Tree-structured neural networks revisited

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

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

Abstract

- TreeRNNs are great, don't work for big data.
- We present the transition-based TreeRNN implementation – makes efficient training possible, show that it helps relative to seq models.
- Further, attention has proven powerful for extending sequence models. We introduce constituent-by-constituent attention, and find it to be effective here, too.

TODO: Anonymize. **TODO:** Name the damn model!

1 Introduction

- Brief history of TreeRNNs
  - Lx theory suggests should be better than comparable sequence models via compositionality (Partee, 1984; Janssen, 1997).
  - In practice, results mixed but positive. Tai et al. 2015 shows dramatic gains on sentiment analysis using tree-structured models over sequence models. Li et al. 2015 shows mixed results. Both of these evaluations are limited to small datasets with only hundreds or thousands of training examples.
  - Larger evaluations have been impossible due to issues of computational efficiency. One aim of this paper is to mitigate these issues to make larger-scale experiments possible.
  - Note, though, that while tree structure can be seen as a bias, simply adding more data within reason may not be enough to get a sequence model to learn without that bias Bowman et al. 2015b.
- Speed issues
  - Efficient batched computation is a crucial enabling technology in allowing NNs to be used on large datasets. Since NNs require large datasets to be able to model complex problems like language understanding, this is crucial in NLP.
  - Tree structured neural networks can't be batched. As the schematic in Figure 1a indicates, different sentences are assigned different trees, and there is no straightforward way of aligning these trees to synchronize computation in the conventional style of implementation.
  - Our proposed model builds on ideas from transition-based parsing to build a model which implements the same computations as a tree-structured model, but which can be efficiently batched. See Figure 1b.
- Attention issues
  - The technique of soft attention has proven to be exceptionally effective as a way of training neural network models for tasks like translation (Bahdanau et al., 2015; Luong et al., 2015) and natural language inference (Rocktäschel et al., 2015).
  - This paper proposes two complementary extensions of the idea of attention to tree-structured models:
    * Constituent-by-constituent attention, in which each node in the tree of one sentence is given a soft alignment to some node or nodes in a second sentence.

*The first two authors contributed equally.

TODO: [SB] Figure showing why trees can't be batched (modeled after slides).

(a) Trees are hard to batch.

TODO: [SB] Figure showing a batch of sequences (modeled after slides).

(b) Sequences are easy to batch.

Figure 1: An illustration of a conventionally-implemented TreeRNN model (a) and our equivalent model (b).

    * The mTreeLSTM, which uses a third tree structured model to accumulate information about the alignment between two sentences into a single feature vector. This builds on the notion of a matching LSTM or mLSTM from Wang and Jiang 2015a.

- Outline of the paper
  - Propose a model
  - Show how the model can be used for NLI
  - Show how attention can be used on the NLI model
  - Experiments and discussion

2 The transition-based sentence model

2.1 Intuition: Transition-based parsing

Our model is based on a formalism adapted from transition-based parsing (**TODO:** What's a good classic citation for this?).

TODO: [SB, JG] Brief into to transition-based parsing.

2.2 The model

Our model, which is shown in Figure 2a, takes the form of a transition-based parser, but it is designed to produce sentence representations given parse structures, rather than producing parse structures as its output, leading to the following differences:

- The model takes a sequence of transitions (SHIFTs or REDUCEs) as input, rather than incorporating a decision function which chooses which transition to perform at each step.
- The intermediate representations in the parser, including the representations of the tree structures on the stack, are vectors of a fixed length.
- The REDUCE operation, which combines two trees or tree nodes on the stack into a single larger tree, is represented using a parametric neural network layer function (in particular, a TreeLSTM layer, after Tai et al. 2015).
- The intermediate states of the stack and buffer during parsing are stored during training. This makes it possible to learn the parameters of the composition function and the input word representations that are fed into the buffer using backpropagation on an objective function that depends on the final sentence representation that emerges at the head of the stack.
- A novel tracking LSTM component is added to improve the model's ability to efficiently

track the context in which each step is being performed.

Equivalence with existing models If we add no additional features, this model computes the same function as a plain TreeRNN. However, we expect it to be substantially faster than conventional TreeRNN implementations. Unlike a TreeRNN, the Model 0 computation graph is essentially static across examples, so examples of varying structures and lengths can be batched together and run on the same graph in a single step. This simply requires ensuring that the graph is run for enough timesteps to finish all of the sentences. This involves some wasted computation, since the composition function will be run $2N-1$ times (with the output of composition at non-REDUCE steps discarded), rather than $N-1$ times in a TreeRNN. However, this loss can be dramatically offset by the gains of batching, which stem from the ability to exploit highly optimized CPU and GPU libraries for batched matrix multiplication. **TODO:** Update.

The buffer and word representations We draw our word representations from the standard 300-dimensional vector package provided with GloVe (Pennington et al., 2014). We do not update these vectors during training. Instead, we use a learned linear transformation to transform the representation of each input word $\vec{x}$ into a pair of vectors that can be used as inputs to a TreeLSTM composition function:

$$(1) \quad \begin{bmatrix} \vec{h} \\ \vec{c} \end{bmatrix} = W_{\text{wd}} \vec{x}$$

The representations in the stack In our model, each row of the stack contains a pair of vectors (h, c), which together represent some node in the parse tree of the sentence. Since the SHIFT operation simply copies entries from the buffer onto the stack, this means that the word representations in the buffer also have two parts, h and c.

The composition function When a REDUCE operation is performed, vector representations of two tree nodes are popped off of the head of the stack and fed into a *c*omposition 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 then pushed on to the stack.

Our composition function is based closely on the TreeLSTM of Tai et al. 2015. The TreeLSTM generalizes the LSTM neural network layer (Hochreiter and Schmidhuber, 1997) to tree- rather than sequence-based inputs, and it shares with that older design the idea of representing intermediate states in a computation using a two-part vector representation containing an $\vec{h}$ vector and a $\vec{c}$ vector, where the latter is meant to serve as a slower-changing long-term memory.

The precise formulation that we use is as follows.

$$(2) \quad \begin{bmatrix} \vec{i} \\ \vec{f_l} \\ \vec{f_r} \\ \vec{o} \end{bmatrix} = \sigma(W_{\text{iffo}} \begin{bmatrix} \vec{h}_s^1 \\ \vec{h}_s^2 \\ \vec{e} \end{bmatrix})$$

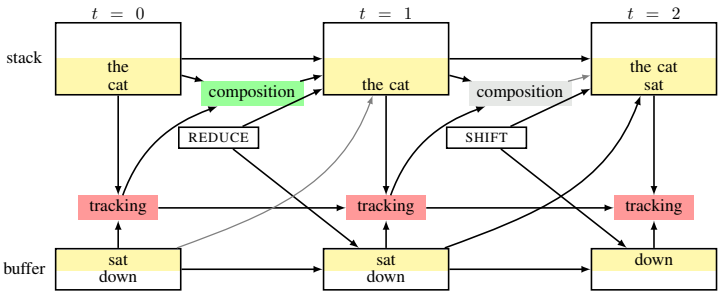
$$(3) \quad \vec{g} = \tanh(W_g \begin{bmatrix} \vec{h}_s^1 \\ \vec{h}_s^2 \\ \vec{e} \end{bmatrix})$$

$$(4) \quad \vec{c} = \vec{f_l} * \vec{c}_s^2 + \vec{f_r} * \vec{c}_s^1 + \vec{i} * \vec{g}$$

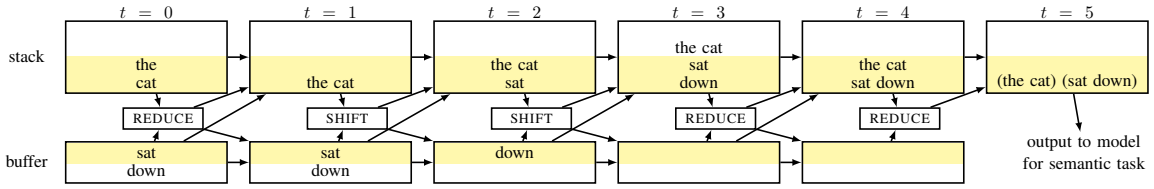
$$(5) \quad \vec{h} = \vec{o} * \vec{c}$$

The results of this function are the pair $(\vec{h}, \vec{c})$, which are placed back on the stack. The two input tree nodes popped off the stack are represented as the pairs $(\vec{h}_s^1, \vec{c}_s^1)$ and $(\vec{h}_s^2, \vec{c}_s^2)$. In addition, $\vec{e}$ is an optional input argument which is either the empty vector [] or a vector from an external source like the tracking LSTM (see below). $*$ denotes the elementwise product. Each vector-valued variable listed here is of dimension D, except $\vec{e}$, which is of dimensions D_t, which can be chosen independently.

Creating a sentence pair classifier This paper presents the transition-based sentence model within the context of *natural language inference* (also known as *recognizing textual entailment*), a sentence pair classification task. To classify a sentence pair, a feature vector is first constructed. This feature vector is based on the final representations of each of the two sentences—the representations that appear at the head of the stack for each sentence after the final transition. In particular, the $\vec{h}$ portions of these representations are used. The final feature vector consists of the concatenation of these two vectors, $\vec{x}_{\text{premise}}$ and $\vec{x}_{\text{hypothesis}}$, their



(a) The model unrolled for two transitions on the input *the cat sat down*.



(b) The fully unrolled model for *the cat sat down* with some layers omitted for clarity.

Figure 2: Two views of the transition-based sentence model. In both views, the lower boxes represent the input buffer, and the upper boxes represent the stack. Yellow highlighting indicates which portions of these data structures are visible to the tracking LSTM and to the composition function. Thin gray arrows indicate connections which are blocked by a gating function, and so contribute no information. **TODO:** [SB] Clean up the arrangements of these figures now that we aren't reporting on Models 1/2.

difference, and their elementwise product:

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

This feature vector is then passed to a series of two (**TODO:** Update) ReLU neural network layers, then passed into a linear transformation, and then finally passed to a softmax layer, which yields a distribution among the three labels.

2.3 Tracking left context with an auxiliary LSTM

The tracking LSTM captures left context. **TODO:** Why?

The tracking LSTM is a standard LSTM RNN. At each step it takes three vectors as inputs: the top element of the buffer $\vec{h}_b^1$, which would be moved in a SHIFT operation, and the two two elements of the stack $\vec{h}_s^1$ and $\vec{h}_s^2$ which would be composed in a REDUCE operation. Its output hidden state at each step is used as the external input to the TreeLSTM composition function for that step.

2.4 Implementing the transition-based sentence model

The size of the stack The size of the stack should be N for sentences of N words, in case the first reduce merges the final two words. The size of the buffer should be N.

Converting parses to transition sequences For Models 0–3, all training data must be parsed in advance into an unlabeled binary constituency tree. In addition, Model 0 requires that parses be available at test time as well. For both SST and SNLI we use the parses included with the corpus distributions whenever parses are needed.

For model 0, training data can be prepared by linearizing the provided parse, then deleting left brackets and replacing right brackets with REDUCE instructions. That is demonstrated here with the example sentence *the cat sat down*:

((the cat) (sat down))⇒
the cat REDUCE sat down REDUCE REDUCE

The input for models 1–4 is simply the word sequence from the parse, with the first two words moved into the stack. The syntactic supervision labels for models 1–3 are simply a binarized version of the Model 0 inputs, with the first two tokens (which are necessarily SHIFT SHIFT) omitted:

((the cat) (sat down))⇒
stack: ⟨the, cat⟩
buffer: ⟨sat, down⟩
ops: REDUCE SHIFT SHIFT REDUCE REDUCE

Handling variable sentence lengths To efficiently implement batched computation, we must stipulate a fixed number of transitions (50) that the model can perform before producing an output. Sentences whose transition sequences are shorter than this length are padded, and sentences whose transition sequences are longer than this length are cropped.

Padding, if done properly, should not impact the output of the model significantly:[1] a model with the fixed parameters should produce the same representation for a 10-word sentence whether it is unrolled for 19 transitions (the minimum to avoid cropping) or 100.

Cropping necessarily discards information, but if done properly, it is nonetheless possible to learn good representations for cropped sentences. When a transition sequence is cropped, the number of removed SHIFT transitions is tracked, and an equal number of word tokens is removed from the tail (left) end of the buffer. This makes it possible for a transition sequence to begin with a REDUCE transition, or to otherwise use a REDUCE transition on a stack that does not have two elements to reduce. In this instance, we simply feed the composition function one or more zero vectors, which represent nonexistent stack elements. If the composition learns to interpret these zero vectors properly, they can be taken to be unknown or incomplete nodes in an otherwise complete tree that is constructed according to the intended parse structure. **TODO:** [SB] Make a figure showing the results of cropping if there's room.

Representing the stack in memory Or, thin stack. **TODO:** [JG] Explain.

Optimization and hyperparameters We use the RMSProp optimizer (**TODO:** Cite.) with 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 (Sergey Ioffe, 2015) to the word embeddings in the buffer (after the linear projection is applied) and to the feature vectors that serve as the inputs and outputs to the MLP that precedes the final entailment classifier. In addition, we apply an L2 regularization penalty to all parameters.

We used random search to tune many of the hyperparameters, including the learning rate, the dropout rate, the L2 regularization weight, the hidden dimension of the tracking LSTM and the number of layers in the MLP **TODO:** [SB] Report the ranges searched for each parameter and the number of runs for each results.

We trained each model for 250k steps in each run, using a minibatch size of 64 for each step. We tracked each model's performance on the development set during training and saving parameters when this performance reached a new peak. We used early stopping, evaluating on the test set using the parameters that performed best on the development set.

Software infrastructure We will make Theano code available.

2.5 TreeRNN-equivalence and efficiency

In its bare form, without the addition of the tracking LSTM, our model computes the precisely the same functions as a conventional tree-structured neural network model, and has the same learning dynamics: the representation of each sentence consists of the representation of the words, combined recursively using a TreeRNN composition function (in our case, the TreeLSTM function). While it adds no expressive power on its own, our model design improves upon the standard TreeRNN implementation in three ways:

- It provides a straightforward strategy for implementing tree-structured composition which allows for batched computation, and provides near-optimal run times on typical computer systems. While our design includes some wasted computation induced by the full application of the composition function at every step, this is made up for by avoiding most of the (potentially expensive) scattered memory access that would be needed for batched computation strategies that more

[1] The tracking LSTM can cause a slight dependence of the final representation on unrolling length. Because of this, we always train and test the model with a fixed unrolling length.

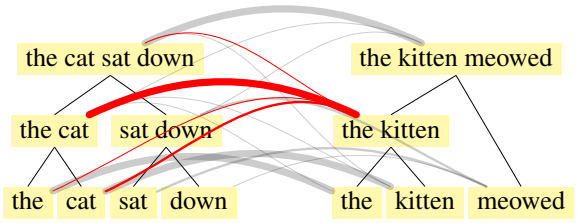


Figure 3: Soft alignments between the nodes of two trees, with alignments to "the kitten" highlighted in red.

closely mimic the standard tree-based architecture.

- It provides a simple mechanism for integrating the use of left-context into tree-structured models through the tracking LSTM, providing a fast mechanism to help these models handle lexical ambiguity.

- By designing the model around the standard transition set used in neural network syntactic parsers, it opens the door to tight integration between parsing and sentence interpretation, including the possibility of joint learning, or even of learning a parser trained entirely on a sentence classification or interpretation objective.

2.6 Related work: Transition-based parsing and neural networks.

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

3 Attention over trees

Constituent-by-constituent attention We aim to build on the results of Rocktäschel et al. 2015 and Wang and Jiang 2015b, who find that neural soft attention models is an extremely effective technique for learning natural language inference. In particular, both papers use versions of word-by-word entailment, in which a latent alignment is learned from every word in the hypothesis to one or more words of the premise. We propose to borrow this basic idea, but to adapt it to a tree-structured setting, proposing *constituent-by-constituent* attention. While these models do attention over a matrix $\mathbf{Y}$ of word-in-context representations from the premise encoder, we will perform attention instead over our own primary data structure, $\mathbf{Y}^{st}$, the matrix of vectors that have appeared at the top of the stack during premise encoding, which correspond one-to-one to the constituents in the tree structure representing the premise. Similarly, while the previous models perform one instance of soft attention conditioned on each word in the hypothesis, we perform one instance of soft attention conditioned on each stack top in the hypothesis encoder, representing the constituents of the hypothesis tree.

In our model, attention is performed at each step t of the premise encoder. At step t, the query vector that drives attention will be S_0^t, the top of the stack.

TODO: [AR, RG] Write up the attention equations that you use.

3.1 Accumulating the results of attention

Accumulating results sequentially (After Rocktschel, Wang and Jiang)

Accumulating results using a second tree layer

4 Experiments

4.1 Natural language inference and SNLI

We evaluate our model on the task of natural language inference (NLI), also known as recognizing textual entailment (RTE). 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, as here:

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

Hypothesis: *A girl outside plays in the snow.*

Correct label: *entailment*

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

In particular, we use the Stanford NLI corpus (SNLI, Bowman et al. 2015a). SNLI is a corpus of 570k human-labeled pairs of scene descriptions like the one above. 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.

4.2 Models evaluated

We evaluate five models on the SNLI task. Each uses 150d hidden states:

- A baseline LSTM model (similar to that of Bowman et al. 2015a) that uses the same classifier architecture as our models, but encodes sentences using a single layer LSTM RNN sequence model.
- The unaugmented transition-based sentence model.
- The transition-based sentence model with the tracking LSTM.
- The transition-based sentence model with the tracking LSTM and sequence-based attention over premise states.
- The transition-based sentence model with the tracking LSTM and tree-based attention over premise states.

Results Table 1 shows our results.

4.3 Step time

TODO: [JG,SB] Does anyone know of a better baseline TreeRNN implementation that we can use on Jagupard? We can use the CoreNLP SST model, but using a Java model as a baseline seems worrying unless we're guaranteed that it's competitively fast.

Figure 4: The alignment induced between two sentences by **TODO:** some model.. Compare with a similar alignment presented by Wang and Jiang 2015a.

Comparing model step time to the plain RNN of Li et al. 2015. We use the small Pang and Lee 2004 sentiment corpus that they use, and train with identical hyperparameters: ...

5 Discussion

Patterns in the examples that the tree model got right that the LSTM didn't

Induced alignments

6 Conclusions and future work

- It is possible to train a tree-structured model, without approximation, on large-scale language learning tasks.
- Tree structure helps in large-scale language learning tasks. Not massively, but measurably.
- The advantages of attention carry over to tree-structured models.

Future work The tracking LSTM that we incorporated into our model 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 of each tree node, including information about that nodes neighbors in the sentence string and its likely position in the final tree, and that that context information could support better sentence encoding. One

Model	Trained params.	Train	Test
Previous non-NN results			
Lexicalized classifier (Bowman et al., 2015a)	–	99.7	78.2
Previous NN results without attention			
100d LSTM encoders (Bowman et al., 2015a)	221k	84.8	77.6
1024d pretrained GRU encoders (Vendrov et al., 2015)	15m	98.8	81.4
300d Tree-based CNN encoders (Mou et al., 2015)	3.5m	83.4	82.1
Previous NN results with attention			
100d LSTM w/ word-by-word attention (Rocktäschel et al., 2015)	252k	85.3	83.5
300d mLSTM word-by-word attention model (Wang and Jiang, 2015a)	1.9m	92.0	86.1
300d LSTMN with deep attention fusion (Cheng et al., 2016) (1.4m params) 92.3 89.1	1.4m	92.3	89.1
New results			
RESULT NEEDED: 150d LSTM RNN	?	?	?
RESULT NEEDED: 150d TB TreeLSTM	?	?	?
RESULT NEEDED: 150d TB TreeLSTM w/ tracking	?	?	?
RESULT NEEDED: 150d TB TreeLSTM w/ tracking, sequence-based attn.	?	?	?
RESULT NEEDED: 150d TB TreeLSTM w/ tracking, tree-based attn.	?	?	**?**

Table 1: Results on SNLI 3-way inference classification. All reported figures are percent accuracy. **RESULT NEEDED:** [JG]Add seconds-per-step *if* we get a reasonably well-optimized model in time.

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 parsing. The model presented here is only capable of operating over sentences which have already been assigned a binary constituency tree by a parser. This is not necessarily a practical limitation, since modern parsers can assign reasonably high quality parses in so little time as to not dramatically impact the performance characteristics of our model **TODO:** [SB,JG,CM] How fast are fast constit parsers?. However, depending upon an external parser can be a missed opportunity.

In future work, we aim to incorporate a parser into our model, allowing it to simultaneously learn to parse sentences (using a conventional parsing objective function) and to use those parses to learn semantic representations for those sentences (using a classification objective function, as in this work). Implementing such a model on top of our proposed stack-based design would learn a model that could productively share representations between the two tasks. Implementing such a model on top of a differentiable version of our stack-based design (possibly following Grefenstette et al. 2015 **TODO:** [SB] Cite FAIR stack paper.) would further make it possible for the model to learn to parse using guidance from the semantic representation objective, essentially allowing learn to produce parses that are, in aggregate, better suited to supporting semantic interpretation than those supplied in the training data.

Acknowledgments

Some of the Tesla K40(s) used for this research was/were donated by the NVIDIA Corporation. **TODO:** [CM,CP] Acknowledge other grants.

References

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

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 (EMNLP)*. Association for Computational Linguistics.

Samuel R. Bowman, Christopher D. Manning, and Christopher Potts. 2015b. Tree-structured composition in neural networks without tree-structured architectures. In *Proceedings of the 2015 NIPS Workshop on Cognitive Computation: Integrating Neural and Symbolic Approaches*.

Jan Buys and Phil Blunsom. Generative incremental dependency parsing with neural networks. *Volume 2: Short Papers*, page 863.

Danqi Chen and Christopher D Manning. 2014. A fast and accurate dependency parser using neural networks. In *Empirical Methods in Natural Language Processing (EMNLP)*.

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

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)*, pages 334–343, Beijing, China, July. Association for Computational Linguistics.

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A. Smith. 2016. Recurrent neural network grammars. arXiv manuscript arXiv:1602.07776.

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

Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. 2015. Learning to transduce with unbounded memory. *arXiv preprint arXiv:1506.02516*.

James Henderson. 2004. Discriminative training of a neural network statistical parser. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, page 95. Association for Computational Linguistics.

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

T.M.V. Janssen. 1997. Compositionality. In van Benthem and ter Meulen (van Benthem and ter Meulen, 1997).

Minh-Thang Luong Li, Jiwei, Dan Jurafsky, and Eudard Hovy. 2015. When are tree structures necessary for deep learning of representations? *Proc. EMNLP*.

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal, September. Association for Computational Linguistics.

Lili Mou, Men Rui, Ge Li, Yan Xu, Lu Zhang, Rui Yan, and Zhi Jin. 2015. Recognizing entailment and contradiction by tree-based convolution. *arXiv preprint arXiv:1512.08422*.

Bo Pang and Lillian Lee. 2004. A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. In *Proc. ACL*, page 271. Association for Computational Linguistics.

B.H. Partee. 1984. Compositionality. In Fred Landman and Frank Veltman, editors, *Varieties of Formal Semantics*. Foris.

Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. GloVe: Global vectors for word representation. In *Proc. EMNLP*.

Tim Rocktäschel, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, and Phil Blunsom. 2015. Reasoning about entailment with neural attention. *arXiv preprint arXiv:1509.06664*.

Christian Szegedy Sergey Ioffe. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift.

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

Kai Sheng Tai, Richard Socher, and Christopher D Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*.

Ivan Titov and James Henderson. 2010. A latent variable model for generative dependency parsing. In *Trends in Parsing Technology*, pages 35–55. Springer.

J. van Benthem and A. ter Meulen, editors. 1997. *Handbook of Logic and Language*. MIT Press and North-Holland.

Ivan Vendrov, Ryan Kiros, Sanja Fidler, and Raquel Urtasun. 2015. Order-embeddings of images and language. *CoRR*, abs/1511.06361.

Shuohang Wang and Jing Jiang. 2015a. Learning natural language inference with LSTM. *CoRR*, abs/1512.08849.

Shuohang Wang and Jing Jiang. 2015b. Learning natural language inference with lstm. *arXiv preprint arXiv:1512.08849*.

Xingxing Zhang, Liang Lu, and Mirella Lapata. 2016. Top-down tree long short-term memory networks. *arXiv preprint arXiv:1511.00060*.

TODO: [SB] Make bibliography style uniform.