# Learning c-command with language models: a syntactic approach

**Avi Cooper**
avi.cooper@yale.edu

**Tal Boger**
tal.boger@yale.edu

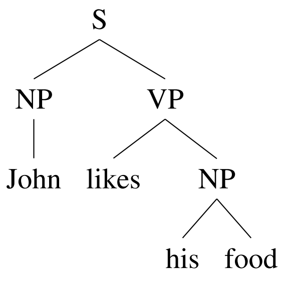
**Daniel Fridman**
daniel.fridman@yale.edu

Code available at: https://github.com/avicooper1/LING-380-Final-Project-Spring-2020/

## 1 Introduction

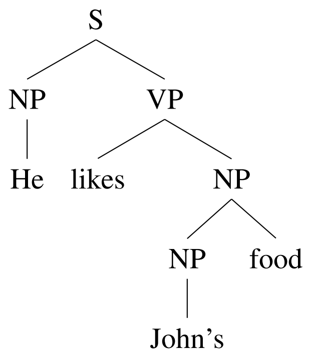
C-command, or constituent command, is a linguistic feature which refers to the relationship between two tokens. First introduced by Tanya Reinhart (1976), c-command is defined as follows: "a node A c-commands a node B iff the first branching node $\alpha$ that dominates A either dominates B, or is immediately dominated by a node $\alpha'$ which dominates B, where $\alpha$ and $\alpha'$ are of the same category type" (Reinhart 1983). We can express this relationship through syntax trees. In syntax trees, node A dominates node B if node A is above node B and we can trace a downward path from A to B. Therefore, an alternative definition for c-command through syntax trees is that A c-commands B if a parent of A dominates B. So, in Tree 1 below, "John" c-commands "his."

(Tree 1)



Interestingly, c-command can be found in the grammar of all human languages, suggesting that it may be a feature of the Universal Grammar. The function of c-command in all human languages appears to be to determine agreement between two tokens. An English language example of this can be demonstrated by the coreference between pronouns and proper nouns. In this case, if the pronoun c-commands the proper noun, then it is clear that the sentence refers to two individuals. For example, consider Tree 2.

(Tree 2)



In the sentence generated by Tree 2, "he" c-commands "John." Therefore, it is clear that "he" and "John" are two distinct individuals. However, in the sentence generated by Tree 1, the reference becomes ambiguous. Here, "John" c-commands "his", making it unclear whether "his" refers to John or a second individual. Similar examples of c-command determining the agreement between two tokens exist in other parts of speech, such as subject-verb agreement.

We attempt to implement neural language models to study whether certain architectures can learn the hierarchical c-command structure from natural language. We will compare how different architectures perform at this task. After training our models on a large corpus of natural language, we will test their performance on specific sentences containing c-command.

Through the implementation of several network architectures, we seek to find a model capable of effectively learning c-command. By doing so, we hope to provide more insight into the underlying structure of c-command and expand our understanding of the role of c-command in the Universal Grammar and human linguistic cognition.

## 2 Methods

To test whether neural network models can implicitly learn the abstract rule of c-command, we first trained models as if they were going to be used for language modeling (next word prediction). Then, we evaluated their performance on specific data sets designed to include ambiguous examples of c-command. If the models performed well on these tests, it would mean that they learned some implicit representation in the natural language training.

We used the following models for the task described above: a simple recurrent network (SRN), a long short-term memory network (LSTM), a gated recurrent unit network (GRU), and finally a stack-augmented parser-interpreter neural network (SPINN). SPINN (Bowman et. al. 2016) "combines parsing and interpretation within a single tree-sequence hybrid model." So, SPINN allows us to represent syntactic structure along with the natural language. Therefore, the SRN, LSTM, and GRU serve as baseline indicators of performance, given that they only process the natural language and not snytactic structure.

For each model, we implemented a simple architecture. Each batch was fed through an embedding layer, a recurrent layer, and a decoding layer.

### 2.1 Embeddings

Rather than training our own embeddings, we used pre-trained embeddings from GloVe (Global Vectors for Word Representation; Pennington et. al. 2015). Specifically, we used the Wikipedia 2014 + Gigaword 5 set (as it was the smallest) which was trained on 6 billion tokens, had a total vocabulary size of 400,000 and was uncased. To reduce training time and use of computational resources, we used the 50 dimension embedding vectors from this vector set.

### 2.2 Recurrent layer

Here, we used the standard PyTorch provided SRN, GRU, and LSTM layers for each respective network. Given that these models represent a baseline level of performance, we made no changes to the number of layers, dropout, etc. For the SPINN model, we first passed the embedded input through a linear layer which doubled the size of the embedding. This is necessary for the syntax operations performed inside the SPINN model. We then passed the new embeddings to the SPINN model. The recurrent layer in each of our four models returned the same size output.

### 2.3 Output layer

After the recurrent layer, we used a standard PyTorch linear layer to map our recurrent output to the total number of words in our data set. We also applied a sigmoid to this output.

## 3 SPINN

In the other network architectures discussed, the model learns an understanding of state over time – it uses hidden encodings of the tokens from previous time steps to influence its decision for the prediction in the current time step. These are helpful in language modeling, as it allows for an understanding of context. However, when trying to learn c-command, the tree structure representation necessary to explicitly state the rule is absent. These networks might have some implicit understanding of the tree structure, and this might help on the c-command evaluation tasks. However, it is likely that if given an explicit representation of the tree structure the network would perform better on these evaluations.

For this reason, we used a SPINN model. This allowed us to input both the tokens of the context, as well as a binary parse representation of the context's tree structure. In a linear pass through the tokens and the binary parse, the network applies its recurrence in such a way that the network develops a hidden understanding of the tree structure as well.

In implementing this model in PyTorch, we used the notes and adapted the GitHub code from an NVIDIA guide on SPINN models (Bradbury 2017). We modified this implementation to be able to use the same training functions as our normal language models.

## 4 Training procedure

We trained each of our models on the Stanford Natural Language Inference (SNLI) data set available in PyTorch. Though SNLI is rarely used for language modeling (instead, it is commonly used for an entailment task), we chose this data set because it includes parsed sentences. This allowed us to feed a large amount of natural language to each model, while also giving us tree structure information to give to the SPINN model.

We trained the models for a typical next-word prediction task. Due to computational limitations, we trained the SRN, GRU, and LSTM on a 1/10 subset of SNLI, and the SPINN on a 1/5 subset.

We trained each model with an embedding size of 50 and a hidden size of 50. Each model could train for a maximum of 50 epochs, though we used early stopping for validation loss with a patience

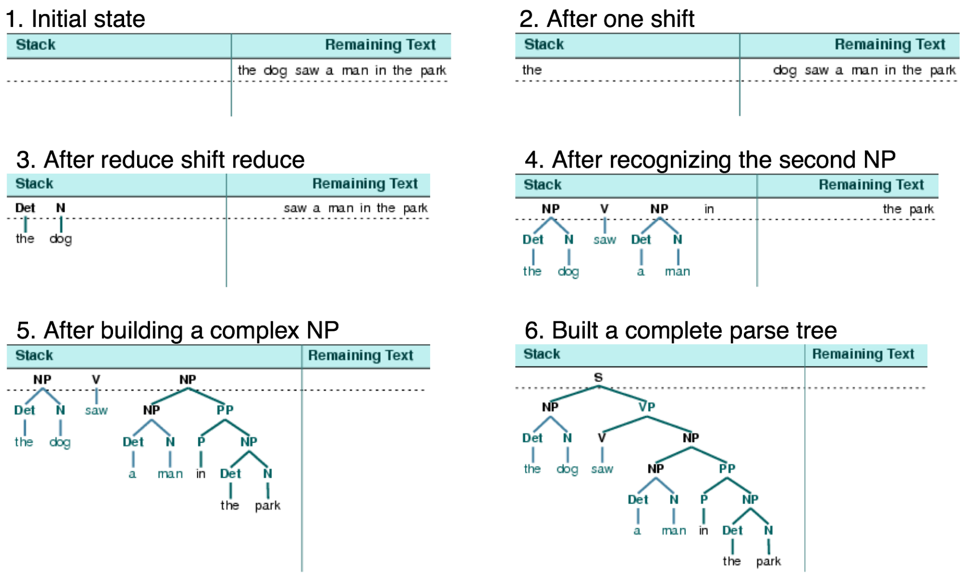


Figure 1: Example of shift-reduce parse (source: NLTK)

of 3 epochs. Table 1 on the following page shows each model's training information. Note that loss is calculated for each word, so models that processed less data will have a smaller loss, because they make fewer predictions.

## 5 Testing procedure

To test our networks' ability to learn c-command, we evaluated their performance on a data set of paired c-command examples. We used the principle A c-command data set from the Benchmark of Linguistic Minimal Pairs (BLiMP) (Wrastadt et. al. 2019), a data set designed to test a language model's ability to learn grammatical phenomena in English. Principle A is a specific type of binding where an anaphor must be bound in its governing category.

The BLiMP data set has 1,000 pairs of sentences that either use c-command correctly or misuse it. The data set provides an example of the correct sentence and an example of the incorrect sentence. In this specific data set, the difference between the correct and incorrect sentence always came at the final word of the sentence, where a pronoun either obeyed c-command or contradicted c-command.

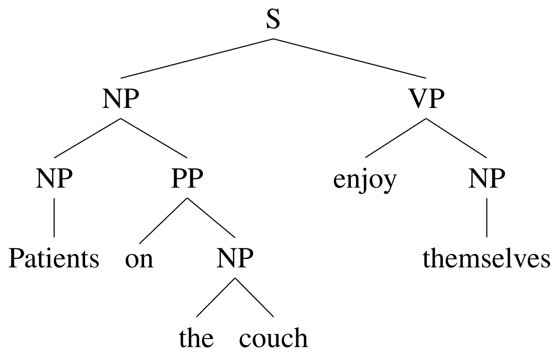
For each of the non-SPINN models, we tokenized the c-command sentence pairs and extracted their context, or the sentence up until the point where the correct and incorrect sentences differ. Then, we fed the sentences to the models and examined the model's final prediction (i.e. the next word given the context).

For the SPINN model, along with the tokenized sentences, we created a shift-reduce parse of each sentence using Stanford Parser (de Marneffe et. al. 2006). A shift-reduce parser processes a sentence by shifting or reducing each word based on its position in the parse tree. See Figure 1 above for an example. We parsed each sentence in the BLiMP data set and tokenized the parse into the same format as our training data. Then, the SPINN received both the context and the parse to make its prediction for the next word.

To evaluate model performance, we compared the network probability for the correct word to the probability for the incorrect word. Each time the network assigned a higher probability to the word that correctly used c-command, the network made a correct prediction. We repeated this process for all 1,000 sentence pairs in the data set to evaluate network performance.

Below is an example of converting a sentence in BLiMP into an acceptable parsed input for the SPINN model.

First, we parse the sentence:



| Model | Train loss | Val. loss | Test loss | Epochs |
|---|---|---|---|---|
| SRN | 58.065[†] | 144.064 | 143.317 | 20 |
| GRU | 58.132[†] | 144.018 | 143.247 | 30 |
| LSTM | 58.170[†] | 144.015 | 143.250 | 32 |
| SPINN | 29.267* | 144.565 | 143.812 | 8 |

Table 1: Model training results.
†: short train conducted with 1/5 of the data. ∗: 1/10 of the data.

Then, we binarize the parse, meaning we group together words from the same node. We mark the start of each node with an open parentheses, and the end with a close parentheses. So, a binarized version of the above parse might look like:

( ( ( patients ) ( on ( the couch ) ) ) ( enjoy ( themselves ) ) )

Each open parentheses and token represents a shift, while each closed parentheses represents a reduce, as the closed parentheses signifies the completion of a node. So, we convert the above to a list of shifts and reduces, then feed it to the model.

## 6 Results

Though next word prediction implies a random performance of (1 / n_vocab), we tested our models on whether they assigned the right word a higher probability than the wrong word. Therefore, the outcome is binary, so random performance is 50% accuracy. Figure 2 shows our raw performance.

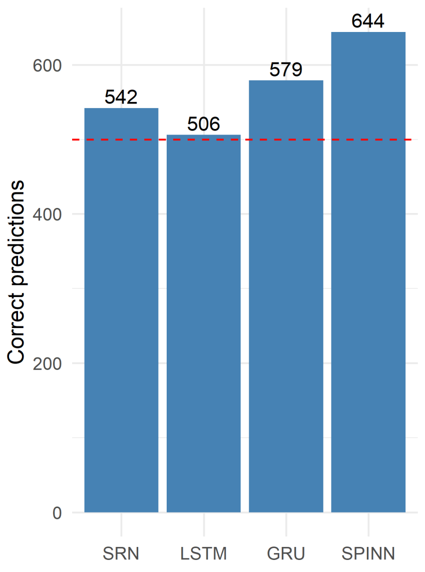


Figure 2: Model accuracy.
Red line indicates random performance (500 correct)

Figure 3 shows each model's improvement over random.

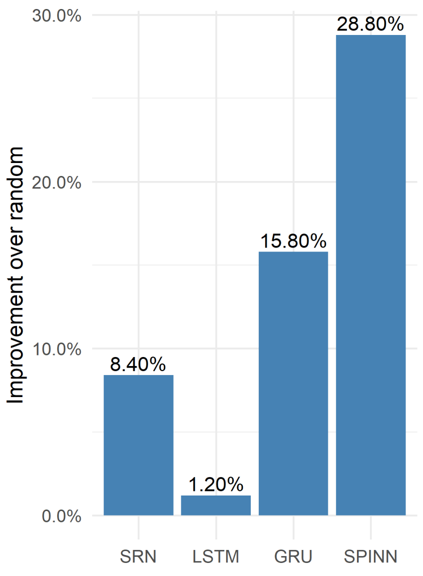


Figure 3: Model improvement over random.

Along with these results, the creators of BLiMP provided results they achieved with their own trials and networks. Table 2 below shows their results.

| Model | One-prefix accuracy | Sentence accuracy |
|---|---|---|
| LSTM | 58.4% | 58.8% |
| GPT-2 | 100% | 100% |
| TXL[†] | 61.7% | 60.8% |
| N-Gram | 56.8% | 57.8% |
| Humans | 86.0% | 86.0% |

Table 2: BLiMP given performance.
†: TXL = Transformer-XL.

Our baseline models produced comparable performance to the BLiMP LSTM and N-Gram models.

Both our non-SPINN models and BLiMP's LSTM and N-Gram models performed marginally better than random. This indicates that learning c-command strictly from a word-level language model is difficult. However, the SPINN model vastly outperforms our other models, as it has an almost 25% improvement over random. Furthermore, it outperforms the TXL model (Dai et. al. 2019). This is notable, given that TXL is much more complex, is intended for natural language, and also is trained on a much larger corpus.

## 7 Discussion of results & future work

Our results suggest that models with knowledge of syntactic structure vastly outperform simple word-level neural language models in learning c-command. This can likely be extended to other phenomena in English grammar that are explained directly by syntactic structure, such as other forms of subject-verb agreement.

Though GPT-2 achieved perfect accuracy on the c-command data set, this is expected given its complexity. As we discussed earlier, we did not train our models a typical natural language data set. Given more time, we would use a data set such as WikiText-103 (which TXL is trained on) instead of SNLI to achieve even better performance. However, because the sentence parsing is computationally expensive, parsing WikiText-103 would not be possible in the given time frame. We initially started with WikiText-103, but instead had to use SNLI because of its already-parsed data.

To achieve better results in future work, one might attempt to stack a SPINN-like model on top of a large transformer model such as BERT or GPT-2. Though these large transformer models seem to implicitly learn syntactic structure (as seen by their performance on the BLiMP test) encoding additional syntactic information may make them even more accurate.

## References

[1] Reinhart, T. “Coreference and bound anaphora: A restatement of the anaphora questions.” *Linguistics and Philosophy* vol. 6, 1983, pp. 47–88. https://doi.org/10.1007/BF00868090

[2] Bowman, Samuel R. et. al. “A Fast Unified Model for Parsing and Sentence Understanding.” *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics. https://arxiv.org/abs/1603.06021

[3] Pennington, Jeffrey et. al. “GloVe: Global Vectors for Word Representation.” 2014. https://nlp.stanford.edu/projects/glove/

[4] Bradbury, James. “Recursive Neural Networks with PyTorch.” *NVIDIA Developer Blog*, 9 Apr. 2017. https://devblogs.nvidia.com/recursive-neural-networks-pytorch/

[5] Warstadt, Alex et. al. “BLiMP: A Benchmark of Linguistic Minimal Pairs for English.” *arXiv preprint*, 2019. https://arxiv.org/abs/1912.00582

[6] de Marneffe, Marie-Catherine et. al. “Generating Typed Dependency Parses from Phrase Structure Parses.” *LREC*, 2006. https://nlp.stanford.edu/software/lex-parser.shtml

[7] Dai, Zihang et. al. “Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context.” *arXiv preprint*, 2019. https://arxiv.org/abs/1901.02860