Transition-Based Dependency Parsing with Stack Long Short-Term Memory

Chris Dyer[♣♠] **Miguel Ballesteros**[♢♠] **Wang Ling**[♠] **Austin Matthews**[♠] **Noah A. Smith**[♠]
[♣]Marianas Labs [♢]NLP Group, Pompeu Fabra University [♠]Carnegie Mellon University
chris@marianaslabs.com, miguel.ballesteros@upf.edu,
{lingwang,austinma,nasmith}@cs.cmu.edu

Abstract

We propose a technique for learning representations of parser states in transition-based dependency parsers. Our primary innovation is a new control structure for sequence-to-sequence neural networks—the stack LSTM. Like the conventional stack data structures used in transition-based parsing, elements can be pushed to or popped from the top of the stack in constant time, but, in addition, an LSTM maintains a continuous space embedding of the stack contents. This lets us formulate an efficient parsing model that captures three facets of a parser's state: (i) unbounded look-ahead into the buffer of incoming words, (ii) the complete history of actions taken by the parser, and (iii) the complete contents of the stack of partially built tree fragments, including their internal structures. Standard backpropagation techniques are used for training and yield state-of-the-art parsing performance.

1 Introduction

Transition-based dependency parsing formalizes the parsing problem as a series of decisions that read words sequentially from a buffer and combine them incrementally into syntactic structures (Yamada and Matsumoto, 2003; Nivre, 2003; Nivre, 2004). This formalization is attractive since the number of operations required to build any projective parse tree is linear in the length of the sentence, making transition-based parsing computationally efficient relative to graph- and grammar-based formalisms. The challenge in transition-based parsing is modeling which action should be taken in each of the unboundedly many states encountered as the parser progresses.

This challenge has been addressed by development of alternative transition sets that simplify the modeling problem by making better attachment decisions (Nivre, 2007; Nivre, 2008; Nivre, 2009; Choi and McCallum, 2013; Bohnet and Nivre, 2012), through feature engineering (Zhang and Nivre, 2011; Ballesteros and Nivre, 2014; Chen et al., 2014; Ballesteros and Bohnet, 2014) and more recently using neural networks (Chen and Manning, 2014; Stenetorp, 2013).

We extend this last line of work by learning representations of the parser state that are sensitive to the complete contents of the parser's state: that is, the complete input buffer, the complete history of parser actions, and the complete contents of the stack of partially constructed syntactic structures. This "global" sensitivity to the state contrasts with previous work in transition-based dependency parsing that uses only a narrow view of the parsing state when constructing representations (e.g., just the next few incoming words, the head words of the top few positions in the stack, etc.). Although our parser integrates large amounts of information, the representation used for prediction at each time step is constructed incrementally, and therefore parsing and training time remain linear in the length of the input sentence. The technical innovation that lets us do this is a variation of recurrent neural networks with long short-term memory units (LSTMs) which we call **stack LSTMs** (§2), and which support both reading (pushing) and "forgetting" (popping) inputs.

Our parsing model uses three stack LSTMs: one representing the input, one representing the stack of partial syntactic trees, and one representing the history of parse actions to encode parser states (§3). Since the stack of partial syntactic trees may contain both individual tokens and partial syntactic structures, representations of individual tree fragments are computed compositionally with recursive (i.e., similar to Socher et al., 2014) neural networks. The parameters are learned with backpropagation (§4), and we obtain state-of-the-art results on Chinese and English dependency parsing tasks (§5).

2 Stack LSTMs

In this section we provide a brief review of LSTMs (§2.1) and then define stack LSTMs (§2.2).

Notation. We follow the convention that vectors are written with lowercase, boldface letters (e.g., $\mathbf{v}$ or $\mathbf{v}_w$); matrices are written with uppercase, boldface letters (e.g., $\mathbf{M}$, $\mathbf{M}_a$, or $\mathbf{M}_{ab}$), and scalars are written as lowercase letters (e.g., s or q_z). Structured objects such as sequences of discrete symbols are written with lowercase, bold, italic letters (e.g., $\boldsymbol{w}$ refers to a sequence of input words). Discussion of dimensionality is deferred to the experiments section below (§5).

2.1 Long Short-Term Memories

LSTMs are a variant of recurrent neural networks (RNNs) designed to cope with the vanishing gradient problem inherent in RNNs (Hochreiter and Schmidhuber, 1997; Graves, 2013). RNNs read a vector $\mathbf{x}_t$ at each time step and compute a new (hidden) state $\mathbf{h}_t$ by applying a linear map to the concatenation of the previous time step's state $\mathbf{h}_{t-1}$ and the input, and passing this through a logistic sigmoid nonlinearity. Although RNNs can, in principle, model long-range dependencies, training them is difficult in practice since the repeated application of a squashing nonlinearity at each step results in an exponential decay in the error signal through time. LSTMs address this with an extra memory "cell" ($\mathbf{c}_t$) that is constructed as a linear combination of the previous state and signal from the input.

LSTM cells process inputs with three multiplicative gates which control what proportion of the current input to pass into the memory cell ($\mathbf{i}_t$) and what proportion of the previous memory cell to "forget" ($\mathbf{f}_t$). The updated value of the memory cell after an input $\mathbf{x}_t$ is computed as follows:

$$\begin{aligned}
\mathbf{i}_t &= \sigma(\mathbf{W}_{ix}\mathbf{x}_t + \mathbf{W}_{ih}\mathbf{h}_{t-1} + \mathbf{W}_{ic}\mathbf{c}_{t-1} + \mathbf{b}_i)\\
\mathbf{f}_t &= \sigma(\mathbf{W}_{fx}\mathbf{x}_t + \mathbf{W}_{fh}\mathbf{h}_{t-1} + \mathbf{W}_{fc}\mathbf{c}_{t-1} + \mathbf{b}_f)\\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} +\\
&\qquad \mathbf{i}_t \odot \tanh(\mathbf{W}_{cx}\mathbf{x}_t + \mathbf{W}_{ch}\mathbf{h}_{t-1} + \mathbf{b}_c),
\end{aligned}$$

where σ is the component-wise logistic sigmoid function, and $\odot$ is the component-wise (Hadamard) product.

The value $\mathbf{h}_t$ of the LSTM at each time step is controlled by a third gate ($\mathbf{o}_t$) that is applied to the result of the application of a nonlinearity to the memory cell contents:

$$\begin{aligned}
\mathbf{o}_t &= \sigma(\mathbf{W}_{ox}\mathbf{x}_t + \mathbf{W}_{oh}\mathbf{h}_{t-1} + \mathbf{W}_{oc}\mathbf{c}_t + \mathbf{b}_o)\\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t).
\end{aligned}$$

To improve the representational capacity of LSTMs (and RNNs generally), LSTMs can be stacked in "layers" (Pascanu et al., 2014). In these architectures, the input LSTM at higher layers at time t is the value of $\mathbf{h}_t$ computed by the lower layer (and $\mathbf{x}_t$ is the input at the lowest layer).

Finally, output is produced at each time step from the $\mathbf{h}_t$ value at the top layer:

$$\mathbf{y}_t = g(\mathbf{h}_t),$$

where g is an arbitrary differentiable function.

2.2 Stack Long Short-Term Memories

Conventional LSTMs model sequences in a left-to-right order.[1] Our innovation here is to augment the LSTM with a "stack pointer." Like a conventional LSTM, new inputs are always added in the right-most position, but in stack LSTMs, the current location of the stack pointer determines which cell in the LSTM provides $\mathbf{c}_{t-1}$ and $\mathbf{h}_{t-1}$ when computing the new memory cell contents.

In addition to adding elements to the end of the sequence, the stack LSTM provides a pop operation which moves the stack pointer to the previous element (i.e., the previous element that was extended, not necessarily the right-most element). Thus, the LSTM can be understood as a stack implemented so that contents are never overwritten, that is, push always adds a new entry at the end of the list that contains a back-pointer to the previous top, and pop only updates the stack pointer.[2] This control structure is schematized in Figure 1.

By querying the output vector to which the stack pointer points (i.e., the $\mathbf{h}_{\mathsf{TOP}}$), a continuous-space "summary" of the contents of the current stack configuration is available. We refer to this value as the "stack summary."

What does the stack summary look like? Intuitively, elements near the top of the stack will

[1] Ours is not the first deviation from a strict left-to-right order: previous variations include bidirectional LSTMs (Graves and Schmidhuber, 2005) and multidimensional LSTMs (Graves et al., 2007).

[2] Goldberg et al. (2013) propose a similar stack construction to prevent stack operations from invalidating existing references to the stack in a beam-search parser that must (efficiently) maintain a priority queue of stacks.

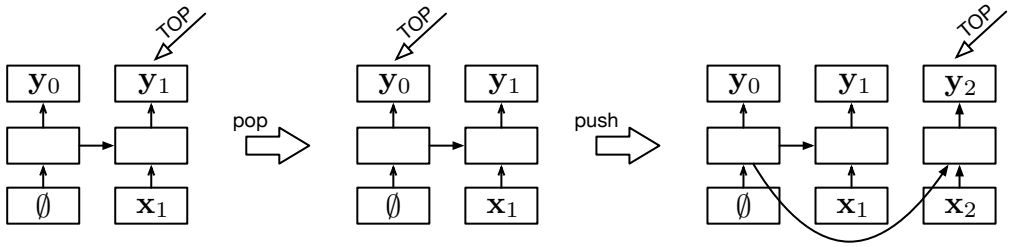

Figure 1: A stack LSTM extends a conventional left-to-right LSTM with the addition of a stack pointer (notated as TOP in the figure). This figure shows three configurations: a stack with a single element (left), the result of a pop operation to this (middle), and then the result of applying a push operation (right). The boxes in the lowest rows represent stack contents, which are the inputs to the LSTM, the upper rows are the outputs of the LSTM (in this paper, only the output pointed to by TOP is ever accessed), and the middle rows are the memory cells (the $\mathbf{c}_t$'s and $\mathbf{h}_t$'s) and gates. Arrows represent function applications (usually affine transformations followed by a nonlinearity), refer to §2.1 for specifics.

influence the representation of the stack. However, the LSTM has the flexibility to learn to extract information from arbitrary points in the stack (Hochreiter and Schmidhuber, 1997).

Although this architecture is to the best of our knowledge novel, it is reminiscent of the Recurrent Neural Network Pushdown Automaton (NNPDA) of Das et al. (1992), which added an external stack memory to an RNN. However, our architecture provides an embedding of the complete contents of the stack, whereas theirs made only the top of the stack visible to the RNN.

3 Dependency Parser

We now turn to the problem of learning representations of dependency parsers. We preserve the standard data structures of a transition-based dependency parser, namely a buffer of words (B) to be processed and a stack (S) of partially constructed syntactic elements. Each stack element is augmented with a continuous-space vector embedding representing a word and, in the case of S, any of its syntactic dependents. Additionally, we introduce a third stack (A) to represent the history of actions taken by the parser.[3] Each of these stacks is associated with a stack LSTM that provides an encoding of their current contents. The full architecture is illustrated in Figure 3, and we will review each of the components in turn.

[3]The A stack is only ever pushed to; our use of a stack here is purely for implementational and expository convenience.

3.1 Parser Operation

The dependency parser is initialized by pushing the words and their representations (we discuss word representations below in §3.3) of the input sentence in reverse order onto B such that the first word is at the top of B and the ROOT symbol is at the bottom, and S and A each contain an empty-stack token. At each time step, the parser computes a composite representation of the stack states (as determined by the current configurations of B, S, and A) and uses that to predict an action to take, which updates the stacks. Processing completes when B is empty (except for the empty-stack symbol), S contains two elements, one representing the full parse tree headed by the ROOT symbol and the other the empty-stack symbol, and A is the history of operations taken by the parser.

The parser state representation at time t, which we write $\mathbf{p}_t$, which is used to is determine the transition to take, is defined as follows:

$$\mathbf{p}_t = \max\{\mathbf{0}, \mathbf{W}[\mathbf{s}_t; \mathbf{b}_t; \mathbf{a}_t] + \mathbf{d}\},$$

where $\mathbf{W}$ is a learned parameter matrix, $\mathbf{b}_t$ is the stack LSTM encoding of the input buffer B, $\mathbf{s}_t$ is the stack LSTM encoding of S, $\mathbf{a}_t$ is the stack LSTM encoding of A, $\mathbf{d}$ is a bias term, then passed through a component-wise rectified linear unit (ReLU) nonlinearity (Glorot et al., 2011).[4]

Finally, the parser state $\mathbf{p}_t$ is used to compute

[4]In preliminary experiments, we tried several nonlinearities and found ReLU to work slightly better than the others.

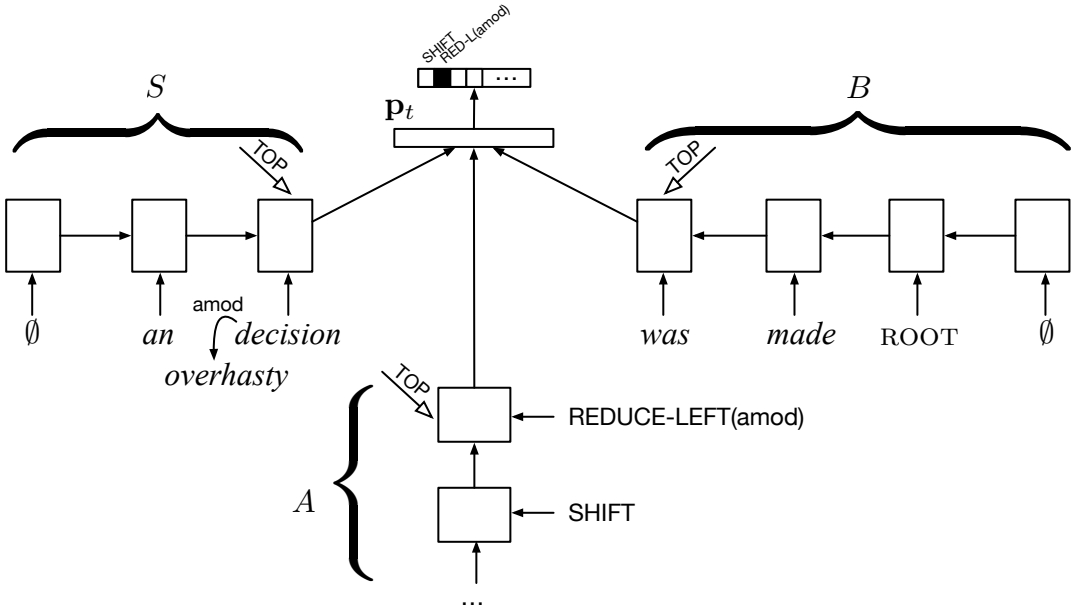

Figure 2: Parser state computation encountered while parsing the sentence "*an overhasty decision was made*." Here S designates the stack of partially constructed dependency subtrees and its LSTM encoding; B is the buffer of words remaining to be processed and its LSTM encoding; and A is the stack representing the history of actions taken by the parser. These are linearly transformed, passed through a ReLU nonlinearity to produce the parser state embedding $\mathbf{p}_t$. An affine transformation of this embedding is passed to a softmax layer to give a distribution over parsing decisions that can be taken.

the probability of the parser action at time t as:

$$p(z_t \mid \mathbf{p}_t) = \frac{\exp\left(\mathbf{g}_{z_t}^\top \mathbf{p}_t + q_{z_t}\right)}{\sum_{z' \in \mathcal{A}(S,B)} \exp\left(\mathbf{g}_{z'}^\top \mathbf{p}_t + q_{z'}\right)},$$

where $\mathbf{g}_z$ is a column vector representing the (output) embedding of the parser action z, and q_z is a bias term for action z. The set $\mathcal{A}(S, B)$ represents the valid actions that may be taken given the current contents of the stack and buffer.[5] Since $\mathbf{p}_t = f(\mathbf{s}_t, \mathbf{b}_t, \mathbf{a}_t)$ encodes information about all previous decisions made by the parser, the chain rule may be invoked to write the probability of any valid sequence of parse actions $\boldsymbol{z}$ conditional on the input as:

$$p(\boldsymbol{z} \mid \boldsymbol{w}) = \prod_{t=1}^{|\boldsymbol{z}|} p(z_t \mid \mathbf{p}_t). \qquad (1)$$

3.2 Transition Operations

Our parser is based on the arc-standard transition inventory (Nivre, 2004), given in Figure 3.

[5] In general, $\mathcal{A}(S, B)$ is the complete set of parser actions discussed in §3.2, but in some cases not all actions are available. For example, when S is empty and words remain in B, a SHIFT operation is obligatory (Sartorio et al., 2013).

Why arc-standard? Arc-standard transitions parse a sentence from left to right, using a stack to store partially built syntactic structures and a buffer that keeps the incoming tokens to be parsed. The parsing algorithm chooses an action at each configuration by means of a score. In arc-standard parsing, the dependency tree is constructed bottom-up, because right-dependents of a head are only attached after the subtree under the dependent is fully parsed. Since our parser recursively computes representations of tree fragments, this construction order guarantees that once a syntactic structure has been used to modify a head, the algorithm will not try to find another head for the dependent structure. This means we can evaluate composed representations of tree fragments incrementally; we discuss our strategy for this below (§3.4).

3.3 Token Embeddings and OOVs

To represent each input token, we concatenate three vectors: a learned vector representation for each word type ($\mathbf{w}$); a fixed vector representation from a neural language model ($\tilde{\mathbf{w}}_{\text{LM}}$), and a learned representation ($\mathbf{t}$) of the POS tag of the token, provided as auxiliary input to the parser. A

Stack$_t$	Buffer$_t$	Action	Stack$_{t+1}$	Buffer$_{t+1}$	Dependency
$(\mathbf{u}, u), (\mathbf{v}, v), S$	B	REDUCE-RIGHT(r)	$(g_r(\mathbf{u}, \mathbf{v}), u), S$	B	$u \xrightarrow{r} v$
$(\mathbf{u}, u), (\mathbf{v}, v), S$	B	REDUCE-LEFT(r)	$(g_r(\mathbf{v}, \mathbf{u}), v), S$	B	$u \xleftarrow{r} v$
S	$(\mathbf{u}, u), B$	SHIFT	$(\mathbf{u}, u), S$	B	—

Figure 3: Parser transitions indicating the action applied to the stack and buffer and the resulting stack and buffer states. Bold symbols indicate (learned) embeddings of words and relations, script symbols indicate the corresponding words and relations.

linear map ($\mathbf{V}$) is applied to the resulting vector and passed through a component-wise ReLU,

$$\mathbf{x} = \max\left\{\mathbf{0}, \mathbf{V}[\mathbf{w}; \tilde{\mathbf{w}}_{\text{LM}}; \mathbf{t}] + \mathbf{b}\right\}.$$

This mapping can be shown schematically as in Figure 4.

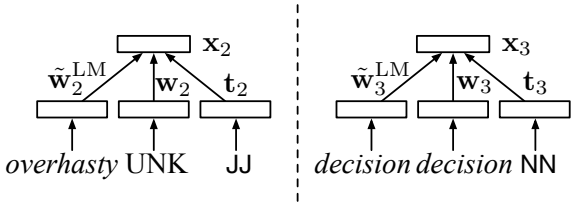

Figure 4: Token embedding of the words *decision*, which is present in both the parser's training data and the language model data, and *overhasty*, an adjective that is not present in the parser's training data but is present in the LM data.

This architecture lets us deal flexibly with out-of-vocabulary words—both those that are OOV in both the very limited parsing data but present in the pretraining LM, and words that are OOV in both. To ensure we have estimates of the OOVs in the parsing training data, we stochastically replace (with $p = 0.5$) each singleton word type in the parsing training data with the UNK token in each training iteration.

Pretrained word embeddings. A veritable cottage industry exists for creating word embeddings, meaning numerous pretraining options for $\tilde{\mathbf{w}}_{\text{LM}}$ are available. However, for syntax modeling problems, embedding approaches which discard order perform less well (Bansal et al., 2014); therefore we used a variant of the skip n-gram model introduced by Ling et al. (2015), named "structured skip n-gram," where a different set of parameters is used to predict each context word depending on its position relative to the target word. The hyperparameters of the model are the same as in the skip n-gram model defined in word2vec (Mikolov et al., 2013), and we set the window size to 5, used a negative sampling rate to 10, and ran 5 epochs through unannotated corpora described in §5.1.

3.4 Composition Functions

Recursive neural network models enable complex phrases to be represented compositionally in terms of their parts and the relations that link them (Socher et al., 2011; Socher et al., 2013c; Hermann and Blunsom, 2013; Socher et al., 2013b). We follow this previous line of work in embedding dependency tree fragments that are present in the stack S in the same vector space as the token embeddings discussed above.

A particular challenge here is that a syntactic head may, in general, have an arbitrary number of dependents. To simplify the parameterization of our composition function, we combine head-modifier pairs one at a time, building up more complicated structures in the order they are "reduced" in the parser, as illustrated in Figure 5. Each node in this expanded syntactic tree has a value computed as a function of its three arguments: the syntactic head ($\mathbf{h}$), the dependent ($\mathbf{d}$), and the syntactic relation being satisfied ($\mathbf{r}$). We define this by concatenating the vector embeddings of the head, dependent and relation, applying a linear operator and a component-wise nonlinearity as follows:

$$\mathbf{c} = \tanh\left(\mathbf{U}[\mathbf{h}; \mathbf{d}; \mathbf{r}] + \mathbf{e}\right).$$

For the relation vector, we use an embedding of the parser action that was applied to construct the relation (i.e., the syntactic relation paired with the direction of attachment).

4 Training Procedure

We trained our parser to maximize the conditional log-likelihood (Eq. 1) of treebank parses given sentences. Our implementation constructs a computation graph for each sentence and runs forward- and backpropagation to obtain the gradients of this

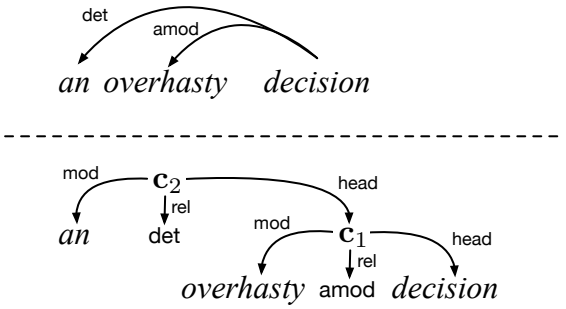

Figure 5: The representation of a dependency subtree (above) is computed by recursively applying composition functions to $\langle \text{head}, \text{modifier}, \text{relation} \rangle$ triples. In the case of multiple dependents of a single head, the recursive branching order is imposed by the order of the parser's reduce operations (below).

objective with respect to the model parameters. The computations for a single parsing model were run on a single thread on a CPU. Using the dimensions discussed in the next section, we required between 8 and 12 hours to reach convergence on a held-out dev set.[6]

Parameter optimization was performed using stochastic gradient descent with an initial learning rate of $\eta_0 = 0.1$, and the learning rate was updated on each pass through the training data as $\eta_t = \eta_0/(1 + \rho t)$, with $\rho = 0.1$ and where t is the number of epochs completed. No momentum was used. To mitigate the effects of "exploding" gradients, we clipped the ℓ_2 norm of the gradient to 5 before applying the weight update rule (Sutskever et al., 2014; Graves, 2013). An ℓ_2 penalty of 1×10^{-6} was applied to all weights.

Matrix and vector parameters were initialized with uniform samples in $\pm\sqrt{6/(r+c)}$, where r and c were the number of rows and columns in the structure (Glorot and Bengio, 2010).

Dimensionality. The full version of our parsing model sets dimensionalities as follows. LSTM hidden states are of size 100, and we use two layers of LSTMs for each stack. Embeddings of the parser actions used in the composition functions have 16 dimensions, and the output embedding size is 20 dimensions. Pretained word embeddings have 100 dimensions (English) and 80 dimensions (Chinese), and the learned word embeddings have 32 dimensions. Part of speech embeddings have 12 dimensions.

These dimensions were chosen based on intuitively reasonable values (words should have higher dimensionality than parsing actions, POS tags, and relations; LSTM states should be relatively large), and it was confirmed on development data that they performed well.[7] Future work might more carefully optimize these parameters; our reported architecture strikes a balance between minimizing computational expense and finding solutions that work.

5 Experiments

We applied our parsing model and several variations of it to two parsing tasks and report results below.

5.1 Data

We used the same data setup as Chen and Manning (2014), namely an English and a Chinese parsing task. This baseline configuration was chosen since they likewise used a neural parameterization to predict actions in an arc-standard transition-based parser.

- For English, we used the Stanford Dependencency (SD) treebank (de Marneffe et al., 2006) used in (Chen and Manning, 2014) which is the closest model published, with the same splits.[8] The part-of-speech tags are predicted by using the Stanford Tagger (Toutanova et al., 2003) with an accuracy of 97.3%. This treebank contains a negligible amount of non-projective arcs (Chen and Manning, 2014).

- For Chinese, we use the Penn Chinese Treebank 5.1 (CTB5) following Zhang and Clark (2008),[9] with gold part-of-speech tags which is also the same as in Chen and Manning (2014).

Language model word embeddings were generated, for English, from the AFP portion of the English Gigaword corpus (version 5), and from the complete Chinese Gigaword corpus (version 2),

[6] Software for replicating the experiments is available from `https://github.com/clab/lstm-parser`.

[7] We did perform preliminary experiments with LSTM states of 32, 50, and 80, but the other dimensions were our initial guesses.

[8] Training: 02-21. Development: 22. Test: 23.

[9] Training: 001–815, 1001–1136. Development: 886–931, 1148–1151. Test: 816–885, 1137–1147.

as segmented by the Stanford Chinese Segmenter (Tseng et al., 2005).

5.2 Experimental configurations

We report results on five experimental configurations per language, as well as the Chen and Manning (2014) baseline. These are: the full stack LSTM parsing model (S-LSTM), the stack LSTM parsing model without POS tags (−POS), the stack LSTM parsing model without pretrained language model embeddings (−pretraining), the stack LSTM parsing model that uses just head words on the stack instead of composed representations (−composition), and the full parsing model where rather than an LSTM, a classical recurrent neural network is used (S-RNN).

5.3 Results

Following Chen and Manning (2014) we exclude punctuation symbols for evaluation. Tables 1 and 2 show comparable results with Chen and Manning (2014), and we show that our model is better than their model in both the development set and the test set.

	Development		Test	
	UAS	LAS	UAS	LAS
S-LSTM	**93.2**	**90.9**	**93.1**	**90.9**
−POS	93.1	90.4	92.7	90.3
−pretraining	92.7	90.4	92.4	90.0
−composition	92.7	89.9	92.2	89.6
S-RNN	92.8	90.4	92.3	90.1
C&M (2014)	92.2	89.7	91.8	89.6

Table 1: English parsing results (SD)

	Dev. set		Test set	
	UAS	LAS	UAS	LAS
S-LSTM	**87.2**	**85.9**	**87.2**	**85.7**
−composition	85.8	84.0	85.3	83.6
−pretraining	86.3	84.7	85.7	84.1
−POS	82.8	79.8	82.2	79.1
S-RNN	86.3	84.7	86.1	84.6
C&M (2014)	84.0	82.4	83.9	82.4

Table 2: Chinese parsing results (CTB5)

5.4 Analysis

Overall, our parser substantially outperforms the baseline neural network parser of Chen and Manning (2014), both in the full configuration and in the various ablated conditions we report. The one exception to this is the −POS condition for the Chinese parsing task, which in which we underperform their baseline (which used gold POS tags), although we do still obtain reasonable parsing performance in this limited case. We note that predicted POS tags in English add very little value—suggesting that we can think of parsing sentences directly without first tagging them. We also find that using composed representations of dependency tree fragments outperforms using representations of head words alone, which has implications for theories of headedness. Finally, we find that while LSTMs outperform baselines that use only classical RNNs, these are still quite capable of learning good representations.

Effect of beam size. Beam search was determined to have minimal impact on scores (absolute improvements of $\leq 0.3\%$ were possible with small beams). Therefore, all results we report used greedy decoding—Chen and Manning (2014) likewise only report results with greedy decoding. This finding is in line with previous work that generates sequences from recurrent networks (Grefenstette et al., 2014), although Vinyals et al. (2015) did report much more substantial improvements with beam search on their "grammar as a foreign language" parser.[10]

6 Related Work

Our approach ties together several strands of previous work. First, several kinds of stack memories have been proposed to augment neural architectures. Das et al. (1992) proposed a neural network with an external stack memory based on recurrent neural networks. In contrast to our model, in which the entire contents of the stack are summarized in a single value, in their model, the network could only see the contents of the top of the stack. Mikkulainen (1996) proposed an architecture with a stack that had a summary feature, although the stack control was learned as a latent variable.

A variety of authors have used neural networks to predict parser actions in shift-reduce parsers. The earliest attempt we are aware of is due to Mayberry and Miikkulainen (1999). The resurgence of interest in neural networks has resulted

[10] Although superficially similar to ours, Vinyals et al. (2015) is a phrase-structure parser and adaptation to the dependency parsing scenario would have been nontrivial. We discuss their work in §6.

in in several applications to transition-based dependency parsers (Weiss et al., 2015; Chen and Manning, 2014; Stenetorp, 2013). In these works, the conditioning structure was manually crafted and sensitive to only certain properties of the state, while we are conditioning on the global state object. Like us, Stenetorp (2013) used recursively composed representations of the tree fragments (a head and its dependents). Neural networks have also been used to learn representations for use in chart parsing (Henderson, 2004; Titov and Henderson, 2007; Socher et al., 2013a; Le and Zuidema, 2014).

LSTMs have also recently been demonstrated as a mechanism for learning to represent parse structure.Vinyals et al. (2015) proposed a phrase-structure parser based on LSTMs which operated by first reading the entire input sentence in so as to obtain a vector representation of it, and then generating bracketing structures sequentially conditioned on this representation. Although superficially similar to our model, their approach has a number of disadvantages. First, they relied on a large amount of semi-supervised training data that was generated by parsing a large unannotated corpus with an off-the-shelf parser. Second, while they recognized that a stack-like shift-reduce parser control provided useful information, they only made the top word of the stack visible during training and decoding. Third, although it is impressive feat of learning that an entire parse tree be represented by a vector, it seems that this formulation makes the problem unnecessarily difficult.

Finally, our work can be understood as a progression toward using larger contexts in parsing. An exhaustive summary is beyond the scope of this paper, but some of the important milestones in this tradition are the use of cube pruning to efficiently include nonlocal features in discriminative chart reranking (Huang and Chiang, 2008), approximate decoding techniques based on LP relaxations in graph-based parsing to include higher-order features (Martins et al., 2010), and randomized hill-climbing methods that enable arbitrary nonlocal features in global discriminative parsing models (Zhang et al., 2014). Since our parser is sensitive to any part of the input, its history, or its stack contents, it is similar in spirit to the last approach, which permits truly arbitrary features.

7 Conclusion

We presented stack LSTMs, recurrent neural networks for sequences, with push and pop operations, and used them to implement a state-of-the-art transition-based dependency parser. We conclude by remarking that stack memory offers intriguing possibilities for learning to solve general information processing problems (Mikkulainen, 1996). Here, we learned from observable stack manipulation operations (i.e., supervision from a treebank), and the computed embeddings of final parser states were not used for any further prediction. However, this could be reversed, giving a device that learns to construct context-free programs (e.g., expression trees) given only observed outputs; one application would be unsupervised parsing. Such an extension of the work would make it an alternative to architectures that have an explicit external memory such as neural Turing machines (Graves et al., 2014) and memory networks (Weston et al., 2015). However, as with those models, without supervision of the stack operations, formidable computational challenges must be solved (e.g., marginalizing over all latent stack operations), but sampling techniques and techniques from reinforcement learning have promise here (Zaremba and Sutskever, 2015), making this an intriguing avenue for future work.

Acknowledgments

The authors would like to thank Lingpeng Kong and Jacob Eisenstein for comments on an earlier version of this draft and Danqi Chen for assistance with the parsing datasets. This work was sponsored in part by the U. S. Army Research Laboratory and the U. S. Army Research Office under contract/grant number W911NF-10-1-0533, and in part by NSF CAREER grant IIS-1054319. Miguel Ballesteros is supported by the European Commission under the contract numbers FP7-ICT-610411 (project MULTISENSOR) and H2020-RIA-645012 (project KRISTINA).

References

[Ballesteros and Bohnet2014] Miguel Ballesteros and Bernd Bohnet. 2014. Automatic feature selection for agenda-based dependency parsing. In *Proc. COLING*.

[Ballesteros and Nivre2014] Miguel Ballesteros and Joakim Nivre. 2014. MaltOptimizer: Fast and

effective parser optimization. *Natural Language Engineering*.

[Bansal et al.2014] Mohit Bansal, Kevin Gimpel, and Karen Livescu. 2014. Tailoring continuous word representations for dependency parsing. In *Proc. ACL*.

[Bohnet and Nivre2012] Bernd Bohnet and Joakim Nivre. 2012. A transition-based system for joint part-of-speech tagging and labeled non-projective dependency parsing. In *Proc. EMNLP*.

[Chen and Manning2014] Danqi Chen and Christopher D. Manning. 2014. A fast and accurate dependency parser using neural networks. In *Proc. EMNLP*.

[Chen et al.2014] Wenliang Chen, Yue Zhang, and Min Zhang. 2014. Feature embedding for dependency parsing. In *Proc. COLING*.

[Choi and McCallum2013] Jinho D. Choi and Andrew McCallum. 2013. Transition-based dependency parsing with selectional branching. In *Proc. ACL*.

[Das et al.1992] Sreerupa Das, C. Lee Giles, and Guo-Zheng Sun. 1992. Learning context-free grammars: Capabilities and limitations of a recurrent neural network with an external stack memory. In *Proc. Cognitive Science Society*.

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

[Glorot and Bengio2010] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proc. ICML*.

[Glorot et al.2011] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Deep sparse rectifier neural networks. In *Proc. AISTATS*.

[Goldberg et al.2013] Yoav Goldberg, Kai Zhao, and Liang Huang. 2013. Efficient implementation of beam-search incremental parsers. In *Proc. ACL*.

[Graves and Schmidhuber2005] Alex Graves and Jürgen Schmidhuber. 2005. Framewise phoneme classification with bidirectional LSTM networks. In *Proc. IJCNN*.

[Graves et al.2007] Alex Graves, Santiago Fernández, and Jürgen Schmidhuber. 2007. Multi-dimensional recurrent neural networks. In *Proc. ICANN*.

[Graves et al.2014] Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural Turing machines. *CoRR*, abs/1410.5401.

[Graves2013] Alex Graves. 2013. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850.

[Grefenstette et al.2014] Edward Grefenstette, Karl Moritz Hermann, Georgiana Dinu, and Phil Blunsom. 2014. New directions in vector space models of meaning. ACL Tutorial.

[Henderson2004] James Henderson. 2004. Discriminative training of a neural network discriminative parser. In *Proc. ACL*.

[Hermann and Blunsom2013] Karl Moritz Hermann and Phil Blunsom. 2013. The role of syntax in vector space models of compositional semantics. In *Proc. ACL*.

[Hochreiter and Schmidhuber1997] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation*, 9(8):1735–1780.

[Huang and Chiang2008] Liang Huang and David Chiang. 2008. Forest reranking: Discriminative parsing with non-local features. In *Proc. ACL*.

[Le and Zuidema2014] Phong Le and Willem Zuidema. 2014. Inside-outside recursive neural network model for dependency parsing. In *Proc. EMNLP*.

[Ling et al.2015] Wang Ling, Chris Dyer, Alan Black, and Isabel Trancoso. 2015. Two/too simple adaptations of word2vec for syntax problems. In *Proc. NAACL*.

[Martins et al.2010] André F. T. Martins, Noah A. Smith, Eric P. Xing, Pedro M. Q. Aguiar, and Mário A. T. Figueiredo. 2010. Turboparsers: Dependency parsing by approximate variational inference. In *Proc. EMNLP*.

[Mayberry and Miikkulainen1999] Marshall R. Mayberry and Risto Miikkulainen. 1999. SARDSRN: A neural network shift-reduce parser. In *Proc. IJCAI*.

[Mikkulainen1996] Risto Mikkulainen. 1996. Subsymbolic case-role analysis of sentences with embedded clauses. *Cognitive Science*, 20:47–73.

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

[Nivre2003] Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *Proc. IWPT*.

[Nivre2004] Joakim Nivre. 2004. Incrementality in deterministic dependency parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*.

[Nivre2007] Joakim Nivre. 2007. Incremental non-projective dependency parsing. In *Proc. NAACL*.

[Nivre2008] Joakim Nivre. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34:4:513–553. MIT Press.

[Nivre2009] Joakim Nivre. 2009. Non-projective dependency parsing in expected linear time. In *Proc. ACL*.

[Pascanu et al.2014] Razvan Pascanu, Çaglar Gülçehre, Kyunghyun Cho, and Yoshua Bengio. 2014. How to construct deep recurrent neural networks. In *Proc. ICLR*.

[Sartorio et al.2013] Francesco Sartorio, Giorgio Satta, and Joakim Nivre. 2013. A transition-based dependency parser using a dynamic parsing strategy. In *Proc. ACL*.

[Socher et al.2011] Richard Socher, Eric H. Huang, Jeffrey Pennington, Andrew Y. Ng, and Christopher D. Manning. 2011. Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. In *Proc. NIPS*.

[Socher et al.2013a] Richard Socher, John Bauer, Christopher D. Manning, and Andrew Y. Ng. 2013a. Parsing with compositional vector grammars. In *Proc. ACL*.

[Socher et al.2013b] Richard Socher, Andrej Karpathy, Quoc V. Le, Christopher D. Manning, and Andrew Y. Ng. 2013b. Grounded compositional semantics for finding and describing images with sentences. *TACL*.

[Socher et al.2013c] Richard Socher, Alex Perelygin, Jean Y. Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. 2013c. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proc. EMNLP*.

[Stenetorp2013] Pontus Stenetorp. 2013. Transition-based dependency parsing using recursive neural networks. In *Proc. NIPS Deep Learning Workshop*.

[Sutskever et al.2014] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. In *Proc. NIPS*.

[Titov and Henderson2007] Ivan Titov and James Henderson. 2007. Constituent parsing with incremental sigmoid belief networks. In *Proc. ACL*.

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

[Tseng et al.2005] Huihsin Tseng, Pichuan Chang, Galen Andrew, Daniel Jurafsky, and Christopher Manning. 2005. A conditional random field word segmenter for SIGHAN bakeoff 2005. In *Proc. Fourth SIGHAN Workshop on Chinese Language Processing*.

[Vinyals et al.2015] Oriol Vinyals, Lukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. 2015. Grammar as a foreign language. In *Proc. ICLR*.

[Weiss et al.2015] David Weiss, Christopher Alberti, Michael Collins, and Slav Petrov. 2015. Structured training for neural network transition-based parsing. In *Proc. ACL*.

[Weston et al.2015] Jason Weston, Sumit Chopra, and Antoine Bordes. 2015. Memory networks. In *Proc. ICLR*.

[Yamada and Matsumoto2003] Hiroyasu Yamada and Yuji Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *Proc. IWPT*.

[Zaremba and Sutskever2015] Wojciech Zaremba and Ilya Sutskever. 2015. Reinforcement learning neural Turing machines. *ArXiv e-prints*, May.

[Zhang and Clark2008] Yue Zhang and Stephen Clark. 2008. A tale of two parsers: Investigating and combining graph-based and transition-based dependency parsing. In *Proc. EMNLP*.

[Zhang and Nivre2011] Yue Zhang and Joakim Nivre. 2011. Transition-based dependency parsing with rich non-local features. In *Proc. ACL*.

[Zhang et al.2014] Yuan Zhang, Tao Lei, Regina Barzilay, and Tommi Jaakkola. 2014. Greed is good if randomized: New inference for dependency parsing. In *Proc. EMNLP*.