Sequence-to-Sequence Dependency Parsing

Felix Sun and Michael Chang

12/11/15

1 Abstract

We develop and test dependency parsers that use an LSTM neural network to directly predict head-tail relationships from input words. This represents a simpler approach to parsing than shift-reduce parsing with a neural classifier, which is the state of the art in dependency parsing. We demonstrate a first-order neural network parser, as well as a method of introducing higher-order dependencies into the LSTM parsing framework.

2 Background and Prior Work

As discussed in class, dependency parsing is a way to create a hierarchy (a tree) from the words in a sentence. In this hierarchy, if a word A has a child B, then B modifies A grammatically. For example, in "a fat dog", "a" and "fat" are children of "dog". A dependency parse can be thought of as a flattening of a full constituency parse, in which each word is also labeled with a part of speech. [1]

The state-of-the-art approach to dependency parsing is using a shift-reduce parser. In a shift-reduce parser, words are copied one at a time into a stack. A machine learning classifier is used to determine what operation to perform on each word as it appears in the stack: the top two words in the stack may be connected with a parent-child relationship, or a new word may be introduced onto the stack. A complete tree is built by repeatedly using the classifier to make local decisions. This class of algorithms can produce any parse tree that is projective, meaning the tree's branches do not overlap if the tree is drawn on top of the sentence.

Shift-reduce parsers differ in the machine learning algorithm that is used to choose the operation at each step. In more traditional parsers like Maltparser [2], a linear classifier like a SVM is used to control the parser. The SVM takes hand-engineered sparse input features, built from the top words on the stack, the state of the already-parsed words, and other state variables of the parser. The classifier is walked through the steps of parsing training data, and is trained to make the correct decision at each step.

More recently, neural nets have been applied to shift-reduce parsing. In [3], the stack and parse state of the parser are transformed via word vectors into a dense representation. This dense representation is then fed through a 2-layer neural network, whose output determines the parser's next action. The neural net requires fewer features to make a good decision, a fact that makes this parser faster than Maltparser.

Currently, the best-performing dependency parser is the RBGParser, described in [4]. This parser uses a learned linear combination of a variety of hand-engineered features to score a candidate parse tree. A greedy hill-climbing algorithm is used to explore the space of candidate trees during testing.

We investigate a simpler approach to dependency parsing that does away with the shift-reduce framework. Since the goal of a dependency parser is to output arcs that go from heads to dependents, learning a parse tree of a sentence reduces to finding the index of the head word for each word in the sentence. Inspired by the neural translation models by [6] and [7], our parser uses LSTMs (long-short term memory neural networks) to process the entire sentence directly, in essence translating the sequence of input words to a sequence of heads. LSTMs (see [5]

for a review) are a type of recurrent neural network, which means they share weights across a time dimension. An LSTM is composed of one unit for each timestep of the input data; in this case, each word of the input sentence would correspond to a timestep. Each unit has the same weights, and each unit receives an input from the unit before it (in addition to the regular input) and emits an output to the unit after it (in addition to the regular output). The result is a neural network that is constrained to learn sequence patterns. We are interested in using bidirectional LSTMs, which consist of two LSTM layers stacked on top of each other; one propagating state in the forward direction, and one in the reverse direction.

3 Technical Approach

3.1 Learning Problem

Given sentence $\mathbf{x} = [x_1, \dots x_n]$ and its corresponding sequence of heads $\mathbf{y} = [y_1, \dots y_n]$, we formulate the learning problem of the dependency parser to computing the estimates $\hat{\mathbf{y}}$ as

$$\arg\min_{\hat{\mathbf{y}}} D(\hat{\mathbf{y}}, \mathbf{y}|\mathbf{x}) \tag{1}$$

where D is the categorical cross-entropy distance function.

In the rest of this paper, our notation will be as follows. A sentence is represented by $x_{1:n}$, where x_i is the i-th word. y_i represents the index of the head of the i-th word. The root of a sentence is represented by a y-value of $n+1$. For example, in the sentence "I love chocolate", $x_{1:n}$ = [I, love, chocolate], and $y_{1:n} = [2, 4, 2]$. Inputs to all models described below are word vectors. Our neural parser models the conditional log probability of the heads $\mathbf{y}$ given the input sentence $\mathbf{x}$ as:

$$\log p(\mathbf{y}|\mathbf{x}) = \sum_{i=1}^{n} \log p(y_i|y_{1:i-1}, \mathbf{x}) \tag{2}$$

The output of the neural parser for each word x_i is a distribution over the possible y_j's. We use various decoding methods to generate predictions y_j from these distributions. In all, the neural parser is a function f that maps $\hat{y}_i = f(\hat{y}_{1:i-1}, x_{1:n})$ - the parser labels the head of one word at a time, taking into account the decisions it made for previous words.

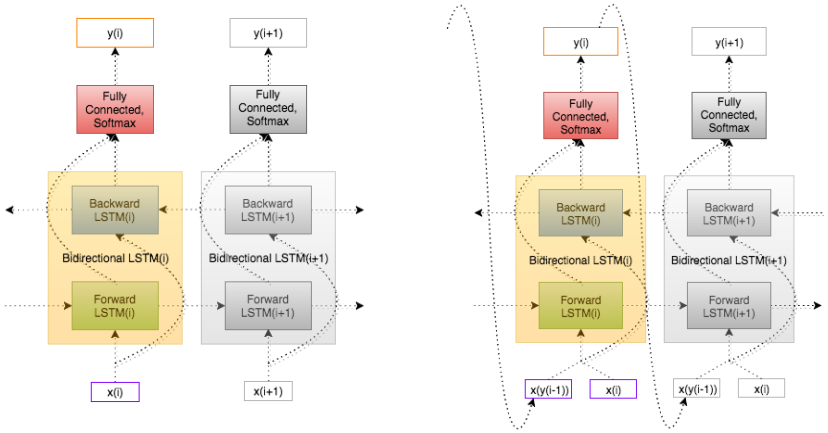
In the sections below, we present various models that we've tried for solving this problem. We start with the simplest model - the bidirectional LSTM - as our baseline to implement a first order parser and then a full parser. We then experiment with the Indexer Model, which uses a different feedback mechanism to implement the full parser. Both the baseline and the Indexer predict the heads y_i via a softmax distribution over the length of longest sentence in the corpus. However, the problem with predicting the index of the heads directly with this softmax is that that would cause positional bias - for example, given phrases "the big dog" and "the dog", the head of "the" ("dog"), plays the same role in both sentences, yet the model would predict different indexes for the two sentences. To make our model completely agnostic to sentence length, we began developing an attention-based model similar to that of the neural translation models developed in class. We leave further development of this attention-based model for future work.

3.2 Baseline Bidirectional LSTM

3.2.1 Model 1 - First-Order Parser

Our baseline model implements a first-order parser with one bidirectional LSTM that directly learns the mapping from $\mathbf{x}$ to their respective heads $\mathbf{y}$. The LSTM outputs are fed into a fully connected softmax layer, whose output is a probability distribution over the head of the current word, y_i. It models the conditional log probability of the heads $\mathbf{y}$ as

$$\log p(\mathbf{y}|\mathbf{x}) = \sum_{i=1}^{n} \log p(y_i|\mathbf{x})$$



(a) First Order Parser

(b) Full Parser

Figure 1: The first order parser and full parser models. Both models use a bidirectional LSTM to transform the input word vectors x_i into head indices y_i. The Full parser also takes $x_{y_{i-1}}$, the previous word's head word, as an input for each new word. This gives the model information about previous decisions it has made.

The model is given in Figure 1a. The model can use all of $\mathbf{x}$ as context to make its prediction because the bidirectional LSTM's internal state vectors incorporate information going forward and backward along the sentence. Since our baseline model is a first order parser, it does not use its previous decisions to inform its current decision. In other words, y_i is solely a function of $\mathbf{x}$, and not of $y_{1:i-1}$.

3.2.2 Model 2 - Full Parser using Word Vector Feedback

It is straightforward to extend the baseline model's modeling capacity by leveraging its past predictions as well (Figure 1b). By feeding its prediction from the previous timestep y_{i-1} as input to the current timestep in addition to x_i, rather than outputting the distribution $\log p(y_j|\mathbf{x})$ at each timestep, it outputs $\log p(y_j|y_{1:j-1}, \mathbf{x})$. With the bidirectional lstm's hidden state, by feeding only the prediction from the previous timestep in as input, the model recursively has access to all of its predictions from word 1 to word i.

3.3 Model 3 - Indexing Model

The indexing model represents a slightly different way to input the previously-decided head y_{i-1} into a neural model. Instead of feeding in the word at the head index, we feed in the LSTM activation at the head index. This is implemented as follows: First, the word vectors $\mathbf{x}$ are fed through a bidirectional LSTM, to obtain activation vectors $\mathbf{h}$. Then, for each word in order, the activation of the previous word's head, $h(y_{i-1})$ is selected. $h(y_{i-1})$ is transformed by another LSTM, and then fed along with $h(i)$ into a fully connected layer, followed by a softmax over possible head positions for y_i. This network is shown in Figure 2.

This feedback scheme is designed to provide higher-level features to the network. In the baseline model, the network has to learn to interpret the previous head words. In this model, we preprocess each head word, using the same transformation that the network learned for preprocessing the sentence words. This will hopefully result in head words that are easier for the network to use.

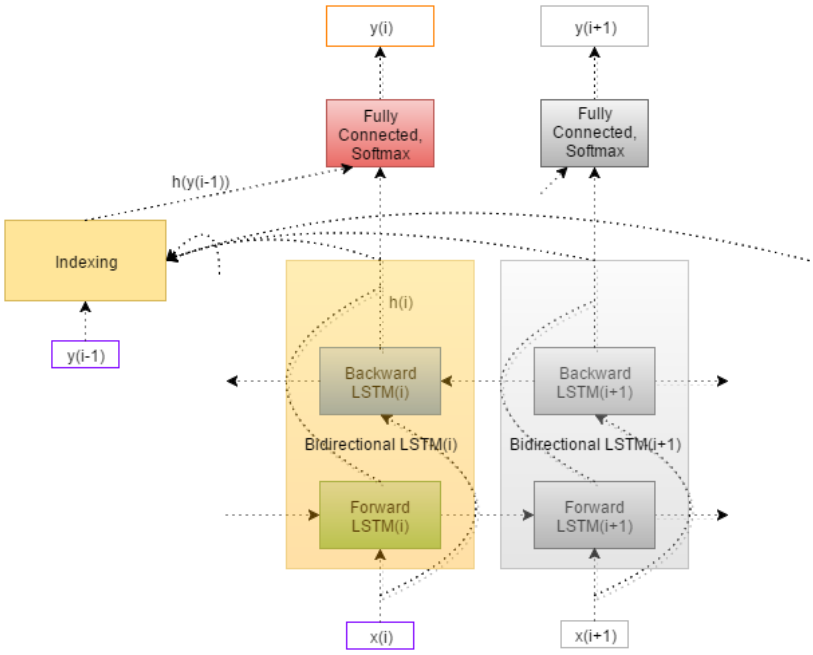


Figure 2: The indexing parser model. The network's decision on the previous word is fed back into the network, by indexing into the LSTM activations. For example, if word 2's head was word 5, the indexing layer will choose the LSTM activation for word 5 when the network is trying to label word 3.

4 Training Details

For all our models, we map words to 256-dimensional word vectors (trained jointly with the LSTM), and use LSTM memory and hidden states of dimension 512. We train on batches that contain sentences of all the same length, with a randomized order of lengths within an epoch. We experimented with a curriculum learning strategy, in which the batches were ordered by increasing sentence length and each batch was applied multiple times in succession, but this did not appear to help with learning.

During decoding, we used beam search to deal with the dependency of y_i on the previous y's. Beam search was performed as follows: A list of candidate head indices $y_{1:i-1}$ are kept. For each candidate, the network is run in feedforward mode to generate a probability distribution over y_i. Out of all the values of $y_{1:i}$ generated by adding all y_i possibilities to each candidate, the *beamwidth* most probable are kept. The process is repeated until the entire sentence has been parsed, and the most likely finished candidate is chosen as the overall parse. A beam width of 20 was used throughout.

We discovered a few small training improvements that each increased performance significantly. These are described below.

4.1 Relative Indexing

Although our models aim to predict the absolute index y_i of the head of the word x_i, it is inefficient to train using the absolute index directly. This is because the absolute index of a word's head is dependent on context from far-away words. Consider two sentences that include the same phrase, "my hat". In the first sentence, "my hat" appears in the beginning of the sentence, and in the second sentence, "my hat" appears in the end of the sentence. Although "my" is the head of "hat" in both sentences, using absolute indexing would train the model to predict a low index for the head of "hat" in the first sentence, and a high index for the head of "hat" in the second sentence. While it is possible for an LSTM to learn this rule, it appears to be rather difficult.

To avoid this problem, we make a slight change to how we predict y_i. Instead of predicting y_i directly, we predict

Parser	Bulgarian	Turkish
Best reported in literature	0.940	0.776
LSTM, first order (M1)	0.802	0.708
LSTM with word vector feedback (M2)	0.791	0.708
Indexing - one layer (M3)	0.521	0.535
Indexing - two layers(M3)	0.735	0.683

Figure 3: Parser accuracies on the Bulgarian and Turkish corpora in the CONLL-X dataset. All accuracies are computed as unlabeled attachment scores. The two layer indexing model has a two-layer LSTM that generates **h**.

$y_i^{(rel)}$, the relative position of the head with respect to x_i. For example, in the phrase "my hat", the $y^{(rel)}$ of "my" is always +1, because its head is one word to the right of it. We assign the ROOT label a value of 0. We found that the baseline model is more accurate by 10 percentage points when trained uisng relative, as opposed to absolute, y indexing.

4.2 Dealing with Out-of-Vocabulary Words

To create a word vector for each word, we first convert each word in the training set into a numerical label. During testing, words not encountered in the training set are given the number 0, corresponding to OOV (out of vocabulary). We found that this approach tended to overfit on the training data. The problem is, the network never sees the OOV token during training, and therefore is never taught what to do with OOV tokens. During testing, it is unable to handle new words, even when the word's context alone is enough to inform a good parse.

Instead, we replace all infrequent words with the OOV tag during training. Words that occur fewer than 3 times in the entire training set are represented by OOV, which allows the network to learn the meaning of this tag. In the Bulgarian dataset, this resulted in the replacement 29k out of 190k words in the training set, and 1300 out of 5900 words in the test set. (This is a count of the total number of words in all the sentences, not a count of the number of unique words in the dictionary.) Of the 1300 OOV words in the test set, 730 of them were not seen in the training set at all; the remainder were seen either once or twice. This modification increased the accuracy of the baseline parser by 4 percentage points, and the indexing parser by over 10 percentage points.

5 Evaluation

We evaluate our parsers on the CONLL-X dataset [8], a widely-used multilingual dataset for evaluating parsing models. We focused on the Turkish and Bulgarian corpora in this dataset, because these two corpora have the lowest and highest accuracies, respectively, in [4]. The results are shown in Figure 3.

We find that adding word vector feedback to the first order parser doesn't seem to affect performance significantly. On the other hand, adding indexing feedback to the first order parser deceases performance significantly, although the effect can be somewhat ameliorated by adding a second layer to the LSTM. (We also experimented with a 2-layer version of the baseline model, which had the same performance as the single-layer baseline model.)

We wish to diagnose why adding feedback to our baseline model does not improve performance. The most obvious suspect is decoding. In a first order parser, decoding is trivial, because the probability of each y_i is independent of the others:

$$\max_{\mathbf{y}} p(\mathbf{y}|\mathbf{x}) = \max_{\mathbf{y}} \prod_i p(y_i|\mathbf{x}) = \prod_i \max_{y_i} p(y_i|\mathbf{x})$$

However, in parsers with feedback, each y_i depends on all previous y_i's, and no efficient exact algorithm exists to find the best $\mathbf{y}$. Beam search only approximates the optimal solution. To measure the degree to which this approximation is affecting performance, we feed back the ground truth y_{i-1} into the network at each word, instead of the network's prediction of y_{i-1}. This gives us an upper bound on how well the network can parse, assuming a perfect decoder algorithm. As shown in Figure 4, we find that the word vector feedback parser (model 2) is already decoding optimally, while the indexing parser (model 3)is not.

Parser	Outputted y_{i-1}	Ground truth y_{i-1}
LSTM, first order (M1)	0.802	0.802
LSTM with word vector feedback (M2)	0.791	0.808
Indexing - one layer (M3)	0.521	0.653
Indexing - two layers (M3)	0.735	0.776

Figure 4: Comparison of parser accuracies when the ground truth y_{i-1} is fed back. The first data column (outputted y_{i-1}) shows accuracies when beam search is used to feed the model's prediction at word $i-1$ back into the model at word i. The second data column shows accuracies when the ground truth y_{i-1} is used instead. The larger the disparity between the two, the less efficient the beam search procedure is.

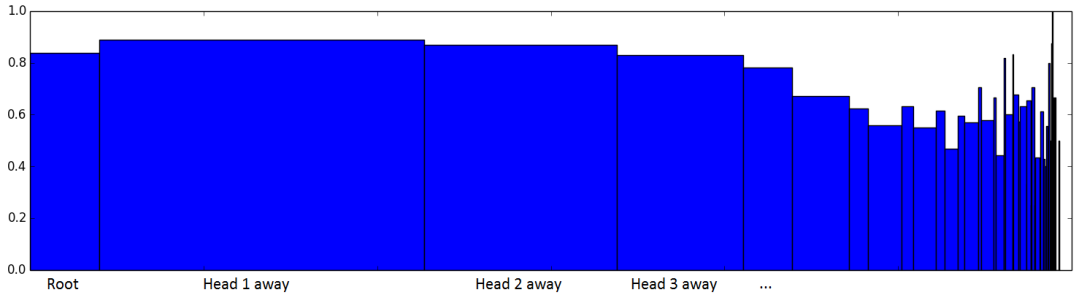


Figure 5: A bar chart of average accuracy of words (y-axis), sorted by the distance between the word and its head. Data from the word vector parser (M2) on Bulgarian. The width of each bar indicates the number of words in that category in the Bulgarian test dataset. The total accuracy of the parser is therefore equal to the overall fraction of blue in the plot. The parser learns to correctly classify the common (i.e. closest) head distances, but performs less well on longer head distances.

The word vector feedback parser has the same performance as the first-order parser, and does not benefit from a better decoding scheme. This suggests that the word vector feedback parser is not effectively integrating the previous y_i's into its decision.

Finally, we analyze the error rate by type of y, in Figure 5. The further the head is from the current word, the more likely it is for our parser to make an error. This is not surprising, given that distant heads are less common in the training set. This reflects an inherent weakness of positional bias that all three of our models exhibit: they have to learn separate rules for outputting each possible value of y, including large values, for which there is very little training data. In the next section, we demonstrate preliminary results for a parser that is agnostic to sentence length and therefore does not have this weakness.

6 Proposals for More Advanced Models

In this section, we describe two models that we implemented, but did not fully test due to time constraints. The vector LSTM model uses a vector dot product to score how well word j fits as the head of word i, for all i and j in a sentence. The attention model is an application of sequence-to-sequence learning to the parsing problem. We present preliminary results for the vector LSTM model.

6.1 Vector Scoring LSTM Model

Our analysis suggests that the performance of our parsers is hindered by a lack of training data on words with distant heads. This training data is needed because our models output a probability distribution over the head distance, so they must separately learn a rule for each head distance. To avoid this problem, we propose *vector scoring* models, in which head probabilities are represented as a dot product of a word vector and a context vector. In general, we

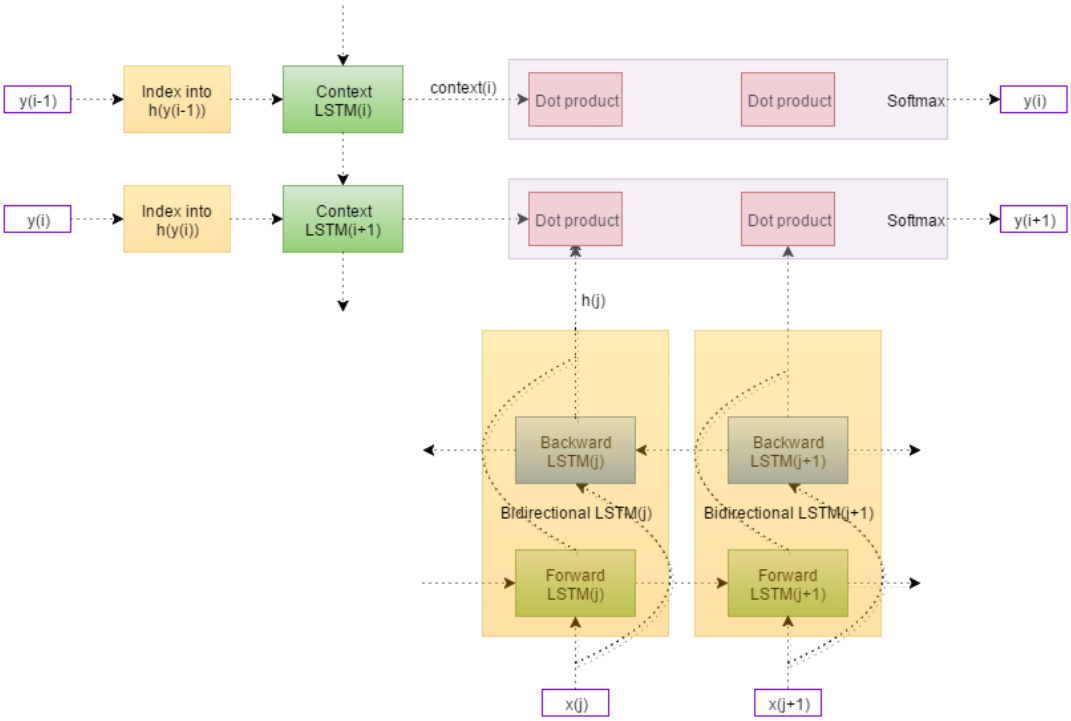


Figure 6: Vector dot product scoring model. Decoding happens in the vertically-oriented LSTM in this figure – after the h's are generated, computation happens from the top down.

express the probability that the word at index j is the head of the word at index i as:

$$p(j|i) = \frac{\exp(h_j \cdot context_i)}{\sum_{j'} \exp(h_{j'} \cdot context_i)} \tag{3}$$

where h_j is a neural representation of word j (like the LSTM activations in the previous models), and $context_i$ is a neural representation of the word i and the decisions made for the heads of previous words.

By formulating the head decision this way, we avoid the problem of head distance entirely. We also avoid the problem of learning absolute head position that was inherent to the absolute indexing scheme. Our decision at each word is not a number, but another word. We use a dot product to compute the "fit" between each context vector, representing the word to be parsed, and each h vector, representing the candidate head. This approach was inspired by [9], which uses the same vector scoring idea to match image boxes to words in a sentence.

The vector scoring LSTM model is a simple extension of the feedback LSTM models, that uses a vector dot product instead of a softmax layer to make the final y decision. The model is shown in Figure 6. In the figure, the h_j vectors are first generated from the input words. Then, for each word index i in the sentence, the h vector for the previous word's head – $h_{y_{i-1}}$ – is fed into a context LSTM. We take the dot product between the output of the context LSTM and each h_j, and interpret the resulting vector, which has the same length as the input sentence, as a softmax probability distribution over the head of word i.

There is one problem in training this model: there is no way for the model to set a word as the root, because the root is not one of the words in the sentence. To fix this, we add an end token representing the root to each sentence.

We found that this model achieves an accuracy of 0.687 on the Bulgarian corpus, compare to 0.521 for the indexing model with the same number of parameters. If a second layer is added to the initial LSTM, the accuracy increases to 0.720, compared to 0.735 for the indexing model. Based on these limited experiments, it appears that vector indexing allows our LSTM model to more efficiently use its parameters to model the training data. However, we did not have time to optimize the model, or to experiment with other languages.

We can also investigate whether the vector scoring model improves performance on words with distant heads. Figure 7 shows a small improvement for words with distant heads, when overall accuracy is normalized.

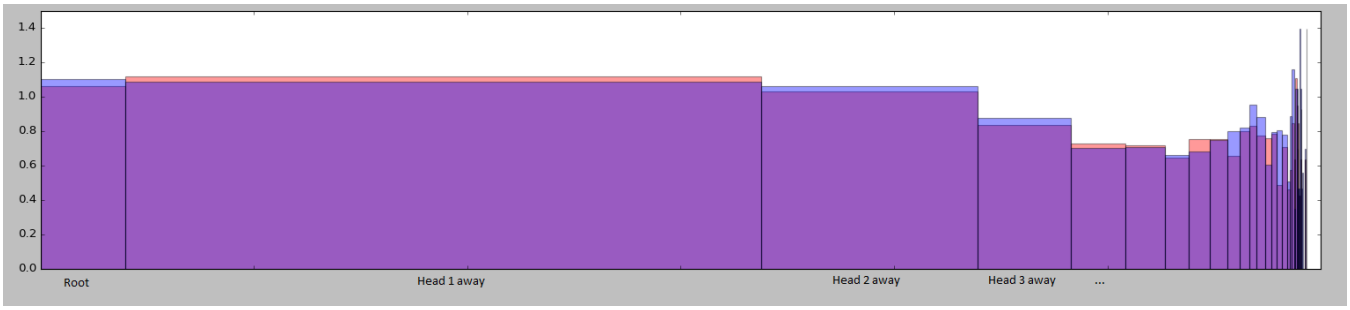


Figure 7: A bar graph of word error rate for different categories of words. As in Figure 5, each bar represents all the words of a certain distance from their heads, and the width represents the number of such words. The y-axis shows the accuracy, as a proportion of the total accuracy on the entire dataset. (This allows us to compare the baseline and vector parsers, which have different overall accuracies.) The baseline parser is shown in red, and the vector LSTM model is shown in blue. Purple shows where they overlap. The vector LSTM model's relative strength is on words with more distant heads, as expected.

6.2 Attention Model

Both the baseline and indexing model predict the index of the head y_i from a softmax distribution over the maximum sentence length. However, as discussed before this approach causes unwanted positional bias. We then began developing a new model that makes predictions independently of sentence length or index. We present below an attention-based model (Figure 8) that chooses which index in the sentence to attend to when predicting the head y_j. This attention model is composed from an encoder and decoder. The encoder maps the input sentence to a hidden activation vector $\mathbf{h}$. While the encoder is bidirectional, the decoder runs only in the forward direction. At each timestep, the decoder predicts a head y_i based on the h and its own evolving hidden activation g.

The model first encodes the input sentence $\mathbf{x}$ with a bidirectional LSTM, producing $\mathbf{h}$, the hidden activation that is the concatenation of the hidden activations from the forward and backward LSTMs.

When predicting head y_i, the decoder first computes a summary vector f_i over the length of the sentence. This vector f_i is the same length as the sentence. Each element f_{ij} summarizes the input sentence at index j as a weighted combination of the concatenation $s_{ij}(h_j, g_{i-1}, \hat{y_{i-1}})$ of the hidden activation h_j, the previous decoder activation g_{i-1}, and the previous prediction y_{i-1} as

$$f_{ij}(h_j, g_{i-1}, \hat{y_{i-1}}) = W^T s_{ij}(h_j, g_{i-1}, \hat{y_{i-1}})$$

With this summary vector f_i, we can represent the probability $p(y_i = j)$ as

$$\alpha_{ij} := \frac{e^{f_{ij}}}{\sum_{j'=1}^{n} e^{f_{ij'}}}$$

Therefore, each α_i is the distribution $p(y_j | y_{1:j-1}, \mathbf{x})$, analogous to the final softmax layer of the baseline model. The key difference is that the softmax distribution constructed in this attention model is independent of the length of the sentence.

The decoder activations are computed with an LSTM. At timestep i, this LSTM takes as input a weighted average of $c_i = \sum_{j=1}^{n} \alpha_{ij} h_j$ of all the h_j's, weighted by the probabilities α_{ij}. These weights α_{ij} are the attention mechanism of this model: because c_i is fed into the decoder LSTM as input, the decoder LSTM attends to a different parts of the sentence with varying degree at timestep i. The output of this decoder LSTM at timestep i is g_i, which gets concatenated with h_{i+1} and y_i to form s_{i+1} at the next timestep.

With this construction, the attention model constructs its distribution over the length of the sentence, without needing to know *a priori* what the length of the sentence is. It is therefore sentence length-agnostic and what we believe the model that is most expressive in modeling the dependency parsing problem accurately. Currently our results for the attention model lie around 20% per-arc accuracy, which indicates that further work needs to be done on the implementation end.

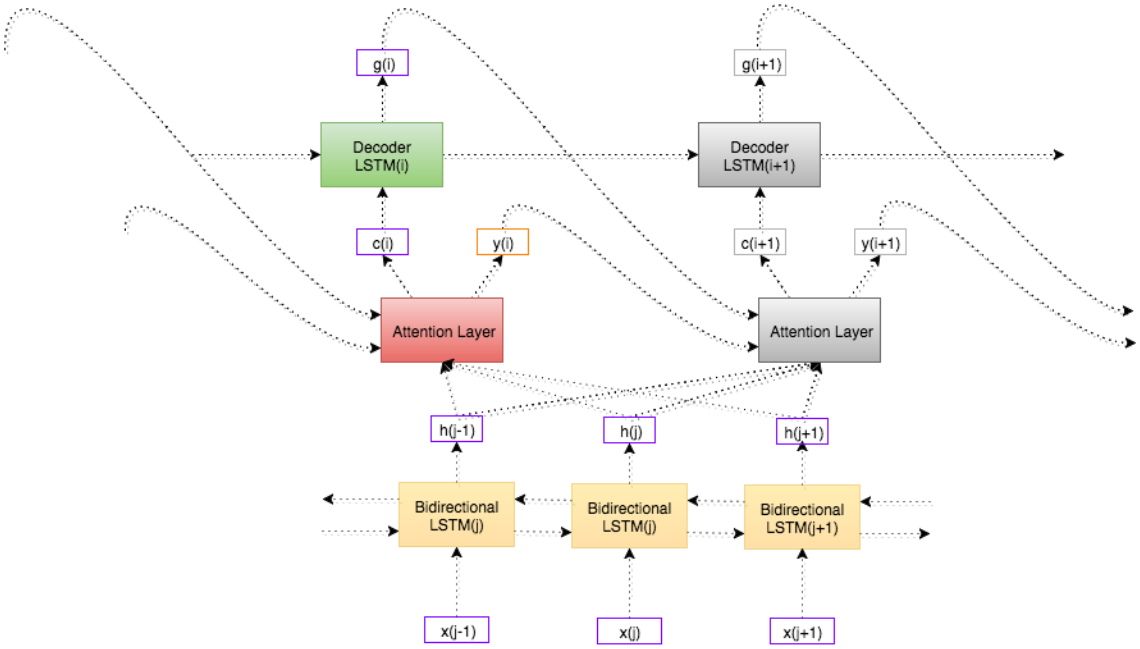


Figure 8: Attention model. The input sentence is encoded once into the hidden activation **h**, which will be used for all decodings. The decoding LSTM steps word by word through the sentence, producing prediction y_i at timestep i. To produce this prediction y_i, the attention layer takes a softmax of a function over all h_j's, the previous decoder LSTM output g_{i-1}, and the previous prediction y_i.

7 Conclusions

In this project, we have shown that to LSTMs can solve the parsing problem, by mapping a sequence of words to a sequence of head indices. We presented several different architectures for LSTM parsing. Though our parsers fall short of state of the art performance, they show that LSTMs can learn the fundamental patterns required for parsing.

Our code is available at: https://github.mit.edu/mbchang/rnn-parsing

References

[1] Covington, M. A. (2001). "A fundamental algorithm for dependency parsing." *In Proceedings of the 39th annual ACM southeast conference* (pp. 95-102).

[2] Nivre, J., Hall, J., and Nilsson, J. (2006). "Maltparser: A data-driven parser-generator for dependency parsing." *In Proceedings of LREC.*

[3] Chen, D., and Manning, C. D. (2014). "A fast and accurate dependency parser using neural networks." In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (Vol. 1, pp. 740-750).

[4] Zhang, Yuan, et al. (2014) "Greed is good if randomized: New inference for dependency parsing." In *Empirical Methods in Natural Language Processing (EMNLP).*

[5] Lipton, Z. C. (2015). "A critical review of recurrent neural networks for sequence learning." *arXiv:1506.00019.*

[6] Luong, Minh-Thang, Hieu Pham, and Christopher D. Manning. "Effective Approaches to Attention-based Neural Machine Translation." *arXiv preprint arXiv:1508.04025* (2015).

[7] Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate." arXiv preprint arXiv:1409.0473 (2014).

[8] Buchholz, S., Marsi, E. (2006, June). CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of the Tenth Conference on Computational Natural Language Learning* (pp. 149-164). Association for Computational Linguistics.

[9] Karpathy, A., Fei-Fei, L. (2014). Deep visual-semantic alignments for generating image descriptions. arXiv preprint arXiv:1412.2306.