Finding Syntactic Representations in Neural Stacks

William Merrill,[*] [†] **Lenny Khazan,**[†] **Noah Amsel,**[†]
Yiding Hao,[†] **Simon Mendelsohn,**[†] and **Robert Frank**[†]
[†] Yale University, New Haven, CT, USA
[*] Allen Institute for Artificial Intelligence, Seattle, WA, USA
firstname.lastname@yale.edu

Abstract

Neural network architectures have been augmented with differentiable stacks in order to introduce a bias toward learning hierarchy-sensitive regularities. It has, however, proven difficult to assess the degree to which such a bias is effective, as the operation of the differentiable stack is not always interpretable. In this paper, we attempt to detect the presence of latent representations of hierarchical structure through an exploration of the unsupervised learning of constituency structure. Using a technique due to Shen et al. (2018a,b), we extract syntactic trees from the pushing behavior of stack RNNs trained on language modeling and classification objectives. We find that our models produce parses that reflect natural language syntactic constituencies, demonstrating that stack RNNs do indeed infer linguistically relevant hierarchical structure.

1 Introduction

Sequential models such as long short-term memory networks (LSTMs; Hochreiter and Schmidhuber, 1997) have been proven capable of exhibiting qualitative behavior that reflects a sensitivity to regularities that are structurally conditioned, such as subject–verb agreement (Linzen et al., 2016; Gulordava et al., 2018). However, detailed analysis of such models has shown that this apparent sensitivity to structure does not always generalize to inputs with a high degree of syntactic complexity (Marvin and Linzen, 2018). These observations suggest that sequential models may not in fact be representing sentences in the kind of hierarchically organized representations that we might expect.

Stack-structured recurrent memory units (Joulin and Mikolov, 2015; Grefenstette et al., 2015; Yogatama et al., 2018; and others) offer a possible method for explicitly biasing neural networks to construct hierarchical representations and make use of them in their computation. Since syntactic structures can often be modeled in a context-free manner (Chomsky, 1956, 1957), the correspondence between pushdown automata and context-free grammars (Chomsky, 1962) makes stacks a natural data structure for the computation of hierarchical relations. Recently, Hao et al. (2018) have shown that stack-augmented RNNs (henceforth *stack RNNs*) have the ability to learn classical stack-based algorithms for computing context-free transductions such as string reversal. However, they also find that such algorithms can be difficult for stack RNNs to learn. For many context-free tasks such as parenthesis matching, the stack RNN models they consider instead learn heuristic "push-only" strategies that essentially reduce the stack to unstructured recurrent memory. Thus, even if stacks allow hierarchical regularities to be expressed, the bias that stack RNNs introduce does not guarantee that the networks will detect them.

The current paper aims to move beyond the work of Hao et al. (2018) in two ways. While that work was based on artificially generated formal languages, this paper considers the ability of stack RNNs to succeed on tasks over natural language data. Specifically, we train such networks on two objectives: language modeling and the *number prediction task*, a classification task proposed by Linzen et al. (2016) to determine whether or not a model can capture structure-sensitive grammatical dependencies. Further, in addition to using visualizations of the pushing and popping actions of the stack RNN to assess its hierarchical sensitivity, we use a technique proposed by Shen et al. (2018a,b) to assess the presence of implicitly-represented hierarchically-organized structure through the task

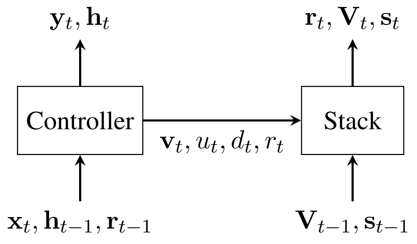
[*] Work completed while the author was at Yale University.

of unsupervised parsing. We extract syntactic constituency trees from our models and find that they produce parses that broadly reflect phrasal groupings of words in the input sentences, suggesting that our models utilize the stack in a way that reflects the syntactic structures of input sentences.

This paper is organized as follows. Section 2 introduces the architecture of our stack models, which extends the architecture of Grefenstette et al. (2015) by allowing multiple items to be pushed to, popped from, or read from the stack at each computational step. Section 3 then describes our training procedure and reports results on language modeling and agreement classification. Section 4 investigates the behavior of the stack RNNs trained on these tasks by visualizing their pushing behavior. Building on this, Section 5 describes how we adapt Shen et al.'s (2018a; 2018b) unsupervised parsing algorithm to stack RNNs and evaluates the degree to which the resulting parses reveal structural representations in stack RNNs. Section 6 discusses our observations, and Section 7 concludes.

2 Network Architecture

In a stack RNN (Grefenstette et al., 2015; Hao et al., 2018), a neural network adhering to a standard recurrent architecture, known as a *controller*, is enhanced with a non-parameterized *stack*. At each time step, the controller network receives an input vector $\mathbf{x}_t$ and a recurrent state vector $\mathbf{h}_{t-1}$ provided by the controller architecture, along with a *read vector* $\mathbf{r}_{t-1}$ summarizing the top elements on the stack. The controller interfaces with the stack by computing continuous values that serve as instructions for how the stack should be modified. These instructions consist of $\mathbf{v}_t$, a vector that is pushed to the top of the stack; d_t, a number representing the *strength* of the newly pushed vector $\mathbf{v}_t$; u_t, the number of items to pop from the stack; and r_t, the number of items to read from the top of the stack. The instructions $\langle \mathbf{v}_t, u_t, d_t, r_t \rangle$ are produced by the controller as output and presented to the stack. The stack then computes the next read vector $\mathbf{r}_t$, which is given to the controller at the next time step. This general architecture is portrayed in Figure 1. In the next two subsections, we describe how the stack computes $\mathbf{r}_t$ using the instructions $\langle \mathbf{v}_t, u_t, d_t, r_t \rangle$ and how the controller computes the stack instructions.



$\mathbf{y}_t, \mathbf{h}_t$
$\mathbf{r}_t, \mathbf{V}_t, \mathbf{s}_t$
Controller
Stack
$\mathbf{v}_t, u_t, d_t, r_t$
$\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{r}_{t-1}$
$\mathbf{V}_{t-1}, \mathbf{s}_{t-1}$

Figure 1: The neural stack architecture.

2.1 Stack Actions

A stack at time t consists of a sequence of vectors $\langle \mathbf{V}_t[1], \mathbf{V}_t[2], \ldots, \mathbf{V}_t[t] \rangle$, organized into a matrix $\mathbf{V}_t$ whose ith row is $\mathbf{V}_t[i]$. By convention, $\mathbf{V}_t[t]$ is the "top" element of the stack, while $\mathbf{V}_t[1]$ is the "bottom" element. Each element $\mathbf{V}_t[i]$ of the stack is associated with a *strength* $\mathbf{s}_t[i] \geq 0$. The strength of a vector $\mathbf{V}_t[i]$ represents the degree to which the vector is on the stack: a strength of 1 means that the vector is "fully" on the stack, while a strength of 0 means that the vector has been popped from the stack. The strengths are organized into a vector $\mathbf{s}_t = \langle \mathbf{s}_t[1], \mathbf{s}_t[2], \ldots, \mathbf{s}_t[t] \rangle$. At time t, the stack receives a set of instructions $\langle \mathbf{v}_t, u_t, d_t, r_t \rangle$ and performs three operations: *popping*, *pushing*, and *reading*, in that order.

The popping operation is implemented by reducing the strength of each item on the stack by a number $\mathbf{u}_t[i]$, ensuring that the strength of each item can never fall below 0.

$$\mathbf{s}_t[i] = \mathrm{ReLU}\left(\mathbf{s}_{t-1}[i] - \mathbf{u}_t[i]\right)$$

The $\mathbf{u}_t[i]$s are computed as follows. The total amount of strength to be reduced is the *pop strength* u_t. Popping begins by attempting to reduce the strength $\mathbf{s}_t[t-1]$ of the top item on the stack by the full pop strength u_t. Thus, as shown below, $\mathbf{u}_t[t-1] = u_t$. For each i, if $\mathbf{s}_{t-1}[i] < \mathbf{u}_t[i]$, then the ith item has been fully popped from the stack, "consuming" a portion of the pop strength of magnitude $\mathbf{s}_{t-1}[i]$. The strength of the next item is then reduced by an amount $\mathbf{u}_t[i-1]$ given by the "remaining" pop strength $\mathbf{u}_t[i] - \mathbf{s}_{t-1}[i]$.

$$\mathbf{u}_t[i] = \begin{cases} u_t, & i = t-1 \\ \mathrm{ReLU}(\mathbf{u}_t[i+1] - \mathbf{s}_{t-1}[i+1]), & i < t-1 \end{cases}$$

The pushing operation simply places the vector $\mathbf{v}_t$ at the top of the stack with strength d_t. Thus,

$\mathbf{V}_t$ and $\mathbf{s}_t[t]$ are updated as follows.

$$\mathbf{s}_t[t] = d_t \qquad \mathbf{V}_t[i] = \begin{cases} \mathbf{v}_t, & i = t \\ \mathbf{V}_{t-1}[i], & i < t \end{cases}$$

Note that $\mathbf{s}_t[1]$, $\mathbf{s}_t[2]$, …, $\mathbf{s}_t[t-1]$ have already been updated during the popping step.

Finally, the reading operation produces a "summary" of the top of the stack by computing a weighted sum of all the vectors on the stack.

$$\mathbf{r}_t = \sum_{i=1}^{t} \min\left(\mathbf{s}_t[i], \rho_t[i]\right) \cdot \mathbf{V}_t[i]$$

The weights $\rho_t[i]$ are computed in a manner similar to the $\mathbf{u}_t[i]$s. The sum should include the top elements of the stack whose strengths add up to the *read strength* r_t. The weight $\rho_t[t]$ assigned to the top item is initialized to the full read strength r_t, while the weights $\rho_t[i]$ assigned to lower items are based on the "remaining" read strength $\rho_t[i+1] - \mathbf{s}_t[i+1]$ after strength has been assigned to higher items.

$$\rho_t[i] = \begin{cases} r_t, & i = t \\ \operatorname{ReLU}\left(\rho_t[i+1] - \mathbf{s}_t[i+1]\right) & i < t \end{cases}$$

2.2 Stack Interface

The architecture of Grefenstette et al. (2015) assumes that the controller is a neural network of the form

$$\langle \mathbf{o}_t, \mathbf{h}_t \rangle = C(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{r}_{t-1})$$

where $\mathbf{h}_t$ is its state at time t, $\mathbf{x}_t$ is its input, $\mathbf{r}_t$ is the vector read from the stack at the previous step, and $\mathbf{o}_t$ is an output vector used to produce the network output $\mathbf{y}_t$ and the stack instructions $\langle \mathbf{v}_t, u_t, d_t, r_t \rangle$.

The stack instructions $\langle \mathbf{v}_t, u_t, d_t, r_t \rangle$ are computed as follows. The read strength r_t is fixed to 1. The other values are determined by passing $\mathbf{o}_t$ to specialized layers. The vectors $\mathbf{y}_t$ and $\mathbf{v}_t$ are computed using a $\tanh$ layer, while the scalar values u_t and d_t are obtained from a sigmoid layer. Thus, the push and pop strengths are constrained to values between 0 and 1.

$$\begin{aligned} \mathbf{y}_t &= \operatorname{softmax}\left(\mathbf{W}^y \mathbf{o}_t + \mathbf{b}^y\right) \\ \mathbf{v}_t &= \tanh\left(\mathbf{W}^v \mathbf{o}_t + \mathbf{b}^v\right) \\ u_t &= \sigma\left(\mathbf{W}^u \mathbf{o}_t + b_u\right) \\ d_t &= \sigma\left(\mathbf{W}^d \mathbf{o}_t + b_d\right) \\ r_t &= 1 \end{aligned} \tag{1}$$

This paper departs from Grefenstette et al.'s architecture by allowing for push, pop, and read operations to be executed with variable strength greater than 1. We achieve this by using an enhanced control interface inspired by Yogatama et al.'s (2018) Multipop Adaptive Computation Stack architecture. In that model, the controller determines how much weight to pop from the stack at each time step by computing a distribution $\mathbb{P}[u]$ describing the probability of popping u units from the stack. The next stack state $\mathbf{V}$ is computed as a superposition of the possible stack states $\mathbf{V}^u$ resulting from popping u units from the stack, weighted by $\mathbb{P}[u]$. Our model follows Yogatama et al. in computing probability distributions over possible values of u_t, d_t, and r_t. However, instead of superimposing stack states, which may hinder interpretability, we simply set the value of each instruction to be the expected value of its associated distribution. For a distribution vector $\mathbf{p}$, define the operator $\mathbb{E}[\mathbf{p}]$ as follows:

$$\mathbb{E}[\mathbf{p}] = \sum_{i=0}^{k} i\mathbf{p}[i+1]$$

$\mathbb{E}[\mathbf{p}]$ denotes the expected value of $\mathbf{p}$ if we treat it as a distribution over $\{0, 1, \ldots, k\}$. The maximum value k is fixed in advance as a hyperparameter of our model. The output $\mathbf{y}_t$ and instructions $\mathbf{v}_t$, u_t, d_t, and r_t are then computed as follows:

$$\begin{aligned} \mathbf{y}_t &= \operatorname{softmax}\left(\mathbf{W}^y \mathbf{o}_t + \mathbf{b}^y\right) \\ \mathbf{v}_t &= \tanh\left(\mathbf{W}^v \mathbf{o}_t + \mathbf{b}^v\right) \\ u_t &= \mathbb{E}\left[\operatorname{softmax}\left(\mathbf{W}^u \mathbf{o}_t + b^u\right)\right] \\ d_t &= \mathbb{E}\left[\operatorname{softmax}\left(\mathbf{W}^d \mathbf{o}_t + b^u\right)\right] \\ r_t &= \mathbb{E}\left[\operatorname{softmax}\left(\mathbf{W}^r \mathbf{o}_t + b^r\right)\right] \end{aligned}$$

The full architecture that we used for language modeling and agreement classification is a controller network which, at time t, reads the word $\mathbf{x}_t$ as well as the previous stack summary $\mathbf{r}_{t-1}$. These vectors are passed through an LSTM layer to produce the vector $\mathbf{o}_t$. Then, instructions for the stack are computed from $\mathbf{o}_t$ according to the equations above. Finally, these instructions are executed to modify the stack state and produce the next stack summary vector $\mathbf{r}_t$. In our experiments, the size of the LSTM layer was 100, and the size of each stack vector was 16.

3 Model Training

This paper considers models trained on a language modeling objective and a classification objective. On each objective, we train several neural stack models along with an LSTM baseline.[1] This section describes the procedure used to train our models and presents the perplexity and classification values they attain on their training objectives.

3.1 Data and Training

Our models are trained using the *Wikipedia corpus*, a subset of the English Wikipedia used by Linzen et al. (2016) for their experiments. The classification task we consider is the *number prediction task*, proposed by Linzen et al. (2016) as a diagnostic for assessing whether or not LSTMs can infer grammatical dependencies sensitive to syntactic structure. In this task, the network is shown a sequence of words forming the beginning of a sentence from the Wikipedia corpus. The next word in the sentence is always a verb, and the network must predict whether the verb is singular (SG) or plural (PL). For example, on input *The cats on the boat*, the network must predict PL to match *cats*. We train and evaluate our models on the number prediction task using Linzen et al.'s (2016) *simple dependency dataset*, which contains 141,948 training examples, 15,772 validation examples, and 1,419,491 testing examples.

We used a model with very few parameters and basic setting of hyperparameters. The LSTM hidden state was fixed to a size of 100, while the vectors placed on the stack had size 16. Including the embedding layer, the Wikipedia model had 1,584,255 parameters. We used the Adam optimizer (Kingma and Ba, 2015) with a learning rate of 0.001. The language models were trained for five epochs, while the agreement classifiers used an early stopping criterion. In addition to the LSTM baseline, for each task, we trained a stack RNN in which u_t is fixed to 1 and d_t ranges from 0 to $k = 4$, as well as a stack RNN in which d_t fixed to 1 and u_t ranges from 0 to $k = 4$. Additionally, for the classification task we trained a stack RNN in which u_t ranges from 0 to $k = 4$ and d_t is computed as in Equation 1.

[1]Our code is available at `https://github.com/viking-sudo-rm/industrial-stacknns`.

	Stack ($u_t = 1$)	**Stack** ($d_t = 1$)	**LSTM**
Perp	92.81	128.28	**91.69**
Agree	93.59	92.28	**93.95**

Table 1: Results for language models trained on the Wikipedia dataset.

3.2 Evaluation

Our language models are evaluated according to two metrics. Firstly, we reserve 10% of the Wikipedia corpus for evaluating test perplexity of the trained language models. Secondly, as a simple diagnostic of sensitivity to syntactic structure, we evaluate the performance of our Wikipedia-trained language models on *number agreement prediction* (Linzen et al., 2016). Under this evaluation regime, we use our language model to simulate the number prediction task and compute the resulting classification accuracy. We do this by presenting the model with an input for the number prediction task and comparing the probabilities assigned to the verb that follows the input in the Wikipedia corpus. For example, if *The cats on the boat purr* appears in the Wikipedia corpus, then we present *The cats on the boat* to the language model and compare the probabilities assigned to the singular and plural forms *purrs* and *purr*, respectively. We consider the language model to make a correct prediction if the form of the next lexical item with the correct grammatical number (SG or PL) is predicted with greater probability than the alternative.

The number prediction classifiers we trained are evaluated according to classification accuracy. For each input sentence, we define the *attractors* of the input to be the nouns intervening between the subject and the verb whose number is being classified. For example, in the input *The cat on the boat*, *cat* is the subject of the following verb, while *boat* is an attractor. We compute the accuracy of our classifiers on the full testing set of the simple dependency data set as well as subsets of the testing set consisting of sentences with a fixed number of attractors.

3.3 Training Results

Table 1 shows the quantitative results for our language models. The stack RNN is comparable to our LSTM baseline in terms of language modeling perplexity and agreement prediction accuracy when u_t is fixed to 1, though the latter per-

Number of Attractors	Stack ($u_t = 1$)	Stack ($d_t = 1$)	Stack	LSTM
Overall	**98.89**	98.88	98.88	**98.89**
0	**99.29**	99.23	99.24	99.26
1	94.75	**95.43**	95.18	95.27
2	89.85	91.70	**91.86**	90.15
3	83.42	86.59	**87.47**	84.30
4	79.50	84.14	**85.56**	78.61
5	71.07	71.70	**77.99**	74.21

Table 2: Number prediction accuracies attained by the three stack RNN classifiers and the LSTM baseline.

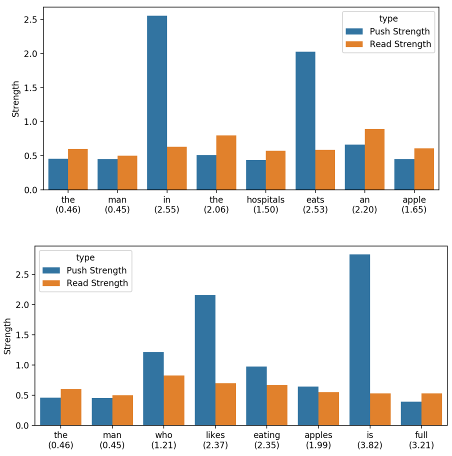


Figure 2: Push and read strengths computed by the $u_t = 1$ language model. Values underneath each word show the total strength remaining on the stack at that step.

forms slightly better according to both metrics. The stack RNN attains a significantly worse perplexity when d_t is fixed to 1, and its agreement prediction accuracy is worse than both the LSTM baseline and the stack RNN with $u_t = 1$.

Table 2 shows test accuracies attained by classifiers trained on the number prediction task. While the stack classifier with u_t fixed to 1 and the LSTM baseline achieve the best overall accuracy, the stack with unrestricted u_t and sigmoid d_t and the stack with d_t fixed to 1 exceed the baseline on sentences with at least 2 attractors. We take this to suggest that the hierarchical bias provided by the stack can improve performance on syntactically complex cases.

4 Interpreting Stack Usage

The results presented in Subsection 3.3 show that the $u_t = 1$ stack RNNs perform comparably to LSTMs in terms of quantitative evaluation metrics. The goal of this section is to assess whether or not stack RNNs achieve this level of performance in an interpretable manner. We do this by visualizing the push and read strengths computed by the $u_t = 1$ language model when processing two example sentences. These visualizations are shown in Figure 2 and Figure 3. Notice that the push strength tends to spike following words with subcategorization requirements. For example, the preposition *in* and the transitive verbs *eat* and *is* both require NP objects, and accordingly the model assigns a high push strength to these words. This suggests that the model is using the stack to capture hierarchical dependencies by keeping track of words that predictably introduce various kinds of phrases.

Figure 4 shows push strengths computed by the $u_t = 1$ language model, aggregated across the entire Wikipedia corpus. We see that push strengths differ systematically based on part of speech. The distribution of push strengths computed by the network upon seeing a noun is tightly concentrated around 0.5, whereas the push strength upon seeing a verb tends to be greater—usually more than 2.5. This phenomenon reflects the fact that verbs typically take objects while nouns do not.

We also find that push strengths assigned to verbs depend on their transitivity. The right panel of Figure 4 shows push strength distributions for a collection of common transitive and intransitive verbs. The model distinguishes between these two types of verbs by assigning high push strengths to transitive verbs and low push strengths to intransitive verbs. We make similar observations for other parts of speech: prepositions, which take objects, typically receive higher push strengths, while determiners and adjectives, which do not take phrasal complements, receive lower push strengths.

5 Inference of Syntactic Structure

Section 4 has shown that the push strengths d_t computed by the $u_t = 1$ language model reflect the subcategorization requirements of the words encountered by the network. Based on this phenomenon, we may interpret the stack to be keeping track of phrases that are "in progress." A high push strength induced by a transitive verb, for example, may be thought to indicate that a verb phrase has begun, and that this phrase ends when the object of the verb is seen. We thus hypothesize that for each time step t, d_t represents the size of the phrase that begins with the word read by the network at time

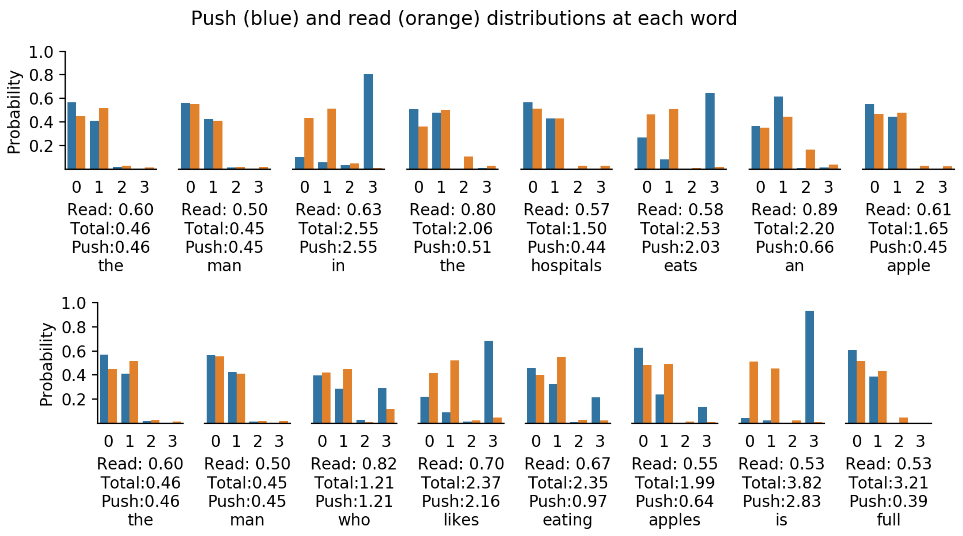
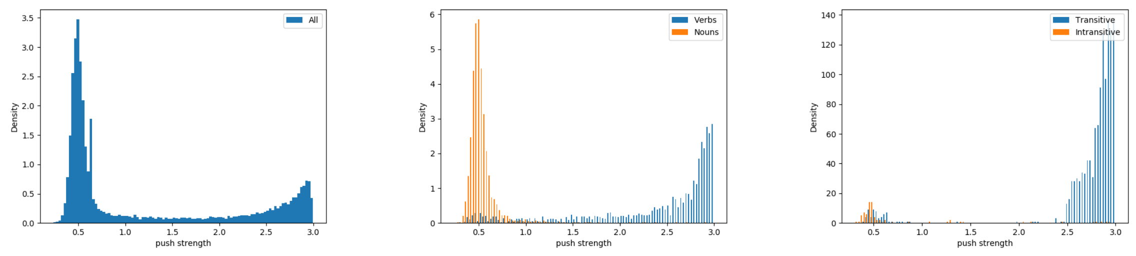


Figure 3: Distributions for push and read strengths at each step of processing example sentences. For example, the push strength chosen after processing *the* (0.46) is the expected value of the blue distribution in the far left plot.



All Words　Nouns and Verbs　Transitive vs. Intransitive Verbs

Figure 4: Distributions of d_t for the $u_t = 1$ language model over all test sentences. The center panel shows the distributions of d_t for nouns and verbs, and the right panel shows the distributions for selected transitive and intransitive verbs.

t. If d_t is low, then this phrase consists of a single word; if it is high, then this is a complex phrase consisting of multiple words.

A similar intuition underlies the unsupervised parsing framework of Shen et al. (2018a,b). Under this framework, constituency structure is induced from a sequence of words by computing a *syntactic distance* between every two adjacent words. Intuitively, the syntactic distance between two words measures the distance from the lowest common parent node of the two words to the bottom of the tree. If two words have a low syntactic distance, then they are likely siblings in a small constituent; if they have a high syntactic distance, then they probably belong to different phrases. Whereas Figure 2 and Figure 3 allow us identify specific time steps at which the stack recognizes the beginning of a phrase, the unsupervised parsing framework allows us to explicitly visualize the phrasal organization of input sequences induced by our interpretation of the push strengths.

Given an input sequence $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$, we define the syntactic distance between each $\mathbf{x}_t$ and $\mathbf{x}_{t-1}$ for our $u_t = 1$ model to be the push strength d_t computed by the controller during time t. If the current word does not open any new constituents, then it belongs to the same constituent as the previous word, and therefore should be assigned a low syntactic distance. On the other hand, if the current word opens a complex constituent, then it is lower in the parse tree than the previous word, and therefore should be assigned a high syntactic distance. Similarly, for our $d_t = 1$ model, we let u_t be the syntactic distance between $\mathbf{x}_t$ and $\mathbf{x}_{t+1}$. Under this interpretation, the pop strength estimates the complexity of the constituents that the current word closes. If the current word closes many complex constituents, then the next word appears at a higher level in the parse tree, and is therefore syntactically distant from the current word.

Algorithm 1 (Shen et al., 2018a,b)

```
1:  procedure MakeTree($\mathbf{X}, \mathbf{d}$)
2:      if $\mathbf{X}$ has at most one word then
3:          return $\mathbf{X}$
4:      else
5:          $i \leftarrow \text{argmax}_j\, \mathbf{d}_j$
6:          $l \leftarrow \text{MakeTree}(\mathbf{X}[:i-1], \mathbf{d}[:i-1])$
7:          $r \leftarrow \text{MakeTree}(\mathbf{X}[i+1:], \mathbf{d}[i+1:])$
8:          if $l$ and $r$ are not empty then
9:              return $\text{Tree}[l, \text{Tree}[\mathbf{X}[i], r]]$
10:         else if $l$ is empty then
11:             return $\text{Tree}[\mathbf{X}[i], r]$
12:         else
13:             return $\text{Tree}[l, \mathbf{X}[i]]$
```

Algorithm 1 shows our procedure for constructing trees. The algorithm takes as input a sequence of words arranged into a matrix $\mathbf{X}$ and a vector $\mathbf{d}$ containing the syntactic distance between each word and the previous word. Following Shen et al. (2018a,b), we recursively split $\mathbf{X}$ into binary constituents. At each recursion level, we greedily choose the word with the highest syntactic distance as the split point. The final output is a binary tree spanning the full sentence.

5.1 Evaluation

We compute F1 scores for the parses obtained from our Wikipedia language models by comparing against parses from Section 23 of the Penn Treebank's Wall Street Journal corpus (WSJ23, Marcus et al., 1994). Since Algorithm 1 produces unlabeled binary trees, our evaluation uses the gold standard of Htut et al. (2018), which consists of unlabeled, binarized versions of the WSJ23 trees. We also decapitalize the first word of every sentence for compatibility with our training data.

As a baseline, we the F1 scores attained by our models to those computed for purely right- and left-branching trees. A right-branching parse is equivalent to the output of Algorithm 1 on a sequence of equal syntactic distances. Thus, the difference between the right-branching F1 score and our models' scores is a measure of the amount of syntactic information encoded by the push and pop strength sequences. We also compare our F1 scores to the results of Htut et al.'s (2018) replication study for the *parsing–reading–predict network* models (PRPN-LM and PRPN-UP), the two syntactic-distance-based unsupervised parsers originally proposed by Shen et al. (2018a).

Model	**Parsing F1**
Stack ($u_t = 1$)	31.2
Stack ($d_t = 1$)	16.0
Right Branching	13.1
Left Branching	7.3
Best PRPN-UP (Htut et al., 2018)	26.3
Best PRPN-LM (Htut et al., 2018)	37.4

Table 3: Unsupervised parsing performance evaluated on the WSJ23 dataset, attained by our stack models (top), the right- and left-branching baselines (middle), and the PRPN models (bottom).

5.2 Results

The F1 evaluation (see Table 3) shows that our Wikipedia model with $u_t = 1$ significantly outperforms the baseline on the Penn Treebank, while our model with $d_t = 1$ performs slightly better than the baseline. This is evidence that the types of hierarchical structures produced by Algorithm 1 resemble expert-annotated constituency parses.

Our results do not exceed those of Htut et al.'s (2018) replication study. It is worth noting that our right- and left-branching baseline scores are somewhat lower than theirs. This suggests that differences in data processing or implementation might make our evaluation more difficult. Regardless, we consider our results to still be somewhat competitive, given that our language models were trained on out-of-domain data with few parameters and minimal hyperparameter tuning.

We provide example parses extracted from the stack RNN language models with $u_t = 1$ in Figure 5. Overall, our unsupervised parses tend to resemble the gold-standard parses with some differences. Periods in our parses systematically attach lower in the structure in our extracted parses than in the gold-standard trees. High attachment would require a high syntactic distance (i.e., high push strength) between the period and the remainder of the sentence. However, the period inherently does not have any subcategorization requirements, so it induces a low push strength. In contrast, prepositional phrases attach higher in our structures than in the gold parses. This may be the result of fixed subcategorization-associated push strengths for prepositions that give rise to fairly high esti-

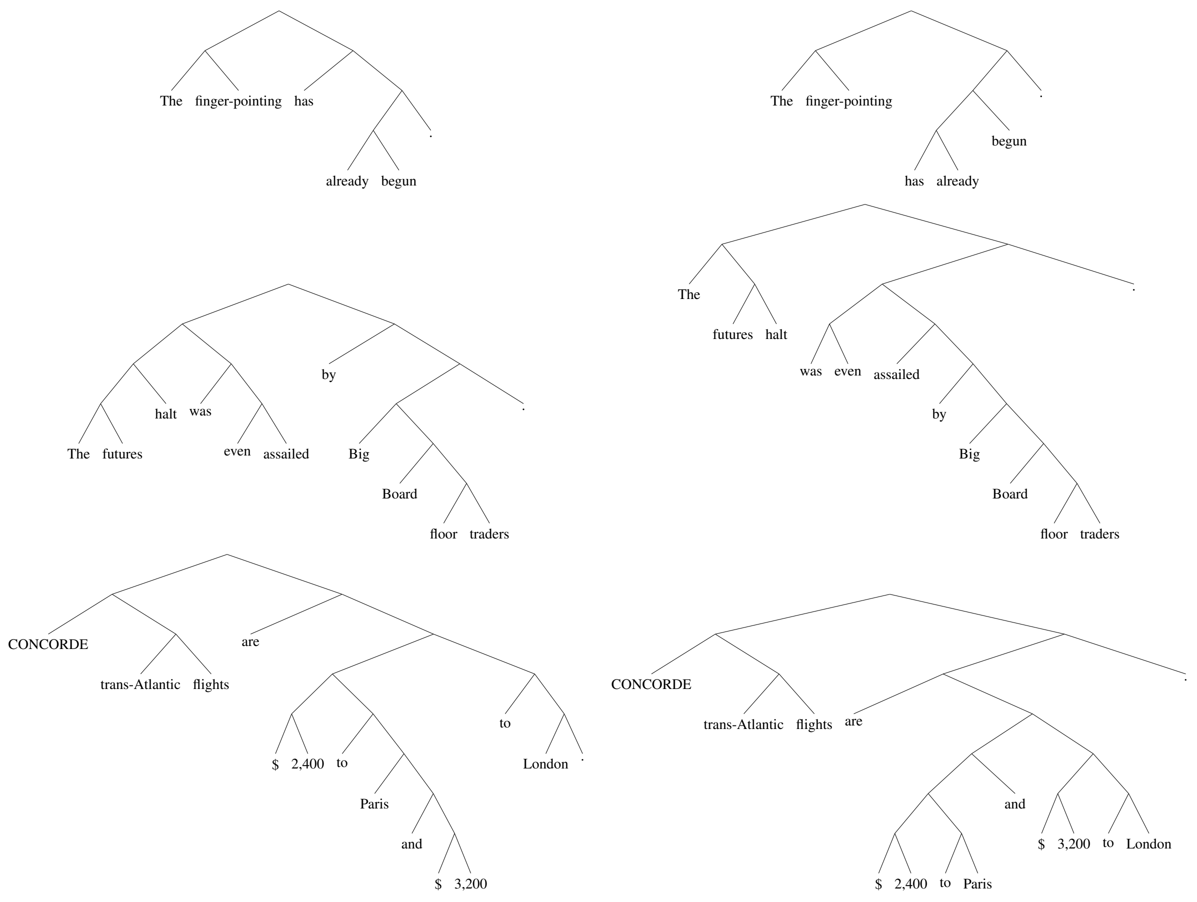


Figure 5: Sample parses obtained from our stack RNN language model with $u_t = 1$ (left), compared to Htut et al.'s (2018) gold-standard parses (right).

mates of syntactic distance.

6 Discussion

Overall, our stack language models show no improvement over the LSTM baseline in terms of perplexity and classification accuracy. Although the $u_t = 1$ language model is comparable to the LSTM on these metrics, it ultimately achieves worse scores than the baseline. However, we have now seen that the pushing behavior of the model reflects subcategorization properties of lexical items that play an important role in determining their syntactic behavior, and that these properties allow reasonable parses to be extracted from this model. These observations show that the $u_t = 1$ model has learned to encode structural representations using the stack. Quantitatively, the importance of this structural information for the training objectives can be seen in Table 2, where the stack at least partially alleviates the difficulty experienced by the LSTM classifier in handling syntactically complex inputs.

While our stack language models do not exceed the LSTM baseline in terms of perplexity and agreement accuracy, Yogatama et al. (2018) find that their Multipop Adaptive Computation Stack architecture substantially outperforms a bare LSTM on these metrics. Compared to their models, we use fewer parameters and minimal hyperparameter tuning. Thus, it is possible that increasing the number of parameters in our controller may lead to similar increases in performance in addition to the structural interpretability that we have observed.

7 Conclusion

The results reported here point to the conclusion that stack RNNs trained on corpora of natural language text do in fact learn to encode sentences in a hierarchically organized fashion. We show that the sequence of stack operations used in the processing of a sentence lets us uncover a syntactic structure that matches standardly assigned structure reasonably well, even if the addition of the stack does not improve the stack RNN's performance over the LSTM baseline in terms of the language modeling objective. We also find that using the stack RNN to predict the grammatical number of a verb results in better hierarchical generalizations in syntactically complex cases than is possible with stackless models. Taken together, these results suggest that the stack RNN model yields comparable performance to other architectures, while producing structural representations that are easier to interpret and that show signs of being linguistically natural.

References

Noam Chomsky. 1956. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124.

Noam Chomsky. 1957. *Syntactic Structures*, 1 edition. Mouton, The Hague, Netherlands.

Noam Chomsky. 1962. Context-free grammars and pushdown storage. Technical Report 65, MIT Research Laboratory for Electronics, Cambridge, MA.

Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. 2015. Learning to Transduce with Unbounded Memory. *Computing Research Repository*, arXiv:1506.02516v3.

Kristina Gulordava, Piotr Bojanowski, Edouard Grave, Tal Linzen, and Marco Baroni. 2018. Colorless Green Recurrent Networks Dream Hierarchically. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, volume 1, pages 1195–1205, New Orleans, LA. Association for Computational Linguistics.

Yiding Hao, William Merrill, Dana Angluin, Robert Frank, Noah Amsel, Andrew Benz, and Simon Mendelsohn. 2018. Context-free transductions with neural stacks. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 306–315, Brussels, Belgium. Association for Computational Linguistics.

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

Phu Mon Htut, Kyunghyun Cho, and Samuel Bowman. 2018. Grammar induction with neural language models: An unusual replication. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 371–373, Brussels, Belgium. Association for Computational Linguistics.

Armand Joulin and Tomas Mikolov. 2015. Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets. In *Advances in Neural Information Processing Systems 28*, pages 190–198, Montreal, Canada. Curran Associates, Inc.

Diederik P. Kingma and Jimmy Lei Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR 2015 Conference Track*, San Diego, CA. arXiv.

Tal Linzen, Emmanuel Dupoux, and Yoav Goldberg. 2016. Assessing the Ability of LSTMs to Learn Syntax-Sensitive Dependencies. *Transactions of the Association for Computational Linguistics*, 4(0):521–535.

Mitchell Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. 1994. The Penn Treebank: Annotating Predicate Argument Structure. In *Proceedings of the Workshop on Human Language Technology*, pages 114–119, Plainsboro, NJ. Association for Computational Linguistics.

Rebecca Marvin and Tal Linzen. 2018. Targeted Syntactic Evaluation of Language Models. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1192–1202, Brussels, Belgium. Association for Computational Linguistics.

Yikang Shen, Zhouhan Lin, Chin-wei Huang, and Aaron Courville. 2018a. Neural Language Modeling by Jointly Learning Syntax and Lexicon. In *ICLR 2018 Conference Track*, Vancouver, Canada. OpenReview.

Yikang Shen, Zhouhan Lin, Athul Paul Jacob, Alessandro Sordoni, Aaron Courville, and Yoshua Bengio. 2018b. Straight to the Tree: Constituency Parsing with Neural Syntactic Distance. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, volume 1: Long Papers, pages 1171–1180, Melbourne, Australia. Association for Computational Linguistics.

Dani Yogatama, Yishu Miao, Gabor Melis, Wang Ling, Adhiguna Kuncoro, Chris Dyer, and Phil Blunsom. 2018. Memory Architectures in Recurrent Neural Network Language Models. In *ICLR 2018 Conference Track*, Vancouver, Canada. OpenReview.