CS224n: Natural Language Processing with Deep Learning [1]

Lecture Notes: Part IV

Dependency Parsing [2]

Winter 2019

[1] Course Instructors: Christopher Manning, Richard Socher

[2] Authors: Lisa Wang, Juhi Naik, and Shayne Longpre

Keyphrases: Dependency Parsing.

1 *Dependency Grammar and Dependency Structure*

Parse trees in NLP, analogous to those in compilers, are used to analyze the syntactic structure of sentences. There are two main types of structures used - constituency structures and dependency structures.

Constituency Grammar uses phrase structure grammar to organize words into nested constituents. This will be covered in more detail in following chapters. We now focus on Dependency Parsing.

Dependency structure of sentences shows which words depend on (modify or are arguments of) which other words. These binary asymmetric relations between the words are called dependencies and are depicted as arrows going from the **head** (or governor, superior, regent) to the **dependent** (or modifier, inferior, subordinate). Usually these dependencies form a tree structure. They are often typed with the name of grammatical relations (subject, prepositional object, apposition, etc.). An example of such a dependency tree is shown in Figure 1. Sometimes a fake ROOT node is added as the head to the whole tree so that every word is a dependent of exactly one node.

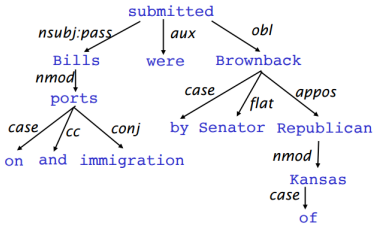

Figure 1: Dependency tree for the sentence "Bills on ports and immigration were submitted by Senator Brownback, Republican of Kansas"

1.1 *Dependency Parsing*

Dependency parsing is the task of analyzing the syntactic dependency structure of a given input sentence S. The output of a dependency parser is a dependency tree where the words of the input sentence are connected by typed dependency relations. Formally, the dependency parsing problem asks to create a mapping from the input sentence with words $S = w_0 w_1 ... w_n$ (where w_0 is the ROOT) to its dependency tree graph G. Many different variations of dependency-based methods have been developed in recent years, including neural network-based methods, which we will describe later.

To be precise, there are two subproblems in dependency parsing (adapted from Kuebler et al., chapter 1.2):

1. *Learning:* Given a training set D of sentences annotated with dependency graphs, induce a parsing model M that can be used to parse new sentences.

2. *Parsing:* Given a parsing model M and a sentence S, derive the optimal dependency graph D for S according to M.

1.2 Transition-Based Dependency Parsing

Transition-based dependency parsing relies on a state machine which defines the possible transitions to create the mapping from the input sentence to the dependency tree. The *learning problem* is to induce a model which can predict the next transition in the state machine based on the transition history. The *parsing problem* is to construct the optimal sequence of transitions for the input sentence, given the previously induced model. Most transition-based systems do not make use of a formal grammar.

1.3 Greedy Deterministic Transition-Based Parsing

This system was introduced by Nivre in 2003 and was radically different from other methods in use at that time.

This transition system is a state machine, which consists of *states* and *transitions* between those states. The model induces a sequence of transitions from some *initial* state to one of several *terminal* states.

States:

For any sentence $S = w_0 w_1 ... w_n$, a state can be described with a triple $c = (\sigma, \beta, A)$:

1. a stack σ of words w_i from S,
2. a buffer β of words w_i from S,
3. a set of dependency arcs A of the form (w_i, r, w_j), where w_i, w_j are from S, and r describes a dependency relation.

It follows that for any sentence $S = w_0 w_1 ... w_n$,

1. an *initial* state c_0 is of the form $([w_0]_\sigma, [w_1, ..., w_n]_\beta, \emptyset)$ (only the ROOT is on the stack σ, all other words are in the buffer β and no actions have been chosen yet),
2. a terminal state has the form $(\sigma, []_\beta, A)$.

Transitions:

There are three types of transitions between states:

1. SHIFT: Remove the first word in the buffer and push it on top of the stack. (Pre-condition: buffer has to be non-empty.)
2. LEFT-ARC$_r$: Add a dependency arc (w_j, r, w_i) to the arc set A, where w_i is the word second to the top of the stack and w_j is the

1. Shift $\sigma, w_i|\beta, A \rightarrow \sigma|w_i, \beta, A$
2. Left-Arc$_r$ $\sigma|w_i|w_j, \beta, A \rightarrow \sigma|w_j, \beta, A \cup \{r(w_j, w_i)\}$
3. Right-Arc$_r$ $\sigma|w_i|w_j, \beta, A \rightarrow \sigma|w_i, \beta, A \cup \{r(w_i, w_j)\}$

Figure 2: Transitions for Dependency Parsing.

word at the top of the stack. Remove w_i from the stack. (Precondition: the stack needs to contain at least two items and w_i cannot be the ROOT.)

3. RIGHT-ARC$_r$: Add a dependency arc (w_i, r, w_j) to the arc set A, where w_i is the word second to the top of the stack and w_j is the word at the top of the stack. Remove w_j from the stack. (Precondition: The stack needs to contain at least two items.)

A more formal definition of these three transitions is presented in Figure 2.

1.4 *Neural Dependency Parsing*

While there are many deep models for dependency parsing, this section focuses specifically on greedy, transition-based neural dependency parsers. This class of model has demonstrated comparable performance and significantly better efficiency than traditional feature-based discriminative dependency parsers. The primary distinction from previous models is the reliance on dense rather than sparse feature representations.

The model we will describe employs the arc-standard system for transitions, as presented in section 1.3. Ultimately, the aim of the model is to predict a transition sequence from some initial configuration c to a terminal configuration, in which the dependency parse tree is encoded. As the model is greedy, it attempts to correctly predict one transition $T \in \{\text{SHIFT}, \text{LEFT-ARC}_r, \text{RIGHT-ARC}_r\}$ at a time, based on features extracted from the current configuration $c = (\sigma, \beta, A)$. Recall, σ is the stack, β the buffer, and A the set of dependency arcs for a given sentence.

Feature Selection:

Depending on the desired complexity of the model, there is flexibility in defining the input to the neural network. The features for a given sentence S generally include some subset of:

1. S_{word}: Vector representations for some of the words in S (and their dependents) at the top of the stack σ and buffer β.

2. S_{tag}: Part-of-Speech (POS) tags for some of the words in S. POS tags comprise a small, discrete set: $\mathcal{P} = \{NN, NNP, NNS, DT, JJ, ...\}$

3. S_{label}: The arc-labels for some of the words in S. The arc-labels comprise a small, discrete set, describing the dependency relation: $\mathcal{L} = \{amod, tmod, nsubj, csubj, dobj, ...\}$

For each feature type, we will have a corresponding embedding matrix, mapping from the feature's one hot encoding, to a d-dimensional dense vector representation. The full embedding matrix for S_{word} is $E^w \in \mathbb{R}^{d \times N_w}$ where N_w is the dictionary/vocabulary size. Correspondingly, the POS and label embedding matrices are $E^t \in \mathbb{R}^{d \times N_t}$ and $E^l \in \mathbb{R}^{d \times N_l}$ where N_t and N_l are the number of distinct POS tags and arc labels.

Lastly, let the number of chosen elements from each set of features be denoted as n_{word}, n_{tag}, and n_{label} respectively.

Feature Selection Example:

As an example, consider the following choices for S_{word}, S_{tag}, and S_{label}.

1. S_{word}: The top 3 words on the stack and buffer: $s_1, s_2, s_3, b_1, b_2, b_3$. The first and second leftmost / rightmost children of the top two words on the stack: $lc_1(s_i), rc_1(s_i), lc_2(s_i), rc_2(s_i)$, $i = 1, 2$. The leftmost of leftmost / rightmost of rightmost children of the top two words on the stack: $lc_1(lc_1(s_i)), rc_1(rc_1(s_i))$, $i = 1, 2$. In total S_{word} contains $n_w = 18$ elements.
2. S_{tag}: The corresponding POS tags for S_{tag} ($n_t = 18$).
3. S_{label}: The corresponding arc labels of words, excluding those 6 words on the stack/buffer ($n_l = 12$).

Note that we use a special NULL token for non-existent elements: when the stack and buffer are empty or dependents have not been assigned yet. For a given sentence example, we select the words, POS tags and arc labels given the schematic defined above, extract their corresponding dense feature representations produced from the embedding matrices E^w, E^t, and E^l, and concatenate these vectors into our inputs $[x^w, x^t, x^l]$. At training time we backpropagate into the dense vector representations, as well as the parameters at later layers.

Feedforward Neural Network Model:

The network contains an input layer $[x^w, x^t, x^l]$, a hidden layer, and a final softmax layer with a cross-entropy loss function. We can either define a single weight matrix in the hidden layer, to operate on a concatenation of $[x^w, x^t, x^l]$, or we can use three weight matrices $[W_1^w, W_1^t, W_1^l]$, one for each input type, as shown in Figure 3. We then apply a non-linear function and use one more affine layer $[W_2]$ so that there are an equivalent number of softmax probabilities to the number of possible transitions (the output dimension).

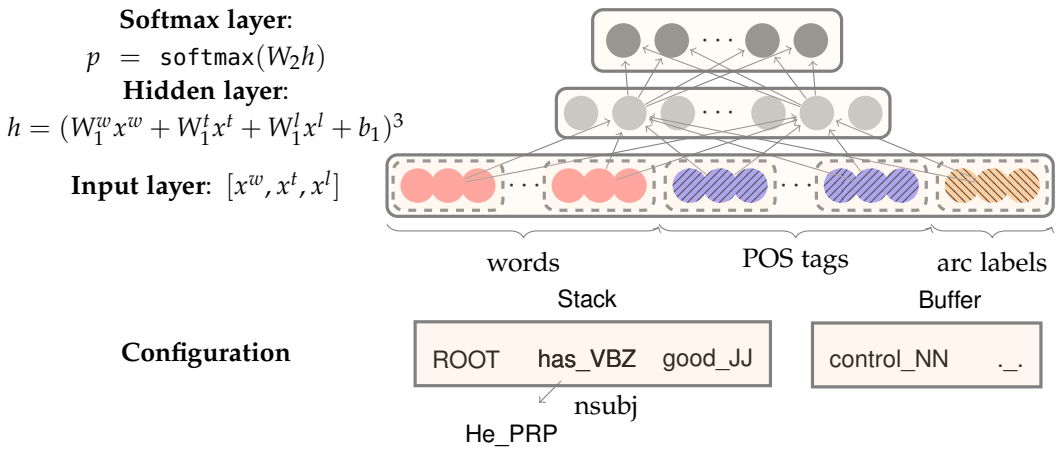

Figure 3: The neural network architecture for greedy, transition-based dependency parsing.

Note that in Figure 3, $f(x) = x^3$ is the non-linear function used.

For a more complete explanation of a greedy transition-based neural dependency parser, refer to "A Fast and Accurate Dependency Parser using Neural Networks" under Further Reading.

Further reading:

Danqi Chen, and Christopher D. Manning. "A Fast and Accurate Dependency Parser using Neural Networks." EMNLP. 2014.

Kuebler, Sandra, Ryan McDonald, and Joakim Nivre. "Dependency parsing." Synthesis Lectures on Human Language Technologies 1.1 (2009): 1-127.