

# Dependency Parser using Neural Network

## Class Project for CS6320.001

Jiashuai Lu

jxl173630@utdallas.edu

**This project's main purpose is reimplementing a transition based dependency parser based on the research that Chen and Manning did in 2014 (1). During the work, we construct a one hidden layer neural network classifier for deciding the choice of dependencies in every step of the process of parsing a sentence in a greedy way, train our model with different choices of features, non-linear functions and compare the results by using our classifier on test data.**

## Introduction

In Natural Language Processing (NLP) area, ambiguities widely exist. Among those, lexical ambiguity is one of the major difficulty during solving NLP problems. It consists of syntactic, semantic, and structural ambiguities. The last one occurs when a sentence could be assigned to more than one possible parse structures. Syntactical parsing is one way to solve the structural ambiguities by using certain grammars. Dependency parsing which use dependency grammar concentrates on relations between pairs of words while CFG-style phrase structure parsing mainly focuses on constituents. Although the former cannot capture full phrase structures as the latter, it has some ad-

vantages when phrasal constituents and phrase-structure rules do not play a direct role in the applications (2).

By using feature-based discriminative dependency parser, considerable achievement has been made before Chen and Manning's research. Especially the transition-based dependency parser's subclass has achieved really impressive speed for parsing (3). Nevertheless, almost all the previous parser have some imperfectness. One big problem is that most of them are using sparse but huge features model. This brought a big problem for estimate the features weights. Intuitively, when we are trying to determine the relationship between two words in a sentence, considering the interaction between more than only these two words improved a lot in the performance of parsing (4). Unfortunately, determining those so many features weights limits the speed of traditional dependency parser to a great extent.

In this project, we try to eliminate those disadvantages without lose the high accuracy the high-order interaction bring us. To achieve this, instead of using sparse features, we adopt the dense features model which has shown its effect in a number of NLP subareas. For example, dense, short vectors for representing words are easy to capture synonymy than sparse vector, since two distinct words are represented by two distinct dimensions in sparse vectors (2). This provide us a good base to training words and their interactions in our model (1).

For exploiting the dense features, we reimplement a neural network for choosing operation for every state during transition-based dependency parsing. It learns compact 48 dense vector representations of words, part-of-speech (POS) tags, and dependency labels respectively (our experimental data is annotated by Stanford dependencies). Our parser achieve a very fast speed without losing too much precision.

## Background

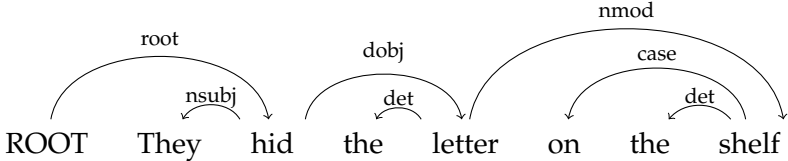
There are two modern approaches to dependency parsing. Optimization-based approaches that search a space of trees for the tree that best matches some criteria while shift-reduce approaches that greedily take actions based on the current word and state. Transition-based dependency parsing is motivated by the shift-reduce parsing (2). It is a stack-based method which defines a stack and always tries to determine the relation between the top two elements in the stack based on current state's configuration.

A configuration typically consists of the stack, input buffer of words, and a set of dependency relations that has been determined till current state. In the initial state's configuration, the stack is empty, all the tokens in the sentence needs to be processed are treated as input for this stack, and the set of dependency relations is empty too. Once the input buffer and stack are both empty, the parsing finishes with returning a set of relations which could be used to construct a corresponding dependency tree.

Our parser is constructed based on an arc-standard system (5). This system defines three basic operations to implement a left-to-right bottom-up dependency parsing as Figure 1 shows. Let  $S$  denotes stack,  $B$  denotes input buffer, and  $A$  denotes set of dependency arcs. The initial configuration for a sentence  $w_1, w_2, \dots, w_n$  is  $S=\emptyset$ ,  $B=[w_1, w_2, \dots, w_n]$ ,  $A=\emptyset$ . Accordingly, the final configuration is a status that  $S=\emptyset$ ,  $B=\emptyset$ .

<i>Initial state</i> : $\langle S = \emptyset, B, A = \emptyset \rangle$
<i>Final state</i> : $\langle S, B = \emptyset, A \rangle$
<i>Left – Reduce</i> : $\langle w_j, w_i   S, B, A \rangle \rightarrow \langle w_j   S, B, A \cup \{(w_j, w_i)\} \rangle$
<i>Right – Reduce</i> : $\langle w_j, w_i   S, B, A \rangle \rightarrow \langle w_i   S, B, A \cup \{(w_i, w_j)\} \rangle$
<i>Shift</i> : $\langle S, w_i   B, A \rangle \rightarrow \langle w_i   S, B, A \rangle$

Figure 1. Left-to-right bottom-up dependency parsing



Transition	Stack	Buffer	A
	[]	[They hid the letter on the shelf]	$\emptyset$
Shift	[They]	[hid the letter on the shelf]	A
Shift	[They hid]	[the letter on the shelf]	A
Left-reduce	[hid]	[the letter on the shelf]	$A \cup \text{nsubj}(\text{hid}, \text{They})$
Shift	[hid the]	[letter on the shelf]	A
Shift	[hid the letter]	[on the shelf]	A
Left-reduce	[hid letter]	[on the shelf]	$A \cup \text{det}(\text{letter}, \text{the})$
Shift	[hid letter on]	[the shelf]	A
Shift	[hid letter on the]	[shelf]	A
Shift	[hid letter on the shelf]	[]	A
Left-reduce	[hid letter on shelf]	[]	$A \cup \text{det}(\text{shelf}, \text{the})$
Left-reduce	[hid letter shelf]	[]	$A \cup \text{case}(\text{shelf}, \text{on})$
Right-reduce	[hid letter]	[]	$A \cup \text{case}(\text{letter}, \text{shelf})$
Right-reduce	[hid]	[]	$A \cup \text{dobj}(\text{hid}, \text{letter})$
Right-reduce	[]	[]	$A \cup \text{root}(\text{hid})$

Figure 2. An example of arc-standard transition-based dependency parsing.

Typically, every word of the sentence will be processed by two basic operation during the parsing. The first is being moved to stack by a shift operation, and the second is being popped out from the stack by a left-reduce or a right-reduce (depends on the configuration). Thus, for a sentence with  $N$  tokens, this system is expected to processed the sentence with use  $2N$  transition operations. Figure 2 is an example of an arc-standard transition based dependency parsing. Consider the different type of labels of the dependency format we are using, the number of different transitions would be  $N_T = 2N_t + 1$  in which  $N_t$  is the number of different type of labels. Basically, our parser is going to predict the correct transition based on current configuration.

For predicting the correct transition, previous approaches often treat the conjunction of 1~3 tokens' word, part-of speech tags and dependency labels in the current

stack or buffer as the features. For example, use  $B_0pB_1pB_2p$ ;  $S_0pB_0pB_1p$ ;  $S_0hpS_0pB_0p$ ;  $S_0pS_0lpB_0p$ ;  $S_0pS_0rpB_0p$ ;  $S_0pB_0pB_0lp$  as 6 different features ( $S_i$  means the  $i$ th element in the stack and  $B_i$  means the  $i$ th element in the buffer,  $p$  denotes the token's POS tag,  $l$  and  $r$  means its leftmost child and rightmost child) (6). This kind of feature model has three main disadvantages: (1) Sparsity, lexicalized features are usually sparse. However, these features are often vital for parsing; (2) Partiality, simply combining two or three words from the configuration is hard to capture all necessary relations; (3) High cost for computing features, extracting specified features from source requires significant computations (1). In the next two sections we will detail our endeavor in this project is trying to introduce a neural network to alleviate those problems.

## Dependency Parsing using Neural Network

**Model** For representing the words without losing the relation between them, we use word embedding to represent words. Every word will be a  $d$ -dimensional vector  $e_i^w \in \mathbb{R}^d$ . And in our project, we use a pre-trained word2vector embeddings given by Stanford CS224n's assignment2. The total embedding matrix is  $E^w \in \mathbb{R}^d * N_w$  in which  $N_w$  is the vocabulary size. Additionally, since word class and higher-order interaction are also important, we assign each part-of-speech tag of a token and each transition label of one configuration's transition choice a vector with same dimension with a word, i.e.,  $e_i^p, e_j^l \in \mathbb{R}^d$ . Therefore, assume  $N_p$  and  $N_l$  are the number of all appeared POS tags and transition labels, total POS tags and transition labels embeddings matrices are  $E^p \in \mathbb{R}^{d \times N_p}$  and  $E^l \in \mathbb{R}^{d \times N_l}$  respectively.

We will choose a set of elements for word, POS tag, transition label respectively based on every configuration's stack, buffer contents. Let them are  $S^w, S^p, S^l$ , and the number of elements in them are  $n^w, n^p, n^l$  separately. Then we have  $x^w = [e_0^w, e_1^w, \dots, e_{n^w}^w]$ ,

$x^p = [e_0^p, e_1^p, \dots, e_{n^p}^p]$ , and  $x^l = [e_0^l, e_1^l, \dots, e_{n^l}^l]$ . As Figure 3 shows, we use  $[x^w, x^p, x^l]$  as our neural network's input layer and use sigmoid as our activation function to map input layer into hidden layer:

$$h = \frac{1}{1 - \exp(-(x^w W^w + x^p W^p + x^l W^l + b_1))}$$

in which  $W^w \in \mathbb{R}^{d \cdot n^w \times d_h}$ ,  $W^p \in \mathbb{R}^{d \cdot n^p \times d_h}$ ,  $W^l \in \mathbb{R}^{d \cdot n^l \times d_h}$  will give us  $d_h$  nodes in our hidden layer.

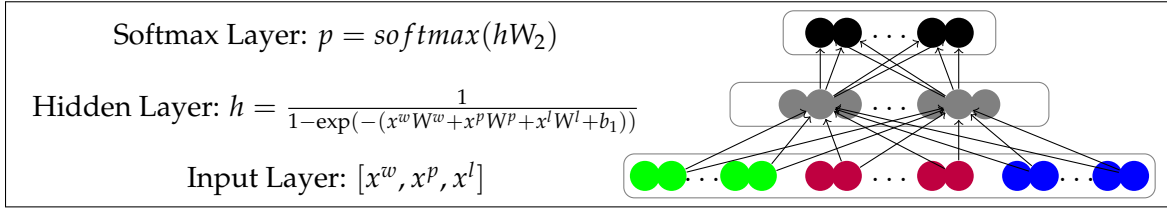


Figure 3. Our neural network with one hidden layer.

At last, we add a softmax layer on the top for calculating possibilities of all  $N_T$  transitions as following:

$$P = \text{softmax}(hW_2)$$

$$P(y = j|h) = \frac{\exp(h \cdot w_j)}{\sum_{k=1}^{N_T} \exp h \cdot w_k} \quad (w_j \text{ is the } j\text{th column in } W_2, 1 < j < N_t)$$

in which  $W_2 \in \mathbb{R}^{d_h \times N_T}$  is the weights for the top layer.

**Set of chosen words** Next thing is to choose a proper set of words from a configuration. Necessarily, since using the left-reduce or right-reduce directly depends on if  $S_0$  is  $S_1$ 's head or  $S_1$  is  $S_0$ 's head, those two words must be included in the set. In order to capture more context information in our features, we also add  $S_2, S_{0l}, S_{0r}, S_{1l}, S_{1r}$  ( $l$  and  $r$  means the leftmost and rightmost children of a token) into our set of chosen words.

In addition to the words in the stack, several top words in the buffer that are near the top words in the buffer play important roles as well. For example, from the arc-standard system’s definition and Figure 2 we could find that it is hard to decide when we can safely do a right-reduce operation unless we know exactly there is no more dependant of  $S_0$  in the buffer. Thus, we will look a little far to the first three tokens ( $B_0, B_1, B_2$ ) in the buffer.

Besides, since higher-order context features have been proved to improve the accuracy of graph-based dependency parsers to improve accuracy (7), we also pick the  $S_{0l2}, S_{0r2}, S_{1l2}, S_{1r2}$  ( $l2$  and  $r2$  denote the second leftmost and rightmost modifier of the element) and  $S_{0ll}, S_{0rr}, S_{1ll}, S_{1rr}$  ( $ll$  and  $rr$  indicate the leftmost modifier of the leftmost modifier and the rightmost modifier of the right modifier of the element).

**POS Tags** After we choose 18 elements for our set of words, we add their POS tags to our feature set, in our experimental data, each token in a sentence has been annotated with its POS tag in Penn Treebank style. So we will have 18 elements in our set of POS tags.

**Transition Labels** In each step’s configuration in Figure 2, we could extract the transition label of each of  $S_{0l}, S_{0r}, S_{1l}, S_{1r}, S_{0l2}, S_{0r2}, S_{1l2}, S_{1r2}, S_{0ll}, S_{0rr}, S_{1ll}, S_{1rr}$  (if exists) from current transition labels set  $A$ . Thus, the size of transition labels set is 12.

**Activation Function** Actually, the objective classes of our prediction are non-linear, as an indispensable part of a neural network, activation function help us to convert the linear combination between weights and input features to non-linear result set. We will use three different activation function *tanh*, *sigmoid*, and *relu* in our hidden layer and compare the effect of them in the experiment phase.

**Training** To train our model, we need feed it a set of configurations and the oracle transition operation choice of each configuration. Table 2 shows a example data of our raw train data. What we do for prepare the input data for training model is implementing a Dependency Parser Simulator which will construct a initial configuration with all tokens in the buffer as Figure 2 shows, and then do a transition operation based on the data.

For example, if we are in the third step’s configuration of Figure 2, we have  $S = [They, hid]$  and we know  $they(1)$ ’s head is  $hid(2)$ . In this configuration, the oracle transition type should be left-reduce. In our experiments, there are in total 78 different type label classes. We will use one hot vector to represent each of them, i.e., the  $i$ th type label class, is a vector that the  $i$ th dimension is 1 and all others is 0.

Another example on this data will bring us a problem, if we are in the 7th step, we have  $S = [hid, letter]$  and  $letter(4)$ ’s head is  $hid(2)$ . However, in this case we cannot execute a right-reduce transition since it will remove  $shelf(7)$ ’s head out of the stack. For solving this problem, our simulator will check if there is any dependants of  $S_0$  in the buffer still, if so, we will prefer a shift operation at first.

Index	Token	POS Tag	Head	Oracle Transition Label
1	They	PRP	2	nsubj
2	hid	VBD	0	root
3	the	DT	4	det
4	letter	NN	2	dobj
5	on	IN	7	case
6	the	DT	7	det
7	shelf	NN	4	nmod

Table 2. An example sentence data in raw form.

By using this simulator, we produce enormous input data and feed them into our model for training the weights. Our training objective is minimize the cross-entropy loss between the oracle transition label and our softmax result:



$$L(\theta) = \frac{1}{N_T} \sum_{n=1}^{N_T} H(p_n, q_n)$$

in which,  $\theta$  is the set of all weights and biases we defined.

**Parsing** Given last step’s training result, for evaluating our dependency parser, we use a way similar with the simulator. We generate features based on every configuration  $c$  we extract from the dev and test data. Then we feed features into the trained classifier and choose the transition type with the best score of  $t = \operatorname{argmax}_{t \in T} h(c)W_t$ , and perform this transition on current configuration to transfer it to next configuration. Meanwhile, if this transition is a left-reduce or right-reduce, we will also get a dependency relation between the top two elements in the stack. However, for the evaluative parser, there is also some hard constraints exists. This will let us do some heuristic transition without consider the classifier’s result. Table 3 lists some heuristics we have.

Condition	Preferred Transition
Stack has less than two elements, and buffer is not empty	Shift
Stack has only one elements, and buffer is empty	Right-reduce, root: $S_0$

Table 3. Constraints of transition types

## Experiments

**Datasets** We use English Penn Treebank (PTB) datasets given by Stanford CS224n course’s assignment2. These datasets are pre-annotated by Penn Treebank’s POS Tags and Stanford’s dependency labels. Our train, dev, and test data has 39832, 1700, and 2416 sentences respectively.

**TensorFlow** TensorFlow is an open source software library which is released by Google. It offer a huge math and machine learning library to support constructing a neural network. In this work, we use python implementing the dependency parser simulator

for extracting features from training data, and build a neural network with one hidden layer with TensorFlow api in python.

**Results** We use embedding size  $d = 50$ , hidden layer size  $h = 200$  for our experiments, and initial learning rate of Adam Optimizer  $r = 0.001$ , and learning rate decay to  $0.96 \times r_{prev}$  after every 5000 steps.

**Components' effectiveness** We test with  $\text{sigmoid}(f(x) = \frac{1}{1+\exp(-x)})$ ,  $\text{tanh}(f(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)})$ ,  $\text{ReLU}(\text{rectifier})(f(x) = \max(0, x))$  (Figure 4 shows 3 different activation functions) in our experiments and compare their effect on our result as Table 4 shows. The result of UAS (unlabeled accuracy) and LAS (unlabeled accuracy) is calculated by using every our parser to extract the dependency tree which is produced by applying every configuration's prediction's transition label on the test and dev data. We could see that usually accuracy of labeled dependency is much less than accuracy of unlabeled data.

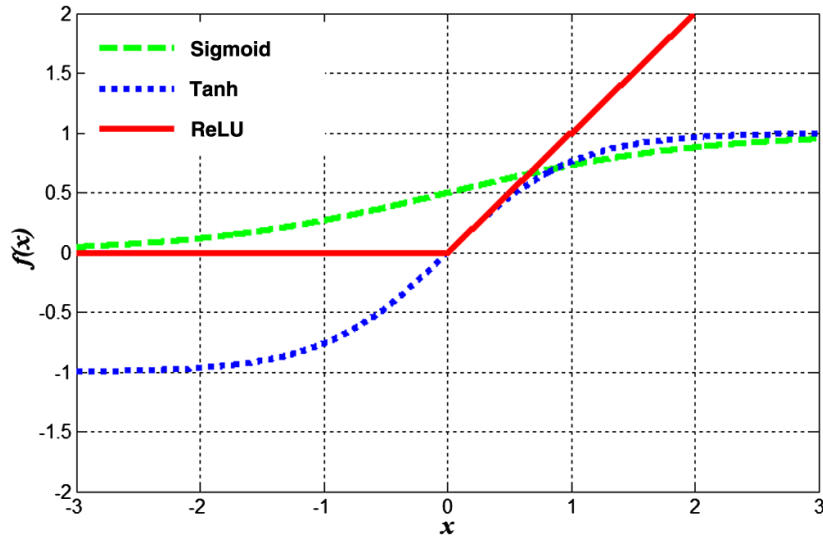


Figure 4. Different Activation Functions

Activation Function	Dev		Test		Features	Dev		Test	
					Word&POS&Label	94.8	86.5	94.7	86.5
Sigmoid	94.8	86.5	94.7	86.5	Word&POS	88.7	64.4	89.0	64.2
Tanh	93.8	85.1	93.8	85.2	Word&Label	86.7	60.8	87.2	60.7
ReLU	95.0	86.8	94.9	86.7	Only Word	80.8	44.0	81.7	43.8

Table 4. Comparison between components and features

**Words POS tags, label embeddings** As table 4 shows, using words+POS tags+transition labels’ effect is about 6.9% and 34% better than using words+POS tags in UAS and LAS. POS tags’ effect is about 2.3% better than label. Using words+POS tags+transition labels’ effect is 17.3% and 96.7% than only using word as features.

## Related Work

We will describe some earlier works of performing dependency parsing with neural network, and their difference from our project here.

A significant difference is that most of early work uses local one-hot word representations but not the distributed representations of modern work. Mayberry III et al. introduced one-hot word representations into their shift reduce constituency parser in 1999 (8) and improve their work in 2005 (9).

In 2004, Henderson used a synchrony network for predicting parse decisions in a constituency parser (10), Titov et al. introduced Incremental Sigmoid Belief Networks into constituency parsing in 2007 (11), and then Garg et al. used a Temporal Restricted Boltzman Machine to extend this work to transition-based dependency parsing (12).

Another two attempts of using neural work for parsing are Collobert did for constituency parsing (13) and Socher et al. built models with dependency representations (14). However, the latter did not actually use this for dependency parsing.

Chen et al. suggested a innovative way what we are reimplementing in this work for learning a neural network classifier for use in a greedy, transition-based dependency parser (1).

Most recently, Weiss et al. used structured training for neural network transition-based parsing on a gold corpus plus a significant number of automatically parsed sentences and reached even better result (15).

## Conclusion

This project give us very precious opportunity to find out what has happened and is happening in the world of Natural Language Processing. In the class, we study about syntactic parsing, constituency parsing, dependency parsing. By trying to finish this project, my understanding of these concept has been improved a lot. Another acquisition is we learned how to extract features from raw data. Now we know the importance of choosing proper features and a fine way of extract them. E.g., by comparing POS tags and transition labels' effect on our experimental result. We know that these features information are both important for a dependency parser and POS tags have a little bit more weight than labels have.

On the other hand, by implementing this neural network classifier, we validate the feasibility of using neural network to involve dense features into dependency parsing, and this model is also able to be trained on more possible feature's conjunctions for getting improvements in the future.

## References and Notes

1. D. Chen, C. Manning, *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)* (2014), pp. 740–750.
2. D. Jurafsky, *Speech & language processing* (Pearson Education India, 2000).
3. S. Kübler, R. McDonald, J. Nivre, *Synthesis Lectures on Human Language Technologies* **1**, 1 (2009).
4. T. Koo, X. Carreras Pérez, M. Collins, *46th Annual Meeting of the Association for Computational Linguistics* (2008), pp. 595–603.
5. J. Nivre, *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together* (Association for Computational Linguistics, 2004), pp. 50–57.
6. Y. Zhang, J. Nivre, *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers-Volume 2* (Association for Computational Linguistics, 2011), pp. 188–193.
7. X. Carreras, *EMNLP-CoNLL* (2007), pp. 957–961.
8. M. R. Mayberry, R. Miikkulainen (1999).
9. M. R. Mayberry, R. Miikkulainen, *Neural Processing Letters* **21**, 121 (2005).
10. J. Henderson, *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics* (Association for Computational Linguistics, 2004), p. 95.
11. I. Titov, J. Henderson, *ANNUAL MEETING-ASSOCIATION FOR COMPUTATIONAL LINGUISTICS* (2007), vol. 45, p. 632.

12. N. Garg, J. Henderson, *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers-Volume 2* (Association for Computational Linguistics, 2011), pp. 11–17.
13. R. Collobert, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (2011), pp. 224–232.
14. R. Socher, A. Karpathy, Q. V. Le, C. D. Manning, A. Y. Ng, *Transactions of the Association for Computational Linguistics* **2**, 207 (2014).
15. D. Weiss, C. Alberti, M. Collins, S. Petrov, *arXiv preprint arXiv:1506.06158* (2015).