows down the parsing speed.

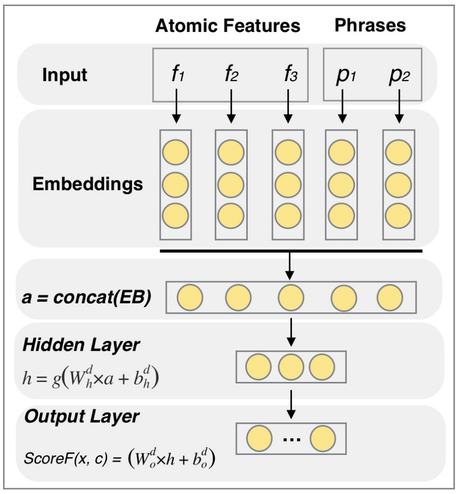


Figure 2: Architecture of the Neural Network

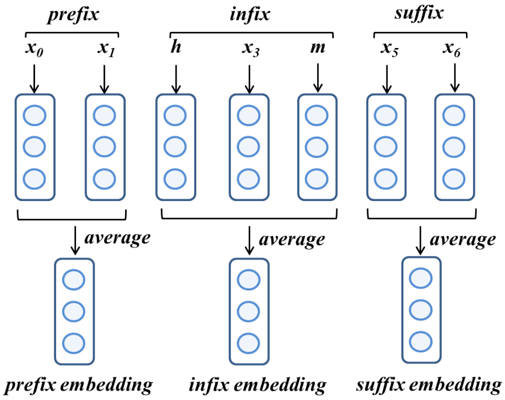


Figure 3: Illustration for phrase embeddings. h, m and x_0 to x_6 are words in the sentence.

In our work, we propose a neural network model for scoring subgraph c in the tree:

$$ScoreF(x, c; \theta) = \boldsymbol{NN}(x, c) \qquad (4)$$

where $\boldsymbol{NN}$ is our scoring function based on neural network (Figure 2). As we will show in the following sections, it alleviates the heavy burden of feature engineering in conventional graph-based models and achieves better performance by automatically learning useful information in the data.

The effectiveness of our neural network depends on five key components: *Feature Embeddings*, *Phrase Embeddings*, *Direction-specific transformation*, *Learning Feature Combinations* and *Max-Margin Training*.

2.1 Feature Embeddings

As shown in Figure 2, part of the input to the neural network is feature representation of the subgraph. Instead of using millions of features as in conventional models, we only use use atomic features (Chen et al., 2014) such as word unigrams and POS-tag unigrams, which are less likely to be sparse. The detailed atomic features we use will be described in Section 3. Unlike conventional models, the atomic features in our model are transformed into their corresponding distributed representations (feature embeddings).

The idea of distributed representation for symbolic data is one of the most important reasons why neural network works in NLP tasks. It is shown that similar features will have similar embeddings which capture the syntactic and semantic information behind features (Bengio et al., 2003; Collobert et al., 2011; Schwenk et al., 2012; Mikolov et al., 2013; Socher et al., 2013; Pei et al., 2014).

Formally, we have a feature dictionary D of size $|D|$. Each feature $f \in D$ is represented as a real-valued vector (feature embedding) $Embed(f) \in \mathbb{R}^d$ where d is the dimensionality of the vector space. All feature embeddings stacking together forms the embedding matrix $M \in \mathbb{R}^{d \times |D|}$. The embedding matrix M is initialized randomly and trained by our model (Section 2.6).

2.2 Phrase Embeddings

Context information of word pairs[1] such as the dependency pair (h, m) has widely believed to be useful in graph-based models (McDonald et al., 2005; McDonald and Pereira, 2006). Given a sentence x, the context for $\boldsymbol{h}$ and $\boldsymbol{m}$ includes three context parts: *prefix*, *infix* and *suffix*, as illustrated in Figure 3. We call these parts *phrases* in our work.

Context representation in conventional models are limited: First, phrases cannot be used as features directly because of the data sparseness problem. Therefore, phrases are backed off to low-order representation such as bigrams and trigrams. For example, Mcdonald et.al (2005) used tri-gram features of *infix* between head-modifier pair (h, m). Sometimes even tri-grams are expensive to use, which is the reason why Mcdonald and Pereira (2006) chose to ignore features over triples of words in their second-order model to prevent from exploding the size of the feature space. Sec-

[1] A word pair is not limited to the dependency pair (h, m). It could be any pair with particular relation (e.g., sibling pair (s, m) in Figure 1). Figure 3 only uses (h, m) as an example.

ond, bigrams or tri-grams are lexical features thus cannot capture syntactic and semantic information behind phrases. For instance, "*hit the ball*" and "*kick the football*" should have similar representations because they share similar syntactic structures, but lexical tri-grams will fail to capture their similarity.

Unlike previous work, we propose to use distributed representation (phrase embedding) of phrases to capture phrase-level information. We use a simple yet effective way to calculate phrase embeddings from word (POS-tag) embeddings. As shown in Figure 3, we average the word embeddings in *prefix*, *infix* and *suffix* respectively and get three global word-phrase embeddings. For pairs where no prefix or suffix exists, the corresponding embedding is set to zero. We also get three global POS-phrase embeddings which are calculated in the same way as words. These embeddings are then concatenated with feature embeddings and fed to the following hidden layer.

Phrase embeddings provide panorama representation of the context, allowing our model to capture richer context information compared with the back-off tri-gram representation. Moreover, as a distributed representation, phrase embeddings perform generalization over specific phrases, thus better capture the syntactic and semantic information than back-off tri-grams.

2.3 Direction-specific Transformation

In dependency representation of sentence, the edge direction indicates which one of the words is the head h and which one is the modifier m. Unlike previous work (McDonald et al., 2005; McDonald and Pereira, 2006) that models the edge direction as feature to be conjoined with other features, we model the edge direction with direction-specific transformation.

As shown in Figure 2, the parameters in hidden layer (W_h^d, b_h^d) and the output layer (W_o^d, b_o^d) are bound with index $d \in \{0, 1\}$ which indicates the direction between head and modifier (0 for left arc and 1 for right arc). In this way, the model can learn direction-specific parameters and automatically capture the interactions between edge direction and other features.

2.4 Learning Feature Combination

The key to the success of graph-based dependency parsing is the design of features, especially combinational features. Effective as these features are, as we have said in Section 1, they are prone to overfitting and hard to design. In our work, we introduce a new activation function that can automatically learn these feature combinations.

As shown in Figure 2, we first concatenate the embeddings into a single vector a. Then a is fed into the next layer which performs linear transformation followed by an element-wise activation function g:

$$h = g(W_h^d a + b_h^d) \tag{5}$$

Our new activation function g is defined as follows:

$$g(l) = tanh(l^3 + l) \tag{6}$$

where l is the result of linear transformation and *tanh* is the *hyperbolic tangent* activation function widely used in neural networks. We call this new activation function *tanh-cube*.

As we can see, without the cube term, *tanh-cube* would be just the same as the conventional non-linear transformation in most neural networks. The cube extension is added to enhance the ability to capture complex interactions between input features. Intuitively, the cube term in each hidden unit directly models feature combinations in a multiplicative way:

$$(w_1a_1 + w_2a_2 + ... + w_na_n + b)^3 = \\ \sum_{i,j,k}(w_iw_jw_k)a_ia_ja_k + \sum_{i,j} b(w_iw_j)a_ia_j...$$

These feature combinations are hand-designed in conventional graph-based models but our model learns these combinations automatically and encodes them in the model parameters.

Similar ideas were also proposed in previous works (Socher et al., 2013; Pei et al., 2014; Chen and Manning, 2014). Socher et.al (2013) and Pei et.al (2014) used a tensor-based activation function to learn feature combinations. However, tensor-based transformation is quite slow even with tensor factorization (Pei et al., 2014). Chen and Manning (2014) proposed to use cube function $g(l) = l^3$ which inspires our *tanh-cube* function. Compared with cube function, *tanh-cube* has three advantages:

- The cube function is unbounded, making the activation output either too small or too big if the norm of input l is not properly controlled, especially in deep neural network. On the contrary, *tanh-cube* is bounded by the *tanh* function thus safer to use in deep neural network.

- Intuitively, the behavior of cube function resembles the "polynomial kernel" in SVM. In fact, SVM can be seen as a special one-hidden-layer neural network where the kernel function that performs non-linear transformation is seen as a hidden layer and support vectors as hidden units. Compared with cube function, *tanh-cube* combines the power of "kernel function" with the *tanh* non-linear transformation in neural network.
- Last but not least, as we will show in Section 4, *tanh-cube* converges faster than the cube function although the rigorous proof is still open to investigate.

2.5 Model Output

After the non-linear transformation of hidden layer, the score of the subgraph c is calculated in the output layer using a simple linear function:

$$ScoreF(x, c) = W_o^d h + b_o^d \tag{7}$$

The output score $ScoreF(x, c) \in \mathbb{R}^{|L|}$ is a score vector where $|L|$ is the number of dependency types and each dimension of $ScoreF(x, c)$ is the score for each kind of dependency type of head-modifier pair (i.e. (h, m) in Figure 1).

2.6 Max-Margin Training

The parameters of our model are $\theta = \{W_h^d, b_h^d, W_o^d, b_o^d, M\}$. All parameters are initialized with uniform distribution within (-0.01, 0.01).

For model training, we use the Max-Margin criterion. Given a training instance (x, y), we search for the dependency tree with the highest score computed as equation (1) in Section 2. The object of Max-Margin training is that the highest scoring tree is the correct one: $y^* = y$ and its score will be larger up to a margin to other possible tree $\hat{y} \in Y(x)$:

$$Score(x, y; \theta) \geq Score(x, \hat{y}; \theta) + \triangle(y, \hat{y})$$

The structured margin loss $\triangle(y, \hat{y})$ is defined as:

$$\triangle(y, \hat{y}) = \sum_{j}^{n} \kappa \mathbf{1}\{h(y, x_j) \neq h(\hat{y}, x_j)\}$$

where n is the length of sentence x, $h(y, x_j)$ is the head (with type) for the j-th word of x in tree y and κ is a discount parameter. The loss is proportional to the number of word with an incorrect head and edge type in the proposed tree. This leads to the regularized objective function for m training examples:

$$\begin{aligned} J(\theta) &= \frac{1}{m} \sum_{i=1}^{m} l_i(\theta) + \frac{\lambda}{2} ||\theta||^2 \\ l_i(\theta) &= \max_{\hat{y} \in Y(x_i)} (Score(x_i, \hat{y}; \theta) + \triangle(y_i, \hat{y})) \\ &\quad - Score(x_i, y_i; \theta)) \end{aligned} \tag{8}$$

We use the diagonal variant of AdaGrad (Duchi et al., 2011) with minibatchs (batch size = 20) to minimize the object function. We also apply dropout (Hinton et al., 2012) with 0.5 rate to the hidden layer.

1-order-atomic	$h_{-2}.w$, $h_{-1}.w$, $h.w$, $h_1.w$, $h_2.w$ $h_{-2}.p$, $h_{-1}.p$, $h.p$, $h_1.p$, $h_2.p$ $m_{-2}.w$, $m_{-1}.w$, $m.w$, $m_1.w$, $m_2.w$ $m_{-2}.p$, $m_{-1}.p$, $m.p$, $m_1.p$, $m_2.p$ *dis(h, m)*
1-order-phrase	+ *hm_prefix.w*, *hm_infix.w*, *hm_suffix.w* + *hm_prefix.p*, *hm_infix.p*, *hm_suffix.p*
2-order-phrase	+ $s_{-2}.w$, $s_{-1}.w$, $s.w$, $s_1.w$, $s_2.w$ + $s_{-2}.p$, $s_{-1}.p$, $s.p$, $s_1.p$, $s_2.p$ + *sm_infix.w*, *sm_infix.p*

Table 1: Features in our three models. w is short for word and p for POS-tag. h indicates head and m indicates modifier. The subscript represents the relative position to the center word. $dis(h, m)$ is the distance between head and modifier. *hm_prefix*, *hm_infix* and *hm_suffix* are phrases for head-modifier pair (h, m). s indicates the sibling in second-order model. *sm_infix* is the *infix* phrase between sibling pair (s, m)

3 Model Implementation

Base on our Neural Network model, we present three model implementations with increasing expressive capabilities in this section.

3.1 First-order models

We first implement two first-order models: **1-order-atomic** and **1-order-phrase**. We use the Eisner (2000) algorithm for decoding. The first two rows of Table 1 list the features we use in these two models.

1-order-atomic only uses atomic features as shown in the first row of Table 1. Specifically, the

	Models	**Dev**		Test		Speed (sent/s)
		UAS	LAS	UAS	LAS	
First-order	MSTParser-1-order	92.01	90.77	91.60	90.39	20
	1-order-atomic-rand	92.00	90.71	91.62	90.41	**55**
	1-order-atomic	92.19	90.94	92.14	90.92	**55**
	1-order-phrase-rand	92.47	91.19	92.25	91.05	26
	1-order-phrase	**92.82**	**91.48**	**92.59**	**91.37**	26
Second-order	MSTParser-2-order	92.70	91.48	92.30	91.06	14
	2-order-phrase-rand	93.39	92.10	92.99	91.79	10
	2-order-phrase	**93.57**	**92.29**	**93.29**	**92.13**	10
Third-order	(Koo and Collins, 2010)	93.49	N/A	93.04	N/A	N/A

Table 2: Comparison with conventional graph-based models.

head word and its local neighbor words that are within the distance of 2 are selected as the head's word unigram features. The modifier's word unigram features is extracted in the same way. We also use the POS-tags of the corresponding word features and the distance between head and modifier as additional atomic features.

We then improved **1-order-atomic** to **1-order-phrase** by incorporating additional phrase embeddings. The three phrase embeddings of head-modifier pair (h, m): *hm_prefix*, *hm_infix* and *hm_suffix* are calculated as in Section 2.2.

3.2 Second-order model

Our model can be easily extended to a second-order model using the second-order decoding algorithm (Eisner, 1996; McDonald and Pereira, 2006). The third row of Table 1 shows the additional features we use in our second-order model.

Sibling node and its local context are used as additional atomic features. We also used the *infix embedding* for the *infix* between sibling pair (s, m), which we call *sm_infix*. It is calculated in the same way as *infix* between head-modifier pair (h, m) (i.e., *hm_infix*) in Section 2.2 except that the word pair is now s and m. For cases where no sibling information is available, the corresponding sibling-related embeddings are set to zero vector.

4 Experiments

4.1 Experiment Setup

We use the English Penn Treebank (PTB) to evaluate our model implementations and Yamada and Matsumoto (2003) head rules are used to extract dependency trees. We follow the standard splits of PTB3, using section 2-21 for training, section 22 as development set and 23 as test set. The Stanford POS Tagger (Toutanova et al., 2003) with ten-way jackknifing of the training data is used for assigning POS tags (accuracy ≈ 97.2%).

Hyper-parameters of our models are tuned on the development set and their final settings are as follows: embedding size d = 50, hidden layer (Layer 2) size = 200, regularization parameter λ = 10^{-4}, discount parameter for margin loss κ = 0.3, initial learning rate of AdaGrad alpha = 0.1.

4.2 Experiment Results

Table 2 compares our models with several conventional graph-based parsers. We use MSTParser[2] for conventional first-order model (McDonald et al., 2005) and second-order model (McDonald and Pereira, 2006). We also include the result of a third-order model of Koo and Collins (2010) for comparison[3]. For our models, we report the results with and without unsupervised pre-training. Pre-training only trains the word-based feature embeddings on Gigaword corpus (Graff et al., 2003) using *word2vec*[4] and all other parameters are still initialized randomly. In all experiments, we report unlabeled attachment scores (UAS) and labeled attachment scores (LAS) and punctuation[5] is excluded in all evaluation metrics. The parsing speeds are measured on a workstation with Intel Xeon 3.4GHz CPU and 32GB RAM.

As we can see, even with random initialization, **1-order-atomic-rand** performs as well as conventional first-order model and both **1-order-phrase-**

[2] http://sourceforge.net/projects/mstparser

[3] Note that Koo and Collins (2010)'s third-order model and our models are not strict comparable since their model is an unlabeled model.

[4] https://code.google.com/p/word2vec/

[5] Following previous work, a token is a punctuation if its POS tag is {“ ” : , .}

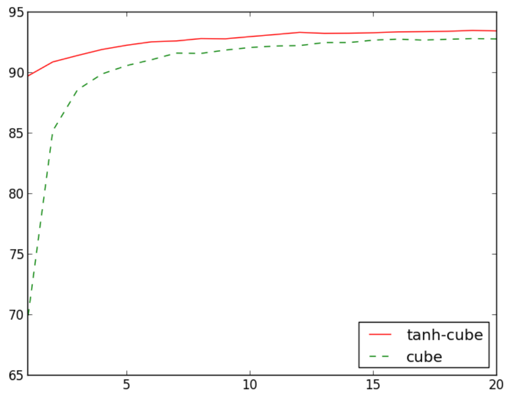


Figure 4: Convergence curve for *tanh-cube* and *cube* activation function.

rand and **2-order-phrase-rand** perform better than conventional models in MSTParser. Pre-training further improves the performance of all three models, which is consistent with the conclusion of previous work (Pei et al., 2014; Chen and Manning, 2014). Moreover, **1-order-phrase** performs better than **1-order-atomic**, which shows that phrase embeddings do improve the model. **2-order-phrase** further improves the performance because of the more expressive second-order factorization. All three models perform significantly better than their counterparts in MSTParser where millions of features are used and **1-order-phrase** works surprisingly well that it even beats the conventional second-order model.

With regard to parsing speed, **1-order-atomic** is the fastest while other two models have similar speeds as MSTParser. Further speed up could be achieved by using pre-computing strategy as mentioned in Chen and Manning (2014). We did not try this strategy since parsing speed is not the main focus of this paper.

Model	*tanh-cube*	*cube*	*tanh*
1-order-atomic	**92.19**	91.97	91.73
1-order-phrase	**92.82**	92.25	92.13
2-order-phrase	**93.57**	92.95	92.91

Table 3: Model Performance of different activation functions.

We also investigated the effect of different activation functions. We trained our models with the same configuration except for the activation function. Table 3 lists the UAS of three models on development set.

Feature Type	**Instance**	**Neighboors**
Words (word2vec)	in	the, of, and, for, from
	his	himself, her, he, him, father
	which	its, essentially, similar, that, also
Words (Our model)	in	on, at, behind, among, during
	his	her, my, their, its, he
	which	where, who, whom, whose, though
POS-tags	NN	NNPS, NNS, EX, NNP, POS
	JJ	JJR, JJS, PDT, RBR, RBS

Table 4: Examples of similar words and POS-tags according to feature embeddings.

As we can see, *tanh-cube* function outperforms *cube* function because of advantages we mentioned in Section 2.4. Moreover, both *tanh-cube* function and *cube* function performs better than *tanh* function. The reason is that the cube term can capture more interactions between input features.

We also plot the UAS of **2-order-phrase** during each iteration of training. As shown in Figure 4, *tanh-cube* function converges faster than *cube* function.

4.3 Qualitative Analysis

In order to see why our models work, we made qualitative analysis on different aspects of our model.

Ability of Feature Abstraction

Feature embeddings give our model the ability of feature abstraction. They capture the inherent correlations between features so that syntactic similar features will have similar representations, which makes our model generalizes well on unseen data.

Table 4 shows the effect of different feature embeddings which are obtained from **2-order-phrase** after training. For each kind of feature type, we list several features as well as top 5 features that are nearest (measured by Euclidean distance) to the corresponding feature according to their embeddings.

We first analysis the effect of word embeddings after training. For comparison, we also list the initial word embeddings in word2vec. As we can see, in word2vec word embeddings, words that are similar to *in* and *which* tends to be those

Phrase	Neighboor
On a Saturday morning	On Monday night football On Sunday On Saturday On Tuesday afternoon On recent Saturday morning
most of it	of it of it all some of it also most of these are only some of
big investment bank	great investment bank bank investment entire equity investment another cash equity investor real estate lending division

Table 5: Examples of similar phrases according to phrase embeddings.

co-occuring with them and for word *his*, similar words are morphologies of *he*. On the contrary, similar words measured by our embeddings have similar syntactic functions. This is helpful for dependency parsing since parsing models care more about the syntactic functions of words rather than their collocations or morphologies.

POS-tag embeddings also show similar behavior with word embeddings. As shown in Table 4, our model captures similarities between POS-tags even though their embeddings are initialized randomly.

We also investigated the effect of phrase embeddings in the same way as feature embeddings. Table 5 lists the examples of similar phrases. Our phrase embeddings work pretty well given that only a simple averaging strategy is used. Phrases that are close to each other tend to share similar syntactic and semantic information. By using phrase embeddings, our model sees panorama of the context rather than limited word tri-grams and thus captures richer context information, which is the reason why phrase embeddings significantly improve the performance.

Ability of Feature Learning

Finally, we try to unveil the mysterious hidden layer and investigate what features it learns. For each hidden unit of **2-order-phrase**, we get its connections with embeddings (i.e., W_h^d in Figure 2) and pick the connections whose weights have absolute value > 0.1. We sampled several hidden units and invenstigated which features their highly weighted connections belong to:

- Hidden 1: $h.w$, $m.w$, $h_{-1}.w$, $m_1.w$
- Hidden 2: $h.p$, $m.p$, $s.p$
- Hidden 3: *hm_infix.p, hm_infix.w, hm_prefix.w*
- Hidden 4: *hm_infix.w, hm_prefix.w, sm_infix.w*
- Hidden 5: *hm_infix.p, hm_infix.w, hm_suffix.w*

The samples above give qualitative results of what features the hidden layer learns:

- Hidden unit 1 and 2 show that atomic features of *head*, *modifier*, *sibling* and their local context words are useful in our model, which is consistent with our expectations since these features are also very important features in conventional graph-based models (McDonald and Pereira, 2006).
- Features in the same hidden unit will “combine” with each other through our *tanh-cube* activation function. As we can see, feature combination in hidden unit 2 were also used in Mcdonald and Pereira (2006). However, these feature combinations are automatically captured by our model without the labor-intensive feature engineering.
- Hidden unit 3 to 5 show that phrase-level information like *hm_prefix*, *hm_suffix* and *sm_infix* are effective in our model. These features are not used in conventional second-order model (McDonald and Pereira, 2006) because they could explode the feature space. Through our tanh-cube activation function, our model further captures the interactions between phrases and other features without the concern of overfitting.

5 Related Work

Models for dependency parsing have been studied with considerable effort in the NLP community. Among them, we only focus on the graph-based models here. Most previous systems address this task by using linear statistical models with carefully designed context and structure features. The types of features available rely on tree factorization and decoding algorithm. Mcdonald et.al (2005) proposed the first-order model which is also know as arc-factored model. Efficient decoding can be performed with Eisner (2000) algorithm in $O(n^3)$ time and $O(n^2)$ space. Mcdonald and Pereira (2006) further extend the first-order model to second-order model where sibling information is available during decoding. Eisner (2000) algorithm can be modified trivially for second-order decoding. Carreras (2007) proposed a more powerful second-order model that can score both sibling and grandchild parts with the cost of $O(n^4)$ time and $O(n^3)$ space. To exploit more structure information, Koo and Collins (2010) proposed three third-order models with computational requirements of $O(n^4)$ time and $O(n^3)$ space.

Recently, neural network models have been increasingly focused on for their ability to minimize the effort in feature engineering. Chen et.al (2014) proposed an approach to automatically learning feature embeddings for graph-based dependency parsing. The learned feature embeddings are used as additional features in conventional graph-based model. Le and Zuidema (2014) proprosed an infinite-order model based on recursive neural network. However, their model can only be used as an reranking model since decoding is intractable.

Compared with these work, our model is a general and standalone neural network model. Both training and decoding in our model are performed based on our neural network architecture in an effective way. Although only first-order and second-order models are implemented in our work, higher-order graph-based models can be easily implemented using our model.

6 Conclusion

In this paper, we propose a general and effective neural network model that can automatically learn feature combinations with our novel activation function. Moreover, we introduce a simple yet effect way to utilize phrase-level information, which greatly improves the model performance. Experiments on the benchmark dataset show that our model achieves better results than conventional models.

Acknowledgments

This work is supported by National Natural Science Foundation of China under Grant No. 61273318 and National Key Basic Research Program of China 2014CB340504. We want to thank Miaohong Chen and Pingping Huang for their valuable comments on the initial idea and helping pre-process the data.

References

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155.

Xavier Carreras. 2007. Experiments with a higher-order projective dependency parser. In *EMNLP-CoNLL*, pages 957–961.

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.

Wenliang Chen, Yue Zhang, and Min Zhang. 2014. Feature embedding for dependency parsing. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pages 816–826, Dublin, Ireland, August. Dublin City University and Association for Computational Linguistics.

Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, 12:2493–2537.

John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 999999:2121–2159.

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

Jason Eisner. 2000. Bilexical grammars and their cubic-time parsing algorithms. In *Advances in probabilistic and other parsing technologies*, pages 29–61. Springer.

David Graff, Junbo Kong, Ke Chen, and Kazuaki Maeda. 2003. English gigaword. *Linguistic Data Consortium, Philadelphia.*

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

Terry Koo and Michael Collins. 2010. Efficient third-order dependency parsers. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1–11. 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.

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

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

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Wenzhe Pei, Tao Ge, and Baobao Chang. 2014. Max-margin tensor neural network for chinese word segmentation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 293–303, Baltimore, Maryland, June. Association for Computational Linguistics.

Holger Schwenk, Anthony Rousseau, and Mohammed Attik. 2012. Large, pruned or continuous space language models on a gpu for statistical machine translation. In *Proceedings of the NAACL-HLT 2012 Workshop: Will We Ever Really Replace the N-gram Model? On the Future of Language Modeling for HLT*, pages 11–19. Association for Computational Linguistics.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. 2013. 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.

Kristina Toutanova, Dan Klein, Christopher D Manning, and Yoram Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 173–180. Association for Computational Linguistics.

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

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.