Exact Solutions for Recursive Principal Components Analysis of Sequences and Trees

Alessandro Sperduti

Department of Pure and Applied Mathematics, University of Padova, Italy
sperduti@math.unipd.it

Abstract. We show how a family of exact solutions to the Recursive Principal Components Analysis learning problem can be computed for sequences and tree structured inputs. These solutions are derived from eigenanalysis of extended vectorial representations of the input structures and substructures. Experimental results performed on sequences and trees generated by a context-free grammar show the effectiveness of the proposed approach.

1 Introduction

The idea to extend well known and effective mathematical tools, such as Principal Component Analysis, to the treatment of structured objects has been pursued directly (e.g. [6,5]) or indirectly (e.g. [2,3,1]) by many researchers. The aim is to devise tools for embedding discrete structures into vectorial spaces, where all the classical pattern recognition and machine learning methods can be applied.

Up to now, however, at the best of our knowledge no exact solution for Recursive Principal Components Analysis has been devised. Here we define sufficient conditions that allow us to construct a family of exact solutions to this problem. Experimental results on significantly large structures demonstrate the effectiveness of our proposal.

2 Recursive Principal Components Analysis

In [6] a connection between Recursive Principal Components Analysis and the representations developed by a simple linear recurrent neural network has been suggested. Specifically, a model with the following linear dynamics is considered:

$$\mathbf{y}_t = \mathbf{W}_{\mathbf{x}}\mathbf{x}_t + \sqrt{\alpha}\mathbf{W}_{\mathbf{y}}\mathbf{y}_{t-1} \tag{1}$$

where t is a discrete time index, $\mathbf{x}_t$ is a zero-mean input vector, $\mathbf{y}_t$ is an output vector, $\alpha \in [0, 1]$ is a gain parameter which modulates the importance of the past history, i.e. $\mathbf{y}_{t-1}$, with respect to the current input $\mathbf{x}_t$, $\mathbf{W}_{\mathbf{x}}$ and $\mathbf{W}_{\mathbf{y}}$ are the matrices of synaptic efficiencies, which correspond to feed-forward and recurrent connections, respectively. In [6] it is assumed that the time series $(\mathbf{x}_t)$ is bounded and stationary. The model is trained using an extension of the Oja's rule, however there is no proof that the proposed learning rule converges to the recursive principal components.

S. Kollias et al. (Eds.): ICANN 2006, Part I, LNCS 4131, pp. 349–356, 2006.

Here, for the sake of clearness, we consider the special case where $\alpha = 1$. The construction we are going to derive does not depend on this specific setting. We focus on the following equations

$$\mathbf{x}_t = \mathbf{W}_{\mathbf{x}}^{\mathsf{T}} \mathbf{y}_t \tag{2}$$

$$\mathbf{y}_{t-1} = \mathbf{W}_{\mathbf{x}} \mathbf{x}_{t-1} + \mathbf{W}_{\mathbf{y}} \mathbf{y}_{t-2} \tag{3}$$

$$= \mathbf{W}_{\mathbf{y}}^{\mathsf{T}} \mathbf{y}_t \tag{4}$$

which have to be satisfied in order for the network to compute recursive principal components, i.e. the sequence is first encoded using eq. (1), and then, starting from an encoding, it should be possible to reconstruct backwards the original sequence using the transposes of $\mathbf{W}_{\mathbf{x}}$ and $\mathbf{W}_{\mathbf{y}}$. In fact, the aim of recursive principal component is to find a low-dimensional representation of the input sequence (or tree) such that the expected residual error is as small as possible.

3 Exact Solutions for Sequences

For $t = 1, \ldots, T$ the following equations should be satisfied

$$\mathbf{x}_t = \mathbf{W}_{\mathbf{x}}^{\mathsf{T}} \underbrace{(\mathbf{W}_{\mathbf{x}} \mathbf{x}_t + \mathbf{W}_{\mathbf{y}} \mathbf{y}_{t-1})}_{\mathbf{y}_t} = \mathbf{W}_{\mathbf{x}}^{\mathsf{T}} \mathbf{W}_{\mathbf{x}} \mathbf{x}_t + \mathbf{W}_{\mathbf{x}}^{\mathsf{T}} \mathbf{W}_{\mathbf{y}} \mathbf{y}_{t-1} \tag{5}$$

$$\mathbf{y}_{t-1} = \mathbf{W}_{\mathbf{y}}^{\mathsf{T}} \overbrace{(\mathbf{W}_{\mathbf{x}} \mathbf{x}_t + \mathbf{W}_{\mathbf{y}} \mathbf{y}_{t-1})} = \mathbf{W}_{\mathbf{y}}^{\mathsf{T}} \mathbf{W}_{\mathbf{x}} \mathbf{x}_t + \mathbf{W}_{\mathbf{y}}^{\mathsf{T}} \mathbf{W}_{\mathbf{y}} \mathbf{y}_{t-1} \tag{6}$$

where it is usually assumed that $\mathbf{y}_0 = \mathbf{0}$. Sufficient conditions for the above equations to be satisfied for $t = 1, \ldots, T$ are as follows:

$$\mathbf{W}_{\mathbf{x}}^{\mathsf{T}} \mathbf{W}_{\mathbf{x}} \mathbf{x}_t = \mathbf{x}_t \tag{7}$$

$$\mathbf{W}_{\mathbf{y}}^{\mathsf{T}} \mathbf{W}_{\mathbf{x}} \mathbf{x}_t = \mathbf{0} \tag{8}$$

$$\mathbf{W}_{\mathbf{y}}^{\mathsf{T}} \mathbf{W}_{\mathbf{y}} \mathbf{y}_{t-1} = \mathbf{y}_{t-1} \tag{9}$$

$$\mathbf{W}_{\mathbf{x}}^{\mathsf{T}} \mathbf{W}_{\mathbf{y}} \mathbf{y}_{t-1} = \mathbf{0} \tag{10}$$

From eqs. (8) and (10) we deduce that the columns of $\mathbf{W}_{\mathbf{x}}$ must be orthogonal to the columns of $\mathbf{W}_{\mathbf{y}}$. Thus, the set of vectors $\mathbf{v}_t = \mathbf{W}_{\mathbf{x}} \mathbf{x}_t$ must be orthogonal to the vectors $\mathbf{z}_t = \mathbf{W}_{\mathbf{y}} \mathbf{y}_t$, since the vectors $\mathbf{x}_t$ and $\mathbf{y}_t$ are projected onto orthogonal subspaces of the same space $\mathbb{S}$. From this observation it is not difficult to figure out how to define a partition of $\mathbb{S}$ into two orthogonal subspaces. Let s be the dimensionality of $\mathbb{S}$. Since $\mathbf{v}_t$ represents the current input information, while $\mathbf{z}_t$ represents the "history" of the input, we can assign the first k dimensions of $\mathbb{S}$ to encode vectors $\mathbf{v}_t$, and the remaining $(s - k)$ dimensions to encode vectors $\mathbf{z}_t$. This can be done by setting to 0 the last $(s - k)$ components for vectors $\mathbf{v}_t$, while setting to 0 the first k components for vectors $\mathbf{z}_t$. Moreover, if we chose k to be equal to the dimension of $\mathbf{x}_t$ and $s = k(q+1)$, where q is the depth of the memory we want to have in our system, we can define vectors $\mathbf{v}_t \in \mathbb{S}$ as

$$\mathbf{v}_t^{\mathsf{T}} \equiv [\mathbf{x}_t^{\mathsf{T}}, \underbrace{\mathbf{0}^{\mathsf{T}}, \ldots, \mathbf{0}^{\mathsf{T}}}_{q}] \tag{11}$$

where $\mathbf{0}$ is the vector with all zeros of dimension k, and vectors $\mathbf{z}_t \in \mathbb{S}$, with $t \leq q$ to explicitly represent the history, according to the following scheme

$$\mathbf{z}_t^\mathsf{T} \equiv [\mathbf{0}^\mathsf{T}, \mathbf{x}_t^\mathsf{T}, \ldots, \mathbf{x}_1^\mathsf{T}, \underbrace{\mathbf{0}^\mathsf{T}, \ldots, \mathbf{0}^\mathsf{T}}_{(q-t)}] \tag{12}$$

Using this encoding, $\mathbf{y}_t \in \mathbb{S}$ is defined as

$$\mathbf{y}_t^\mathsf{T} = \mathbf{v}_t^\mathsf{T} + \mathbf{z}_{t-1}^\mathsf{T} \;=\; [\mathbf{x}_t^\mathsf{T}, \ldots, \mathbf{x}_1^\mathsf{T}, \underbrace{\mathbf{0}^\mathsf{T}, \ldots, \mathbf{0}^\mathsf{T}}_{(q-t+1)}]. \tag{13}$$

Recalling that $\mathbf{z}_t = \mathbf{W}_\mathbf{y}\mathbf{y}_t$, it becomes evident that the function implemented by $\mathbf{W}_\mathbf{y}$ is just a shift of k positions of the $\mathbf{y}_t$ vector, i.e.

$$\mathbf{W}_\mathbf{y} \equiv \begin{bmatrix} \mathbf{0}_{k\times kq} & \mathbf{0}_{k\times k} \\ \mathbf{I}_{kq\times kq} & \mathbf{0}_{kq\times k} \end{bmatrix}, \tag{14}$$

and recalling that $\mathbf{v}_t = \mathbf{W}_\mathbf{x}\mathbf{x}_t$, we have

$$\mathbf{W}_\mathbf{x} \equiv \begin{bmatrix} \mathbf{I}_{k\times k} \\ \mathbf{0}_{kq\times k} \end{bmatrix}. \tag{15}$$

It can be readily verified that the defined vectors and matrices satisfy eqs. (7)-(10). In fact, eq. (7) is satisfied since $\mathbf{W}_\mathbf{x}^\mathsf{T}\mathbf{W}_\mathbf{x} = \mathbf{I}_{k\times k}$, while eqs. (8) and (10) are satisfied because by construction the columns of $\mathbf{W}_\mathbf{x}$ are orthogonal to columns of $\mathbf{W}_\mathbf{y}$, and finally eq. (9) is satisfied since $\mathbf{W}_\mathbf{y}^\mathsf{T}\mathbf{W}_\mathbf{y} = \begin{bmatrix} \mathbf{I}_{kq\times kq} & \mathbf{0}_{kq\times k} \\ \mathbf{0}_{k\times kq} & \mathbf{0}_{k\times k} \end{bmatrix}$ and all $\mathbf{y}_t,\ t = 0, \ldots, T-1$, have the last k components equal to 0.

The problem with this encoding is that s is too large, and information is not compressed at all. This problem can be easily fixed by computing the principal components of vectors $\mathbf{y}_t$.

Let

$$\bar{\mathbf{y}} = \frac{1}{T}\sum_{i=1}^{T}\mathbf{y}_i \quad \text{and} \quad \mathbf{C}_\mathbf{y} \;=\; \frac{1}{T}\sum_{i=1}^{T}(\mathbf{y}_i - \bar{\mathbf{y}})(\mathbf{y}_i - \bar{\mathbf{y}})^\mathsf{T} \;=\; \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\mathsf{T} \tag{16}$$

where $\mathbf{\Lambda}$ is a diagonal matrix with elements equal to the eigenvalues of $\mathbf{C}_\mathbf{y}$, and $\mathbf{U}$ is the matrix obtained by collecting by column all the corresponding eigenvectors. Let $\widetilde{\mathbf{U}} \in \mathbb{R}^{s\times p}$ be the matrix obtained by $\mathbf{U}$ removing all the eigenvectors corresponding to null eigenvalues. Notice that in some cases we can have $p \ll s$. Then, we have

$$\tilde{\mathbf{y}}_t \;=\; \widetilde{\mathbf{U}}^\mathsf{T}(\mathbf{y}_t - \bar{\mathbf{y}}) \quad \text{and} \quad \mathbf{y}_t \;=\; \widetilde{\mathbf{U}}\tilde{\mathbf{y}}_t + \bar{\mathbf{y}} \tag{17}$$

and using eq. (2)

$$\begin{aligned}
\tilde{\mathbf{y}}_t &= \widetilde{\mathbf{U}}^\mathsf{T}(\mathbf{W}_\mathbf{x}\mathbf{x}_t + \mathbf{W}_\mathbf{y}\mathbf{y}_{t-1} - \bar{\mathbf{y}}) && (18)\\
&= \widetilde{\mathbf{U}}^\mathsf{T}\mathbf{W}_\mathbf{x}\mathbf{x}_t + \widetilde{\mathbf{U}}^\mathsf{T}\mathbf{W}_\mathbf{y}\mathbf{y}_{t-1} - \widetilde{\mathbf{U}}^\mathsf{T}\bar{\mathbf{y}} && (19)\\
&= \widetilde{\mathbf{U}}^\mathsf{T}\mathbf{W}_\mathbf{x}\mathbf{x}_t + \widetilde{\mathbf{U}}^\mathsf{T}\mathbf{W}_\mathbf{y}(\widetilde{\mathbf{U}}\tilde{\mathbf{y}}_{t-1} + \bar{\mathbf{y}}) - \widetilde{\mathbf{U}}^\mathsf{T}\bar{\mathbf{y}} && (20)\\
&= \begin{bmatrix} \widetilde{\mathbf{U}}^\mathsf{T}\mathbf{W}_\mathbf{x} & \widetilde{\mathbf{U}}^\mathsf{T}(\mathbf{W}_\mathbf{y} - \mathbf{I}_{s\times s})\bar{\mathbf{y}} \end{bmatrix}\begin{bmatrix} \mathbf{x}_t \\ 1 \end{bmatrix} + \widetilde{\mathbf{U}}^\mathsf{T}\mathbf{W}_\mathbf{y}\widetilde{\mathbf{U}}\tilde{\mathbf{y}}_{t-1} && (21)\\
&= \widetilde{\mathbf{W}}_\mathbf{x}\tilde{\mathbf{x}}_t + \widetilde{\mathbf{W}}_\mathbf{y}\tilde{\mathbf{y}}_{t-1}, && (22)
\end{aligned}$$

where $\tilde{\mathbf{x}}_t \equiv \begin{bmatrix} \mathbf{x}_t \\ 1 \end{bmatrix}$, $\widetilde{\mathbf{W}}_{\mathbf{x}} \equiv \begin{bmatrix} \widetilde{\mathbf{U}}^{\mathsf{T}}\mathbf{W}_{\mathbf{x}} & \widetilde{\mathbf{U}}^{\mathsf{T}}(\mathbf{W}_{\mathbf{y}} - \mathbf{I}_{s\times s})\bar{\mathbf{y}} \end{bmatrix} \in \mathbb{R}^{p\times(k+1)}$ and $\widetilde{\mathbf{W}}_{\mathbf{y}} \equiv \widetilde{\mathbf{U}}^{\mathsf{T}}\mathbf{W}_{\mathbf{y}}\widetilde{\mathbf{U}} \in \mathbb{R}^{p\times p}$.

3.1 Trees

When considering trees, the encoding used for $\mathbf{z}$ vectors is a bit more complex. First of all, let us illustrate what happens for binary complete trees. Then, we will generalize the construction to (in)complete b-ary trees. For $b = 2$, we have the following linear model

$$\mathbf{y}_u = \mathbf{W}_{\mathbf{x}}\mathbf{x}_u + \mathbf{W}_{\mathbf{l}}\mathbf{y}_{ch_l[u]} + \mathbf{W}_{\mathbf{r}}\mathbf{y}_{ch_r[u]} \tag{23}$$

where u is a vertex of the tree, $ch_l[u]$ is the left child of u, $ch_r[u]$ is the right child of u, $\mathbf{W}_{\mathbf{l}}, \mathbf{W}_{\mathbf{r}} \in \mathbb{R}^{s\times s}$. In this case, the basic idea is to partition $\mathbb{S}$ according to a perfectly balanced binary tree. More precisely, each vertex u of the binary tree is associated to a binary string $id(u)$ obtained as follows: the binary string "1" is associated to the root of the tree. Any other vertex has associated the string obtained by concatenating the string of its parent with the string "0" if it is a left child, "1" otherwise. Then, all the dimensions of $\mathbb{S}$ are partitioned in s/k groups of k dimensions. The label associated to vertex v is stored into the j-th group, where j is the integer represented by the binary string $id(u)$. E.g. the label of the root is stored into group 1, since $id(root) =$"1", the label of the vertex which can be reached by the path ll starting from the root is stored into group 4, since $id(u) =$"100", while the label of the vertex reachable through the path rlr is stored into group 13, since $id(u) =$"1101". Notice that, if the input tree is not complete, the components corresponding to missing vertexes are set to be equal to 0. Using this convention, vectors $\mathbf{v}_u$ maintain the definition of eq. (11), and are used to store the current input label, i.e. the label associated to the root of the (sub)tree presented up to now as input, while vectors $\mathbf{z}_u$ are defined according to the scheme described above, with the difference that the first k components (i.e., the ones storing the label of the root) are set to 0.

Matrices $\mathbf{W}_{\mathbf{l}}$ and $\mathbf{W}_{\mathbf{r}}$ are defined as follows. Both matrices are composed of two types of blocks, i.e. $\mathbf{I}_{k\times k}$ and $\mathbf{0}_{k\times k}$. Matrix $\mathbf{W}_{\mathbf{l}}$ has to implement a push-left operation, i.e. the tree $\mathcal{T}$ encoded by a vector $\mathbf{y}_{root(\mathcal{T})}$ has to become the left child of a new node u whose label is the current input $\mathbf{x}_u$. Thus $root(\mathcal{T})$ has to become the left child of u and also all the other vertexes in $\mathcal{T}$ have their position redefined accordingly. From a mathematical point of view, the new position of any vertex a in $\mathcal{T}$ is obtained by redefining $id(a)$ as follows: *i)* the most significative bit of $id(a)$ is set to "0", obtaining the string $id_0(a)$; *ii)* the new string $id_{new}(a) =$"1"$+id_0(a)$ is defined, where $+$ is the string concatenation operator. If $id_{new}(a)$ represents a number greater than s/k then this means that the vertex has been pushed outside the available memory, i.e. the vertex a is *lost*. Consequently, groups which correspond to *lost* vertexes have to be annilated. Thus, $\mathbf{W}_{\mathbf{l}}$ is composed of $(q+1)\times(q+1)$ blocks, all of type $\mathbf{0}_{k\times k}$, except for the blocks in row $id_{new}(a)$ and column $id(a)$, with $id_{new}(a) \leq s/k$, where a block $\mathbf{I}_{k\times k}$ is placed. Matrix $\mathbf{W}_{\mathbf{r}}$ is defined similarly: it has to implement a push-right operation, i.e.: *i)* the most significative bit of $id(a)$ is set to "1", obtaining the string $id_1(a)$; *ii)* the new string $id_{new}(a) =$"1"$+id_1(a)$ is defined. Matrix $\mathbf{W}_{\mathbf{x}}$ is defined as in eq. (15).

Generalization of the above scheme for complete b-ary trees is not difficult. The linear model becomes

$$\mathbf{y}_u = \mathbf{W}_{\mathbf{x}}\mathbf{x}_u + \sum_{c=0}^{b-1} \mathbf{W}_{\mathbf{c}}\mathbf{y}_{ch_c[u]} \tag{24}$$

where $ch_c[u]$ is the $c+1$-th child of u, and a matrix $\mathbf{W}_{\mathbf{c}}$ is defined for each child. The string associated to each vertex is defined on the alphabet {"0","1",...,"b-1"}, since there are b children. The symbol $b-1$ is associated with the root and b push operations have to be defined. The new string associated to any vertex a in $\mathcal{T}$, after a c-push operation, is obtained by redefining $id(a)$ as follows: *i)* the most significative symbol of $id(a)$ is set to c, obtaining the string $id_c(a)$; *ii)* the new string $id_{new}(a)$ ="b-1"+$id_c(a)$ is defined. E.g., if $b = 5$ and c ="3", then *i)* the most significative symbol of $id(a)$ is set to "3", obtaining the string $id_3(a)$; *ii)* the new string $id_{new}(a)$ ="b-1"+$id_3(a)$ is defined. Matrix $\mathbf{W}_{\mathbf{c}}$ is defined by placing blocks $\mathbf{I}_{k\times k}$ in positions $(id_{new}(a), id(a))$ only if $id_{new}(a) \leq s/k$, where $id_{new}(a)$ is interpreted as a number represented in base b. Performing the eigenspace analysis, we obtain

$$\tilde{\mathbf{y}}_u = \widetilde{\mathbf{W}}_{\mathbf{x}}\tilde{\mathbf{x}}_u + \sum_{c=0}^{b-1} \widetilde{\mathbf{W}}_{\mathbf{c}}\tilde{\mathbf{y}}_{ch_c[u]}, \tag{25}$$

where $\tilde{\mathbf{x}}_u \equiv \begin{bmatrix} \mathbf{x}_u \\ 1 \end{bmatrix}$, $\widetilde{\mathbf{W}}_{\mathbf{x}} \equiv \begin{bmatrix} \widetilde{\mathbf{U}}^{\mathsf{T}}\mathbf{W}_{\mathbf{x}} & \widetilde{\mathbf{U}}^{\mathsf{T}}(\sum_{c=0}^{b-1}\mathbf{W}_{\mathbf{c}} - \mathbf{I}_{s\times s})\bar{\mathbf{y}} \end{bmatrix} \in \mathbb{R}^{p\times(k+1)}$ and $\widetilde{\mathbf{W}}_{\mathbf{c}} \equiv \widetilde{\mathbf{U}}^{\mathsf{T}}\mathbf{W}_{\mathbf{c}}\widetilde{\mathbf{U}} \in \mathbb{R}^{p\times p},\ c = 0,\ldots,b-1$.

A problem in dealing with complete trees is that very soon there is a combinatorial explosion of the number of paths to consider, i.e. in order for the machine to deal with moderately deep trees, a huge value for s needs to be used. In practical applications, however, the observed trees tend to follow a specific generative model, and thus there may be many topologies which are never, or very seldomly, generated. For this reason we suggest to use the following approach. Given a set of trees $\mathbf{T}$, the optimized graph $G_{\mathbf{T}}$ [4] is obtained by joining all the trees in such a way that any (sub)tree in $\mathbf{T}$ is represented only once. The optimized graph $G_{\mathbf{T}}$, which is a DAG, is then visited bottom-up, generating for each visited vertex v the set of id strings associated to the tree rooted in v, thus simulating all the different push operations which should be performed when presenting the trees in $\mathbf{T}$ to the machine. Repeated id strings are removed. The obtained set P is then used to define the state space of the machine: each string is associated to one group of k coordinates. In this way, only paths which appear in the set $\mathbf{T}$ (including all subtrees) are represented, thus drastically reducing the size of s, which will be equal to $|P| \times k$. One drawback of this approach is that if a new tree with "unknown" paths is presented to the machine, the vertexes which are reached by those paths are lost.

A final practical consideration concerns the observation that by introducing a dummy vector $\mathbf{y}_{dummy} = -\sum_i \mathbf{y}_i$, eq. (25) is simplified since $\bar{\mathbf{y}} = \mathbf{0}$, and the corresponding derived weight matrices appear to be much more effective. In the experiments reported in this paper, we have used this trick.

Sentence	*Noun Phrase*	*Verb Phrase*	*Prepositional Phrase*	*Adjectival Phrase*
s → np vp s → np V	np → D ap np → D N np → np pp	vp → V np vp → V pp	pp → P np	ap → A ap ap → A N

Fig. 1. Context-Free Grammar used in the experiments

4 Experiments

For testing our approach, we have considered the context-free grammar shown in Figure 1, and already used by Pollack [2]. Sequences and corresponding parse trees are randomly generated from the grammar, rejecting sequences longer than 30 items. In all, 177 distinct sequences (and corresponding parse trees) are generated. Among them, 101 sequences are randomly selected for training, and the remaining 76 sequences are used for testing the generalization ability of the machine. In Table 1 we have reported some statistics about the data. The dataset for trees is obtained by considering the parse trees corresponding to the selected sequences.

Sequences are composed of terminal symbols, which are encoded by 5-dimensional "one-hot" vectors (i.e. $k = 5$). Since there are up to 30 items in a sequence, $s = 150$. Trees also include nonterminal symbols. In this case, symbols are represented by 6-dimensional vectors (i.e. $k = 6$), where the first component is 0 for terminal symbols and 3 for nonterminal symbols, while the remaining 5 components follow a "one-hot" coding scheme. The state space $\mathbb{S}$ is obtained by computing the optimization graph for the training set and generating all the possible paths following the procedure described at the end of Section 3.1. In all, 351 distinct paths where generated, leading to a final dimension for the state space equal to $s = 6 \times 351 = 2106$. The computation of the optimized graph also allowed the identification of 300 unique (sub)trees, thus allowing us just to consider the same number of different non-null states. The dummy state $\mathbf{y}_{dummy}$ is used for both datasets to get zero-mean vectors.

The spectral analysis for sequences required 0.1 cpu/sec on an Athlon 1900+ based computer, while it required 377.77 cpu/sec for trees. Results are shown in Figure 2. In Figure 3 we have reported for the sequence dataset the error in label decoding (left) and the mean square error for labels (right) plotted versus the number of used components.

Table 1. Statistical properties of the datasets

Dataset/Split	# examples	Max. length (depth)	Max. number item per example	Tot. number items	Tot. number unique (sub)trees
Sequences/Training	101	30	30	1463	-
Sequences/Test	76	30	30	1158	-
Tree/Training	101	14	59	2825	300
Tree/Test	76	15	59	2240	242

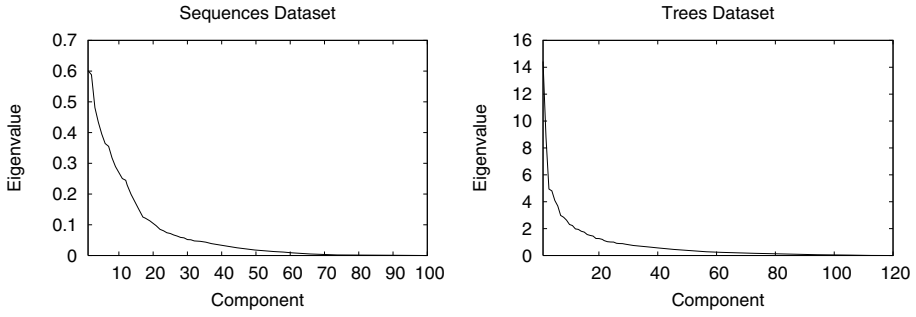

Fig. 2. Eigenvalues for sequences and trees. The most significant eigenvalue, caused by the introduction of $\mathbf{y}_{dummy}$, is not shown since it is very high (2197.62 for sequences, and 10781.63 for trees), as well as null eigenvalues beyond the shown maximum x-value (Component).

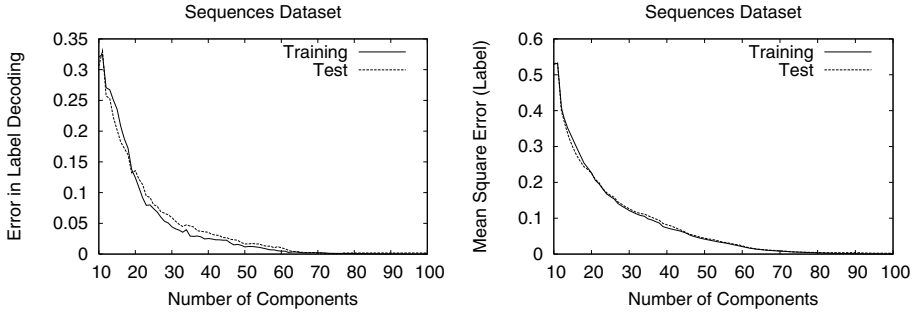

Fig. 3. Experimental results for sequences

The error in label decoding is computed as follows. Each sequence is first fed into the machine, so to get the final state for the sequence. Then the final state is decoded so to regenerate all the items (labels) of the sequence. A decoded label is considered to be correct if the position of the highest value in the decoded label matches the position of the 1 in the correct label, otherwise a loss of 1 is suffered. The final error is computed as the ratio between the total loss suffered and the total number of items (labels) in the dataset. The mean square error for labels is computed by considering the total Euclidean distance between correct and decoded labels. The final result is normalized by the number of total items. For sequences it can be seen that the machine exhibits an almost perfect generalization capability. The same result in not true for trees (see Figure 4), where in the test set there was a tree of depth 15, i.e. deeper than the deepest tree in the training set (depth 14). Thus, for this test tree the state space was not able to store all the necessary information to reconstruct it. Moreover, new paths appear in the test set which cannot as well be properly treated by the machine. Notwithstanding these difficulties, which could have been avoided by using a larger training set, the label decoding error of the machine is below 7.5% for a number of components higher than 95.

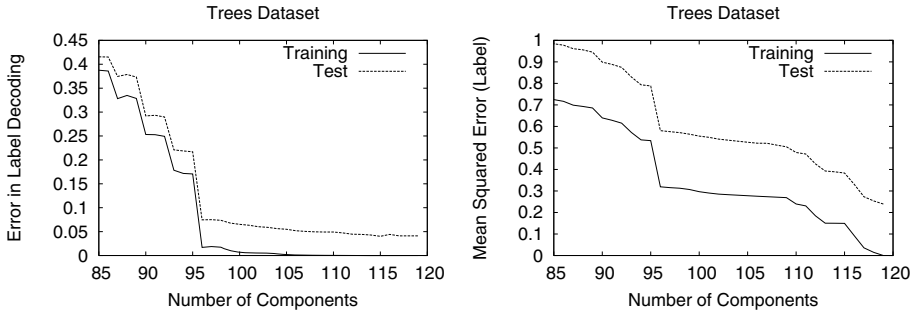

Fig. 4. Experimental results for trees

5 Conclusion

We have shown how to derive a family of exact solutions for Recursive Principal Components Analysis of sequences and trees. Basically, we have demonstrated that for these solutions there is a very close relationship with the principal components computed on explicit flat representations of the input structures, where substructures are considered as well. From a "recursive" point of view this is quite disappointing, although we have experimentally shown that in practical applications the number of parameters which a recursive solution needs is significantly lower than the number of parameters required by the (almost) equivalent flat solution.

From a mathematical point of view, the solutions are exact only if all the non-null components are used. We have still not investigated whether this property is maintained when using a subset of such components. The empirical results shown in the paper seems to indicate that using a subset of such components quite good results are obtained, even if the solution may be suboptimal. It should be pointed out that, while dealing with sequences is quite easy, the proper treatment of trees is not so trivial, due to the potential combinatorial explosion of the number of distinct paths. Thus, further study is required to devise an effective strategy for designing the explicit state space for trees.

References

1. Callan, R. E., Palmer-Brown, D.: (S)RAAM: An analytical technique for fast and reliable derivation of connectionist symbol structure representations. Connection Science, **9**(1997)139–160.
2. Pollack, J.B.: Recursive distributed representations. Artificial Intelligence **46** (1990) 77–105.
3. Sperduti, A.: Labeling RAAM. Connection Science **6** (1994) 429–459.
4. Sperduti, A., Starita, A.: Supervised neural networks for the classification of structures. IEEE Transactions on Neural Networks **8** (1997) 714–735.
5. T. Voegtlin, T., Dominey, P. F.: Linear Recursive Distributed Representations. Neural Networks **18** (2005) 878–895.
6. Voegtlin, T.: Recursive Principal Components Analysis. Neural Networks **18** (2005) 1040–50.