A deep architecture for non-projective dependency parsing

Conference of the North American Chapter of the Association for Computational Linguistics - Human Language Technologies; Workshop on Vector Space Modeling for Natural Language Processing, I, 2015, Denver.
http://www.producao.usp.br/handle/BDPI/48991

Downloaded from: Biblioteca Digital da Produção Intelectual - BDPI, Universidade de São Paulo
A Deep Architecture for Non-Projective Dependency Parsing

Erick R. Fonseca
University of São Paulo
Avenida Trabalhador São-carlense, 400
São Carlos, Brazil
erickrf@icmc.usp.br

Sandra M. Aluísio
University of São Paulo
Avenida Trabalhador São-carlense, 400
São Carlos, Brazil
sandra@icmc.usp.br

Abstract

Graph-based dependency parsing algorithms commonly employ features up to third order in an attempt to capture richer syntactic relations. However, each level and each feature combination must be defined manually. Besides that, input features are usually represented as huge, sparse binary vectors, offering limited generalization. In this work, we present a deep architecture for dependency parsing based on a convolutional neural network. It can examine the whole sentence structure before scoring each head/modifier candidate pair, and uses dense embeddings as input. Our model is still under ongoing work, achieving 91.6% unlabeled attachment score in the Penn Treebank.

1 Introduction

Graph-based dependency parsing works by assigning scores to each possible dependency arc between two words (plus the root), and then creating a dependency tree by selecting the arcs which yield the highest score sum (McDonald et al., 2005). The Chu-Liu-Edmonds algorithm is commonly used to extract the maximum spanning tree (MST) of the resulting graph in polynomial time, and inherently allows for non-projective trees.

Most such parsing algorithms obtain the score for an arc from word $i$ to $j$ as the dot product of a weight vector and a vector of binary features, $s(i,j) = w \cdot f(i,j)$. Their training procedure is thus essentially optimizing the weight vector.

The features, however, often follow redundant patterns: the same classifier may use as separate features: (i) head word and its POS tag, (ii) head word, and (iii) head word POS tag. This is justified first by data sparseness, since a given word may not have been seen many times in the training set (or not with a given POS tag), and the last two features serve as a fallback. Second, most approaches are based on linear classifiers, which cannot learn complex interactions between features.

Given that the scoring function deals with an arc at a time, graph-based parsers are usually restricted to features of local pairs. This is problematic when determining the head of a given word depends on its modifiers. For example, consider the two sentences in Figure 1, where the preposition with may be attached to a verb or a noun, depending on its complement. Including neighboring words as features in the arc scoring function may alleviate the problem, but doesn’t account for long range dependencies. A more efficient solution is second or high order features, which include child or sibling arcs in the scoring function (McDonald and Pereira, 2006). Some authors explored higher order features, including, for example, grandparents and grand-siblings (Koo and Collins, 2010) or non-adjacent siblings (Carreras, 2007).

However, each new level (i.e., each higher order) must be defined through manually designed features. Furthermore, finding the exact non-projective MST in such cases is computationally intractable, making it necessary to resort to approximate solutions.$^1$

Another disadvantage of such systems is that fea-

$^1$The projective MST, however, can be obtained in $O(n^{m+1})$ time for a model of $m$-th order. A common practice is to find the projective MST and then swap some edges.
He ate spaghetti with a fork
He ate spaghetti with meatballs
Figure 1: Example of dependency trees with different head words for with, depending on its complement.

tures are usually binary. Thus, each word in the system vocabulary is represented as a separate, independent feature. By contrast, a growing trend in the NLP community is to use word embeddings, which are low dimensional, dense vectors representing words (Turian et al., 2010; Collobert, 2011; Mikolov et al., 2013). Word embeddings have the advantage to deliver similar representations to words that tend to occur in the same contexts (and usually have a related meaning), and lower out-of-vocabulary impact.

In this work, we address the limitations described above with a graph-based parser architecture inspired in the SENNA system (Collobert, 2011). It takes word embeddings and POS tags as input, and uses a convolutional neural network that allows it to examine the whole sentence before giving a score for each head-dependent pair. The complexity of the scoring procedure is \( O(n^3) \).

The remaining of this paper is organized as follows. Section 2 presents relevant related work with dependency parsing, word embeddings and neural architectures. Section 3 describes our model. Section 4 shows our experimental setup and results found for English, German and Dutch, and Section 5 presents our conclusions.

2 Related Work

Graph based parsers were combined with transition based ones in studies aimed at exploiting global features, which fit better with the latter (Martins et al., 2008; Nivre and McDonald, 2008). Beam search has also been used instead of exact inference in order to allow more complex features and keep the problem computationally tractable (Zhang and Clark, 2008). In contrast, our method works by examining the whole sentence in a straightforward manner before assigning a score to an arc.

There has also been studies on generating word embeddings based on syntactic relations of each word instead of its neighbors in a fixed size window (Padó and Lapata, 2007). Recently, Bansal et al. (2014) and Levy and Goldberg (2014) used similar variants of the skip-gram model (Mikolov et al., 2013) to this end: both studies parsed huge corpora with a dependency parser and then used dependency relations as context for the skip-gram algorithm.

The skip-gram model induces word representations such as to maximize the capabilities of predicting neighboring words \( w' \) given a word \( w \). By considering neighbors the words with a dependency edge between them, instead of merely occurring near each other, the embeddings are able to capture more syntactic knowledge.

Some other studies employed neural architectures and word embeddings to address parsing. Socher et al. (2013), for example, recurrently combined word vectors into phrase vectors in constituency-based parse trees. Chen and Manning (2014) used an MLP network with one hidden layer to perform transition-based dependency parsing. Their network decides, for each state configuration, which action to take next.

More related to this work, Collobert (2011) used a convolutional network to address constituent parsing. Words are tagged in multiple levels, according to the constituents they are part of. A key component of the network is the convolution layer, which is capable of turning the representation of a sentence of variable size into a fixed size vector.

A very similar architecture had been previously used by Collobert et al. (2011) to perform semantic role labeling. For this task, the network had to classify each token with respect to each predicate in the sentence. We draw on this idea, making our dependency parser, implemented as a convolutional neural network, score each word with respect with a candidate head.

3 Deep Architecture

A way to avoid the need of defining each higher level of features manually is a deep architecture that examines the whole sentence before making each local decision.
Our parser first identifies unlabeled dependency arcs between words and then labels them. In the first stage, it computes a score \( s(h, m, x) \) for assigning a given head \( h \) to a modifier word \( m \) within a sentence \( x \). After having computed scores for all \((h, m)\) combinations, we perform the Chu-Liu-Edmond's algorithm to find the maximum spanning tree.

Then, in the second stage, for each pair \((h, m)\) previously detected, we must label the arc connecting the words. We assign a score \( s(l, h, m, x) \) for each possible label \( l \), and the label \( l' \) with the highest score is selected by the parser.

### 3.1 Word Representations

Each word \( t \) is represented as a concatenation of four embedding vectors: one representing the word itself, one for its POS tag, one for the relative distance between \( t \) and \( h \) and one for the relative distance between \( t \) and \( m \). As such, the final representation varies according to each pair \((h, m)\) being processed.

The four vectors mentioned above have independent dimensions \( d_{\text{word}}, d_{\text{POS}}, d_{\text{hdist}} \) and \( d_{\text{mdist}} \). The vectors are drawn from matrices \( M_{\text{word}}, M_{\text{POS}}, M_{\text{hdist}} \) and \( M_{\text{mdist}} \). As usual in research with vector space models, we take advantage of previously trained embeddings to initialize \( M_{\text{word}} \). The other three matrices are initialized randomly; all four are adjusted during training.

The relative distance between two words \( t_1 \) and \( t_2 \) is determined as the difference in their positions in the sentence, clipped to a maximum absolute value:

\[
\text{dist}(t_1, t_2) = \min(\alpha, \max(-\alpha, i - j)) \tag{1}
\]

Where \( i \) and \( j \) are the numerical positions of \( t_1 \) and \( t_2 \) in \( x \), and \( \alpha \) is a threshold value. A positive distance means that \( t_1 \) comes first in the sentence, and a negative distance means otherwise. The matrices \( M_{\text{hdist}} \) and \( M_{\text{mdist}} \) need \( 2\alpha + 3 \) entries: each positive and negative distance, plus a vector for distances greater than the threshold (also positive and negative) and zero. Zero distance means that \( t_1 \) and \( t_2 \) are the same.

### 3.2 Edge Detection

For the edge detection stage, the neural network performs as follows. All possible \((h, m)\) pairs are considered, and all words in the sentence are examined for each decision. A convolution layer turns a variable sized input (i.e., the sentence) into a fixed size intermediate vector.

For each \((h, m)\) candidate pair, the convolution layer applies a default weight matrix multiplication over the vectors representing all words and stores the results:

\[
[C]_i = W_1 \cdot \text{wr}(i, h, m), 1 \leq i \leq |x| \tag{2}
\]

Where \( W_1 \) is a weight matrix, \( \text{wr}(i, h, m) \) is the representation (concatenation of the four vectors) for the \( i \)-th word in the sentence, considering a pair \((h, m)\), \( C \) is a matrix containing the convolution results over the whole sentence and \([C]_i\) denotes its \( i \)-th row.

After all words in the sentence have been examined, each convolution neuron outputs the maximum value it found and a bias is added to the resulting vector. The whole operation is described in Equations 3 and 4.

\[
[c_{\text{max}}]_j = \max_{1 \leq i \leq |x|} [C]_{ij}, 1 \leq j \leq |c_{\text{max}}| \tag{3}
\]

\[
c_{\text{out}} = c_{\text{max}} + b_1 \tag{4}
\]

Where \( c_{\text{max}} \) is the fixed size vector obtained after the convolution and \( c_{\text{out}} \) has the values forwarded to the next layer. Their dimension is equal to the number of convolution neurons. \([c_{\text{max}}]_j\) indicates the \( j \)-th element in the vector, and \([C]_{ij}\) indicates the element at cell \((i, j)\) of the matrix. \( b_1 \) is a bias vector.

The second hidden layer performs another matrix multiplication and adds another bias vector. We apply a non-linear function over the resulting values: for speed, we use a hard version of the hyperbolic tangent, which just clips values greater than 1 or

\[\text{In fact, the actual implementation is slightly different in order to avoid repeated calculations: we store a lookup table with pre-computed values in the convolution layer considering only distance vectors, and when scoring a sentence, we create another lookup table with the results without considering distance vectors. Then, for each \((h, m)\), we just have to sum the appropriate entries.}\]
Also, the output layer has one neuron for each possible label, requiring a weight matrix instead of a weight vector. Thus, instead of Equation 6, we have Equation 9 for determining the network output.

\[ y = W_3 \cdot h + b_3 \]  \hspace{1cm} (9)

\( W_3 \) and \( b_3 \) are, respectively, a weight matrix and a bias vector. We pick the label with the highest score in the output vector \( y \) as the parser answer. During training, we apply a softmax on it in order to determine probabilities for each label. Error gradients are found with the same rationale than edge detection, the only difference being that we maximize the log probability of the correct label instead of the correct head.

4 Experiments

We performed experiments with English, German and Dutch data. For English, we used the default Penn Treebank data set, converted to constituency trees to CoNLL dependencies (Johansson and Nugues, 2007) using the LTH conversion tool\(^3\). We trained on sections 2-21, validated on 22, and tested on section 23. We trained and validated models using gold POS tags; for testing, we used a neural network based tagger trained on the default WSJ POS tagging data set (sections 0-18).

For German and Dutch, we used the CoNLL 2006 datasets. We chose these two languages because they have the highest rate of non-projective edges among all languages in CoNLL 2006, and one of our method’s strengths is precisely finding non-projective edges as easily as it would find projective ones. As common practice, we used gold POS tags in training, validating and testing on these languages.

We report results obtained with the English word embedding matrix \( M_{word} \) initialized with data from SkipDep and Levy and Goldberg (2014)\(^4\) (L&G for short). For German and Dutch, we used word embeddings provided by the Polyglot project\(^5\) (Al-Rfou et al., 2013), generated by a neural language model.

\(^3\)http://nlp.cs.lth.se/software/treebank_converter/
\(^4\)It is important to note that neither of them included the WSJ corpus in the data used to generate the embeddings.
\(^5\)Available at http://bit.ly/embeddings
Table 1: Parameter values used in experiments. (U) indicates the unlabeled stage, and (L) the labeled one. When neither is present, the same configuration was used in both.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_{\text{word}}$ embeddings size (en)</td>
<td>6 100</td>
</tr>
<tr>
<td>$M_{\text{word}}$ embeddings size (de/nl)</td>
<td>64</td>
</tr>
<tr>
<td>$M_{\text{POS}}$ embeddings size</td>
<td>10</td>
</tr>
<tr>
<td>$M_{\text{mdist}}$ embeddings size</td>
<td>5</td>
</tr>
<tr>
<td>$M_{\text{hdist}}$ embeddings size</td>
<td>5</td>
</tr>
<tr>
<td>Distance threshold $\alpha$</td>
<td>10</td>
</tr>
<tr>
<td>Iterations</td>
<td>15</td>
</tr>
<tr>
<td>Learning rate at epoch $i$</td>
<td>0.01</td>
</tr>
<tr>
<td>Convolution layer size (U)</td>
<td>100</td>
</tr>
<tr>
<td>Convolution layer size (L)</td>
<td>200</td>
</tr>
<tr>
<td>Second hidden layer size (U)</td>
<td>500</td>
</tr>
<tr>
<td>Second hidden layer size (L)</td>
<td>200</td>
</tr>
</tbody>
</table>

Table 2: Accuracy values

<table>
<thead>
<tr>
<th></th>
<th>Dev Vectors</th>
<th>Test Vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UAS</td>
<td>LAS</td>
</tr>
<tr>
<td>SKIP$_{\text{DEP}}$</td>
<td>91.9%</td>
<td>89.0%</td>
</tr>
<tr>
<td>L&amp;G</td>
<td>91.6%</td>
<td>88.6%</td>
</tr>
<tr>
<td>Dutch</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>German</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

5 Conclusions

We have presented a graph-based dependency parser built upon a deep architecture as an alternative to explicitly engineered high order features. However, contrary to some advancements recently obtained by such models, ours fell short of state-of-the-art accuracy.

We believe that a more elaborate version of our architecture could achieve competitive performance, while still avoiding the problems related to the input representation pointed out in the introduction. Our code and trained models are available at https://github.com/erickrf/nlpnet.

References


