What is inside Distributed Representations? Encoding and Decoding Structures in Vectors without Learning

Lorenzo Ferrone[a,1], Fabio Massimo Zanzotto[a,1], Xavier Carreras[b,1]

[a] *Università degli studi di Roma Tor Vergata*
Via del Politecnico 1, 00133 Roma, Italia
[b] *Xerox Research Centre Europe*
6 chemin de Maupertuis 38240 Meylan, France

Abstract

Encoding linguistic information in *distributed representations* is a strong and promising trend in Natural Language Processing. In fact, these distributed representations are vectors or tensors which are successfully used in learning over linguistic data. Distributed representations are linguistic representations ready to be used as feature vectors. However, this success starts to pose legitimate questions: what is exactly *concealed* in these distributed representations? And more importantly, is it possible to *decode* distributed representations to recover back the encoded linguistic information?

In this paper, we propose *decoders* to invert very complex encoding functions: the distributed tree encoders [1] which embed trees in low dimensional vectors without learning. As these functions are not invertible per se, we treated this problem as a parsing problem. Hence, we propose two classes of decoders based on parsing algorithms: one for binary trees based on CYK and one for general trees based on CYK+. Experiments show that, with a sufficient dimensionality, the encoding-decoding process has a high accuracy.

1. Introduction

Encoding linguistic information in vectors, matrices, or high order tensors is a strong and promising trend in Natural Language Processing. Dictionaries are replaced by collections of distributional vectors [2] or word embeddings [3]. The meaning of sentences is represented in vectors or in tensors computed by compositional distributional semantic models [4, 5, 6, 7, 8] or by deep learning models such as recursive neural networks [9, 10, 11]. And, finally, syntactic structures have been encoded in distributed representations without learning

Email addresses: lorenzo.ferrone@gmail.com (Lorenzo Ferrone), fabio.massimo.zanzotto@uniroma2.it (Fabio Massimo Zanzotto), xavier.carreras@xrce.xerox.com (Xavier Carreras)

[12]. Hence, components of these vectors or tensors may encode the *essence of linguistic information*: semantic properties of words [13] as well as structural information for sentences.

Looking at linguistic information as vectors is very attractive when learning functions over linguistic data. In fact, these vectors capture the essence of linguistic information and can be directly used as inputs in downstream learning functions, thus, avoiding the need of feature engineering to map a discrete structure to a vector of features. Hence, in this way, Long-Short Term Memories (LSTMs) are obtaining astonishing results in many high level semantic tasks [14, 15] and support vector machines [16] can start to exploit structural information in very large datasets [12, 17]. In fact, vectors encoding syntactic structures are sufficient to approximate symbolic convolution kernels in kernel machines [12].

However, a legitimate question arises: what is exactly concealed in these vectors representing encoded linguistic information? And more importantly, is it possible to decode these vectors to recover back the linguistic information? As in the first surge of deep learning methods and distributed representations, decoding vectors to extract back linguistic information is becoming an important issue. Distributed representations have been originally designed to store information [18]. Hence, it should be possible to encode information as well as decode it back.

In this paper, we propose decoders to invert very complex encoding functions: the distributed tree encoders [1] which embed trees in low dimensional vectors without learning. As these functions are not invertible per se, we treated this problem as a parsing problem. Hence, we propose two classes of decoders based on parsing algorithms: one for binary trees and one for general trees. We introduced the first decoder based on the CYK algorithm in [19]. In this paper, we generalize the approach to general trees in two new ways: (1) by using a binarizer and a debinarizer along with the first decoder; and, (2) by adapting CYK+ [20, 21] that is an algorithm for parsing general context free grammars in a probabilistic setting. Experiments show that, with a sufficient dimensionality, the encoding-decoding process has a high accuracy.

The rest of the paper is organized as follows. Section 2 gives a brief background on encoding and decoding symbolic information in vectors and tensors. Section 3 introduces the *Distributed Trees* as our encoder for trees in vectors. Section 4 revises the model for binary trees that is based on the CYK parsing algorithm which is used for Chomsky Normal Form probabilistic grammars and describes two new models to deal with general trees, one based on the CYK algorithm followed by a process of debinarization, and one based on the CYK+ algorithm. Finally, Section 5 presents an experimental investigation of these decoders, and Section 6 then concludes with a few remarks and plans on possible directions for future work.

2. Related Work

Representing observations in vectors or tensors is a classic idea in machine learning. In fact, before learning or before applying learned functions, observa-

tions are mapped in vector or tensor spaces, often called *feature spaces*. In some cases, such as kernel machines [16], the mapping is implicit but nonetheless the learning machine operates in such vector space.

In neural network approaches, and in deep learning, this way of representing observations is even more evident and these vectors or tensors take the name of distributed representations [22]. The important fact is that functions mapping observations to distributed representations are generally learned from data. This research line is called *representation learning* [23]. This aspect is different from the distributed tree encoders we use in this article, which are not learned.

Yet, there is a general need to understand what is in these distributed representations as they appear to be mysterious. Generally, distributed representations stored in neural networks are somehow represented with pictures. This results in a very clear explanation when these networks represent chunks of images (for example, [24, 25]). Yet, they are a little bit more obscure when linguistic phenomena are represented. For example, in [15], the effect of the network on the observation is depicted as the degree of correlation between word pairs. This is interesting but is not extremely evocative.

Hence, there is a very interesting line of research that investigates ways to invert the encoding process. Auto-encoders [26] are a first noticeable example on the importance of demonstrating that distributed representations store information in low-dimensional spaces. In fact, auto-encoders are simple neural networks whose objective function is to maximize the ability of the network to reproduce a given input as output. Yet, auto-encoders are generally encode and decode symbols as single units. For example, $eat(John, apple)$ is treated as a single symbol even if it is made up of parts.

Pollack [9] proposed recursive auto-encoders applied to trees, which make use of encoding operations in a recursive way. These were defined along a reconstruction (or decoding) component, that maps the resulting vectors back to their structure. This reconstruction component is used as part of the learning objective, and assumes that the structure of the tree is available: thus it reconstructs the identities of the nodes in the tree. In contrast, in this article, we present decoding methods for distributed tree representations that do not assume access to the structure of the tree. In other words, we parse distributed vectors. More recently Socher *et al.* have also proposed recursive auto-encoders for parse trees [10], but they do not focus in recovering trees out of the encoded vectors.

Holographic distributed representations instead aim to encode flat structures. In general, these representations have been used to encode logical propositions such as the above $eat(John, apple)$. In this case, each part has an associated vector and the vector for the compound is obtained by combining these vectors. The major concern here is to build encoding functions that can be decoded, that is, having a representation for $eat(John, apple)$ it is possible to retrieve the parts by inverting the encoding function.

In these holographic representations, parts are represented as vectors in $N(0, \frac{1}{d}I_d)$ and the composition function is called circular convolution $\otimes$, that is a function derived from signal processing. Given two vectors $\mathbf{a}$ and $\mathbf{b}$, circular

convolution is defined as follows:

$$z_j = (\mathbf{a} \otimes \mathbf{b})_j = \sum_{k=0}^{n-1} a_k b_{j-k}$$

where subscripts are modulo n. A way to approximate the inverse of $\otimes$ is circular correlation $\oplus$ defined as follows:

$$c_j = (\mathbf{z} \oplus \mathbf{b})_j = \sum_{k=0}^{n-1} z_k b_{j+k}$$

where again subscripts are modulo n. In the decoding with $\oplus$, parts of the structures can be derived in an approximated way, that is:

$$(\mathbf{a} \otimes \mathbf{b}) \oplus \mathbf{b} \approx \mathbf{a}$$

For example, having the vectors $\mathbf{e}$, $\mathbf{J}$, and $\mathbf{a}$ for *eat*, *John* and *apple*, respectively, the following encoding and decoding produces a vector that approximates the original vector for *John*:

$$\mathbf{J} \approx (\mathbf{J} \otimes \mathbf{e} \otimes \mathbf{a}) \oplus (\mathbf{e} \otimes \mathbf{a})$$

Holographic representations have severe limitations as it is possible to encode and decode only simple flat structures. These representations are based on the circular convolution function which is commutative. Hence, parts in larger structures risk to be confused as the order is not taken into consideration.

Distributed trees [1] have shown that the principles expressed in holographic representation can be applied to encode larger structures. These distributed trees are encoding functions that transform trees into low-dimensional vectors containing the encoding of the substructures of the tree. Thus, these distributed trees are particularly attractive as they can be used to represent structures in linear learning machines which are computationally efficient.

However, distributed trees are not easily invertible.

3. Encoding Structures with Distributed Trees

Distributed Trees (DT) [1] are dense, low-dimensional vectors of real numbers which embed structural information of trees, specifically they embed the subtrees of a given tree. No learning is required to define distributed trees. These vectors have been introduced to reduce the computational complexity of learning with tree kernels [27]. Yet, this technique is extremely important for its representation power: an highly expressive embedding without any need of learning.

Encoders for DTs are functions that map trees to vectors: each type of tree kernel (TK) [28, 29, 30, 31] defines a particular decomposition of a tree into subtrees, and thus each has its associated encoding function into distributed trees An encoding function takes the following form:

$$\begin{aligned} \mathrm{DT}\colon T &\rightarrow \mathbb{R}^d \\ t &\mapsto \mathrm{DT}(t) = \mathbf{t} \end{aligned}$$

such that:

$$\langle \mathrm{DT}(t_1), \mathrm{DT}(t_2) \rangle \approx TK(t_1, t_2) \tag{1}$$

where $t \in T$ is a tree, $\langle \cdot, \cdot \rangle$ indicates the standard inner product in $\mathbb{R}^d$ and $TK(\cdot, \cdot)$ represents the original tree kernel.

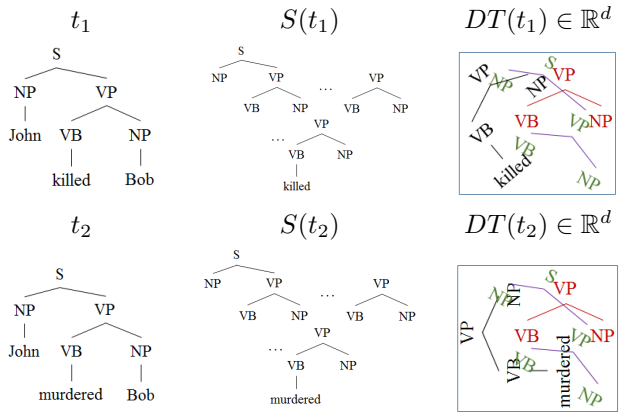


Figure 1: The idea behind Distributed Trees

The intuitive idea of why these encoders approximate tree kernels is depicted in Figure 1. Given two trees, t_1 and t_2, and their respective sets of subtrees $S(t_1)$ and $S(t_2)$, a tree kernel $TK(t_1, t_2)$ performs a weighted count of common subtrees, that is, trees in $S(t_1) \cap S(t_2)$. Encoders of Distributed Trees pack sets $S(t_1)$ and $S(t_2)$ in small vectors. In the illustration of Figure 1, this idea is conveyed by packing images of subtrees in small spaces, that is, the boxes under $DT(t_1)$ and $DT(t_2)$. By rotating and coloring subtrees, pictures in the boxes still allow to recognize these subtrees. Consequently, it seems to be possible to count how many subtrees are similar by comparing the picture in the box under $DT(t_1)$ with the one under $DT(t_2)$. In Distributed Trees, boxes correspond to vectors of fixed dimensionality, and sets of subtrees are represented in these vectors.

More formally, encoders for distributed trees embed the space of the tree fragments in a smaller space by defining the set of *distributed tree fragments*. In fact, it is possible to represent subtrees $\tau \in S(t)$ of a given tree t in distributed tree fragments $\mathrm{DTF}(\tau) \in \mathbb{R}^d$ such that:

$$\langle \mathrm{DTF}(\tau_1), \mathrm{DTF}(\tau_2) \rangle \approx \delta(\tau_1, \tau_2) \tag{2}$$

where δ is the Kronecker's delta function. Hence, distributed tree fragments approximate the orthonormal base of the corresponding tree kernels in the smaller space $\mathbb{R}^d$. Consequently, distributed trees are sums of distributed tree fragments of trees, that is:

$$\mathrm{DT}(t) = \sum_{\tau \in S(t)} \sqrt{\lambda}^{|\mathcal{N}(\tau)|} \mathrm{DTF}(\tau) \tag{3}$$

where λ is the standard decaying factor in tree kernels and $\mathcal{N}(\tau)$ is the set of the nodes of the subtree τ. With this definition, the property in Eq. 1 holds [1].

Distributed tree fragments (DTF) carry a lot of interesting structural information as they are defined recursively.

Let $\mathcal{N}$ be the set of node labels in trees. To each node label $n \in \mathcal{N}$ we associate a random vector $\mathbf{n}$ drawn from the d-dimensional hypersphere. Given a tree fragment τ, let $\mathcal{N}(\tau)$ be the set of nodes occurring in τ. The encoding function for τ will pack the vectors associated with $\mathcal{N}(\tau)$.

Random vectors of high dimensionality have the property of being quasi-orthonormal (that is, they obey a relationship similar to 2). Then, we recursively combine these random vectors with the shuffled circular convolution operation[1], which has the property of preserving quasi-orthonormality between vectors. DTF are recursively defined as follows:

$$\mathrm{DTF}(\tau) = \begin{cases} \mathbf{n} \text{ if } \tau \text{ is a terminal node } n \\ \mathbf{n} \odot [\bigodot_i DTF(\tau_i)] \text{ otherwise} \end{cases}$$

where n is the head node of the tree fragment τ, $\mathbf{n}$ is the vector representation for that node, and τ_i are the direct children of n in τ.

The recursive definition of the distributed tree fragments allows to introduce a fast and linear algorithm to compute distributed trees. First, we will slightly abuse notation in the following way. We will assume a fixed tree t and use n to refer both to a node of t and to the subtree of t headed by n itself. In accordance, we will use n_i to denote one subtree that is direct children of n in t, and the bold versions $\mathbf{n}$ and $\mathbf{n}_i$ will denote the vectors associated with the corresponding node labels of such subtrees. Let us define a recursive function $\mathrm{SN}(n)$ that collects all the distributed tree fragments τ of t that start at n, computes the vector representation $\mathrm{DTF}(\tau)$ of each τ, and aggregates such vectors. The function is as follows:

$$\mathrm{SN}(n) = \begin{cases} \mathbf{0} \text{ if } n \text{ is terminal} \\ \mathbf{n} \odot \bigodot_i \sqrt{\lambda}\,[\mathbf{n_i} + SN(n_i)] \text{ otherwise} \end{cases} \tag{4}$$

where n_i are the direct children of n in the tree t. With this recursive function, the distributed tree encoder $\mathrm{DT}(t)$ defined in Eq. 3 can be computed as the sum of the function $SN(n)$ over all the nodes n of the tree t:

$$\mathrm{DT}(t) = \sum_{n \in N(t)} \mathrm{SN}(n) \tag{5}$$

where $N(t)$ is the set of nodes of t.

Hence, distributed tree encoders are fast and effective ways to encode trees in small vectors. The approximation can get arbitrarily good by increasing the

[1]The circular convolution between $\mathbf{a}$ and $\mathbf{b}$ is defined as the vector $\mathbf{c}$ with component $c_i = \sum_j a_j b_{i-j \bmod d}$. The shuffled circular convolution is the circular convolution after the vectors have been randomly shuffled.

dimension of the vector space. Thus, the vector $\mathrm{DT}(t)$ encodes the information about the tree t. However, it is not immediately obvious how one could go about using this encoded information to recreate the symbolic version of the parse tree.

4. Decoding Distributed Trees: Reconstructing Symbolic Trees from Vectors

Distributed tree encoders are powerful functions which represent trees in small vectors, *but are these encoders invertible*? Is it possible to reconstruct trees encoded in distributed trees? This is an important question as the answer may show that distributed trees can be an interpretable mapping of syntactic trees in distributed representations. Hence, distributed trees are attractive representations to be used as the interface between learning models and syntactic structure, especially when interpreting parts of the learning model is relevant.

Our main contribution is to propose decoders to invert functions that encode trees in distributed trees. As these functions are not invertible per se, we treat this problem as a parsing problem. That is, we are given a sentence s with a distributed tree $\mathbf{t} = \mathrm{DT}(t)$, and the goal is to reconstruct the original tree t. To do so the parsing method will use a large context-free grammar G that defines the space of trees. From now on we will assume a fixed grammar G and sentence s, and we will denote a decoder as a function $\mathrm{DT}^{-1}(t)$.

We propose two classes of decoders based on parsing algorithms: one for binary trees and one for general trees. We introduced the first decoder based on the CYK algorithm in [19]. In this paper, we generalize the approach to general trees in two new ways: (1) by using a binarizer and a debinarizer along with the first decoder; and, (2) by adapting CYK+ that is an algorithm for parsing general context free grammars in a probabilistic setting. One benefit of using CYK to decode distributed trees is that the recursive computations for distributed trees $\mathrm{SN}(n)$ and $\mathrm{DT}(t)$ in equations 4 and 5 factorize across the CYK chart. This means that the best partial trees at each CYK cell can be stored as distributed trees, to be augmented during the bottom up process.

The rest of the section is organized as follows. Section 4.1 revises the model for binary trees that is based on the CYK parsing algorithm which is used for Chomsky Normal Form probabilistic grammars. Section 4.2 describes two new models to deal with general trees, one based on the CYK algorithm followed by a process of debinarization, and one based on the CYK+ algorithm [20, 21] which has been introduced for general probabilistic context free grammars.

4.1. Decoders for Binary Trees

This section revises our first decoder $\mathrm{DT}_B^{-1}(\mathbf{t})$ for distributed trees $DT(t) = \mathbf{t}$ that deals only with binary trees. We presented this solution in [19]. This first decoder was implemented as a variant of a probabilistic version of the CYK algorithm (see [19] for more details on the CYK algorithm). Instead of the probabilities of parsing rules, our decoder uses the information contained in

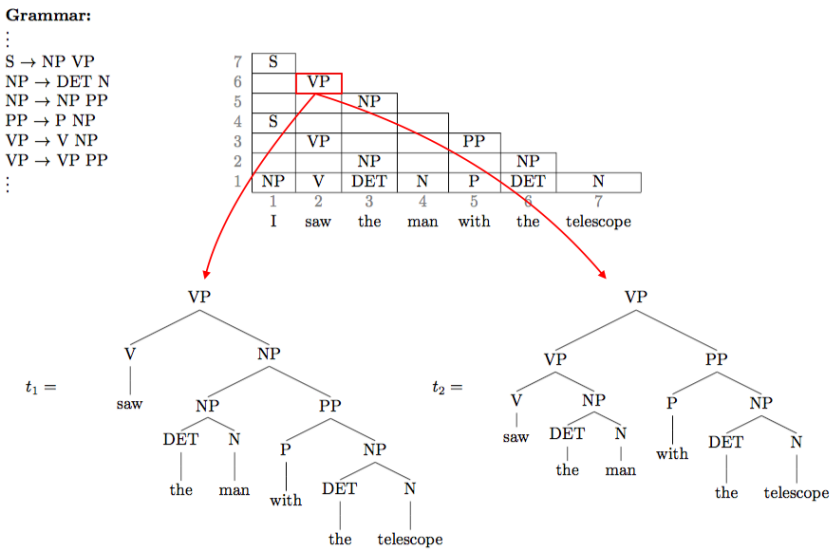


Figure 2: Parsing Example

distributed trees to guide the reconstruction of the original parse tree among all the possible parse trees of a given sentence.

In the rest of the section, we first describe our variant of the CYK algorithm applied to distributed trees and, then, we report on some additional rules that we introduced to improve the computational efficiency of the decoder.

4.1.1. CYK over Distributed Trees

CYK is one of the most efficient algorithms for parsing context-free grammars in Chomsky-Normal Form, that is, with rules of the form $A \rightarrow B\ C$ and $A \rightarrow w$. In its most common form the CYK algorithm works by constructing a 2-dimensional table P, where the cell $P[i, j]$ contains the symbols of the trees that can cover the span from word j to word $i + j - 1$. For example, Figure 2 depicts the table P for the sentence "I saw the man with the telescope" and P[2,6] contains VP as, according to the grammar G, this symbol is the head of some trees covering "saw the man with the telescope". In the probabilistic version of CYK, each cell $P[i, j]$ contains the k-most-probable parse trees.

Our decoder $\mathrm{DT}_B^{-1}(\mathbf{t})$ is based on the CYK algorithm and uses the information encoded in the distributed tree $\mathrm{DT}(t) = \mathbf{t}$ to guide the parsing which aims to reconstruct trees t from sentences s. The idea is simple: instead of using rule and subtree probabilities to guide the parsing, we seek whether candidate rules r or reconstructed subtrees t_ℓ in a cell *are* in the distributed tree $\mathrm{DT}(t) = \mathbf{t}$. The rationale is as follows. Assume that a production rule r appears once in the

target tree t, then the dot product between the distributed tree of r and $\mathbf{t}$ is:

$$\langle \mathrm{DTF}(r), \mathbf{t} \rangle \approx \lambda^{\frac{3}{2}}$$

as rules are trees with 3 nodes. If r occurs c times in t then the dot product will approximately be c times the above quantity, and in particular if the rule does not occur in t, the dot product will be zero. More generally, if t_ℓ is a candidate reconstructed partial tree that occurs in the target tree t, then the dot product with $\mathbf{t}$ is:

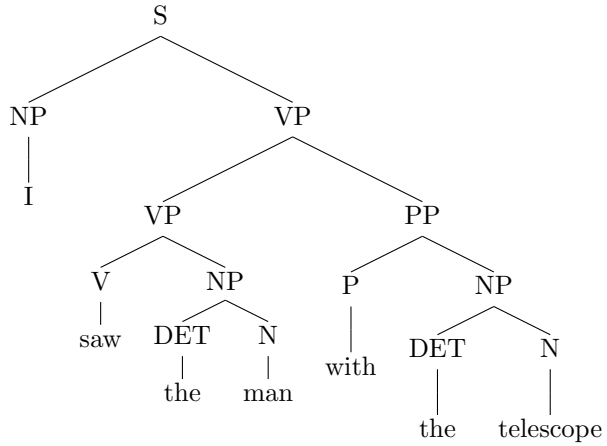
$$\langle \mathrm{SN}(t_\ell), \mathbf{t} \rangle \approx \sum_{\tau \in S(t_\ell)} \lambda^{\frac{|\mathcal{N}(\tau)|}{2}}$$

as $\mathrm{SN}(t_\ell)$ represents the sum of the vectors of the subtrees of t_ℓ rooted in its root. Hence, we use the following score to drive the choice of the m-best reconstructed trees t_ℓ in each cell $P[i, j]$:

$$\mathrm{score}(t_\ell, \mathbf{t}) = \frac{\langle \mathrm{SN}(t_\ell), \mathbf{t} \rangle}{\langle \mathrm{DTF}(r), \mathbf{t} \rangle}$$

This score is 1 if t_ℓ is a rule occurring in t, and is around 1 if t_ℓ is a subtree of $\mathbf{t}$.

More formally, the pseudocode for our algorithm is presented in Alg. 1. The key steps are in 16-18 where candidate trees and candidate rules are selected. When we are in the cell $P[i, j]$ for all rules r in the grammar of the form $A \to B\ C$, we check for all trees that we already built in $P[j, k]$ and $P[j+k, i-k]$ that starts with $B \to \bullet\ \bullet$ and $C \to \bullet\ \bullet$, respectively. Let us call these two lists L_B and L_C, each composed of at most m trees. We create then the list L_{BC} composed of at most m^2 trees that have A as root, a subtree from L_B as the left child and one coming from L_C as the right child. For each one of these trees t_ℓ we compute $\mathrm{score}(t_\ell, \mathbf{t})$. We then sort the list L_{BC} in decreasing order according to score and store only the first m trees. For example, in Fig. 2, we want to reconstruct the tree:



given the sentence "I saw the man with the telescope" and the grammar G. In this case, in the cell $P(2, 6)$ we store two trees t_1 and t_2, to which correspond

two distributed vectors that we use to score and sort the trees. We have that $\text{score}(t_1, \mathbf{t}) < \text{score}(t_2, \mathbf{t})$, and so it is more probable that t_2 will lead us to the correct interpretation of the sentence.

Algorithm 1 CYK, DTK variant

Input: sentence $s = w_1, w_2, \ldots, w_n$, grammar G, distributed vector $\mathbf{t}$

1: **Initialization:** fill an $n \times n$ table P with zeroes
2: **for** $i = 1$ to n **do**
3: **for all** symbol $A \in G$ **do**:
4: rule $\leftarrow (A \rightarrow w_i)$
5: ruleScore $\leftarrow \frac{\langle \text{SN(rule)}, \mathbf{t} \rangle}{\|\text{SN(rule)}\|}$
6: **append** (rule, ruleScore) to $P[i, 0]$
7: **sort** $P[i, 1]$ decreasingly by **score**
8: take the first m element of $P[i, 1]$
9: **for** $i = 2$ to n **do**
10: **for** $j = 1$ to $n - i + 1$ **do**
11: **for** $k = 1$ to $i - 1$ **do**
12: **for all** pairs $(B \rightarrow \bullet\ \bullet, C \rightarrow \bullet\ \bullet) \in P[j, k] \times P[j + k, i - k]$ **do**
13: **for all** rule $(A \rightarrow B\ C) \in G$ **do**
14: ruleScore $\leftarrow \frac{\langle \text{DTF(rule)}, \mathbf{t} \rangle}{\|\text{DTF(rule)}\|}$
15: **if** ruleScore $\geq$ threshold **then**
16: tree $\leftarrow A \rightarrow [B \rightarrow \bullet\ \bullet, C \rightarrow \bullet\ \bullet]$
17: score $\leftarrow \langle \text{DT(tree)}, \mathbf{t} \rangle$
18: **append** (tree, score) to $P[i, j]$
19: **sort** $P[i, j]$ decreasingly by **score**
20: take the first m element of $P[i, j]$
21: **if** $P[1, n]$ is **not** empty **then**
22: **return** $P[1, n]$
23: **else**
24: **return** Not parsed

Unfortunately, the decoder may fail to reconstruct the tree. In fact, as we do not store each possible candidate tree in each cells, it may happen that this algorithm will fail to recognize a sentence as belonging to the grammar. Hence, we need some heuristic rules to better drive the reconstruction/parsing and, in the meantime, keeping the complexity low.

4.1.2. Improving Computational Efficiency and Accuracy with Heuristic Rules

To control the execution time and giving the possibility to increase the m-best list of candidate subtrees, we introduced a filter that filters out useless rules. The filter (see line 14 of Alg. 1) avoids to cycle on every rule of the grammar and focuses the search on promising rules, which are in the distributed tree. Such score is computed again as a similarity between the rule r (viewed as a tree) and

$\mathbf{t}$:

$$\text{rulescore}(r, \mathbf{t}) = \frac{\langle \text{DTF}(r), \mathbf{t} \rangle}{\|\text{DTF}(r)\|}$$

the threshold (line 15) is defined as:

$$\frac{\lambda^{\frac{3}{2}}}{p}$$

The numerator $\lambda^{\frac{3}{2}}$ represents the exact score (not an approximation) that a rule would get if it appeared exactly once in the whole tree, while the parameter $p > 1$ relaxes the requirement by lowering this threshold, in order to take in account the fact that we are dealing with approximated scores.

To control the quality and to give better chances to the algorithm to retrieve a parse tree, we introduced two strategies: (1) the introduction of a constraint which augments the variability of the tree roots in cell; and (2) a final reranking of the extracted trees. The first strategy aims to avoid the algorithm getting stuck on the wrong choice. Hence, after the sorting and trimming of the list of the first m trees in each cell, we also add another m trees (again, sorted by score) but for which the root node is different than the node of the first tree in the cell. The second strategy instead is a final reranking of the solutions. Candidate trees t_c in the final list are reranked according to this simple score:

$$score(t_c, \mathbf{t}) = \langle \text{DT}(t_c), \mathbf{t} \rangle$$

The performance of the decoder will then depend on: the size d of the space encoding distributed trees as vectors in $\mathbb{R}^d$, the parameter p to select rules with $\text{rulescore}(r, \mathbf{t})$ and m of the m-best trees retained in a cell. We evaluated these parameters in the experimental section.

4.2. Decoding General Trees

The decoder revised in the previous section has a severe problem: it works only on binary trees. This is a limitation as natural language parse trees are usually not binary, and must therefore be artificially binarized.

The main innovation of this paper is to introduce two decoders that can decode with general trees. The first approach is very simple and uses the previous binary distributed tree decoder. In the encoding phase, general trees are binarized before the application of the distributed tree encoder. In the decoding phase, the binary distributed tree decoder retrieves binary trees from distributed trees and, then, a debinarizer is applied. The second approach is a dedicated algorithm that decodes general trees from distributed trees encoding generic trees.

In the following, we will report on both these approaches.

4.2.1. Debinarizing Reconstructed Trees

Our first decoder on generalized trees $\text{DT}_{G1}^{-1}(\mathbf{t})$ is a straightforward extension of what we did before, the decoder on binarized tree $\text{DT}_{B}^{-1}(\mathbf{t})$. In fact, the

decoder $\mathrm{DT}_B^{-1}(\mathbf{t})$ is obtained with a debinarizer B^{-1} applied in cascade to the decoder for binary distributed trees:

$$\mathrm{DT}_{G1}^{-1}(\mathbf{t}) = B^{-1}(\mathrm{DT}_B^{-1}(\mathbf{t}))$$

where $B^{-1}(t)$ is a debinarizer. Given $B(t)$ as the binarizing function, the overall process to encode and decode a general tree is the following:

$$t \mapsto B(t) \mapsto \mathrm{DT}_B(B(t)) = \mathbf{B}(\mathbf{t}) \mapsto B^{-1}(\mathrm{DT}_B^{-1}(\mathbf{B}(\mathbf{t}))) = t$$

The debinarization is achieved with the same model that produces the binary trees and is a lossless deterministic process.

4.2.2. Decoders for General Trees with CYK+

Our second decoder on generalized trees $\mathrm{DT}_{G2}^{-1}(\mathbf{t})$ is instead designed to natively treat general trees. In fact, general trees are encoded with the distributed tree encoder as they are and, then, the decoder $\mathrm{DT}_{G2}^{-1}(\mathbf{t})$ extract back trees with no need of reworking.

To define the decoder, we adapted the CYK+ algorithm [20] to exploit information stored in distributed trees.

CYK+ is an algorithm for parsing with general probabilistic context-free grammars which uses a matrix structure P to store partial parses. The matrix of CYK+ is richer than the one of CYK. Each cell $P[i, j]$ has two disjoint lists of items: (1) Complete Rules; and, (2) Incomplete Rules. The Complete Rules are rules of the form:

$$A \to B_1\ B_2\ \ldots\ B_n$$

that parse the span subtended by the cell, that is, the words: $w_j \ldots w_{i+j-1}$. For brevity, an element of this type will be represented by the non-terminal in the left side, A, keeping in mind that the algorithm can keep track of all the partial derivation via backpointers, just as in the standard CYK algorithm. The Incomplete Rules are strings of non-terminal symbols:

$$[B_1, B_2, \ldots, B_n]$$

that again span $w_j \ldots w_{i+j-1}$ and for which there exist a rule in the grammar of the form

$$A \to B_1\ B_2\ \ldots\ B_n\ \beta$$

where β is a non-empty string of non-terminal symbols. In other words, the second list contains partial subtrees that could be completed into entire parse trees later on in the parsing algorithm. An element of this type will be indicated with $\alpha\bullet$, where α is representing $[B_1, B_2, \ldots, B_n]$.

The main operations of CYK+ on a given cell $P[i, j]$ of the matrix P are two: the *completer* and the *self-filler*. The *completer* wants to find new (possibly partial) rules for each cell $P[i, j]$. Hence, it seeks the combinations that could form a new (possibly partial) rule in the cells $P[j, k]$ and $P[j + k, i - k]$ (see lines 31-48 of Alg. 2). More precisely, an incomplete rule $\alpha\bullet$ in $P[j, k]$ is combined

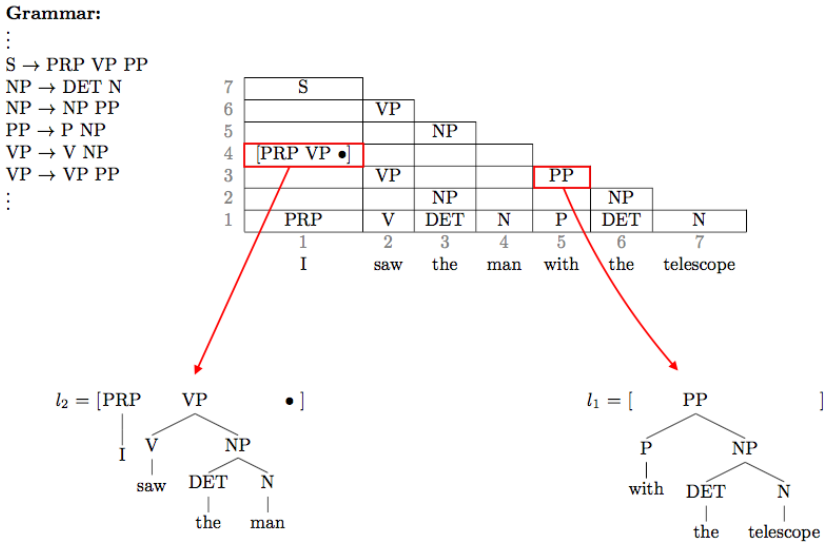


Figure 3: Parsing Example

with a complete rule B in $P[j+k, i-k]$ if there is a rule in the grammar of the form $A \rightarrow \alpha B\gamma$, where γ is a (possibly empty) list of non terminal symbols. If γ is empty the rule A represents a complete rule and is thus placed into the list of complete rules of the active cell, otherwise the sequence $\alpha B\bullet$ is put in the list of incomplete rules to be expanded later (if possible). The *self-filler* is instead a procedure needed to deal with unary rules (see lines 50-64 of Alg. 2). For each complete rule of the form B in the active cell $P[i, j]$ the procedure looks in the grammar for rule $A \rightarrow B\gamma$. If γ is empty, this new rule A is put back into the same list, otherwise $B\bullet$ is put into the list of incomplete rules.

An example on how the CYK+ works is given in Figure 3. The *completer* fills the cell $P[1, 7]$ with S as the cell $P[1, 4]$ contains the incomplete rule [PRP VP •] and the cell $P[5, 3]$ contains the complete rule [PP]. These two rules can be combined together with the complete rule S → PRP VP PP. In the example we can also see how in each cell we store entire subtrees for each type of rule.

Our second decoder $\text{DT}^{-1}_{G2}(\mathbf{t})$ on generalized trees then extends CYK+ adding the ranking of the complete rules and incomplete rules in a cell using distributed trees. The principle is the same, the decoder seeks in distributed trees whether or not the rule or the incomplete rule exists by using the dot product. Yet, seeking incomplete rules in the distributed tree is fairly more complex as incomplete rules are not directly encoded. What is in the distributed trees is a complete rule or a partial tree. Then, this is what we can find. Hence, we need to derive which complete rules or subtrees are implied by the list of incomplete rules in a

cell $P[i, j]$. This list is of the form:

$$L = [\alpha\bullet, \beta\bullet, \ldots, \zeta\bullet]$$

where each element is itself a list of non terminal symbols (together with its own subtrees) which represents a partial rule. Moreover, for each $\alpha\bullet = [t_1, t_2, \ldots, t_n]$, there exists rules R_j in the grammar of the form $R_j \rightarrow \alpha\,\beta$. Hence, $\alpha\bullet$ will receive a score according to the following equation:

$$\text{score}(\alpha\bullet) = \max_{R_j} \text{score}_{\text{rule}}(R_j) \cdot \frac{1}{n} \sum_i \frac{\text{score}_{\text{tree}}(t_i)}{\sqrt{\#nodes(t_i)}}$$

where R_j ranges over all the rules that could complete $\alpha\bullet$, and whose score is defined as:

$$\text{score}_{\text{rule}}(R) = \frac{\langle DTF(R), \mathbf{t} \rangle}{\lambda^{\frac{\#nodes(R)}{2}}}$$

and $\text{score}_{\text{tree}}(t)$ is instead the score of the entire subtree:

$$\text{score}_{\text{tree}}(t_i) = \langle DTK(t_i), \mathbf{t} \rangle$$

The score is used in line 69 of the Alg. 2. In words, we are scoring a partial rule by the average of the score of its constituent trees, multiplied by the maximum possible score among all the rules that *could* complete it. The rest of the scoring functions are similar to what done for the decoder on binary trees. Similarly to the decoder for binary trees the performance of the algorithm will depends on 3 parameters: the dimension d in which we embed the distributed vectors, the threshold parameter p and the number m of trees that we store in each cell.

Algorithm 2 CYK+ DTK variant

Input: sentence $s = w_1, w_2, \ldots, w_n$, grammar G, distributed vector $\mathbf{t}$

Initialization: fill an $n \times n$ table P with zeroes
for $i = 1$ to n **do**
    for all symbol $A \in G$ **do**:
        rule $\leftarrow (A \rightarrow w_i)$
        ruleScore $\leftarrow \frac{\langle \mathrm{SN(rule)}, \mathbf{t} \rangle}{\|\mathrm{SN(rule)}\|}$
        append (rule, ruleScore) to completeList of $P[i, 1]$
    sort $P[i, 1]$.completeList decreasingly by **score**
    take the first m element of $P[i, 1]$.completeList

for all tree $T \rightarrow \bullet \in P[i, 1].\mathit{completeList}$ **do**:
    for all rule $(A \rightarrow T\ \gamma) \in G$ **do**:
        divide between complete and incomplete rules:
        if γ is empty **then**:
            $A \rightarrow T$ is a complete rule
        else:
            [T] is an incomplete rule

    for all complete rule $A \rightarrow T$ **do**:
        tree $\leftarrow A \rightarrow [T \rightarrow \bullet]$
        score $\leftarrow \langle \mathrm{SN(tree)}, \mathbf{t} \rangle$
        append (tree, score) to P[i, 1].completeList

    if there is at least one incomplete rule $A \rightarrow T\ \alpha$ **then**:
        append $[T \rightarrow \bullet]$ to P[i,1].incompleteList

sort $P[i, 1]$.completeList decreasingly by **score**
take the first m element of $P[i, 1]$.completeList

for $i = 2$ to n **do**
    for $j = 1$ to $n - i + 1$ **do**
        for $k = 1$ to $i - 1$ **do**
            Partial $\leftarrow$ P[j, k].incompleteList
            Complete $\leftarrow$ P[j+k,i-k].completeList
            for all pairs $([B_1, B_2, \ldots, B_n], C) \in$ Partial $\times$ Complete **do**:
                for all rule $(A \rightarrow B_1\ B_2\ \ldots\ B_n\ C\ \gamma) \in G$ **do**:
                    divide between complete and incomplete rules:
                    if γ is empty **then**:
                        $A \rightarrow B_1\ B_2\ \ldots\ B_n\ C$ is a complete rule
                    else:
                        $[B_1\ B_2\ \ldots\ B_n\ C]$ is an incomplete rule

                    for all complete rule $(A \rightarrow B_1\ B_2\ \ldots\ B_n\ C)$ **do**:
                        tree $\leftarrow A \rightarrow [B_1\ B_2\ \ldots\ B_n\ C]$
                        score $\leftarrow \langle \mathrm{SN(tree)}, \mathbf{t} \rangle$
                        append (tree, score) to P[i, j].completeList

                    if there is an incomplete rule $(A \rightarrow B_1\ \ldots\ B_n\ C\ \gamma)$ **then**:
                        append $[B_1\ B_2\ \ldots\ B_n\ C]$ to P[i,j].incompleteList

```
49:         for all tree T → • ∈ P[i, j].completeList do:
50:             for all rule (A → T γ) ∈ G do:
51:                 divide between complete and incomplete rules:
52:                 if γ is empty then:
53:                     A → T is a complete rule
54:                 else:
55:                     [T] is an incomplete rule
56:
57:             for all complete rule A → T do:
58:                 tree ← A → [T → •]
59:                 score ← ⟨SN(tree), t⟩
60:                 append (tree, score) to P[i, j].completeList
61:
62:             if there is at least one incomplete rule A → T α then:
63:                 append [T → •] to P[i,j].incompleteList
64:
65:         sort P[i, j].completeList decreasingly by score
66:         take the first m element of P[i, j].completeList
67:         sort P[i, j].incompleteList decreasingly by score
68:         take the first m element of P[i, j].incompleteList
69:
70: if P[1, n] is not empty then
71:     final reranking by dtk
72:     return P[1, n]
73: else
74:     return  Not parsed
```

5. Experiments

With the 3 different decoders we proposed (DT_B^{-1}, DT_{G1}^{-1} and DT_{G2}^{-1}) there is the possibility of building encoding-decoding chains for trees which transforms trees in distributed trees and back. The experiments in this section investigate if these chains are lossless. Hence, we want to determine whether tree encoders preserve the information so that decoders can extract back trees.

To investigate if encoding-decoding chains are lossless, we performed two different sets of experiments: one for the binary encoder-decoder chain and one for the general encoder-decorer chain. In the first set, we binarized the entire dataset beforehand (and thus we will only ever work with a binary grammar and binary parse trees). In the second set, we instead explored the two aforementioned ways to deal with a dataset of trees as they are produced by the parser. In both experiments, we assessed the methodology with richer trees containing words and with poorer trees containing numbers as terminal symbols. This was useful to understand if these encoding-decoding chains are affected by the sparseness of the symbols.

In the rest of the section, first, we describe the experimental setup and, then, we report on the results of the experiments.

5.1. Experimental Setup

As our aim was to experiment with natural language parse trees, we used the Penn Treebank [32], namely, the WSJ section. Even if this is not a "parsing" experiment, we used the standard split as we need sections to generate the credible large grammar for the two underlying parsing algorithms. Hence, we used sections 00 to 22 to generate the grammar G and section 23 for testing consisting of a total of $2,389$ sentences. The generated grammar is not a probabilistic one, nor there is any learning involved. Instead, it is just a collection of all the rules that appear in parse trees in those sections. We generated two grammars: (1) a binary grammar for DT_B^{-1} that contains 95 (non-terminal) symbols and 1706 (non-terminal) production rules; (2) a general grammar for DT_{G1}^{-1} and DT_{G2}^{-1} that contains 16443 different rules, ranging in complexity from unary rules like $NP \rightarrow ADVP$ to pathological ones often composed of very long sequences of noun phrases such as $NP \rightarrow NP,\ NP,\ NP,\ NP,\ NP,\ NP,\ NP\ CC\ NP$.

To evaluate the encoding-decoding chains, we used the classical metrics –labeled recall, labeled precision and labeled f-measure– along with a stricter *percentage of correctly reconstructed trees*. This latter evaluation measure is extremely demanding as it asks that trees are perfectly reconstructed. Hence, it describes whether or not the encoding-decoding is a perfectly lossless function.

We experimented with different parameter setting to understand their role in the encoding and decoding power of the chains. The parameters have been reported at the end of each section describing decoders (Secs. 4.1 and 4.2). The values we experimented with are:

- For the dimension d of the distributed trees: $1024, 2048, 4096, 8192$ and 16384

- For the threshold p of the rule filter: $p = 1.5, 2$ and 2.5
- For the m of the m-best solutions for each node: we tried the values up to 10 but found out that increasing this value does not increase the performance of the algorithm, while at the same time increasing significantly the computational complexity. For this reason we fixed the value of this parameter to 2 and only report results relative to this value.

Finally, in order to be ready to work in a distributed parsing setting [33] and in line with [14] where words are not taken into consideration, we experimented with two settings: a *lexicalized setting* and an *unlexicalized setting*. In the *lexicalized setting*, binary and general trees contain words. In the *unlexicalized setting*, binary and general trees contain numbers representing word positions instead of words. This latter setting is designed to remove an unnecessary complication. In this case, decoders deal with a reduced space of terminals and not with the entire space of terminal rules. One simple way to achieve a "semantic-free" tree representation is thus to just replace each word in a sentence s with a progressive numeral number. In this way the set of terminal rules is reduced and does not carry any semantic meaning, rather the terminal rules are only placeholders for the position of a word in the sentence, as in [14].

5.2. Results

In this section we report our results on the dataset. We will report result for the experiments on binary trees and general trees separately in the next subsections.

5.2.1. Binary trees

For the binary trees we summarize the results presented in [19]. In table (1) we report the percentage of *exactly* reconstructed sentences together with precision and recall on the entire dataset. We let the dimension d vary while the parameter λ is kept fixed at 0.6. Here we only report the value for the parameter p fixed at 2, which provides the best results. We report the results both for the lexicalized and unlexicalized settings.

Table 1: DT_B^{-1}: percentage of *correctly reconstructed trees*, precision and recall; $p = 2$

d	% correct trees	precision	recall
1024	23.5%	0.78	0.477
2048	60.46%	0.912	0.78
4096	81.39%	0.967	0.929
8192	91.86%	0.994	0.967
16384	92.54%	0.995	0.976

(a) lexicalized

d	% correct trees	precision	recall
1024	24.2%	0.78	0.49
2048	50.6%	0.91	0.78
4096	83.2%	0.976	0.946
8192	92.4%	0.991	0.988
16384	95.8%	0.996	0.991

(b) unlexicalized

As we can see, for the experiment where words are kept the number of correctly reconstructed sentences grows significantly with the increasing of the

dimension (as expected) topping at 92.54% for $d = 16384$. On the other hand lower values of d, while yielding a low percentage of reconstructed sentences, still can provide a high precision with value as low as 1024 resulting in a precision of 0.78, and a top score of 0.995. Removing words increases the performance even more, achieving a top score of 95.8% for $d = 16384$, and a precision of 0.91 for d as low as 2048.

In conclusion it seems that the main parameter to influence the algorithm is the dimension d, which was what we expected, because the quality of the approximation depends on d, and thus the amount of information that can be stored in $\mathrm{DT}(t)$ without too much distortion.

5.2.2. General trees

In this section we present similar results for the reconstruction of general trees for both method of reconstruction. For general trees however we will present the results in more detail: in table (2) we report the percentage of exactly reconstructed trees for the CYK algorithm followed by debinarization, with varying dimension and threshold parameter p and for both the lexicalized and unlexicalized settings. Table (3) reports the same results for the CYK+ algorithm.

Table 2: DT_{G1}^{-1}: percentage of *correctly reconstructed sentences*

	p		
d	1.5	2	2.5
1024	21.0%	24.4%	19.4%
2048	37.8%	45.6%	41.8%
4096	61.2%	65.6%	60.2%
8192	69.4%	70.2%	69.8%
16384	**72.4**%	**72.4**%	72.2%

(a) lexicalized

	p		
d	1.5	2	2.5
1024	21.0%	25.0%	22.4%
2048	41.4%	46.4%	43.4%
4096	65.2%	66.6%	64.0%
8192	70.0%	70.4%	70.4%
16384	**72.6**%	**72.6**%	72.0%

(b) unlexicalized

Table 3: DT_{G2}^{-1}: percent of *correctly reconstructed trees.*

	p		
d	1.5	2	2.5
1024	13.8%	15.0%	13.0%
2048	30.6%	32.2%	28.2%
4096	44.2%	47.4%	44.4%
8192	54.4%	54.8%	53.8%
16384	63.0%	**63.4**%	63.0%

(a) lexicalized

	p		
d	1.5	2	2.5
1024	15.0%	18.0%	15.4%
2048	30.8%	33.8%	32.6%
4096	49.6%	52.4%	47.4%
8192	60.2%	62.4%	59.6%
16384	67.4%	**68.0**%	67.6%

(b) unlexicalized

In table (4) and (5) we report the average precision and recall on the entire

Table 4: DT^{-1}_{G1}: Average precision and recall.

d	p 1.5	2	2.5
1024	0.894	0.774	0.713
2048	0.928	0.872	0.825
4096	0.96	0.942	0.917
8192	0.962	0.961	0.957
16384	**0.964**	0.963	0.963

(a) precision - lexicalized

d	p 1.5	2	2.5
1024	0.905	0.788	0.725
2048	0.946	0.896	0.841
4096	0.961	0.948	0.930
8192	0.963	0.960	0.957
16384	**0.964**	**0.964**	0.963

(b) precision - unlexicalized

d	p 1.5	2	2.5
1024	0.305	0.469	0.508
2048	0.539	0.743	0.766
4096	0.811	0.904	0.903
8192	0.935	0.949	0.946
16384	**0.958**	0.956	0.955

(c) recall - lexicalized

d	p 1.5	2	2.5
1024	0.302	0.464	0.51
2048	0.57	0.742	0.779
4096	0.845	0.914	0.918
8192	0.934	0.952	0.951
16384	0.956	**0.958**	0.954

(d) recall - unlexicalized

dataset for different values of p, fixed $\lambda = 0.6$, varying the dimension d, and different reconstruction approach. Again, for each decoder we list both the case where words are present and the case were words are removed.

We can immediately see that the general case is considerably more difficult to tackle than the binary one, particularly using the DT^{-1}_{G2} decoder. While precision and recall are only marginally lower than the previous case, the number of exactly reconstructed sentences drops significantly. This is probably due to the bigger size of the search space and the fact that the algorithm has to prune a considerable part of it at each step, moreover the process of scoring incomplete rules in the CYK+ algorithm is more indirect and error-prone then the scoring of complete rules, for which the dot-product readily provides the information that we want (that is, whether a tree is a subtree of the target tree). The general trend is the same as in the previous experiment, with increasing performance with the increasing of dimensionality, and the parameter p not influencing the result as much, however the results are generally lower: the top result is achieved debinarizing the output of the CYK algorithm, where the lexicalized and unlexicalized setting achieve similar results of 72.4% and 72.6%, respectively. Precision and recall are instead only marginally lower: the best results are again achieved with the debinarizing approach where the top precision and recall achieved are 0.964 and 0.958, respectively, both for the setting with and without words. The best recall score is achieved with the debinarizing approach, with a top score of 0.958 in both settings. The precision instead is higher with the CYK+ approach: the top score is 0.991 in the setting without

Table 5: DT_{G2}^{-1}: Average precision and recall.

d	p 1.5	2	2.5
1024	0.958	0.818	0.714
2048	0.965	0.893	0.807
4096	0.981	0.96	0.918
8192	0.987	0.980	0.970
16384	**0.989**	0.988	0.988

(a) precision - lexicalized

d	p 1.5	2	2.5
1024	0.94	0.80	0.71
2048	0.96	0.90	0.82
4096	0.98	0.96	0.93
8192	0.99	0.99	0.98
16384	**0.991**	0.99	0.989

(b) precision unlexicalized

d	p 1.5	2	2.5
1024	0.181	0.360	0.434
2048	0.427	0.588	0.692
4096	0.601	0.709	0.727
8192	0.710	0.733	0.754
16384	0.801	0.804	**0.819**

(c) recall - lexicalized

d	p 1.5	2	2.5
1024	0.20	0.39	0.49
2048	0.43	0.63	0.71
4096	0.67	0.77	0.78
8192	0.77	0.81	0.82
16384	0.84	0.863	**0.87**

(d) recall unlexicalized

words, and 0.989 for the setting with words.

6. Conclusions and Future Work

Representing linguistic information in distributed representation is a promising trend in modern Natural Language Processing approaches, which combine machine learning models along with linguistic models.

In this paper, we investigated models to show that encoding functions of complex linguistic information, that is syntactic trees, are invertible. We proposed two nearly lossless chains of encoder-decoding process for syntactic trees. Hence, distributed representations for trees are good candidates to store and represent this kind of linguistic information.

Our results enhance the comprehension of what is the real encoding power of these distributed representations. Thus, we shed a new light on how these representations are used in learning models. As supposed in [34], our results make possible to investigate the relation between distributed representations used in learning machines and convolution kernels.

Finally, our contributions to decode distributed trees introduced in [1] open new research avenues: exploring novel approaches to parsing. Distributed trees may be the intermediate representation for novel learning-based parsers. The so-called *distributed representation parsers* [33, 35] could finally become complete parsers for natural language utterances.

[1] F. M. Zanzotto, L. Dell'Arciprete, Distributed Tree Kernels, in: Proceedings of International Conference on Machine Learning, –, 2012.

[2] P. D. Turney, P. Pantel, From Frequency to Meaning: Vector Space Models of Semantics, J. Artif. Intell. Res. (JAIR) 37 (2010) 141–188.

[3] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient Estimation of Word Representations in Vector Space, CoRR abs/1301.3781, URL http://arxiv.org/abs/1301.3781.

[4] J. Mitchell, M. Lapata, Vector-based Models of Semantic Composition, in: Proceedings of ACL-08: HLT, Association for Computational Linguistics, Columbus, Ohio, 236–244, URL http://www.aclweb.org/anthology/P/P08/P08-1028, 2008.

[5] M. Baroni, R. Zamparelli, Nouns are Vectors, Adjectives are Matrices: Representing Adjective-Noun Constructions in Semantic Space, in: Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, Cambridge, MA, 1183–1193, URL http://www.aclweb.org/anthology/D10-1115, 2010.

[6] S. Clark, B. Coecke, M. Sadrzadeh, A Compositional Distributional Model of Meaning, Proceedings of the Second Symposium on Quantum Interaction (QI-2008) (2008) 133–140.

[7] E. Grefenstette, M. Sadrzadeh, Experimental support for a categorical compositional distributional model of meaning, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing, EMNLP '11, Association for Computational Linguistics, Stroudsburg, PA, USA, ISBN 978-1-937284-11-4, 1394–1404, URL http://dl.acm.org/citation.cfm?id=2145432.2145580, 2011.

[8] F. M. Zanzotto, I. Korkontzelos, F. Fallucchi, S. Manandhar, Estimating Linear Models for Compositional Distributional Semantics, in: Proceedings of the 23rd International Conference on Computational Linguistics (COLING), 2010.

[9] J. B. Pollack, Recursive Distributed Representations, Artificial Intelligence 46 (1-2) (1990) 77–105, ISSN 0004-3702, doi:\bibinfo{doi}{10.1016/0004-3702(90)90005-K}, URL http://dx.doi.org/10.1016/0004-3702(90)90005-K.

[10] R. Socher, J. Pennington, E. H. Huang, A. Y. Ng, C. D. Manning, Semi-Supervised Recursive Autoencoders for Predicting Sentiment Distributions, in: Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, Edinburgh, Scotland, UK., 151–161, URL http://www.aclweb.org/anthology/D11-1014, 2011.

[11] R. Socher, B. Huval, C. D. Manning, A. Y. Ng, Semantic Compositionality Through Recursive Matrix-Vector Spaces, in: Proceedings of the 2012 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2012.

[12] F. Zanzotto, L. Dell'Arciprete, Distributed tree kernels, in: Proceedings of International Conference on Machine Learning, 193–200, URL http://www.scopus.com/inward/record.url?eid=2-s2.0-84867126965&partnerID=40&md5=0d51c0ed7070baf730f887c818a8c177, 2012.

[13] M. Baroni, A. Lenci, Distributional memory: A general framework for corpus-based semantics, Comput. Linguist. 36 (4) (2010) 673–721, ISSN 0891-2017, doi:\bibinfo{doi}{10.1162/coli_a_00016}, URL http://dx.doi.org/10.1162/coli_a_00016.

[14] O. Vinyals, L. u. Kaiser, T. Koo, S. Petrov, I. Sutskever, G. Hinton, Grammar as a Foreign Language, in: C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, R. Garnett (Eds.), Advances in Neural Information Processing Systems 28, Curran Associates, Inc., 2755–2763, URL http://papers.nips.cc/paper/5635-grammar-as-a-foreign-language.pdf, 2015.

[15] J. Cheng, L. Dong, M. Lapata, Long Short-Term Memory-Networks for Machine Reading, CoRR abs/1601.06733, URL http://arxiv.org/abs/1601.06733.

[16] N. Cristianini, J. Shawe-Taylor, An Introduction to Support Vector Machines and Other Kernel-based Learning Methods, Cambridge University Press, ISBN 0521780195, URL http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/0521780195, 2000.

[17] S. Filice, D. Croce, R. Basili, F. M. Zanzotto, Linear Online Learning over Structured Data with Distributed Tree Kernels, in: 12th International Conference on Machine Learning and Applications, ICMLA 2013, Miami, FL, USA, December 4-7, 2013, Volume 1, 123–128, doi:\bibinfo{doi}{10.1109/ICMLA.2013.28}, URL http://dx.doi.org/10.1109/ICMLA.2013.28, 2013.

[18] P. Kanerva, Sparse Distributed Memory, MIT Press, Cambridge, MA, USA, ISBN 0262111322, 1988.

[19] L. Ferrone, F. M. Zanzotto, X. Carreras, Decoding Distributed Tree Structures, in: Statistical Language and Speech Processing - Third International Conference, SLSP 2015, Budapest, Hungary, November 24-26, 2015, Proceedings, 73–83, doi:\bibinfo{doi}{10.1007/978-3-319-25789-1_8}, URL http://dx.doi.org/10.1007/978-3-319-25789-1_8, 2015.

[20] J.-C. Chappelier, M. Rajman, et al., A Generalized CYK Algorithm for Parsing Stochastic CFG. .

[21] R. Sennrich, A CYK+ Variant for SCFG Decoding Without a Dot Chart, in: Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation, Association for Computational Linguistics, Doha, Qatar, 94–102, URL http://www.aclweb.org/anthology/W14-4011, 2014.

[22] G. E. Hinton, J. L. McClelland, D. E. Rumelhart, Distributed representations, in: D. E. Rumelhart, J. L. McClelland (Eds.), Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations, MIT Press, Cambridge, MA., 1986.

[23] Y. Bengio, A. Courville, P. Vincent, Representation Learning: A Review and New Perspectives, IEEE Trans. Pattern Anal. Mach. Intell. 35 (8) (2013) 1798–1828, ISSN 0162-8828, doi:\bibinfo{doi}{10.1109/TPAMI.2013.50}, URL http://dx.doi.org/10.1109/TPAMI.2013.50.

[24] J. Schmidhuber, Multi-column Deep Neural Networks for Image Classification, in: Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), CVPR '12, IEEE Computer Society, Washington, DC, USA, ISBN 978-1-4673-1226-4, 3642–3649, URL http://dl.acm.org/citation.cfm?id=2354409.2354694, 2012.

[25] A. Krizhevsky, I. Sutskever, G. E. Hinton, ImageNet Classification with Deep Convolutional Neural Networks, in: F. Pereira, C. J. C. Burges, L. Bottou, K. Q. Weinberger (Eds.), Advances in Neural Information Processing Systems 25, Curran Associates, Inc., 1097–1105, URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf, 2012.

[26] D. E. Rumelhart, G. E. Hinton, R. J. Williams, Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1, chap. Learning Internal Representations by Error Propagation, MIT Press, Cambridge, MA, USA, ISBN 0-262-68053-X, 318–362, URL http://dl.acm.org/citation.cfm?id=104279.104293, 1986.

[27] M. Collins, N. Duffy, Convolution Kernels for Natural Language, in: NIPS, 625–632, 2001.

[28] F. Aiolli, G. Da San Martino, A. Sperduti, Route kernels for trees, in: Proceedings of the 26th Annual International Conference on Machine Learning, ICML '09, ACM, New York, NY, USA, ISBN 978-1-60558-516-1, 17–24, doi:\bibinfo{doi}{10.1145/1553374.1553377}, URL http://doi.acm.org/10.1145/1553374.1553377, 2009.

[29] M. Collins, N. Duffy, New Ranking Algorithms for Parsing and Tagging: Kernels over Discrete Structures, and the Voted Perceptron, in: Proceedings of ACL02, 2002.

[30] S. V. N. Vishwanathan, A. J. Smola, Fast Kernels for String and Tree Matching, in: S. Becker, S. Thrun, K. Obermayer (Eds.), NIPS, MIT Press, ISBN 0-262-02550-7, 569–576, 2002.

[31] D. Kimura, T. Kuboyama, T. Shibuya, H. Kashima, A subpath kernel for rooted unordered trees, in: Proceedings of the 15th Pacific-Asia conference on Advances in knowledge discovery and data mining - Volume Part I, PAKDD'11, Springer-Verlag, Berlin, Heidelberg, ISBN 978-3-642-20840-9, 62–74, URL http://dl.acm.org/citation.cfm?id=2017863.2017871, 2011.

[32] M. P. Marcus, B. Santorini, M. A. Marcinkiewicz, Building a Large Annotated Corpus of English: The Penn Treebank, Computational Linguistics 19 (1993) 313–330.

[33] F. M. Zanzotto, L. Dell'Arciprete, Transducing Sentences to Syntactic Feature Vectors: an Alternative Way to "Parse"?, in: Proceedings of the Workshop on Continuous Vector Space Models and their Compositionality, 40–49, URL http://www.aclweb.org/anthology/W13-3205, 2013.

[34] F. M. Zanzotto, L. Ferrone, M. Baroni, When the Whole is Not Greater Than the Combination of Its Parts: A "Decompositional" Look at Compositional Distributional Semantics, Comput. Linguist. 41 (1) (2015) 165–173, ISSN 0891-2017, doi:\bibinfo{doi}{10.1162/COLI_a_00215}, URL http://dx.doi.org/10.1162/COLI_a_00215.

[35] F. F. L. R. L. Senay, G; Zanzotto, Predicting Embedded Syntactic Structures from Natural Language Sentences with Neural Network Approaches, in: Cognitive Computation: Integrating Neural and Symbolic Approaches, Workshop at NIPS, Montreal, Canada, 2105.