ELSEVIER

Neurocomputing 14 (1997) 15–39

NEUROCOMPUTING

Assigning meaning to data: Using sparse distributed memory for multilevel cognitive tasks

Larry M. Manevitz [a,*], Yigal Zemach [b]

[a] *Department of Mathematics and Computer Science, University of Haifa, Haifa, Israel*
and
Polytechnic University, New York, NY, USA
[b] *Department of Mathematics and Computer Science, University of Haifa, Haifa, Israel*
and
Intel Corporation, Haifa, Israel

Received 23 September 1994; accepted 3 October 1995

Abstract

It is shown how a single homogeneous SDM memory can be organized to link between low level information and high level correlations. To illustrate this, we report on experiments run in a *unified* memory retrieval system, that combined pattern recognition of individual English characters followed by the assignment of 'meaning' to a string by giving it a Hebrew translation. Symmetry allows the reverse action on the same memory (i.e. Hebrew character identification followed by translation of a string to English).

Keywords: Sparse distributed memory; High-order-correlation; Associative memory

0. Introduction

Many cognitive tasks require multilevel organization. Consider, for example, a musician who can identify a piece of music from its score: first he has to identify the individual notes from his visual input, then identify the music from the sequence of notes.

To artificially accomplish a task of this sort, an associative memory would seem appropriate, of which several models [6,8,13,5,9] have been studied in recent years.

* Corresponding author. Email: manevitz@mathcs2.haifa.ac.il

SSDI 0925-2312(95)00130-1

However, to perform such a task, not only must the memory work with information on different levels, performing what seems to be different sorts of tasks on the different levels, it must also deal with pattern recognition of sequences which require multi-order correlations.

Most tasks which have been performed on artificial associative memories have limited themselves to single-level cognitive tasks and first-order correlation affects. [1] Some unspecified sort of linkage of the different mechanisms into a structured or hierarchical architecture [2] is supposed to account for multilevel effects.

It is not clear how this linkage is to be accomplished and in any case is unsatisfactory on several accounts:

(1) Any such memory processing system will seem to be completely ad hoc; each different task might require a separate retrieval system.

(2) Many memory systems emphasize their 'naturality' [8,9,10,2], i.e. in some sense they are supposed to be *explanatory* for a natural neural mechanism as well as functional. Without a unified system this explanatory ability is lost.

Kanerva's model of Sparse Distributed Memory (SDM), in particular, emphasizes

(i) the naturality of the model, and

(ii) computation as an outcome of the organization of the memory.

(This model uses associative and probabilistic memory access.)

Most experiments with the model, however, have worked on single correlation experiments, i.e. simple pattern recognition problems. Kanerva himself comments ([8], Ch. 8) that first-order predictions (single address-data relations) have a very limited value in real life situations. Yet his proposed solution (a multi-stage design) besides being somewhat complicated (requiring delay systems), still relies on combinations of first order correlations alone and therefore is stochastically insufficient. [3] (However, Kanerva has pointed out ([8], p. 89) that his design does allow the possibility of a 'more general memory' that could react when a specific 'sequence of events has just occurred'. This work can be seen as a specific realization of this suggestion.)

The goal of this work is to show how a single associative memory retrieval system can accomplish multilevel cognitive tasks. We do this in the context of Sparse Distributed Memory [8]; however the ideas are essentially adaptable to any associative memory that provides a 'best match' capability.

---

[1] Consider, for example, NETtalk [12] which uses a neural network to translate written text to phonemes. While the translation is context dependent, it manages the contextual problem by relating each 7-letters-string to a phoneme. Since there is no constraint between adjacent phonemes this is a first order correlation. See also [7] where a similar task is performed using the SDM model.

[2] For example, [12] notes that 'NETtalk is clearly limited in its ability to handle ambiguities that require syntactic and semantic levels of analysis'. They suggest the possibility of using some 'structured network' to combine information from larger parts of sentences.

[3] e.g., a memory which learned the sequences: FAT, FEW, GET, SET, FIT, RAW, PAW, NOW; given FE, and relying on 1st-order corelations alone, will predict T as the next item (since $Pr(T \mid F*) = 2/3$ and $Pr(T \mid *E) = 2/3$), albeit it is obvious that W can be deduced with certainty (given by the 2nd-order correlation $Pr(W \mid FE) = 1$). Notice that Kanerva's j-step transition is a pair, hence it can not store a multi-order correlation.

The underlying methodology used, is to assume that each item in memory is associated with a short code (e.g. 256 bit length information with 32 bit length code; in the application in this paper 64 bit information with 8 bit code was used). Then the system can first identify the code and then use code combinations to represent addresses for the higher level information. Note that there is no distinction between 'levels' – *the memory structure is entirely uniform.*

Our emphasis in this paper is not on coding per-se, but rather on the uniform storage and retrieval of different levels; accordingly, in this work we hand-chose the codes (other authors (see [4]) have pointed out that use of codes would be necessary; [1] have given one example of how a neural network could naturally assign codes).

To illustrate and test the ideas of linking low level information with high level correlations, we ran experiments in a single unified memory retrieval system that combined visual (pixel) pattern recognition of English letters followed by the assignment of 'meaning' to a string by giving it a Hebrew translation. This task was chosen because there should be little correlation between the Hebrew translation and the English pattern. That is, it is a true high level correlation problem which is solved here by multi-level processing on a uniform memory. (Of course, we are not claiming that this explains how people translate.) The structure of SDM is such that symmetry allows the reverse action on the same memory (i.e. Hebrew character identification followed by translation of a string to English).

The paper is organized as follows: Section 1 describes the organization of the memory model for use with multilevel processing; Section 2 describes how this organization is specified for use in the example of two-way translation and character identification; Section 3 gives the detailed description of the experiments; Section 4 has the tables of results; finally Section 5 has the discussion of the results and a summary. (Appendix A has the list of translation words stored in memory; appendix B has visual examples of translation outputs.)

1. Memory organization

We work within the context of the SDM model (for details see [8]). Briefly, SDM is designed to be a generalized random access memory, that provides the 'best match' for an arbitrary address. The association with the address occurs by averaging the contents of all addresses ('hard locations' in terminology of [8]) sufficiently (Hamming) close to the input address. This is implemented by a three level feed-forward neural network, with hard thresholds at levels two and three. The neurons of the first level define the input pattern, the neurons of the hidden level correspond to the 'hard' locations, the neurons in level three define the output pattern. The weights between level one and two are fixed (usually chosen randomly) for the model, establishing the addresses for the hard locations, the threshold for all level two neurons correspond to the fixed Hamming distance; the threshold in level three is set to zero, and the weights between level two and three are set by a form of Hebbian learning where a '0' bit decreases the appropriate weight by 1 while a '1' increases it by 1. (In other words, if one is storing a vector Φ of bits at an address, for each hidden neuron i that responded to the address, one modifies the weights between the hidden and output level, by $W_{i,j}^{\text{new}} = W_{i,j}^{\text{old}} + \Phi_j - \Phi_j^{\text{complement}}$.)

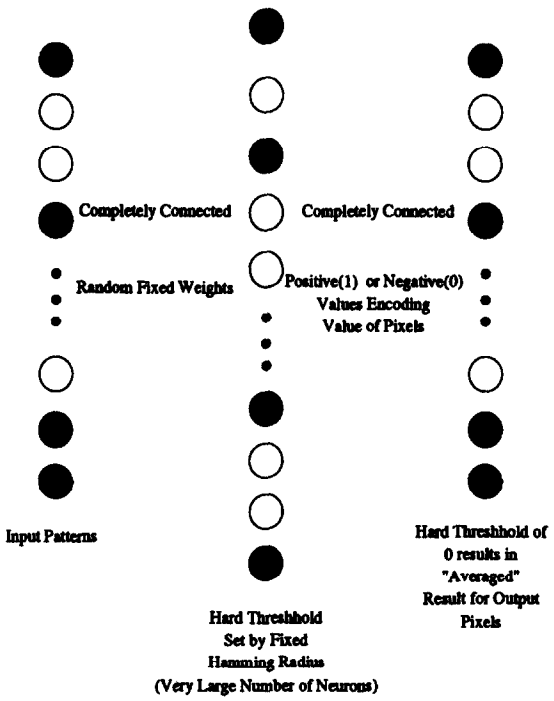


Fig. 1. Sparse distributed memory as a 3 level neural network. Note the huge number of hidden level neurons and the fixed weights between levels 1 and 2.

The result of this arrangement is that for a given input, only the set of neurons in the second level which are in the given Hamming radius of the input fire; the 0 threshold in the output level implies that on retrieval the output is a form of averaging of the storage at all the hard locations in the Hamming radius of the input. One can think of this as the network reacting as if it could respond to an arbitrary address from the 2^m possibilities, despite having only a relatively small number of 'hard' locations corresponding to the hidden level neurons. This occurs because a *set* of neurons in the hamming radius responds to each input. Kanerva's analysis [8] shows that one can choose the hamming radius appropriately so that this model works reliably. The size of this network is usually quite large; for example, the example run in Section 2 has 5400 neurons in the hidden level. (See Fig. 1)

We fix some notation, to aid in describing our use of the model.

Let m be the length (number of bits) of an address; (in the experiments in Section 2, m = 64.

c will be the length of a code (in the experiments, $c = 8$).

α will denote an m-bit pattern.

$\underline{\alpha}$ will denote a sequence of items $(\alpha_1, \alpha_2, \ldots, \alpha_j)$, each of m-bit length.

$\hat{\alpha}$ will denote a (c-bit) code associated with the m-bit pattern α.

α^+ will denote the ($c + m$ bits) concatenation of α and $\hat{\alpha}$.

Abusing notation somewhat, we shall denote by $\underline{\hat{\alpha}}$ the collection of $(\hat{\alpha}_1, \hat{\alpha}_2, \ldots, \hat{\alpha}_j)$ and by α^+ the sequence $(\alpha_1^+, \alpha_2^+, \ldots, \alpha_j^+)$.(Later on we shall define and use $\mathrm{Mg}(\hat{\alpha})$, for MAGNIFY $(\hat{\alpha})$.)

In our case each memory location consists of $m + c$ registers, i.e. the number of neurons on the output level is $m + c$. (m/c is the upper limit for the number of items in a sequence which can be handled by a single operation of the sort to be discussed.) This is a slight deviation from the most common organization of SDM where each location has m bits (i.e. address length = number of counters). This enhances the flexibility of the model since from any location operations will apply either to the first m bits or to the last c bits of the output level.

We assume that the system is capable of assigning a different code to each distinct, frequently-occurring write data. [4]

Since the address length is m there are 2^m 'virtual' locations of which, as is usual in SDM, a small random sample ('hard' locations) is physically realized. (This is the number of neurons in the hidden level.)

1.1 Operations on the memory

Our memory has the following basic operations:

- WRITE
- READ
- TRUNC
- CHUNK (with components EXTRACT, JOIN)
- EXPAND (with components MAGNIFY, BREAK)

These are described briefly below.

WRITE WRITE to memory (data at address): This is as in standard SDM ([8]). That is, an address is supplied to the memory; all 'hard locations' (hidden level neurons) sufficiently close to the address respond and update their storage (i.e. weights between levels 2 and 3) according to the data. When the codes are given by the system, the external data and address would both be of m bits.

READ READ from memory (at an address): This is as in standard SDM, except that the output has $m + c$ bits; i.e. longer than the m-bit input.

TRUNC TRUNC is a new operation, which takes the first m bits of an $m + c$ bit vector and ignores the last c bits. Since (see [8]), *rehearsals* (i.e. a loops of repetitive READ operations, each taking the output of the last iteration as an input), are done in standard SDM memories to converge to the best retrieval; here we have to input only the first m bits of the READ output – hence we combine our READ with TRUNC. So, our rehearsal is a READ-TRUNC-READ-TRUNC... loop.

[4] This code should not be confused with data-tagging (proposed by [11]), since the extra code bits are truncated in rehearsals, and therefore can not help to enhance resolution between close patterns.

Aside: WRITE and READ are the original names given by Kanerva; they are a little bit confusing (when x reads a book, x WRITEs to her memory...). Hence we shall sometimes use 'store' for 'WRITE' and 'retrieve' for 'READ'; however, these terms will be used also in the context of the multi-level connections, while READ and WRITE will apply exclusively to individual-item operations. (When 'WRITE' or 'store' is used with an m-bit argument, the intention is that the information is entered at the first m bits of the memory location, while the remaining c bits are irrelevant).

CHUNK CHUNK is a new operation that takes several 'low level' items in memory (each $m+c$ long), and produce an (m-bit) 'high level' address. It is a double-stage process consisting of several uses of EXTRACT and JOIN.

EXPAND EXPAND (also new) takes one 'high level' item in memory and returns the addresses of 'low level' associated components. It is double-stage process consisting of BREAK and several uses of MAGNIFY.

We place 'high level' and 'low level' in quotation marks because the memory is homogeneously organized; i.e. there is no architectural distinction between the levels of information. That is one of the main points of this work, showing how multilevel processing can be done in a natural homogeneous environment.

EXTRACT EXTRACT takes from an $(m+c)$-bit vector only its code part, the last c bits; it is the complement of TRUNC. That is, EXTRACT $(\alpha^+) = \hat{\alpha}$.

JOIN JOIN takes several codes (which were previously EXTRACTed) and concatenates them to a single address. Now several EXTRACTions (typically from several READ outputs) and one JOIN, make a complete CHUNK. Thus CHUNK $(\underline{\alpha}^+) \equiv$ JOIN $\hat{\alpha} \equiv$ JOIN $(\hat{\alpha}_1, \ldots \hat{\alpha}_j)$. [5]

BREAK BREAK takes an m-bit vector and separates it to (m/c) c-bits (e.g. a 256-bit vector can be BREAKed into 8 codes of 32 bits each).

MAGNIFY MAGNIFY takes a single code and duplicates it in some predefined variations to form an m-bit address (the copies are concatenated). The only formal requirement on these variations is that each be given by a 1-1 transformation. However, simple repetition is not advisable since repetitive structures are highly above random in 'real' inputs, which would lead to early clustering. This is pertinent in SDM memories since their capacity is adversely affected by such clustering [11]. Our results are not sensitive to the particular choice of such transformations; specifically we used combinations of bitwise negation and bit-subgroup order changes. (Additional discussion of

[5] A technical problem of size matching arises when $j < m/c$. Since CHUNK concludes with an m-bit address, we need to JOIN m/c codes, while we have only j. Our way out was to JOIN several copies of the codes-sequence plus a special 'end-of-sequence' code. See the later translation diagrams (5,6) to get the idea; the '-' is the 'end-of sequence' sign in those diagrams, added automatically when chunking short words.

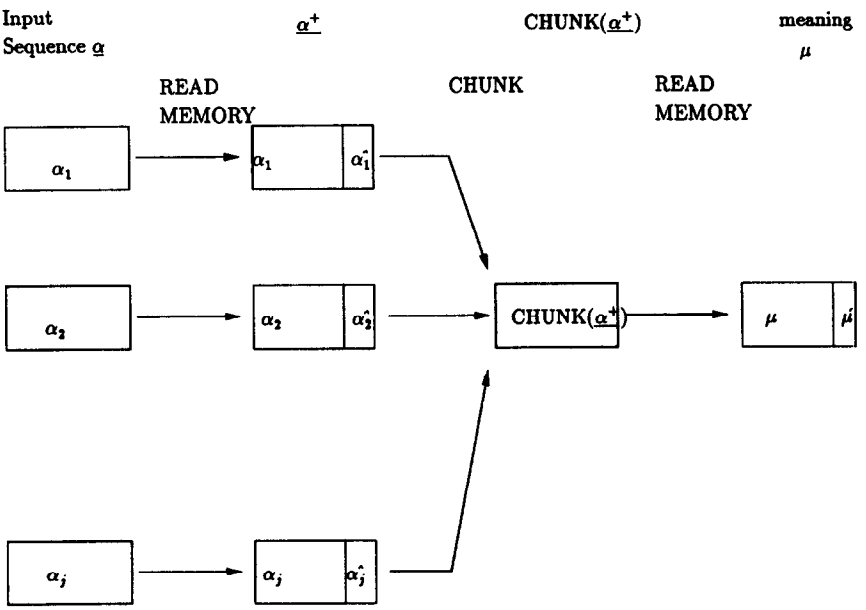


Fig. 2. Sequence-meaning association (general).

MAGNIFY and JOIN is presented at the end of the paper.) We shall abbreviate MAGNIFY ($\hat{\alpha}$) by Mg ($\hat{\alpha}$). One BREAK and several MAGNIFYs are needed to complete an EXPAND operation. MAGNIFY has the property that EXPAND (CHUNK ($\underline{\alpha}^+$)) = Mg($\underline{\hat{\alpha}}$) (i.e. the sequence Mg($\hat{\alpha}_1$),..., Mg($\hat{\alpha}_j$)).

1.2 Entries in memory

During the learning phase, the following information is stored:

(1) If α is a 'known' pattern, store α^+ at location α. (This is our version of self-addressing. Recall α^+ is α followed by its code $\hat{\alpha}$.)
In addition, store α at location MAGNIFY ($\hat{\alpha}$) [6]. Recall that MAGNIFY has the property that EXPAND (CHUNK ($\underline{\alpha}^+$)) = Mg($\underline{\hat{\alpha}}$) (i.e. the sequence Mg($\hat{\alpha}_1$),..., Mg($\hat{\alpha}_j$)).

(2) When a sequence $\underline{\alpha} = (\alpha_1, \ldots, \alpha_j)$ is to be associated with a 'meaning' μ (some associated bit pattern), store μ at CHUNK ($\underline{\alpha}^+$).

(3) If a bit-pattern μ is to be associated with a sequence $\underline{\alpha}$, store CHUNK ($\underline{\alpha}^+$) at location TRUNC(μ).

1.3 General description of retrieval of meaning and sequence

Retrieval of 'meaning' from a sequence: (See Fig. 2). We start with a given sequence $\underline{\alpha}$: READ memory at each address α_1, α_2,..., α_j. Since we wrote α^+ at each α, (see (1)

[6] Note that the code part of the storage is missing. Here, it is unused. But such 'unused' code areas are in fact of great importance, since through them other, more complicated connections can be made.

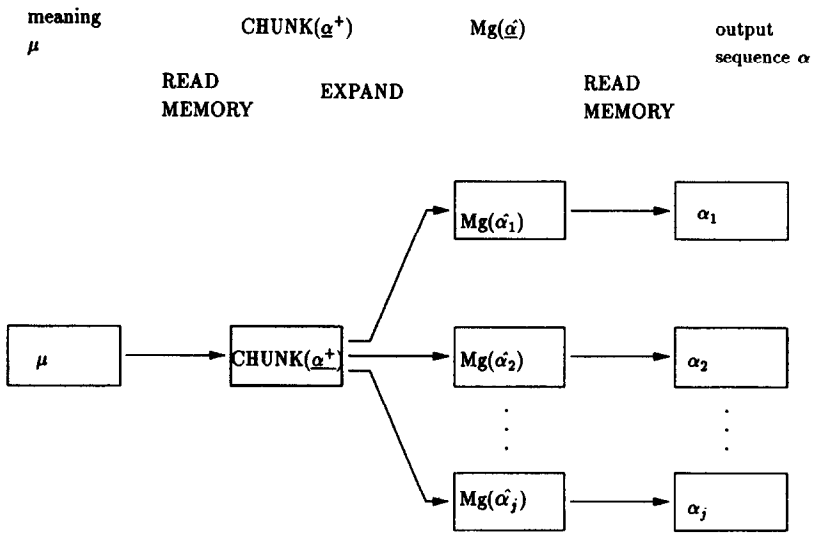


Fig. 3. Meaning-sequence association (general).

above), the outputs will be $(\alpha_1^+, \alpha_2^+, \ldots, \alpha_j^+)$. CHUNK these to form CHUNK $(\underline{\alpha}^+)$. Finally make another READ at the address CHUNK $(\underline{\alpha}^+)$; since we stored μ at the same address (according to (2) above), we now retrieve it. In brief, this retrieval is a READ-CHUNK-READ process.

Retrieval of a sequence from 'meaning': (See Fig. 3). We start with a given address μ, or, if it is an $m + c$ bit vector, TRUNC (μ): READ memory at this address will result in CHUNK $(\underline{\alpha}^+)$, which we stored there according to (3) above. Next we EXPAND what we have got; but EXPAND (CHUNK $(\underline{\alpha}^+)$) gives Mg$(\underline{\hat{\alpha}})$ (see (1)), so we get the sequence Mg$(\hat{\alpha}_1)$, Mg$(\hat{\alpha}_2), \ldots,$ Mg$(\hat{\alpha}_j)$, which are j addresses (they are the MAGNIFICATIONS of $\hat{\alpha}_1, \hat{\alpha}_2, \ldots, \hat{\alpha}_j$). According to (1) above, READ at each of these Mg$(\hat{\alpha}_i)$, $1 \leq i \leq j$, will come up with the desired patterns $(\alpha_1, \alpha_2, \ldots, \alpha_j)$, since we stored them exactly there. To sum up, this retrieval is a READ-EXPAND-READ procedure.

1.4 Detailed description of learning and retrieval

Let us take a closer look at the learning phase, as it is supposed to take place in actual operation. The 'known' α's are stimuli that have been entered previously when they were both code-assigned and self-addressed stored (i.e. α^+ at α). Moreover, note that whenever the system assigns a code $\hat{\alpha}$ to α, it MAGNIFYs that code to produce Mg$(\hat{\alpha})$ and stores α at Mg$(\hat{\alpha})$.

When a sequence-meaning association is stored, the elements of $\underline{\alpha}$ are first retrieved (READ), including rehearsal, and CHUNKed to form a single m-bit (address-sized) pattern, CHUNK $(\underline{\alpha}^+)$; at that address we WRITE the 'meaning' μ.

Learning the other way round, high to low level connection, begins with the same READ-rehearse-CHUNK operated on the given $\underline{\alpha}$ (a low level stimuli sequence); only

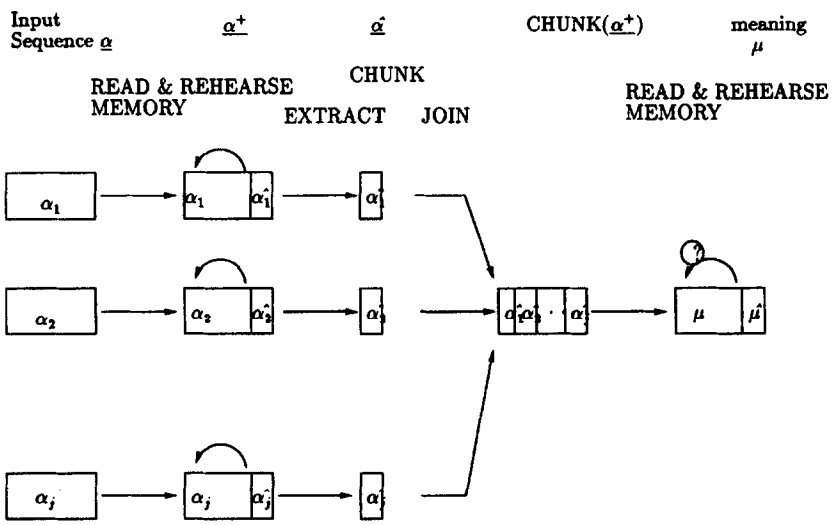


Fig. 4. Sequence-meaning association (detailed).

now WRITE that CHUNK ($\underline{\alpha}^+$) at the given μ, or TRUNC (μ), (the 'meaning') as address. So storing association is a READ-CHUNK-WRITE process, although it is only the WRITE that affects the memory state (the counters).

Now let us look in more detail at the retrieval processes.

Retrieval of 'meaning' from a sequence: (See Fig. 4). We start with a given sequence $\underline{\alpha}$: READ memory at each address $\alpha_1, \alpha_2, \ldots, \alpha_j$, produces $(\alpha_1^+, \alpha_2^+, \ldots, \alpha_j^+)$, as explained earlier. To improve recognition, a rehearsal (always possible with self-addressed locations) is made. CHUNK makes CHUNK ($\underline{\alpha}^+$) ≡ JOIN ($\hat{\alpha}_1, \hat{\alpha}_2, \ldots, \hat{\alpha}_j$), by two stages: first, for each $1 \leq i \leq j$ EXTRACT (α_i^+) to get ($\hat{\alpha}_i$); then JOIN them all. Now READ again at the address CHUNK ($\underline{\alpha}^+$) retrieves the stored μ; μ is also improved by rehearsing. (The question-mark drawn in Fig. 4 on the rehearsal arrow, hints that the rehearsal may be of a more complex type, exemplified in the 2-way-translation (described below)).

Retrieval of a sequence from 'meaning' (See Fig. 5). Start with a given address μ or TRUNC (μ): READ memory at it will result in CHUNK ($\underline{\alpha}^+$), as was shown earlier. CHUNK ($\underline{\alpha}^+$) is rehearsed (again in the simple or complex type, depending on what was stored at it). After this, BREAK the improved CHUNK ($\underline{\alpha}^+$) to the individual codes ($\hat{\alpha}_1, \hat{\alpha}_2, \ldots, \hat{\alpha}_j$); MAGNIFYing each will complete the EXPAND of CHUNK ($\underline{\alpha}^+$). Each such MAGNIFICATION – Mg($\hat{\alpha}_i$), $1 \leq i \leq j$ – serves as an address for READ, again with rehearsal to improve it, and comes up with the stored desired patterns $(\alpha_1^+, \alpha_2^+, \ldots, \alpha_j^+)$. If we wish to output the m-bit long $(\hat{\alpha}_1, \hat{\alpha}_2, \ldots, \hat{\alpha}_j)$ items, we simply TRUNC each.

Remark. Note that the mechanism linking levels is general. That is, there is no limit to (and no design necessity to specify) the number of levels and connection types. In

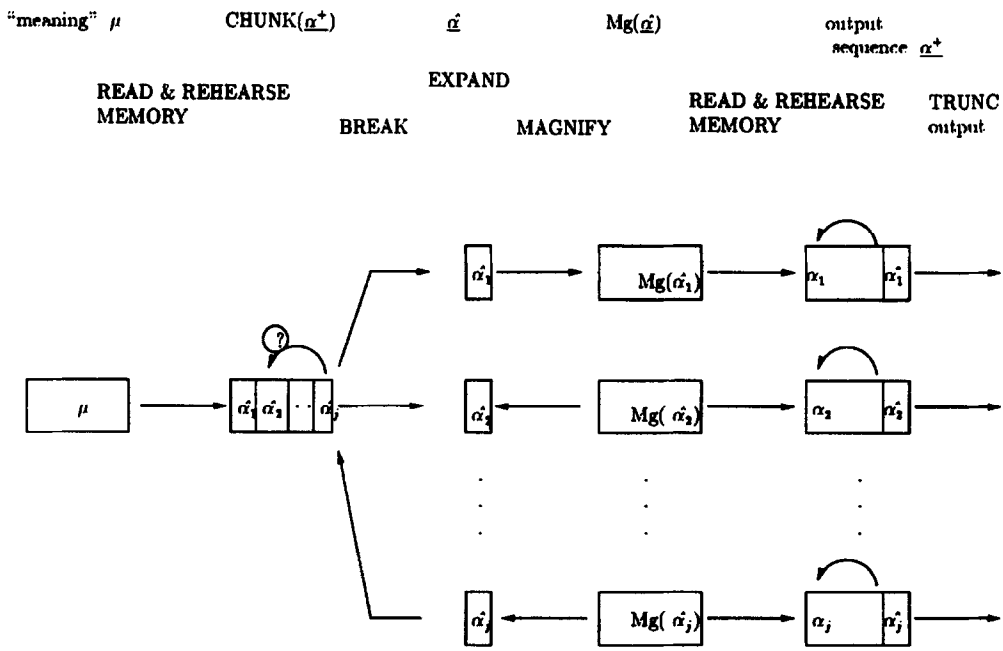


Fig. 5. Meaning-sequence association (detailed).

principle, every data vector written in memory can be code-assigned and connected to other vectors. In fact, the levels are completely external to the model, since the architecture is homogeneous; and, so for example, there is no obstruction to CHUNKing codes from different levels.

2. Language translation as a demonstration

To test these ideas, we chose a cognitive task which it seems must take place in different levels.

Input is a word in English or Hebrew, presented as a sequence of patterns corresponding to letters in the appropriate alphabet; while output should be the translation of the word to the other language, presented as a sequence of letter patterns (in the other alphabet).

To accomplish this task, then, the following must take place:

- Identification of the individual patterns (a 'classical' use of SDM)
- Access of the word from the sequence of letters
- Association of the word with its translation
- Access of the sequence of letter-patterns from the (translated) word

Each word serves as the 'meaning', in the sense of the previous section, associated with the letter sequence of the *other* language. Since there is no correlation between the 'meaning' (i.e. in this case the translation) of a work and its letter sequence, there is clearly a multi-level cognitive task to be performed.

Two versions of application of the model to this task were implemented: One-Way Translation and Two-Way Translation.

2.1 One-way translation

We shall denote the set of source language words by S and the set of destination language words by D. For example, if the input of that particular run is a Hebrew character-sequence (and the desired output is an English character-sequence - the translation) then S is the set of all Hebrew words written to memory for that run and D is the set of all the English words written to memory for that run. (In this work $|S| = |D|$ since each word at either set has exactly one translation-twin in the other set.)

2.2 Entries in memory

During the learning phase, the following information is stored:

For each character α (in either alphabet), store α^+ at α, and also at Mg($\hat{\alpha}$), (in accordance with rule (1) of Section 1, 'Entries in Memory').

For each θ-Φ translation word pair, where $\theta \in S$ and $\Phi \in D$; θ is represented by the character patterns sequence $\underline{\theta} \equiv (\hat{\theta}_1, \hat{\theta}_2, \ldots \hat{\theta}_j)$, and Φ is represented by the character patterns sequence $\underline{\Phi} \equiv (\Phi_1, \Phi_2, \ldots, \Phi_k)$: store JOIN ($\underline{\hat{\Phi}}$) at JOIN ($\underline{\hat{\Phi}}$) (in order to allow rehearsal), and also at JOIN ($\underline{\hat{\theta}}$), ($\underline{\hat{\Phi}} \equiv \hat{\Phi}_1, \hat{\Phi}_2, \ldots \hat{\Phi}_k$), and $\underline{\hat{\theta}} \equiv (\hat{\theta}_1, \hat{\theta}_2, \ldots, \hat{\theta}_j)$; in accordance with rule (2) of Section 1, 'Entries in Memory').

2.3 Translation phase

The basic structure of translation is READ-CHUNK-READ-EXPAND-READ, where each READ is accompanied by rehearsal. Let us follow the details of the example in Fig. 6.

We start with a sequence of four bit-patterns sequence, each graphically representing an English character, 'S', 'T', 'A', 'R'. Memory is READ at the locations of each of those four patterns, producing S^+, T^+, A^+, R^+; each of them is rehearsed. Next the codes $\hat{S}$, $\hat{T}$, $\hat{A}$, $\hat{R}$, are EXTRACTed one by one, and JOINed to form the address ($\hat{S}\hat{T}\hat{A}\hat{R}$) (actually, to complete the full m-bit address, the hyphen '-' code was added by the program, to indicate end of string, and then the codes are re-copied until the vector is filled in. This explains the ($\hat{S}\hat{T}\hat{A}\hat{R}$-$\hat{S}\hat{T}\hat{A}$) CHUNKing; see also footnote 5). READing at that address now, produces the Hebrew counter-part (ˆכˆיˆכ– ˆבˆכˆיˆכ) which has been written at ($\hat{S}\hat{T}\hat{A}\hat{R}$-$\hat{S}\hat{T}\hat{A}$) (since they are a translation-pair). The asterisk '*' at the (ˆכˆיˆכ– ˆבˆכˆיˆכ) code area shows another case where we don't actually use an optional code, but this can be used in other tasks.

(ˆכˆיˆכ– ˆבˆכˆיˆכ) is rehearsed (it was self-addressed written as a destination-language JOIN ($\underline{\hat{\Phi}}$), so it can be rehearsed) and EXPANDed.

EXPANDING starts by BREAKing (ˆבˆכˆיˆכ) to the individual codes כ, י, כ, ב (ignoring those after the hyphen), and MAGNIFYing each of them; the results are the addresses Mg(ˆכ), Mg(ˆי), Mg(ˆכ), Mg(ˆב). But at these locations the individual

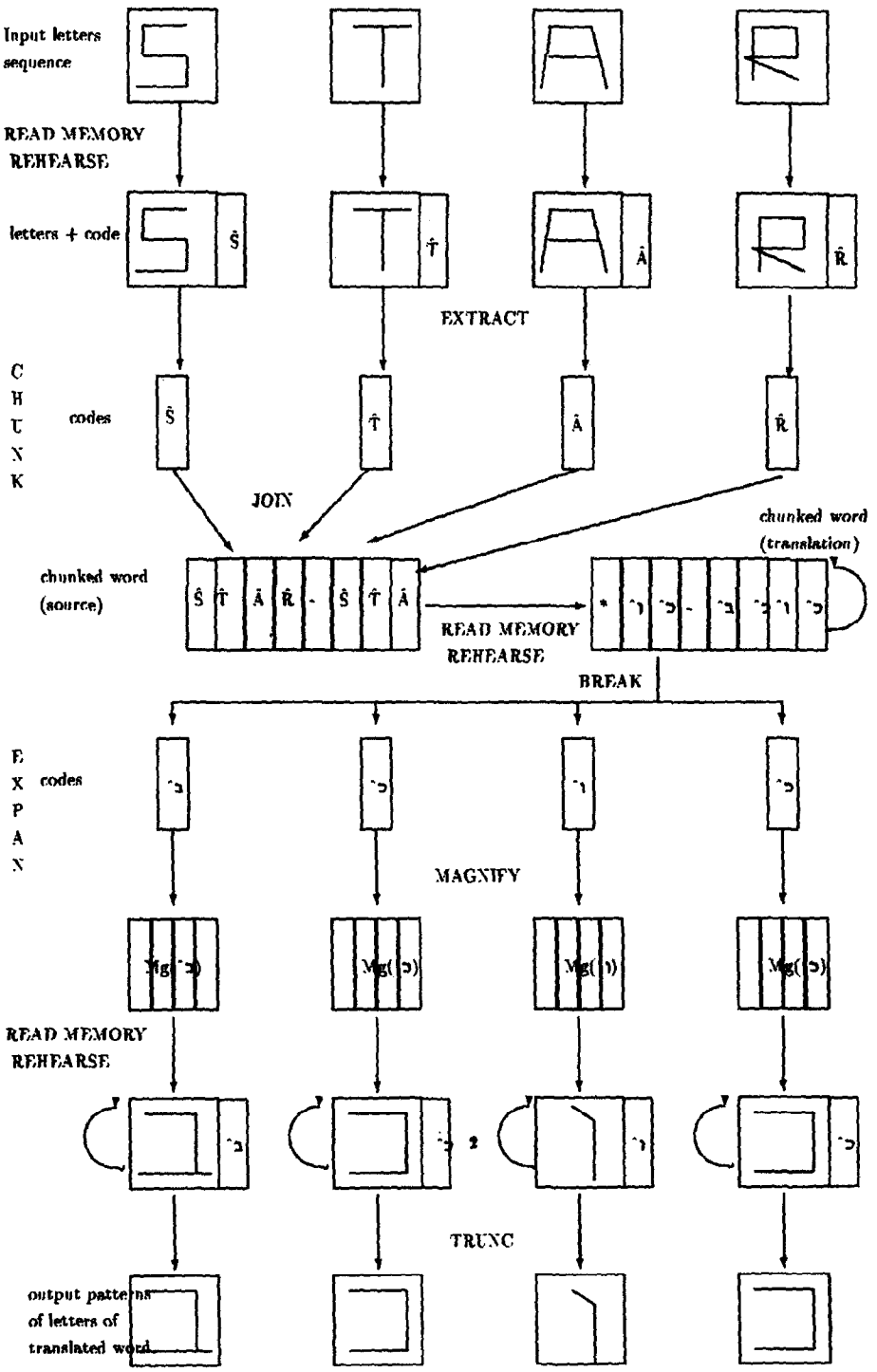

Input letters sequence
READ MEMORY REHEARSE
letters + code
EXTRACT
CHUNK
codes
JOIN
chunked word (translation)
chunked word (source)
READ MEMORY REHEARSE
BREAK
EXPAN
codes
MAGNIFY
READ MEMORY REHEARSE
TRUNC
output patterns of letters of translated word

Fig. 6.

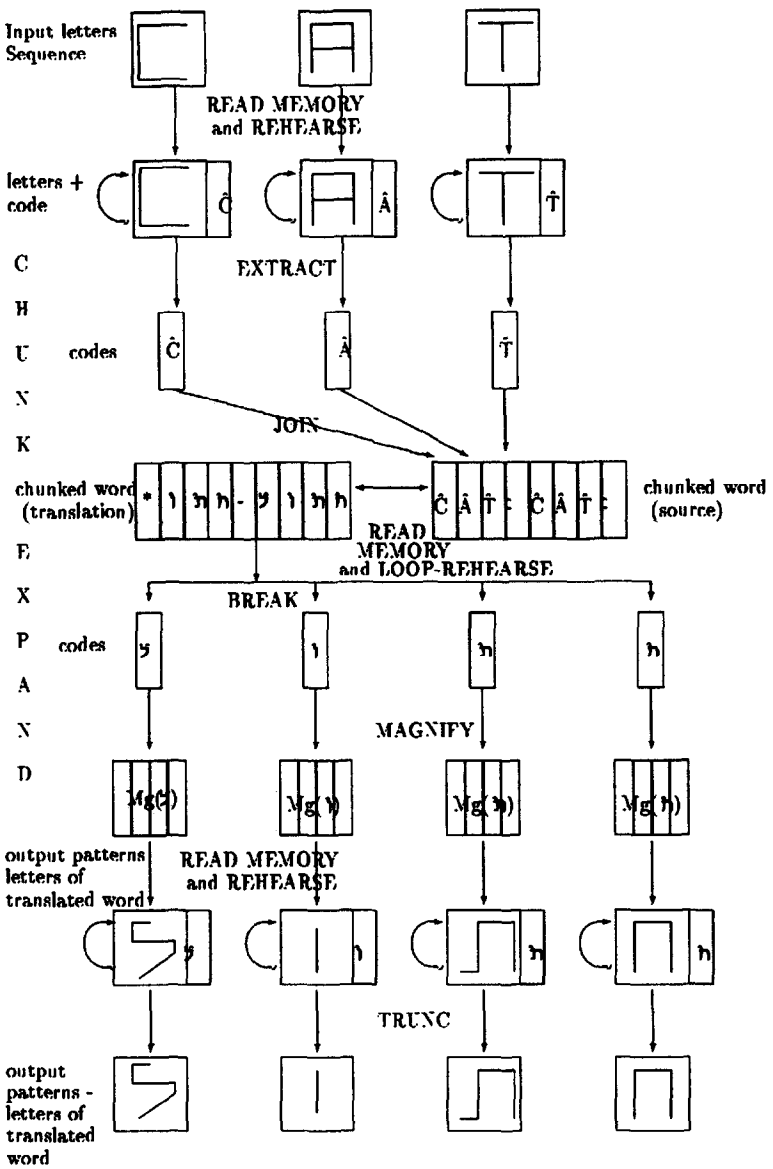


Fig. 7. 2-way translation (small loop).

corresponding Hebrew character-patterns were written, so READ and rehearse (as the letters were also self-addressed written) obtain the character-patterns כ+, ו+, כ+, ב+; so now we just TRUNCate them and get (כ, ו, כ, ב) i.e., the word "כוכב" for output. [7]

[7] Although it is not shown in the diagram, the hyphen ('end of word' sign) itself was also included and printed out (if identified).

The translation included the two kinds of level transitions and an ordinary SDM association at the higher level (English to Hebrew word).

2.4 2-way translation (small loop)

The 1-way translation either works from Hebrew to English or vice versa, but not both with the same memory states; this is because the destination language JOIN ($\underline{\hat{\Phi}}$) (the coded (ˆכˆוˆכ- ˆבˆכˆוˆכ)) is self addressed WRITTEn, which cannot be done for the source language JOIN ($\underline{\hat{\theta}}$) (the coded ($\hat{S}\hat{T}\hat{A}\hat{R}$-$\hat{S}\hat{T}\hat{A}$)), since at location ($\hat{S}\hat{T}\hat{A}\hat{R}$-$\hat{T}\hat{A}$) we stored (ˆכˆוˆכ– ˆבˆכˆוˆכ). Therefore, the only way to make the method symmetric in both directions is mutually write each one of them at the address of the other.

In such a case the rehearsal at the high level is of what we call small loop type: reading at the JOIN ($\underline{\hat{\theta}}$) address retrieves JOIN ($\underline{\hat{\Phi}}$) data and using its TRUNCation as an address retrieves the JOIN ($\underline{\hat{\theta}}$) one etc. As in the simple rehearsal, with less than a critical distance to start (see [8]), and not-too-overloaded memory, convergence by this complex rehearsal is expected, implying successful translation (at the high level) when the number of rehearsals is odd.

Thus mutual translations can be executed by referring to the same state of memory.

To sum up, the entries stored during the learning phase are:

For each individual character α (in either alphabet), store α^+ at α, and also at Mg($\hat{\alpha}$) (in accordance with requirement (1) of Section 1, 'entries in memory').

For each $\theta - \Phi$ translation word pair, where θ is represented by the character patterns sequence $\underline{\theta} \equiv (\hat{\theta}_1, \hat{\theta}_2, \ldots \hat{\theta}_j)$, and Φ is represented by the character patterns sequence ($\underline{\Phi} \equiv \Phi_1, \Phi_2, \ldots, \Phi_k$); store JOIN ($\underline{\hat{\Phi}}$) at JOIN ($\underline{\hat{\theta}}$), and JOIN ($\underline{\hat{\theta}}$), at JOIN ($\underline{\hat{\Phi}}$); where ($\underline{\hat{\Phi}}$) $\equiv (\Phi_1, \Phi_2, \ldots, \Phi_k)$, and ($\underline{\hat{\theta}}$) $\equiv (\hat{\theta}_1, \hat{\theta}_2, \ldots, \hat{\theta}_k)$. (Both according to requirement (2), and to allow the complex loop-rehearsal for JOIN ($\underline{\hat{\theta}}$)-JOIN ($\underline{\hat{\Phi}}$)).

For the translation phase, refer to Fig. 7; the details are quite similar to those of the 1-way case (Fig. 6), except for the loop discussed; the ($\hat{C}\hat{A}\hat{T}$-$\hat{C}\hat{A}\hat{T}$) – (ˆוˆתˆחˆ- ˆלˆוˆתˆח) – ($\hat{C}\hat{A}\hat{T}$-$\hat{C}\hat{A}\hat{T}$) – (ˆוˆתˆחˆ- ˆלˆוˆתˆח)... rehearsal. Still, the basic structure is READ-CHUNK-READ-EXPAND-READ.

2.5 Two-way translation (big loop)

If both Hebrew-English and English-Hebrew translations are simultaneously possible, one can rehearse by starting with one word, translate to the other language, then back and so on. This idea has not been tested yet.

3. Implementation details and experiment design

Program units were written in Turbo-Pascal [8] (4) and run on a 640K memory ordinary PC-compatible. Bits numbers were: $m = 64$ for address and $c = 8$ for code.

[8] The Hamming-distance function was written in Turbo-assembler.

The radius [9] was 21 and 5400 hard locations were used (taking 6 heap segments for counters and almost an entire segment for the array describing the locations). All addresses and data were condensed to bits and handled by bitwise operators.

A pool of 30 word-pairs (listed in Appendix A) was used, with 49 (26 + 23) characters and a hyphen as an 'end of word' inner sign. In part of the trials random stimuli were added to the memory writings, half of them self-addressed and half not.

The small scale of the model implementation, implies severe constraints on memory capacity: at the same time, the nature of the needed entries and the specific task chosen, implies quite a heavy memory load (e.g. to start up one writes to memory some 99 times for the 2 characters sets and hyphen alone - each character pattern at 2 places). Even more critical is the fact that the storing locations and stored items (i.e. character-patterns, chunked word-sequences, and magnified codes) were far-from-randomly distributed, that is, heavily clustered.

On the other hand, such conditions give us the opportunity to take a look at the model performance near its limits. Hence we controlled (and checked memory performance at combinations of) the following parameters:

Distance: Distance of the input letter-patterns from the learned ones. It took two values: 0 (exact input) or 4 ('damaged' randomly). That is, for trials (a 'trial' means a single translation under a specific set of parameters values) in which the parameter 'distance' was assigned the value 4, from each of the letter patterns (64 bits) of the input, 4 bits were randomly chosen and negated, thus creating a pattern of Hamming distance 4 from the original learned pattern. Each new trial with dist = 4 (even with the same 'word' to translate), randomization of the damaged bits was restarted.

Number of Words: By how many word-pairs was the memory loaded at that particular trial time. It took 4 values: 5, 10, 20, 30. That is, if on a particular trial this parameter (abbreviated nW) took the value 10, the trial was run after a learning phase of ten word-pairs. Actually, after such a learning session all members of the learned words-set were presented as an input for translation (each considered a separate trial).

The nW mostly affects the local density aspect of memory load.

Number of Random additions: 0 or 100 additional random writings to memory, stored at the learning phase, to check for general overload effects. Thus it is a parameter of two possible values, $nR = 0$ or $nR = 100$. In some settings of the other parameters, it made some sense to check for $nR = 200$ (as in the 5 words case), and this was done as well. For most of the parameter configurations, however, the results with $nR = 100$ were so poor that it was pointless to go beyond that number.

The overall number T of writings to memory is given by: $T = 99 + 2nW + nR$.

Way: Was the trial run under the 1-WAY or 2-WAY setting. Again it is a double-valued parameter, allowing a comparison of the two settings.

[9] After some disappointing experimentation with greater radii. Some authors (see [11]) adjusted their radius to get approximately $\sqrt{N}$ hard locations (where N is the total number of hard locations) in each select (i.e. within a sphere of that radius). In our case, that will correspond to 73 hard locations; while with $r = 21$ the average number of selected hard locations is 22. Perhaps the close affinity between the letters (in spite of our efforts to draw them in such graphic style that will be taken apart from each other), made it necessary to sharpen memory resolution at the cost of somewhat lowering its error-resistance.

Two other parameters were changed for the sake of the experiment balance and noise reduction rather than an interest in their exact effects:

Hebrew-English: (Abbreviated *H-E*): Which language was the source (input) and which the destination on that particular trial.

File-Order: We have a pool of 30 words; but at values of $nW < 30$, we store (and retrieve) only a subgroup of the pool. That subgroup choice, i.e. who are the 5 words at the $nW = 5$ trials, who the 10, etc., was balanced using 4 controlled versions of the input file (pool) order. However, for $nW = 30$, the changes in *FO* (the abbreviation of File Order), means no more than new randomizations.

So the experiment ran over all the combinations of:

$$\begin{pmatrix}\text{distance} = 0\\ \text{distance} = 4\end{pmatrix} X \begin{pmatrix} nW = 5\\ nW = 10\\ nW = 20\\ nW = 30\end{pmatrix} X \begin{pmatrix} nR = 0\\ nR = 100\end{pmatrix} X \begin{pmatrix}1 - WAY\\ 2 - WAY\end{pmatrix} X \begin{pmatrix}H \to E\\ E \to H\end{pmatrix} X \begin{pmatrix} FO_1\\ FO_2\\ FO_3\\ FO_4\end{pmatrix}$$

with some additional $nR = 200$ trials.

Hence a total of: $2 \times (5 + 10 + 20 + 30) \times 2 \times 2 \times 2 \times 4 = 4160$ (not including the $nR = 200$ trials) word-translation-trials were run.

Unconditioned variables (measuring task performance): Each character in the source stimuli was re-checked after the first rehearsal against the learned pattern; if it was more than 3 bits away, we considered it 'not identified'. So the first variable is the *percentage of detected source characters*, which has nothing to do with the special coding mechanism of our model, but is rather a standard SDM characteristic, will be denoted by *LI*.

A word was considered 'identified completely' if its *translation* characters were all detected in the above sense; a word was considered 'partially identified' if at least half of its characters, but not all, were identified. [10] Only numbers of completely and partially identified *words* were preserved at the destination side, and not of single characters.

Percentage of completely identified words will be denoted by *CW*, and *Percentage of partially identified words* will be denoted by *PW*; the rest (not identified) – by *NW*. Hence $CW + PW + NW = 1$.

4. Results

Experiment results are summarized in Tables 1–4.

For examples of output, see Appendix B.

[10] This criterion is highly above random; On the other hand, with less than half characters identified, one can hardly expect any useful functioning from the word, in further processing.

Table 1
Percentage of identified characters, completely identified words and parrtially identified words, averaged over *HE* and *FO*

Way		1-WAY						2-WAY					
distance		$d=0$			$d=4$			$d=0$			$d=4$		
nR	*nW*	*LI*	*CW*	*PW*	*LI*	*CW*	*PW*	*LI*	*CW*	*PW*	*LI*	*CW*	*PW*
	5	100.	65.	30.	75.4	37.5	22.5	100.	62.5	35.	72.7	32.5	30.
0	10	100.	70.	23.8	72.5	37.5	12.5	100.	68.8	25.	70.3	35.	17.5
	20	99.7	56.9	37.5	68.6	23.7	19.4	100.	69.4	23.7	70.4	34.4	15.6
	30	99.3	56.7	41.7	64.4	18.8	22.9	100.	55.	31.7	65.5	23.3	17.9
	5	97.9	30.	55.	63.1	12.5	22.5	98.9	37.5	45.	65.8	15.	17.5
100	10	86.6	2.5	17.5	42.5	0.	2.5	85.6	0.	11.3	39.	0.	0.
	20	56.3	0.	1.3	19.	0.	0.	61.5	0.	0.	15.7	0.	0.
	30	36.9	0.	0.	8.8	0.	0.	33.8	0.	0.	5.5	0.	0.
200	5	100.	22.5	57.5	54.7	2.5	15.	94.6	23.3	40.	56.	1.7	8.3

LI - Letters Identified.*CW* - Completely Identified Words.*PW* - Partially Identified Words.*NW* - Not Identified Words.*T* - Total No. of Writings to Memory.

Table 2
Percentage of identifications, Averaged over number of words, *HE* and *FO*. Note: Since the majority of trials were made in the last two (20, 30) categories of *nW*, the figures are *biased towards the heavy-load* performance characteristics

Way	1-WAY						2-WAY					
distance	$d=0$			$d=4$			$d=0$			$d=4$		
nR	*LI*	*CW*	*PW*	*LI*	*CW*	*PW*	*LI*	*CW*	*PW*	*LI*	*CW*	*PW*
0	99.6	59.4	36.7	67.8	24.6	20.2	100.	62.1	28.5	68.3	29.2	18.1
100	55.2	2.7	7.3	21.3	1.	2.1	55.3	2.9	5.2	18.4	1.2	1.3

Table 3
Percentage of identifications, averaged over way, *HE*, *FO*

Distance			$d=0$				$d=4$			
nR	*nW*	*T*	*LI*	*CW*	*PW*	*NW*	*LI*	*CW*	*PW*	*NW*
	5	109	100.0	63.8	32.5	3.7	74.1	35.	26.3	38.7
0	10	119	100.	69.4	24.4	6.2	71.4	36.3	15.	48.7
	20	139	99.9	63.2	30.6	6.2	69.5	29.1	17.5	53.4
	30	159	99.7	55.8	36.7	7.5	65.5	21.1	20.4	59.5
	5	209	98.4	33.8	50.	16.2	64.5	13.8	20.	66.2
100	10	219	86.1	1.3	14.4	84.3	40.7	0.	1.3	98.7
	20	239	58.9	0.	0.6	99.4	17.4	0.	0.	100.
	30	259	35.4	0.	0.	100.	7.2	0.	0.	100.
200	5	309	97.3	22.9	48.8	28.3	55.3	2.1	11.7	86.2

Table 4
Percentage of identifications, averaged over way, distance, *HE* and *FO*

	$nR = 0$				$nR = 100$				$nR = 200$			
nW	*LI*	*CW*	*PW*	*NW*	*LI*	*CW*	*PW*	*NW*	*LI*	*CW*	*PW*	*NW*
5	87.	49.4	29.4	21.2	81.4	23.8	35.	41.2	76.3	12.5	30.2	57.3
10	85.7	52.8	19.7	27.5	63.4	0.6	7.8	91.6				
20	84.7	46.1	24.1	29.8	38.1	0.	0.3	99.7				
30	82.3	38.5	28.5	33.	21.3	0.	0.	100.				

5. Discussion

(I) The potential of the model was experimentally demonstrated. Multilevel processing can be done in principle and in practice.
The results were sensitive both to load and to noise in the input stimulus; at least part of the reason for this is due to the limitations of the implementation and the nature of the test items (i.e. small scale and highly correlated input). Further discussion follows below.

(II) The 1-way and 2-way versions achieved approximately the same level of performance; the differences are insignificant. This illustrates the flexibility and richness of the model. That is, changing the task somewhat was simple and did not affect performance.

(III) When the loading is not very close to the memory capacity limits, the performance of the multilevel task is in the same order of magnitude as the performance of the standard (single item), low level pattern recognition task. Obviously it must be lower, since the accomplishment of the multi-level task depends on many single level tasks.
Look, for example, at Table 4, where the performance in the standard task is measured by *LI*, and that of the multi-level by *CW* and *PW*. In the not-too-loaded conditions, the sum of $CW + PW$ is pretty close to *LI*. As loading increases, there appears a sharp decrease in the multilevel performance.

(IV) In analyzing the interactions among factors in our experiment, the most interesting is the strong mutual enhancement of the load parameters *nR*, *nW*. The combined effect of general and local loads is much greater than the sum of each one of them alone. This is clearly seen in Table 4.

(V) The critical factor in loading is not the number of writings (T) to memory per se, but rather clustering – local density of close patterns. (In the context of the present work this is not so obvious, since one could expect that local density will affect only a specific stage of the process, while overall loading will harm every stage independently). The crucial role of clustering was revealed through:
(i) The relatively great effect of *nW* (as opposed to *nR*), that is, the local load at the high level elements region.
(ii) In earlier version of the 1-Way translation, we did not store the character patterns of the source language at their codes MAGNIFY. That is, if we translated from Hebrew to English we did not need a way to reconstruct Hebrew characters

from a word. In the final 1-WAY version we made that store only to make a fair comparison between the 1-WAY and the 2-WAY. So we have earlier results in which there were 23 writings less – writings which both their address (MAGNIFied codes) and their data (characters) are heavily crowded.

Comparison of the results of this early version with the later 1-WAY yields that these 23 writings had a dramatic effect: e.g. in the earlier version, for $nR = 100$, $nW = 30$, $d = 4$, 60% of the words were partially or completely identified (vs. none even with $d = 0$ in the final version); with $d = 0$, we still found a considerable performance at the $nR = 200$, $nW = 30$ conditions and $nR = 300$, $nW = 5$ conditions. So it turns out that these 23 writings had a very serious effect.

Now this sensitivity at the $\mathrm{Mg}(\hat{\alpha})$ area deserves closer attention. It seems to be an artifact of the particular technical choice of JOIN and MAGNIFY.

The code is the most vulnerable part of the described mechanism, due to its small dimensions and because it does not take part in the rehearsals (TRUNCed away). The codes in our implementation were of size $c = 8$, and the nearest neighbours between them were 2 bits apart. (This was inevitable since 50 codes were needed and the maximal number of 8-bit codes which are not less than 3 bits apart is only 16.)

MAGNIFY was chosen so that if two codes, $\hat{\alpha}$, $\hat{\beta}$ are k bits apart then $\mathrm{Mg}(\hat{\alpha})$, $\mathrm{Mg}(\hat{\beta})$, are km/c apart, which seems to be too close (in our case, 16 bits). Similarly, if $\underline{\hat{\theta}}$ and $\underline{\hat{\Phi}}$ are two sequences of m/c codes each, separated by a total distance d, then JOIN $\underline{\hat{\theta}}$ and JOIN $(\underline{\hat{\Phi}})$ are also separated by distance d.

However, different choice of these functions could have been made which increases the separation of close codes. (The major constraint is to choose JOIN so that its inverse BREAK is easily calculated.)

6. Summary

The idea of using a single homogeneous SDM memory to carry out multilevel information processing, was successfully demonstrated by a translation task. The model also proved to be flexible – changing the task somewhat did not affect performance. The performance of the model near its capacity limits was experimentally investigated, and local density seems to play the most critical role.

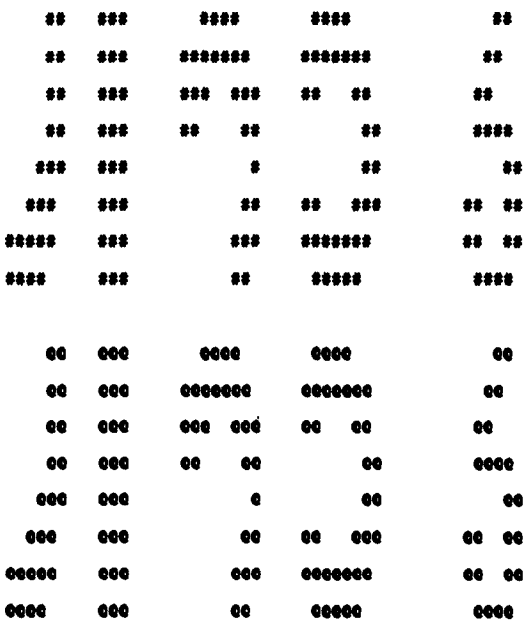
Appendix A: List of input words

1.	LETTER	אות
2.	JOKE	בדיחה
3.	BIG	גדול
4.	FIREFLY	גחלילית
5.	MODEL	דגם
6.	MEMORY	זכרון
7.	CAT	חתול
8.	NATURE	טבע
9.	RELATION	יחס
10.	STAR	כוכב
11.	HEART	לב
12.	BRAIN	מוח
13.	COMPUTER	מחשב
14.	CAR	מכונית
15.	POLITE	מנומס
16.	GUM	מסטיק
17.	POINT	נקודה
18.	THRESHOL(D)	סף
19.	ON	על
20.	EVENING	ערב
21.	OPEN	פתוח
22.	COLOR	צבע
23.	FROG	צפרדע
24.	SET	קבוצה
25.	WIZZARD	קוסם
26.	BOX	קופסה
27.	VISION	ראיה
28.	VARIETY	רבגוניות
29.	QUESTION	שאלה
30.	CELL	תא

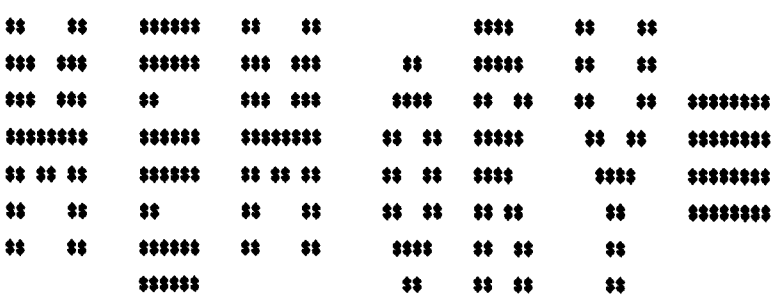
Appendix B: Translation output examples

The four examples below shows two cases of complete word identification (one with exact input, i.e. $d = 0$; the other three with damaged input, $d = 4$), one case of partial word identification, and one case of identification failure. In each example, the first word shown (using the '#' character) is the input – the bit pattern sequence fed into the

Example 1. Hebrew (זכרון) to English (MEMORY); exact stimulus (d = 0); complete word identification.

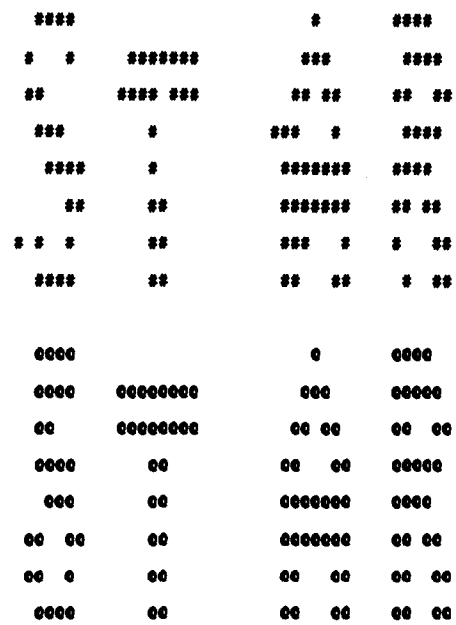


```
5 HEBREW LETTERS IDENTIFIED. 0 NOT. WORD IDENTIFIED COMPLETELY
```

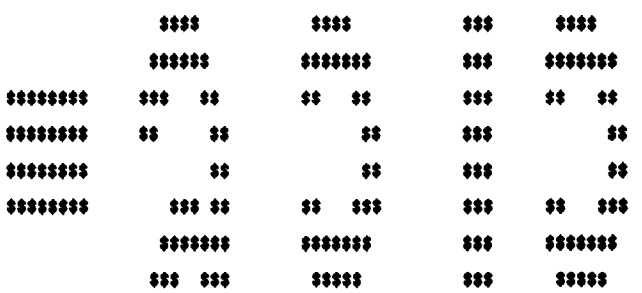


translation process. The second sequence (using '@' character) is the result of the first READ and rehearse – the ordinary SDM pattern recognition task. The last patterns sequence (using the '$' character) is the output of the translation process. Hebrew words were reversed before printing, in accordance with the direction of the Hebrew writing.

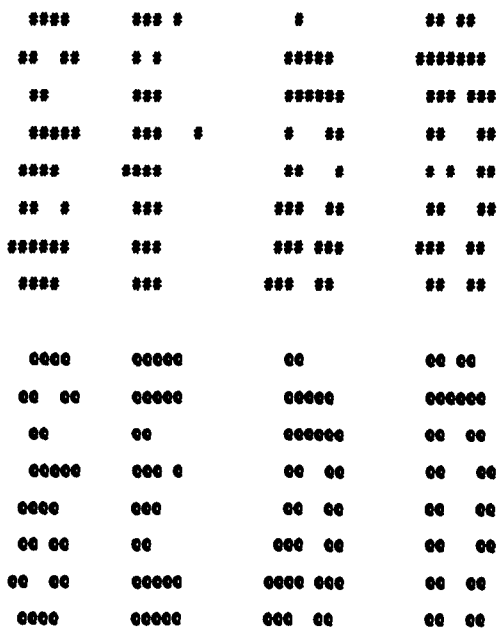
Example 2. English (STAR) to Hebrew (גכוכ) damaged stimulus (d = 4); complete word identification.



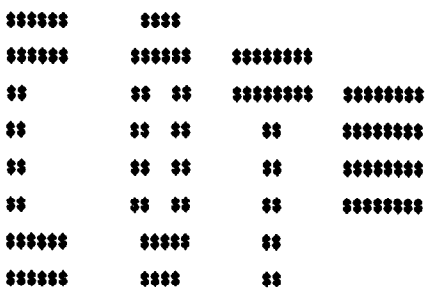
```
3 ENGLISH LETTERS IDENTIFIED. 1 NOT. WORD IDENTIFIED COMPLETELY
```



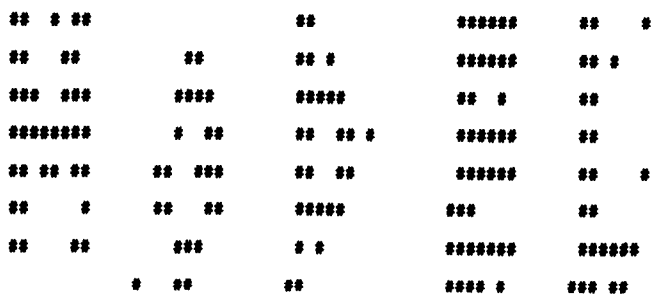
Example 3. Hebrew (חתול) to English (CAT); damaged stimulus (d = 4); partial word identification [2 letters of 3].

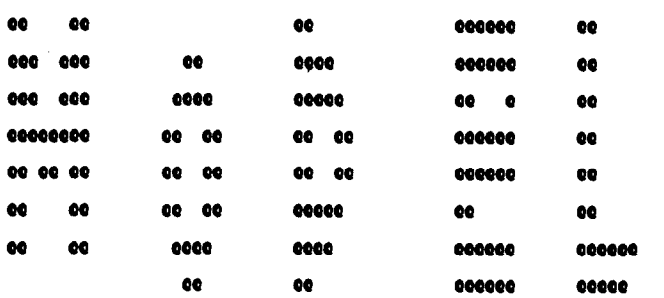


```
3 HEBREW LETTERS IDENTIFIED. 1 NOT. WORD PARTIALLY IDENTIFIED.
```

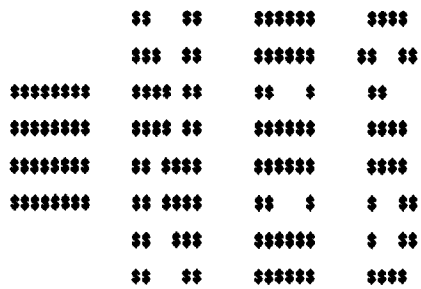


Example 4. English (MODEL) to Hebrew (דגמ); damaged stimulus (d = 4); word identification failed [1 letter out of 3].





```
5 ENGLISH LETTERS IDENTIFIED. 0 NOT. WORD NOT IDENTIFIED.
```



References

[1] D.H. Ackley, G.E. Hinton and T.J. Sejnowski, A learning algorithm for Boltzmann machines, *Cognitive Science*, 9 (1985) 147–169 (reprinted in [3]).

[2] J.S. Albus, *Brains, Behavior and Robotics*. (BYTE Books/McGraw-Hill, Peterborough, NH, 1981).

[3] J.A. Anderson and F. Rosenfeld, *NeuroComputing: Foundation of Research* (Bradford Book, MIT Press, 1988).
[4] J.A. Feldman and D.H. Ballard, Connectionist models and their properties, *Cognitive Science* 6 (1982) 205–254 (reprinted in [3]).
[5] S. Grossberg, How does a brain build a cognitive code? *Psychological Rev.* 87 (1980) 1–254, (reprinted in [3]).
[6] J.J. Hopfield, Neural networks and physical systems with emergent collective computational abilities, *Proc. Nat. Acad. Sci.* 79 (1982) 2554–2558 (reprinted in [3]).
[7] U.D. Joglekar, Learning to read aloud: A neural netword approach using Sparse Distributed Memory, RIACS Technical report 89.27, 1989.
[8] P. Kanerva, *Sparse Distributed Memory* (MIT Press, Cambridge, MA, 1988).
[9] T. Kohonen, *Self-Organization and Associative Memory* (Springer-Verlag, Berlin, 1984).
[10] D. Marr, A theory of cerebral cortex, *J. Physiology* 202 (1969) 437–470.
[11] D. Rogers, Using data tagging to improve the performance of Kanerva's sparse distributed memory, RIACS Technical report 88.1, 1988.
[12] T.J. Sejnowski and C.R. Rosenberg, NETtalk: A parallel NETtalk that learns to read aloud John Hopkins University Technical report JHU/EECS-86/01, 1986.
[13] H. Sompolinski and I. Kanter, *Physical Rev. Letters* Vol. 57 (1986) 2861–2864.

Larry Manevitz is on the faculty of the University of Haifa and has held recent visiting positions at Polytechnic University (Brooklyn Poly), Courant Institute of Mathematical Sciences (NYU), CUNY (Baruch) and NASA (Ames Research Laboratory). His degrees are from Yale University (Ph.D. in mathematics-applied logic), and Brooklyn College, CUNY (B.S.). He has done applied work with IBM. His wife Jenny is a speech pathologist interested in cognitive models of speech and language. They have three children (Zev, Shoul and Miriam) involved in the study of power rangers.

His professional work mostly revolves around developing and applying mathematical models of reasoning and mental processes. He has worked in artificial intelligence, neural networks and mathematical logic.

In artificial intelligence, his work includes several papers on combining information; as well as helping design an expert system for the use of the finite element method. In neural networks he has worked on the development of a new general learning method; and applied neural networks as well to the finite element method, and to an automatic translation device. He has also designed an associative memory model appropriate for temporal storage and retrieval. He has applied the logic of 'non-standard analysis' to solve problems in group theory, topology and stochastic processes. He has also done foundational work in the logical structure of number theory.

Yigal Zemach was born in Haifa, Israel 1953; and now lives in Or-Aqiva, Israel. He received the B.A. degree in mathematics at the Haifa University (1992) in the computer-science course. Since 1992 he has been a software engineer at Intel, Israel Development Center.