The Inside-Outside Recursive Neural Network model for Dependency Parsing

Phong Le and Willem Zuidema
Institute for Logic, Language, and Computation
University of Amsterdam, the Netherlands
\{p.le,zuidema\}@uva.nl

Abstract

We propose the first implementation of an infinite-order generative dependency model. The model is based on a new recursive neural network architecture, the Inside-Outside Recursive Neural Network. This architecture allows information to flow not only bottom-up, as in traditional recursive neural networks, but also top-down. This is achieved by computing content as well as context representations for any constituent, and letting these representations interact. Experimental results on the English section of the Universal Dependency Treebank show that the infinite-order model achieves a perplexity seven times lower than the traditional third-order model using counting, and tends to choose more accurate parses in \(k\)-best lists. In addition, reranking with this model achieves state-of-the-art unlabelled attachment scores and unlabelled exact match scores.

1 Introduction

Estimating probability distributions is the core issue in modern, data-driven natural language processing methods. Because of the traditional definition of discrete probability

\[ Pr(A) = \frac{\text{the number of times } A \text{ occurs}}{\text{the size of event space}} \]

counting has become a standard method to tackle the problem. When data are sparse, smoothing techniques are needed to adjust counts for non-observed or rare events. However, successful use of those techniques has turned out to be an art. For instance, much skill and expertise is required to create reasonable reduction lists for back-off, and to avoid impractically large count tables, which store events and their counts.

An alternative to counting for estimating probability distributions is to use neural networks. Thanks to recent advances in deep learning, this approach has recently started to look very promising again, with state-of-the-art results in sentiment analysis (Socher et al., 2013), language modelling (Mikolov et al., 2010), and other tasks. The Mikolov et al. (2010) work, in particular, demonstrates the advantage of neural-network-based approaches over counting-based approaches in language modelling: it shows that recurrent neural networks are capable of capturing long histories efficiently and surpass standard \(n\)-gram techniques (e.g., Kneser-Ney smoothed 5-gram).

In this paper, keeping in mind the success of these models, we compare the two approaches. Complementing recent work that focused on such a comparison for the case of finding appropriate word vectors (Baroni et al., 2014), we focus here on models that involve more complex, hierarchical structures. Starting with existing generative models that use counting to estimate probability distributions over constituency and dependency parses (e.g., Eisner (1996b), Collins (2003)), we develop an alternative based on recursive neural networks. This is a non-trivial task because, to our knowledge, no existing neural network architecture can be used in this way. For instance, classic recurrent neural networks (Elman, 1990) unfold to left-branching trees, and are not able to process arbitrarily shaped parse trees that the counting approaches are applied to. Recursive neural networks (Socher et al., 2010) and extensions (Socher et al., 2012; Le et al., 2013), on the other hand, do work with trees of arbitrary shape, but process them in a bottom-up manner. The probabilities we need to estimate are, in contrast, defined by top-down generative models, or by models that require information flows in both directions (e.g., the probability of generating a node depends on the whole fragment rooted at its just-generated sis-
dependency parsing. The generation process is top-down: starting at the ROOT, it generates left dependents and then right dependents for the ROOT. After that, it generates left dependents and right dependents for each of ROOT’s dependents. The process recursively continues until there is no further dependent to generate. The whole process is captured in the following formula

\[
P(T(H)) = \prod_{l=1}^{L} P\left(H^L_l \mid C^L_{H^L_l}\right) P\left(T(H^L_l)\right) \times \prod_{r=1}^{R} P\left(H^R_r \mid C^R_{H^R_r}\right) P\left(T(H^R_r)\right)
\]

where \(H\) is the current head, \(T(N)\) is the fragment of the dependency parse rooted in \(N\), and \(C^*_N\) is the context in which \(N\) is generated. \(H^L, H^R\) are respectively \(H\)'s left dependents and right dependents, plus \(EOC\) (End-Of-Children), a special token to indicate that there are no more dependents to generate. Thus, \(P(T(\text{ROOT}))\) is the probability of generating the entire dependency structure \(T\). We refer to \(\langle H^L_l, C^L_{H^L_l}\rangle, \langle H^R_r, C^R_{H^R_r}\rangle\) as “events”, and \(\langle C^L_{H^L_l}\rangle, \langle C^R_{H^R_r}\rangle\) as “conditioning contexts”.

In order to avoid the problem of data sparsity, the conditioning context in which a dependent \(D\) is generated should capture only part of the fragment generated so far. Based on the amount of information that contexts hold, we can define the order of a generative model (see Hayashi et al. (2011, Table 3) for examples)

- first-order: \(C^1_D\) contains the head \(H\),
- second-order: \(C^2_D\) contains \(H\) and the just-generated sibling \(S\),
- third-order: \(C^3_D\) contains \(H, S\), the sibling \(S'\) before \(S\) (tri-sibling); or \(H, S\) and the grand-head \(G\) (the head of \(H\)) (grand-sibling) (the fragment enclosed in the blue dotted contour in Figure 2),
- \(\infty\)-order: \(C^\infty_D\) contains all of \(D\)'s ancestors, theirs siblings, and its generated siblings (the fragment enclosed in the red dashed contour in Figure 2).

In the original models (Eisner, 1996a), each dependent \(D\) is a 4-tuple \(\langle \text{dist}, w, c, t \rangle\)

- \(\text{dist}(H, D)\) the distance between \(D\) and its head \(H\), represented as one of the four ranges 1, 2, 3-6, 7-∞.
Figure 2: Example of different orders of context of “diversified”. The blue dotted shape corresponds to the third-order outward context, while the red dashed shape corresponds to the ∞-order left-to-right context. The green dot-dashed shape corresponds to the context to compute the outer representation.

- word(D) the lowercase version of the word of D,
- cap(D) the capitalisation feature of the word of D (all letters are lowercase, all letters are uppercase, the first letter is uppercase, the first letter is lowercase),
- tag(D) the POS-tag of D,

Here, to make the dependency complete, deprel(D), the dependency relation of D (e.g., SBJ, DEP), is also taken into account.

3 Third-order Model with Counting

The third-order model we suggest is similar to the grandsibling model proposed by Sangati et al. (2009) and Hayashi et al. (2011). It defines the probability of generating a dependent D = (dist, d, w, c, t) as the product of the distance-based probability and the probabilities of generating each of its components (d, t, w, c, denoting dependency relation, POS-tag, word and capitalisation feature, respectively). Each of these probabilities is smoothed using back-off according to the given reduction lists (as explained below).

\[
P(D|\mathcal{C}_D) = P(dist(H, D), dwct(D)|H, S, G, dir) \times P(D|\mathcal{D})
\]

\[
= P(dist(H, D), dwct(D)|H, S, G, dir) \times P(t(D)|d(D), H, S, G, dir)
\]

\[
\text{reduction list: } \begin{align*}
& \{ tw(H), tw(S), tw(G), dir \\
& tw(H), tw(S), t(G), dir \\
& t(H), tw(S), t(G), dir \\
& t(H), t(S), t(G), dir
\} \\
\end{align*}
\]

\[
\times P(t(D)|d(D), H, S, G, dir)
\]

\[
\text{reduction list: } \begin{align*}
& \{ d(D), dw(H), t(S), dir \\
& d(D), d(H), t(S), dir \\
& d(D), d(D), dir
\} \\
\end{align*}
\]

The reason for generating the dependency relation first is based on the similarity between relation/dependent and role/filler: we generate a role and then choose a filler for that role.

Back-off The back-off parameters are identical to Eisner (1996b). To estimate the probability \( P(A|\text{context}) \) given a reduction list \( L = (l_1, l_2, ..., l_n) \) of context, let

\[
P_i = \begin{cases} 
\text{count}(A, l_i) + 0.005 & \text{if } i = n \\
\text{count}(A, l_i) + 0.5 & \text{count}(A, l_{i+1}) + 3p_{i+1} & \text{otherwise}
\end{cases}
\]

then \( P(A|\text{context}) = p_1 \).

4 The Inside-Outside Recursive Neural Network

In this section, we first describe the Recursive Neural Network architecture of Socher et al. (2010) and then propose an extension we call the Inside-Outside Recursive Neural Network (IORNN). The IORNN is a general architecture for trees, which works with tree-based generative models including those employed by Eisner (1996b) and Collins (2003). We then explain how to apply the IORNN to the ∞-order model. Note that for the present paper we are only concerned with the problem of computing the probability of...
a tree; we assume an independently given parser is available to assign a syntactic structure, or multiple candidate structures, to an input string.

4.1 Recursive Neural Network

The architecture we propose can best be understood as an extension of the Recursive Neural Networks (RNNs) proposed by Socher et al. (2010), that we mentioned above. In order to see how an RNN works, consider the following example. Assume that there is a constituent with parse tree (p2 (p1 x y) z) (Figure 3), and that x, y, z ∈ ℝn are the (inner) representations of the three words x, y and z, respectively. We use a neural network which consists of a weight matrix W1 ∈ ℝn×n for left children and a weight matrix W2 ∈ ℝn×n for right children to compute the vector for a parent node in a bottom up manner. Thus, we compute p1 as follows

\[ p_1 = f(W_1 x + W_2 y + b) \]

where b is a bias vector and f is an activation function (e.g., tanh or logistic). Having computed p1, we can then move one level up in the hierarchy and compute p2:

\[ p_2 = f(W_1 p_1 + W_2 z + b) \]

This process is continued until we reach the root node. The RNN thus computes a single vector for each node p in the tree, representing the content under that node. It has in common with logical semantics that representations for compounds (here xyz) are computed by recursively applying a composition function to meaning representations of the parts. It is difficult to characterise the expressivity of the resulting system in logical terms, but recent work suggests it is surprisingly powerful (e.g., Kanerva (2009)).

4.2 IORNN

We extend the RNN-architecture by adding a second vector to each node, representing the context of the node, shown as white rectangles in Figure 4. The job of this second vector, the outer representation, is to summarize all information about the context of node p so that we can either predict its content (i.e., predict an inner representation), or pass on this information to the daughters of p (i.e., compute outer representations of these daughters). Outer representations thus allow information to flow top-down.

We explain the operation of the resulting Inside-Outside Recursive Neural Network in terms of the same example parse tree (p2 (p1 x y) z) (see Figure 4). Each node u in the syntactic tree carries two vectors ou and iu, the outer representation and inner representation of the constituent that is covered by the node.

Computing inner representations Inner representations are computed from the bottom up. We assume for every word w an inner representation iw ∈ ℝn. The inner representation of a non-terminal node, say p1, is given by

\[ i_{p_1} = f(W_1^i i_x + W_2^i i_y + b^i) \]

where W1i, W2i are n × n real matrices, bi is a bias vector, and f is an activation function, e.g. tanh. (This is the same as the computation of non-terminal vectors in the RNNs.) The inner representation of a parent node is thus a function of the inner representations of its children.

Computing outer representations Outer representations are computed from the top down. For a node which is not the root, say p1, the outer repre-
sentation is given by
\[ o_p = g(W^o_1 o_p + W^o_2 i_x + b^o) \]
where \( W^o_1, W^o_2 \) are \( n \times n \) real matrices, \( b^o \) is a bias vector, and \( g \) is an activation function. The outer representation of a node is thus a function of the outer representation of its parent and the inner representation of its sisters.

If there is information about the external context of the utterance that is being processed, this information determines the outer representation of the root node \( o_{root} \). In our first experiments reported here, no such information was assumed to be available. In this case, a random value \( o_\emptyset \) is chosen at initialisation and assigned to the root nodes of all utterances; this value is then adjusted by the learning process discussed below.

Training Training the IORNN is to minimise an objective function \( J(\theta) \) which depends on the purpose of usage where \( \theta \) is the set of parameters. To do so, we compute the gradient \( \partial J / \partial \theta \) and apply the gradient descent method. The gradient is effectively computed thanks to back-propagation through structure (Goller and Kühler, 1996). Following Socher et al. (2013), we use AdaGrad (Duchi et al., 2011) to update the parameters.

4.3 The \( \infty \)-order Model with IORNN
The RNN and IORNN are defined for context-free trees. To apply the IORNN architecture to dependency parses we need to adapt the definitions somewhat. In particular, in the generative dependency model, every step in the generative story involves the decision to generate a specific word while the span of the subtree that this word will dominate only becomes clear when all dependents are generated. We therefore introduce partial outer representation as a representation of the current context of a word in the generative process, and compute the final outer representation only when all its siblings have been generated.

Consider an example of head \( h \) and its dependents \( x, y \) (we ignore directions for simplicity) in Figure 5. Assume that we are in the state in the generative process where the generation of \( h \) is complete, i.e. we know its inner and outer representations \( i_h \) and \( o_h \). Now, when generating \( h \)'s first dependent \( x \) (see Figure 5-a), we first compute \( x \)'s partial outer representation (representing its context at this stage in the process), which is a function of the outer representation of the head

\[ o_1 = f(W_h i_h + W_o o_h + b_o) \]
where \( W_h, W_o \) are \( n \times n \) real matrices, \( b_o \) is a bias vector, \( f \) is an activation function.

With the context of the first dependent determined, we can proceed and generate its content. For this purpose, we assume a separate weight matrix \( W \), trained (as explained below) to predict a specific word given a (partial) outer representation. To compute a proper probability for word \( x \), we use the softmax function:

\[ \text{softmax}(x, o_1) = \frac{e^{u(x, o_1)}}{\sum_{w\in V} e^{u(w, o_1)}} \]

where \( [u(w_1, o_1), \ldots, u(w_{|V|}, o_1)]^T = W o_1 + b \) and \( V \) is the set of all possible dependents.

Note that since \( o_h \), the outer representation of \( h \), represents the entire dependency structure generated up to that point, \( o_1 \) is a vectorial representation of the \( \infty \)-order context generating the first dependent (like the fragment enclosed in the red dashed contour in Figure 2). The softmax function thus estimates the probability \( P(D = x|C_0^D) \).

The next step, now that \( x \) is generated, is to compute the partial outer representation for the second dependent (see Figure 5-b)

\[ o_2 = f(W_h i_h + W_o o_h + W_{dr(x)} i_x + b_o) \]
where \( W_{dr(x)} \) is a \( n \times n \) real matrix specific for the dependency relation of \( x \) with \( h \).

Next \( y \) is generated (using the softmax function above), and the partial outer representation for the third dependent (see Figure 5-c) is computed:

\[ o_3 = f(W_h i_h + W_o o_h + \frac{1}{2} (W_{dr(x)} i_x + W_{dr(y)} i_y) + b_o) \]

Since the third dependent is the End-of-Children symbol (EOC), the process of generating dependents for \( h \) stops. We can then return to \( x \) and \( y \) to replace the partial outer representations with complete outer representations\(^1\) (see

\(^1\) According to the IORNN architecture, to compute the outer representation of a node, the inner representations of the whole fragments rooting at its sisters must be taken into account. Here, we replace the inner representation of a fragment by the inner representation of its root since the meaning of a phrase is often dominated by the meaning of its head.
Figure 5: Example of applying IORNN to dependency parsing. Black, grey, white boxes are respectively inner, partial outer, and outer representations. For simplicity, only links related to the current computation are drawn (see text).

Figure 5-d,e:
\[
\begin{align*}
o_x &= f(W_{h_i}i_h + W_{ho}o_h + W_{dr(y)}i_y + b_o) \\
o_y &= f(W_{h_i}i_h + W_{ho}o_h + W_{dr(x)}i_x + b_o)
\end{align*}
\]
In general, if \( u \) is the first dependent of \( h \) then
\[
o_u = f(W_{h_i}i_h + W_{ho}o_h + b_o)
\]
otherwise
\[
o_u = f(W_{h_i}i_h + W_{ho}o_h + b_o + \frac{1}{|S(u)|} \sum_{v \in S(u)} W_{dr(v)}i_v)
\]
where \( S(u) \) is the set of \( u \)'s sisters generated before it. And, if \( u \) is the only dependent of \( h \) (ignoring EOC) then
\[
o_u = f(W_{h_i}i_h + W_{ho}o_h + b_o)
\]
otherwise
\[
o_u = f(W_{h_i}i_h + W_{ho}o_h + b_o + \frac{1}{|S(u)|} \sum_{v \in S(u)} W_{dr(v)}i_v)
\]
where \( S(u) \) is the set of \( u \)'s sisters.

We then continue this process to generate dependents for \( x \) and \( y \) until the process stops.

**Inner Representations** In the calculation of the probability of generating a word, described above, we assumed inner representations of all possible words to be given. These are, in fact, themselves a function of vector representations for the words (in our case, the word vectors are initially borrowed from Collobert et al. (2011)), the POS-tags and capitalisation features. That is, the inner representation at a node \( h \) is given by:
\[
i_h = f(W_wW_h + W_pP_h + W_cC_h)
\]
where \( W_w \in \mathbb{R}^{n \times d_w}, W_p \in \mathbb{R}^{n \times d_p}, W_c \in \mathbb{R}^{n \times d_c} \), \( w_h \) is the word vector of \( h \), and \( p_h, c_h \) are respectively binary vectors representing the POS-tag and capitalisation feature of \( h \).

**Training** Training this IORNN is to minimise the following objective function which is the regularised cross-entropy
\[
J(\theta) = -\frac{1}{m} \sum_{T \in D} \sum_{w \in T} \log(P(w|\bar{o}_w)) + \frac{1}{2} (\lambda_W \|\theta_W\|^2 + \lambda_L \|\theta_L\|^2)
\]
where \( D \) is the set of training dependency parses, \( m \) is the number of dependents; \( \theta_W, \theta_L \) are the weight matrix set and the word embeddings \((\theta = (\theta_W, \theta_L))\); \( \lambda_W, \lambda_L \) are regularisation hyper-parameters.

**Implementation** We decompose a dependent \( D \) into four features: dependency relation, POS-tag, lowercase version of word, capitalisation feature of word. We then factorise \( P(D|C_D) \) similarly to Section 3, where each component is estimated by a softmax function.

5 Experiments

In our experiments, we convert the Penn Treebank to dependencies using the Universal dependency annotation (McDonald et al., 2013); this yields a dependency tree corpus we label PTB-U. In order to compare with other systems, we also experiment with an alternative conversion using the head rules of Yamada and Matsumoto (2003); this yields a dependency tree corpus we label PTB-YM. Sections 2-21 are used for training, section 22 for development, and section 23 for testing. For the PTB-U, the gold POS-tags are used. For the PTB-YM, the development and test sets are tagged by the Stanford POS-tagger trained on the whole
Table 1: Perplexities of the two models on PTB-U-22.

<table>
<thead>
<tr>
<th>Perplexity</th>
<th>3rd-order model</th>
<th>$\infty$-order model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1736.73</td>
<td>236.58</td>
</tr>
</tbody>
</table>

Table 2: Percentages of events extracted from PTB-U-22 appearing more than twice in the training set. Events are grouped according to the reduction lists in Equation 2. $d, t, w, c$ stand for dependency relation, POS-tag, word, and capitalisation feature.

<table>
<thead>
<tr>
<th>back-off level</th>
<th>d</th>
<th>t</th>
<th>w</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>47.4</td>
<td>61.6</td>
<td>43.7</td>
<td>87.7</td>
</tr>
<tr>
<td>1</td>
<td>69.8</td>
<td>98.4</td>
<td>77.8</td>
<td>97.3</td>
</tr>
<tr>
<td>2</td>
<td>76.0, 89.5</td>
<td>99.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>97.9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>76.1</td>
<td>86.6</td>
<td>60.7</td>
<td>92.5</td>
</tr>
</tbody>
</table>

Table 1 shows results. The perplexity of the third-order model is more than seven times higher than the $\infty$-order model, which is identical to Sangati et al. (2009). Moreover, as in many mixture-model-based approaches, we define the mixture reranker as a combination of the generative model and the MST discriminative model (Hayashi et al., 2011)

$$T^* = \arg \max_{T \in \mathcal{D}(S)} \alpha \log P(T(\text{ROOT})) + (1-\alpha)s(S, T)$$

where $s(S, T)$ is the score given by the MSTParser, and $\alpha \in [0, 1]$.

Results Figure 6 shows UASs of the generative reranker on the development set. The MSTParser achieves 92.32% and the Oracle achieve 96.23% when $k = 10$. With the third-order model, the generative reranker performs better than the MSTParser when $k < 6$ and the maximum improvement is 0.17%. Meanwhile, with the $\infty$-order model, the generative reranker always gains higher UASs than the MSTParser, and with $k = 6$, the difference reaches 0.7%. Figure 7 shows UASs of the mixture reranker on the same set. $\alpha$ is optimised by searching with the step-size 0.005. Unsurprisingly, we observe improvements over the
We first compare the mixture reranker using the \( \infty \)-order model against the state-of-the-art dependency parser TurboParser (with the full mode) (Martins et al., 2013) on PTB-U-23. Table 4 shows LASs and UASs. When taking labels into account, the TurboParser outperforms the reranker. But without counting labels, the two systems perform comparably, and when ignoring punctuation the reranker even outperforms the TurboParser. This pattern is also observed when the exact match metrics are used (see Table 4). This is due to the fact that the TurboParser performs significantly better than the MSTParser, which generates \( k \)-best lists for the reranker, in labelling: the former achieves 96.03% label accuracy score whereas the latter achieves 94.92%.

One remarkable point is that reranking with the \( \infty \)-order model helps to improve the exact match scores 4% - 6.4% (see Table 4). Because the exact match scores correlate with the ability to handle global structures, we conclude that the IORNN is able to capture \( \infty \)-order contexts. Figure 8 shows distributions of correct-head accuracy over CPOS-tags and Figure 9 shows F1-scores of binned HEAD distance. Reranking with the \( \infty \)-order model is clearly helpful for all CPOS-tags and dependent-to-head distances, except a minor decrease on PRT.

We compare the reranker against other systems on PTB-YM-23 using the UAS metric ignoring punctuation (as the standard evaluation for English) (see Table 5). Our system performs slightly better than many state-of-the-art systems such as Martins et al. (2013) (a.k.a. TurboParser), Zhang and McDonald (2012), Koo and Collins (2010). It outperforms Hayashi et al. (2011) which is a reranker using a combination of third-order generative models with a variational model learnt
<table>
<thead>
<tr>
<th>System</th>
<th>LAS (w/o punc)</th>
<th>UAS (w/o punc)</th>
<th>LEM (w/o punc)</th>
<th>UEM (w/o punc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSTParser</td>
<td>89.97 (90.54)</td>
<td>91.99 (92.82)</td>
<td>32.37 (34.19)</td>
<td>42.80 (45.24)</td>
</tr>
<tr>
<td>w. $\infty$-order ($k = 9$)</td>
<td>91.02 (91.51)</td>
<td>93.08 (93.84)</td>
<td>37.58 (39.16)</td>
<td>49.17 (51.53)</td>
</tr>
<tr>
<td>TurboParser</td>
<td>91.56 (92.02)</td>
<td>93.05 (93.70)</td>
<td>40.65 (41.72)</td>
<td>48.05 (49.83)</td>
</tr>
</tbody>
</table>

Table 4: Comparison with the TurboParser on PTB-U-23. LEM and UEM are respectively the labelled exact match score and unlabelled exact match score metrics. The numbers in brackets are scores computed excluding punctuation.

Figure 8: Distributions of correct-head accuracy over CPOS-tags (PTB-U-23).

<table>
<thead>
<tr>
<th>System</th>
<th>UAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Huang and Sagae (2010)</td>
<td>92.1</td>
</tr>
<tr>
<td>Koo and Collins (2010)</td>
<td>93.04</td>
</tr>
<tr>
<td>Zhang and McDonald (2012)</td>
<td>93.06</td>
</tr>
<tr>
<td>Martins et al. (2013)</td>
<td>93.07</td>
</tr>
<tr>
<td>Bohnet and Kuhn (2012)</td>
<td>93.39</td>
</tr>
<tr>
<td>Reranking</td>
<td></td>
</tr>
<tr>
<td>Hayashi et al. (2011)</td>
<td>92.89</td>
</tr>
<tr>
<td>Hayashi et al. (2013)</td>
<td>93.12</td>
</tr>
<tr>
<td>MST+$\infty$-order ($k = 12$)</td>
<td>93.12</td>
</tr>
</tbody>
</table>

Table 5: Comparison with other systems on PTB-YM-23 (excluding punctuation).

<table>
<thead>
<tr>
<th>System</th>
<th>UAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Huang and Sagae (2010)</td>
<td>92.1</td>
</tr>
<tr>
<td>Koo and Collins (2010)</td>
<td>93.04</td>
</tr>
<tr>
<td>Zhang and McDonald (2012)</td>
<td>93.06</td>
</tr>
<tr>
<td>Martins et al. (2013)</td>
<td>93.07</td>
</tr>
<tr>
<td>Bohnet and Kuhn (2012)</td>
<td>93.39</td>
</tr>
</tbody>
</table>

6 Related Work

Using neural networks to process trees was first proposed by Pollack (1990) in the Recursive Autoassociative Memory model which was used for unsupervised learning. Socher et al. (2010) later introduced the Recursive Neural Network architecture for supervised learning tasks such as syntactic parsing and sentiment analysis (Socher et al., 2013). Our IORNN is an extension of the RNN: the former can process trees not only bottom-up like the latter but also top-down.

Elman (1990) invented the simple recurrent neural network (SRNN) architecture which is capable of capturing very long histories. Mikolov et al. (2010) then applied it to language modelling and gained state-of-the-art results, outperforming the the standard $n$-gram techniques such as Kneser-Ney smoothed 5-gram. Our IORNN architecture for dependency parsing bears a resemblance to the SRNN in the sense that it can also capture long ‘histories’ in context representations (i.e., outer representations in our terminology). Moreover, the IORNN can be seen as a generalization of the SRNN since a left-branching tree is equivalent to a chain and vice versa.

The idea of letting parsing decisions depend on arbitrarily long derivation histories is also explored in Borensztajn and Zuidema (2011) and is related to parsing frameworks that allow arbitrarily large elementary trees (e.g., Scha (1990), O’Donnell et al. (2009), Sangati and Zuidema (2011), and van Cranenburgh and Bod (2013)).

Titov and Henderson (2007) were the first proposing to use deep networks for dependency parsing. They introduced a transition-based generative dependency model using incremental sigmoid belief networks and applied beam pruning for searching best trees. Differing from them, our work uses the IORNN architecture to rescore $k$-best candidates generated by an independent
graph-based parser, namely the MSTParser.

Reranking $k$-best lists was introduced by Collins and Koo (2005) and Charniak and Johnson (2005). Their rerankers are discriminative and for constituent parsing. Sangati et al. (2009) proposed to use a third-order generative model for reranking $k$-best lists of dependency parses. Hayashi et al. (2011) then followed this idea but combined generative models with a variational model learnt on the fly to rerank forests. In this paper, we also followed Sangati et al. (2009)’s idea but used an $\infty$-order generative model, which has never been used before.

7 Conclusion

In this paper, we proposed a new neural network architecture, the Inside-Outside Recursive Neural Network, that can process trees both bottom-up and top-down. The key idea is to extend the RNN such that every node in the tree has two vectors associated with it: an inner representation for its content, and an outer representation for its context. Inner and outer representations of any constituent can be computed simultaneously and interact with each other. This way, information can flow top-down, bottom-up, inward and outward. Thanks to this property, by applying the IORNN to dependency parses, we have shown that using an $\infty$-order generative model for dependency parsing, which has never been done before, is practical.

Our experimental results on the English section of the Universal Dependency Treebanks show that the $\infty$-order generative model approximates the true dependency distribution better than the traditional third-order model using counting, and tends to choose more accurate parses in $k$-best lists. In addition, reranking with this model even outperforms the state-of-the-art TurboParser on unlabelled score metrics.

Our source code is available at: github.com/lephantong/iornn-depparse.

Acknowledgments

We thank Remko Scha and three anonymous reviewers for helpful comments.

References


Christoph Goller and Andreas Küchler. 1996. Learning task-dependent distributed representations by backpropagation through structure. In International Conference on Neural Networks. IEEE.

Katsuhiko Hayashi, Taro Watanabe, Masayuki Asahara, and Yuji Matsumoto. 2011. Third-order variational reranking on packed-shared dependency


