Dependency Parsing with Stack Long Short–term Memory and Neural Network

Manish Kumar (kumar20@iu.edu)
Arpit Agarwal (arpiagar@iu.edu)
Rahul Sampat (rrsampat@umail.iu.edu)

ABSTRACT

In this project, we try to learn representations of parser states in transition based dependency parsers. In recent years, Dependency-based methods for syntactic parsing have become increasingly popular in natural language processing world. A dependency parser analyzes the grammatical structure of a sentence, establishing relationships between "head" words and words which modify those heads. On the other hand,We also know that Deep Neural Networks (DNNs) are powerful models that have achieved excellent performance on difficult learning tasks. Although DNNs work well whenever large labeled training sets are available, they cannot be used to map sequences to sequences. To overcome this issue, we have used a general end-to-end approach to sequence learning that makes minimal assumptions on the sequence structure. Thus this method uses a multilayered Long Short-Term Memory (LSTM) to map the input sequence to a vector of a fixed dimensionality, and then another deep LSTM to decode the target sequence from the vector.

This enabled us to formulate an efficient parsing model that captures three aspects of a parser's state: **(i)** unbounded look-ahead into the buffer of incoming words, **(ii)** the complete history of actions taken by the parser, and **(iii)** the complete contents of the stack of partially built tree fragments, including their internal structures. We have used standard backpropagation techniques that are used for training and yield state-of-the-art parsing performance.

1. Introduction:

Despite a long and rich history in descriptive linguistics, dependency grammar has until recently played a fairly marginal role both in theoretical linguistics and in natural language processing. The increasing interest in dependency-based representations in natural language parsing in recent years appears to be motivated both by the potential usefulness of bi-lexical relations in disambiguation and by the gains in efficiency that result from the more constrained parsing problem for these representations. There is a

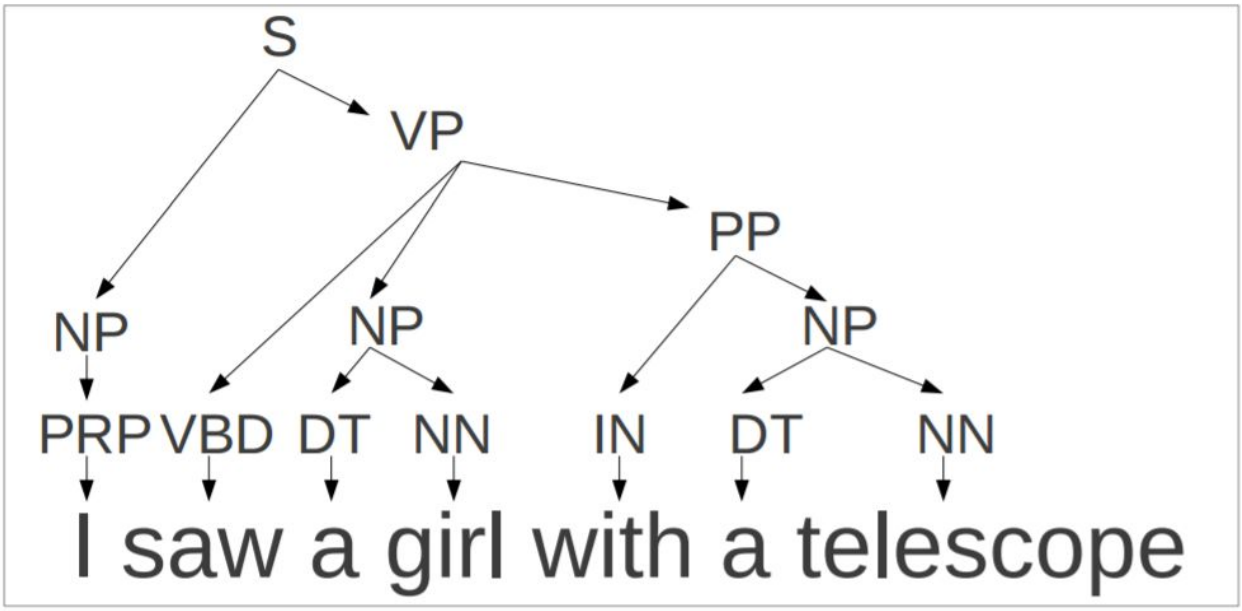
tradeoff between the expressivity of syntactic representations and the complexity of syntactic parsing, and we believe that dependency representations provide a good compromise in this respect

In fact it's a matter of great pride for us as our university's professor Sandra Kubler has done profound work in the field of Dependency Parsing. (Dependency Parsing from Kübler, McDonald, and Nivre). In a nutshell, Transition-based dependency parsing formalizes the parsing problem as a series of decisions that read words sequentially from a buffer and combine them incrementally into syntactic structures. One of the major advantages of this formalization is that the number of operations required to build any projective parse tree is linear in the length of the sentence, making transition-based parsing computationally efficient relative to graph- and grammar based formalisms. But , the challenge encountered in transition based parsing is modeling which action should be taken in each of the unboundedly many states encountered as the parser progresses. To overcome this challenge, researchers have been developing alternative transition sets that simplify the modeling problem by making better attachment decisions through feature engineering and using neural networks. To build on these previous works by learning representations of the parser state that are sensitive to the complete contents of the parser's state: that is, the complete input buffer, the complete history of parser actions, and the complete contents of the stack of partially constructed syntactic structures. This new approach of "global" sensitivity to the state is in complete contrasts with previous works in transition based dependency parsing that uses only a narrow view of the parsing state when constructing representations Although the parser developed by us integrates large amounts of information, the representation used for prediction at each time step is constructed incrementally, and therefore parsing and training time remain linear in the length of the input sentence. Advent of Recurrent neural networks with long short-term memory units (LSTMs) enabled us to achieve this milestone.

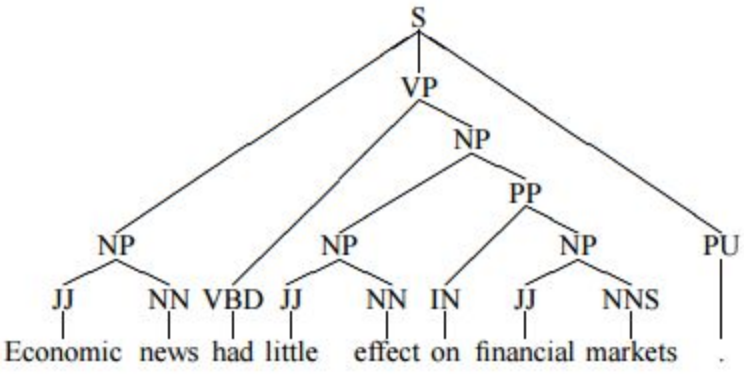
This parsing model uses three stack LSTMs: one representing the input, one representing the stack of partial syntactic trees, and one representing the history of parse actions to encode parser states . Since the stack of partial syntactic trees may contain both individual tokens and partial syntactic structures, representations of individual tree fragments are computed compositionally with recursive neural networks. The parameters are learned with backpropagation and we obtain state-of-the-art results on English dependency parsing tasks .

2.1 Dependency Parsing: Primarily, there are 2 types of parsing:

1. **Phrase structure:** A dependency tree or phrase structure focuses on identifying relations using recursive tree. In this nodes are labeled with words from V, and where the root node is labeled with the special symbol ROOT. The children of a node are ordered with respect to each other and the dependency representations was intimately tied to formalizations of dependency grammar that were very close to context-free grammar and the node itself, so that the node has both left children that precede it and right children that follow it



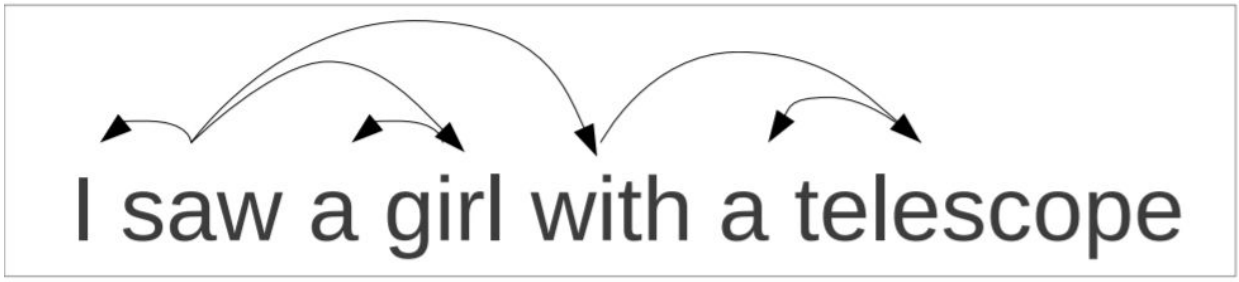
Phrase Structure



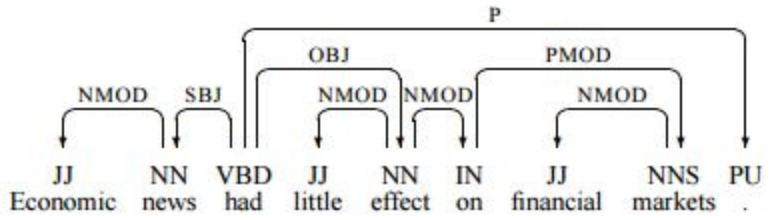
Constituent structure for English sentence from the Penn Treebank

2. Dependency: In this we rely on the relation between the words. It is based on the idea that the syntactic structure of a sentence consists of binary asymmetrical

relations between the words of the sentence. The rules or protocols for deciding dependency relations, and for distinguishing the head and the dependent in such relations, are clearly of central importance for dependency grammar.



Dependency Parsing



Dependency structure for English sentence from the Penn Treebank

Transition based dependency parser builds a parser by performing a linear scan over the words of a sentence. At each step it maintains a partial parse, a stack of words which are currently being processed, and a buffer of words yet to be processed. The parser continues to apply transitions to its state until its buffer is empty and the dependency graph is completed.

The initial state is to have all of the words in order on the buffer, with a single dummy ROOT node on the stack. The following transitions can be applied:

LEFT-ARC: Marks the second item on the stack as a dependent of the first item, and removes the second item from the stack (if the stack contains at least two items).

RIGHT-ARC: Marks the first item on the stack as a dependent of the second item, and removes the first item from the stack (if the stack contains at least two items).

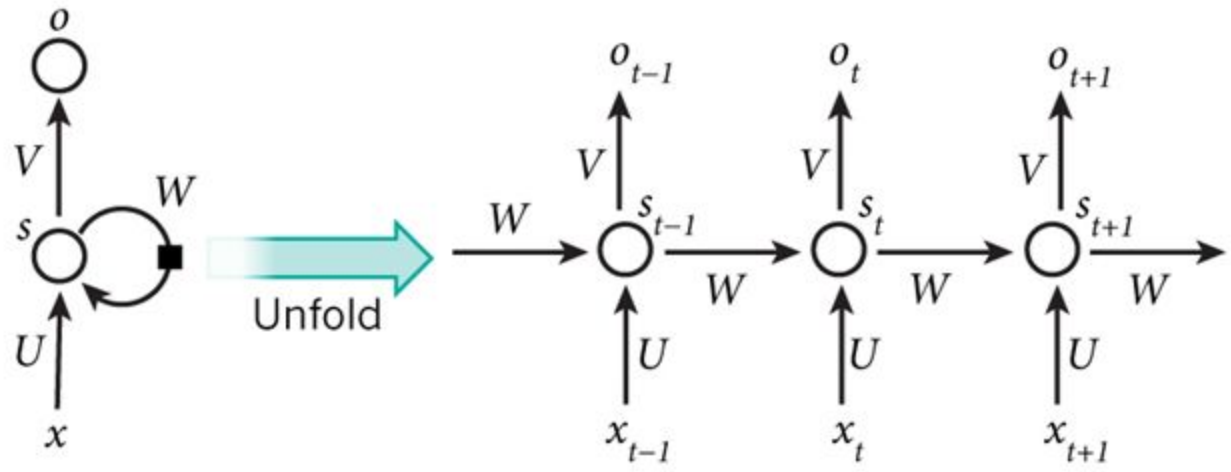
SHIFT: Removes a word from the buffer and pushes it onto the stack (if the buffer is not empty).

Having these three transitions defined, a parser can now generate any projective dependency parse. The parser decides among transitions at each state using a neural network classifier. Distributed representations (dense, continuous vector representations) of the parser's current state are provided as inputs to this classifier, which then chooses among the possible transitions to make next.

2.2 Recurrent Neural Network:

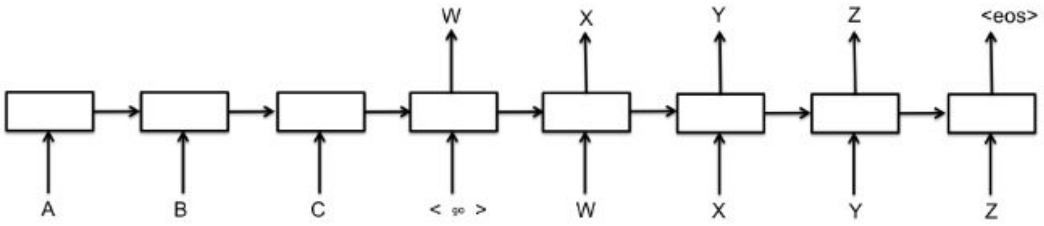
A recurrent neural network (RNN) is a kind of neural network in which connections between units form a directed cycle. Unlike feedforward neural networks, RNNs can use their internal memory to process arbitrary sequences of inputs. This makes them applicable to tasks such as unsegmented connected handwriting recognition or speech recognition. Hence RNN's are considered to be very useful when it comes to application in the fields of Natural Language Processing

There are various different architectures of RNN such as – Fully Recurrent Network, Recursive Neural Network etc.



The idea behind RNNs is to make use of sequential information. In a traditional neural network we assume that all inputs (and outputs) are independent of each other. But for many tasks that's a very bad idea. If you want to predict the next word in a sentence you better know which words came before it. RNNs are called *recurrent* because they perform the same task for every element of a sequence, with the output being depended on the previous computations. Another way to think about RNNs is that they have a "memory" which captures information about what has been calculated so far.

2.3 Sequence-to-Sequence Learning: Deep learning models have shown great results in solving various computer vision problems like visual object tracking , speech recognition. We can use these models to solve the number of tasks in the field of Natural Language processing. Recent works on language modelling, paraphrase detection and word embedding extraction using DNNs have shown some promising results. Sequences-to-sequence parsing poses a challenge for DNNs because they require that the dimensionality of the inputs and outputs is known and fixed. Sequence to sequence prediction attempts to predict words of a sequence on the basis of the preceding words. This means that wi,wi+1,wi+2,.....wj → wj+1 . i.e. for a given sentence with words wi,wi+1,wi+2,.....wj we can make predictions of next word wj+1. This can lead to sequence generation as we are adding new words to the sequence on the basis of previous or existing words. We can check if this new sequence is legitimate by doing sequence recognition. It is generally a difficult task as designing models(Markov chains, Hidden Markov models, or a recurrent neural Network Model) to predict the validity is very difficult.



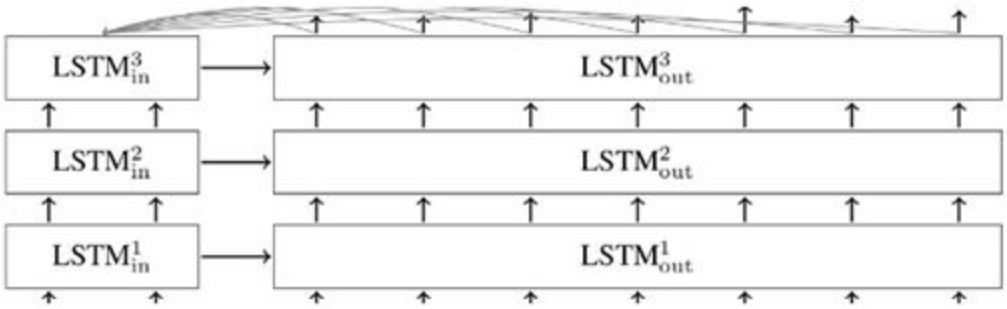
A basic sequence-to-sequence model consists of two recurrent neural networks (RNNs):

1. Encoder that processes the input
2. decoder that generates the output.

Square box in the picture above represents a cell of the RNN, most commonly a GRU cell or an LSTM cell. Every input has to be encoded into a fixed-size state vector, as that is the only thing passed to the decoder. Encoder and decoder can share weights or, as is more common, use a different set of parameters. Sequence-to-sequence models uses multi-layer cells successfully.

2.4 Long short-term memory (LSTM): Long short-term memory (LSTM) is a recurrent neural network (RNN) architecture proposed in 1997 by Sepp Hochreiter and Jürgen Schmidhuber. LSTM networks are quite popular these days.

Below is a simple representation of a LSTM:



LSTMs don't have a fundamentally different architecture from RNNs, but they use a different function to compute the hidden state. LSTM's are normally augmented by recurrent gates called forget gates.

The memory in LSTMs are called *cells* and you can think of them as black boxes that take as input the previous state and current input . Internally these cells decide what to keep in (and what to erase from) memory. LSTM RNNs prevent back-propagated errors from vanishing or exploding.

They then combine the previous state, the current memory, and the input. It turns out that these types of units are very efficient at capturing long-term dependencies. Unlike traditional RNNs, an LSTM network is well-suited to learn from experience to classify, process and predict time series when there are very long time lags of unknown size between important events. This is one of the main reasons why LSTM outperforms alternative RNNs and hidden Markov models and other sequence learning methods in numerous applications. For example, LSTM achieved the best known results in unsegmented connected handwriting recognition and automatic speech recognition.

2.4 Stack Long Short-Term Memories: As we all know that the conventional LSTM's model would sequence in a left-right order. Although there are some variations which include bidirectional LSTM's and multidimensional LSTM's.
But after studying Chris Dyer's and Miguel Ballesteros' work on Transition-Based Dependency Parsing with Stack Long Short-Term Memory, we decided to go with an

approach/model involving Stack Long Short-Term Memories. In this approach, the authors augment the LSTM with a “stack pointer”.
Slack LSTM’s, like conventional LSTM’s add the new inputs in the right most position, but unlike the conventional LSTM’s, the current location’s slack pointer is used to determine which cell in the LSTM provides c_{t-1} and h_{t-1} when computing the contents of the new memory cell. This gives a special ability to SLSTM. It now provides PUSH & POP operations. The POP operation moves the stack pointer to the previous element, whereas the PUSH operation always adds a new entry (and does not overwrite and entry) whose pointer points towards the first element and then the POP updates the stack pointer.

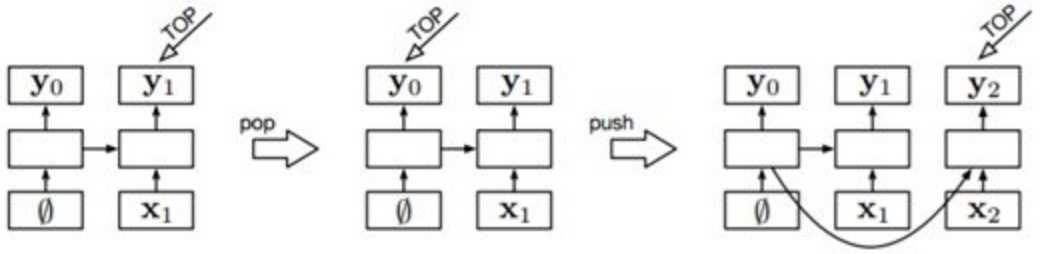


Figure 1: A stack LSTM extends a conventional left-to-right LSTM with the addition of a stack pointer (notated as TOP in the figure).

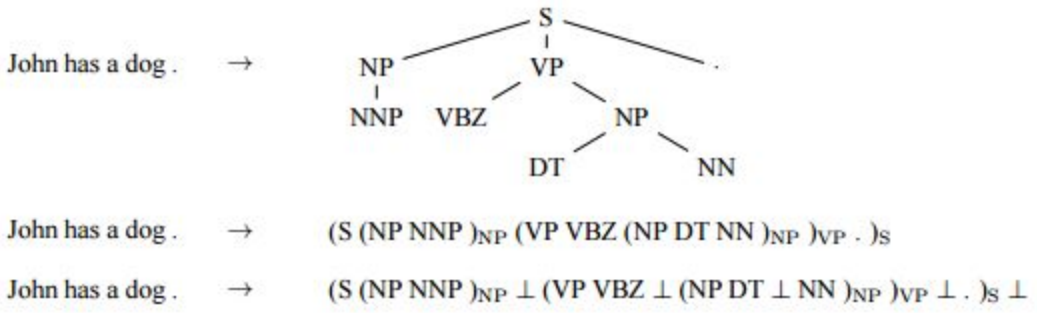
Also, the SLSTM has the flexibility to extract information from arbitrary points in the stack.

3. Algorithm: Our algorithm systematically tries to solve the the problem of learning representations of dependency parsers with the help of Recurrent Neural Network. We preserve the standard data structures of a transition-based dependency parser, namely a buffer of words (B) to be processed and a stack (S) of partially constructed syntactic elements. Each stack element is augmented with a continuous-space vector embedding representing a word and, in the case of S, any of its syntactic dependents. Additionally, we introduce a third stack (A) to represent the history of actions taken by the parser.Each of these stacks is associated with a stack LSTM that provides an encoding of their current contents.

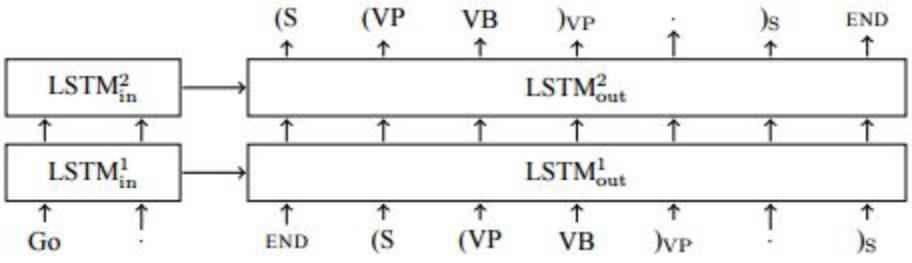
3.1 Converting Word Vectors: To this algorithm to work, first we need to convert word into continuous space vector. We used ‘word2vec’ tool to convert word into vector of this continuous space vector space.

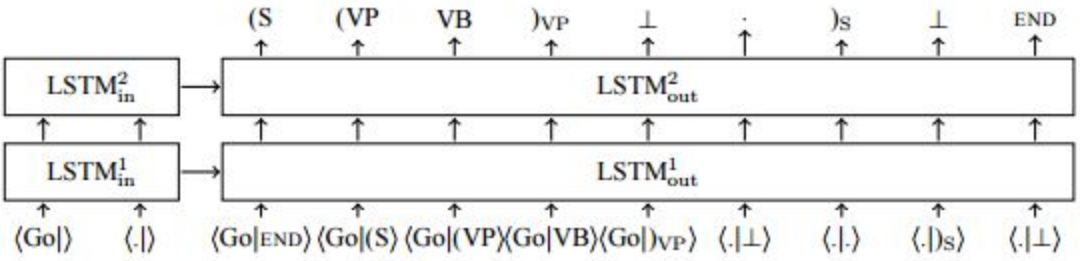
3.2 Parse Operation: We now reverse the input sentence onto B such that first word is at top and ROOT symbol is at bottom, with S and A each containing an empty stack toke. Now, the dependency parser is initialized by pushing these words along with their representations. While parsing, at each step, we predict the next action to be taken by computing a composite representation of stack states, thus updating the stack. This process is repeated until B gets empty (except for the empty-stack symbol). Finally, we would have S with two elements, one representing full parse tree with ROOT symbol and an empty-stack and A containing history of operations taken by the parser.

A simple implementation of dependency parsing using LSTM.



Example parsing task, a basic linearization, and one with stack control symbols.





3.3 Arc Standard Transition Operation: Arc-standard transitions construct a dependency tree in bottom-up fashion, since right-dependents of head are only

attached after the subtree until fully parsed. They use stack for storing syntactic structures while parsing a sentence from left to right and a buffer that keeps the incoming tokens to be parsed. Due to the recursive method implemented by the parser, the construction of the tree guarantees that algorithm will not try to find another head for the dependent structure, once a head is modified allowing evaluation of composite representations of tree fragments incrementally**.**

3.4 Token Embeddings: To represent each input token, we concatenate three vectors: a learned vector representation for each word type (w); a fixed vector representation from a neural language model (w˜ LM), and a learned representation (t) of the POS tag of the token, provided as auxiliary input to the parser.

3.5 Composition Functions: Recursive neural network models are used as composition functions enabling complex phrases to be represented in their parts and relations that link them by embedding previous lines dependency tree fragments that are present in the stack S in same vector space as the token embedding discussed above. Parameterization of the composite function is simplified combining head modifier pairs one at a time, building up more complicated structures in the order they are reduced in parser.

4. Procedure:

We used the Stanford Dependencency (SD) treebank. Stanford dependencies provides a representation of grammatical relations between words in a sentence. They have been designed to be easily understood and effectively used by people who want to extract textual relations.
2 graphical representations: the standard dependencies (collapsed and propagated) and the basic dependency representation in which each word in the sentence (except the head of the sentence) is the dependent of one other word (no collapsing, no propagation).

As a part of training, we maximized the conditional log-likelihood of treebank parses given sentences. In our approach, for each sentence, we compute a graph which is used to run forward and back-propagation to obtain the gradients of this objective with respect to the model parameters. The computations for a single parsing model were run on a single thread on a CPU. Using the mentioned dimensions, we required a huge amount of time for convergence, due to limited number of resources.

Parameter optimization was performed using stochastic gradient descent with an initial learning rate of $\eta_0 = 0.5$. The learning rate was updated on each pass through the training data as $\eta_t = \eta_0/(1 + \rho_t)$, with $\rho = 0.2$ and where t is the number of epochs completed. Any kind of momentum was not used. To mitigate the effects of "exploding" gradients, a `2 penalty of 1×10^{-6} was applied to all weights.

The full version of our parsing model sets dimensionalities as follows. LSTM hidden states are of size 100, and we use two layers of LSTMs for each stack. Pre Trained word embeddings, the learned word embeddings and Part of speech embeddings have a value of their own respective dimensions. These dimensions were chosen based on intuitively reasonable values (words should have higher dimensionality than parsing actions, POS tags, and relations; LSTM states should be relatively large)

Future work might more carefully optimize these parameters to balance between minimizing computational expense and finding solutions that work.

5. Result: We are quite happy with the result obtained in the very short span of time available to us. Overall, our parser substantially outperforms the baseline neural network parser. Although we obtained reasonable parsing performance in some limited cases only, but that can be attributed to the limited data set with the limited availability of the computing resources as we did not have access to GPU . We also learned the benefits of transition based dependency in PCFG analysis over phrasal structure. We also find that using composed representations of dependency tree fragments outperforms using representations of head words alone, which has implications for theories of headedness. Finally, we find that while LSTMs outperform baselines that use only classical RNNs, these are still quite capable of learning good representations.

6. Conclusion:

In the above experiments, we learned and presented stack LSTMs and recurrent neural networks for sequences (with push and pop operations). We then saw how these can be used to implement a transition-based dependency parser. We conclude by remarking that stack memory offers intriguing possibilities for learning to solve general information processing problems.

Here, we learned from observable stack manipulation operations and the computed embeddings of final parser states were not used for any further prediction.

Such an extension of the work would make it an alternative to architectures that have an explicit external memory such as neural Turing machines and memory networks.

However, as with those models, without supervision of the stack operations, formidable computational challenges must be solved, but sampling techniques and techniques from reinforcement learning have shown promises, making this an intriguing avenue for future work.

7. Future Work: Given time and access to computing resources, we would like to train our model on GPU instead of CPU. It will give us the flexibility to train our model further and help in learn better representation. We would also like to delve deep into dependency parsing further and how it can be integrated better with the current innovative neural network methods to get even better result.

References:

1. http://stp.lingfil.uu.se/~nivre/docs/eacl3.pdf
2. http://www.phontron.com/slides/nlp-programming-en-11-depend.pdf
3. https://arxiv.org/pdf/1505.08075.pdf
4. http://stp.lingfil.uu.se/~nivre/docs/eacl3.pdf
5. http://nlp.stanford.edu/software/nndep.shtml
6. http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf
7. https://arxiv.org/pdf/1505.08075.pdf
8. http://anthology.aclweb.org/D/D13/D13-1176.pdf
9. http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf
10. https://arxiv.org/pdf/1506.03134v1.pdf
11. http://www.morganclaypool.com/doi/pdf/10.2200/S00169ED1V01Y200901HLT002
12. http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/
13. http://www.petrovi.de/data/arxiv15.pdf
14. http://stp.lingfil.uu.se/~nivre/docs/05133.pdf