Generalising Recursive Neural Models by Tensor Decomposition

Daniele Castellana and Davide Bacciu
Dipartimento di Informatica
Università di Pisa
Largo B. Pontecorvo 3
Pisa, Italy
Email: {daniele.castellana,bacciu}@di.unipi.it

***Abstract*—Most machine learning models for structured data encode the structural knowledge of a node by leveraging simple aggregation functions (in neural models, typically a weighted sum) of the information in the node's neighbourhood. Nevertheless, the choice of simple context aggregation functions, such as the sum, can be widely sub-optimal. In this work we introduce a general approach to model aggregation of structural context leveraging a tensor-based formulation. We show how the exponential growth in the size of the parameter space can be controlled through an approximation based on the Tucker tensor decomposition. This approximation allows limiting the parameters space size, decoupling it from its strict relation with the size of the hidden encoding space. By this means, we can effectively regulate the trade-off between expressivity of the encoding, controlled by the hidden size, computational complexity and model generalisation, influenced by parameterisation. Finally, we introduce a new Tensorial Tree-LSTM derived as an instance of our framework and we use it to experimentally assess our working hypotheses on tree classification scenarios.**

I. INTRODUCTION

Trees are complex data which arise in multiple contexts: in Natural Language Processing, parse trees are used to represent natural language sentences; on the web, most of the data (e.g. HTML and XML documents) are represented using the Document Object Model i.e. a tree which represents the document structure. Regardless of the application domain, trees represent hierarchical information: they are composed of atomic entities, called *nodes*, combined together through a *parent-child* relationship which defines the tree structure.

The first approaches for learning from tree data were developed in the nineties; models, such as the *RAAM* [15] and the *LRAAM* [17], and theories, such as the *Backpropagation Through Structure* (BPTS) [10] and the *generalised recursive neuron* [18], were elaborated in those years. All these concepts have been unified in a general framework for processing of structured data [9] through, so-called, recursive learning models.

In this paper we focus on supervised tasks which admit a *recursive state representation*, i.e. it exists a hidden representation $\mathbf{H}$ which makes the output $\mathbf{Y}$ independent from the input $\mathbf{X}$ [9]. The hidden state $\mathbf{H}$, the input $\mathbf{X}$ and the output $\mathbf{Y}$ are trees with the same skeleton (classification tasks can be implemented as special case through a deterministic reduction of structure $\mathbf{Y}$ as in [9], see Figure 1). Moreover, the hidden representation $\mathbf{H}$ can be computed recursively on the structure. The latter property induces the concept of *context* of a node, i.e. the minimal set of nodes which contains all the necessary information to compute the target node hidden state. Clearly, the definition of a context is strictly related to the partial order used to define the recursion. For example, if the hidden state is computed recursively top-down, it means that the context of each tree node is its parent node; the root, which has no parent, is the base case of the recursion. Vice versa, if the recursion direction is bottom-up, the context of a node is the set of all its children; leaf nodes, which have no children, are the recursion base case.

The effect of the parsing direction on the properties and complexity of the model has been widely studied in the context of generative models [1], [2]. On the other hand, neural models tend to be less studied from this perspective. In particular, bottom-up neural models seem immune to the exponential growth of the parameters space with respect to the tree output degree which instead is considered a limiting factor in bottom-up generative models where approximations are required [2], [3], [7].

Within the scope of this paper, we attempt to fill this gap by showing that there exists a function which combines children hidden states through a tensor operator also in the case of L-ary trees and that this leads to an exponential number of learnable parameters. Moreover, we discuss how a formulation of the recursive neural state transition function in terms of such tensor operator is a generalisation of the weighted summation that is usually adopted in the literature for the same purpose.

A central contribution of this paper is an approximation of such tensor state transition function which leverages a tensor decomposition known as Tucker decomposition [20]. This approximation introduces a new hyper-parameter which controls the complexity of the parameters space of the transition function, breaking up the exponential dependence on hidden state size. This allows combining together the advantages of sum-based and tensor-based state transitions. In particular, we enable to deploy a model with a large hidden space (as it happens in models using simple sum-based aggregation) while computing state transitions trough a powerful aggregation function (as it happens in the full tensor model). Note that larger hidden states increase the ability of the neural model

978-1-7281-6926-2/20/$31.00 ©2020 IEEE

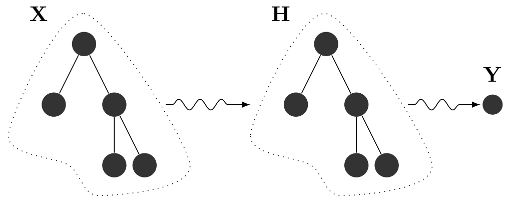

Fig. 1. An example of a tree sample $\mathbf{X}$, its hidden representation $\mathbf{H}$ and corresponding output label $\mathbf{Y}$.

of representing diverse and articulated substructures by means of the activation of its hidden neurons. On the other hand, the use of tensor-based aggregation allows the state transition function to explicitly capture direct dependencies between the encodings of the children of a node.

As part of this work, we also introduce two new Tree-LSTM based models using, respectively, the full tensor state transition function and its Tucker-based approximation. While tensor decompositions are already used in neural networks to compress full neural layers, this is the first work that, to the extent of our knowledge, leverages tensor decomposition as a more expressive alternative aggregation function for neurons in structured data processing. Among the others, we experimentally assess the advantages that this yields in terms of decoupling the parameter space size and the hidden state size, testing our models and a standard sum-based Tree-LSTM [19] on two challenging tree classification benchmarks.

II. Related Work

Using a tensor to compose input information has been applied successfully in other contexts: for example, in [4], the authors use tensor to fuse visual and textual representations; in [22], the authors use tensors to combine predicate, object and subject in event triplets. In both papers, the number of input to combine is fixed to two or three. Our proposal differs from these works because we use tensors to compose several inputs whose number is not fixed but depends on input data.

Regarding the processing of structured data, the first high-order neural network to process tree-structured contextual information has been discussed in [9]. However, the authors provided only the formulation for the case of binary trees, without any discussion on its properties or any experimental analysis. As far as we know, the only implementation of an high-order neural network for structured data is the one discussed in [16]. Again, only the case of binary trees is considered. Our framework extends this works taking into account L-ary trees and by introducing the use of tensor decompositions as a viable and expressive tradeoff between the simplicity of sum aggregation and the complexity of full tensor aggregations.

Tensor decompositions also aroused much interest in the community thanks to their compression ability. For example, in [6], [13], the authors show their use in compressing a fully connected layer with minor reduction in performance. The compression is obtained straightforwardly by storing the weight matrix using tensor decomposition. Differently, in this work, we propose tensor decompositions as an alternative to weighted sum input aggregation in recursive neurons, focusing in particular on the expressiveness of the newly proposed aggregation function.

III. Neural Tensor State Transition Function

The *state transition* function, in structured data processing, is the recursive function which computes the hidden representation of a node v given its label x_v and its associated context, e.g. the ordered set of its children $v_1, \dots, v_L$. In neural terms, the state transition function can be interpreted as the encoding of v corresponding to the vector of hidden neuron activations for the node. Without loss of generality, let us focus on a bottom-up state transition function $\hat{f}$, defined as [9]:

$$h_v = \hat{f}\left(x_v, h_{v1}, \dots . h_{vL}\right), \tag{1}$$

where h_v, x_v and $\{h_{v1}, \dots, h_{vL},\}$ are the hidden state, the input label and the context of target node v, respectively. The context contains L hidden states, where L is the maximum outdegree, i.e. the maximum number of child nodes. The term h_{vj} denotes the hidden state of the j-th child node of v.

The function $\hat{f}$ models the relationships between the pieces composing the structured sample (i.e. the nodes); $\hat{f}$ is typically unknown and, hence, the goal is estimating it from data. In neural models, $\hat{f}$ is typically approximated through the composition of a non-linear activation function σ and an affine map f:

$$h_v = \sigma\left(f\left(x_v, h_{v1}, \dots, h_{vL}\right)\right). \tag{2}$$

When dealing with flat (vectorial) data, f takes only one input (the label x_v since there is no context). In this case, the affine map is defined as $f(x_v) = Wx_v + b$ where W is a matrix and b a vector; equivalently, using the homogeneous coordinate $\bar{x}_v = [x_v; 1]$, $f(x_v) = \bar{W}\bar{x}_v$ where $\bar{W}$ is the *augmented matrix* obtained concatenating b to the last column of W. The operator $[a; n]$ adds the scalar n as the last element of the vector a.

With tree data, instead, we also consider the contribution of a context of size L. Therefore, the function

$$f : \mathbb{R}^m \times \overbrace{\mathbb{R}^c \times \dots \times \mathbb{R}^c}^{L \text{ times}} \to \mathbb{R}^c \tag{3}$$

becomes a multi-affine map, where m and c denote the size of the input label and hidden states, respectively (i.e. $x_v \in \mathbb{R}^m$ and $h_v \in \mathbb{R}^c$).

In this work, we assume that the multi-affine map f is represented as an *augmented tensor* $\mathbf{T} \in \mathbb{R}^{(m+1)\times(c+1)\cdots\times(c+1)\times c}$ (in the supplementary material[1] we prove that such tensor represents a multi-affine map). The last dimension of the tensor is associated to the output vector. As for the matrix case, each input vector v is represented using the homogeneous coordinate $\bar{v} = [v; 1]$.

[1] http://pages.di.unipi.it/castellana/publication/generalise_tree_rnn/

In the following, we use $f^{\mathbf{T}}$ to denote that f is a multi-affine map defined by the tensor $\mathbf{T}$. Therefore, we can rewrite Eq. (4) as:

$$\begin{aligned} h_v(k) &= \sigma(f^{\mathbf{T}}(x_v, h_{v1}, \dots, h_{v_L}))(k) = \\ &= \sigma\left(\sum_{i=1}^{m+1}\sum_{j_1=1}^{c+1}\cdots\sum_{j_L=1}^{c+1}\mathbf{T}(i, j_1, \dots, j_L, k)\times\right. \\ &\qquad \left.\times\ \bar{x}(i)\bar{h}_1(j_1)\dots\bar{h}_L(j_L)\right). \end{aligned} \tag{4}$$

We use brackets to denote entries of vectors, matrices and tensors: for instance, $W(i, j)$ denotes the element of the matrix W in the i-th row and j-th column.

Each entry of the tensor $\mathbf{T}$ should be learned during the training phase to approximate the true state transition function $\hat{f}$. Hence, we have a number of parameters which is equal to $(m + 1)(c + 1)^L c \approx O(mc^{(L+1)})$.

The exponential growth of parameters space with respect to outdegree L imposes strong limitations on the choice of the hidden state size (i.e. the number of hidden neurons in an internal layer). With a full tensorial formulation, even for small values of L (e.g. $L = 5$), a hidden state of size 10 would require about one million of parameters. This limitation can lead to poor performances since a small hidden state can be insufficient to encode all structural relationships and topological variations that a rich representation such as a tree can bring. For this reason, the prominent approach is to avoid the expressive but exponential formulation in Eq. (4) in favour of a simplified hidden state transition function which allows deploying a relatively large number of hidden neurons. In Section III-A, we formally show how such widely used transition function can be derived as an approximation of our general formulation in Eq. (4). In Section III-B, instead, we introduce a new approximated state transition function which builds on the Tucker tensor decomposition. This decomposition introduces a new hyper-parameter which breaks the exponential relation between the parameters space size and hidden state size. By this means, it allows deploying a large hidden state size without incurring in an exponential growth of the number of model parameters.

A. Weighted Sum Approximation

In most neural models for trees [9], [16], the state transition function is defined as:

$$h_v = \sigma\left(Wx_v + \sum_{j=1}^{L} U_j h_{vj} + b\right). \tag{5}$$

The hidden state h_v is obtained by applying a linear transformation W to the input label x_v and a linear transformation U_j to each child hidden state h_{vj}. The results of such transformations are summed together with a bias b, and fed to a non-linear function $\sigma(\cdot)$. The summation defines a multi-affine map f^+ which can be rewritten as:

$$f^+(x, h_1, \dots, h_L) = Wx + U_1h_1 + \dots + U_Lh_L + b.$$

In this formulation, each input vector (from label or context) contributes separately to the computation of the hidden state h_v and, therefore, it is completely independent from the others. This section aims to prove that f^+ can be obtained by imposing a full independence assumption among input vectors of the multi-affine map $f^{\mathbf{T}}$ in Eq. (4). To this end, we show there exists a tensor $\mathbf{T}^+$ such that:

$$\begin{aligned} f^+(x, h_1, \dots, h_L)(k) &= f^{\mathbf{T}^+}(x, h_1, \dots, h_L)(k) \\ &= \sum_{i=1}^{m+1}\sum_{j_1=1}^{c+1}\cdots\sum_{j_L=1}^{c+1}\mathbf{T}^+(i, j_1, \dots, j_L, k)\times \\ &\qquad \times\ \bar{x}(i)\bar{h}_1(j_1)\dots\bar{h}_L(j_L). \end{aligned} \tag{6}$$

For the sake of clarity, we have omitted the subscript related to current node v. In order to impose independence between input vectors, we must have all those entries of $\mathbf{T}^+$ which are multiplied with more than one input element set to 0. In fact, in f^+ above, there are no terms with inputs multiplied together. For example, the entry $\mathbf{T}^+(1, 1, \dots, 1, k)$ is multiplied by the first element of each input vector (i.e. $x(1), h_1(1), \dots, h_L(1)$) and therefore must be equal to 0.

Recalling that we are using homogeneous coordinates (hence the last element of each input vector is equal to 1), the only non-zero entries in $\mathbf{T}^+$ are the ones which have all input indexes (except one) pointing to the last entry of the input vector. For example, the entry $\mathbf{T}^+(i, c+1, \dots, c+1, k)$ multiplies solely to $x(i)$, since $\bar{h}_1(c+1) = \dots = \bar{h}_L(c+1) = 1$. Therefore, we obtain the following equation:

$$\begin{aligned} h(k) &= f^+(x, h_1, \dots, h_L)(k) = \\ &= \sum_{i=1}^{m}\mathbf{T}^+(i, c+1, \dots, c+1, k)x(i) + \\ &+ \sum_{j_1=1}^{c}\mathbf{T}^+(m+1, j_1, \dots, c+1, k)h_1(j_1) + \\ &\cdots \\ &+ \sum_{j_L=1}^{c}\mathbf{T}^+(m+1, c_1, \dots, j_L, k)h_L(j_L) + \\ &+ \mathbf{T}^+(m+1, c+1, \dots, c+1, k), \end{aligned} \tag{7}$$

where each row in the equation, except the last one, contains the application of a linear map; the last row contains a vector. Finally, Eq. (4) can be derived by setting each entry of the tensor to:

$$\mathbf{T}^+(i, j_1, \dots, j_L, k) = \\ = \begin{cases} W(i, k) & \text{if } j_1 = \dots = j_L = c+1 \\ U_l(j_l, k) & \text{if } i = m+1 \land j_{\neq l} = c+1 \\ b(k) & \text{if } i = m+1 \land j_1 = \dots = j_L = c+1 \\ 0 & \text{otherwise} \end{cases}. \tag{8}$$

This approximation requires $cm + Lc^2 + c \approx O(Lc^2)$ parameters, removing the exponential dependence between the number of parameters and the hidden state size.

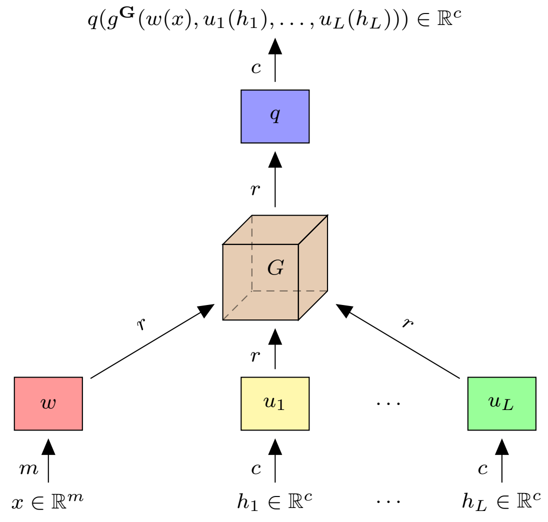

Fig. 2. A graphical representation of the HOSVD state transition function. The input $(x, h_1, \ldots, h_L)$ are transformed using the mode matrices $(w, u_1, \ldots, u_L)$ respectively. Then, the core tensor $\mathbf{G}$ is used to combine the transformed values; its output is fed to the last mode matrix q. Letters on links indicate the dimension of the vector sent along that direction.

B. HOSVD Approximation

The general tensor $\mathbf{T}$ in Eq. (4) can be approximated using a tensor factorisation. To this end, we propose to use the factorisation known as *Tucker* decomposition; it was originally developed by Tucker to obtain a method for identifying relations in a three-way tensor of psychometric data [20]. Later, the same idea has been generalised to a p-way tensor proving that it is a multi-linear generalisation of the matrix Singular Value Decomposition (SVD) [8]. Hence, the same decomposition is also called High-Order Singular Value Decomposition (HOSVD) [8].

Let $\mathbf{T} \in \mathbb{R}^{I_1 \times \cdots \times I_p}$ be a tensor with p dimensions, each entry $\mathbf{T}(i_1, \ldots, i_p)$ can be factorised as [8]:

$$\mathbf{T}(i_1, \ldots, i_p) = \\ = \sum_{j_1=1}^{R_1} \cdots \sum_{j_p=1}^{R_p} \mathbf{G}(j_1, \ldots, j_p) \prod_{z=1}^{p} U_z(j_z, i_z), \tag{9}$$

where $\mathbf{G}(j_1, \ldots, j_p)$ are elements of a new p-way tensor $\mathbf{G} \in \mathbb{R}^{R_1 \times \cdots \times R_p}$ called *core* tensor and $U_z(j_z, i_z)$ are elements of matrices $U_z \in \mathbb{R}^{R_z \times I_z}$ called *mode* matrices. The value R_z is the *rank* along the z-th dimension.

Ranks play a fundamental role since the number of parameters needed by the approximation is strictly related to it. In fact, we need $\prod_{z=1}^{p} R_z$ parameters to represent the core tensor and $\sum_{z=1}^{p} R_z \times I_z$ to represent the mode matrices. Therefore, the lower the ranks, the more we can compress the original tensor $\mathbf{T}$. Even if we have the guarantee that a HOSVD exists for every tensor [8], we have no guarantee on the rank values. In the worst case, $R_z = I_z$ and the core tensor has the same dimension as the initial one; hence, there is no compression.

Within the context of our neural functions, we would like to use the HOSVD to limit the parameters explosion of the general tensor state transition function. To this end, it is convenient to interpret the HOSVD in terms of mappings. Let $f^{\mathbf{T}}$ be the state transition function represented by a tensor $\mathbf{T}$, then we can decompose $f^{\mathbf{T}}$ by applying the HOSVD as

$$f^{\mathbf{T}}(x, h_1, \ldots, h_L) \approx q(g^{\mathbf{G}}(w(x), u_1(h_1), \ldots, u_L(h_L))), \tag{10}$$

where $w(\cdot)$ is the linear map represented by the mode matrix W along the first dimension (i.e. the dimension of the input label). The terms $u_1(\cdot), \ldots, u_L(\cdot)$ denote the linear maps represented by the mode matrices $U_1, \ldots, U_L$ along the other input dimensions (from the second one to to the second-to-last), while $g^{\mathbf{G}}$ is a multi-affine map represented by the core tensor $\mathbf{G}$. Finally, $q(\cdot)$ is the linear map represented by the mode matrix Q along the output dimension (the last one). We report in Figure 2 a graphical representation of Equation (10).

Thanks to the core tensor, the HOSVD approximation allows the state transition to combine input information without assuming full independence on them (as in the weighted sum approximation). Nevertheless, such combination is performed in a reduced space cutting down the number of parameters required. The value of the rank r, that we assume equal for each dimension, indicates the size of the reduced input space. Hence, the size of the tensor ultimately performing the computations needed for the state transition depends on the rank rather than on the size of the hidden states. More in detail: the mode matrix W requires $m \times r$ parameters; each of the mode matrices $U_1 \ldots, U_L$ and Q require $c \times r$ parameters; the core tensor $\mathbf{G}$ requires $r(r+1)^{(L+1)}$ parameters. Hence, the HOSVD approximation requires a total of $mr + (L+1)cr + r(r+1)^{(L+1)}$ parameters.

IV. TENSOR TREE-LSTM MODELS

The Tree-LSTM [19] is a generalisation of the standard LSTM architecture to tree-structured data. As in the standard LSTM, each LSTM unit contains: the *input gate*, the *output gate*, a *forget gate* for each child node, and the *memory cell*. For our purpose, we focus on the L-ary Tree-LSTM defined by the following equations [19]:

$$\begin{aligned}
i_v &= \sigma\left(W^i x_v + \sum_{j=1}^{L} U_j^i h_{vj} + b^i\right), \\
o_v &= \sigma\left(W^o x_v + \sum_{j=1}^{L} U_j^o h_{vj} + b^o\right), \\
u_v &= \sigma\left(W^u x_v + \sum_{j=1}^{L} U_j^u h_{vj} + b^u\right), \\
f_{vj} &= \sigma\left(W^f x_v + U_j^f h_{vj} + b_j^f\right), \\
c_v &= i_v \odot u_v + \sum_{j=1}^{L} f_{vj} \odot c_{vj}, \\
h_v &= o_v \odot \tanh(c_v);
\end{aligned} \tag{11}$$

where i_v, o_v, c_v are the input gate, the output gate and the memory cell, respectively, and f_{vj} is the forget gate associated to the j-th child node. Following [19], we assume that the forget gate f_{vj} depends only on hidden state h_{vj} rather than on the whole context. The term u_v is the value that we use to update the hidden state in a classic Tree-RNN.

The values i_v, o_v, u_v and f_{vj} are computed using a different neural components; among them, only i_v, o_v and u_v depend on the context. Observing Equation 11, it is clear that these three neural elements aggregate the contextual information using a sum-weighted function, as it happens in the state transition function defined in (5). For this reason, in the following, we will refer to this baseline model as Sum-LSTM.

In Section IV-A, we generalise the definition of Tree-LSTM using tensor-based aggregation functions to compute the quantities i_v, o_v, u_v. In Section IV-B, we propose a new tensor-based Tree-LSTM model leveraging the HOSVD approximation discussed above.

A. Full Tensor Tree-LSTM

The Full Tensor Tree-LSTM (Full-LSTM) is defined by extending the computation of the input gate i_v, the output gate o_v and the update value u_v using the general recursive state transition function introduced in (4), that is

$$\begin{aligned} i_v &= \sigma(f^{\mathbf{I}}(x_v, h_{v1}, \ldots, h_{v_L})), \\ o_v &= \sigma(f^{\mathbf{O}}(x_v, h_{v1}, \ldots, h_{v_L})), \\ u_v &= \sigma(f^{\mathbf{U}}(x_v, h_{v1}, \ldots, h_{v_L})), \end{aligned} \tag{12}$$

where $f^{\mathbf{I}}$, $f^{\mathbf{O}}$ and $f^{\mathbf{U}}$ are the multi-affine maps used to compute i_v, o_v and u_v, respectively. The remainder of the computations needed to complete the forward pass of the LSTM unit can be performed as in (11). All Full-LSTM parameters are learned using backpropagation. The update rules can be derived easily from the forward computation in (12) using automatic differentiation.

B. HOSVD Tree-LSTM

Starting from the tensor-extended formulation of the Full-LSTM, we are now allowed to introduce novel TreeLSTM models leveraging richer and, yet, computationally feasible alternatives to the sum-weighted state transition function. Our proposed HOSVD Tree-LSTM (Hosvd-LSTM) uses the HOSVD approximated state transition function in (10) to compute the input gate i_v, the output gate o_v and the update value u_v:

$$\begin{aligned} i_v &= \sigma(q^i(g^{\mathbf{G}^{\mathbf{i}}}(w^i(x), u_1^i(h_1), \ldots, u_L^i(h_L)))), \\ o_v &= \sigma(q^o(g^{\mathbf{G}^{\mathbf{o}}}(w^o(x), u_1^o(h_1), \ldots, u_L^o(h_L)))), \\ u_v &= \sigma(q^u(g^{\mathbf{G}^{\mathbf{u}}}(w^u(x), u_1^u(h_1), \ldots, u_L^u(h_L)))), \end{aligned} \tag{13}$$

where each gate has its own approximation parameters. The superscripts i, o and u indicate that a parameter is used to compute i_v, o_v and u_v respectively. As before, the remainder of the forward-pass computations can be performed as in (11). The Hosvd-LSTM update rules can be obtained by backpropagation as for the full tensor case.

V. Experimental Results

To assess the advantages that we obtain by breaking the relationship between the hidden state size and the parameter space size, we test the Tree-LSTM models introduced in Section IV on two classification problems.

A. Experimental Setting

In both tasks, tree data are obtained building the syntax tree of input strings according to a specific grammar. In these syntax trees, each label on internal nodes contains an operation that is applied to its child nodes. Hence, we allow models to use different parameters for each operator. It is worth to point out that this parameterisation is consistent with the tensor state transition function. In fact, if we use one-hot encoding to represent operators on internal input labels, the state transition tensor can be sliced obtaining one smaller tensor for each operator; hence, there are no parameters shared between different operators.

To make a fair comparison between different models, we use the label-dependent parametrisation also for the Sum-LSTM: in each internal node, we use the input label (i.e. the operator) to select a different LSTM cell. Also, we do not use word embedding to represent visible labels. In fact, a word embedding layer can map the input label in a space where the aggregation is easier/harder making the comparison unfair.

All the models are trained using AdaDelta algorithm [23] and therefore no learning rate is set. All the weights are initialised using Kaimining normal function [11]. We have implemented all the models using PyTorch [14] and Deep Graph Library [21]. The code can be found here[2].

All the reported results are averaged over three executions, to account for randomisation effects due to initialisation. Accuracy is used to assess model performance in both tasks. Also, we report the number of parameters used to represent a single aggregation function in each model.

B. Relations between Logical Sentences

The goal of the Logical Relations Trees (LRT) task is to determine what kind of logical consequence relation holds between two logical sentences. Each element in the dataset consists of a pair of logical sentences and a relation which exists among them. See [5] for more details on how the dataset is created.

Each logical sentence is represented with its syntax tree generated according to the grammar:

$$\mathrm{S} := \mathrm{S \ or \ S} \mid \mathrm{S \ and \ S} \mid \mathrm{not \ S} \mid \{a, b, c, d, e, f\}, \tag{14}$$

where $\{a, b, c, d, e, f\}$ are boolean variables and *or*, *and* , *not* the common logical operators. In Figure 3 we show an example of syntax tree. Due to the arity of the logical operator, the dataset contains only binary trees. Also, the boolean variable are represented using one-hot encoding.

The dataset is already divided into training and test set containing respectively 80% and 20% of the data [5]. We

[2] https://github.com/danielecastellana22/tensor-tree-nn

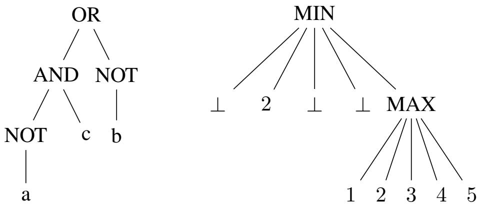

Fig. 3. Example of syntax tree for the LRT (on the left) and ListOps (on the right) dataset.

further sample 10% of the training set in order to build a validation set. During the training we use only trees which contain at most 4 logical operator [5], obtaining a training set of around 60k trees. Validation and test set contains around 20k and 36k tree pairs of all size respectively.

The architecture used to solve this task follows the one described in [5]. We use a single *encoder* model to compress each tree in a fixed size representation (i.e. the root hidden state) and a *classifier* model which predicts one of the seven relations starting from the encoder outputs.

In our experiments, we fix the classifier architecture. Following [5], the two trees representation are combined using a tensor and then a non-linear function is applied to the combination result. The non-linear function used is the leaky rectified linear function $\sigma(x) = \max(x, 0) + 0.01\min(x, 0)$. Finally, a softmax is used to predict the relation. The negative log-likelihood of the correct label with L2 regularisation is the objective function.

For each Tree-LSTM model used as encoder, we try different configurations. In the Full-LSTM and Sum-LSTM we vary the hidden state size $c \in \{3, 5, 10, 20, 50, 100\}$. In the Hosvd-LSTM, we try three different values of the rank r for each hidden state size c. Hence, we try the following configuration: $c = 10$ and $r \in \{3, 5, 7\}$, $c = 20$ and $r \in \{5, 10, 15\}$, $c = 50$ and $r \in \{10, 20, 30\}$, $c = 100$ and $r \in \{20, 50, 70\}$. All the other hyper-parameters are set to the same value for all models: the L2 regularisation weight is set to 0.01 and the batch size is set to 25.

1) Results: The results obtained on the validation set using different configurations of Full-LSTM, Sum-LSTM and Hosvd-LSTM are reported in Table I. Also, the same results are plotted in Figure 4 against the number of parameters. For the Hosvd-LSTM, we report only the best result obtained for each value of c; all the results obtained by other Hosvd-LSTM configurations are reported in the supplementary material.

The performances obtained by all models are similar for all configurations except the one which set $c = 10$. In that case, using the Full-LSTM we obtain an accuracy of 84.00%; with the same configuration, the Sum-LSTM reaches an accuracy of 64.79% while the Hosvd-LSTM reaches 76.91%. This suggests that the model with $c < 10$ are not able to solve the task because the hidden state size is too small to encode all the necessary structural information. Setting $c > 10$ yields to a data representation which is rich enough to solve the task using a simple combination of them such as the sum. Instead, setting $c = 10$, we observe the advantage of leveraging a more general aggregation function; the expressivity of such function allows us to achieve satisfactory results using a small hidden state size. The price to pay is, of course, a larger number of parameters. Also, it is interesting to compare the best configurations for each model. All models reach similar performances (see Table I), but the number of parameters used by the best Hosvd-LSTM is half of the parameters used by the best Sum-LSTM and a hundredth of the parameters used by the best Full-LSTM.

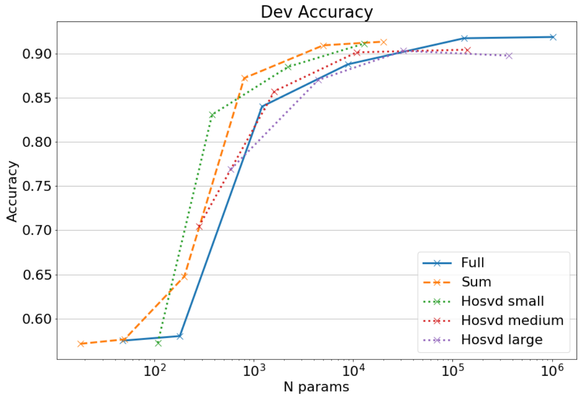

Fig. 4. Validation accuracy against number of parameters for each configuration of Full-LSTM, Sum-LSTM, Hosvd-LSTM on LRT dataset. For the Hosvd-LSTM, we plot one line for each rank values: *small*, *medium* and *large* indicates the smallest, the central and the largest rank values respectively.

In Table I, we also report the accuracy on the test set obtained by the best models configurations. The Full-LSTM and Sum-LSTM best configuration is $c = 100$; the Hosvd-LSTM best configuration is $c = 100$ and $r = 20$.

Note that our results cannot be compared with the ones obtained in [5] due to the different experimental settings, i.e. the different parsing strategies used to build trees and the encoding of input labels.

C. Operation on List

The goal of the ListOps task is to predict the solution of a sequence of summary operations on lists of single-digit integers, written in prefix notation. Each element in the dataset consists of a sequence of operations and its solution (which is also a single-digit integer). See [12] for more details on the dataset generation. Each sequence is represented as a tree using its syntax tree built according to the following grammar:

$$\begin{aligned} S := &\text{max[S,S,S,S,S]} \mid \text{min[S,S,S,S,S]} \mid \text{sum-m[S,S,S,S,S]} \mid \\ &\text{med[S,S,S,S,S]} \mid \{0, \dots, 9\} \mid \bot, \end{aligned} \tag{15}$$

where the bottom symbol $\bot$ represents a missing operand. We show an example of syntax tree in Figure 3. Each operation has at most five inputs; hence, trees in this dataset have a maximum output degree of five (i.e. $L = 5$). Also, the input

TABLE I

VALIDATION ACCURACY AND TEST ACCURACY ON LRT DATASET. ALL SCORES ARE AVERAGED OVER 3 RUNS (STD IN BRACKETS). BEST RESULTS ARE IN BOLD.

		Full-LSTM		Sum-LSTM		Hosvd-LSTM	
	c	N par.	Acc.	N par.	Acc.	N par.	Acc.
Validation	3	48	57.52 (0.17)	18	57.16 (0.25)	-	-
	5	180	58.03 (0.28)	50	57.66 (0.12)	-	-
	10	1210	84.00 (3.23)	200	64.79 (7.89)	588	76.91 (4.30)
	20	8820	88.77 (0.12)	800	87.21 (1.13)	4440	86.99 (0.46)
	50	130050	91.72 (0.78)	5000	90.92 (0.21)	31830	90.29 (0.32)
	100	1020100	**91.86** (0.24)	20000	**91.33** (0.51)	12820	**91.11** (0.76)
Test	100	1020100	**91.46** (0.10)	20000	91.03 (0.60)	12820	90.61 (0.89))

Note: for the Hosvd-LSTM, we report the results obtained using the best rank.

digit k is represented using a vector x of size 10 which has the first $k+1$ entries equals to 1 and the other equals to 0 (e.g. if $k=2$, $x=[1,1,1,0,0,0,0,0,0,0]$).

The dataset is already divided into training and test splits containing respectively 90% and 10% of the data [12]. We further sample 9% of the training set in order to build a validation set. Hence, we obtain a training set which contains around 80k trees; validation and test set contains around 20k and 36k trees respectively.

The architecture used to tackle this task is straightforward: we use a Tree-LSTM model to encode trees in a fixed size representation (i.e. the root hidden state) and then a classifier to predict the correct result starting from the tree representation. In our experiments, the classifier is fixed and it is implemented as a two-layer neural network with 20 hidden units for each layer. The cost function is the negative log-likelihood with respect to the ground-truth label with an L2 regularisation term.

For each Tree-LSTM model used as encoder, we test multiple configurations. In the Full-LSTM, we vary the hidden state size $c \in \{3,5,7\}$; in the Sum-LSTM, we vary $c \in \{25,88,214\}$; in the Hosvd-LSTM, we vary $c \in \{10,20,50\}$ and $r \in \{3,5,7\}$. The values of c and r are chosen to have models with the same number of parameters. All the other hyper-parameters are set to the same value for all models: the L2 regularisation weight is set to 0.01 and the batch size is set to 25.

D. Results

The results obtained on the validation set using different configurations of Full-LSTM, Sum-LSTM and Hosvd-LSTM are reported in Table II. Also, the same results are plotted in Figure 5 against the number of parameters. For the Hosvd-LSTM, we report only the best result obtained for each value of c; all the results obtained by other Hosvd-LSTM configurations are reported in the supplementary material. These results highlight the superior performance of Hosvd-LSTM compared to the other models. In particular, all Hosvd-LSTM configurations reach an accuracy that is higher than all configurations of the Sum-LSTM and Full-LSTM. This result becomes even more interesting if we compare the number of parameters used by the best Hosvd-LSTM model: it uses only 3k parameters, while Full-LSTM and Sum-LSTM are not able to reach the same performance even with more than 1 million parameters. The accuracy of Hosvd-LSTM decreases with increasing the number of parameters, an aspect which needs further investigations.

We argue that the Full-LSTM is not able to obtain satisfactory performance due to the small hidden state size. In fact, we vary $c \in \{3,5,7\}$; larger values of c would lead to a huge number of model parameter due to the maximum output degree of this dataset (i.e. $L=5$). Nevertheless, also the Sum-LSTM performs poorly on this dataset. We argue that this is due to the aggregation function which is too simple to implement operations in the dataset. Therefore, the Hosvd-LSTM outperforms the other two models thanks to the new approximation which allows combining a powerful aggregation function with a large hidden state size.

In Table II, we also report the accuracy on the test set obtained by the best model configurations. The Full-LSTM best configuration is $c=7$; the Sum-LSTM best configuration is $c=214$; the Hosvd-LSTM best configuration is $c=20$ and $r=3$.

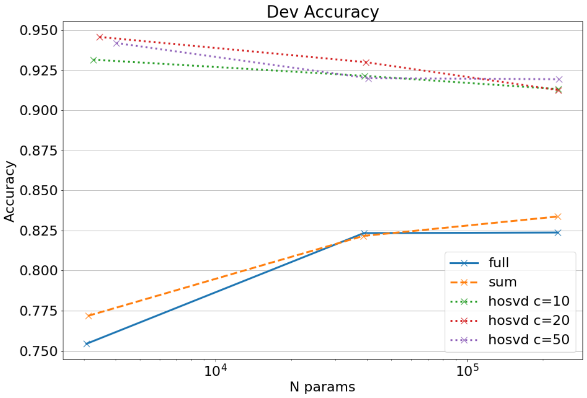

Fig. 5. Validation accuracy against number of parameters for each configuration of Full-LSTM, Sum-LSTM, Hosvd-LSTM on ListOps dataset. For the Hosvd-LSTM, we plot one line for each hidden state size c.

TABLE II

Validation accuracy and test accuracy on ListOps dataset. All scores are averaged over 3 runs (std in brackets). Best results are in bold.

	Full-LSTM			Sum-LSTM			Hosvd-LSTM		
	c	N par.	Acc.	c	N par.	Acc.	c	N par.	Acc.
Val.	3	3072	75.44 (1.18)	25	3125	77.19 (0.81)	10	3222	93.15 (0.61)
	5	38880	82.34 (0.61)	88	38720	82.16 (0.98)	20	3372	**94.57** (0.70)
	7	229376	**82.38** (0.82)	214	228980	**83.37** (1.18)	50	3822	94.19 (0.30)
Test	7	229376	82.02 (0.65)	214	228980	83.01 (1.06)	20	3372	**94.26** (0.48)

Note: for the Hosvd-LSTM, we report the results obtained using the best rank.

VI. Conclusion

In this paper, we have introduced a general framework to model context aggregation in tree-structured data based on a tensor formulation. In particular, we have shown that a Bottom-Up parsing direction of the input data leads to the definition of a tensor-based state transition function. The size of such tensor increases exponentially with respect to the maximum out-degree of input trees, imposing strong limitations on the choice of the hidden state size.

The central contribution of this paper is an approximation of such tensor which leverages the HOSVD tensor decomposition. This approximation allows breaking the exponential relation between the number of model parameters and the hidden state size without losing a tensor-based aggregation function. The results of our experiments show the advantage of our approximation, especially when the maximum out-degree of the tree increases as, for instance, in the ListOps dataset.

Such results pave the way to the definition of a tensor-based framework for structured data processing, stimulating the development of new adaptive tree models which leverage tensor factorisation. To this end, the next step would be to explore the applicability of other tensor factorisations, such as the canonical decomposition and the tensor train decomposition, and to study which are the bias introduced by each of them.

Ultimately, we would like to study how to include in our theoretical framework other tree model architectural biases. For example, we believe that positional stationarity can be implemented in our framework by adding a symmetry constraint on the state transition tensor.

Acknowledgment

The work is supported by project MIUR-SIR 2014 LIST-IT (grant n. RBSI14STDE).

References

[1] Davide Bacciu and Daniele Castellana. Learning Tree Distributions by Hidden Markov Models. In *Workshop on Learning and Automata (LearnAut'18)*, 2018.

[2] Davide Bacciu, Alessio Micheli, and Alessandro Sperduti. Compositional Generative Mapping for Tree-Structured Data - Part I: Bottom-Up Probabilistic Modeling of Trees. *IEEE Transactions on Neural Networks and Learning Systems*, 23(12):1987–2002, 2012.

[3] Davide Bacciu, Alessio Micheli, and Alessandro Sperduti. Compositional Generative Mapping for Tree-Structured Data - Part II: Topographic Projection Model. *IEEE Transactions on Neural Networks and Learning Systems*, 2013.

[4] Hedi Ben-Younes, Rémi Cadene, Matthieu Cord, and Nicolas Thome. Mutan: Multimodal tucker fusion for visual question answering. In *Proceedings of the IEEE international conference on computer vision*, pages 2612–2620, 2017.

[5] Samuel R. Bowman, Christopher Potts, and Christopher D. Manning. Recursive Neural Networks Can Learn Logical Semantics. *Proceedings of the 3rd Workshop on Continuous Vector Space Models and their Compositionality*, pages 12–21, 2015.

[6] Giuseppe Giovanni Calvi and et al. Tucker tensor layer in fully connected neural networks. *CoRR*, abs/1903.06133, 2019.

[7] Daniele Castellana and Davide Bacciu. Bayesian Tensor Factorisation for Bottom-up Hidden Tree Markov Models. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 7 2019.

[8] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. A Multilinear Singular Value Decomposition. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1253–1278, 1 2000.

[9] Paolo Frasconi, Marco Gori, and Alessandro Sperduti. A general framework for adaptive processing of data structures. *IEEE Transactions on Neural Networks*, 9(5):768–786, 1998.

[10] C. Goller and A. Kuchler. Learning task-dependent distributed representations by backpropagation through structure. *Proceedings of International Conference on Neural Networks (ICNN'96)*, 1:347–352, 1996.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE ICCV 2015*, 2015.

[12] Nikita Nangia and Samuel Bowman. ListOps: A Diagnostic Dataset for Latent Tree Learning. In *Proceedings of NAACL 2018: Student Research Workshop*, pages 92–99, Stroudsburg, PA, USA, 2018. Association for Computational Linguistics.

[13] Alexander Novikov, Dmitry Podoprikhin, Anton Osokin, and Dmitry Vetrov. Tensorizing neural networks. In *Proceedings of NIPS 2015 - Volume 1*, NIPS'15, page 442–450, Cambridge, MA, USA, 2015. MIT Press.

[14] Adam Paszke and et al. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[15] Jordan B. Pollack. Recursive distributed representations. *Artificial Intelligence*, 1990.

[16] Richard Socher, Alex Perelygin, and Jy Wu. Recursive deep models for semantic compositionality over a sentiment treebank (RNTN). *Proceedings of EMNLP 2013*, pages 1631–1642, 2013.

[17] Alessandro Sperduti. Labeling RAAM. Technical report, Connection Science, 1994.

[18] Alessandro Sperduti and Antonina Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 8(3):714–735, 1997.

[19] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. *Proceedings of 53rd ACL annual meeting and the 7th IJCNLP (Volume 1: Long Papers)*, pages 1556–1566, 2015.

[20] Ledyard R Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, 1966.

[21] Minjie Wang and et al. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[22] Noah Weber, Niranjan Balasubramanian, and Nathanael Chambers. Event representations with tensor-based compositions. In *AAAI*, 2017.

[23] Matthew D Zeiler. ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.