Hierarchical Associative Memories and Sparse Coding

Clara de Paiva
clara.paiva@tecnico.ulisboa.pt
Supervisor: Prof. Andreas Wichert

Instituto Superior Técnico, Lisboa, Portugal
June 2016

Several models for artificial neural networks are found in the literature to implement associative memories. We describe two of them (the Lernmatrix ([10]) and a Hierarchical Associative Memory model ([7]). We focus particularly on the second, which consists in a multi-layered hierarchy of matrices based on Lernmatrix, where matrices in lower positions in the hierarchy act as selective filters in the retrieval process across the hierarchy. On top of that, we propose an optimizing algorithm that reorganizes the information stored, making the retrieval more efficient (in terms of number of computations needed), while remaining as effective as the mentioned model (no added errors in the output).

Keywords: associative memories, neural networks, Lernmatrix, Learning matrix, hierarchical associative memories, ordered hierarchical associative memories, pruning, sparse code

1. Introduction

1.1. Motivation
The field of neural networks is one active field in modern biology: in the last century approximately ten percent of the Nobel Prizes for Physiology and Medicine were awarded to scientists who contributed to the understanding of the brain ([6]).

In tasks as common as memorizing and recalling information, our brains still surprise us with their ability for approximative matching or content-based retrieval, and doing this with few resources. Difficulties arise in computers to reach the same effectiveness in associative memory tasks (regarding, for instance, the capacity of storage and flexibility/capacity of generalization) while restricted by limited amounts of resources (such as processing units (neurons) and energy (sugar, electricity)).

1.2. Associative Memories
The goal of associative type of memories is to store a finite set of $P$ associations or pairs, linking a stimulus or piece of information (seen as an input $x$) to another piece of information (seen as an output $y$):

$$S \doteq (x^n \rightarrow y^p) : n = 1, \ldots, P$$

But unlike conventional memories, who are content-addressed, these associations are based on the similarity of their content or their proximity in space or time, which is an inspiration from the human's brain, about which memory Aristotle's principles ([9]) regarded as the Classical Laws of Association seem to be appropriate. Associative memories link pieces of information by pairs. Both can be served as a key to fetch one another, and both convey useful information for storage and retrieval.

2. Background

2.1. State of the art
The observation and/or guessing of how our neurons work motivated the creation of simple architectures such as the brain-inspired computational models of McCulloch and Pitts ([4]), followed by compositions of that one and others. These mathematical realizations of artificial neural networks have found widespread adoption as function prediction or general statistical tools, but since their inception they have also been used to model associative memories.

2.1.1 Lernmatrix
Lernmatrix: introduction of free parameters for learning
Karl Steinbuch’s model [10], the Lernmatrix, is a correlation matrix $W$ implementing a neural network capable of learning association patterns in a supervised fashion.

The Lernmatrix is a matrix initialized with zeros and represents, with its entries $w_{ij}$, the frequency of the correlations between components, or, in other words, the strength of the "links" between each bit $x^i$ of an input $x^n$ to each bit $y^j$ of an output $y^p$. This is biologically inspired by the coincidence learning detected by D. Hebb[2], whose so-called Hebb's Rule/Postulate refers that simultaneous activation of neurons leads to pronounced increases in synaptic strength (electric currents) between those neurons,
and this is translated as a positive update (1-valued) of the entry $w_{ij}$ of the matrix linking those observed bits of the input/output pair. This shall be done iteratively for $p$ associated pairs $(x^a, y^a)$, making possible the adaptation (learning) to those different pairs in one only matrix.

This weight update can be written as an additive learning rule:

$$w_{ij}^{new} = w_{ij}^{old} + (y_i x_j)$$

which is equivalent to a kind of boolean sum (OR), which is a clipped or normalized expression that employs also a notation usually preferred, in hardware implementations:

$$w_{ij}^{new} = w_{ij}^{old} \lor (y_i \land x_j)$$

It was this introduction of the synaptic weights as free parameters (parameters that can be adjusted to make the model fit the data) that permitted the learning capacity.

**Prediction**

One can use the Lernmatrix to check or predict the output $y$ for an $x$ given as input cue. That will appear to be the associated $y^a$ if $x$ is equal to a $x^a$ belonging to a previously learned pattern-pair; otherwise, it will be predicted or estimated component by component (bit by bit) according to a given threshold established for that column/component. The binary components of $y^a$ (the bits $y^a_i$), are expressed as threshold functions of the currents allowed, or weights:

$$y_i = \begin{cases} 
  1 & \text{if } \sum_{j=1}^{N} w_{ij} x_j \geq T_i \\
  0 & \text{otherwise}
\end{cases}$$

**Figure 1:** Output component $i$ of the neuron ($y_i$), which may be seen as a binary automaton, whose states belong to $\{1,0\}$. The neuron’s soma calculates the dendritic potential (weighted sum), and makes a decision comparing its value with a certain threshold $\theta$. Adapted from [1]

A diagram of these computations, concerning the retrieval of each component of the output, can be seen in figure 1.

**Thresholds, lossless learning, and sparse coding regime Note:** Here in this summary it will be exposed mainly the conditional and operational aspects necessary for the models that will follow. Other aspects that are relevant and characteristic from this model are only mentioned in this summary: a lossless learning, a certain degree of noise and incompleteness tolerance, and a retrieval that in non-optimal conditions only brings add-errors, for which the choice of the threshold is important; and high storage capacity or number of patterns learned $L$:

$$L \approx (\ln 2)(N^2/K^2)$$

This is valid for sets of equally sized (with dimension $N$) learned patterns, with the same number $K$ of ones each (These can be distributed over any coordinate of each vector), and with the optimal value for $K$:

$$K \approx \log_2 N^2 = \log_2 (n/4)$$

as demonstrated by many researchers ([5, 3, 8, 11]). This condition (logarithmic number of 1s relatively to the length of vector) is called a sparse coding regime. A sparse coding regime suggests a long search (multiplications, additions, and comparisons as big as the number of columns $N$).

**2.1.2 Hierarchical Associative Memory**

**Approaching through hierarchies thanks to an hierarchical structure**

Sacramento’s model proposes an extension of the previous model: instead of a single layer of neurons, we have now $R - 1$ layers ($R$ populations of neurons in total), each of which has a different matrix.

These layers share the same input and address neuron input $m$- dimensional population, but have distinct hierarchically ordered and dimensionally sized content space dimensions:

$$n_1 < n_2 < \cdots < n_{r-1} < n_r < \cdots < n_{R-1} < n_R$$

with $r = 1, \ldots, R$ as the levels of the hierarchy. The state of the network is hence represented by an ordered set of $R$ weight-matrices $W^r$:

$$W = (W^{(1)}, W^{(2)}, \ldots, W^{(r-1)}, W^{(r)}, \ldots, W^{(R)},)$$

in which the dimension of each $W^r$ is $m \times n_r$.

The hierarchical layers comprehend states for the network at different resolutions, depending on their output dimension ($n_r$):

- The upper layer, the $R$-th layer, contains the state of the original Lernmatrix ($W^{(R)} = W$, and $n_R = n$), which is materialized by the highest-resolution matrix.
- All the remaining layers below learn patterns of increasingly lower resolution (from the higher-resolution level $R - 1$, to the lower-resolution level 1).
Different resolutions

The procedure for learning in this extended structure includes Steinbuch’s learning, and an extra operation of aggregation between hierarchies:

- The $R$-th layer and its $W^{(R)} = W$ matrix proceeds to the original Lernmatrix learning based on Hebb’s rule of all the $(x^r, y^r)$ pairs of input-output patterns:

  \[ w_{ij} = \min(1, \sum_{\mu=1}^{m} x_i^\mu y_j^\mu) \]  

(8)

- Successively (from $r = R - 1$ to $r = 1$), each $r$-th layer’s matrix $W^r$ applies the same rule but based on an output $y^{(r),\mu}$ adapted to the resolution of that layer:

  \[ w_{ij}^r = \min(1, \sum_{\mu=1}^{m} x_i^\mu y_j^{(r),\mu}) \]  

(9)

This output of the $r$-th layer, $y^{(r),\mu}$, is defined as an aggregated version of the output $y^{(r+1),\mu}$ (belonging to the layer of immediately higher resolution $r + 1$), adapted to the resolution of its layer and that works as a synthesis of the output used for the learning in the upper layer:

\[ y^{r,\mu} = \zeta_r(y^{(r+1),\mu}) \]  

(10)

where the aggregation is made by the $\zeta_r$ function defined as:

\[ \zeta_r(a_r, y^{(r+1),\mu}) : y^{(r+1),\mu} \to y^{(r),\mu}, \]

with $y^{(r+1),\mu} \in \{0, 1\}^{n_{r+1}}$, $y^{(r),\mu} \in \{0, 1\}^{n_r}$

\[ \zeta_r : y^{(r),\mu} = \zeta_r(a_r, y^{(r+1),\mu}) = \bigwedge_{j=1}^{a_r} \bigvee_{\mu=1}^{\mu} \left[ y^{(r+1),\mu}\right]_{j}, \]

with $a_r \in \mathbb{N}$

(11)

The integer $a_r$ is constant for each layer, and this aggregation formula means that each component with index $j$ of the input $y^r$, $y^r_j$, is an aggregation of $a_r$ components from the input of the layer $r + 1$ above.

In other words, $a_r$ represents the size of the aggregation window used by $\zeta_r$ in layer $r$ to create $y^r$ from $y^{r+1}$ (output from the layer above, $r + 1$). $a_r$ relates every consecutive layers’ dimensions $n_r$ and $n_{r+1}$:

\[ n_r = n_{r+1}/a_r \]  

(12)

being that when $r = R$ (highest resolution), $a_r = 1$, which is equivalent to $n_r = N$ (the respective matrix $W$ can be considered to be a Lernmatrix).

One can visualize how this works with the illustrative example in figure 2.

Retrieval shortcuts

The retrieval procedure in each matrix is the same as with the Lernmatrix, but benefiting from the cut of the subsets of neurons that are necessarily unrelated to our input pattern, which shortens the search.

In our previous example (figure 2), it begins at level $r = 1$, the lowest-resolution layer. There, as for the Lernmatrix, an input cue $\hat{x}$ is presented to the matrix (W$^1$), and the dendritic potential $d_j$ is calculated for each column $j$, followed as well by the non-linear activation function for each component of the output of that layer, $y^{(1)}_j$.

At this layer 1 (as well as any following upper layer $r$), and because of the OR-based $\zeta_r$ function just presented, a non-firing component $y^{(1)}_j = 0$ (or $y^{(r)}_j = 0$) means that in the corresponding extended output-set in the layer immediately above, \( \{ja_r - (a_r - 1), \ldots, ja_r\} \) has only 0s.

\[ y^{(1)}_j = 0 \iff y^{(r+1)}_j = 0, \forall j \in \{ja_r - (a_r - 1), \ldots, ja_r\} \]  

(13)

i.e., those output-population neurons of layer $r + 1$ have no information (firing 1s) to retrieve. Hence, the search does not pursue through those columns: they are ignored in the next layers. (see figure 3)

Conversely, regarding firing components of the in-
put $y^r$:

$$\forall y_j^r = 1, \exists y_j^{r+1} = 1 \in \{ y_j^{r+1} | ja_r - (a_r - 1), \ldots, y_j^{r+1} | ja_r \}$$

which means that there is at least one possibility amongst the possible outputs to generate at layer $r+1$ some fire shot. Contrarily to the case where there were 0s in the output of the previous layers, this makes it worth to proceed (see figure 4) with the dendritic sum:

$$h_j^{r+1} = \begin{cases} \sum_{i=1}^{m} w_{ji}x_i & \text{if } y_j^r = 0 \\ 0 & \text{otherwise} \end{cases}$$

and proceed with the thresholded decision:

$$\tilde{y}_j^{r+1} = \begin{cases} \Theta[d_j^{r+1}] - \theta & \text{if } y_j^r = 0 \\ 0 & \text{otherwise} \end{cases}$$

2.2. Matter: unoptimized aggregations

In the previous model, making aggregations seems to be simply about grouping together $a_r$ columns altogether (based on the function OR). Now, this is a preformatted resume, equally repeated along the columns. In other words, obtaining performant aggregations is a matter of luck. In some cases, as in figure 5, where the information in the matrix is quite uniformly distributed, one can spot few differences between the columns of the aggregated matrix regarding the content of their lines (specially in the case of the columns 2, 3 and 4).

This means they codify similar correlation patterns, and hence, they may present similar results (fire shooting or not) regarding the threshold decision.

What could be done to attenuate this effect?

3. Proposed solution: Ordered Hierarchical Models

3.1. Motivation: criteria for filtering, while preserving the integrity of neurons

One manner to attenuate the effect of aggregating columns (in the example of the figures, this concerns layer $r = 1$) codifying similar correlations with inputs seems to be avoiding to have groups of similar columns (in this case, avoid having similar columns (of layer $r = 2$, in the images) in different aggregating groups of 4 columns. Or, in other words: grouping columns of an upper layer that are different in different groups may generate more differentiated aggregated columns in the lower layer. Or, in another word: group together similar columns.

See Figure 6 in which the columns of upper layer $r = 1$ are the same as in Figure 5 but were considered in another sequence: a sequence that tried to group, in on hand (in the left-hand side of the matrix) all the columns that have information (some 1s) and, in the other hand (in the right-hand side of the matrix), all the columns that have no information (that is, they only have 0s). After this reorganization of the columns, we proceed to the OR-based aggregation as in the Hierarchical Associative Memory model presented in the previous section.

One can see, comparing the model of the normal Hier-
The intuitive idea is that the benefits of the use of the hierarchical model seem to be, hence, potenti-ted by an organization of 1s in the matrix that allows windows to group more 0s (or more 1s) together. What if it is possible to, after the learning phase of the hierarchical model, rearrange the 1s in the matrix closer together, as well as the 0s?

Intuitively, reorganizations of the matrix might have no obvious way to work: in the retrieval of information of output patterns, the one and only information that matters to be retrieved is the presence and position of 1s in the output vector, that is: the results of the multiplication of the input vector per each column of the matrix (which represents a neuron, which is be responsible for the result of a particular component of the output). This outcomes directly from what the matrix contains in each line, or, in other words, from what each column has in each column (or the correlations with inputs learned it codifies). Isn’t reorganizing the 1s in the matrix spoiling the information codified? An uncareful reorganization would be, for the simple reason that each column is a neuron and codifies a pattern of correlations with the inputs learned (Hebbs rule’s correlations are codified thanks to the position (line, column) in which the 1s are written).

This concern could be right: How can one then reorganize a matrix without losing information? One would lose information . . . unless one keeps track of the changes made.

3.2. Idea: how to aggregate similar patterns

3.2.1 Formulation

To sum up the idea more formally: considering, in the previous sections, both the idea and the concern with the codification of the correlations, one is motivated to create a new model that:

- reorganizes the matrix grouping similar columns together
- with the condition, or restriction, of preserving the integrity of the columns
- managing to obtain the expected output

In other words, a solution seems to be reordering the columns of the matrix (the correlations, codified as the entries of each line, are preserved), while being
able to remember their original order to be able to establish the correct output again, later. How to respect this last concern? Keeping note of the original sequence of the numbers of the columns, and keeping that saved until the end of the algorithm, when it will be useful. One could then reconstruct, in the end and in an equivalent way, the correct sequence of 1s and 0s.

3.2.2 One possible method for aggregating

Let's go through the matrix line per line, and sort its entries in order to form a cluster of 1s on one side, and a cluster of 0s on the other side. The entries are labelled with the number of the respective columns.

In each iteration, that is, in the effort of clustering of a particular line, we should not lose the effort of clustering done in the previous iteration. As such, the clustering will be done inside each group of grouped columns.

In the next section we proceed to a more precise description of the algorithm.

3.3. Learning phase

Learning of patterns

The procedure repeated for learning every pair of patterns is exactly the same as for the Lernmatrix (which applies Hebb learning rule).

After this step, we obtain a matrix for level \( r = 3 \). There is a polishing step that accomplishes the idea (before the filling of the lower matrices by aggregation of values) and consists in the reorganization of the columns of the matrix of the top-layer in order to cluster 1s and 0s as close to their peers as possible, while respecting, in the same time, the integrity of the columns.

Reordering the matrix

The proposed algorithm for this reordering would be, in simplified pseudocode:

```plaintext
for each Line \( L \) of the Lernmatrix:
    for each Sub-Sequence \( SS \)
        in AllSequenceList(iteration \( L \))
            if (Ones&Zeros unclustered?(\( SS \)))
                SplitAndOrder(\( SS \));
```

Before explaining the algorithm, one has to mention the methods used to keep track of the sequence of columns and the related existence of auxiliary structures, as well as initialization details:

Essential auxiliary structures:
- All the columns are identified or labelled with the numbers of their position (1, \( \ldots \), \( N \) - with \( N \) as the number of columns in the original matrix (Lernmatrix)).
- These identifiers are kept ordered in arrays (corresponding to nodes of a double linked list), which that are sometimes called "Sub-Sequence" for simplification, as they contain a sub-sequence of the indexes of the columns.
- The doubly linked list is unique and is called allSequenceList. It forms a sequence of nodes (that is, a sequence of sequences).

Essential functions:
- In OnesandZerosunclustered?(Sub – Sequence), it is checked if there are different values (1s and 0s) within a node, mixed altogether. If so, in SplitAndOrder(Sub – Sequence), the node should be split in 2 nodes that will take its place in allSequenceList. The indexes (numbers of the columns) migrate (in sequential order) to the left node if their content is '1', and to the right node if there are 0s.

Notes:
- The maximum (worst-case) number of partitions in a step/iteration/line \( l \) (\( l = 1, \ldots M \)) is \( 2^l \) (reached if there is the case where all the partitions have 1s and 0s mixed, that is, they are required to be split).
- We use stable sorting: In each iteration \( l \) of the algorithm, we do not reorder or destroy the reordering of the previous iteration, \( l – 1 \): we do the split over indexes inside each Sub-Interval, keeping indexes from different Sub-Sequences apart from the ones of other Sub-Sequences. Stable sorting allows our algorithm to be conservative (not destroy the ordering work of the previous iteration) and to be deterministic: if one runs it through the same matrices, one can reproduce the same ordering of the partitions and columns.
- Variations of the algorithm can be made in the order upon which we choose the lines we iterate over; the natural (i.e sequential) order is called the "naive" variant of the algorithm.

Illustrative example:

In each step or iteration of the algorithm (for each line), we are going to present the state of the matrix representing the auxiliary structures (allSequenceList and its subcomponents(nodes))

Matrix (Lernmatrix) after traditional learning (see figure 7):

**Figure 7:** State of the matrix of layer \( r = 3 \) after learning

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Initialization:

Iteration 1: line 1 Let's isolate the first line of the matrix (figure 8). By this time,
Figure 8: First line of the matrix (in the original, unordered, matrix)

\[
\begin{matrix}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\end{matrix}
\]

AllSequenceList’s unique node (fig. 9) contains the sequence of indexes as it is (ordered increasingly, from 1 to 12).

Figure 9: Initially, AllSequenceList is a single node. This contains the sequence of all indexes, in their initial (ascending) order.

Let’s reorder it in order to have all the 1s in a new node on the left and the 0s in a new node on the right (see fig. 10).

Figure 10: First line of the matrix, after the respective iteration over it for reordering. As we detect two kinds of values ('1' (in indexes 1, 4, 9, 10 and 11) and '0' (in indexes 2, 3, 5, 6, 7, 8 and 12) as components of the sequence of this node, it will be partitioned into two sequences. Each of these sequences maintains the order that there was previously of each index relatively to the others.

Remarks:
- It will be essential (for the retrieval phase) that we keep the information of the indexes (the one in shaded lines) associated to each column.
- Stable sorting is employed. Amongst the columns of each Sub-Sequence: the relative order of the columns is maintained. (for example and in this case: In the left Sub-Sequence (the one containing the 1s), we have: 1 < 4 < 9 < 10 < 11.)

So, in our representation, AllSequenceList becomes (see fig. 11):

Figure 11: AllSequenceList after the first iteration of the ordering algorithm, with two nodes.

\[
\begin{matrix}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\end{matrix}
\]

Iteration 2: line 2 In the context of that con-
tent of line 2 (isolated in fig. 12), and in the context of the previous partition (obtained after iteration 1, and represented in fig. 11), the content of this line considering AllSequenceList is like it is illustrated in figure 13.

Figure 12: Content of the second line of the (original, unordered) matrix.

\[
\begin{matrix}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\end{matrix}
\]

In both cases (in the case of the left partition, as well as in the case of the right partition), we observe that there are 1s and 0s in its content. So, both Sub-Sequences will be split: the columns with 1s will be placed on the left partition, and the columns with 0s will be placed on the right.

After reordering and splitting, our representation of AllSequenceList becomes as in fig. 14:

Figure 13: Content of the second line of the (original, unordered) matrix, in the context of the auxiliary structures of our algorithm (and before the ordering of the iteration proceeding over that line).

\[
\begin{matrix}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\end{matrix}
\]

The iterations over lines 3, 4 and 5 are analogous.

In the end of learning, the sequence of indexes of columns is then: 4, 9, 10, 1, 5, 3, 2, 7, 8, 12, 6. Let’s consider the new Lernmatrix, with the columns reordered, and propagate that reordered information by OR-aggregation to the lower layers 1. Then, we can compare this with what would be obtained previously (based on the ”normal”, not reordered, Lernmatrix): see figure 15.

3.4. Retrieval phase

We proceed exactly as in the (non-ordered) Hierarchical model to retrieve an output through the layers of matrices. The only difference is that in the end, one has to construct the final output putting the values in the original position of the indexes.

For example: Let’s consider the final order of the indexes after reordering of the matrix, of the previous example: indexesSequence = 4, 9, 10, 1, 5, 3, 2, 7, 8, 12, 6.

Let’s suppose we use the Ordered Hierarchical Model hierarchy of matrices and obtain \([1,0,0,0,1,0,0,0,0,0,0,0]\) as a (temporary) output. There, we have \(y_i = 1\) for \(i = 1\) and \(i = 5\). The corresponding original indexes \(j\) are

---

1 One can proceed to the learning of many (all) of the pair-patterns in the Lernmatrix (matrix of the upper level) and “propagate” these correlations all at once for the layer below \((r – 1)\) by simple aggregation of the entries of each window of \(a_{r-1}\) entries of the matrix, for each line. These two methods (that is: On one hand, learning pattern by pattern, and using a different output (aggregation of the one of the upper layer) for each layer), and on the other hand, learn all the patterns at once in the upper layer and then fill the lower layers by aggregation of the columns, without using intermediate output vectors) result the same.
Indicators

Along the execution of all the models, some metrics are collected, stored and displayed, as indicators for the state of the matrices, quality, and performance of the execution of the associative memory task:

- Performance: Number of different types of computational steps for each retrieval
- Quality: Hamming distance, Add Errors, Miss Errors between the retrieved output and the output associated to the learned input that is the most resembling to the input cue introduced.
- State of the matrices: evolution of the memory load after each pattern is learned, memory distribution (display of information bits (1s) in matrices, number of 1s per column, accumulation of 1s) The observations made in the results were based on some of these indicators or combinations of them (one in function of the other) programmed in Julia.

Phases and parameters (structure of the network and data) Every test comprehends the following phases, for every model:

**Setups**

- Independently of the models, a data base of patterns is set only once, and according to:
  - Parameters pre-established:
    - *a specified number of patterns* (number of pairs of inputs and outputs)
    - *K*: number of 1s per vector (the number of 1s used is the same for inputs and for outputs, in all patterns)
    - *N*: number of components of the inputs and outputs (which are equally sized. *N* is also called *number of neurons, number of columns* or *width*, or, as the vectors have the same length and as such the matrices are square, *height* or *number of lines*)
    - The distribution chosen for the 1s in the components of the vectors. (It may be an uniform distribution or other)
  - This data set is the one used by all the models and all the tests. The experiences are as conservative from the point of view of the data used as possible: When *p* (the number of patterns learned) varies along the experience, that is, when two tests (let’s say *test1* and *test2*) use a different *p* (*p1 < p2*, for instance), the set of the *p1* patterns patterns is included in the one of the *p2* patterns: there are just some extra *p2 − p1* patterns learned (the ones of indexes *p1* . . . *p2*, in the database) in *test2*.
  - The setup of the structure of the network (matrices and their initialization of all entries to 0) is made for each test (overriding the previous test) and according to:
    - Parameters pre-established:
      - *N* (already referred. As the inputs and outputs have the same length, the matrices are square and their size if $\text{N} \times \text{N}$)
      - Number of layers *R* The Lernmatrix is the only single-layered model, with *r* = 1. All the others use arbitrary *R*. The layer of level *r* = *R* is the biggest (size $\text{N} \times \text{N}$) and corresponds to the original Lernma-
trix. All the remaining $R - 1$ layers are smaller matrices that work as filters for the retrieval in upper layers.

* Aggregation factors $a_r$. An array of integers defines the relation between the widths of the matrices of the different layers of the hierarchy. In general, the number of columns is set as:

$$N_r = \text{roundup}(\text{div}(N_r + 1, a_r))$$

where $\text{div}$ is the truncated division, and $\text{roundup}$ does nothing when $N_r + 1$ is a perfect multiple of $a_r$, and increases the value by 1 otherwise.

**Learning phase** Before the execution of each model the matrices are initialized again. The patterns to learn are reused as much as possible.

After the learning phase we generate files for inspection of the memory load and information distribution (1s) in the matrices. (Some of them will be used in the results section)

**Retrieval phase** In order to be able to establish accurate comparisons more easily between tests, the set of patterns we tried to retrieve in each experience has an arbitrarily fixed size, and includes the same patterns.

Along the retrieval procedures we collect many performance and quality factors that we keep in memory and present in files generated in the end. (Some of them will be shown in the results section)

### 4.2. Steps (Performance) and addErrors (quality)

**Method** As mentioned previously, most of the cost of the operation of the network (in biological systems, mostly from the chemical reactions in and between neurons and, in artificial systems, it is at least related to the number of the computations made) is associated to the following kinds of operations, or calculations, that were all counted:

- Multiplications
- Additions (both intervening in the dendritic sum in the soma body of the neuron)
- Comparisons (corresponding to the thresholded decision)
- Fire shots (activation of one component of the neuron, or of one position of the output)

**Total number of steps in layer $r = 3$**

Because we guessed that the total number of steps for the retrievals could depend on how much loaded the memories were, we measured it for tests classifiable by memory load.

**Experience default** Number of total steps in function of memory load for all models:

- For the retrieval of 20 patterns
- 120 tests
- Constant size of the network ($N$ constant and set to 512)

- Constant number of 1s per input and per output ($K = 1$), uniformly distributed (Normal, or Gauss, distribution) over the length of the vectors
- Variable number of patterns learned ($P$)
- Constant ticks of memory load (of 0.0005)

In the plot:

**Figure 16**: Number of total steps in function of the memory load of matrix of layer $r = 3$, for all models, and for 120 tests.

- Each point belongs to a colored curve representing the behavior of some model.
- Each point corresponds to a test. That is: to the retrieval of an arbitrarily set number of outputs (20, in this experience) based on (the same, and 20) input cues selected from the battery of inputs patterns.
- These tests (points) are numbered, for identification. Each identifier matches the test with the parameters used in that test.
- In the y-axis is represented the total number of steps, which is the sum of the total number of steps necessary for retrieving all the set of input patterns.
- The memory load, in the x-axis, refers to the ratio between the number of 1s in the matrix (as a numerator) and the total number of entries (in the denominator) in the last ($r = 3$) layer of the hierarchy. The setting of the memory load depends indirectly on the parameters chosen for the network ($N$) and patterns ($K$, $P$). In the case of this experiment: The number of 1s per vector ($K$) is arbitrarily set to 1, as well as the number of neurons ($N$) is set to 512. ($Number\ of\ columns = width\ of\ matrix\ of\ layer\ r = 3, = length\ of\ input\ and\ output\ vectors$). Hence, the tests were parameterized by $p$, and in order each test has a memory load increment of approximately 0.00005, compared with the memory load of the previous test. The constancy of the step was made for practical reasons.

The combinations of parameters used for the tests identified by Test ID are presented in the Table 1, as well as the memory load.
4.3. Settings

The results for all the models for the total number of steps are shown in the plot of the Figure 19.

**Observations** The most striking observations about this experiment (and its variants) are:

1. The (blue) curve of the Lernmatrix stands always much higher than all the remaining curves: The total number of steps of the Lernmatrix ($\approx 10,5 \times 10^6$) is much bigger than the others ($< 2,0 \times 10^6$)
2. The curves of the hierarchical models without the discard of the null-columns in the right of the matrix (that is: curves red, green, black) seem to stand all close to each other. A closer look allow us to see the number of steps: Hierarchical model > Hierarchical with ordered columns > Hierarchical with ordered columns choosing the ones with more 0s first > the same as the previously but discarding null columns, in the right of the matrix.
   It seems that just by reordering columns, it is possible to obtain advantage in performance.
3. The model with ordering and discard of null columns has by far the best performance in lower memory loads.
   However, as the memory load increases, it seems that its advantage disappears, as its performance gets closer to the one of the other models hierarchical models.

**Discussion** Here follow interpretations corresponding to the observations made in the previous point:

1. **Benefits of the hierarchie**
   Applying hierarchies as filters brings a considerable performance advantage: The filtering associated with the hierarchies brought by hierarchical models allows to prune a considerable number of branches in the retrieval ($\approx 80\%$, since $2/10,5 \approx 1/5 = 20\%$)
   The pruning brought by hierarchies was already referred in [7]
2. **Performance of ordered models**
   Another factor allowing more pruning seems to be the release of memory: one interesting point is that ordering the columns frees memory, or, in other words, makes the matrices have more 0s while they still obtain the same outputs.

We did not figure this out clearly at first. To observe and try to illustrate what the ordering does that brings benefits, we first plotted the state of the memory occupation for the different models for instance for testId = 10 ($p = 144N = 64k = 1$).
But this releasing of memory was better observed for instance with the plots of the figures 19, 18 and 17, where the same values (total number of computational steps) are depicted, but in function of the memory loads of the different layers. If we consider the markers of the points, which are the Ids for the tests, we remark a shift towards the left (in the ordered models, relatively to the un-ordered) in the
aggregated levels ($r = 2$ or $3$). This means that: for the same test, we present, in the aggregations resulting from the reordered matrices, matrices with lower memory loads, relatively to the aggregations resulting from un-ordered matrices. As long as we do have empty space in the matrices (low-density matrices), we can benefit from reorganizing it - we have operating space.

3. **Accumulation of null-columns on the right** As a result of the ordering, we observed in the end that most of the 0s were not only together, but also much more in the right of the matrix than in the left. This is natural, as our reorganization of the columns is based on partitioning each line of the matrix (1s to the left, 0s to the right). Therefore, probably because of this organization, some null columns happen to be placed in the right columns of the matrix. And as we use sparse coding it happens that there is a certain number of null columns. **Note:** The lower the memory load is, more null columns it tends to exist in the right of the matrix. We understand then that the emptier the memory load is, more columns are discarded which allows the model to have a better performance. The inverse seems to be also right: the more loaded the memories are, the less benefits there can be in this model, and as such, its performance converges to the ones of the others. Still, there is not much interest in using higher loaded matrices, as the number of Add Errors increases with memory load.

4.4. **Different kinds of steps**

The number for each of the different kinds of computations is not always equal for all models. For this group of algorithms, the number of additions and multiplications are exactly the same (as these operations are performed altogether in these modelisations of the dendritic sum): In the models of this chapter, this is always modeled as the multiplication of one line-vector (input) by a column-vector (neuron). Whenever we do that operation, there are $N$ multiplications (length of the input, as well as length of the column) relating each component (index) of the input with each component of the output, and $N$ additions, between the $N$ multiplications already made. The evolution of these kinds of steps is proportional to the one showed, of the total number of steps. Although the number of comparisons is different (much lower) than those of additions or multiplications, the evolution is the same (we observe easily that the curves have exactly the same small peaks or fluctuations for instance).

This is also normal, as we expect these kinds of steps to be proportional: for each dendritic sum considered, we do a comparison (with a threshold). Hence, and as each dendritic sum evolves $N$ multiplications and $N$ additions, we should expect the plot of the number of comparison steps to be analogue to the previous one by a proportionality factor of $2N$.

On the other hand, the evolution of the number of fireshots (see Figure 20) is the most contrasting one, as it doesn’t present the same evolution of the other operations.

**Figure 19:** Number of total steps in function of the memory load of matrix of layer $r = 3$, for all models, and for 60 tests.

**Figure 20:** Number of fire shot steps per memory load test, in the same experiment (65 tests. $N=32$. Growing $p$, function of memory load)
retrieve 20 patterns, and we have \( K = 1 \) patterns, one should have 20 fire shots in the case of the Lernmatrix (corresponding to the firing of the 1-components of the output). This is explained by the apparition of add-errors: taking for example a value easy to read: in test no. 38, we have 40 fireshots (instead of 20). If we refer to the previous graph, we can check that for test no. 38, we have 20 add-errors, and if we sum those to the 20 1-components, we have the 40 fireshots we see in this plot.

2. The hierarchical models present a bigger number of fire shots. This is also understandable: as there are extra layers (relatively to Lernmatrix), there are intermediate fire shots in the temporary outputs that serve as interfaces between the different matrices of the layers, in the retrieval.

3. The ordered columns' models present less fire shots (than the unordered hierarchical model) because they managed to prune some extra columns in the retrieval search (This diminution of the number of fire shots concerns, of course, only intermediate fire shots. The fire shots at the exit of the last layer are always the same number (in all models), as the outputs obtained should be, and are, the same).

4. discarded columns' model has the same number of fire shots as the other because in every case, null columns will not bring extra fire shots, as they will never fire.

**Note about applications:** The counting of the different kinds of steps associated to the energy cost of these operations in biology or other may be useful to apply in energy budgets. In fact, these different kinds of computations require different energy costs in biology and in hardware. Considering this can be useful in all deliberations for energy budgets.

5. **Conclusion**

5.1. **Achievements**

5.1.1 **Considerable savings**

The number of computational steps saved is considerable: considering for example the total number of steps (steps of all kinds), we may have:

- Relatively to the Lernmatrix: savings from 98\% (in the sparsest regime tested, in test 1 (memory load of 9.92\(e\)-5)) to 81\% (in the test 60 (memory load of 3.05\(e\)-3)) (see Table 2)
- Relatively to the Hierarchical model: savings from 88\% (in test 1) to 20\% (in test 60) (see Table 3)

And this is the main measure of success. This can be used in associative memories 'applications, perhaps even biological, thanks to the energy savings that we achieve by not doing unnecessary operations (chemical reactions in the case of cells. More precise energy budgets can be made if the different kinds of steps are considered, and confronted with the updated values for the different kinds of cells found in the Biology literature).

5.1.2 **Compatibility with sparse coding regimes**

This can be considered as a successful implementation, since the regimes in which associative memories operate - sparse coding regimes - are prone to the the reorganization of the useful information (1s).

This placement of the 1s in a reorganized way allowed the retrieval to be more result-oriented than in the previous models, rather than extensive scrolling throughout the network at every retrieval.

**Note:** This does not preclude the use of the same strategies that would already be used in previous models (Lernmatrix, Hierarchical model) - such as the parallelization of calculus operations and comparisons of each column (neuron) of the matrix. This only restricts the universe in which we need to operate, which always is an advantage.

5.1.3 **Worthy trade-offs**

The ordered associative memory models definitely introduce a greater effort in the phases of setup and learning. This is evident at the infrastructure level (hierarchies bring smaller, but extra, matrices; and ordered models use an extra list (\textit{AllSequenceList})) and at the computational level (aggregation for filling of the matrices of the lower levels, in the case of the hierarchical matrices that serve as filters; and reordering, in the case of the ordered models) which undeniably spends more resources (both memory and processing).

However, it is worth spending resources (memory and processing) in a scenario where the neural networks are used more for retrieving than for set-ups and learning. This is a mostly common scenario for neural networks 'application; and in those cases, it is worth to consider that there will be no additional costs beyond these at the beginning. And the more retrievals are requested, the more steps we will be saving and the more the initial investment will pay off.

5.2. **Future work**

5.2.1 **Adapt aggregation factor to memory distribution**

As seen previously, our Ordered Hierarchical Models reorganize the matrices, and pretend or at least try to form clusters of 1s and 0s. This changes the density of 1s in different zones of the matrices. As such, one could think of using variable aggregation factors: maybe narrower windows should be used on the left-side of the matrix, but then, with the "dilution" of 1s in the right-side of the matrix, larger ones could be used, allowing, maybe, a lower number of threshold decisions to be made.
5.2.2 Data with correlations

In this hierarchical memories model (as well as in most of the neural networks models), the patterns are considered to have an average activity level $K$, and to be uncorrelated.

However, the latter might seem biologically unplausible, as most data comes along with other data close to it in time and/or space.

Hence, it could be interesting to see how naturally correlated data affects performance capacities, as well as what could be updated to adapt the model to those possibly adverse effects.

Also, with correlated data probably some zones of the matrix will be "overpopulated" while others can be quite empty. In that case, would the use of different aggregation factors depending in the zones/columns of the matrix be useful?

5.2.3 High-fidelity sparsification

Another open question concerns how to transform natural and correlated patterns in sparse data, conserving the similarity with the original data.

References


Table 1: Combinations of parameters used for the tests of our experience-default. Each test is identified by testID and the characteristics in bold. $N$ and $K$ are kept constant, and $p$ varies, being chosen in order to originate tests with a constant pace in the difference of the memory loads relatively to the other tests.
Table 2: Total number of computing steps: Comparison (in %) of models' performance relatively to the Lernmatrix in the tests identified by ID

<table>
<thead>
<tr>
<th>Test ID</th>
<th>Hierarch., Columns reordered</th>
<th>Hierarch., Columns reordered, Select More 0s</th>
<th>Hierarch., Columns reordered, Select More 0s, disc. null</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17.774%</td>
<td>17.774%</td>
<td>3.516%</td>
</tr>
<tr>
<td>5</td>
<td>17.774%</td>
<td>17.774%</td>
<td>3.516%</td>
</tr>
<tr>
<td>10</td>
<td>17.872%</td>
<td>17.872%</td>
<td>3.516%</td>
</tr>
<tr>
<td>15</td>
<td>17.969%</td>
<td>17.863%</td>
<td>3.516%</td>
</tr>
<tr>
<td>20</td>
<td>18.116%</td>
<td>17.801%</td>
<td>3.516%</td>
</tr>
<tr>
<td>25</td>
<td>18.438%</td>
<td>17.936%</td>
<td>3.516%</td>
</tr>
<tr>
<td>30</td>
<td>18.536%</td>
<td>18.204%</td>
<td>10.782%</td>
</tr>
<tr>
<td>35</td>
<td>18.633%</td>
<td>18.077%</td>
<td>11.046%</td>
</tr>
<tr>
<td>40</td>
<td>18.829%</td>
<td>18.175%</td>
<td>11.925%</td>
</tr>
<tr>
<td>45</td>
<td>19.027%</td>
<td>18.341%</td>
<td>12.872%</td>
</tr>
<tr>
<td>50</td>
<td>19.376%</td>
<td>18.380%</td>
<td>13.692%</td>
</tr>
<tr>
<td>55</td>
<td>19.625%</td>
<td>18.409%</td>
<td>14.398%</td>
</tr>
<tr>
<td>60</td>
<td>19.925%</td>
<td>18.565%</td>
<td>15.245%</td>
</tr>
<tr>
<td>65</td>
<td>19.288%</td>
<td>18.409%</td>
<td>15.245%</td>
</tr>
<tr>
<td>70</td>
<td>19.438%</td>
<td>18.653%</td>
<td>15.245%</td>
</tr>
<tr>
<td>75</td>
<td>19.581%</td>
<td>18.770%</td>
<td>16.231%</td>
</tr>
<tr>
<td>80</td>
<td>19.728%</td>
<td>19.083%</td>
<td>16.944%</td>
</tr>
<tr>
<td>85</td>
<td>19.933%</td>
<td>19.603%</td>
<td>17.149%</td>
</tr>
<tr>
<td>90</td>
<td>20.216%</td>
<td>19.142%</td>
<td>17.579%</td>
</tr>
<tr>
<td>95</td>
<td>20.412%</td>
<td>19.327%</td>
<td>17.705%</td>
</tr>
<tr>
<td>100</td>
<td>20.599%</td>
<td>19.601%</td>
<td>18.129%</td>
</tr>
<tr>
<td>105</td>
<td>20.599%</td>
<td>19.728%</td>
<td>18.751%</td>
</tr>
<tr>
<td>110</td>
<td>20.599%</td>
<td>19.777%</td>
<td>18.993%</td>
</tr>
<tr>
<td>115</td>
<td>20.607%</td>
<td>19.849%</td>
<td>19.093%</td>
</tr>
<tr>
<td>120</td>
<td>20.841%</td>
<td>20.059%</td>
<td>19.357%</td>
</tr>
</tbody>
</table>

Table 3: Total number of computing steps: Comparison (in %) of the Ordered and null-columns discarded model performance relatively to the Hierarchical associative memory model.

<table>
<thead>
<tr>
<th>Test ID</th>
<th>Hierarchical, Columns reordered, Selected line More 0s, zero-columns discarded</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12.091%</td>
</tr>
<tr>
<td>5</td>
<td>19.783%</td>
</tr>
<tr>
<td>10</td>
<td>29.510%</td>
</tr>
<tr>
<td>15</td>
<td>38.209%</td>
</tr>
<tr>
<td>20</td>
<td>45.446%</td>
</tr>
<tr>
<td>25</td>
<td>50.637%</td>
</tr>
<tr>
<td>30</td>
<td>58.185%</td>
</tr>
<tr>
<td>35</td>
<td>59.123%</td>
</tr>
<tr>
<td>40</td>
<td>63.331%</td>
</tr>
<tr>
<td>45</td>
<td>68.099%</td>
</tr>
<tr>
<td>50</td>
<td>72.157%</td>
</tr>
<tr>
<td>55</td>
<td>75.206%</td>
</tr>
<tr>
<td>60</td>
<td>80.134%</td>
</tr>
<tr>
<td>65</td>
<td>79.240%</td>
</tr>
<tr>
<td>70</td>
<td>81.909%</td>
</tr>
<tr>
<td>75</td>
<td>82.892%</td>
</tr>
<tr>
<td>80</td>
<td>83.841%</td>
</tr>
<tr>
<td>85</td>
<td>85.881%</td>
</tr>
<tr>
<td>90</td>
<td>86.955%</td>
</tr>
<tr>
<td>95</td>
<td>87.032%</td>
</tr>
<tr>
<td>100</td>
<td>89.856%</td>
</tr>
<tr>
<td>105</td>
<td>91.427%</td>
</tr>
<tr>
<td>110</td>
<td>92.618%</td>
</tr>
<tr>
<td>115</td>
<td>92.663%</td>
</tr>
<tr>
<td>120</td>
<td>92.842%</td>
</tr>
</tbody>
</table>