Transition-based Neural Constituent Parsing

Taro Watanabe[*] and **Eiichiro Sumita**
National Institute of Information and Communications Technology
3-5 Hikaridai, Seika-cho, Soraku-gun, Kyoto, 619-0289 JAPAN
tarow@google.com, eiichiro.sumita@nict.go.jp

Abstract

Constituent parsing is typically modeled by a chart-based algorithm under probabilistic context-free grammars or by a transition-based algorithm with rich features. Previous models rely heavily on richer syntactic information through lexicalizing rules, splitting categories, or memorizing long histories. However enriched models incur numerous parameters and sparsity issues, and are insufficient for capturing various syntactic phenomena. We propose a neural network structure that explicitly models the unbounded history of actions performed on the stack and queue employed in transition-based parsing, in addition to the representations of partially parsed tree structure. Our transition-based neural constituent parsing achieves performance comparable to the state-of-the-art parsers, demonstrating F1 score of 90.68% for English and 84.33% for Chinese, without reranking, feature templates or additional data to train model parameters.

1 Introduction

A popular parsing algorithm is a cubic time chart-based dynamic programming algorithm that uses probabilistic context-free grammars (PCFGs). However, PCFGs learned from treebanks are too coarse to represent the syntactic structures of texts. To address this problem, various contexts are incorporated into the grammars through lexicalization (Collins, 2003; Charniak, 2000) or category splitting either manually (Klein and Manning, 2003) or automatically (Matsuzaki et al., 2005; Petrov et al., 2006). Recently a rich feature set was introduced to capture the lexical contexts in each span without extra annotations in grammars (Hall et al., 2014).

Alternatively, transition-based algorithms run in linear time by taking a series of shift-reduce actions with richer lexicalized features considering histories; however, the accuracies did not match with the state-of-the-art methods until recently (Sagae and Lavie, 2005; Zhang and Clark, 2009). Zhu et al. (2013) show that the use of better transition actions considering unaries and a set of non-local features can compete with the accuracies of chart-based parsing. The features employed in a transition-based algorithm usually require part of speech (POS) annotation in the input, but the delayed feature technique allows joint POS inference (Wang and Xue, 2014).

In both frameworks, the richer models require that more parameters be estimated during training which can easily result in the data sparseness problems. Furthermore, the enriched models are still insufficient to capture various syntactic relations in texts due to the limited contexts represented in latent annotations or non-local features. Recently Socher et al. (2013) introduced compositional vector grammar (CVG) to address the above limitations. However, they employ reranking over a forest generated by a baseline parser for efficient search, because CVG is built on cubic time chart-based parsing.

In this paper, we propose a neural network-based parser — *transition-based neural constituent parsing (TNCP)* — which can guarantee efficient search naturally. TNCP explicitly models the actions performed on the stack and queue employed in transition-based parsing. More specifically, the queue is modeled by recurrent neural network (RNN) or Elman network (Elman, 1990) in backward direction (Henderson, 2004). The stack structure is also modeled similarly to RNNs, and its top item is updated using the previously constructed hidden representations saved in the

[*] The first author is now affiliated with Google, Japan.

stack. The representations from both the stack and queue are combined with the representations propagated from the partially parsed tree structure inspired by the recursive neural networks of CVGs. Parameters are estimated efficiently by a variant of max-violation (Huang et al., 2012) which considers the worst mistakes found during search and updates parameters based on the expected mistake.

Under similar settings, TCNP performs comparably to state-of-the-art parsers. Experimental results obtained using the Wall Street Journal corpus of the English Penn Treebank achieved a labeled F1 score of 90.68%, and the result for the Penn Chinese Treebank was 84.33%. Our parser performs no reranking with computationally expensive models, employs no templates for feature engineering, and requires no additional monolingual data for reliable parameter estimation. The source code and models will be made public[1].

2 Related Work

Our study is largely inspired by recursive neural networks for parsing, first pioneered by Costa et al. (2003), in which parsing is treated as a ranking problem of finding phrasal attachment. Such network structures have been used successfully as a reranker for k-best parses from a baseline parser (Menchetti et al., 2005) or parse forests (Socher et al., 2013), and have achieved gains on large data. Stenetorp (2013) showed that the recursive neural networks are comparable to the state-of-the-art system with a rich feature set under dependency parsing. Our model is not a reranking model, but a discriminative parsing model, which incorporates the representations of stacks and queues employed in the transition-based parsing framework, in addition to the representations of the tree structures. The use of representations outside of the partial parsed trees is very similar to the recently proposed inside-outside recursive neural networks (Le and Zuidema, 2014) which can assign probabilities in a top-down manner, in the same way as PCFGs.

Henderson (2003) was the first to demonstrate the successful use of neural networks to represent derivation histories under large-scale parsing experiments. He employed synchrony networks, i.e., feed-forward style networks, to assign a probability for each step in the left-corner parsing conditioning on all parsing steps. Henderson (2004) later employed a discriminative model and showed further gains by conditioning on the representation of the future input in addition to the history of parsing steps. Similar feed-forward style networks are successfully applied for transition-based dependency parsing in which limited contexts are considered in the feature representation (Chen and Manning, 2014). Our model is very similar in that the score of each action is computed by conditioning on all previous actions and future input in the queue.

The use of neural networks for transition-based shift-reduce parsing was first presented by Mayberry and Miikkulainen (1999) in which the stack representation was treated as a hidden state of an RNN. In their study, the hidden state is updated recurrently by either a shift or reduce action, and its corresponding parse tree is decoded recursively from the hidden state (Berg, 1992) using recursive auto-associative memories (Pollack, 1990). We apply the idea of representing a stack in a continuous vector; however, our method differs in that it memorizes all hidden states pushed to the stack and performs push/pop operations. In this manner, we can represent the local contexts saved in the stack explicitly and use them to construct new hidden states.

3 Transition-based Constituent Parsing

Our transition-based parser is based on a study by Zhu et al. (2013), which adopts the shift-reduce parsing of Sagae and Lavie (2005) and Zhang and Clark (2009). However, our parser differs in that we do not differentiate left or right head words. In addition, POS tags are jointly induced during parsing in the same manner as Wang and Xue (2014). Given an input sentence $w_0, \cdots, w_{n-1}$, the transition-based parser employs a stack of partially constructed constituent tree structures and a queue of input words. In each step, a transition action is applied to a state $\langle i, f, S \rangle$, where i is the next input word position in the queue w_i, f is a flag indicating the completion of parsing, i.e., whether the *ROOT* of a constituent tree covering all the input words is reached, and S represents a stack of tree elements, $s_0, s_1, \cdots$.

The parser consists of five actions:

shift-*X* consumes the next input word, w_i, from the queue and pushes a non-terminal symbol (or a POS label) as a tree of $X \to w_i$.

[1]`http://github.com/tarowatanabe/trance`

$$\begin{array}{rl}
\text{axiom} & 0 : \langle 0, \text{false}, \langle \text{eps} \rangle \rangle : 0 \\
\text{goal} & (2+u)n : \langle n, \text{true}, S \rangle : \rho \\
\\
\text{shift-}X & \dfrac{j : \langle i, \text{false}, S \rangle : \rho}{j+1 : \langle i+1, \text{false}, S|X \rangle : \rho + \rho_{\mathsf{sh}}} \\
\text{reduce-}X & \dfrac{j : \langle i, \text{false}, S|s_1|s_0 \rangle : \rho}{j+1 : \langle i, \text{false}, S|X \rangle : \rho + \rho_{\mathsf{re}}} \\
\text{unary-}X & \dfrac{j : \langle i, \text{false}, S|s_0 \rangle : \rho}{j+1 : \langle i, \text{false}, S|X \rangle : \rho + \rho_{\mathsf{un}}} \\
\text{finish} & \dfrac{j : \langle n, \text{false}, S \rangle : \rho}{j+1 : \langle n, \text{true}, S \rangle : \rho + \rho_{\mathsf{fi}}} \\
\text{idle} & \dfrac{j : \langle n, \text{true}, S \rangle : \rho}{j+1 : \langle n, \text{true}, S \rangle : \rho + \rho_{\mathsf{id}}}
\end{array}$$

Figure 1: Deduction system for shift-reduce parsing, where j is a step size and ρ is a score.

reduce-X pops the top two items s_0 and s_1 out of the stack and combines them as a partial tree with the constituent label X as its root, and with s_0 and s_1 as right and left antecedents, respectively ($X \rightarrow s_1 s_0$). The newly created tree is then pushed into the stack.

unary-X is similar to reduce-X; however, it consumes only the top most item s_0 from the stack and pushes a new tree of $X \rightarrow s_0$.

finish indicates the completion of parsing, i.e., reaching the $ROOT$.

idle preserves completion until the goal is reached.

The whole procedure is summarized as a deduction system in Figure 1. We employ beam search which starts from an axiom consisting of a stack with a special symbol ⟨eps⟩, and ends when we reach a goal item (Zhang and Clark, 2009). A set of agenda $\boldsymbol{B} = B_0, B_1, \cdots$ maintains the k-best states for each step j at B_j, which is first initialized by inserting the axiom in B_0. Then, at each step $j = 0, 1, \cdots$, every state in the agenda B_j is extended by applying one of the actions and the new states are inserted into the agenda B_{j+1} for the next step, which retains only the k-best states.

We limit the maximum number of consecutive unary actions to u (Sagae and Lavie, 2005; Zhang and Clark, 2009) and the maximum number of unary actions in a single derivation to $u \times n$. Thus, the process is repeated until we reach the final step of $(2+u)n$, which keeps the completed states. The idle action is inspired by the padding method of Zhu et al. (2013), such that the states in an agenda are comparable in terms of score even if differences exist in the number of unary actions. Unlike Zhu et al. (2013) we do not terminate parsing even if all the states in an agenda are completed ($f =$ true).

The score of a state is computed by summing the scores of all the actions leading to the state. In Figure 1, ρ_{sh}, ρ_{re}, ρ_{un}, ρ_{fi} and ρ_{id} are the scores of shift-X, reduce-X, unary-X, finish and idle actions, respectively, which are computed on the basis of the history of actions.

4 Neural Constituent Parsing

The score of a state is defined formally as the total score of transition actions, or a (partial) derivation $\boldsymbol{d} = d_0, d_1, \cdots$ leading to the state as follows:

$$\rho(\boldsymbol{d}) = \sum_{j=0}^{|\boldsymbol{d}|-1} \rho(d_j | d_0^{j-1}). \tag{1}$$

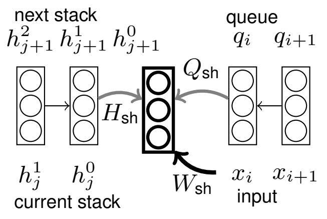
Note that the score of each action is dependent on all previous actions. In previous studies, the score is computed by a linear model, i.e., a weighted sum of feature values derived from limited histories, such as those that consider two adjacent constituent trees in a stack (Sagae and Lavie, 2005; Zhang and Clark, 2009; Zhu et al., 2013). Our method employs an RNN or Elman network (Elman, 1990) to represent an unlimited stack and queue history.

Formally, we use an m-dimensional vector for each hidden state unless otherwise stated. Here, let $x_i \in \mathbb{R}^{m' \times 1}$ be an m'-dimensional vector representing the input word w_i and the dimension may not match with the hidden state size m. $q_i \in \mathbb{R}^{m \times 1}$ denotes the hidden state for the input word w_i in a queue. Following the RNN in backward direction (Henderson, 2004), the hidden state for each word w_i is computed right-to-left, q_{n-1} to q_0, beginning from a constant q_n:

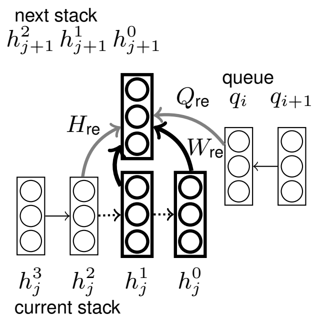
$$q_i = \tau \left(H_{\mathsf{qu}} q_{i+1} + W_{\mathsf{qu}} x_i + b_{\mathsf{qu}} \right), \tag{2}$$

where $H_{\mathsf{qu}} \in \mathbb{R}^{m \times m}$, $W_{\mathsf{qu}} \in \mathbb{R}^{m \times m'}$, $b_{\mathsf{qu}} \in \mathbb{R}^{m \times 1}$ and $\tau(x)$ is hard-tanh applied element-wise[2].

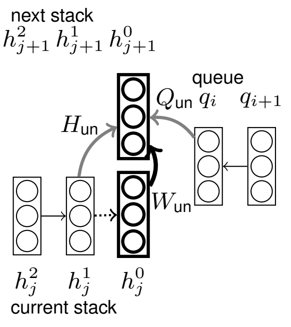
[2] $\tau(x) = -1$ for $x < 1$, 1 for $x > 1$ otherwise x.



(a) shift-X action


next stack
h^2_{j+1} h^1_{j+1} h^0_{j+1}
queue
Q_{re} q_i q_{i+1}
H_{re}
W_{re}
h^3_j h^2_j h^1_j h^0_j
current stack

(b) reduce-X action



(c) unary-X action

Figure 2: Example neural network for constituent parsing. The thick arrows indicate the context of tree structures, and the gray arrows represent interactions from the stack and queue. The dotted arrows denote popped states.

Shift: Now, let $h^l_j \in \mathbb{R}^{m\times 1}$ represent a hidden state associated with the lth stack item for the jth action. We define the score of a shift action:

$$h^0_{j+1} = \tau\left(H^X_{\text{sh}} h^0_j + Q^X_{\text{sh}} q_i + W^X_{\text{sh}} x_i + b^X_{\text{sh}}\right) \quad (3)$$

$$\rho(d_j = \text{shift-}X | d^{j-1}_0) = V^X_{\text{sh}} h^0_{j+1} + v^X_{\text{sh}} \quad (4)$$

where $H^X_{\text{sh}} \in \mathbb{R}^{m\times m}$, $Q^X_{\text{sh}} \in \mathbb{R}^{m\times m}$, $W^X_{\text{sh}} \in \mathbb{R}^{m\times m'}$ and $b^X_{\text{sh}} \in \mathbb{R}^{m\times 1}$. Figure 2(a) shows the network structure for Equation 3. H^X_{sh} represents an RNN-style architecture that propagates the previous context in the stack. Q^X_{sh} can reflect the queue context q_i, or the future input sequence from w_i through w_{n-1}, while W^X_{sh} directly expresses the leaf of a tree structure using the shifted input word representation x_i for w_i. The hidden state h^0_{j+1} is used to compute the score of a derivation $\rho(d_j|d^{j-1}_0)$ in Equation 4, which is based on the matrix $V^X_{\text{sh}} \in \mathbb{R}^{1\times m}$ and the bias term $v^X_{\text{sh}} \in \mathbb{R}$. Note that $h^l_{j+1} = h^{l-1}_j$ for $l = 1, 2, \cdots$ because the stack is updated by the newly created partial tree label X associated with the new hidden state h^0_{j+1}.

Inspired by CVG (Socher et al., 2013), we differentiate the matrices for each non-terminal (or POS) label X rather than using shared parameters. However, our model differs in that the parameters are untied on the basis of the left hand side of a rule, rather than the right hand side, because our model assigns a score discriminatively for each action with the left hand side label X unlike a generative model derived from PCFGs.

Reduce: Similarly, the score for a reduce action is obtained as follows:

$$h^0_{j+1} = \tau\left(H^X_{\text{re}} h^2_j + Q^X_{\text{re}} q_i + W^X_{\text{re}} h^{[0:1]}_j + b^X_{\text{re}}\right) \quad (5)$$

$$\rho(d_j = \text{reduce-}X | d^{j-1}_0) = V^X_{\text{re}} h^0_{j+1} + v^X_{\text{re}}, \quad (6)$$

where $H^X_{\text{re}} \in \mathbb{R}^{m\times m}$, $Q^X_{\text{re}} \in \mathbb{R}^{m\times m}$, $W^X_{\text{re}} \in \mathbb{R}^{m\times 2m}$, $b^X_{\text{re}} \in \mathbb{R}^{m\times 1}$, and $h^{[l:l']}$ denotes the vertical matrix concatenation of hidden states from h^l to $h^{l'}$.

Note that the reduce-X action pops top two items in the stack that correspond to the two hidden states of $h^{[0:1]}_j$ as represented by Figure 2(b). By pushing a newly created tree with the constituent X, its corresponding hidden state h^0_{j+1} is pushed to the stack with each remaining hidden state $h^l_{j+1} = h^{l+1}_j$ for $l = 1, 2, \cdots$. The hidden state of the top stack item h^0_j is a representation of the right antecedent of a newly created binary tree with h^0_{j+1} as a root, while the hidden state of the next top stack item h^1_j corresponds to the left antecedent of the binary tree. Thus, the two hidden states capture the recursive neural network-like structure (Costa et al., 2003), while $h^2_j = h^1_{j+1}$ represents the RNN-like linear history in the stack.

Unary: In the same manner as the reduce action, the unary action is defined by simply reducing a single item from a stack and by pushing a new item (Figure 2(c)):

$$h^0_{j+1} = \tau\left(H^X_{\text{un}} h^1_j + Q^X_{\text{un}} q_i + W^X_{\text{un}} h^0_j + b^X_{\text{un}}\right) \quad (7)$$

$$\rho(d_j = \text{unary-}X | d^{j-1}_0) = V^X_{\text{un}} h^0_{j+1} + v^X_{\text{un}}, \quad (8)$$

where $H^X_{\text{un}} \in \mathbb{R}^{m\times m}$, $Q^X_{\text{un}} \in \mathbb{R}^{m\times m}$, $W^X_{\text{un}} \in \mathbb{R}^{m\times m}$ and $b^X_{\text{un}} \in \mathbb{R}^{m\times 1}$. Note that $h^l_{j+1} = h^l_j$ for $l = 1, 2, \cdots$, because only the top item is updated in the stack by creating a partial tree with h^0_j together with the stack history h^1_j.

In summary, the number of model parameters for the three actions is $9\times m^2 + m\times m' + 6\times m + 3$ for each non-terminal label X. The scores for a finish action and an idle action are defined analogous to the unary-X action with special labels for X, $\langle$finish$\rangle$ and $\langle$idle$\rangle$, respectively[3].

5 Parameter Estimation

Let $\boldsymbol{\theta} = \left\{H^X_{\mathsf{sh}}, Q^X_{\mathsf{sh}}, \cdots\right\} \in \mathbb{R}^M$ be an M-dimensional vector of all model parameters. The parameters are initialized randomly by following Glorot and Bengio (2010), in which the random value range is determined by the size of the input/output layers. The bias parameters are initialized to zeros.

We employ a variant of max-violation (Huang et al., 2012) as our training objective, in which parameters are updated based on the worst mistake found during search, rather than the first mistake as performed in the early update perceptron algorithm (Collins and Roark, 2004). Specifically, given a training instance $(\boldsymbol{w}, \boldsymbol{y})$ where $\boldsymbol{w}$ is an input sentence and $\boldsymbol{y}$ is its gold derivation, i.e., a sequence of actions representing the gold parse tree for $\boldsymbol{w}$, we seek for the step j^* where the difference of the scores is the largest:

$$j^* = \arg\min_{j} \left\{ \rho_{\boldsymbol{\theta}}(y^j_0) - \max_{\boldsymbol{d} \in B_j} \rho_{\boldsymbol{\theta}}(\boldsymbol{d}) \right\}. \quad (9)$$

Then, we define the following hinge-loss function:

$$L(\boldsymbol{w}, \boldsymbol{y}; \boldsymbol{B}, \boldsymbol{\theta}) = \max\left\{0, 1 - \rho_{\boldsymbol{\theta}}(y^{j^*}_0) + \mathbb{E}_{\tilde{B}_{j^*}}[\rho_{\boldsymbol{\theta}}]\right\}, \quad (10)$$

wherein we consider the subset of sub-derivations $\tilde{B}_{j^*} \subset B_{j^*}$ consisting of those scored higher than $\rho_{\boldsymbol{\theta}}(y^{j^*}_0)$:

$$\tilde{B}_{j^*} = \left\{ \boldsymbol{d} \in B_{j^*} \middle| \rho_{\boldsymbol{\theta}}(\boldsymbol{d}) > \rho_{\boldsymbol{\theta}}(y^{j^*}_0) \right\} \quad (11)$$

$$p_{\boldsymbol{\theta}}(\boldsymbol{d}) = \frac{\exp(\rho_{\boldsymbol{\theta}}(\boldsymbol{d}))}{\sum_{\boldsymbol{d}' \in \tilde{B}_{j^*}} \exp(\rho_{\boldsymbol{\theta}}(\boldsymbol{d}'))} \quad (12)$$

$$\mathbb{E}_{\tilde{B}_{j^*}}[\rho_{\boldsymbol{\theta}}] = \sum_{\boldsymbol{d} \in \tilde{B}_{j^*}} p_{\boldsymbol{\theta}}(\boldsymbol{d}) \rho_{\boldsymbol{\theta}}(\boldsymbol{d}). \quad (13)$$

Unlike Huang et al. (2012) and inspired by Tamura et al. (2014), we consider all incorrect sub-derivations found in $\tilde{B}_{j^*}$ through the expected score $\mathbb{E}_{\tilde{B}_{j^*}}[\rho_{\boldsymbol{\theta}}]$[4]. The loss function in Equation 10 can be intuitively considered an expected mistake suffered at the maximum violated step j^*, which is measured by the Viterbi violation in Equation 9. Note that if we replace $\mathbb{E}_{\tilde{B}_{j^*}}[\rho_{\boldsymbol{\theta}}]$ with $\max_{\boldsymbol{d} \in B_{j^*}} \rho_{\boldsymbol{\theta}}(\boldsymbol{d})$ in Equation 10, it is exactly the same as the max-violation objective (Huang et al., 2012)[5].

To minimize the loss function, we use a diagonal version of AdaDec (Senior et al., 2013) — a variant of diagonal AdaGrad (Duchi et al., 2011) — under mini-batch settings. Given the sub-gradient $\boldsymbol{g}_t \in \mathbb{R}^M$ of Equation 10 at time t computed by the back-propagation through structure (Goller and Küchler, 1996), we maintain additional parameters $\boldsymbol{G}_t \in \mathbb{R}^M$:

$$\boldsymbol{G}_t \leftarrow \gamma \boldsymbol{G}_{t-1} + \boldsymbol{g}_t \odot \boldsymbol{g}_t, \quad (14)$$

where $\odot$ is the Hadamard product (or the element-wise product). $\boldsymbol{\theta}_{t-1}$ is updated using the element specific learning rate $\boldsymbol{\eta}_t \in \mathbb{R}^M$ derived from $\boldsymbol{G}_t$ and a constant $\eta_0 > 0$:

$$\boldsymbol{\eta}_t \leftarrow \eta_0 \left(\boldsymbol{G}_t + \epsilon\right)^{-\frac{1}{2}} \quad (15)$$

$$\boldsymbol{\theta}_{t-\frac{1}{2}} \leftarrow \boldsymbol{\theta}_{t-1} - \boldsymbol{\eta}_t \odot \boldsymbol{g}_t \quad (16)$$

$$\boldsymbol{\theta}_t \leftarrow \arg\min_{\boldsymbol{\theta}} \frac{1}{2} \|\boldsymbol{\theta} - \boldsymbol{\theta}_{t-\frac{1}{2}}\|^2_2 + \lambda \boldsymbol{\eta}_t^\top \operatorname{abs}(\boldsymbol{\theta}). \quad (17)$$

Compared with AdaGrad, the squared sum of the sub-gradients decays over time using a constant $0 < \gamma \leq 1$ in Equation 14. The learning rate in Equation 15 is computed element-wise and bounded by a constant $\epsilon \geq 0$, and if we set $\epsilon \geq \eta_0^2$, it is always decayed[6]. In our preliminary studies, AdaGrad eventually becomes very conservative to update parameters when training longer iterations. AdaDec fixes the problem by ignoring older histories of sub-gradients in $\boldsymbol{G}$, which is reflected in the learning rate $\boldsymbol{\eta}$. In each update, we employ ℓ_1 regularization through FOBOS (Duchi and Singer, 2009) using a hyperparameter $\lambda \geq 0$ to control the fitness in Equation 16 and 17. For testing, we found that taking the average of the parameters over period $\frac{1}{T+1}\sum_{t=0}^{T} \boldsymbol{\theta}_t$ under training iterations T was very effective as demonstrated by Hashimoto et al. (2013).

Parameter estimation is performed in parallel by distributing training instances asynchronously

[3]Since h^1_j and q_n are constants for the finish and idle actions, we enforce $H^X_{\mathsf{un}} = 0$ and $Q^X_{\mathsf{un}} = 0$ for those special actions.

[4]We can use all the sub-derivations in B_{j^*}; however, our preliminary studies indicated that the use of $\tilde{B}_{j^*}$ was better.

[5]Or, setting $p_{\boldsymbol{\theta}}(\boldsymbol{d}^*) = 1$ for the Viterbi derivation $\boldsymbol{d}^* = \arg\max_{\boldsymbol{d} \in B_{j^*}} \rho_{\boldsymbol{\theta}}(\boldsymbol{d})$ and zero otherwise.

[6]Note that AdaGrad is a special case of AdaDec with $\gamma = 1$ and $\epsilon = 0$.

in each shard and by updating locally copied parameters using the sub-gradients computed from the distributed mini-batches (Dean et al., 2012). The sub-gradients are broadcast asynchronously to other shards to reflect the updates in one shard. Unlike Dean et al. (2012), we do not keep a central storage for model parameters; the replicated parameters are synchronized in each iteration by choosing the model parameters from one of the shards with respect to the minimum of ℓ_1 norm[7]. Note that we synchronize $\boldsymbol{\theta}$, but $\boldsymbol{G}$ is maintained as shard local parameters.

6 Experiments

6.1 Settings

We conducted experiments for transition-based neural constituent parsing (TNCP) for two languages — English and Chinese. English data were derived from the Wall Street Journal (WSJ) of the Penn Treebank (Marcus et al., 1993), from which sections 2-21 were used for training, 22 for development and 23 for testing. Chinese data were extracted from the Penn Chinese Treebank (CTB) (Xue et al., 2005); articles 001-270 and 440-1151 were used for training, 301-325 for development, and 271-300 for testing. Inspired by jackknifing (Collins and Koo, 2005), we reassigned POS tags for training data using the Stanford tagger (Toutanova et al., 2003)[8]. The treebank trees were normalized by removing empty nodes and unary rules with X over X (or $X \rightarrow X$), then binarized in a left-branched manner.

The possible actions taken for our shift-reduce parsing, e.g., $X \rightarrow w$ in shift-X, were learned from the normalized treebank trees. The words that occurred twice or less were handled differently in order to consider OOVs for testing: They were simply mapped to a special token $\langle$unk$\rangle$ when looking up their corresponding word representation vector. Similarly, when assigning possible POS tags in shift actions, they fell back to their corresponding "word signature" in the same manner as the Berkeley parser[9]. A maximum number of consecutive unary actions was set to $u = 3$ for WSJ and $u = 4$ for CTB, as determined by the treebanks.

	rep. size	32	64	128	256	512	1024
dev	WSJ-32	89.91	90.15	90.48	90.70	90.75	**90.87**
	64	90.37	90.73	90.81	90.62	90.71	**91.11**
	CTB-32	79.25	81.59	82.80	82.68	84.17	**85.12**
	64	84.04	83.29	82.92	85.12	85.24	**85.77**
test	WSJ-32	89.03	89.49	89.75	**90.45**	90.37	90.01
	64	89.74	90.16	90.48	90.06	89.91	**90.68**
	CTB-32	75.19	78.29	80.46	81.87	**83.16**	82.64
	64	80.11	81.35	81.67	82.91	83.76	**84.33**

Table 1: Comparison of various state/word representation dimension size measured by labeled F1(%). "-32" denotes the hidden state size $m = 32$. The numbers in bold indicate the best results for each hidden state dimension.

Parameter estimation was performed on 16 cores of a Xeon E5-2680 2.7GHz CPU. It took approximately one day for 100 training iterations with $m = 32$ and $m' = 128$ under a mini-batch size of 4 and a beam size of 32. Doubling either one of m or m' incurred approximately double training time. We chose the following hyperparameters by tuning toward the development data in our preliminary experiments[10]: $\eta_0 = 10^{-2}$, $\gamma = 0.9$, $\epsilon = 1$. The choice of λ from $\{10^{-5}, 10^{-6}, 10^{-7}\}$ and the number of training iterations were very important for different training objectives and models in order to avoid overfitting. Thus, they were determined by the performance on the development data for each different training objective and/or network configuration, e.g., the dimension for a hidden state. The word representations were initialized by a tool developed in-house for an RNN language model (Mikolov et al., 2010) trained by noise contrastive estimation (Mnih and Teh, 2012). Note that the word representations for initialization were learned from the given training data, not from additional unannotated data as done by Chen and Manning (2014).

Testing was performed using a beam size of 64 with a Xeon X5550 2.67GHz CPU. All results were measured by the labeled bracketing metric PARSEVAL (Black et al., 1991) using EVALB[11] after debinarization.

6.2 Results

Table 1 shows the impact of dimensions on the parsing performance. We varied the hid-

[7]We also tried averaging among shards. However we observed no gains likely because we performed averaging for testing.

[8]`http://nlp.stanford.edu/software/tagger.shtml`

[9]`https://code.google.com/p/berkeleyparser/`

[10]We confirmed that this hyperparameter setting was appropriate for different models experimented in Section 6.2 through our preliminary studies.

[11]`http://nlp.cs.nyu.edu/evalb/`

	model	tree	+stack	+queue
dev	WSJ	77.70	90.54	**91.11**
dev	CTB	69.74	84.70	**85.77**
test	WSJ	76.48	90.00	**90.68**
test	CTB	66.03	82.85	**84.33**

Table 2: Comparison of network structures measured by labeled F1(%).

	loss	Viterbi	expected
dev	WSJ	90.89	**91.11**
dev	CTB	84.94	**85.77**
test	WSJ	90.21	**90.68**
test	CTB	82.62	**84.33**

Table 3: Comparison of loss functions measured by labeled F1(%).

den vector size $m = \{32, 64\}$ and the word representation (embedding) vector size $m' = \{32, 64, 128, 256, 512, 1024\}$[12]. As can be seen, the greater word representation dimensions are generally helpful for both WSJ and CTB on the closed development data (*dev*), which may match with our intuition that the richer syntactic and semantic knowledge representation for each word is required for parsing. However, overfitting was observed when using a 32-dimension hidden vector in both tasks, i.e., drops of performance on the open test data (*test*) when $m' = 1024$, probably caused by the limited generalization capability in the smaller hidden state size. In the rest of this paper, we show the results with $m = 64$ and $m' = 1024$ as determined by the performance on the development data, wherein we achieved 91.11% and 85.77% labeled F1 for WSJ and CTB, respectively. The total number of parameters were approximately 28.3M and 22.0M for WSJ and CTB, respectively, among which 17.8M and 13.4M were occupied for word representations, respectively.

Table 2 differentiated the network structure. The *tree* model computes the new hidden state h^0_{j+1} using only the recursively constructed network by ignoring parameters from the stack and queue, e.g., by enforcing $H^X_{\mathsf{sh}} = 0$ and $Q^X_{\mathsf{sh}} = 0$ in Equation 3, which is essentially similar to the CVG approach (Socher et al., 2013). Adding the context from the stack in *+stack* boosts the performance significantly. Further gains are observed when the queue context *+queue* is incorporated in the model. These results clearly indicate that explicit representations of the stack and queue are very important when applying a recursive neural network model for transition-based parsing.

We then compared the expected mistake with the Viterbi mistake (Huang et al., 2012) as our training objective by replacing $\mathbb{E}_{\tilde{B}_{j*}}[\rho_{\boldsymbol{\theta}}]$ with $\max_{\boldsymbol{d} \in B_{j*}} \rho_{\boldsymbol{\theta}}(\boldsymbol{d})$ in Equation 10. Table 3 shows that the use of the expected mistake (*expected*) as a loss function is significantly better than that of the Viterbi mistake (*Viterbi*) by considering all the incorrect sub-derivations at maximum violated steps during search. Figure 3 and 4 plot the training curves for WSJ and CTB, respectively. The plots clearly demonstrate that the use of the expected mistake is faster in convergence and stabler in learning when compared with that of the Viterbi mistake[13].

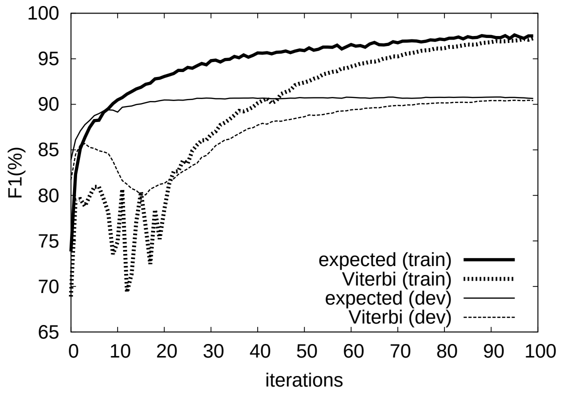


Figure 3: Plots for training iterations and labeled F1(%) on WSJ.

Next, we compare our parser, TNCP, with other parsers listed in Table 4 for WSJ and Table 5 for CTB on the test data. The Collins parser (Collins, 1997) and the Berkeley parser (Petrov and Klein, 2007) are chart-based parsers with rich states, either through lexicalization or latent annotation. SSN is a left-corner parser (Henderson, 2004), and CVG is a compositional vector grammar-based parser (Socher et al., 2013)[14]. Both parsers rely on neural networks to represent rich contexts, similar to our work; however they differ in that they essentially perform reranking from either the k-best parses or parse forests[15]. The word representa-

[12]We experimented larger dimensions in Appendix A.

[13]The labeled F1 on those plots are slightly different from EVALB in that all the syntactic labels are considered when computing bracket matching. Further, the scores on the training data are approximation since they were obtained as a by-product of online learning.

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

[15]Strictly speaking, SSN can work as a standalone parser; Table 4 shows the result after reranking (Henderson, 2004).

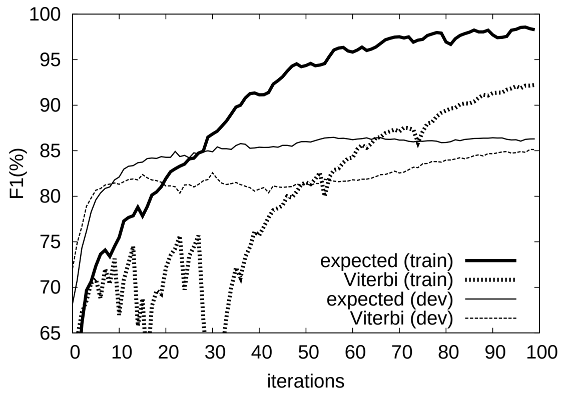


Figure 4: Plots for training iterations and labeled F1(%) on CTB.

parser	test
Collins (Collins, 1997)	87.8
Berkeley (Petrov and Klein, 2007)	90.1
SSN (Henderson, 2004)	90.1
ZPar (Zhu et al., 2013)	90.4
CVG (Socher et al., 2013)	90.4
Charniak-R (Charniak and Johnson, 2005)	**91.0**
This work: TNCP	90.7

Table 4: Comparison of different parsers on the WSJ test data measured by labeled F1(%).

tion in CVG was learned from large monolingual data (Turian et al., 2010), but our parser learns word representation from only the provided training data. Charniak-R is a discriminative reranking parser with non-local features (Charniak and Johnson, 2005). ZPar is a transition-based shift-reduce parser (Zhu et al., 2013)[16] that influences the deduction system in Figure 1, but differs in that scores are computed by a large number of features and POS tagging is performed separately. The results shown in Table 4 and 5 come from the feature set without extra data, i.e., semi-supervised features. Joint is the joint POS tagging and transition-based parsing with non-local features (Wang and Xue, 2014). Similar to ZPar, we present the result without cluster features learned from extra unannotated data.

Finally, we measured the speed for parsing by varying beam size and hidden dimension (Table 6). When testing, we applied a pre-computation technique for layers involving word representation vectors (Devlin et al., 2014), i.e., W_{qu} in Equation 2 and W^X_{sh} in Equation 3. Thus, the parsing speed was influenced by only the hidden state size m. It is clear that the enlarged beam size improves performance by trading off run time in most cases. Note that Berkeley, CVG and ZPar took 4.74, 1.54 and 37.92 sentences/sec, respectively, with WSJ. Although it is more difficult to compare with other parsers, our parser implemented in C++ is on par with Java implementations of Berkeley and CVG. The large run time difference with the C++ implemented ZPar may come from the network computation and joint POS inference in our model which impact parsing speed significantly.

[16] `http://sourceforge.net/projects/zpar/`

parser	test
ZPar (Zhu et al., 2013)	83.2
Berkeley (Petrov and Klein, 2007)	83.3
Joint (Wang and Xue, 2014)	**84.9**
This work: TNCP	84.3

Table 5: Comparison of different parsers on the CTB test data measured by labeled F1(%).

beam	32	64	128
WSJ-32	15.42/89.95	7.90/90.01	3.97/**90.04**
64	7.31/90.56	3.56/90.68	1.76/**90.73**
CTB-32	13.67/82.35	6.95/82.64	3.68/**82.84**
64	6.15/84.12	3.11/**84.33**	1.53/83.83

Table 6: Comparison of parsing speed by varying beam size and hidden dimension; each cell shows the number of sentences per second/labeled F1(%) measured on the test data.

6.3 Error Analysis

To assess parser error types, we used the tool proposed by Kummerfeld et al. (2012)[17]. The average number of errors per sentence is listed in Table 7 for each error type on the WSJ test data. Generally, our parser results in errors that are comparable to the state-of-the-art parsers; however, greater reductions are observed for various attachments errors. One of the largest gains comes from the clause attachment, i.e., 0.12 reduction in average errors from Berkeley and 0.05 from CVG. The average number of errors is also reduced by 0.09 from Berkeley and 0.06 from CVG for the PP attachment. We also observed large reductions in coordination and unary rule errors.

7 Conclusion

We have introduced transition-based neural constituent parsing — a neural network architecture that encodes each state explicitly — as a continuous vector by considering the recurrent se-

[17] `https://code.google.com/p/berkeley-parser-analyser/`

error type	Berkeley	CVG	TNCP
PP Attach	0.82	0.79	**0.73**
Clause Attach	0.50	0.43	**0.38**
Diff Label	0.29	0.29	0.29
Mod Attach	0.27	0.27	0.27
NP Attach	0.37	**0.31**	0.32
Co-ord	0.38	0.32	**0.29**
1-Word Span	**0.28**	0.31	0.30
Unary	0.24	0.22	**0.18**
NP Int	**0.18**	0.19	0.20
Other	**0.41**	**0.41**	0.45

Table 7: Comparison of different parsers on the WSJ test data measured by average number of errors per sentence; the numbers in bold indicate the least errors in each error type.

quences of the stack and queue in the transition-based parsing framework in addition to recursively constructed partial trees. Our parser works in a standalone fashion without reranking and does not rely on an external POS tagger or additional monolingual data for reliable estimates of syntactic and/or semantic representations of words. The parser achieves performance that is comparable to state-of-the-art systems.

In the future, we plan to apply our neural network structure to dependency parsing. We are also interested in using long short-term memory neural networks (Hochreiter and Schmidhuber, 1997) to better model the locality of propagated information from the stack and queue. The parameter estimation under semi-supervised setting will be investigated further.

Acknowledgments

We would like to thank Lemao Liu for suggestions while drafting this paper. We are also grateful for various comments from anonymous reviewers.

References

George Berg. 1992. A connectionist parser with recursive sentence structure and lexical disambiguation. In *Proc. of AAAI '92*, pages 32–37.

Ezra Black, Steve Abney, Dan Flickinger, Claudia Gdaniec, Ralph Grishman, Phil Harrison, Don Hindle, Robert Ingria, Fred Jelinek, Judith Klavans, Mark Liberman, Mitchell Marcus, Salim Roukos, Beatrice Santorini, and Tomek Strzalkowski. 1991. Procedure for quantitatively comparing the syntactic coverage of english grammars. In *Proc. of the Workshop on Speech and Natural Language*, pages 306–311, Stroudsburg, PA, USA.

Eugene Charniak and Mark Johnson. 2005. Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proc. of ACL 2005*, pages 173–180, Ann Arbor, Michigan, June.

Eugene Charniak. 2000. A maximum-entropy-inspired parser. In *Proc. of NAACL 2000*, pages 132–139, Stroudsburg, PA, USA.

Danqi Chen and Christopher Manning. 2014. A fast and accurate dependency parser using neural networks. In *Proc. of EMNLP 2014*, pages 740–750, Doha, Qatar, October.

Michael Collins and Terry Koo. 2005. Discriminative reranking for natural language parsing. *Computational Linguistics*, 31(1):25–70, March.

Michael Collins and Brian Roark. 2004. Incremental parsing with the perceptron algorithm. In *Proc. of ACL 2004*, pages 111–118, Barcelona, Spain, July.

Michael Collins. 1997. Three generative, lexicalised models for statistical parsing. In *Proc. of ACL '97*, pages 16–23, Madrid, Spain, July.

Michael Collins. 2003. Head-driven statistical models for natural language parsing. *Computational Linguistics*, 29(4):589–637, December.

Fabrizio Costa, Paolo Frasconi, Vincenzo Lombardo, and Giovanni Soda. 2003. Towards incremental parsing of natural language using recursive neural networks. *Applied Intelligence*, 19(1-2):9–25, May.

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. 2012. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc.

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 *Proc. of ACL 2014*, pages 1370–1380, Baltimore, Maryland, June.

John Duchi and Yoram Singer. 2009. Efficient online and batch learning using forward backward splitting. *Journal of Machine Learning Research*, 10:2899–2934, December.

John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, July.

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

Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proc. of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS-10)*, volume 9, pages 249–256.

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

David Hall, Greg Durrett, and Dan Klein. 2014. Less grammar, more features. In *Proc. of ACL 2014*, pages 228–237, Baltimore, Maryland, June.

Kazuma Hashimoto, Makoto Miwa, Yoshimasa Tsuruoka, and Takashi Chikayama. 2013. Simple customization of recursive neural networks for semantic relation classification. In *Proc. of EMNLP 2013*, pages 1372–1376, Seattle, Washington, USA, October.

James Henderson. 2003. Inducing history representations for broad coverage statistical parsing. In *Proc. of HLT-NAACL 2003*, pages 24–31, Stroudsburg, PA, USA.

James Henderson. 2004. Discriminative training of a neural network statistical parser. In *Proc. of ACL 2004*, pages 95–102, Barcelona, Spain, July.

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

Liang Huang, Suphan Fayong, and Yang Guo. 2012. Structured perceptron with inexact search. In *Proc. of NAACL-HLT 2012*, pages 142–151, Montréal, Canada, June.

Dan Klein and Christopher D. Manning. 2003. Accurate unlexicalized parsing. In *Proc. of ACL 2003*, pages 423–430, Sapporo, Japan, July.

Jonathan K. Kummerfeld, David Hall, James R. Curran, and Dan Klein. 2012. Parser showdown at the wall street corral: An empirical investigation of error types in parser output. In *Proc. of EMNLP-CoNLL 2012*, pages 1048–1059, Jeju Island, Korea, July.

Phong Le and Willem Zuidema. 2014. The inside-outside recursive neural network model for dependency parsing. In *Proc. of EMNLP 2014*, pages 729–739, Doha, Qatar, October.

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

Takuya Matsuzaki, Yusuke Miyao, and Jun'ichi Tsujii. 2005. Probabilistic CFG with latent annotations. In *Proc. of ACL 2005*, pages 75–82, Ann Arbor, Michigan, June.

Marshall R. Mayberry and Risto Miikkulainen. 1999. Sardsrn: A neural network shift-reduce parser. In *Proc. of IJCAI '99*, pages 820–827, San Francisco, CA, USA.

Sauro Menchetti, Fabrizio Costa, Paolo Frasconi, and Massimiliano Pontil. 2005. Wide coverage natural language processing using kernel methods and neural networks for structured data. *Pattern Recognition Letters*, 26(12):1896–1906, September.

Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černocký, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Proc. of INTERSPEECH 2010*, pages 1045–1048.

Andriy Mnih and Yee W. Teh. 2012. A fast and simple algorithm for training neural probabilistic language models. In John Langford and Joelle Pineau, editors, *Proc. of ICML-2012*, pages 1751–1758, New York, NY, USA.

Slav Petrov and Dan Klein. 2007. Improved inference for unlexicalized parsing. In *Proc. of NAACL-HLT 2007*, pages 404–411, Rochester, New York, April.

Slav Petrov, Leon Barrett, Romain Thibaux, and Dan Klein. 2006. Learning accurate, compact, and interpretable tree annotation. In *Proc. of COLING-ACL 2006*, pages 433–440, Sydney, Australia, July.

Jordan B. Pollack. 1990. Recursive distributed representations. *Artificial Intelligence*, 46(1-2):77–105, November.

Kenji Sagae and Alon Lavie. 2005. A classifier-based parser with linear run-time complexity. In *Proc. of the Ninth International Workshop on Parsing Technology*, pages 125–132, Vancouver, British Columbia, October.

Andrew Senior, Georg Heigold, Marc'Aurelio Ranzato, and Ke Yang. 2013. An empirical study of learning rates in deep neural networks for speech recognition. In *Proc. of ICASSP 2013*, pages 6724–6728, May.

Richard Socher, John Bauer, Christopher D. Manning, and Ng Andrew Y. 2013. Parsing with compositional vector grammars. In *Proc. of ACL 2013*, pages 455–465, Sofia, Bulgaria, August.

Pontus Stenetorp. 2013. Transition-based dependency parsing using recursive neural networks. In *Proc. of Deep Learning Workshop at the 2013 Conference on Neural Information Processing Systems (NIPS)*, Lake Tahoe, Nevada, USA, December.

Akihiro Tamura, Taro Watanabe, and Eiichiro Sumita. 2014. Recurrent neural networks for word alignment model. In *Proc. of ACL 2014*, pages 1470–1480, Baltimore, Maryland, June.

Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proc. of HLT-NAACL 2003*, pages 173–180, Stroudsburg, PA, USA.

	rep. size	32	64	128	256	512	1024	2048	4096
dev	WSJ-32	89.91	90.15	90.48	90.70	90.75	90.87	**91.11**	91.04
	64	90.37	90.73	90.81	90.62	90.71	91.11	91.34	**91.36**
	CTB-32	79.25	81.59	82.80	82.68	84.17	85.12	85.61	**85.76**
	64	84.04	83.29	82.92	85.12	85.24	85.77	86.28	**86.94**
test	WSJ-32	89.03	89.49	89.75	**90.45**	90.37	90.01	90.33	90.40
	64	89.74	90.16	90.48	90.06	89.91	90.68	**91.05**	90.94
	CTB-32	75.19	78.29	80.46	81.87	83.16	82.64	83.13	**83.67**
	64	80.11	81.35	81.67	82.91	83.76	84.33	83.76	**84.38**

Table 8: Comparison of various state/word representation dimension size measured by labeled F1(%). "-32" denotes the hidden state size $m = 32$. The numbers in bold indicate the best results for each hidden state dimension.

Joseph Turian, Lev-Arie Ratinov, and Yoshua Bengio. 2010. Word representations: A simple and general method for semi-supervised learning. In *Proc. of ACL 2010*, pages 384–394, Uppsala, Sweden, July.

Zhiguo Wang and Nianwen Xue. 2014. Joint pos tagging and transition-based constituent parsing in chinese with non-local features. In *Proc. of ACL 2014*, pages 733–742, Baltimore, Maryland, June.

Naiwen Xue, Fei Xia, Fu-dong Chiou, and Marta Palmer. 2005. The penn chinese treebank: Phrase structure annotation of a large corpus. *Natural Language Engineering*, 11(2):207–238, June.

Yue Zhang and Stephen Clark. 2009. Transition-based parsing of the chinese treebank using a global discriminative model. In *Proc. of the 11th International Conference on Parsing Technologies (IWPT'09)*, pages 162–171, Paris, France, October.

Muhua Zhu, Yue Zhang, Wenliang Chen, Min Zhang, and Jingbo Zhu. 2013. Fast and accurate shift-reduce constituent parsing. In *Proc. of ACL 2013*, pages 434–443, Sofia, Bulgaria, August.

A Additional Results

We conducted additional experiments by enlarging the word representation vector size m' in Table 8. In general, we observed further gains with richer word representation, but suffered overfitting effects when setting $m' = 4096$. The results with $m = 64$ and $m' = 4096$ achieved the best performance on the development data, 91.36% and 86.94% labeled F1 for WSJ and CTB, respectively, wherein we observed the accuracies of 90.94% and 84.38% on the test data, respectively. Note that it took approximately one week to train the model when $m' = 4096$ under WSJ, which was impractical to analyze the results further, e.g. comparison with other training objectives.