A Fast Unified Model for Parsing and Sentence Understanding

Samuel R. Bowman[1,2,3,*] sbowman@stanford.edu
Jon Gauthier[2,3,4,*] jgauthie@stanford.edu
Abhinav Rastogi[3,5] arastogi@stanford.edu
Raghav Gupta[2,3,6] rgupta93@stanford.edu
Christopher D. Manning[1,2,3,6] manning@stanford.edu
Christopher Potts[1,6] cgpotts@stanford.edu

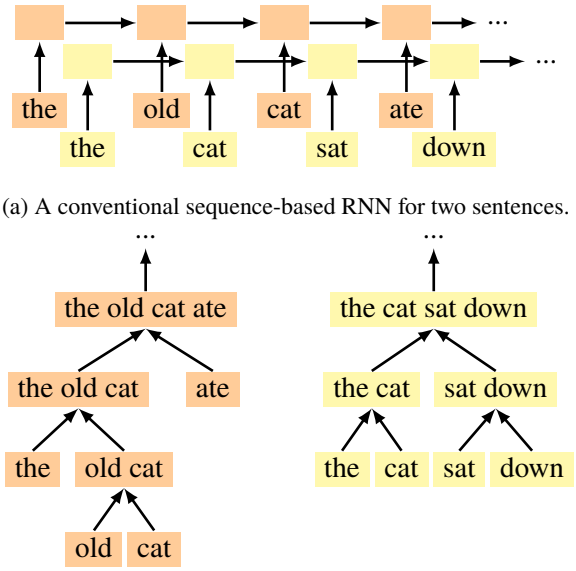
[1]Stanford Linguistics [2]Stanford NLP Group [3]Stanford AI Lab
[4]Stanford Symbolic Systems [5]Stanford Electrical Engineering [6]Stanford Computer Science

Abstract

Tree-structured neural networks exploit valuable syntactic parse information as they interpret the meanings of sentences. However, they suffer from two key technical problems that make them slow and unwieldy for large-scale NLP tasks: they usually operate on parsed sentences and they do not directly support batched computation. We address these issues by introducing the Stack-augmented Parser-Interpreter Neural Network (SPINN), which combines parsing and interpretation within a single tree-sequence hybrid model by integrating tree-structured sentence interpretation into the linear sequential structure of a shift-reduce parser. Our model supports batched computation for a speedup of up to 25× over other tree-structured models, and its integrated parser can operate on unparsed data with little loss in accuracy. We evaluate it on the Stanford NLI entailment task and show that it significantly outperforms other sentence-encoding models.

1 Introduction

A wide range of current models in NLP are built around a neural network component that produces vector representations of sentence meaning (e.g., Sutskever et al., 2014; Tai et al., 2015). This component, the sentence encoder, is generally formulated as a learned parametric function from a sequence of word vectors to a sentence vector, and this function can take a range of different forms. Common sentence encoders include sequence-based recurrent neural network models (RNNs, see Figure 1a) with Long Short-Term Memory (LSTM, Hochreiter and Schmidhuber, 1997), which accumulate information over the sentence sequentially; convolutional neural networks (Kalchbrenner et al., 2014; Zhang et al., 2015), which accumulate information using filters over short local sequences of words or characters; and tree-structured recursive neural networks (TreeRNNs, Goller and Küchler, 1996; Socher et al., 2011a, see Figure 1b), which propagate information up a binary parse tree.



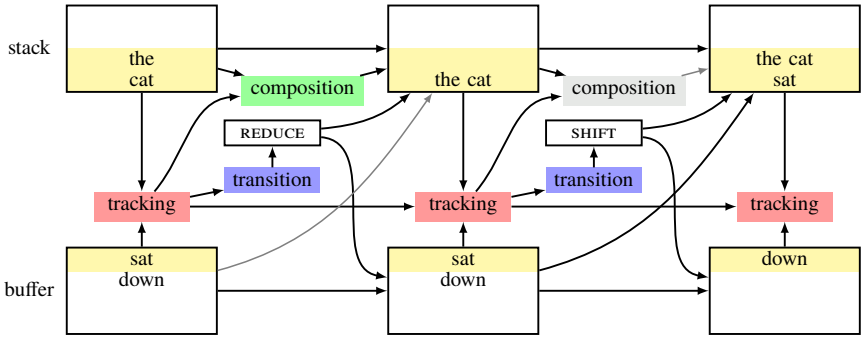
(a) A conventional sequence-based RNN for two sentences.

(b) A conventional TreeRNN for two sentences.

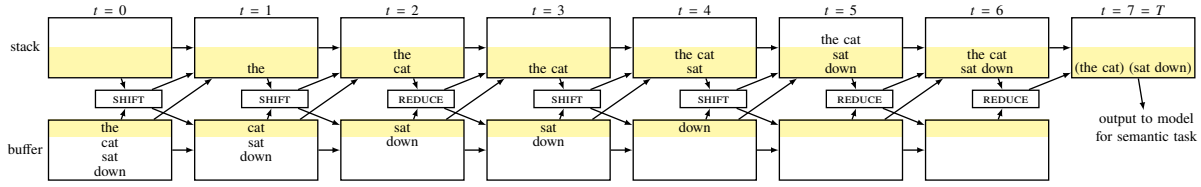
Figure 1: An illustration of two standard designs for sentence encoders. The TreeRNN, unlike the sequence-based RNN, requires a substantially different connection structure for each sentence, making batched computation impractical.

Of these, the TreeRNN appears to be the principled choice, since meaning in natural language sentences is known to be constructed recursively according to a tree structure (Dowty, 2007, i.a.). TreeRNNs have shown promise (Tai et al., 2015; Li et al., 2015; Bowman et al., 2015b), but have

*The first two authors contributed equally.



(a) The SPINN model unrolled for two transitions during the processing of the sentence *the cat sat down*. 'Tracking', 'transition', and 'composition' are neural network layers. Gray arrows indicate connections which are blocked by a gating function.



(b) The fully unrolled SPINN for *the cat sat down*, with neural network layers omitted for clarity.

Figure 2: Two views of the Stack-augmented Parser-Interpreter Neural Network (SPINN).

largely been overlooked in favor of sequence-based RNNs because of their incompatibility with batched computation and their reliance on external parsers. Batched computation—performing synchronized computation across many examples at once—yields order-of-magnitude improvements in model run time, and is crucial in enabling neural networks to be trained efficiently on large datasets. Because TreeRNNs use a different model structure for each sentence, as in Figure 1, efficient batching is impossible in standard implementations. Partly to address efficiency problems, standard TreeRNN models commonly only operate on sentences that have already been processed by a syntactic parser, which slows and complicates the use of these models at test time for most applications.

This paper introduces a new model to address both these issues: the Stack-augmented Parser-Interpreter Neural Network, or SPINN, shown in Figure 2. SPINN executes the computations of a tree-structured model in a linearized sequence, and can incorporate a neural network parser that produces the required parse structure on the fly. This design improves upon the TreeRNN architecture in three ways: At test time, it can simultaneously parse and interpret unparsed sentences, removing the dependence on an external parser at nearly no additional computational cost. Secondly, it supports batched computation for both parsed and unparsed sentences, yielding dramatic speedups over standard TreeRNNs. Finally, it supports a novel tree-sequence hybrid architecture for handling local linear context in sentence interpretation. This model is a basically plausible model of human sentence processing and yields substantial accuracy gains over pure sequence- or tree-based models.

We evaluate SPINN on the Stanford Natural Language Inference entailment task (SNLI, Bowman et al., 2015a), and find that it significantly outperforms other sentence-encoding-based models, even with a relatively simple and underpowered implementation of the built-in parser. We also find that SPINN yields speed increases of up to 25× over a standard TreeRNN implementation.

2 Related work

There is a fairly long history of work on building neural network-based parsers that use the core operations and data structures from transition-based parsing, of which shift-reduce parsing is a variant (Henderson, 2004; Emami and Jelinek, 2005; Titov and Henderson, 2010; Chen and Manning, 2014; Buys and Blunsom, 2015; Dyer et al., 2015; Kiperwasser and Goldberg, 2016). In addition, there has been recent work proposing models designed primarily for generative language modeling tasks that use this architecture as well (Zhang et al., 2016; Dyer et al., 2016). To our knowledge, SPINN is the first model to use this architecture for the purpose of sentence interpretation, rather than parsing

or generation.

Socher et al. (2011a,b) present versions of the TreeRNN model which are capable of operating over unparsed inputs. However, these methods require an expensive search process at test time. Our model presents a much faster alternative approach.

3 Our model: SPINN

3.1 Background: Shift-reduce parsing

SPINN is inspired by shift-reduce parsing (Aho and Ullman, 1972), which builds a tree structure over a sequence (e.g., a natural language sentence) by a single left-to-right scan over its tokens. The formalism is widely used in natural language parsing (e.g., Shieber, 1983; Nivre, 2003).

A shift-reduce parser accepts a sequence of input tokens $\mathbf{x} = (x_0, \ldots, x_{N-1})$ and consumes transitions $\mathbf{a} = (a_0, \ldots, a_{T-1})$, where each $a_t \in \{\text{SHIFT}, \text{REDUCE}\}$ specifies one step of the parsing process. In general a parser may also generate these transitions on the fly as it reads the tokens. It proceeds left-to-right through a transition sequence, combining the input tokens $\mathbf{x}$ incrementally into a tree structure. For any binary-branching tree structure over N words, this requires $T = 2N - 1$ transitions through a total of $T + 1$ states.

The parser uses two auxiliary data structures: a stack S of partially completed subtrees and a buffer B of tokens yet to be parsed. The parser is initialized with the stack empty and the buffer containing the tokens $\mathbf{x}$ of the sentence in order. Let $\langle S, B\rangle = \langle \emptyset, \mathbf{x}\rangle$ denote this starting state. It next proceeds through the transition sequence, where each transition a_t selects one of the two following operations. Below, the | symbol denotes the *cons* (concatenation) operator. We arbitrarily choose to always *cons* on the left in the notation below.

SHIFT: $\langle S, x \mid B\rangle \rightarrow \langle x \mid S, B\rangle$. This operation pops an element from the buffer and pushes it on to the top of the stack.

REDUCE: $\langle x \mid y \mid S, B\rangle \rightarrow \langle (x, y) \mid S, B\rangle$. This operation pops the top two elements from the stack, merges them, and pushes the result back on to the stack.

3.2 Composition and representation

SPINN is based on a shift-reduce parser, but it is designed to produce a vector representation of a sentence as its output, rather than a tree as in standard shift-reduce parsing. It modifies the shift-reduce formalism by using fixed length vectors to represent each entry in the stack and the buffer. Correspondingly, its REDUCE operation combines two vector representations from the stack into another vector using a neural network function.

The composition function When a REDUCE operation is performed, the vector representations of two tree nodes are popped off of the stack and fed into a *composition function*, which is a neural network function that produces a representation for a new tree node that is the parent of the two popped nodes. This new node is pushed on to the stack.

The TreeLSTM composition function (Tai et al., 2015) generalizes the LSTM neural network layer to tree- rather than sequence-based inputs, and it shares with the LSTM the idea of representing intermediate states as a pair of an active state representation $\vec{h}$ and a memory representation $\vec{c}$. Our version is formulated as:

$$\text{(1)} \quad \begin{bmatrix} \vec{i} \\ \vec{f}_l \\ \vec{f}_r \\ \vec{o} \\ \vec{g} \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ \sigma \\ \tanh \end{bmatrix} \left(W_{\text{comp}} \begin{bmatrix} \vec{h}_s^1 \\ \vec{h}_s^2 \\ \vec{e} \end{bmatrix} + \vec{b}_{\text{comp}} \right)$$

$$\text{(2)} \quad \vec{c} = \vec{f}_l \odot \vec{c}_s^2 + \vec{f}_r \odot \vec{c}_s^1 + \vec{i} \odot \vec{g}$$

$$\text{(3)} \quad \vec{h} = \vec{o} \odot \vec{c}$$

where σ is the sigmoid activation function, $\odot$ is the elementwise product, the pairs $\langle \vec{h}_s^1, \vec{c}_s^1 \rangle$ and $\langle \vec{h}_s^2, \vec{c}_s^2 \rangle$ are the two input tree nodes popped off the stack, and $\vec{e}$ is an optional vector-valued input argument which is either empty or comes from an external source like the tracking LSTM (see Section 3.3). The result of this function, the pair $\langle \vec{h}, \vec{c} \rangle$, is placed back on the stack. Each vector-valued variable listed is of dimension D except $\vec{e}$, of the independent dimension D_{tracking}.

The stack and buffer The stack and the buffer are arrays of N elements each (for sentences of up to N words), with the two D-dimensional vectors $\vec{h}$ and $\vec{c}$ in each element.

Word representations We use word representations based on the 300D vectors provided with GloVe (Pennington et al., 2014). We do not update these representations during training. Instead, we use a learned linear transformation to map each input word vector $\vec{x}_{\text{GloVe}}$ into a vector pair $\langle \vec{h}, \vec{c} \rangle$ that is stored in the buffer:

$$\text{(4)} \quad \begin{bmatrix} \vec{h} \\ \vec{c} \end{bmatrix} = W_{\text{wd}} \vec{x}_{\text{GloVe}} + \vec{b}_{\text{wd}}$$

3.3 The tracking LSTM

In addition to the stack, buffer, and composition function, our full model includes an additional component: the tracking LSTM. This is a simple sequence-based LSTM RNN that operates in tandem with the model, taking inputs from the buffer and stack at each step. It is meant to maintain a low-resolution summary of the portion of the sentence that has been processed so far, which is used for two purposes: it supplies feature representations to the transition classifier, which allows the model to stand alone as a parser, and it additionally supplies a secondary input $\vec{e}$ to the composition function—see (1)—allowing context information to enter the construction of sentence meaning and forming what is effectively a tree-sequence hybrid model.

The tracking LSTM's inputs (yellow in Figure 2) are the top element of the buffer $\vec{h}_b^1$ (which would be moved in a SHIFT operation) and the top two elements of the stack $\vec{h}_s^1$ and $\vec{h}_s^2$ (which would be composed in a REDUCE operation).

Why a tree-sequence hybrid? Lexical ambiguity is ubiquitous in natural language. Most words have multiple senses or meanings, and it is generally necessary to use the context in which a word occurs to determine which of its senses or meanings is meant in a given sentence. Even though TreeRNNs are more effective at composing meanings in principle, this ambiguity can give simpler sequence-based sentence-encoding models an advantage: when a sequence-based model first processes a word, it has direct access to a state vector that summarizes the left context of that word, which acts as a cue for disambiguation. In contrast, when a standard tree-structured model first processes a word, it only has access to the constituent that the word is merging with, which is often just a single additional word. Feeding a context representation from the tracking LSTM into the composition function is a simple and efficient way to mitigate this disadvantage of tree-structured models. Using left linear context to disambiguate is also a plausible model of human interpretation.

It would be straightforward to augment SPINN to support the use of some amount of right-side context as well, but this would add complexity to the model that we think is largely unnecessary: humans are very effective at understanding the beginnings of sentences before having seen or heard the ends, suggesting that it is possible to get by without the unavailable right-side context.

3.4 Parsing: Predicting transitions

For SPINN to operate on unparsed inputs, it needs to produce its own transition sequence **a** rather than relying on an external parser to supply it as part of the input. To do this, the model predicts a_t at each step using a simple two-way softmax classifier whose input is the state of the tracking LSTM:

$$\vec{p}_{\text{a}} = \text{softmax}(W_{\text{trans}}\vec{h}_{\text{tracking}} + \vec{b}_{\text{trans}}) \quad (5)$$

The above model is nearly the simplest viable implementation of a transition decision function. In contrast, the decision functions in state-of-the-art transition-based parsers tend to use significantly richer feature sets as inputs, including features containing information about several upcoming words on the buffer. The value $\vec{h}_{\text{tracking}}$ is a function of only the very top of the buffer and the top two stack elements at each timestep.

At test time, the model uses whichever transition (i.e., SHIFT or REDUCE) is assigned a higher (unnormalized) probability. The prediction function is trained to mimic the decisions of an external parser. These decisions are used as inputs to the model during training. For SNLI, we use the binary Stanford PCFG Parser parses that are included with the corpus. We did not find scheduled sampling (Bengio et al., 2015)—having the model use its own transition decisions sometimes at training time—to help.

3.5 Implementation issues

Representing the stack efficiently A naïve implementation of SPINN needs to handle a size $O(N)$ stack at each timestep, any element of which may be involved in later computations. A naïve backpropagation implementation would then require storing each of the $O(N)$ stacks for a backward pass, leading to a per-example space requirement of $O(NTD)$ floats. This requirement is prohibitively large for significant batch sizes or sentence lengths N. Such a naïve implementation would also require copying a largely unchanged stack at each timestep, since each SHIFT or REDUCE operation writes only one new representation to the top of the stack.

We propose a space-efficient stack representation inspired by the zipper technique (Huet, 1997) that we call *thin stack*. For each input sentence, we

Algorithm 1 The thin stack algorithm

```
1: function STEP(bufferTop, a, t, S, Q)
2:     if a = SHIFT then
3:         S[t] := bufferTop
4:     else if a = REDUCE then
5:         right := S[Q.pop()]
6:         left := S[Q.pop()]
7:         S[t] := COMPOSE(left, right)
8:     Q.push(t)
```

represent the stack with a single $T \times D$ matrix S. Each row $S[t]$ (for $0 < t \leq T$) represents the top of the actual stack at timestep t. At each timestep we can SHIFT a new element onto the stack, or REDUCE the top two elements of the stack into a single element. To shift an element from the buffer to the top of the stack at timestep t, we simply write it into the location $S[t]$. In order to perform the REDUCE operation, we need to retrieve the top two elements of the actual stack. We maintain a queue Q of pointers into S which contains the row indices of S which are still present in the actual stack. The top two elements of the stack can be found by using the final two pointers in the queue Q. These retrieved elements are used to perform the REDUCE operation, which modifies Q to mark that some rows of S have now been replaced in the actual stack. Algorithm 1 describes the full mechanics of a stack feedforward in this compressed representation. It operates on the single $T \times D$ matrix S and a backpointer queue Q. Table 1 shows an example run.

This stack representation requires substantially less space. It stores each element involved in the feedforward computation exactly once, meaning that this representation can still support efficient backpropagation. Furthermore, all of the updates to S and Q can be performed batched and in-place on a GPU, yielding substantial speed gains over both a more naïve SPINN implementation and a standard TreeRNN implementation. We describe speed results in Section 3.7.

Preparing the data At training time, SPINN requires both a transition sequence **a** and a token sequence **x** as its inputs for each sentence. The token sequence is simply the words in the sentence in order. **a** can be obtained from any constituency parse for the sentence by first converting that parse into an unlabeled binary parse, then linearizing it (with the usual in-order traversal), then taking each word token as a SHIFT transition and each ')' as a REDUCE transition, as here:

Unlabeled binary parse: ((the cat) (sat down))
x: *the*, *cat*, *sat*, *down*
a: SHIFT, SHIFT, REDUCE, SHIFT, SHIFT, REDUCE, REDUCE

t	$S[t]$	Q_t	a_t
0		—	SHIFT
1	*Spot*	1	SHIFT
2	*sat*	1 2	SHIFT
3	*down*	1 2 3	REDUCE
4	(*sat down*)	1 4	REDUCE
5	(*Spot* (*sat down*))	5	

Table 1: The thin-stack algorithm operating on the input sequence **x** = (*Spot*, *sat*, *down*) and the transition sequence shown in the rightmost column. $S[t]$ shows the top of the stack at each step t. The last two elements of Q (underlined) specify which rows t would be involved in a REDUCE operation at the next step.

Handling variable sentence lengths For any sentence model to be trained with batched computation, it is necessary to pad or crop sentences to a fixed length. We fix this length at $N = 25$ words, longer than about 98% of sentences in SNLI. Transition sequences **a** are cropped at the left or padded at the left with SHIFTS. Token sequences **x** are then cropped or padded with empty tokens at the left to match the number of SHIFTS added or removed from **a**, and can then be padded with empty tokens at the right to meet the desired length N.

3.6 TreeRNN-equivalence

Without the addition of the tracking LSTM, SPINN (in particular the SPINN-PI-NT variant, for *parsed input, no tracking*) is precisely equivalent to a conventional tree-structured neural network model in the function that it computes, and therefore it also has the same learning dynamics. In both, the representation of each sentence consists of the representations of the words combined recursively using a TreeRNN composition function (in our case, the TreeLSTM function). SPINN, however, is dramatically faster, and supports both integrated parsing and a novel approach to context through the tracking LSTM.

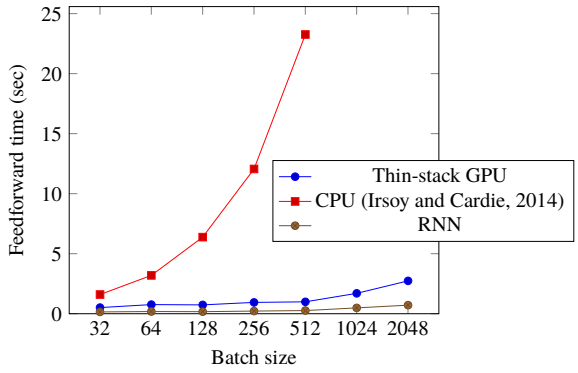


Figure 3: Feedforward speed comparison.

3.7 Inference speed

In this section, we compare the test-time speed of our SPINN-PI-NT with an equivalent TreeRNN implemented in the conventional fashion and with a standard RNN sequence model. While the full models evaluated below are implemented and trained using Theano (Theano Development Team, 2016), which is reasonably efficient but not perfect for our model, we wish to compare well-optimized implementations of all three models. To do this, we reimplement the feedforward[1] of SPINN-PI-NT and an LSTM RNN baseline in C++/CUDA, and compare that implementation with a CPU-based C++/Eigen TreeRNN implementation from Irsoy and Cardie (2014), which we modified to perform exactly the same computations as SPINN-PI-NT.[2] TreeRNNs like this can only operate on a single example at a time and are thus poorly suited for GPU computation.

Each model is restricted to run on sentences of 30 tokens or fewer. We fix the model dimension D and the word embedding dimension at 300. We run the CPU performance test on a 2.20 GHz 16-core Intel Xeon E5-2660 processor with hyperthreading enabled. We test our thin-stack implementation and the RNN model on an NVIDIA Titan X GPU.

Figure 3 compares the sentence encoding speed of the three models on random input data. We observe a substantial difference in runtime between the CPU and thin-stack implementations that increases with batch size. With a large but practical batch size of 512, the largest on which we tested the TreeRNN, our model is about 25× faster than the standard CPU implementation, and about 4× slower than the RNN baseline.

Though this experiment only covers SPINN-PI-NT, the results should be similar for the full SPINN model: most of the computation involved in running SPINN is involved in populating the buffer, applying the composition function, and manipulating the buffer and the stack, with the low-dimensional tracking and parsing components adding only a small additional load.

4 NLI Experiments

We evaluate SPINN on the task of natural language inference (NLI, a.k.a. recognizing textual entailment, or RTE; Dagan et al., 2006). NLI is a sentence pair classification task, in which a model reads two sentences (a premise and a hypothesis), and outputs a judgment of *entailment*, *contradiction*, or *neutral*, reflecting the relationship between the meanings of the two sentences. Below is an example sentence pair and judgment from the SNLI corpus which we use in our experiments:

Premise: Girl in a red coat, blue head wrap and jeans is making a snow angel.

Hypothesis: A girl outside plays in the snow.

Label: entailment

SNLI is a corpus of 570k human-labeled pairs of scene descriptions like this one. We use the standard train–test split and ignore unlabeled examples, which leaves about 549k examples for training, 9,842 for development, and 9,824 for testing. SNLI labels are roughly balanced, with the most frequent label, *entailment*, making up 34.2% of the test set.

Although NLI is framed as a simple three-way classification task, it is nonetheless an effective way of evaluating the ability of a model to extract broadly informative representations of sentence meaning. In order for a model to perform reliably well on NLI, it must be able to represent and reason with the core phenomena of natural language semantics, including quantification, coreference, scope, and several types of ambiguity.

4.1 Applying SPINN to SNLI

Creating a sentence-pair classifier To classify an SNLI sentence pair, we run two copies of SPINN with shared parameters: one on the premise sentence and another on the hypothesis sentence. We then use their outputs (the $\vec{h}$ states at the top of each

[1] We chose to reimplement and evaluate only the feedforward/inference pass, as inference speed is the relevant performance metric for most practical applications.

[2] The original code for Irsoy & Cardie's model is available at `https://github.com/oir/deep-recursive`. Our optimized C++/CUDA models and the Theano source code for the full SPINN are available at `https://github.com/stanfordnlp/spinn`.

Param.	Range	Strategy	RNN	SP.-PI-NT	SP.-PI	SP.
Initial LR	2×10^{-4}–2×10^{-2}	LOG	5×10^{-3}	3×10^{-4}	7×10^{-3}	2×10^{-3}
L2 regularization λ	8×10^{-7}–3×10^{-5}	LOG	4×10^{-6}	3×10^{-6}	2×10^{-5}	3×10^{-5}
Transition cost α	0.5–4.0	LIN	—	—	—	3.9
Embedding transformation dropout	80–95%	LIN	—	83%	92%	86%
Classifier MLP dropout	80–95%	LIN	94%	94%	93%	94%
Tracking LSTM size D_{tracking}	24–128	LOG	—	—	61	79
Classifier MLP layers	1–3	LIN	2	2	2	1

Table 2: Hyperparameter ranges and values. *Range* shows the hyperparameter ranges explored during random search. *Strategy* indicates whether sampling from the range was uniform, or log-uniform. Dropout parameters are expressed as keep rates rather than drop rates.

stack at time $t = T$) to construct a feature vector $\vec{x}_{\text{classifier}}$ for the pair. This feature vector consists of the concatenation of these two sentence vectors, their difference, and their elementwise product (following Mou et al., 2016):

$$\vec{x}_{\text{classifier}} = \begin{bmatrix} \vec{h}_{\text{premise}} \\ \vec{h}_{\text{hypothesis}} \\ \vec{h}_{\text{premise}} - \vec{h}_{\text{hypothesis}} \\ \vec{h}_{\text{premise}} \odot \vec{h}_{\text{hypothesis}} \end{bmatrix} \quad (6)$$

This feature vector is then passed to a series of 1024D ReLU neural network layers (i.e., an MLP; the number of layers is tuned as a hyperparameter), then passed into a linear transformation, and then finally passed to a softmax layer, which yields a distribution over the three labels.

The objective function Our objective combines a cross-entropy objective $\mathcal{L}_{\text{s}}$ for the SNLI classification task, cross-entropy objectives $\mathcal{L}_{\text{p}}^t$ and $\mathcal{L}_{\text{h}}^t$ for the parsing decision for each of the two sentences at each step t, and an L2 regularization term on the trained parameters. The terms are weighted using the tuned hyperparameters α and λ:

$$\mathcal{L}_{\text{m}} = \mathcal{L}_{\text{s}} + \alpha \sum_{t=0}^{T-1} (\mathcal{L}_{\text{p}}^t + \mathcal{L}_{\text{h}}^t) + \lambda \|\theta\|_2^2 \quad (7)$$

Initialization, optimization, and tuning We initialize the model parameters using the nonparametric strategy of He et al. (2015), with the exception of the softmax classifier parameters, which we initialize using random uniform samples from $[-0.005, 0.005]$.

We use minibatch SGD with the RMSProp optimizer (Tieleman and Hinton, 2012) and a tuned starting learning rate that decays by a factor of 0.75 every 10k steps. We apply both dropout (Srivastava et al., 2014) and batch normalization (Ioffe and Szegedy, 2015) to the output of the word embedding projection layer and to the feature vectors that serve as the inputs and outputs to the MLP that precedes the final entailment classifier.

We train each model for 250k steps in each run, using a batch size of 32. We track each model's performance on the development set during training and save parameters when this performance reaches a new peak. We use early stopping, evaluating on the test set using the parameters that perform best on the development set.

We use random search to tune the hyperparameters of each model, setting the ranges for search for each hyperparameter heuristically (and validating the reasonableness of the ranges on the development set), and then launching eight copies of each experiment each with newly sampled hyperparameters from those ranges. Table 2 shows the hyperparameters used in the best run of each model.

4.2 Models evaluated

We evaluate four models. The four all use the sentence-pair classifier architecture described in Section 4.1, and differ only in the function computing the sentence encodings. First, a single-layer LSTM RNN (similar to that of Bowman et al., 2015a) serves as a baseline encoder. Next, the minimal SPINN-PI-NT model (equivalent to a TreeLSTM) introduces the SPINN model design. SPINN-PI adds the tracking LSTM to that design. Finally, the full SPINN adds the integrated parser.

We compare our models against several baselines, including the strongest published non-neural network-based result from Bowman et al. (2015a) and previous neural network models built around several types of sentence encoders.

Model	Params.	Trans. acc. (%)	Train acc. (%)	Test acc. (%)
Previous non-NN results				
Lexicalized classifier (Bowman et al., 2015a)	—	—	99.7	78.2
Previous sentence encoder-based NN results				
100D LSTM encoders (Bowman et al., 2015a)	221k	—	84.8	77.6
1024D pretrained GRU encoders (Vendrov et al., 2016)	15m	—	98.8	81.4
300D Tree-based CNN encoders (Mou et al., 2016)	3.5m	—	83.4	82.1
Our results				
300D LSTM RNN encoders	3.0m	—	83.9	80.6
300D SPINN-PI-NT (*parsed input, no tracking*) encoders	3.4m	—	84.4	80.9
300D SPINN-PI (*parsed input*) encoders	3.7m	—	89.2	**83.2**
300D SPINN (unparsed input) encoders	2.7m	92.4	87.2	82.6

Table 3: Results on SNLI 3-way inference classification. Params. is the approximate number of trained parameters (excluding word embeddings for all models). Trans. acc. is the model's accuracy in predicting parsing transitions at test time. Train and test are SNLI classification accuracy.

4.3 Results

Table 3 shows our results on SNLI. For the full SPINN, we also report a measure of agreement between this model's parses and the parses included with SNLI, calculated as classification accuracy over transitions averaged across timesteps.

We find that the bare SPINN-PI-NT model performs little better than the RNN baseline, but that SPINN-PI with the added tracking LSTM performs well. The success of SPINN-PI, which is the hybrid tree-sequence model, suggests that the tree- and sequence-based encoding methods are at least partially complementary, with the sequence model presumably providing useful local word disambiguation. The full SPINN model with its relatively weak internal parser performs slightly less well, but nonetheless robustly exceeds the performance of the RNN baseline.

Both SPINN-PI and the full SPINN significantly outperform all previous sentence-encoding models. Most notably, these models outperform the tree-based CNN of Mou et al. (2016), which also uses tree-structured composition for local feature extraction, but uses simpler pooling techniques to build sentence features in the interest of efficiency. Our results show that a model that uses tree-structured composition fully (SPINN) outperforms one which uses it only partially (tree-based CNN), which in turn outperforms one which does not use it at all (RNN).

The full SPINN performed moderately well at reproducing the Stanford Parser's parses of the SNLI data at a transition-by-transition level, with 92.4% accuracy at test time.[3] However, its transition prediction errors are fairly evenly distributed across sentences, and most sentences were assigned partially invalid transition sequences that either left a few words out of the final representation or incorporated a few padding tokens into the final representation.

4.4 Discussion

The use of tree structure improves the performance of sentence-encoding models for SNLI. We suspect that this improvement is largely due to the more efficient learning of accurate generalizations overall, and not to any particular few phenomena. However, some patterns are identifiable in the results.

While all four models under study have trouble with negation, the tree-structured SPINN models do quite substantially better on these pairs. This is likely due to the fact that parse trees make the scope of any instance of negation (the portion of the sentence's content that is negated) relatively easy to identify and separate from the rest of the sentence. For test set sentence pairs like the one below where negation (*not* or *n't*) does not appear in the premise but does appear in the hypothesis, the RNN shows 67% accuracy, while all three tree-structured models exceed 73%. Only the RNN got the below example wrong:

Premise: The rhythmic gymnast completes her floor exercise at the competition.

Hypothesis: The gymnast cannot finish her exercise.

Label: contradiction

Note that the presence of negation in the hypothesis is correlated with a label of *contradiction* in SNLI, but not as strongly as one might intuit—only 45% of these examples in the test set are labeled as

[3]Note that this is scoring the model against automatic parses, not a human-judged gold standard.

contradictions.

In addition, it seems that tree-structured models, and especially the tree-sequence hybrid models, are more effective than RNNs at extracting informative representations of long sentences. The RNN model falls off in test accuracy more quickly with increasing sentence length than SPINN-PI-NT, which in turn falls of substantially faster than the two hybrid models, repeating a pattern seen more dramatically on artificial data in Bowman et al. (2015b). On pairs with premises of 20 or more words, the RNN's 76.7% accuracy, while SPINN-PI reaches 80.2%. All three SPINN models labeled the following example correctly, while the RNN did not:

Premise: A man wearing glasses and a ragged costume is playing a Jaguar electric guitar and singing with the accompaniment of a drummer.

Hypothesis: A man with glasses and a disheveled outfit is playing a guitar and singing along with a drummer.

Label: entailment

We suspect that the hybrid nature of the full SPINN model is also responsible for its surprising ability to perform better than an RNN baseline even when its internal parser is relatively ineffective at producing correct full-sentence parses. It may act somewhat like the tree-based CNN, only with access to larger trees: using tree structure to build up local phrase meanings, and then using the tracking LSTM, at least in part, to combine those meanings.

Finally, as is likely inevitable for models evaluated on SNLI, all four models under study did several percent worse on test examples whose ground truth label is *neutral* than on examples of the other two classes. *Entailment–neutral* and *neutral–contradiction* confusions appear to be much harder to avoid than *entailment–contradiction* confusions, where relatively superficial cues might be more readily useful.

5 Conclusions and future work

We introduce a model architecture (SPINN-PI-NT) that is equivalent to a TreeLSTM, but an order of magnitude faster at test time. We expand that architecture into a tree-sequence hybrid model (SPINN-PI), and show that this yields significant gains on the SNLI entailment task. Finally, we show that it is possible to exploit the strengths of this model without the need for an external parser by integrating a fast parser into the model (as in the full SPINN), and that the lack of external parse information yields little loss in accuracy.

Because this paper aims to introduce a general purpose model for sentence encoding, we do not pursue the use of soft attention (Bahdanau et al., 2015; Rocktäschel et al., 2016), despite its demonstrated effectiveness on the SNLI task.[4] However, we expect that it should be possible to productively combine our model with soft attention to reach state-of-the-art performance.

Our tracking LSTM uses only simple, quick-to-compute features drawn from the head of the buffer and the head of the stack. It is plausible that giving the tracking LSTM access to more information from the buffer and stack at each step would allow it to better represent the context at each tree node, yielding both better parsing and better sentence encoding. One promising way to pursue this goal would be to encode the full contents of the stack and buffer at each time step following the method used by Dyer et al. (2015).

For a more ambitious goal, we expect that it should be possible to implement a variant of SPINN on top of a modified stack data structure with differentiable PUSH and POP operations (as in Grefenstette et al., 2015; Joulin and Mikolov, 2015). This would make it possible for the model to learn to parse using guidance from the semantic representation objective, which currently is blocked from influencing the key parsing parameters by our use of hard SHIFT/REDUCE decisions. This change would allow the model to learn to produce parses that are, in aggregate, better suited to supporting semantic interpretation than those supplied in the training data.

Acknowledgments

We acknowledge financial support from a Google Faculty Research Award, the Stanford Data Science Initiative, and the National Science Foundation under grant nos. BCS 1456077 and IIS 1514268. Some of the Tesla K40s used for this research were donated by the NVIDIA Corporation. We also thank Kelvin Guu, Noah Goodman, and many others in the Stanford NLP group for helpful comments.

[4] Attention-based models like Rocktäschel et al. (2016), Wang and Jiang (2016), and the unpublished Cheng et al. (2016) have shown accuracies as high as 86.3% on SNLI, but are more narrowly engineered to suit the task and do not yield sentence encodings.

References

Alfred V. Aho and Jeffrey D. Ullman. 1972. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *Proc. ICLR*. San Diego, CA.

Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled sampling for sequence prediction with recurrent neural networks. In *Proc. NIPS*. Montréal, Québec, pages 1171–1179.

Samuel R. Bowman, Gabor Angeli, Christopher Potts, and Christopher D. Manning. 2015a. A large annotated corpus for learning natural language inference. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Lisbon, Portugal, pages 632–642.

Samuel R. Bowman, Christopher D. Manning, and Christopher Potts. 2015b. Tree-structured composition in neural networks without tree-structured architectures. In *Proc. 2015 NIPS Workshop on Cognitive Computation: Integrating Neural and Symbolic Approaches*. Montréal, Québec, pages 50–55.

Jan Buys and Phil Blunsom. 2015. Generative incremental dependency parsing with neural networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*. Association for Computational Linguistics, Beijing, China, pages 863–869.

Danqi Chen and Christopher Manning. 2014. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, pages 740–750.

Jianpeng Cheng, Li Dong, and Mirella Lapata. 2016. Long short-term memory-networks for machine reading. arXiv:1601.06733.

Ido Dagan, Oren Glickman, and Bernardo Magnini. 2006. The PASCAL recognising textual entailment challenge. In *Machine learning challenges. Evaluating predictive uncertainty, visual object classification, and recognising tectual entailment*, Springer, pages 177–190.

David Dowty. 2007. Compositionality as an empirical problem. In *Proc. Brown University Conference on Direct Compositionality*. Oxford Univ. Press.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. 2015. Transition-based dependency parsing with stack long short-term memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Beijing, China, pages 334–343.

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A. Smith. 2016. Recurrent neural network grammars. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, San Diego, California, pages 199–209.

Ahmad Emami and Frederick Jelinek. 2005. A neural syntactic language model. *Machine learning* 60(1–3):195–227.

Christoph Goller and Andreas Küchler. 1996. Learning task-dependent distributed representations by backpropagation through structure. In *Proceedings of the IEEE International Conference on Neural Networks*. Washington, DC, pages 347–352.

Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. 2015. Learning to transduce with unbounded memory. In *Proc. NIPS*. Montréal, Québec, pages 1828–1836.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proceedings of the International Conference on Computer Vision*. Santiago, Chile, pages 1026–1034.

James Henderson. 2004. Discriminative training of a neural network statistical parser. In *Proceedings of the 42nd Meeting of the Association*

for Computational Linguistics (ACL'04), Main Volume. Barcelona, Spain, pages 95–102.

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

Gérard Huet. 1997. The zipper. *Journal of functional programming* 7(5):549–554.

Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the International Conference on Machine Learning*. Lille, France.

Ozan Irsoy and Claire Cardie. 2014. Deep recursive neural networks for compositionality in language. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, Curran Associates, Inc., pages 2096–2104.

Armand Joulin and Tomas Mikolov. 2015. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Proc. NIPS*. Montréal, Québec.

Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. 2014. A convolutional neural network for modelling sentences. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Baltimore, Maryland, pages 655–665.

Eliyahu Kiperwasser and Yoav Goldberg. 2016. Easy-first dependency parsing with hierarchical tree LSTMs. arXiv:1603.00375.

Jiwei Li, Thang Luong, Dan Jurafsky, and Eduard Hovy. 2015. When are tree structures necessary for deep learning of representations? In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Lisbon, Portugal, pages 2304–2314.

Lili Mou, Men Rui, Ge Li, Yan Xu, Lu Zhang, Rui Yan, and Zhi Jin. 2016. Natural language inference by tree-based convolution and heuristic matching. In *Proc. ACL*. Berlin, Germany.

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*. pages 149–160.

Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, pages 1532–1543.

Tim Rocktäschel, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, and Phil Blunsom. 2016. Reasoning about entailment with neural attention. In *Proceedings of the International Conference on Learning Representations*. San Juan, Puerto Rico.

Stuart M. Shieber. 1983. Sentence disambiguation by a shift-reduce parsing technique. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Cambridge, Massachusetts, USA, pages 113–118.

Richard Socher, Cliff Chiung-Yu Lin, Andrew Ng, and Chris Manning. 2011a. Parsing natural scenes and natural language with recursive neural networks. In Lise Getoor and Tobias Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. ACM, New York, NY, USA, ICML '11, pages 129–136.

Richard Socher, Jeffrey Pennington, Eric H. Huang, Andrew Y. Ng, and Christopher D. Manning. 2011b. Semi-supervised recursive autoencoders for predicting sentiment distributions. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Edinburgh, Scotland, UK., pages 151–161.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *JMLR* 15:1929–1958.

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

Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Vol-*

ume 1: Long Papers). Association for Computational Linguistics, Beijing, China, pages 1556–1566.

Theano Development Team. 2016. Theano: A Python framework for fast computation of mathematical expressions. arXiv:1605.02688.

Tijmen Tieleman and Geoffrey Hinton. 2012. Lecture 6.5 – RMSProp: Divide the gradient by a running average of its recent magnitude. In *Neural Networks for Machine Learning*, Coursera.

Ivan Titov and James Henderson. 2010. A latent variable model for generative dependency parsing. In Harry Bunt, Paola Merlo, and Joakim Nivre, editors, *Trends in Parsing Technology*, Springer, Netherlands, pages 35–55.

Ivan Vendrov, Ryan Kiros, Sanja Fidler, and Raquel Urtasun. 2016. Order-embeddings of images and language. In *Proc. ICLR*. San Juan, Puerto Rico.

Shuohang Wang and Jing Jiang. 2016. Learning natural language inference with lstm. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, San Diego, California, pages 1442–1451.

Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. In *Proc. NIPS*. Montréal, Québec, pages 649–657.

Xingxing Zhang, Liang Lu, and Mirella Lapata. 2016. Top-down tree long short-term memory networks. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, San Diego, California, pages 310–320.