SOCIETÀ NAZIONALE DI SCIENZE, LETTERE E ARTI IN NAPOLI
MEMORIE DELL'ACCADEMIA
DI SCIENZE FISICHE E MATEMATICHE

LIGUORI EDITORE

SOCIETÀ NAZIONALE DI SCENZE, LETTERE E ARTI IN NAPOLI
MEMORIE DELL'ACCADEMIA
DI SCIENZE FISICHE E MATEMATICHE
3

F.E. LAURIA R. PREVETE
INFM & Dipartimento di Scienze Fisiche, Università di Napoli Federico II
Complesso Universitario di Monte Sant'Angelo, Via Vicinale Cupa Cintia 16
I-80126 Napoli, Italy (EU)
lauria@na.infn.it
http://www.na.infn.it/Gener/cyber/nnet.html

AN ALGEBRAIC APPROACH TO THE AUTONOMOUSLY SELF-ADAPTABLE BOOLEAN NEURAL NETS

LIGUORI EDITORE

Pubblicato con i contributi della Regione Campania e del Ministero per i Beni Culturali

Tutti i diritti sono riservati. Nessuna parte di questa pubblicazione può essere tradotta, riprodotta, copiata o trasmessa senza l'autorizzazione scritta dell'editore. L'AIDRO (Associazione Italiana per i Diritti di Riproduzione delle Opere dell'Ingegno), via delle Erbe 2, 20121 Milano, potrà concedere una licenza di riproduzione a pagamento per una porzione non superiore a un decimo del presente volume.

Prima edizione italiana Aprile 2001
Liguori Editore, Srl
via Posillipo 394
I 80123 Napoli
http://www.liguori.it

Copyright © Accademia di Scienze matematiche Fisiche e Naturali, 2001

Lauria, Francesco E. :
An algebraic approach to the autonomously self-adaptable Boolean neural nets/Francesco E. Lauria, Roberto Prevete
Napoli : Liguori, 2001
ISBN 88 - 207 - 3266 - 1

1. Boolean nets 2. Automously adaptable I. Titolo

Ristampe:

9 8 7 6 5 4 3 2 1 0 2006 2005 2004 2003 2002 2001

Questo volume è stato stampato in Italia dalle Officine Grafiche Liguori - Napoli
su carta inalterabile, priva di acidi, a PH neutro, conforme alle norme Iso 9706 ∞.

TABLE OF CONTENT

Figure list IX

Table list X

Chapter 1. **The boolean neural net, or BNN** 1
1.1. Introduction 1
1.2. The McCulloch and Pitts net 2
1.3. The Caianiello' s net 5
1.3.1. The logical connectives 6
1.3.1.1. The BNN computing the logical conjunction 7
1.3.1.2. The BNN computing the logical disjunction 8
1.3.1.3. The BNN computing the existential quantifier in the time domain 8
1.3.1.4. The BNN computing the logical negation 9
1.4. The fault tolerant BNN 10

Chapter 2. **The knowledge representation in the BNN** 13
2.1. Introduction 13
2.2. The BNN as a control structure 14
2.2.1. The primitive BNN 16
2.2.1.1. The BNN controlling the moving commands 17
2.2.1.2. The BNN controlling the write-one command 18
2.2.1.3. The BNN controlling the erase-one command 18
2.2.1.4. The BNN controlling the write-zero command 19
2.2.1.5. The BNN controlling the erase-zero command 20
2.2.1.6. The BNN controlling the write-+ command 21
2.2.1.7. The BNN controlling the erase-+ command 21
2.2.2. The adder BNN 22
2.3. Conclusions 30

Chapter 3. **The data structure embedded in a BNN** 33
3.1 Introduction 33
3.1.1. Names and variables list 36
3.2. The structure components 37
3.2.1. The records and the arrays 37
3.2.2. The DLDS constant part 38
3.2.3. The DLDS variable part 38
3.2.3.1. The linked lists as intermediate structures. 38
3.2.3.2. The key holding structures 40
3.2.3.2.1. The key trees 40
3.2.3.2.2. The key vectors 40
3.2.4. The intermediate three-records order 41
3.2.5. The keys order 41
3.2.6. An illustrative example 41
3.3. The reading access 42
3.3.1. Reading a sequence of symbols from a record 42
3.3.2. Reading a linked list three-record content 43
3.3.3. Reading a key 43
3.3.4. Reading an item 43
3.3.5. The cost to retrieve and to read an item 44
3.3.6. Retrieving and reading the conjunction and the disjunction of keys 44
3.3.6.1. The conjunction case and its cost 45
3.3.6.2. The disjunction case and its cost 46
3.3.6.3. The logical negation case and its cost 47
3.4. The writing access 47
3.4.1. Inserting the first item in an empty DLDS 48
3.4.1.1. The cost of inserting the first item in an empty DLDS 49
3.4.2. Inserting an item in a non-empty DLDS 49

3.4.2.1. The searching procedure terminates on an empty record 50
3.4.2.2. The searching procedure terminates on a non-empty record 51
3.4.2.3. The cost of inserting an item in a non-empty DLDS 52
3.4.3.3.1. The most favorable case 52
3.4.2.3.2. The most unfavorable case 53
3.4.3. Reordering a dirty key vector 53
3.4.3.1. The cost of reordering a dirty key vector 56
3.4.4. The reordering of a DLDS and its cost 56
3.4.5. The case of equal keys 57
3.5. The erasing access 58
3.5.1. Erasing an item from a DLDS organized with vectors 59
3.5.2. Erasing an item from a DLDS organized with red-black trees 60
3.5.3. The cost of erasing an item from a DLDS 60
3.6. To insert or to erase keys 60
3.6.1. Inserting a new key in all the items 60
3.6.2. The cost of inserting a new key 61
3.6.3. Erasing a key from all the items 62
3.6.4. The cost of erasing a key 62
3.7. Conclusions 62

Chapter 4. **The Caianiello formalism and the Hebbian rule** 63
4.1. The facilitatory connection case 64
4.1.1. The Hebbian rule 64
4.1.1.1. The first Hebbian rule 66
4.1.1.2. The second Hebbian rule 67
4.1.2. The Hebbian rule and the logical connectives 69
4.2. The inhibitory connection case 69
4.3. A necessary condition for the implementation of the Hebbian rule in a BNN 70

Chapter 5. **The adaptable BNN, or ABNN** 73
5.1. Introduction 73
5.2. Some problems arising with a Hebbian rule implementation 75
5.2.1. An informal discussion 75
5.2.2. A formal approach 76
5.3. On the sufficient conditions for an ABNN 77
5.3.1. The IBNS activity and the ABNN input 78
5.3.2. The Hebbian sets 79
5.3.3. The parent and the offspring assemblies 80
5.4. The six node IBNS 80
5.4.1. The six node assembly blueprint 80
5.4.2. The six node assembly functioning 81
5.4.3. The six node primitive 83
5.4.4. The six node primitive internal activity 83
5.4.5. The mask for the abnn organized in six node IBNSs 83
5.4.6. The six node ibns pitfalls 85
5.5. The sixteen node IBNS 86
5.5.1. The sixteen node assembly 86
5.5.2. The mask and the control for a sixteen node ABNN 89
5.5.3. The sixteen node ABNN primitive 90
5.5.4. The sixteen node IBNS pitfalls 91
5.6. The general assembly 93
5.6.1. Introduction 93
5.6.2. The general assembly 94
5.6.2.1. The first case 100
5.6.2.2. The second case 100
5.6.3. The general assembly functioning 100
5.6.3.1. The first phase algorithm 101
5.6.3.2. The second phase algorithm 104
5.6.4. The general assembly wiring diagram 108
5.6.5. The general assembly input control subnet 112
5.6.5.1. The input control first task 112
5.6.5.2. The input control second task 113
5.6.5.3. The input control third task 114
5.6.5.4. The input control fourth task 114

5.6.6. The general assembly output control subnet 116
5.6.6.1. The general assembly output control first task 116
5.6.6.2. The general assembly output control second task 117
5.6.7. The general assembly store subnet 117
5.6.8. The general assembly "or" subnet 117
5.6.9. The general assembly compare subnet 118
5.6.10. The general assembly reset subnet 118
5.6.11. The primitive subnet 118
5.6.12. The primitive input control subnet 119
5.6.13. The primitive output control subnet 120
5.6.13.1. The primitive output control first task 121
5.6.13.2. The primitive output control second task 122
5.6.14. The general assembly pitfalls 122
5.7. A formal characterization 123
5.7.1. The finite state machine immanent in a GABNN 123
5.7.1.1. The next state function 124
5.7.1.2. The next output function 124
5.7.1.3. The GABNN formal characterization at work: an illustrative example 125

Chapter 6. **The GABNN as an autonomously self-trainable control system** 127
6.1. The neural network applications 127
6.1.1. The GABNN applications 127
6.1.2. The GABNN and the writing robot 128
6.2. Some auxiliary training procedures 129
6.2.1. The GABNN and the controlled robot environment 129
6.2.2. The primitive pair training 131
6.2.3. The macro training 132
6.2.3.1.1. Training the GABNN to control the robot large movements 132
6.2.3.1.2. Training the GABNN to control the robot extra large movements 133
6.2.3.1.3. Training the GABNN to control the robot megamovements 134
6.2.3.1.4. Training the GABNN to control the robot almost megamovements 135
6.2.3.2. Training the GABNN to output the command write-one 136
6.2.3.3. Training the GABNN to output the command erase-one 137
6.2.3.4. Training the GABNN to output the command write-zero 138
6.2.3.5. Training the GABNN to output the command erase-zero 140
6.2.3.6. Training the GABNN to output the command write-+ 142
6.2.3.7. Training the GABNN to output the command erase-+ 143
6.2.3.8. Training the GABNN to output the command write-x 145
6.2.3.9. Training the GABNN to output the command erase-x 146
6.2.3.10. Training the GABNN to mark an environment vector component 148
6.2.3.11. Training the GABNN to erase the mark from an environment vector component 150
6.2.3.12. Training the GABNN to erase an environment vector component 150
6.2.3.13. Training the GABNN to erase an environment vector 153
6.2.3.14. Training the GABNN to copy an environment vector component unitary square 155
6.2.3.14.1. Training the GABNN to copy an unitary square one and five vectors below, up 155
6.2.3.14.2. Training the GABNN to copy one unitary square five vector less one component below, up 156
6.2.3.14.3. Training the GABNN to copy one unitary square on the next component at the left 158
6.2.3.15. Training the GABNN to copy an environment vector component 160
6.2.3.16. Training the GABNN to copy an environment vector 164
6.2.3.17. The training procedures implemented 166
6.3. Training the GABNN to compute the addition 170
6.3.1. Training the GABNN to respond to (+, b, 0, v, b) and to (+, b, v, 0, b) 171
6.3.2. Training the GABNN to respond to (+, b, v, 1, b) and to (+, b, 1, v, b) 173
6.3.3. Training the GABNN to respond to (+, 1, v, v', b) 174
6.3.4. Training the GABNN to respond to (+, b, 1, 1, b) 176
6.3.5. Training the GABNN to respond to (+, 1, v, 1, b) and to (+, 1, 1, v, b) 178
6.3.6. Training the GABNN to respond to (+, 1, 1, 1, b) 181
6.3.7. Training the GABNN to respond to (+, b, b, b, b) 184
6.4. Training the GABNN to control the conditional actions 185
6.4.1. Training the GABNN to shift an expression 185
6.4.2. Training the GABNN to modify an expression 195

6.4.2.1. To erase a sequence of symbols 195
6.4.2.2. To write a sequence of equal symbols 195
6.4.2.3. To erase a sequence of symbols and then to write one symbol 196
6.4.2.4. An illustrative example: training the GABNN to erase the addition first and second operand 197
6.4.3. The conditional movements 198
6.4.4. Chaining the procedures 202
6.4.4.1. An illustrative example 202
6.4.4.2. A second illustrative example 204
6.4.4.3. Training the GABNN to find an expression central comma 211
6.5. Training the GABNN to compute the multiplication 216
6.6. Training the GABNN to compute the delta function 219
6.7. Training the GABNN to compute the substitution 220
6.8. Training the GABNN to compute the minimalization schema 232

Chapter 7. **The Java package it.na.cy.nnet** 241
7.1. Introduction 241
7.2. The syntax and semantics of the it.na.cy.nnet package 242
7.2.1. The BoolHebbNet class 243
7.2.2. The NeuralNetSim class 246
7.2.3. The classes of the Java package it.na.cy.nnet 249
7.2.3.1. The interface it.na.cy.nnet.NetworkConstants 249
7.2.3.2. The class it.na.cy.nnet.Node 250
7.2.3.3. The class it.na.cy.nnet.RecordNode 252
7.2.3.4. The class it.na.cy.nnet.Connection 253
7.2.3.5. The class it.na.cy.nnet.InfoConnection 254
7.2.3.6. The class it.na.cy.nnet.NetworkVector 256
7.2.3.7. The class it.na.cy.nnet.ConnectionVector 257
7.2.3.8. The class it.na.cy.nnet.Connections 258
7.2.3.9. The class it.na.cy.nnet.NodeVector 259
7.2.3.10. The class it.na.cy.nnet.PossibleActiveNodes 260
7.2.3.11. The class it.na.cy.nnet.RecordNodeVector 261
7.2.3.12. The class it.na.cy.nnet.ActiveNodes 262
7.2.3.13. The class it.na.cy.nnet.Brain 262
7.2.3.14. The class it.na.cy.nnet.History 263
7.2.3.15. The class it.na.cy.nnet.NeuralNetSim 264
7.2.3.16. The class it.na.cy.nnet.BoolHebbNet 269
7.2.3.17. The class it.na.cy.nnet.ReadStream 271
7.2.3.18. The class it.na.cy.nnet.ReadPrintRaf 272
7.3. Evaluating the average runtime 274
7.4. The low level language LLNN 276
7.4.1.The syntax 276
7.4.2.The semantics 277
7.5. Some results 279
7.5.1.The C1, C2, C3 cases 281
7.5.2.The C4, C5, C6 cases 284
7.5.3.Associating one network input to one general assembly 288
7.6. Conclusions 290

Chapter 8. **Conclusions** 291
8.1. Introduction 291
8.2. The Caianiello-Hebb transform 292
8.3. The autonomously self-adaptable boolean neural nets 292

References 295

Index 297

Figure list

		Chapter
Figure 1.1.	*The BNN realizing a delay line*	1.1.
Figure 1.2.	*The net computing the disjunction*	1.2.
Figure 1.3.	*The net computing the conjunction*	1.2.
Figure 1.4.	*The net computing the conjoined negation*	1.2.
Figure 1.5.	*The relative to absolute inhibition conversion*	1.2.
Figure 1.6.	*The BNN computing the N inputs conjunction*	1.3.1.1.
Figure 1.7.	*The BNN computing the N inputs disjunction*	1.3.1.2.
Figure 1.8.	*The BNN computing the existential quantifier in the time domain*	1.3.1.3.
Figure 1.9.	*The BNN computing the logical negation*	1.3.1.4.
Figure 1.10.	*The BNN computing the exclusive or, ie, the xor*	1.4.
Figure 1.11.	*The fault tolerant net computing the xor*	1.4.
Figure 2.1.	*The block diagram of the BNN computing the sum of operands with an unlimited number of digits*	2.2.
Figure 2.2.	*The primitive BNN: the moving commands*	2.2.1.1.
Figure 2.3.	*The primitive BNN: the write-one command*	2.2.1.2.
Figure 2.4.	*The primitive BNN: the erase-one command*	2.2.1.3.
Figure 2.5.	*The primitive BNN: the write-zero command*	2.2.1.4.
Figure 2.6.	*The primitive BNN: the erase-zero command*	2.2.1.5.
Figure 2.7.	*The primitive BNN: the write-+ command*	2.2.1.6.
Figure 2.8.	*The primitive BNN: the erase-+ command*	2.2.1.7.
Figure 2.9.	*The delayed BNN, or dB subnet*	2.2.2.
Figure 2.10.	*The adder subnet responding to the (+, b, v, v', b) environment vector*	2.2.2.
Figure 2.11.	*The adder subnet responding to the (+, b, p, q, b) environment vector*	2.2.2.
Figure 2.12.	*The adder subnet responding to the (+, I, v, v', b) environment vector*	2.2.2.
Figure 5.10.	*The general assembly output control subnet*	5.6.6.1.
Figure 5.11.	*The store subnet*	5.6.7.
Figure 5.12.	*The "or" subnet*	5.6.8.
Figure 5.13	*The compare subnet*	5.6.9.
Figure 5.14	*The n-xor subnet*	5.6.9.
Figure 5.15	*The reset subnet*	5.6.10.
Figure 5.16	*The p-th primitive subnet*	5.6.11.
Figure 5.17.	*The primitive input control subnet*	5.6.12.
Figure 5.18.	*The primitive output control subnet*	5.6.13.1.
Figure 6.1.	*The environment vector*	6.2.1.
Figure 7.1.	*The network model*	7.1.
Figure 7.2.	*Main components of a simulator implemented by the package it.na.cy.nnet*	7.3.
Figure 7.3.	*An instance of a neural network*	7.4.2.
Figure 7.4.	*The sample mean and the standard deviation of the variables x1, x2,x3,d3 in the cases C1, C2 and C3*	7.5.1.
Figure 7.5.	*The d3 values in the cases C1, C2 and C3*	7.5.1.
Figure 7.6.	*Normal test*	7.5.1.
Figure 7.7.	*The sample mean and the standard deviation of the variables x1, x2, x3, x4, x5, d3, d4 and d5 in the case C4*	7.5.2.
Figure 7.8.	*The sample mean and the standard deviation of the variables x1, x2, x3, x4, x5, d3, d4 and d5 in the case C5*	7.5.2.
Figure 7.9.	*The sample mean and the standard deviation of the variables x1, x2, x3, x4, x5, d3, d4 and d5 in the case C6*	7.5.2.
Figure 7.10.	*The d3 sample in the cases C4, C5 and C6*	7.5.2.
Figure 7.11.	*The d4 sample in the cases C4, C5 and C6*	7.5.2.
Figure 7.12.	*The d5 sample in the cases C4, C5 and C6*	7.5.2.
Figure 7.13.	*Normal test*	7.5.2.
Figure 7.14.	*The distribution of the z1 sample in the cases a), b) and c)*	7.5.3.

Table list

Table 2.1.	*The adder input-output correspondence*	2.2.2.
Table 2.2.	*The environment vectors transformation*	2.2.2.
Table 5.1.	*The sixteen node assembly execution sequence*	5.5.1.
Table 6.1.	*The identifiers of the general assemblies introduced in the auxiliary training Procedures*	6.2.3.17.
Table 6.2.	*The gabnn input-output correspondences for the addition*	6.3.
Table 6.3.	*The gabnn input-output correspondences for the addition as a subroutine for the multiplication algorithm*	6.4.4.2.
Table 6.4.	*The gabnn input-output correspondences for the multiplication*	6.5.
Table 6.5.	*The gabnn input-output correspondences for the delta function*	6.6.
Table 7.1.	*An instance of ‹nodeto›*	7.4.2.
Table 7.2.	*An instance of ‹nodedescription›*	7.4.2.
Table 7.3.	*An istance of ‹networkdescription›*	7.4.2.
Table 7.4.	*The mean and the standard deviation of the varibles x1, x2, x3, d3 and d12 in the cases C1, C2 and C3*	7.5.1.
Table 7.5.	*The sample mean and the standard deviation of the variables x1, x2, x3, x4, x5, d3, d4, d5, d12 and d34*	7.5.2.
Table 7.6.	*The mean and the standard deviation of the variable z1 in the cases a), b) and c)*	7.5.3.

cibernetica *T. scient.* la scienza che, movendo dallo studio del sistema nervoso umano, costruisce, seguendo le stesse direttive, le complicate macchine calcolatrici.

[**cybernetics** *scient. T.* the science which, moving from the study of the human nervous system, builds, applying the very same directives, the complicated computing machines]

F. Palazzi – "Nuovissimo Dizionario della Lingua Italiana", Casa Editrice Ceschina (Milano), second ed., (1964), 1406 + xv.

Chapter 1

THE BOOLEAN NEURAL NET, OR BNN.

1.1. Introduction

The *Boolean Neural Nets*, or *BNN*, were first proposed by McCulloch and Pitts in their seminal paper, see [1], as tools to compute a logical calculus. Later on, see [2], Caianiello introduced the *state vector* and the *coupling coefficients matrix*, ie, introduced an effective algebraic formalism to represent a Boolean neural net, see the chapter 1.3.

To obtain a BNN we connect finitely many nodes, by means of monodirectional channels, or *connections*, so to satisfy the following five conditions:

i) the time is a discrete quantity and we call *tau*, τ, its smallest indivisible bit;

ii) to the node k is associated the non negative *threshold* value T_k. If all the BNN nodes have the same threshold, we call it T;

iii) the BNN nodes, as well as the monodirectional output channels stemming out of them, are in one of two possible states: zero-one, active-inactive, firing-silent. That is, we shall say either one of them is in the state zero or in the state one, or it is active or inactive, or it is firing or not firing, or a node is sending a signal, or the output connection is carrying or transmitting, a signal or it is silent or it is active or it is inactive;

iv) at each new tau onset, each BNN node, together with its output connections, assumes one of the two states introduced in iii), depending both on the node threshold value and on the state assumed, the previous tau, by the very same node input connections, ie, by the monodirectional channels impinging onto it. Actually, from each node it stems just one output connection: however, each output connection can have many ramifications impinging onto one or more different nodes. In the neural network literature, e. g. see [3], 1) a node is called an input node if at least one of its input channels originates outside the BNN; 2) a node is called an output node if at least one of its output channel ramifications terminates outside the BNN; 3) any other node is called an hidden node;

v) whereas the state of the output channel connecting the node k to the node h is the same as the node k state, see iii), at each tau, the node h state depends, see iv), on the former being taken with either a positive or a negative sign. If, to compute the node h state, we take the node k state with a positive sign, then, we call *facilitatory* the ramification of the connection between the two nodes. Otherwise, we call it *inhibitory*. An inhibitory or facilitatory attribute associated to the ramification of a connection is chosen once and for all when drawing the BNN blueprint, ie, in a BNN

the quality to be an inhibitory or a facilitatory ramification of a connection is a time independent constant.

Because of iv), the node h activity at the time t depends on the node k activity at the time t - τ . Let us consider the chain of nodes k, h_n, ..., h_1, see the Figure 1.1., such that the activity in one of them causes the next to become active one τ later, accordingly, the activity of the node k is followed by the activity of the node h_1 after n taus. In other words, we can consider the chain of nodes as delay lines storing the signals flowing in the net. We could have the very same result if we built a suitable finite delay in the channel connecting the node k to the node h. As a BNN is composed by finitely many nodes, then, from now on we shall consider, equivalently, both nets with delay lines composed by a finite number of suitably connected nodes as well as nets with delay lines composed by suitably delayed node output connection ramifications, with a given finite upper bound on their latencies. Eventually, in a BNN each node has one output connection and up to as many input connections as the number of delays there are in the communications between the net nodes times the very same nodes number. In other words, in an N node net having, in the communications between any two nodes, the greatest delay D, each node has just one output and no more than D* N inputs.

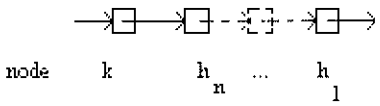


Fig. 1.1. *The BNN realizing a delay line.* The nodes common threshold is T. The arrows represent the facilitatory connections greater than the threshold T. Because of the threshold and of the connection values we have chosen, if the node k is active at the time t, then, the node h_1 is active at the time $t + n\tau$.

To conclude, the function computed by a BNN depends on the nodes thresholds, on the connection ramifications, ie, on the net wiring diagram and on the communications different delays.

1.2. The McCulloch and Pitts formalism.

A McCulloch and Pitts node has one output channel with as many ramifications as needed and receives many input channels. In a McCulloch and Pitts net there is one threshold T for all the net nodes. At the time t, a node fires if the sum of all its facilitatory connections carrying a signal less the sum of all its inhibitory connections carrying a signal is no smaller than the threshold. Otherwise it does not fire. To give a formal representation to the node h firing they introduce the variable $N_h(t)$: it assumes the value one if the node h fires at the time t. Otherwise, it assumes the value zero.

Actually, in [1] McCulloch and Pitts obtain a fundamental result, namely they give the wiring diagrams of the nets computing the well formed formulas of the propositional calculus. In the Figure 1.2. we present the net they found to compute the connective:

$$N_3(t) \equiv N_1(t-\tau) \vee N_2(t-\tau)$$

which we can read: the variable $N_3(t)$ is true at the time t if either the variable $N_1(t-\tau)$ or the variable $N_2(t-\tau)$ or both have been true at the time t minus one tau. In the Figure 1.3. we present the net they found to compute the connective:

$$N_3(t) \equiv N_1(t-\tau) \,\&\, N_2(t-\tau)$$

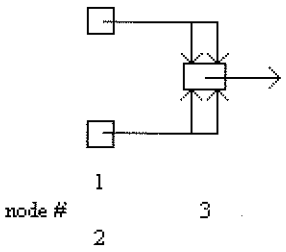


Fig. 1.2. *The net computing the disjunction.* Each arrow represents one facilitatory ramification of a connection. The threshold T is two. Because both the node 1 and the node 2 have a connection impinging with two facilitatory ramifications onto the node 3, the node 3 fires at the time t if either the node 1 or the node 2 or both the nodes 1 and 2 have fired at the time t minus one tau.

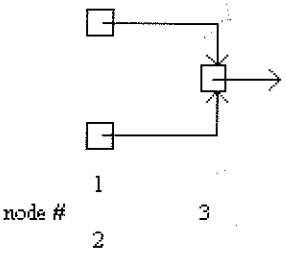


Fig. 1.3. *The net computing the conjunction.* Each arrow represents one facilitatory connection. The threshold T is two. The node 3 fires at the time t if both the node 1 and the node 2 have fired at the time t minus one tau.

which we can read: the variable $N_3(t)$ is true at the time t if both the variable $N_1(t\text{-}\tau)$ and the variable $N_2(t\text{-}\tau)$ have been true at the time t minus one tau. In the Figure 1.4. we present the net they found to compute the connective:

$$N_3(t) \equiv N_1(t\text{-}\tau) \,\&\, \neg N_2(t\text{-}\tau)$$

which we can read: the variable $N_3(t)$ is true at the time t if both the variable $N_1(t\text{-}\tau)$ has been true and the variable $N_2(t\text{-}\tau)$ has been false at the time t minus one tau.

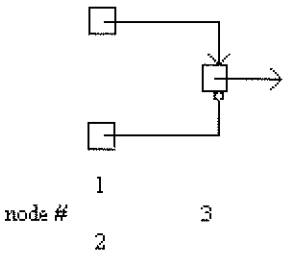


Fig. 1.4. *The net computing the conjoined negation.* Each arrow represents one facilitatory connection. The empty circle represents one inhibitory connection. The threshold T is one. The node 3 fires at the time t if both the node 1 has fired and the node 2 has not fired at the time t minus one tau.

The net computing the third connective, namely $N_3(t)$, introduces us to the problem of the *relative* and *absolute* inhibition. By definition, an inhibitory connection is absolute if it stems out of the node k and impinges onto the node h with a number of ramifications no smaller than the sum of all the facilitatory connections impinging onto the

very same node h. An inhibitory connection is relative if it stems out of the node k and impinges onto the node h with a number of ramifications smaller than the sum of all the facilitatory connections impinging onto the very same node h less the node h threshold. Given a net computing a function g with the relative inhibitory connections, only, there exist a net computing the same g with the absolute inhibitory connections, only?

It is easy to see how to redraw the wiring for a net containing the absolute inhibitory connections so to obtain a new net with relative inhibitory connections, only. To this end, it is sufficient to increase the number of the inhibitory ramifications, stemming out of the node k output channel and impinging onto the node h as one absolute inhibitory connection, until the number of inhibitory ramifications, stemming out of the node k output channel and impinging onto the node h, is greater than the number of facilitatory connections impinging onto the very same node h. To prove the converse, ie, to respond to the above question, we start with a two input net.

Because the net computing the conjoined negation works equally well with both the absolute and the relative negation, if we have a well formed formula written in terms of the above given connectives, then, we can always draw a net either with the absolute or with the relative inhibition. Next, we examine the case of nets with more than two inputs.

For a net computing the disjunction of more than two inputs there is no problem. If some of them have either the absolute or the relative inhibitory connections impinging onto the net output node, the latter fires, or does not fire, in both cases, for the same inputs. Instead, for a net computing the conjunction of more than two inputs the solution is given in the Figure 1. 5., ie, it is sufficient to introduce some delay in between the net inputs and the net output.

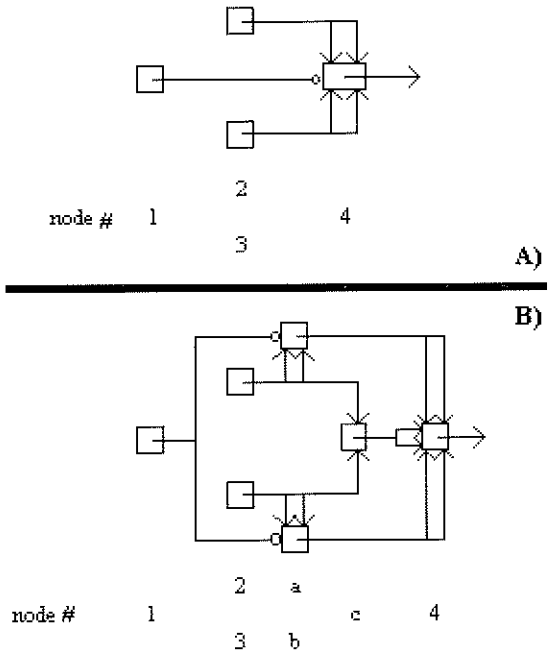


Fig. 1.5. *The relative to absolute inhibition conversion.* Each arrow represents one facilitatory connection. Each empty circle represents one inhibitory connection: a relative one in **A)** and an absolute one in **B)**. The threshold T is two. In **A)** the net computes the expression: $N_4(t) \equiv [\neg N_1(t\text{-}\tau)\ \&\ [N_2(t\text{-}\tau)\ v\ N_3(t\text{-}\tau)]]\ v\ [N_1(t\text{-}\tau)\ \&\ N_2(t\text{-}\tau)\ \&\ N_3(t\text{-}\tau)]$. That is, the node 4 fires at the time t either if the node 1 does not fire at the time t minus tau and at least one of the nodes 2 and 3 fire at the time t minus tau or if all the three nodes 1, 2 and 3 fire at the time t minus tau. In **B)** the net computes the expression: $N_4(t) \equiv [\neg N_1(t\text{-}2\tau)\ \&\ [N_2(t\text{-}2\tau)\ v\ N_3(t\text{-}2\tau)]]\ v\ [N_1(t\text{-}2\tau)\ \&\ N_2(t\text{-}2\tau)\ \&\ N_3(t\text{-}2\tau)]$. That is, the node 4 fires at the time t either if the node 1 does not fire at the time t minus two taus and at least one of the nodes 2 and 3 fire at the time t minus two taus or if all the three nodes 1, 2 and 3 fire at the time t minus two taus. In **B)** the nodes a, b, c are the delays introduced to obtain the net computing the same function as in **A)**.

For the remaining nets, ie, for nets receiving more than two inputs and computing either the disjunctions or the conjunctions of their conjoined negations, again, we apply to the latter the solution given in the last quoted figure. For a more formal proof with a more standard set of connectives, see the next section.

To conclude, the relative and the absolute inhibitory connections are equivalent, in the sense that given a net, computing the function g with a delay d, containing the relative inhibitory connections, only, it is always possible to find another net, computing the very same g with a delay greater than d, containing the absolute inhibitory connections, only.

As we have seen, McCulloch and Pitts basic result, really a fundamental and by no means a trivial one, is to draw the nets computing the well formed formulas of the propositional calculus. However, the formalism they introduce to this end is more like a formal metaphor, than, a practical tool, because the symbols they employ in their formulas bear little relations to the nets represented by the very same formulas. The introduction of more practical tools as the Caianiello's algebraic formalism, we are going to discuss next, consent us to implement a Hebbian rule in a Boolean neural net, as we shall discuss in the chapters 4., 5. and 6..

1.3. The Caianiello's net.

In accordance with the BNN rule iii) and following the Caianiello's formalism, we say the net node h is associated to the *state function* $f_h[t]$ taking one of the two values: either zero or one. Given an N node BNN, we call *state vector* the N components array having as h-th component the state function $f_h[t]$ associated to the node h.

To follow the Caianiello' s symbolism, we call coupling coefficients the efficacies associated to the BNN connections. By contrast, in the probabilistic neural network literature, eg see [4], the coupling coefficient associated to a connection is usually called its weight. Moreover, we call *coupling coefficient matrix* the three dimensional array having as element (h, k, d)-th the *coupling coefficient* $C_{h,k,d}$ associated to the monodirectional channel stemming out of the node k, impinging onto the node h and assuming the state of the node k with the *delay* d. That is, the coupling coefficient $C_{h,k,d}$ is associated to the monodirectional channel transmitting the state of the node k to the node h with the delay d. Because the association between the connections and the coupling coefficients is a one-to-one correspondence, we indifferently consider the former to be associated to the latter as well as the latter to be associated to the former. Eventually, in accordance with the BNN rule iv), the state vector is composed by the three subsets: the *input subvector*; the *hidden subvector* and the *output subvector*.

Here, we consider the BNN connections fixed at manufacturing time. That is, we consider the BNN coupling coefficients as time independent constants. In the chapter 4., we shall introduce BNNs with time dependent connections, ie, BNNs containing connections associated to time dependent coupling coefficients varying with a Hebbian rule, see [5].

Accordingly, for the time being a BNN coupling coefficient $C_{h,k,d}$ is always a constant. It is positive, or *excitatory* or facilitatory, if and only if it is associated to a facilitatory, or excitatory, connection. It is *negative*, or inhibitory, if and only if it is associated to an inhibitory connection. If $C_{h,k,d}$ is zero, then, there is no output channel ramification stemming out of the node k to carry a signal onto the node h.

Let us call c the sum of the positive coupling coefficients associated to the connections impinging onto the node h. To the relative inhibitory connection connecting the node k to the node h, with the delay d, we associate a coupling coefficient $C_{h,k,d}$ in absolute value smaller than c less the node h threshold T_h. To the absolute inhibitory connec-

tion connecting the node k to the node h, with the delay d, we associate a coupling coefficient $C_{h,k,d}$ in absolute value no smaller than c. Unless otherwise explicitly stated we consider absolute inhibitory connections, only.

Let us represent with 1[x] the function equal to zero if x is smaller than zero, equal to one otherwise, ie, if x is no smaller than zero. Given the positive constant D, ie, given the *greatest delay* D, we define the N node BNN *state function* as:

$$f_h[t] = 1[\sum_{1 \le k \le N,\ 1 \le d \le D} C_{h,k,d} * f_k[t - d\tau] - T_h].$$

In accordance with the neural network literature, we call *linear threshold unit*, or LTU, a BNN node. Accordingly, a BNN node, as well as its monodirectional output channel, is in one of two possible states: zero-one, active-inactive, firing-silent. That is, we shall say either one of them is in the state zero or in the state one, or it is active or inactive, or it is firing or not firing, or a node is sending a signal, or the connection stemming out of a node is carrying or transmitting, a signal or it is silent or it is active or it is inactive

Given N, to find the BNN computing a given function g, we must determine its coupling coefficient matrix C. That is, we must satisfy a given inequality system, linear in the unknown coupling coefficients and thresholds, for N given. The system admits a solution only if there is a plane separating the BNN input values, namely the values defining the function g, in two mutually exclusive sets: a set where the BNN state function takes the value zero and a set where the BNN state function takes the value one. In other words, once N has been given, we can determine the matrix C only if the input set is linearly separable. The illustrative examples discussed in the next chapter prove the linear separability of the conjunction with two arguments, of the inclusive disjunction with two arguments and of the logical negation.

1.3.1. *The logical connectives.*

For simplicity sake, we drop the delay d dependency in the next state function expression. Moreover, as the coupling coefficients are measured in terms of the node threshold, to simplify our presentation, we take just one threshold value T for the entire BNN, ie, we drop the T dependency on h. Accordingly, we use the next state function in the simplified form:

$$f_h[t] = 1[\sum_{1 \le k \le N} C_{h,k} * f_k[t - \tau] - T].$$

Later on, see the chapter 5., we shall discuss the *assemblies*, ie, the BNNs specifically wired to implement a Hebbian rule so to obtain the *Adaptable BNN*, or *ABNN*. Because of their complexity, an assembly can have some tens of nodes and we need a formalism compact as well as expressive, so we give two equivalent ways to present a BNN, ie, either a wiring diagram or a coupling coefficient matrix. The former is a more compact and easier to be dealt with formal presentation. However, it is a formalism fully equivalent to the latter, also if less conventional. Later on we shall present the BNNs by giving their wiring diagrams, ie, by giving the algorithm they compute. To obtain the coupling coefficient matrix from a given wiring diagram is a very simple exercise: we set the non-zero matrix elements to the values given for the coupling coefficients in the diagram, simply. The default value is one for the delay, for the threshold and for the positive coupling coefficients. As we are considering absolute inhibitory connections, then, for the negative coupling coefficients the default absolute value is equal to the sum of all the positive coupling coefficients. In the wiring diagrams the facilitatory connections are represented by arrows, the inhibitory connections are represented by empty circles.

As illustrative examples we present the BNNs computing the propositional calculus elementary connectives.

1.3.1.1. *The BNN computing the logical conjunction.*

Let us consider the positive coupling coefficients:

$$0 < C_{\&, k1}, C_{\&, k2}, ..., C_{\&, kn}$$

such that

$$C_{\&, k'1} + C_{\&, k'2} + ...+ C_{\&, k'n-1} < T \leq C_{\&, k1} + C_{\&, k2} + ...+ C_{\&, kn},$$

where

$$C_{\&, k'1} + C_{\&, k'2} + ...+ C_{\&, k'n-1} = \text{Max}\,((C_{\&, k1} + C_{\&, k2} + ...+ C_{\&, kn-1}), (C_{\&, k1} + C_{\&, k2} + ...+ C_{\&, kn-2} + C_{\&, kn}), ..., (C_{\&, k2} + C_{\&, k3} + ...+ C_{\&, kn}))$$

then, the BNN computing the conjunction of the N facilitatory connections stemming out of the nodes 1, 2, ..., N and impinging onto the node &, ie, computing the relation

$$f_{\&}[t] \equiv f_1[t - \tau] \,\&\, f_2[t - \tau] \,\&\, ... \,\&\, f_N[t - \tau]$$

is represented in the Figure 1.6.

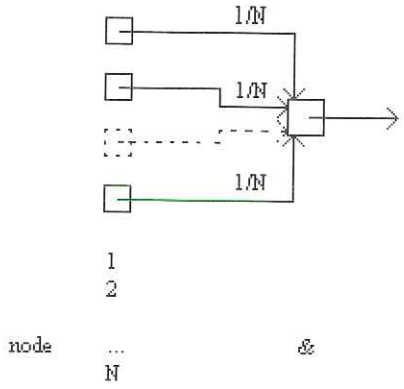


Fig. 1.6. *The BNN computing the N inputs conjunction.* The arrows represent the facilitatory connections. The labels associated to the arrows represent the coupling coefficient values. The threshold is one. The node & is active at the time t if and only if the nodes 1, 2, ..., N have been active at the time t minus one tau.

The proof is straightforward: with the given coupling coefficients, the function

$$f_{\&}[t] = 1[\sum_{1 \leq k \leq N} C_{\&, k}^{*}\, f_k[t - \tau] - T]$$

is one if and only if all its N arguments, ie, all the N functions $f_k[t - \tau]$ are equal to one. It is zero otherwise, ie, whenever no more than N minus one of its arguments are equal to one.

1.3.1.2. *The BNN computing the logical disjunction.*

Let us consider the positive coupling coefficients:

$$0 < T \leq C_{v,k1}, C_{v,k2}, ..., C_{v,kn},$$

then, the BNN computing the disjunction of the N facilitatory connections stemming out of the nodes 1, 2, ..., N and impinging onto the node v, ie, computing the relation

$$f_v[t] \equiv f_1[t - \tau] \vee f_2[t - \tau] \vee ... \vee f_{N-1}[t - \tau]$$

is represented in the Figure 1.7.

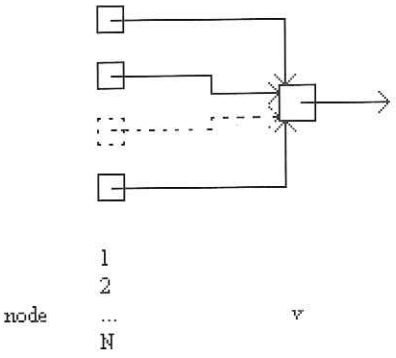


Fig. 1. 7.. *The BNN computing the N inputs disjunction.* Each arrow represents one facilitatory connection greater than the threshold. The node v is active at the time t if at least one of the nodes 1, 2, ..., N has been active at the time t minus one tau.

The proof is straightforward: with the given coupling coefficients, the function

$$f_v[t] = 1[\sum_{1 \leq k \leq N} C_{v,k} * f_k[t - \tau] - T]$$

is one if at least one of its N arguments, ie, one of the N functions $f_k[t]$ is equal to one. It is zero otherwise, ie, whenever none of its N arguments is equal to one.

1.3.1.3. *The BNN computing the existential quantifier in the time domain.*

To give the BNN computing the logical negation of its input value, we need the BNN with the output equal one if in the past at least once its input has assumed the value one. The latter net has the output zero otherwise, ie, if its input has never assumed the value one in the past. We call $\exists[t]$ the output node of the BNN computing the existential quantifier.

Let us consider the coupling coefficient value

$$T \leq C_{\exists,\exists},$$

then, the BNN computing the existential quantifier in the time domain, ie, assuming the output value one if at the least once in the past it has assumed the value one, ie, computing the relation

$$\exists[t] \equiv \exists[t - \tau] \vee \exists[t - 2\tau] \vee ... \vee \exists[0]$$

is represented in the Figure 1. 8..

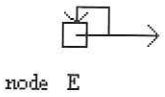


Fig. 1.8. *The BNN computing the existential quantifier in the time domain.* Each arrow represents one facilitatory connection greater than the threshold. The node ∃ is active at the time t if it has been active at the time t minus one tau.

The proof is straightforward: with the given coupling coefficients, the function

$$\exists[t] = 1[C_{\exists,\exists} * \exists[t - \tau] - T]$$

is one if ∃[t - τ] is equal to one. It is zero otherwise, ie, if ∃[t - τ] is equal to zero.

1.3.1.4. *The BNN computing the logical negation.*

Let us consider the coupling coefficients:

$$C_{\neg,k} < 0 < T \leq C_{\neg,\exists} = |C_{\neg,k}|,$$

then, the BNN computing the conjunction of the node ∃ and of the inhibition of the input $f_k[t]$, both impinging onto the node $f_\neg[t]$, ie, computing the relation

$$f_\neg[t] \equiv \exists[t - \tau] \,\&\, \neg f_k[t - \tau]$$

is represented in the Figure 1.9.

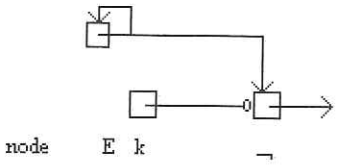


Fig. 1.9. *The BNN computing the logical negation.* Each arrow represents one facilitatory connection greater than the threshold. The empty circle represents the absolute inhibitory connection. If we have activated the node ∃ at the time zero, ie, at the BNN manufacturing time, then, the node ¬ is active at the time t whenever the node k is inactive at the time t minus one tau. And it is inactive at the time t whenever the node k is active at the time t minus one tau.

The proof is straightforward: with the given coupling coefficients, the function

$$f_\neg[t] = 1[C_{\neg,\exists} * \exists[t - \tau] + C_{\neg,k} * f_k[t - \tau] - T]$$

is one if $f_k[t - \tau]$ is equal to zero and ∃[t - τ] is equal to one. It is zero otherwise, ie, if either $f_k[t - \tau]$ is equal to one or ∃[t - τ] is equal to zero. Accordingly, if we initialize the BNN so to have the node ∃ firing at the time zero, ie, if we set

$$\exists[0] = 1,$$

that is,

$$\exists[t] - 1 \text{ for } 0 < t,$$

we end with a net computing the logical negation of the input k.

1.4. The fault tolerant BNN

Let us consider one of the last given BNNs with one, or more, malfunctioning nodes. Under the said conditions the function computed by the BNN is no more the original one: it depends on the number of faulty nodes and on the particular way they fail to work properly. E. g., in the BNN, depicted by the Figure 1.10., computing the exclusive or, let us assume the node four is broken down and cannot become active: accordingly, we are considering a BNN computing the inclusive or.

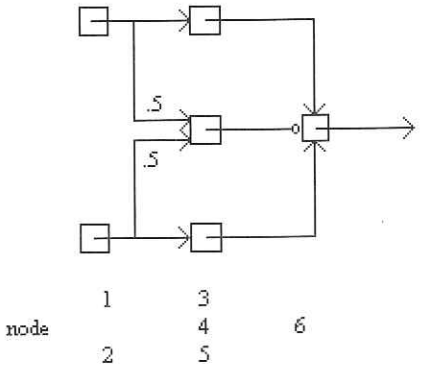


Fig. 1.10. *The BNN computing the exclusive or, ie, the xor.* The arrows represent the facilitatory connections. The labels associated to the arrows represent the coupling coefficient values. The connections without a label have a coupling coefficient greater than the threshold. The empty circle represents the absolute inhibitory connection. The threshold is one. The node 6 is active at the time t if just one of the nodes 1 or 2 has been active at the time t minus one tau. It is inactive at the time t if both the nodes 1 and 2 are either active or inactive at the time t minus one tau.

We are interested to investigate whether or not it is possible to draw an N node net guaranteeing to compute the correct function also if a fraction, ie, either 1/N, or 2/N or ... or R/N of their nodes fail to work properly. Taking advantage of the Caianiello's formalism we are going to discuss a solution to the said problem if there is no lump of connected nodes failing to work as planned, see also [6]. To this end, we must modify the next state function so to take into account the *redundancy* R.

To this end, instead of the single node q, let us consider the set q containing R nodes, the R-redundant set q, such that any node belonging to the set receives the same input connections as the node q. However, the set, as well as its output channel, is active if and only if the majority of its nodes are active, otherwise it is inactive. The coupling coefficient $C_{h,\,q}$ associated to the output channel stemming out of the R-redundant set q and impinging onto the node h, is equal to the coupling coefficient $C_{h,\,q}$ associated to the node q output connection impinging onto the node h. Accordingly, each node belonging to the R-redundant set q receives the very same input connections, associated to the same coupling coefficients, as the node q. That is all the properly working nodes belonging to the same R-redundant set are in the very same state. The Figure 1.11. depicts the fault tolerant net computing the exclusive or also computed by the BNN depicted in the Figure 1.10..

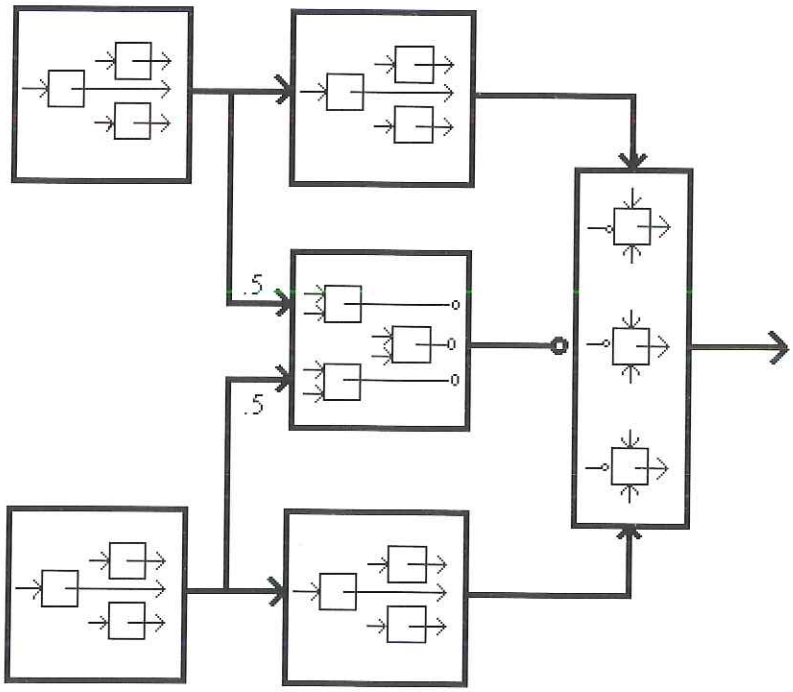


Fig. 1.11. *The fault tolerant net computing the xor.* The arrows represent the facilitatory connections. The empty circles represent the absolute inhibitory connections. The threshold is one. The heavy lines individuate the R-redundant sets: in the picture R is equal three. That is, an R-redundant set becomes active, together with its output connection, if at least two of its nodes are active. Otherwise it is inactive. See the text and the Figure 1. 10. legend.

Let $k_q \in \{R_q\}$ be defined on the set of indexes associated to the nodes belonging to the R-redundant set q. Then, the node h next state function characterizing the fault tolerant net equivalent to a N node BNN is:

$$f_h[t] = 1[\sum_{1 \le q \le N} C_{h,q} * 1[\sum_{kq \in \{Rq\}} f_{kq}[t - d\tau] - (R + 1)/2] - T].$$

Only when the number of active node belonging to the R-redundant set q is no smaller than R/2, the coupling coefficient $C_{h,q}$, associated to the very same R-redundant set q output connection, contributes a non zero addend to the external sum. Otherwise, there is no such contribute to the external sum. That is, if the R-redundant set A receives one, either excitatory or inhibitory, connection with the associated coupling coefficient $C_{A,B}$ from the R-redundant set B, then, each A node receives just one connection with the associated coupling coefficient $C_{A,B}$ from all the B nodes.

Let us call $\Im_q$ the greatest integer no greater than (R - 1)/2. Then, a fault tolerant net with a redundancy R works properly even if no more than R - $\Im_q$ nodes work properly, per R-redundant set. Eventually, given the redundancy R and the N node BNN computing the function g, we can always find the N * R node fault tolerant net computing the very same function g, correctly, even if there are up to $\Im_q$ malfunctioning nodes per R-redundant set.

Because of the last proposition and to avoid a cumbersome notation, we consider only nets with the redundancy R equal to one.

Chapter 2

THE KNOWLEDGE REPRESENTATION IN THE BNN.

2.1. Introduction

One of the problems about the knowledge representation is relative to the relations between something and its representation when more persons, eg two discussants, communicate. Here we are going to discuss the knowledge representation within a system like a Turing Machine black box, or TM, or a General Purpose Architecture, or GPA, or a Central Nervous System of a vertebrate, or CNS, or a BNN. That is, within a system interacting through its I/O channels with an unknown and unpredictably evolving environment. In general such a system control its interactions with the environment by transforming an input state subvector sequence in an output state subvector sequence. In other words, we are going to discuss the knowledge representation in a system embodying the correspondence between its input sequences and its output sequences. To this end, let us ask how a system control flow is affected by the present input.

In a TM the quintuple to be executed next depends jointly on the system state and on the tape content, ie, in a TM the present input affects both the present and the next action, at the least. In a GPA the control flow depends on its central processing unit input, ie, on the instruction register present content. That is, in a von Neumann architecture the central processing unit input is the content of the memory location having as address the program counter content. As a rule, at each clock the program counter content is increased by the fixed amount a. That is, usually the control flow goes through consecutive memory locations, unless the instruction register present content is a taken jump instruction. In the latter case, the program counter content is no more computed by adding a to its last value: it is obtained by means of a different procedure. Accordingly, it is so much easier to implement macros, ie, routines or functions in a GPA. To do that, just two jump instructions are sufficient: the first, namely the macro calling sequence, to the memory location containing the macro first instruction. The second a jump, namely the macro output or return instruction, usually to the memory location next to the one containing the macro calling sequence, ie, to the memory location containing the instruction which would have been executed next if the calling sequence was a non taken jump. Obviously, also in a TM it is possible to implement macros, however it is a more laborious affair. To better appreciate the relation between the control flow and the knowledge representation, let us consider the Pascal language sequence Sqrt(Num), ie, the calling sequence of the function restituting the square root of the argument Num. To find the square root of Num either we could have inserted a pointer, whenever we need the square root, to the Num entry of a double entry table stored in the GAP main memory, the table second entry being the square root of the first entry, namely of Num. Or we could have inserted the code to

compute the square root, whenever we need it. Or we could have inserted, whenever we need it, the mnemonic Sqrt(Num), ie, a pointer to the address corresponding to the first instruction of the code computing the square root. Unless we need the square root of no more than two Num values, however see also the discussion in [11], the last option is the most efficient. In other words, a pointer to a procedure can be an efficient way to store the data. Now let us see what happens in the neural networks.

In general, in the neural networks there are three kinds of nodes: the input, the hidden and the output nodes. That is, in the state vector we distinguish the three, not necessarily disjoint, subvectors: the input, the hidden and the output subvector. Somehow the sensors states are projected, as in a bitmap, in the input node states. Then, the hidden nodes concurrently process the inputs and the result is read as the output node state sequences.

In the non-Boolean neural nets either the hidden nodes processing is statistical in character. Or the input states somehow induces a more or less repetitive behavior in the hidden node state sequences. Once reached a steady state, the hidden node states oscillation is the net output. In any case, the transformations performed by the net on the input sequences depend on the number of nodes lying between the input and the output nodes. From this point of view, a neural net recalls a TM without the possibility to implement macros.

For a discussion on the knowledge representation in a non-Boolean neural network see [3 part I sect. 3 pg. 77-109, 7, 8]. In effect, the net environment representation is inspired to the net designer vision of the world. Actually, if the net designer is interested primarily in pattern classification or recognition, then, a declarative approach is more than adequate, as proved by the many works on the subject. However, as we are interested in studying the autonomously self-trainable systems, we need a more adequate knowledge representation. In this respect, the AI community seems to realize the possible existence of less anthropocentric approaches, besides the declarative one. In other words, for some authors, e. g. see [9, 10], an agent internal representation of its environment should take into account the agent own sensory and motor systems. However, the knowledge representation reflects, always, the designer goal in realizing an agent interacting with its own environment. E. g., in a general purpose machine main memory the alphanumeric characters can be represented either symbolically as ASCII codes or graphically as bitmaps.

To fully appreciate the relevance of the knowledge representation choice, it is sufficient to figure how the standard programs for symbolic manipulations, e. g. a searching or sorting routine, have to be modified, if the data are graphically given as bitmaps. Or the irrelevance of the character recognition problem, if the data are given in their symbolic form as ASCII codes.

By contrast, because in the BNN case the hidden node processing is deterministic, the input bitmap can play the macro calling sequence role and we can have feedback connections, eg from the output nodes, playing the role of the macro return instruction. Similarly, in the CNS the proprioception is implemented as nervous connections feeding back the system own state. In other words in a BNN, as well as in a CNS, there is the technical possibility to implement macros. Accordingly, as we can implement in a BNN a suitable knowledge representation, we are going to discuss how the environment events can be represented in a BNN as pointers to the output sequences, ie, to the output routines. Or, in other words, how the knowledge representation in a system can be based on the system output responses to the very same environment events to represent.

2.2. The BNN as a control structure

To write a numeral, namely a sequence of digits, we follow some conventions: e. g., we employ the positional notation to save both space and writing efforts. By contrast, let us consider the notation a^b, with a and b finite sequences of digits. The symbol a repre-

sents a number and the symbol b represents the number of operations we have to perform to know the number represented by a^b. In other words, whereas a is the datum upon which the operation, actually a multiplication, must be executed, b indicates the number of times the operation has to be executed. That is, a^b is like a shorthand notation for a program: to represent a^b in the positional notation we must execute the program. Whereas both a and a^b represent the numbers, eg of apples or diamonds, b represents the number of multiplications, always. Accordingly, in the a^b notation a and b meanings are different and implicitly associated to their relative positions.

Next, let us give to a poor typist the task to reproduce a typescript by banging onto the typewriter keys with a hammer, one key at the time. Once the text has been copied we have two typewritten texts. That is, we started with one sequence of symbols, namely the original typescript, and ended up with two, namely the two identical typescripts. Obviously, to copy the symbols the typist had each one of them typed, by suitably arranging the hammer position. That is, under the typist control a flux of knowledge went through the hammer, from the original to the typescript copy. So we can ask what is the hammer role in the knowledge representation context.

If we label the hammer with its present position relative to one, well specified, keyboard, then, the labeled hammer is just a symbol. Perhaps a cumbersome way to depict one. Certainly a way to represent the symbols present in a given keyboard by labeling, simply, the hammer with all its possible positions, relative to the keyboard. Eventually, if we strip the hammer of its labels, then, we are left with just a pointer. Labeled pointer are a quite common feature of modern life. Once upon a time they used to say that all the roads lead to Rome, so they do not need the pointers. But that was then. Nowadays, as we move outside our friendly neighborhood, to avoid ending up in the wrong place, we need labeled pointers to safely arrive at our destination. However, as before said, the labeled pointers are a cumbersome way to depict symbols. Instead, in the a^b notation there is an implicit meaning associated to both the symbol a and the symbol b. Accordingly, let us discuss a symbolic knowledge representation based on pointers with implicit labels.

If we want a BNN to be a store based on pointers with implicit labels, it is sufficient to consider the BNN as the control system for a robot interacting with its own environment. In other words, it is sufficient, first, to have each of its output nodes, or equivalently each of its output state sequences, controlling just one, always the same, robot effector as a function of the robot environment events. Second, to choose the BNN input sequences, accordingly. The emphasis is on the order we have to follow to set the BNN output and input states correspondence.

Accordingly, we start by characterizing the knowledge storing BNN, then, as illustrative example, we give the net controlling a writing robot. Actually, we present a BNN controlling the execution of the addition of, binary, operands with an unlimited number of digits. To conclude the discussion on the knowledge representation, in the next chapter we present a traditional data structure storing the data as the said BNN does, see also [6 cap. 5.1.1].

Our BNN output states are supposed to control a writing robot, ie, to control the robot to execute its primitive movements. That is, to control the robot to execute the six unitary actions: move to the left, move to the right, move up, move down, write, erase. The BNN input states represent two sets of patterns. One set contains the inputs needed to control the robot primitive commands: left, right, up, down, write, erase or `l`, `r`, `u`, `d`, `w`, `e`. The other set contains the five times twenty-five digit binary numerals, ie, the sequences composed by one hundred and twenty five ones and zeros. They represent the state of the five components *environment vector*. By definition each component is composed by twenty-five unitary squares, ie, by twenty-five squares having the side length equal to one of the robot primitive movements.

In the Figure 2.1., we depict the block diagram of a BNN controlling the writing robot in the execution of the addition of binary numbers of unbounded length, see the chapter 2.2.1. It is composed by the *primitive* subnet and the *adder* subnet. For simplicity

sake, both we have presented the adder as the only higher level net and we deal with binary numbers, only. We leave to the interested reader to modify the block diagram with more higher level nets, e. g. by adding a subtracter and a multiplier net. And to extend the arithmetic operations to numbers represented by numerals in the base ten.

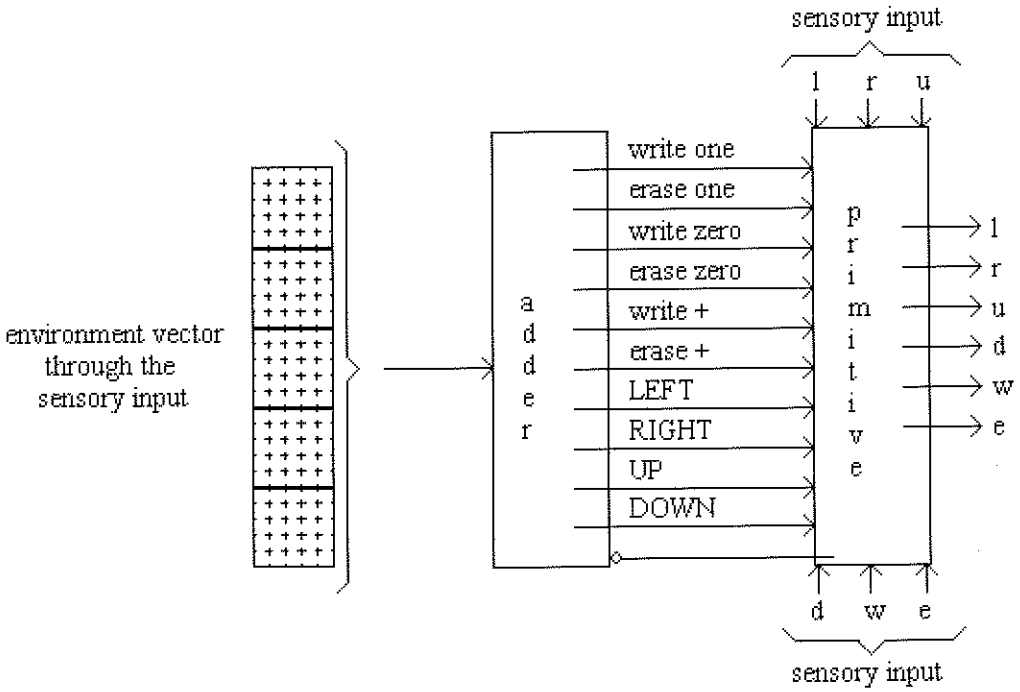


Fig. 2.1. *The block diagram of the BNN computing the sum of operands with an unlimited number of digits.* On the left the environment vector is the input of the adder BNN, see the Figures from 2.10. to 2.16. The latter outputs are the primitive BNN inputs, see the Figures 2.2. to 2.8. On the extreme right the `l`, `r`, `u`, `d`, `w`, and `e` outputs represent the robot effector controls. Both the adder and the primitive nets accept signals from the robot sensory inputs. The former as the sequence of one hundred and twenty five zeros and ones representing the environment vector content, see the chapter 2.2.1. The latter as the primitive commands `l`, `r`, `u`, `d`, `w`, and `e`. The primitive block sends an inhibitory feedback connection to the adder block, to avoid jumbling up the BNN outputs.

2.2.1. *The primitive BNN.*

Let us call primitive net the BNN controlling the writing robot motors, in a one-to-one correspondence. That is, each time one output node of the primitive net becomes active, one of the robot effectors, always the same, executes one unitary movement, after a fixed delay. Without danger of confusion, we call both the nodes and the primitive movements they control left, right, up, down, write, erase or `l`, `r`, `u`, `d`, `w`, `e`, for short. The `l`, `r`, `u`, `d`, `w` and `e` nodes activity are the primitive commands issued by the primitive BNN to control the robot primitive movements. The first four control the primitive movements parallel to the writing plane, the last control the robot in its writing and erasing capabilities. That is, the robot moves one step at the time, depending on the command it has to execute. Similarly, it writes, or erases, only when the relative command is issued by the primitive net output node. In other words, if we want the robot to write two consecutive times in the leftward direction, then, the primitive net output nodes must become active in the following sequence: `lwlw`. Instead if we want the robot to move in the leftward direction leaving unaltered the environment state, then, the primitive net output nodes must become active in the following output sequence: `ll`. To help the higher level nets, in the present case the adder net, to control the robot effectors, the primitive net sends to them an inhibitory feedback signal whenever one of its output nodes is active.

The primitive net receives six input connections from the robot sensory inputs, one for each of its six output nodes. That is, the primitive net output nodes can be independently activated by special purpose sensory signals. Moreover, the primitive net receives input connections from the higher level net to carry the latter commands. E. g. from the adder it receives the ten input connections "write one", "write zero", "write +", "erase one", "erase zero", "erase +", LEFT, or L, RIGHT, or R, UP, or U, DOWN, or D, we shall describe in the chapter 2.2.1.1. as the adder net outputs.

To avoid drawing too many lines in one figure, we have represented in the Figures 2.2. to 2.8. the primitive BNN controlling the robot to execute the moving commands L, R, U, D, and to write, to erase, the binary digits and the sum sign, in response to the respective command trough either the sensory inputs or the inputs from the adder. As the time required to execute a command sequence depends on the particular digit or sign being outputted, we have inserted in the primitive BNN a node inhibiting the input activity until the task presently in execution is terminated. The very same node activity is fed back also to the adder, to regulate its output signals flow.

2.2.1.1. *The BNN controlling the moving commands.*

To move from an environment vector to the next we introduce the sequences of five identical primitive moving commands. That is, the four adder output commands `LEFT`, `RIGHT`, `UP`, `DOWN` or for short `L`, `R`, `U`, `D`. They stand for the primitive command sequences `lllll`, `rrrrr`, `uuuuu`, `ddddd`, respectively. The first two move the robot over the next environment vector at the left, right, of the present one. The second two move the robot on the next, up or down, component. By convention, the next of the topmost, bottommost, component of a given environment vector is the bottommost, topmost, component of the next environment vector in the upward, downward, direction. As it easy to verify, the execution of the `L`, `R`, `U`, `D` commands leaves the robot over the unitary square homologous of the one it started on.

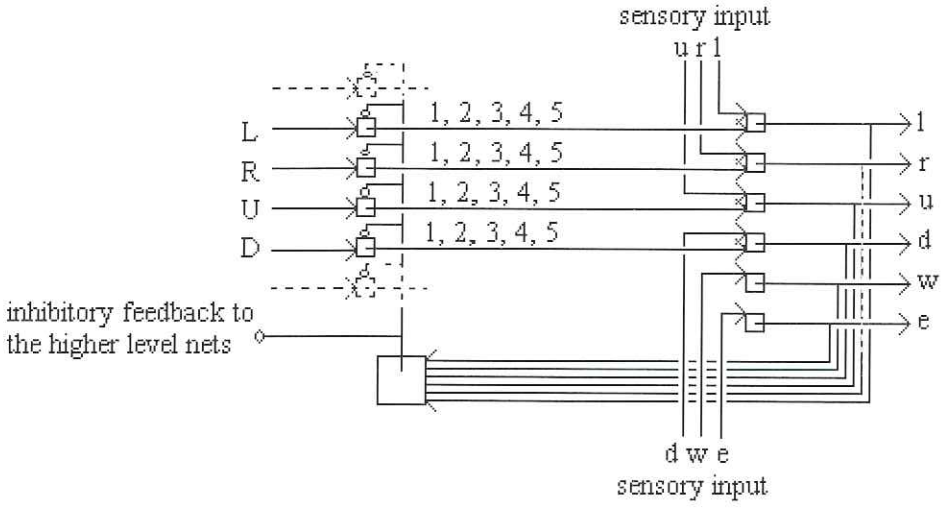


Fig. 2.2. *The primitive BNN: the moving commands.* Each arrow represents one facilitatory connection greater than the threshold. The empty circles represent the absolute inhibitory connections. The label near the connections represent the delays. That is, instead of inserting as many nodes to have the right delays, we have given the connection delays.

In the Figure 2.2. we depict the Boolean net controlling the moving commands. The input connections on the left represent the adder outputs, see the Figure 2.1.. The sensory

inputs stem from the robot sensors. The output connection on the left represents the primitive inhibitory feedback to the adder and to the other higher level nets. The output connections on the right represent the BNN output toward the robot effectors. E. g., to move the robot on the next right environment vector, the command sequence is rrrrr, ie, the input R must be activated.

2.2.1.2. *The BNN controlling the write-one command.*

To control the robot in writing the digit "one", starting from the central square belonging to the presently observed environment vector component, the primitive BNN output sequence is: uwdwdwu. For simplicity sake, we have given the corresponding BNN with the relative delays, actually, it could have been given with as many intermediate nodes.

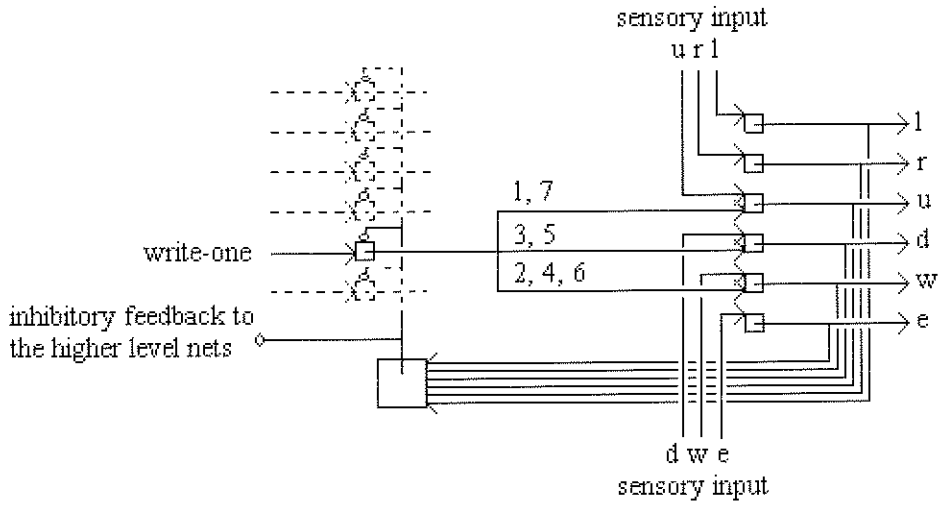


Fig. 2.3. *The primitive BNN: the write-one command.* Each arrow represents one facilitatory connection greater than the threshold. The empty circles represent the absolute inhibitory connections. The label near the connections represent the delays. That is, instead of inserting as many nodes to have the right delays, we have given the connection delays.

In the Figure 2.3. we depict the Boolean net controlling the write-one command. The input connections on the left represent the adder outputs, see the Figure 2.1. The sensory inputs stem from the robot sensors. The output connection on the left represents the primitive inhibitory feedback to the adder and to the other higher level nets. The output connections on the right represent the BNN output toward the robot effectors. E. g., to control the robot in writing the digit "one", the primitive BNN output sequence is: uwdwdwu, ie, the sequence obtained by following the delays of the output connections stemming out of the node having the sequence "write-one" as input.

2.2.1.3. *The BNN controlling the erase-one command.*

To control the robot in erasing the digit "one", starting from the central square belonging to the presently observed environment vector component, the primitive BNN output sequence is: uededeu. For simplicity sake, we have given the corresponding BNN

with the relative delays, actually, it could have been given with as many intermediate nodes.

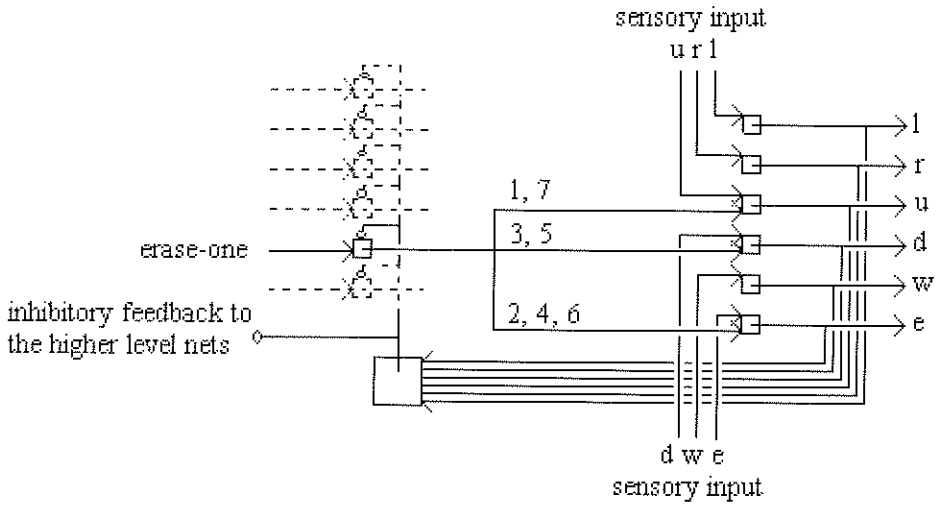


Fig. 2.4. *The primitive BNN: the erase-one command.* Each arrow represents one facilitatory connection greater than the threshold. The empty circles represent the absolute inhibitory connections. The label near the connections represent the delays. That is, instead of inserting as many nodes to have the right delays, we have given the connection delays.

In the Figure 2.4. we depict the Boolean net controlling the erase-one command. The input connections on the left represent the adder outputs, see the Figure 2.1. The sensory inputs stem from the robot sensors. The output connection on the left represents the primitive inhibitory feedback to the adder and to the other higher level nets. The output connections on the right represent the BNN output toward the robot effectors. E. g., to control the robot in erasing the digit "one", the primitive BNN output sequence is: `uededeu`, ie, the sequence obtained by following the delays of the output connections stemming out of the node having the sequence "erase-one" as input.

2.2.1.4. *The BNN controlling the write-zero command.*

To control the robot in writing the digit "zero", starting from the central square belonging to the presently observed environment vector component, the primitive BNN output sequence is: `uwrwdwdwlwlwuwuwrd`. For simplicity sake, we have given the corresponding BNN with the relative delays, actually, it could have been given with as many intermediate nodes.

In the Figure 2.5. we depict the Boolean net controlling the write-zero command. The input connections on the left represent the adder outputs, see the Figure 2. 1.. The sensory inputs stem from the robot sensors. The output connection on the left represents the primitive inhibitory feedback to the adder and to the other higher level nets. The output connections on the right represent the BNN output toward the robot effectors. E. g., to control the robot in writing the digit "zero", the primitive BNN output sequence is: `uwrwdwdwlwlwuwuwrd`, ie, the sequence obtained by following the delays of the output connections stemming out of the node having the sequence "write-zero" as input.

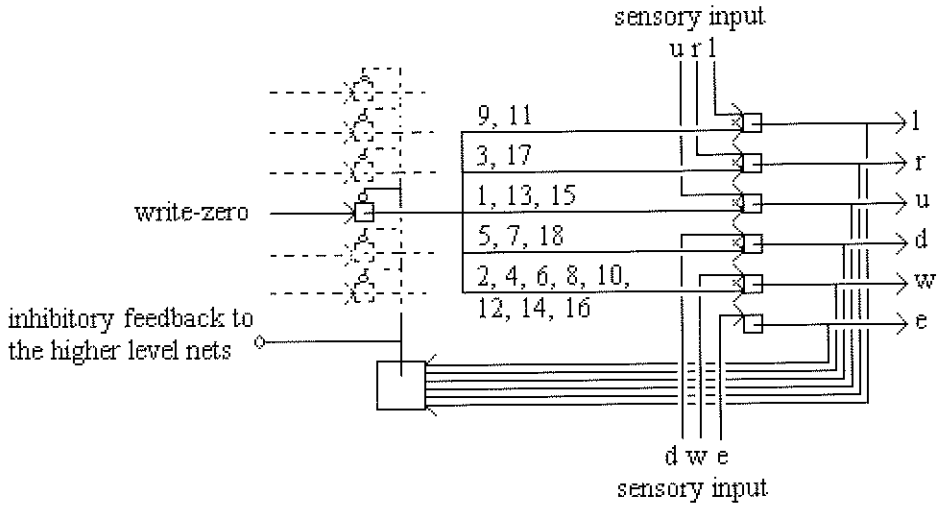


Fig. 2.5. *The primitive BNN: the write-zero command.* Each arrow represents one facilitatory connection greater than the threshold. The empty circles represent the absolute inhibitory connections. The label near the connections represent the delays. That is, instead of inserting as many nodes to have the right delays, we have given the connection delays.

2.2.1.5. *The BNN controlling the erase-zero command.*

To control the robot in erasing the digit "zero", starting from the central square belonging to the presently observed environment vector component, the primitive BNN output sequence is: `ueredededeleleueuerd`. For simplicity sake, we have given the corresponding BNN with the relative delays, actually, it could have been given with as many intermediate nodes.

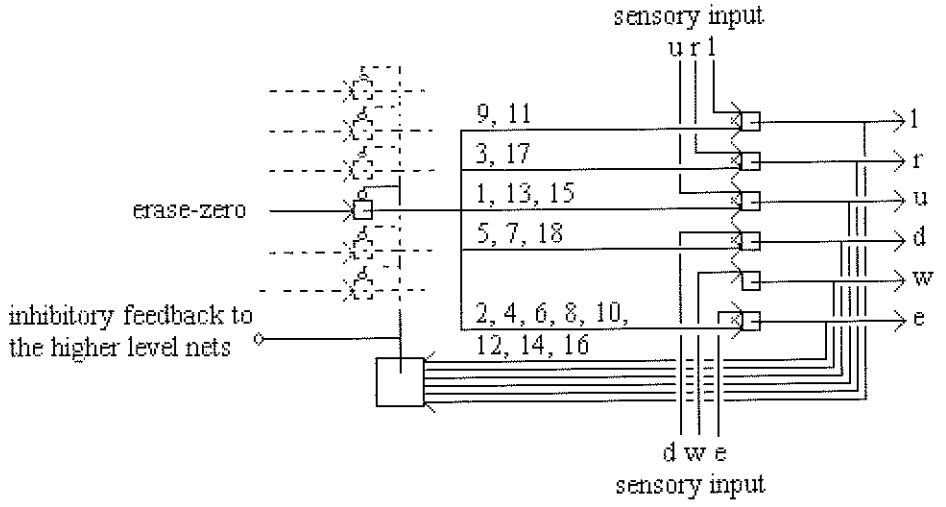


Fig. 2.6. *The primitive BNN: the erase-zero command.* Each arrow represents one facilitatory connection greater than the threshold. The empty circles represent the absolute inhibitory connections. The label near the connections represent the delays. That is, instead of inserting as many nodes to have the right delays, we have given the connection delays.

In the Figure 2.6. we depict the Boolean net controlling the erase-zero command. The input connections on the left represent the adder outputs, see the Figure 2.1. The sen-

sory inputs stem from the robot sensors. The output connection on the left represents the primitive inhibitory feedback to the adder and to the other higher level nets. The output connections on the right represent the BNN output toward the robot effectors. E. g., to control the robot in erasing the digit "zero", the primitive BNN output sequence is: `uerededeleleueuerd`, ie, the sequence obtained by following the delays of the output connections stemming out of the node having the sequence "erase-zero" as input.

2.2.1.6. *The BNN controlling the write-+ command.*

To control the robot in writing the sign "+", starting from the central square belonging to the presently observed environment vector component, the primitive BNN output sequence is: `wlwrrwldwuuwd`. For simplicity sake, we have given the corresponding BNN with the relative delays, actually, it could have been given with as many intermediate nodes.

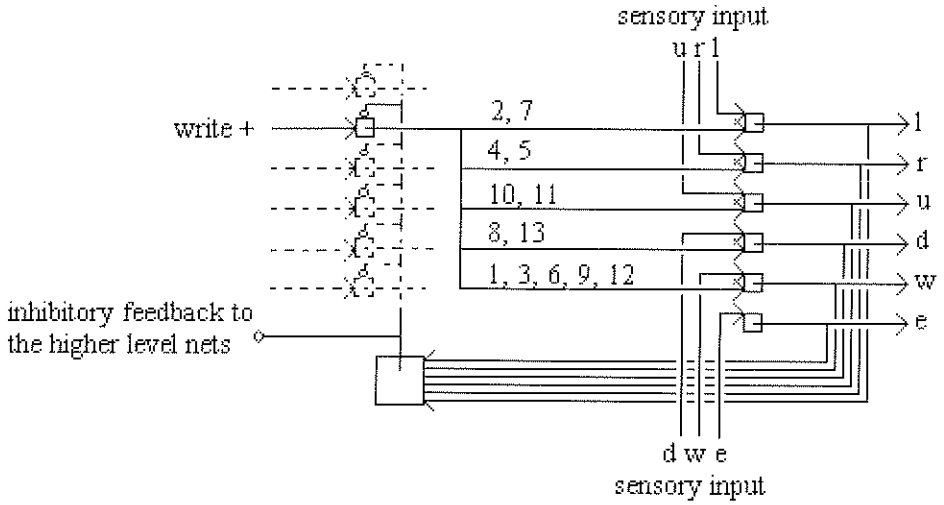


Fig. 2.7. *The primitive BNN: the write-+ command.* Each arrow represents one facilitatory connection greater than the threshold. The empty circles represent the absolute inhibitory connections. The label near the connections represent the delays. That is, instead of inserting as many nodes to have the right delays, we have given the connection delays.

In the Figure 2.7. we depict the Boolean net controlling the write-+ command. The input connections on the left represent the adder outputs, see the Figure 2.1. The sensory inputs stem from the robot sensors. The output connection on the left represents the primitive inhibitory feedback to the adder and to the other higher level nets. The output connections on the right represent the BNN output toward the robot effectors. E. g., to control the robot in writing the plus sign, the primitive BNN output sequence is: `wlwrrwldwuuwd`, ie, the sequence obtained by following the delays of the output connections stemming out of the node having the sequence "write +" as input.

2.2.1.7. *The BNN controlling the erase-+ command.*

To control the robot in erasing the sign "+", starting from the central square belonging to the presently observed environment vector component, the primitive BNN output sequence is: `elerreldeuued`. For simplicity sake, we have given the corresponding

BNN with the relative delays, actually, it could have been given with as many intermediate nodes.

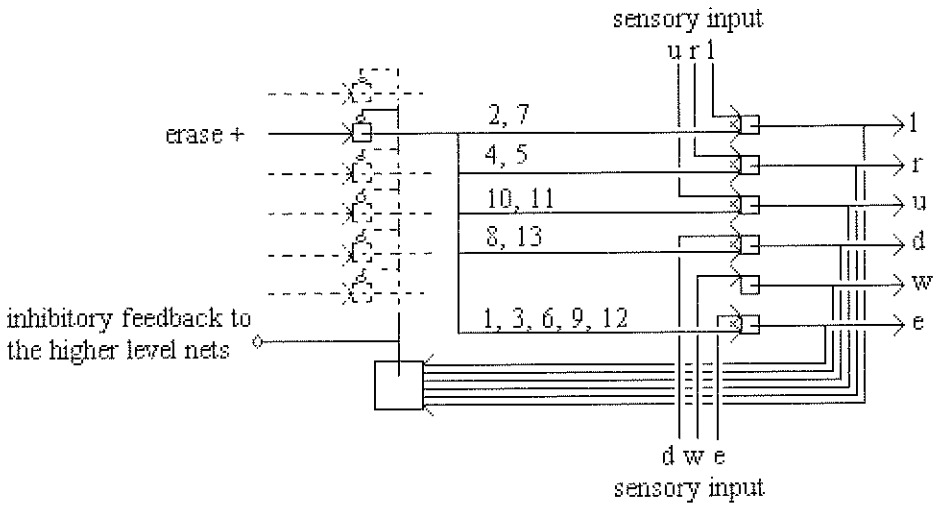


Fig. 2.8. *The primitive BNN: the erase-+ command.* Each arrow represents one facilitatory connection greater than the threshold. The empty circles represent the absolute inhibitory connections. The label near the connections represent the delays. That is, instead of inserting as many nodes to have the right delays, we have given the connection delays.

In the Figure 2.8. we depict the Boolean net controlling the erase-+ command. The input connections on the left represent the adder outputs, see the Figure 2.1. The sensory inputs stem from the robot sensors. The output connection on the left represents the primitive inhibitory feedback to the adder and to the other higher level nets. The output connections on the right represent the BNN output toward the robot effectors. E. g., to control the robot in erasing the plus sign, the primitive BNN output sequence is: `elerreldeuued`, ie, the sequence obtained by following the delays of the output connections stemming out of the node having the sequence "erase +" as input.

2.2.2. *The adder BNN.*

The Adder sensory input states represent the five times twenty-five digit binary numerals, ie, the sequences composed by one hundred and twenty five ones and zeros. They represent the state of the five components environment vector. Each component is composed by twenty-five unitary squares, ie, by twenty-five squares having the side length equal to one of the robot primitive movements.

To normalize the control of the robot movements, so to facilitate their concatenation, all the robot journeys start and terminate on the central unitary square of the middle component of the environment vector. Accordingly, all the robot journeys relative to just one environment vector must be composed by as many up, or left, as down, or right, primitive movements. That is, let us associate a plus one to both the `u` and `l` commands and a minus one to both the `d` and `r` commands. Then, it must be zero the sum of all the values associated to the sequence of commands controlling any of the robot possible journeys on the same environment vector.

The environment vector unitary squares can be either blank or written on. If the robot writes on the previously written unitary square S, then, S remains written. If the robot erases the blank, either the initially blank or the previously erased, unitary square S, then, S remains blank. If the robot moves on the unitary square S without executing neither the `w` nor the `e` command, then, the S state remains unchanged. That is, if S was

a written, blank, unitary square before the robot started to move, then, S remains written, blank, once the robot as terminated the moving command execution.

To write and to erase the arithmetic operation signs and the digits one and zero, we have introduced the six adder output commands write-one, erase-one, write-zero, erase-zero, write-+, erase-+, see the Figures 2.3., 2.4., 2.5., 2.6., 2.7., 2.8., respectively. They stand for the primitive command sequences `wlwrrwldwuuwd`, `elerreldeuued`, `uwdwdwu`, `uededeu`, `uwrwdwdwlwlwuwuwrd`, `ueredededeleleueuerd`, `respectively`.

Eventually, the sequence of one hundred and twenty-five ones and zeros, entering the visual input channel, represent the sequence of written, the ones, and blank, the zeros, unitary squares. An input sequence represents the unitary squares in the left-right and top-down direction, starting from the uppermost and leftmost unitary square belonging to the uppermost component of the environment vector.

TABLE 2.1.

(+, b, 0, v, b)⇒D, D, write-zero, U, U, U, U, erase-+, L, write-+, D, D
(+, b, v, 0, b)⇒D, D, write-zero, U, U, U, U, erase-+, L, write-+, D, D
(+, b, v, 1, b)⇒D, D, write-one, U, U, U, U, erase-+, L, write-+, D, D
(+, b, 1, v, b)⇒D, D, write-one, U, U, U, U, erase-+, L, write-+, D, D
(+, 1, v, v', b)⇒D, D, write-one, U, U, U, erase-one, U, erase-+, L, write-+, D, D
(+, b, 1, 1, b)⇒D, D, write-zero, U, U, U, U, erase-+, L, write-+, D, write-one, D
(+, 1, v, 1, b)⇒D, D, write-zero, U, U, U, erase-one, U, erase-+, L, write-+, D, write-one, D
(+, 1, 1, v, b)⇒D, D, write-zero, U, U, U, erase-one, U, erase-+, L, write-+, D, write-one, D
(+, 1, 1, 1, b)⇒D, D, write-one, U, U, U, erase-one, U, erase-+, L, write-+, D, write-one, D
(+, b, b, b, b)⇒U, U, erase-+, L, D, D

Table 2.1. *The adder input-output correspondence.* Instead of the real BNN input sequences, ie, instead of one hundred and twenty-five binary digit sequences, we give the ten environment vectors represented by them. To each of them we have associated the adder generated output sequences when they are its inputs. The symbol b represents a blank space and the symbols v and v' represent, independently one from the other, either a blank space or the digit zero.

TABLE 2.2.

... (b, b, x, y, b)	**(+, b, 0, v, b)** ... ⇒ ...	**(+, b, x, y, b)** (b, b, 0, v, 0) ...
... (b, b, x, y, b)	**(+, b, v, 0, b)** ... ⇒ ...	**(+, b, x, y, b)** (b, b, v, 0, 0) ...
... (b, b, x, y, b)	**(+, b, 1, v, b)** ... ⇒ ...	**(+, b, x, y, b)** (b, b, 1, v, 1) ...
... (b, b, x, y, b)	**(+, b, v, 1, b)** ... ⇒ ...	**(+, b, x, y, b)** (b, b, v, 1, 1) ...
... (b, b, x, y, b)	**(+, 1, v, v', b)** ... ⇒ ...	**(+, b, x, y, b)** (b, b, v, v', 1) ...
... (b, b, x, y, b)	**(+, b, 1, 1, b)** ... ⇒ ...	**(+, 1, x, y, b)** (b, b, 1, 1, 0) ...
... (b, b, x, y, b)	**(+, 1, v, 1, b)** ... ⇒ ...	**(+, 1, x, y, b)** (b, b, v, 1, 0) ...
... (b, b, x, y, b)	**(+, 1, 1, v, b)** ... ⇒ ...	**(+, 1, x, y, b)** (b, b, 1, v, 0) ...
... (b, b, x, y, b)	**(+, 1, 1, 1, b)** ... ⇒ ...	**(+, 1, x, y, b)** (b, b, 1, 1, 1) ...
... (x', y', x, y, z)	**(+, b, b, b, b)** ... ⇒ ...	**(x', y', x, y, z)** (b, b, b, b, b) ...

Table 2.2.. *The environment vectors transformation.* At the left, right, hand side we have represented the environment vectors as seen by the robot sensory inputs before, after, the execution of the adder controlled transformation. The presently seen environment vector is in **bold** font. The symbol b represents a blank space, the symbols v and v' represent, independently one from the other, either a blank space or the digit zero, the symbol x' represents either the plus sign + or a blank space, the symbol y' represents either the digit one or a blank space and the symbols x, y, z represent, one independently of the others, either the blank space or the digit zero or the digit one.

To simplify our BNN, we assume the addend being written one below the other, namely with the respective digits being written in the third and fourth component, from the top, of consecutive environment vectors, and a plus sign on top of the rightmost digit, namely in the uppermost and rightmost component of the environment vector sequence. Accordingly, to simplify the presentation, we call *sign component* the environment vector topmost component, *carry component* the next to the topmost component, *first addend component* or *first operand component* the middle component, *second addend component* or *second operand component* the fourth from the top component, *result component* the bottommost component.

As the primitive BNN output sequences are of different lengths, we must retard some of the adder output, ie, some of the primitive input signals, to this end we present in the Figure 2.9. the delayed BNN, for short the dB. As one of the primitive outputs becomes active, it sends an inhibitory feed back signal to the adder, in the Figure 2.9. represented by the c input connection. The adder signal to be retarded, namely the one travelling on the dB a input connection, is dynamically stored in the existential node, namely in the Figure 2.9. leftmost node. As soon as there are no more inhibiting signals coming through the c connection from the primitive, namely once the primitive has terminated its task, the dB d output connection becomes active. In turn, the very same d activity inhibits both the existential and the d output node. So the delayed BNN is automatically reset as it terminates its task.

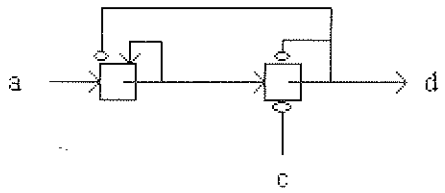


Fig. 2.9.. *The delayed BNN, or dB subnet.* In the adder, represented in the Figures 2. 8. to 2.16., we need to retard some signals to give time to the primitive BNN to execute its elaborations.

To avoid drawing too many lines in one figure, we have represented in the Figures 2.10., 2.11., 2.12., 2.13., 2.14., 2.15. and 2.16. the adder BNN sending to the primitive the signals needed to control the robot to add two binary numerals composed by an unbounded number of digits. That is, in each figure we have presented the adder BNN nodes activated by the following ten environment vectors: (+, b, 0, v, b), (+, b, v, 0, b), (+, b, v, 1, b), (+, b, 1, v, b), (+, 1, v, v, b), (+, b, 1, 1, b), (+, 1, v, 1, b), (+, 1, 1, v, b), (+, 1, 1, 1, b), (+, b, b, b, b), where b is a blank and v is either a blank or a zero. For each of the ten environment vectors, we have given the adder output sequence in the Table 2.1. and in the Table 2.2. the ten environment vector pairs resulting after the robot has completed its task.

In the Figure 2.10. we depict the adder subnet responding to the (+, b, v, v', b) environment vector, see the Table 2.2.. The sequence of commands the adder transmits to the primitive to change the environment vector content, from the one given at the left hand side into the one given at the right hand side, is: D, D, write-zero, U, U, U, U, erase-+, L, write-+, D, D. At the adder left hand side, the dB nets, see the Figure 2.9., are activated sequentially with a four τ delay, ie, each becomes active four τ after the previous one has terminated its activity. The first becomes active four τs after the adder input becomes active. The adder remaining delays are one τ each. The arrows represent the excitatory connections. The empty circles represent the inhibitory connections. The coupling coefficients are one threshold unit, unless otherwise specified. That is, the existential node at the very center of the figure has an output connection with an associated coupling coefficient of one half of the threshold. We have put the label c = .5 to the connection between the eighth dB net, following their activation order, and the erase-+ adder output node because the plus sign must be erased if the present environment vector is not the one containing the rightmost digits of the two addends, see also the Figure 2.16.

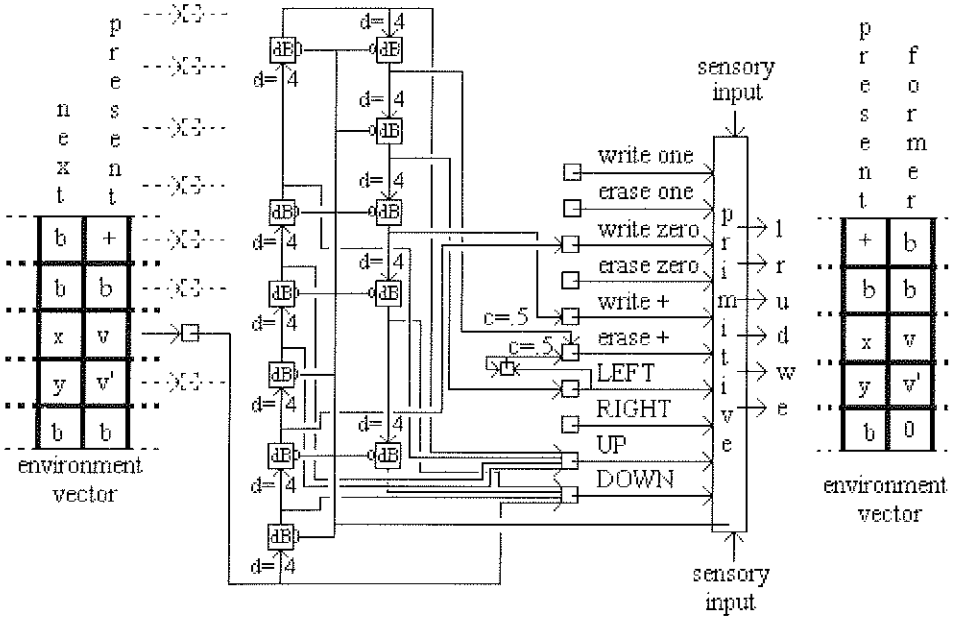


Fig. 2.10. *The adder subnet responding to the (+, b, v, v', b) environment vector.* In the environment vectors, the symbols v and v' represent either the v' for either a blank or the digit zero and the v for the digit zero or the v for either a blank or the digit zero and the v' for the digit zero, the symbol b represents a blank space and the symbol x, y, represents either a blank space or the digit zero or the digit one. At the left hand side we have depicted the present and the next environment vectors, as seen by the robot sensory inputs, see the Table 2.2. At the right hand side we have depicted the former environment vector, namely the left hand side present one, and the present environment vector, namely the left hand side next one, as seen by the robot sensory inputs once the BNN has terminated its task.

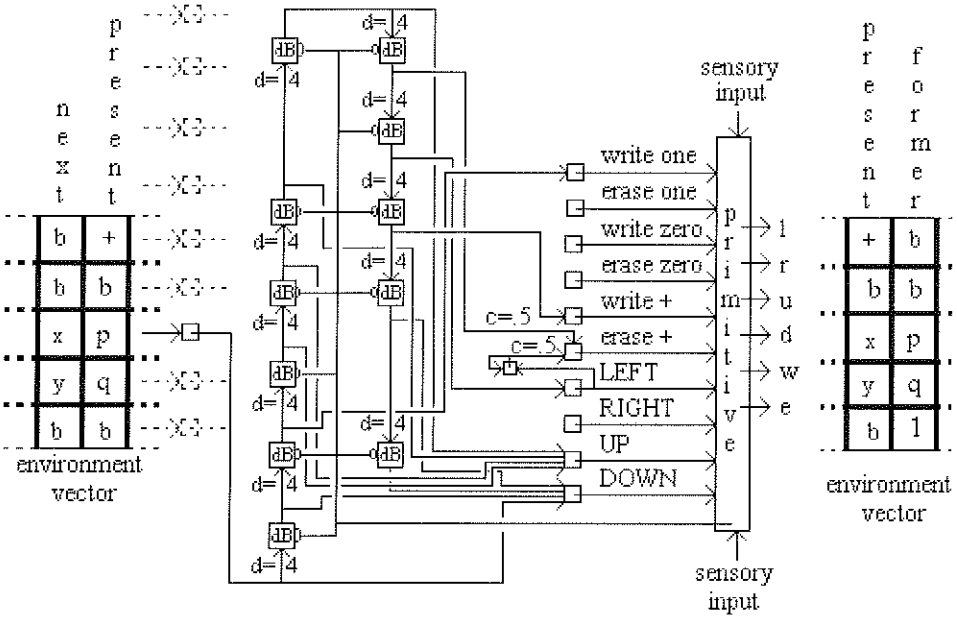


Fig. 2.11. *The adder subnet responding to the (+, b, p, q, b) environment vector.* In the environment vectors, the symbols q and p stay either the p for either a blank or the digit zero and the q for the digit one or the q for either a blank or the digit zero and the p for the digit one, the symbol b represents a blank space and the symbol x, y, represents either a blank space or the digit zero or the digit one. At the left hand side we have depicted the present and the next environment vectors, as seen by the robot sensory inputs, see the Table 2.2. At the right hand side we have depicted the former environment vector, namely the left hand side present one, and the present environment vector, namely the left hand side next one, as seen by the robot sensory inputs once the BNN has terminated its task.

In the Figure 2.11. we depict the adder subnet responding to the (+, b, p, q, b) environment vector, see the Table 2.2.. The sequence of commands the adder transmits to the primitive to change the environment vector content, from the one given at the left hand side into the one given at the right hand side, is: D, D, write-one, U, U, U, U, erase-+, L, write-+, D, D. At the adder left hand side, the dB nets, see the Figure 2.9., are activated sequentially with a four τ delay, ie, each becomes active four τs after the previous one has terminated its activity. The first becomes active four τs after the adder input becomes active. The adder remaining delays are one τ each. The arrows represent the excitatory connections. The empty circles represent the inhibitory connections. The coupling coefficients are one threshold unit, unless otherwise specified. That is, the existential node at the very center of the figure has an output connection with an associated coupling coefficient of one half of the threshold. We have put the label c = .5 to the connection between the eighth dB net, following their activation order, and the erase-+ adder output node because the plus sign must be erased if the present environment vector is not the one containing the rightmost digits of the two addends, see also the Figure 2.16.

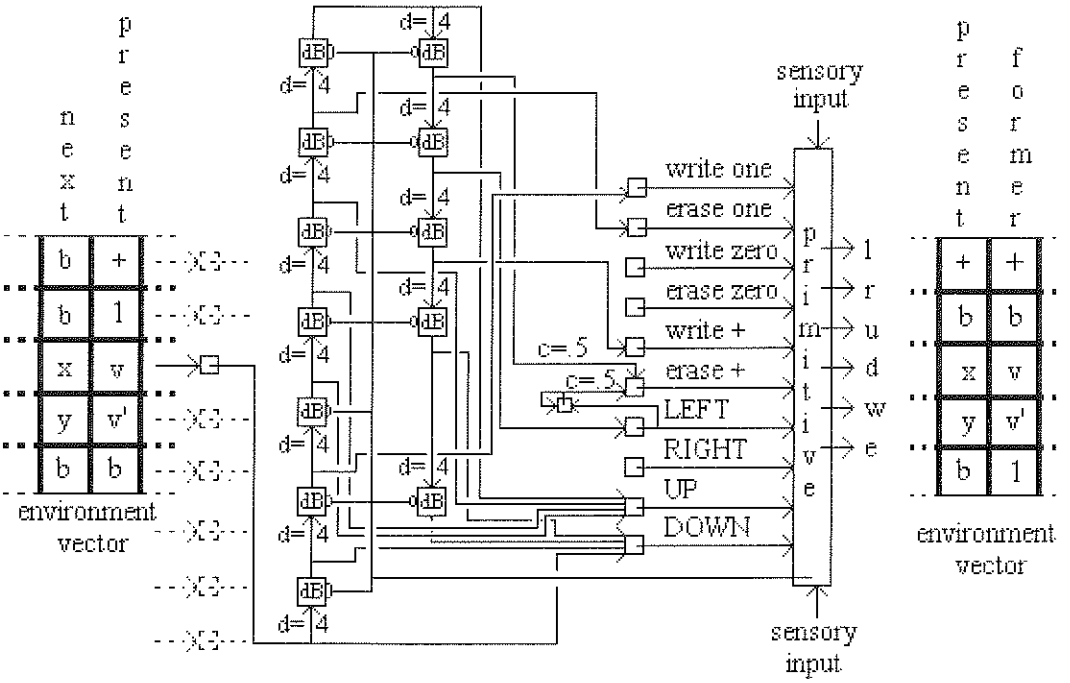


Fig. 2.12. *The adder subnet responding to the (+, 1, v, v', b) environment vector.* In the environment vectors, the symbols v and v' represent either the v' either a blank or the digit zero and the v the digit zero or the v either a blank or the digit zero and the v' the digit zero, the symbol b represents a blank space and the symbol x, y, represents either a blank space or the digit zero or the digit one. At the left hand side we have depicted the present and the next environment vectors, as seen by the robot sensory inputs, see the Table 2.2.. At the right hand side we have depicted the former environment vector, namely the left hand side present one, and the present environment vector, namely the left hand side next one, as seen by the robot sensory inputs once the BNN has terminated its task.

In the Figure 2.12. we depict the adder subnet responding to the (+, 1, v, v', b) environment vector, see the Table 2.2.. The sequence of commands the adder transmits to the primitive to change the environment vector content, from the one given at the left hand side into the one given at the right hand side, is: D, D, write-one, U, U, U, erase-one, U, erase-+, L, write-+, D, D. At the adder left hand side, the dB nets, see the Figure 2.9., are activated sequentially with a four τ delay, ie, each becomes active four τs after the previous one has terminated its activity. The first becomes active four τs after the adder input

becomes active. The adder remaining delays are one τ each. The arrows represent the excitatory connections. The empty circles represent the inhibitory connections. The coupling coefficients are one threshold unit, unless otherwise specified. That is, the existential node at the very center of the figure has an output connection with an associated coupling coefficient of one half of the threshold. We have put the label c = .5 to the connection between the eighth dB net, following their activation order, and the erase-+ adder output node because the plus sign must be erased if the present environment vector is not the one containing the rightmost digits of the two addends, see also the Figure 2.16.

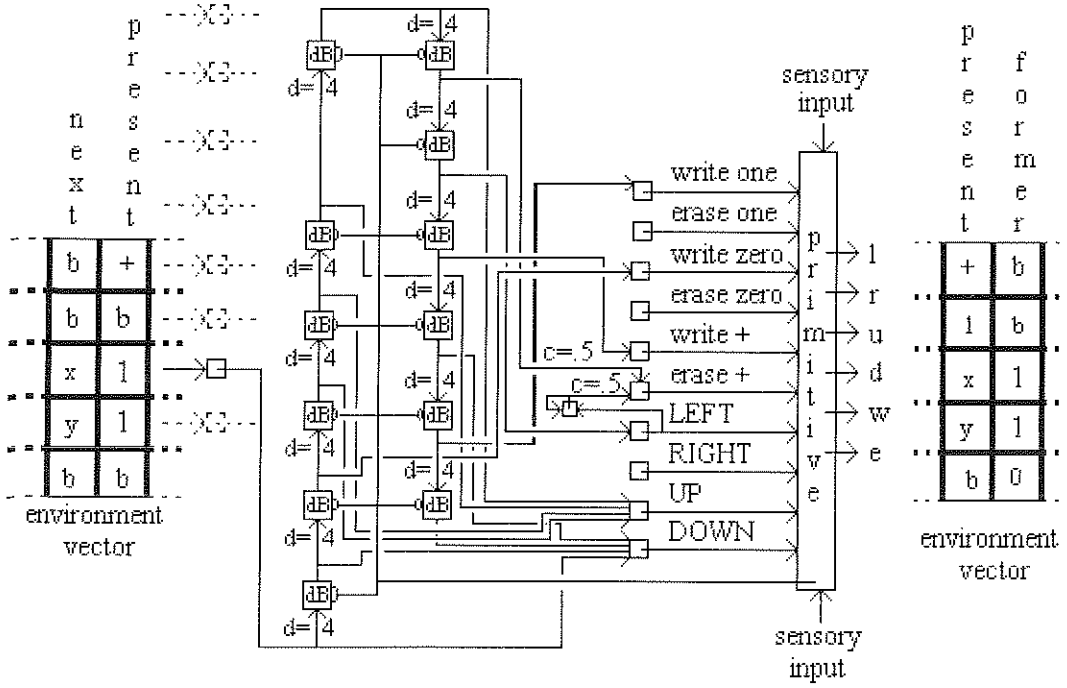


Fig. 2. 13.. *The adder subnet responding to the (+, b, 1, 1, b) environment vector.* In the environment vectors, the symbol b represents a blank space and the symbol x, y, represents either a blank space or the digit zero or the digit one. At the left hand side we have depicted the present and the next environment vectors, as seen by the robot sensory inputs, see the Table 2. 2.. At the right hand side we have depicted the former enviro n-ment vector, namely the left hand side present one, and the present environment vector, namely the left hand side next one, as seen by the robot sensory inputs once the BNN has terminated its task.

In the Figure 2.13. we depict the adder subnet responding to the (+, b, 1, 1, b) environment vector, see the Table 2.2.. The sequence of commands the adder transmits to the primitive to change the environment vector content, from the one given at the left hand side into the one given at the right hand side, is: D, D, write-zero, U, U, U, U, erase-+, L, write-+, D, write-one, D. At the adder left hand side, the dB nets, see the Figure 2.9., are activated sequentially with a four τ delay, ie, each becomes active four τs after the previous one has terminated its activity. The first becomes active four τs after the adder input becomes active. The adder remaining delays are one τ each. The arrows represent the excitatory connections. The empty circles represent the inhibitory connections. The coupling coefficients are one threshold unit, unless otherwise specified. That is, the existential node at the very center of the figure has an output connection with an associated coupling coefficient of one half of the threshold. We have put the label c = .5 to the connection between the seventh dB net, following their activation order, and the erase-+ adder output node because the plus sign must be erased if the present environment vector is not the one containing the rightmost digits of the two addends, see also the Figure 2.16.

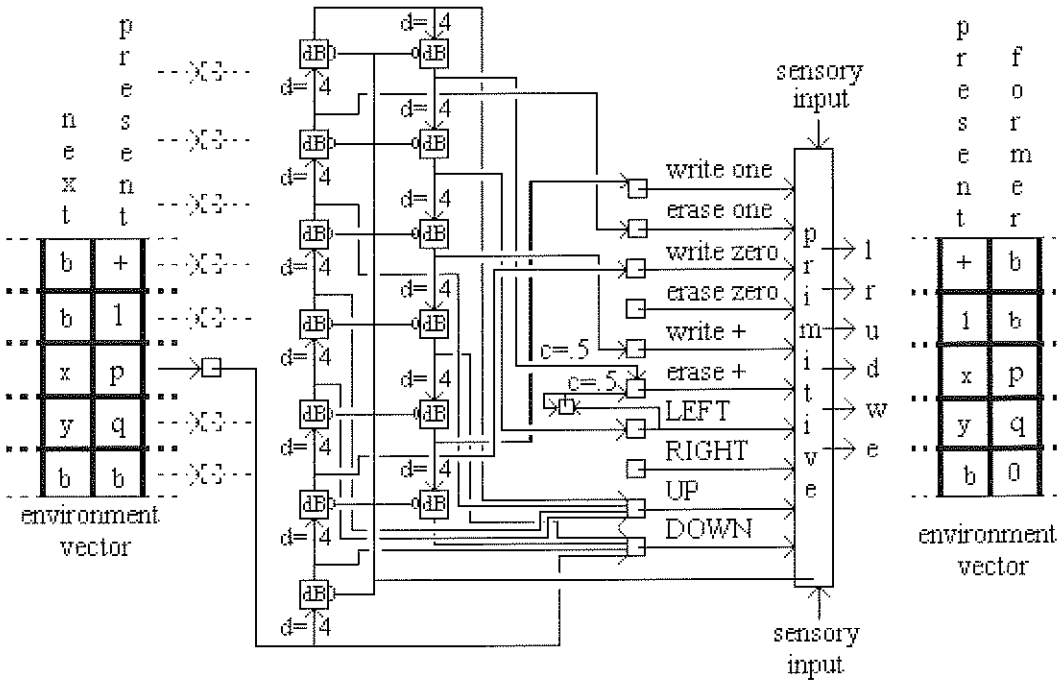


Fig. 2.14. *The adder subnet responding to the (+, 1, p, q, b) environment vector.* In the environment vectors, the symbols q and p stay either the p for either a blank or the digit zero and the q for the digit one or the q for either a blank or the digit zero and the p for the digit one, the symbol b represents a blank space and the symbol x, y, represents either a blank space or the digit zero or the digit one. At the left hand side we have depicted the present and the next environment vectors, as seen by the robot sensory inputs, see the Table II. At the right hand side we have depicted the former environment vector, namely the left hand side present one, and the present environment vector, namely the left hand side next one, as seen by the robot sensory inputs once the BNN has terminated its task.

In the Figure 2.14. we depict the adder subnet responding to the (+, 1, p, q, b) environment vector, see the Table 2.2.. The sequence of commands the adder transmits to the primitive to change the environment vector content, from the one given at the left hand side into the one given at the right hand side, is: D, D, write-zero, U, U, U, erase-one, U, erase-+, L, write-+, D, write-one, D. At the adder left hand side, the dB nets, see the Figure 2.9, are activated sequentially with a four τs delay, ie, each becomes active four τ after the previous one has terminated its activity. The first becomes active four τs after the adder input becomes active. The adder remaining delays are one τ each. The arrows represent the excitatory connections. The empty circles represent the inhibitory connections. The coupling coefficients are one threshold unit, unless otherwise specified. That is, the existential node at the very center of the figure has an output connection with an associated coupling coefficient of one half of the threshold. We have put the label c = .5 to the connection between the eighth dB net, following their activation order, and the erase-+ adder output node because the plus sign must be erased if the present environment vector is not the one containing the rightmost digits of the two addends, see also the Figure 2.16.

In the Figure 2.15. we depict the adder subnet responding to the (+, 1, 1, 1, b) environment vector, see the Table 2.2.. The sequence of commands the adder transmits to the primitive to change the environment vector content, from the one given at the left hand side into the one given at the right hand side, is: D, D, write-one, U, U, U, erase-one, U, erase-+, L, write-+, D, write-one, D. At the adder left hand side, the dB nets, see the Figure 2. 9., are activated sequentially with a four τ delay, ie, each becomes active four τs after the previous one has terminated its activity. The first becomes active four τs after the adder input becomes active. The adder remaining delays are one τ each. The arrows repre-

sent the excitatory connections. The empty circles represent the inhibitory connections. The coupling coefficients are one threshold unit, unless otherwise specified. That is, the existential node at the very center of the figure has an output connection with an associated coupling coefficient of one half of the threshold. We have put the label c = .5 to the connection between the eighth dB net, following their activation order, and the erase-+ adder output node because the plus sign must be erased if the present environment vector is not the one containing the rightmost digits of the two addends, see also the Figure 2.16.

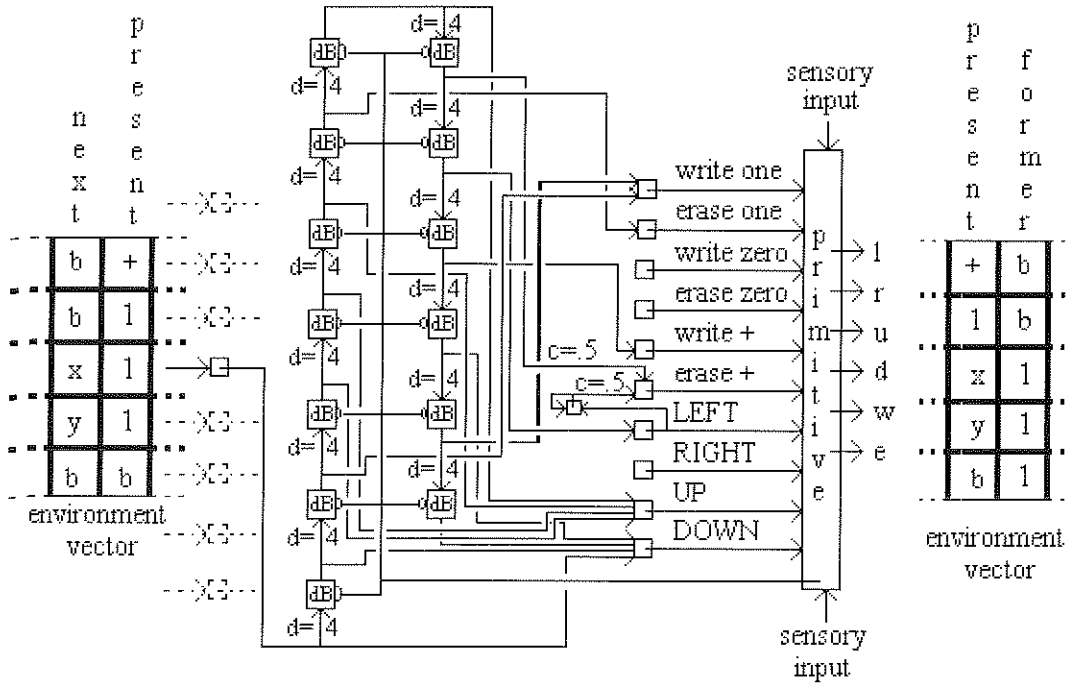


Fig. 2.15. *The adder subnet responding to the (+, I, I, I, b) environment vector.* In the environment vectors, the symbol b represents a blank space and the symbol x, y, represents either a blank space or the digit zero or the digit one. At the left hand side we have depicted the present and the next environment vectors, as seen by the robot sensory inputs, see the Table 2. 2.. At the right hand side we have depicted the former enviro n-ment vector, namely the left hand side present one, and the present environment vector, namely the left hand side next one, as seen by the robot sensory inputs once the BNN has terminated its task.

In the Figure 2.16. we depict the adder subnet responding to the (+, b, b, b, b) environment vector, see the Table 2.2.. The sequence of commands the adder transmits to the primitive to change the environment vector content, from the one given at the left hand side into the one given at the right hand side, is: U, U, erase-+, L, D, D. At the adder left hand side, the dB nets, see the Figure 2.9., are activated sequentially with a four τ delay, ie, each becomes active four τs after the previous one has terminated its activity. The first becomes active four τs after the adder input becomes active. The adder remaining delays are one τ each. The arrows represent the excitatory connections. The empty circles represent the inhibitory connections. The coupling coefficients are one threshold unit, unless otherwise specified. That is, the existential node at the very center of the figure has an output connection with an associated coupling coefficient of one half of the threshold. We have put the label c = 1 to the connection between the second, from the bottom, dB net and the erase-+ adder output node, to stress its difference from the previous cases. Here the coupling coefficient associated to the said connection has the value one, because the plus sign must be erased if the present environment vector is (+, b, b, b, b).

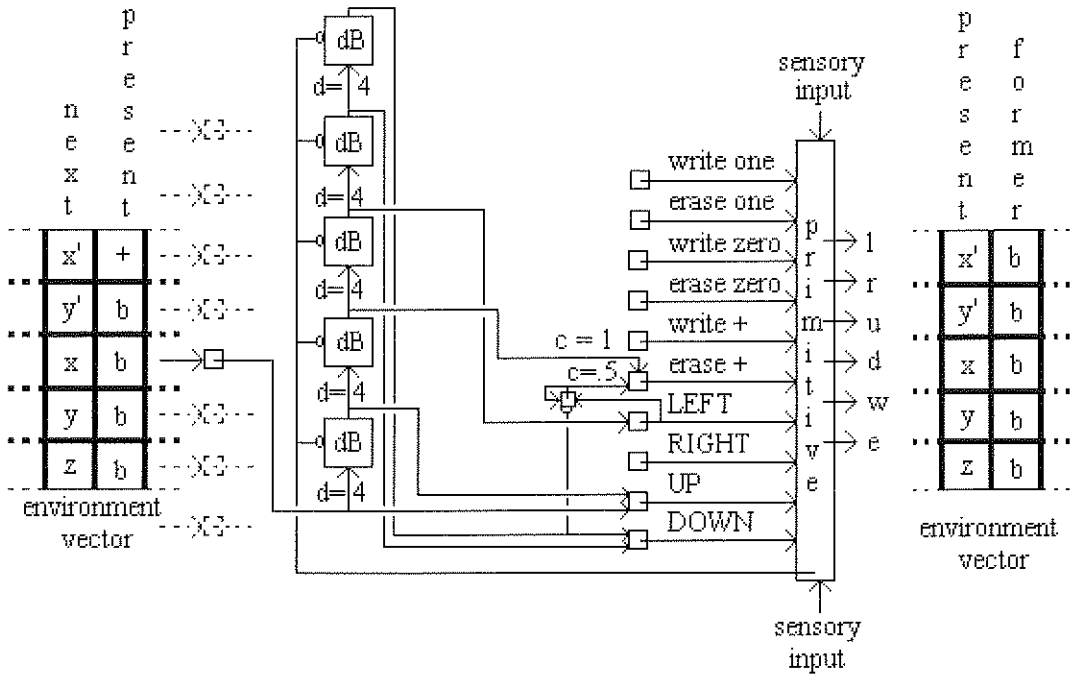


Fig. 2.16.. *The adder subnet responding to the (+, b, b, b, b) environment vector.* In the environment vectors, the symbol b represents a blank space, the symbol x' represents either the plus sign + or a blank space, the symbol y' represents either the digit one or a blank space and the symbols x, y, z represent, one independently of the others, either the blank space or the digit zero or the digit one. At the left hand side we have depicted the present and the next environment vectors, as seen by the robot sensory inputs, see the Table 2. 2.. At the right hand side we have depicted the former environment vector, namely the left hand side present one, and the present environment vector, namely the left hand side next one, as seen by the robot sensory inputs once the BNN has terminated its task.

As the ten vectors we have given in the Table 2.1. and 2.2. exhaust all the possible elements belonging to the Pythagorean table for the addition, taking as well into account the carry, then, the BNN outputting the correct response for each of them computes the addition for binary numbers represented by numerals of unbounded length.

2.3. Conclusions

As seen in the chapter 1. 3., we can consider the BNN state vector composed by the three subsets: the input subvector, the hidden subvector and the output subvector. Given a code, eg the ASCII code, we shall say *the BNN input subvector is set to the value V, if the sequence composed by the input subvector component values is the word representing the value V in the given code.* On the other hand, given a one-to-one correspondence projecting the sequences composed by the BNN output subvector component values into either the primitive commands, or primitives, controlling a robot effectors, first case, or into an alphabet, second case, we shall say *the BNN output subvector components are set to the value V, if the output subvector component value sequence is the macros sequence controlling the robot to write the value V, in the first case, is the sequence of symbols belonging to the given alphabet, ie, the word representing the value V, in the second case.* Accordingly, from now on, we shall say that *a BNN computes the function g(x) if, once the BNN input subvector components have been set to the pair [x, g], where g can be lacking if the independent variable value x uniquely defines the function g, the BNN output subvector components are set to the value g(x), after finitely many τs.*

We have presented the BNN controlling the robot adding two binary numbers represented by numerals composed by an arbitrary number of digits. It is composed by the

adder and the primitive subnets: the former output signals activates the latter output sequences. In neither of the two nets there is a node representing the symbol plus sign or the digit one or the digit zero. Instead, we have stored in them, especially in the adder, the relations the robot has to have with the said symbols. Namely, to write or to erase-one particular symbol or to move up or down on the presently observed environment vector or on the next environment vector. In other words, whereas the adder outputs towards the primitive is reminiscent of the hypertext links, in the end both of them are pointers to knowledge stored elsewhere. As earlier discussed, in the BNNs the output connections of any node play the unlabelled pointer roles, because we have fixed the adder and the primitive output connections. Accordingly, the nodes activity sequence is the BNN control flow. That is, the adder and the primitive BNN subnets inactive, active, output connections are the quiescent, actual, unlabelled pointers storing the knowledge, ie, the sequences of actions to be controlled by the BNN.

Let us consider a BNN computing the multiplication, always on operands represented by numerals with an unbounded number of digits. Obviously, we can arrange for the new BNN to have the adder-primitive subnet executing its sums. In other words, the knowledge can be incrementally increased. This is a particular aspect of a general property: the algorithms can be functionally multiplied, once their input and output conditions have been suitably normalized. E. g., to normalize the adder-primitive BNN we have introduced the convention: all the journeys must start and end on the present environment vector middle component central unitary square.

As earlier noted, the character recognition problem becomes irrelevant if we give the data, ie, the characters as ASCII codes. Analogously, a number of problems, among them the "*grand-mother neuron*" problem or how the information is coded into the signals flowing between the neurons, dissolves into insignificance. To see it, let us consider the Pascal sequence like Sqrt(Num): it is the calling sequence for the macro computing the function restituting the square root of the argument Num. Actually, Sqrt is a pointer, ie, the name of the address of the memory location containing the first instruction of the said macro. In other words, the knowledge can be represented in a non coded form as a pointer, namely as the input bitmap representing the name of the place containing a way to output the very same knowledge. That is, the knowledge can be stored as a pointer to a program or to a correspondence table restituting a representation of the desired symbol sequence. However a pointer has to be symbolically represented as a word in the input alphabet in sequentially organized machines. Eg, in a von Neumann architecture the Next Instruction Register is a bottleneck requiring precisely such representation. As in the central nervous system the equivalent of a pointer at run time is the activity in a number of axons, simply there is no need to code the information flowing between the neurons. Accordingly, the input pattern corresponding to the grand-mother can be seen as the pointer to a macro to be computed with the values for the independent variable values given at run time. Eg, it is long time since I saw the grand-mother or today is my birthday and the grand-mother is coming with a pie, or today I broke the grand-mother spectacles or I have crushed the grand-mother' s cat' s tail. However, if in each of the said input subvector states the grand-mother is substituted with the mother or the aunt Mary the system response is substantially the same. Eventually, the input subvector pattern corresponding to the grand-mother is simply one of the many calling sequences for some output macro. As such it does not characterize or correspond to any particular set of neurons.

To control the robot in the execution of the sum or of the difference or the multiplication, the very same output subsequence, e. g. write-one, becomes active in response to many different input sequences. In other words, any instance of the digit one, or of the grand-mother, is just a subsequence of one of the many input sequences associated to one output sequence. Any particular output subsequence can be present in the response to many different input sequences. E. g., to the environment vector with a plus, or a minus, sign in the sign component, or to the smiling mother, or grand-mother. A BNN can output a subsequence, in any case a primitive sequence, in response to many different input

subsequences as the output subsequence is part of the system response to each of the latter. Accordingly, in a BNN no node represents a particular environment event and the output subsequences do not characterize the environment events. Moreover, we do not need to modulate the signals to represent the information flowing between the adder BNN nodes. The algorithms computed by a BNN are stored in its wiring diagram, ie, in its coupling coefficient matrix. The connections between the nodes play the unlabelled pointer roles: it is sufficient a signal presence to activate the rightly connected neuron.

To dispel the conclusion apparent peculiarity, let us consider the relation

$$y = P[x].$$

The y represents a datum, ie, the result of the computation of the given relation and x represents the independent variable. Whereas we give always, implicitly or explicitly, the set of values on which both x and y are defined, the name P[x] does not represent the actual computation, ie, the actual sequence of steps we need to execute to compute the relation. That is, whereas P represents only one out of the many possible computations, even if we say it is, eg, the exponential or the factorial function, the data belong to well defined sets. E. g., in a^b the digit b always represents the number of multiplication we have to perform, the particular algorithm computing it is deemed irrelevant. Usually, in a von Neumann architecture a routine is stored as the sequence of values the machine interprets as the instructions it has to execute to compute the routine. That is, it is stored as a sequence composed by subsequences, each present in the computation of many different routines. It is not stored as a sequence of values the machine interprets as the routine output value. Or as a sequence uniquely characterizing one particular input sequence. Similarly, in a BNN the grand-mother input is just one out of the many calling sequences for one of the many possible output routines. As in a BNN an entire message information content, not just the single bit, is associated to a signal presence, we do not need to code the data to transmit them. In the next chapter we are going to discuss how the BNN knowledge representation can be implemented in a traditional data structure.

Chapter 3

THE DATA STRUCTURE EMBEDDED IN A BNN.

3.1. Introduction

Once and for all, let us stress we are dealing, in this chapter, only apparently with the traditional data structures, like vectors and records or pointers. Actually, we are dealing with BNNs and nodes or connections. The likes of the dB subnet, see the figure 2.9., and of the *assemblies*, we shall discuss in the chapter 5, justify the delayed reading of the pointers contained in the records, see the chapter 3.2. Because of the way the data are stored in a BNN, see the chapter 2., and, as we shall see later on in the chapter 5., also in an adaptable Boolean net, it is convenient to discuss here a traditional data structure equivalent to the one found in the BNNs. Actually, the matter we are dealing with here is not a logical prerequisite for the subsequent chapters content. However, to understand the data structure embedded in and the way the data are represented and stored in a net organized in assemblies, the characterization discussed in the present chapter is essential.

Elsewhere, see [6, chap.5], we have presented a data structure, here called the dataless data structure, or *DLDS*. A DLDS is a traditional version of an unlabelled pointers data structure. As such it enjoys some properties we would unsuccessfully look for in the traditional data structures.

Usually, e. g. see [11, 12, 13], a record is the elementary component of a data structure holding the information in a number of fields. In other words, a record is a monodimensional array with finitely many fields as components. Eventually, a component capacity depends on the amount of the data we want to store in it. To sort the records with respect to their fields content, we choose one key. Unless we are ready to pay the price for the additional fields to store the pointers needed to sort the records with respect to different keys. Accordingly, in general a traditional data structure is sorted with respect to just one key.

By contrast, we can arrange the pointers in a DLDS, so to have it sorted on many different orders at once, as in a DLDS the information is stored in the pointers. Moreover for the very same reason, the DLDS storage requirements are smaller than and the reading access times comparable with respect to the data structures holding the data as sequences of symbols drawn from a given alphabet.

To understand the DLDS organization, we have to discuss how to store an item so to save storage space. An item is defined as the set of K keys characterizing a datum. Usually, a K keys item is stored as a K + 1 fields record, namely a record holding just one key per field, plus the information holding field. By contrast, in a DLDS an item is the ordered set of K keys, ie, the data are stored as keys. That is, in a DLDS each key, of

each item, is stored in a standard record holding only pointers. And all the k-th keys, for $0 < k \leq K$, are the component of one sequentially or randomly accessed sorted structure. Then, it is sufficient to sort circularly the keys, to retrive an item on any key whatever. However, this requires at least one pointer per key to individuate the next key. Accordingly, our solution seems to increase the amount of storage space required to hold an item. Unless we can prove the convenience of storing the data by means of pointers to string of symbols common to many different data, even belonging to different data structures.

To this end let us consider both the alphabet A, having the cardinality A, and the sequence S of symbols belonging to A. As the number of symbols in S increases and becomes greater than A, also the number of symbol repetitions increases. Next, let us consider the alphabet A_2, having cardinality A^2, containing all the A pairs: as the number of symbols in S increases and becomes greater than A^2, increases also the number of A_2 element repetitions. In other words, as the number of symbols we want to store increases and becomes greater than the alphabet cardinality, the number of repeated subsequences composed by single symbols, pairs of symbols, triples of symbols, and so on, increases. Eg, once given the numerical base, all the sequences composed by two, three, four, and so on, digits are repeated as the number to be represented by the numeral increases. By contrast, once fixed a natural language, only some of the alphabetic character sequences are repeated, again and again.

Actually, given the alphabet A of cardinality A, let us consider a string, or S-word, S composed by S symbols belonging to A. That is, we call S, namely the number of symbols in S, the string length, or word length: for short we call b-word, b-string, a word, a string, of length b. Obviously, for $S > A$, the string contains at most $A - 1$ symbols present just once, ie, in S there are no less than $S - A + 1$ repeated symbols. Now if B is the alphabet of all the b-words, then the alphabet B cardinality is A^b. Accordingly, for $S > (A^b + 1)b$ the string S contains no less than $S - A^b + 1$ repeated b-strings, ie, there are at most $A^b - 1$ b-words present just once in S.

In general, to store a symbol we need s bytes, accordingly, to store a b-word, as a string of symbols, we need b times the amount we need to store a single symbol, to store A^b symbols as a string of symbols, we need A^b times the said amount and to store A^b b-words, as strings of symbols, we need $A^b bs$ bytes. However, if to store a pointer we need p times the storage space needed to store a single symbol and p is smaller than b, we can be cleaver enough to store a b-word first occurrence so that from its second occurrence onward, instead to store it as many times as the very same repeated string of symbols, we store instead as many pointers to the address of the memory word holding its first occurrence. That is, after a b-word first occurrence, namely from its second occurrence onward, we save $(b - p)s$ bytes every time we store it as a pointer to its first occurrence. Let us call n the total number of b-word repetitions in the string S, then, we start to save storage space as soon as

$$n(b - p) > A^b b,$$

ie, as n is greater than $A^b b / (b - p)$. Accordingly, if S length is no smaller than $(A^b + 1)b / (b - p)$ it is more efficient to store S by means of the pointers than as a string of symbols. That is, to store a string having the length $(A^b + n)b$, as a string of symbols, we need $(A^b + n)bs$ bytes. That is, for

$$(A^b + n)b > (A^b + 1)b / (b - p),$$

ie, for

$$n > (A^b + 1) / (b - p) - A^b$$

it is more efficient to store the string by means of the pointers than as a string of symbols.

Accordingly, if S length is greater than $(A^b + 1)bs / (bs - p)$ it is more efficient to store the data by means of pointers to b-words than as a sequence of symbols. Eg, for A = 100, namely if we consider the alphabet of the keyboard symbols, b = 3, namely if we consider the symbol triplets, s = 1, namely if a symbol requires just one byte to be stored, and p = 2, namely if a pointer requires two bytes, ie, two times the amount required for a symbol to be stored and we must store more than 3,000,003 symbols, then, we can hold them more efficiently by means of pointers to triplet of symbols than as symbols. Instead, for A = 15, namely if we consider the numerals in base ten plus some special character as the blank space and the return, b = 4, s = 1 and p = 2, namely if a symbol requires just one, a pointer two bytes to be stored and we must store more than 101,252 digits, then, we can hold them more efficiently by means of pointers to four digit numerals than in a traditional way as symbol sequences.

Obviously, as we increase the S length, we must also increase the number of bytes allocated for a pointer. However, whereas it is always possible to find a b greater than p as b increases, for a given amount of data to store, the single b-word frequency decreases, ie, the number of single occurrence b-words increases. Accordingly, the storage space saved decreases. We shall give the algorithms to operate the DLDSs, for small b. As we shall see, once chosen b, we can start to save space at a lower S length if we store only the strings both having the length greater than b and actually present in the data, ie, if we do not store all the strings having the length greater than b.

We start by assigning an alphabet, eg the keyboard character set. Next, we use pointers to represent the symbol sequences, eg to represent the items to be stored in the DLDSs. After a sequence first occurrence, independently of the number of symbols it is composed of, we represent it just as one pointer to the said first occurrence. Accordingly, as the number of sequences to store in a DLDS increases, it increases both the number of previously stored sequence first occurrences and the amount of storage space we save.

As we shall see, each key of each item is stored in a standard record, ie, a record holding a fixed number of pointers. Instead of storing an item in one single randomly accessed list sorted on one particular key, to store an item we have assigned as many randomly accessed lists as the number of keys per item, each list being sorted independently of the others. That is, to store H items, each represented by K keys, we keep all the k-th keys, with $0 < k \leq K$, stored in the k-th randomly accessed H component list. And, as the item keys are one independent of the others, we sort the k-th randomly accessed list on the k-th keys order. The multisorted structure is also the DLDS Achilles' heel.

Because to represent a key we need at least three pointers, every time we insert a new K keys item, we need to set no less than 3K pointers. On top of this, we need to set a number of pointers which depends on the number of symbols any new sequence representing a key to insert is composed of. As we shall see, similar considerations apply to the erasing accesses.

We shall present two DLDS versions: in one version, an item k-th key is represented by three pointers held in a record which, in turn, is a component of the k-th key vector. In the other version, each record holds six pointers, plus a monobit field, and is a node of the k-th red-black tree, see [3]. Whereas the former version requires less storage space and longer writing access times than the latter, in the former more reorderings can be batched together to be executed at a later time, both without seriously degrading the reading access times and at the same time spreading their cost on more writing accesses.

Some mono-sorted DLDS versions have been discussed in [6, 14, 15, 16].

3.1.1. *Names and variables list.*

A	the alphabet and its cardinality: the items are represented as sequences of symbols belonging to A: Chapter 3.1.
birecord	a two field record holding two pointers: Chapter 3.2.1.
Bottom	a T vector bottom component: Chapter 3.4.2.2.
constant part	subset of the DLDS arrays common to many different data structures: Chapter 3.2.1.
C	counter: Chapter 3.2.1.
CONT[FP[STRUCTURE, RECORD]]	the content of the first field of the record pointed to by the first pointer held RECORD, where RECORD is a record belonging to the structure STRUCTURE: Chapter 3.2.1.
CONT[x]	value of the content of the record first field pointed to by the pointer x: Chapter 3.2.1.
dirty key vector	a key vector holding supernumerary keys: Chapter 3.4.2.2.
DLDS	Data Less Data Structure: Chapter 3.1.
FF	first field: Chapter 3.2.1.
FP	first pointer: Chapter 3.2.1.
h, k	temporary variables: Chapter 3.3.6.
H	number of K key items stored in the DLDS: Chapter 3.1.
intermediate list	a list holding records common to many different data structure: Chapter 3.2.3.1.
K	number of keys for item: Chapter 3.1.
K'	the order of the number of pointers we must follow to retrieve and read a K key item, starting from any one of its keys: Chapter 3.3.5
K"	the order of the number of pointers we must follow to read an item having one of its keys equal to one of the keys of the previously retrieved and read item: Chapter 3.3.6.1.
key structure	either a key tree or a key vector: Chapter 3.2.3.
k-th key tree	a H nodes red-black tree: in the k-th red-black tree the h-th node holds the pointers to the strings of symbols representing the k-th key of the h-th item stored in the DLDS: Chapter 3.2.3.2.1.
k-th key vector	a H components vector: in the k-th key vector the h-th component holds the pointers to the strings of symbols representing the k-th key of the h-th item stored in the DLDS: Chapter 3.2.3.2.2.
MAX	the length of the longest sequence stored in a linked list birecord first field: actually, in the DLDSs we discuss, all the sequences having the MAX length are held in the third linked list birecord first fields: Chapter 3.2.3.1.
monorecord	a one field record holding one pointer: Chapter 3.2.1.
N	number of A symbols in the sequence representing an item key, also the sequence length: Chapter 3.2.3.2.
record	either a monorecord or a birecord or a three-record or a six-record: Chapter 3.2.1.
SF	second field: Chapter 3.2.1.
six-record	a seven field record holding six pointers and a monobit flag in the seventh field: Chapter 3.2.1.
SP	second pointer: Chapter 3.2.1.
TF	third field: Chapter 3.2.1.
three-record	a three field record holding three pointers: Chapter 3.2.1.
Top	a T vector top component: Chapter 3.4.2.2.

TP	third pointer: Chapter 3.2.1.
T vector	temporary bi-component vector holding a supernumerary key in a dirty key vector: Chapter 3.4.2.2.
variable part	subset of the DLDS arrays and liked lists characterizing just one data structure: Chapter 3.2.1.
◊	the symbol representing the blank space in the illustrative examples: Chapter 3.2.6.

3.2. The structure components

3.2.1. *The records and the arrays*

We call *monorecord* a record holding either one symbol or one 2-word, ie, one sequence of just two symbols.A monorecord has jut one field, or FF, holding one symbol: the monorecord is identified by the symbol it holds.

We call *birecord* a record holding two pointers. A birecord has two fields: the first field, or FF, and the second field, or SF. In a birecord each field holds a pointer: FF holds the first pointer, or FP, and SF holds the second pointer, or SP.

We call *three-record* a record holding three pointers. A three-record has three fields: the first, the second and the third field, respectively the FF, the SF and the TF. In a three-record each field contains a pointer: the FF holds the first, the SF holds the second and the TF holds the third pointer, respectively the FP, the SP and the TP.

We call *six-record* a record holding six pointers. A six-record has seven fields: the first, or FF, the second, or SF and the third, or TF, field holding, respectively, the first, or FP, the second, or SP, and the third, or TP, pointer. That is, the first three fields holds the data. The next four fields are needed to operate the red-black tree: three fields hold, respectively, the pointers: parent, right-offspring and left-offspring. The seventh is a field holding a monobit flag: its content represents the node color, ie, it is said to be either red or black.

From now on a *record* is either a monorecord or a birecord or a three-record or a six-record or a combination of them.

In an DLDS there is a *constant part*, ie, a given set of vectors having as components the monorecords holding the symbols, the symbol pairs, the symbol triplets etc.. And a *variable part*, ie, a number of vectors, or red-black trees, and linked lists holding their first pointers to the constant part monorecords. In the DLDS version holding the keys in vectors, the variable part vectors have as components the three-records and are sorted circularly. In the DLDS version holding the keys in red-black trees, the variable part red-black trees have as components the six-records and are sorted circularly. Obviously, the number of variable part vectors, or red-black trees, is equal to the number of keys per item. By contrast, the variable part linked lists are linearly ordered: whereas the first linked lists have the three-records as components the last linked list components are birecords.

If the alphabet is composed by A symbols and if the total number of symbols representing all the data to store is A^m, then the number of monorecords must be kept small with respect to A^m, eg in the order of $A^m/10^2$. That is, if the alphabet has a cardinality of 10^2 and the total number of symbols needed to represent the items to store is 10^6, then it is convenient to have in the order of 10^4 monorecords as the structure constant part.

Tu sum it up, it is convenient to have a constant part composed by one set of vectors, the *first vector*; the *second vector* and so forth. The first vector contains all the alphabet symbols; the second vector contains all the 2-words, ie, all the two symbol strings; and so on.

We call CONT[x] the content, of the first field, of the record pointed to by x: eg, in the monorecord case CONT[x] is a b-word, with b either 1 or 2. Similarly, we call CONT[NEXT[FP[STRUCTURE, RECORD]]] the content of the second field of the record pointed to by the pointer held in the first field of the record RECORD belonging to the structure STRUCTURE, where a structure is either a linked list, a vector or a red-black tree.

To operate a DLDS we need also some temporary pointers and a counter C, ie, a store for an integer no smaller than K, the keys number.

3.2.2. *The DLDS constant part.*

A constant part vector is set up once for all as the number of its components depends on the alphabet cardinality. Accordingly, the constant part could be common to many distinct DLDSs, so sharing its cost between them.

Given the A symbols alphabet A, we consider the first vector having as components A monorecords, ie, its h-th component contains the alphabet h-th symbol. The second vector has A^2 monorecords as components, each containing one 2-word, with the pairs lexicographically sorted. Depending on the representation of the data to be stored, eg if the data are numbers represented by numerals, it could be convenient to have also a third vector holding all the 3-words. Instead, in the DLDSs discussed here we store in a linked list, the *third linked list*, the 3-words actually present in the data to be stored: we consider the constant part composed by the first and second vectors, only.

3.2.3. *The DLDS variable part.*

As discussed in the chapter 2. 1., it is convenient to set an upper bound on the number of vectors constituting the constant part. That is, it is convenient to set an upper bound on the number of vectors holding **all** the strings composed by a given number of symbols. Accordingly, in the variable part there must be both records, we call them the *intermediate* records, in turn holding pointers to the constant part vector components and *key records* holding the pointers to the constant part and to the intermediate records so to control the output of the strings of symbols representing the customer keys. The key records are held by the *key structures*: either red-black trees, for short *key trees*, or vectors, for short *key vectors*. Whereas the key structures components are either three-records or six-records, the intermediate records are either three-records or birecords and belongs to the linked lists we discuss next. Among the variable part structures, the linked lists holding the intermediate records can be common to many distinct DLDSs, so sharing their cost between them.

3.2.3.1. *The linked lists as intermediate structures.*

As previously seen, the *intermediate* linked lists components are the three-records and the birecords. In a three-record belonging to the linked list L, the first two pointers, namely the FP and the SP, point either to the first field of a record belonging to a linked list different from L or to a monorecord belonging to a constant part vector. The third pointer, namely the TP, points to the very same linked list next three-record.

In the DLDS we discuss here, there is just one linked list, the *third linked list*, having birecords as components. In the third linked list, each birecord first field holds one of the 3-words actually present in the stored data and the second pointer, namely the SP, points to the very same linked list next birecord. The present choice depends on the rela-

relative cost to store symbols and pointers and both let us save storage space and simplify the routines to access the DLDS.

Thanks to our choice, in any given linked list all the three-records hold their pointers to the subset of the sequences of symbols both having a given length and actually stored in the DLDS. In other words, a given linked list does not hold the pointers to **all** the possible sequences composed by the same number of symbols: it does hold only the pointers to the sequences composed by the same number of symbols and actually present in the DLDS.

A linked list name is an ordinal greater than the ordinals identifying the vectors belonging to the DLDS constant part. Because in our DLDS constant part there are only the first and the second vector, we call the linked lists as *third*, *fourth*, and so on.

Given the three-record B holding two pointers pointing to the three-records C and D, to avoid loops in a DLDS *the number of symbols present in the sequence pointed to by B first two pointers must be greater than the number of symbols present in the sequences pointed to by the first two pointers of either C or D.* In the following, we shall observe the said condition.

We represent as `FP[THIRD, intermediate]`, `SP[THIRD, intermediate]`, the first, second, pointer held by the birecord `intermediate` belonging to the linked list `THIRD`. We represent as `FP[FOURTH, intermediate]`, `SP[FOURTH, intermediate]`, `TP[FOURTH, intermediate]`, the first, second, third, pointer held by the three-record `intermediate` belonging to the linked list `FOURTH`. Similarly, with `FF[FOURTH, intermediate]`, `SF[FOURTH, intermediate]`, we represent the first, second, field of the three-record `intermediate` belonging to the fourth linked list. We represent as `FP[HERE]`, `SP[HERE]`, `TP[HERE]`, the first, second, third, pointer held by the record pointed to by the pointer `HERE`.

As we have seen, the constant part monorecords and the linked list birecords hold symbol sequences of length one, two and three, respectively. By contrast the three-records content has to be chosen at the writing access time. Let us call MAX the length of the longest sequences held by a third linked list birecord first field: in the present DLDS the MAX value is three. In the three-records either the first two pointers have to point eventually to a string having a total length greater than MAX, or each of them have to point either to a birecord or to a monorecord. Accordingly, to store a string of symbols having a length greater than MAX and to simplify the reading and writing routines, we shall abide by the convention: *a three-record first pointer points to a string having the length MAX, ie, in the DLDSs here discussed, to a third list birecord. A three-record second pointer points to a record, ie, either to an intermediate three-record or to a third linked list birecord or to a constant part monorecord.* Among the many possible choices, the adopted convention yields an optimal result.

To prove the latter, let us call N the length of a key symbolic representation, for a suitable n and for $b0 \leq STM < MAX$, we have:

$$N = n^{*}MAX + STM.$$

However, n - 1 is also the number of records holding the $2(n-1)$ pointers eventually pointing, in turn, to the records holding the subsequences of the symbol sequence representing the key. The given convention yields a degenerate full binary tree, ie, a binary tree with all, but the last, internal nodes, actually the DLDS key and intermediate records, having as sons both one leaf, actually a third linked list birecord, and one internal node. The last internal node two sons are both leaves: one holding a sequence of symbols having the length MAX and the other holding a sequence having either the length STM, if STM is greater than zero, or MAX, if STM is zero. As a binary tree with n leaves has no less than $n-1$ internal nodes, the given convention yields a binary three with the minimum number of records and, hence, of pointers.

Once the total number of symbols needed to represent an item keys has been fixed, the number of records needed to hold more short keys is smaller than the number of records needed to hold few long keys. To hold a key represented by a sequence of n*MAX symbols we need n – 1 records. Accordingly, if we break the key in two halves and n is even, we need (n/2) – 1 records to hold each half and 2((n/2) – 1), ie, n – 2 records to hold them both. Otherwise, namely if n is odd, we need (((n – 1)/2) – 1) + (((n – 1)/2) – 1) + 1, ie, still n – 2 records to hold them both. In other words, to hold many short keys represented by a total number TOT of symbols we need a smaller number of records than to hold few long keys represented by the same TOT number of symbols. Moreover, as the sequences' length becomes smaller both the DLDS reading access time decreases, as we shall see in the chapter 3. 3. 5., and the probability to save storage space increases, as the storage space needed to hold a DLDS depends on the number of intermediate records common to many different keys.

3.2.3.2. *The key holding structures.*

To hold the keys we present two different data structures: the key trees and the key vectors. Whereas the former have smaller writing and erasing access times the latter require less storage space. Both have reading access times in the order of $C\log_2 H$, H being the number of items to store: however, whereas in the former C is a constant in the latter it depends on the number N of symbols the key to read is composed of.

Depending on the number of keys represented by identical symbol sequences or on the length of the symbol sequence representing a key it could be more efficient to have birecords as a key tree or a key vector component. That is, if we have many identical keys, eg the John Smith instances in a telephone book, or if we have many keys represented by symbol sequences smaller than MAX, it is certainly more efficient to represent a key with one single pointer. However, as these are special cases, we prefer to discuss a DLDS with three-records as component of the key structures.

By convention, we represent as `FP[KEY, key]`, `SP[KEY, key]` and `TP[KEY, key]`, the first, second and third pointer held by the six-record, or of the three-record, `key` held, in turn, by the key red-black tree, or key vector, `KEY`. Similarly, we represent as `FP[HERE]`, `SP[HERE]`, `TP[HERE]`, the first, second, third, pointer held by the record pointed to by the pointer `HERE`.

3.2.3.2.1. *The key trees.*

To store the k-th key of the h-th item in a six-record of the k-th red-black tree we call the routine executing the insertion, and rebalancing, algorithm for red-black trees, see [13]. Whereas the six-record color and its fourth, fifth and sixth pointers are set by applying the latter routine, its first, second and third pointers are set by following the algorithms we discuss in the chapter 4.. In a six-record the first pointer points either to a third linked list birecord or to a monorecord. Its second pointer points either to an intermediate record or to a monorecord. The third pointer points to the first field of a six-record belonging to a different key tree. Initially, the key tree six-record first three pointers are set to the End Of Job, or EOJ, routine.

3.2.3.2.2. *The key vectors.*

The key vectors have as components the three-records holding the pointers to the sequences of symbols representing the item keys, ie, the information to store. In a key vector three-record the first pointer points either to a third list birecord or to a monorecord. Its second pointer points either to an intermediate record or to a monorecord. In a key vector three-record the third pointer points to the first field of a three-record belonging to

a different key vector. Initially, the key vector three-record pointers are set to the End Of Job, or EOJ, routine.

3.2.4. *The intermediate three-records order.*

Because in a DLDS the order is of paramount importance, we shall sort the linked list on the length of the sequences, ie, on the number of the symbols present in the sequences pointed to by both their first and their second pointer. That is, we say the record B *precedes* the record C, or C *follows* B, if the total number of symbols pointed to by B first and second pointer is greater than the total number of symbols pointed to by C first and second pointer.

As in any linked list there are no two records holding identical pointers, in a linked list the three-records are sorted on the pointed sequences lexicographic order. Accordingly, to avoid inserting loops in the DLDS, it is sufficient to set the record A first and second pointer to the records B and C following A.

Eventually, the intermediate records follow the records belonging to the key structures and precede the monorecords belonging to the constant part vectors.

3.2.5. *The keys order.*

The key structures are identified by the key names and sorted circularly, not necessarily on the lexicographic order of the key names. In other words, instead of storing all the keys in one single sorted structure, we consider many different key structures, one for each key, each sorted on its own lexicographic order. Eg, in a telephone directory we consider the `surname` vector or red-black tree holding the customer surnames lexicographically sorted, the `name` vector or red-black tree holding the customer names lexicographically sorted, the `address` vector or red-black tree holding the customer addresses lexicographically sorted, the `civic#` or red-black tree vector holding the customer civic numbers lexicographically sorted, the `telephone#` vector or red-black tree holding the customer telephone numbers lexicographically sorted, etc.. Then, to access the data randomly, the key vectors, ie, the keys are sorted circularly.

In each key structure, the records are sorted lexicographically on the strings pointed to by their first two pointers, as long as the latter strings are different. Two equal strings, ie, two equal keys are entered with the order they have in the preceding key structure, see also the chapter 3. 4. 5..

By convention: *in a three-record belonging to a key structure, the first pointers value is the address either of a birecord belonging to the linked list three or of a monorecord belonging to the constant part. The second pointer value is the address either of an intermediate three-record or of a birecord belonging to the linked list three or of a monorecord belonging to the constant part. The third pointer value is the address of a record belonging to the next key structure in the circular order.*

In other words, in a DLDS organized accordingly to the given convention, if we follow the first and second pointers held in a key structure three-record, we obtain the symbol sequence representing the current key. If we follow the third pointer held in a key structure three-record, we visit the three-record holding, in its first and second fields, the pointers to the symbol sequence representing the same item next key, in the circular order.

3.2.6. *An illustrative example.*

We want to store in a DLDS a telephone directory: eg, let one entry be the customer John Smith having the telephone number 31415.

We consider the, three- or six-, record keySmith holding in its first two fields the pointers to the sequence of symbols representing the key Smith and belonging to the key structure surname. The latter is followed, in the circular order, by the key structure name containing the, three- or six-, record keyJohn holding the first two pointers to the sequence of symbols representing the key John. Because John Smith telephone number is 31415, the latter sequence is pointed to by the first two pointers held in the, three- or six-, record Smith# belonging to the key structure telephone#. Let us represent with the symbol ◊ the blank space.

Following our conventions, we call, first, FP[surname, keySmith], SP[surname, keySmith], TP[surname, keySmith], the record keySmith first, second, third, pointer; second, FP[name, keyJohn], SP[name, keyJohn], TP[name, keyJohn], the record keyJohn first, second, third, pointer; third, FP[telephone#, Smith#], SP[telephone#, Smith#], TP[telephone#, Smith#], the record Smith# first, second, third, pointer. The pointer TP[surname, keySmith] points to the record keyJohn, the pointer TP[name, keyJohn] points to the record Smith#, the pointer TP[telephone#, Smith#] points to the record keySmith.

To store by means of pointers in the surname key structure the key Smith◊, namely a subsequence of the sequence Smith◊John◊31415, we set:

CONT[FP[surname, keySmith]] = **Smi**
CONT[SP[surname, keySmith]] = **th◊**

To store by means of pointers in the name key structure the key John◊, namely a subsequence of the sequence Smith◊John◊31415, we set:

CONT[FP[name, keyJohn◊]] = **Joh**
CONT[SP[name, keyJohn◊]] = **n◊**

To store by means of pointers in the key structure telephone# the key 31415, namely a subsequence of the sequence Smith◊John◊31415, we set:

CONT[FP[telephone#, Smith#]] = **314**
CONT[SP[telephone#, Smith#]] = **15**

3.3. The reading access.

To start with, let us consider a DLDS having one first and one second vector, the linked lists three, four, ... and the key structure, with the intermediate three-records first and second pointers set in accordance with the rules given in the chapter 3. 2.. To read a record content, first, we must retrieve the record, to this end we must first call an appropriate searching algorithm, eg see [11, 12], second, whenever the searching procedure terminates with a success, we must follows the record first two pointers, in the order, all the way down. The algorithm implementing the second step are discussed in the following chapters.

3.3.1. *Reading a sequence of symbols from a record.*

Let us consider the symbol sequence stored in one of the monorecords belonging either to the first, or to the second, vector or in one of the birecords belonging to the third

linked list. To fix the ideas, the search routine terminates with a success and its result is the record pointed to by the first pointer of the record PRESENT belonging to the linked list or key structure Present, ie, pointed to by FP[Present, PRESENT]. If Output[x] is the output routine needed to read the symbol sequence held in the third list birecord or in the monorecord pointed to by x, then:

```
Output[FP[Present, PRESENT]].
```

reads out the desired sequence.

3.3.2. *Reading a linked list three-record content.*

To read the content of the three-record pointed to by the pointer HERE, we must read the b-word pointed to by the three-record first pointer and then the symbol sequence pointed to by its second pointer, in the order. To this end, let us consider the following reading routine, with the calling sequence READ[HERE]:

```
READ [HERE]
 Output [FP[HERE]]
 if SP[HERE] points either to a monorecord or to a birecord
       then Output [SP[HERE]]
       else READ [SP[HERE]]
END
```

3.3.3. *Reading a key.*

Because both in the three-records and in the six-records the first two pointers point to the sequence of symbols representing the key, we do not have to distinguish the two cases to discuss their reading accesses. If HERE is a pointer either to a three-record or to a six-record, belonging to a key structure, to read the symbol sequence representing the key it is sufficient to call the routine READ[HERE].

3.3.4. *Reading an item.*

For the moment being we consider the case of an item characterized by the key KEY: we shall discuss in the chapter 3.3.6. the case of two items with, up to K – 1, equal keys.

To read an item from a DLDS we need the counter C, ie, a store for an integer no smaller than K, the number of the keys per item. The calling sequence for the reading routine is READitem[KEY], where KEY is the pointer to the six-, or three-, record holding one of the K keys of the item we want to access.

```
READitem [KEY]
 C = 0
 READ[KEY]
 C = C + 1
 if C < K
       then READitem[TP[KEY]]
       else C = 0
END
```

3.3.5. *The cost to retrieve and to read an item.*

As a search algorithm, either the binomial search algorithm in the key vector case or the red-black tree search algorithm, consents to retrieve a record, or the lack of it, in the order of $\log_2 H$ steps, to determine the presence of a record holding a given key we must read the content of a number of records proportional to $\log_2 H$. To read the content of a record holding a symbol sequence representing a key, we have to follow the pointers from the record down to the third linked list, or to the first two vectors. If the key k-th is represented by a sequence of N_k symbols, with $MAX^*[n_k - 1] < N_k \leq MAX^* n_k$, for MAX definition see the chapter 3. 2. 3. 1., then, to read the k-th key, once it has been retrieved, we have to follow at the most $2n_k$ pointers. Accordingly, to retrieve and read the content of the record holding the k-th key we must follow a number of pointers proportional to $2n_k \log_2 H$. And, if each item is composed by K keys, to retrieve and read an item starting from its k'-th key we must follow in the order of K' pointers, with

$$K' = \sum_{0 < k \neq k' \leq K} 2n_k + 2n_{k'} \log_2 H.$$

3.3.6. *Retrieving and reading the conjunction and the disjunction of keys.*

Let a DLDS be defined by the key structures `Key1`, `Key2`, ..., `Key[k]`, `Key[k + 1]`, ..., `KeyK`, sorted in the circular order. As long as the k-th keys are all different from one another, it is immediate to sort them in the k-th structure, eg by their lexicographic order. However, if there are two or more equal k-th keys, there is the problem to insert them in the structure `Key[k]`. To ease their retrieval, we sort the equal keys by preserving into the next structure, in the key circular order, the present order. That is, let us consider a DLDS holding the item A represented, as sequences of symbols, by the keys: KEY1, KEY2, ..., KEY[k], KEY[k + 1], ..., KEYK, respectively, held in the key structures `Key1`, `Key2`, ..., `Key[k]`, `Key[k + 1]`, ..., `KeyK`. We want to insert in the said DLDS the item B represented, as sequences of symbols, by the keys: KEY1, KEY2, ..., KEY[k], KEY[k + 1], ..., KEyK, to be inserted, respectively, in the key structures `Key1`, `Key2`, ..., `Key[k]`, `Key[k + 1]`, ..., `KeyK`. That is, the items A and B are represented by K – 1 identical sequences of symbols, their K – 1 first keys, and they are characterized by their K-th sequence of symbols, their K-th key, only. In the case of the item A and of the item B, to insert KEY[k + 1] in a three-record we must set its first, and its second, pointer with the same value for both A and B. Then, we must set their third pointers so that they permit us to arrive to the `KEYK` and `KEyK` values, respectively. Apparently, the order in which we insert the key KEY[k + 1] two instances in the key structure `Key[k + 1]` is indifferent, as both possible solutions give the same result. We choose to insert KEY[k + 1] two instances, see the chapter 3. 4. 5., with the same order as we have inserted KEY[k] two instances. As A and B must differ in at the least one key, otherwise they are one and the same item, the rule consent us to sort all the keys in all the key structures.

Thanks to the just introduced internal order for the key structures, we can implement very simple rules to retrieve items characterized by conjunctions or disjunctions of keys or set of items characterized by keys belonging to given intervals of values. As in the chapter 3. 3. 3., 3. 3. 4., we have presented the algorithms to read a key, an item, the problem now is to search for the conjunctions, disjunctions and logical negation of keys or for the keys belonging to given sets of values.

3.3.6.1. *The conjunction case and its cost.*

To discuss the retrieval of the items characterized by the conjunctions of keys, let K1, K2, be a key pointed by the pointers held in the record `key1`, `key2`, belonging to the key structure `Key1`, `Key2`. Let `prev[key1]`, `prev[key2]`, and `next[key1]`, `next[key2]`, be the record preceding and following `key1`, `key2`, in the key structure `Key1`, `Key2`. That is, even if do not know whether `key1`, `key2`, is represented by the very same symbol sequence representing either `prev[key1]`, `prev[key2]`, and/or `next[key1]`, `next[key2]`, we do know their order.

We call `RETRIEVEitem[key1 & key2]` the subroutine to retrieve the item characterized by the conjunction of the keys, ie, the sequence of symbols pointed to by the first two pointers held in the records `key1` and `key2`. To fix the ideas, let us apply the search procedure to the key K1 in the structure `Key1`. If the search procedure terminates with an unsuccess, as obviously there is no item having K1 as its key, the procedure terminates. Otherwise, the search procedure terminates on the record `key1`.

`RETRIEVEitem[key1 & key2]`
First, set `TEMP = key1`;
second, `READitem[TEMP]`;
third, if `CONT[key2]` = K2 then store the item read in the second step;
third, otherwise ignore the item read in the second step;
fourth, set `TEMP = prev[TEMP]`;
fifth, if `key1 = TEMP` then go to second;
sixth, set `TEMP = next[key1]`;
seventh, `READitem[TEMP]`;
eighth, if `CONT[key2]` = K2 then store the item read in the seventh step;
eighth, otherwise discard the item read in the seventh step;
ninth, set `TEMP = next[TEMP]`;
tenth, if `key1 = TEMP` then go to seventh;
eleventh, `END` the algorithm.

Then the searched for items can be read from the store. As the procedure `RETRIEVEitem[key1 & key2]` retrieves the set of items characterized by a conjunction of two keys, we can employ it to retrieve the items comprised in a given interval, as they are in a finite number.

The cost to retrieve and to read an item characterized by the conjunction of two keys depends on the number E1 of equal keys we must read. To retrieve and read the first item we must follow in the order of K' pointers, see the chapter 3. 3. 5..Then, to read a previously retrieved item, in the present case having one of its keys equal to one of the keys of the previously read item, we must follow in the order of

$$K'' = \sum_{0 < k \leq K} 2n_k$$

pointers, as the item does not have to be retrieved. Moreover, every time we read an item we must adjourn the pointer `TEMP`. Accordingly, to retrieve and read an item characterized by the conjunction of two given keys we must follow in the order of $E1^*K'' + K'$ pointers and set E1 times the pointer `TEMP`. Eventually, the cost to retrieve and read an item characterized by the conjunctions of more than two given keys depends on the number of connectives defining the item, ie, it is the sum of the costs of retrieving and reading the single conjunction of keys.

3.3.6.2. *The disjunction case and its cost.*

To discuss the retrieval of the items characterized by the disjunctions of keys, let K1, K2, be a key pointed by the pointers held in the record key1, key2, belonging to the key structure Key1, Key2. Let prev[key1], prev[key2], and next[key1], next[key2], be the record preceding and following key1, key2, in the key structure Key1, Key2. That is, even if do not know whether key1, key2, is represented by the very same symbol sequence representing either prev[key1], prev[key2], and/or next[key1], next[key2], we do know their order.

We call RETRIEVEitem[key1 v key2] the subroutine to retrieve the item characterized by the disjunction of the keys, ie, the sequence of symbols pointed to by the first two pointers held in the records key1 and key2. First, we apply the search procedure to the key K1 in the structure Key1. The search procedure terminates either with a success, ie, on the record key1 or with an unsuccess. Second, we apply the search procedure to the key K2 in the structure Key2. Again, the search procedure terminates either on the record key2 or with an unsuccess. If both search procedures terminate with an unsuccess, then, there are no items with the keys K1 or K2.

RETRIEVEitem[key1 v key2]
First, set TEMP = key1;
second, READitem[TEMP];
third, if CONT[key2] = K2 store the just read item in the BUFFER store;
fourth, set TEMP = prev[TEMP];
fifth, if key1 = TEMP then go to the second step;
sixth, set TEMP = next[key1];
seventh, READitem[TEMP];
eighth, if CONT[key2] = K2 store the just read item;
ninth, set TEMP = next[TEMP];
tenth, if key1 = TEMP then go to the seventh step;
eleventh, set TEMP = key2;
twelfth, READitem[TEMP];
thirteenth, if [[CONT[key2] = K2] & [the BUFFER does not contain the just read item]] store the just read item in the BUFFER store;
fourteenth, set TEMP = prev[TEMP];
fifteenth, if key2 = TEMP then go to the twelfth step;
sixteenth, set TEMP = next[key2];
seventeenth, READitem[TEMP];
eighteenth, if [[CONT[key2] = K2] & [the BUFFER store does not contain the just read item]] then store the just read item in the BUFFER store;
nineteenth, set TEMP = next[TEMP];
twentieth, if key2 = TEMP then go to the seventeenth step;
twenty-first, END the algorithm.

Then the searched for items can be read from the BUFFER store.

The cost to retrieve and to read an item characterized by the given disjunction of two keys depends on the number E1 and E2 of equal keys we must read. To read the first item we must follow in the order of K' pointers. Then, we must follow K" pointers to read the items sharing one of the keys with the previously read item, as they do not have to be retrieved. Moreover, every time we read an item we must adjourn the pointer TEMP. Accordingly, to retrieve and read an item characterized by the disjunction of two given keys we must follow in the order of [E1 + E2]*K" + K' pointers and set E1 + E2 times the pointer TEMP. Eventually, the cost to retrieve and read an item characterized by the disjunctions of more than two given keys depends on the number of connectives defining the item, ie, it is the sum of the costs of retrieving and reading the single disjunction of keys.

3.3.6.3. *The logical negation case and its cost.*

To discuss the retrieval of the items characterized by the logical negation of keys, we consider a DLDS with the key structure `Key1` holding the record `key1` and H being the number of the records held by `Key1`.

The items characterized by the logical negation of the key K1 are those represented by the symbol sequences pointed to by the first pointers held by `Key1` records, in turn, holding keys different from K1.

We call `RETRIEVEitem[¬K1]` the subroutine to retrieve the item characterized by the logical negation of the key which should have to be represented by the first two pointers held in the record `key1` belonging to the structure `Key1`. The procedure results in the set of items which do have a key pointed to by the pointers held by `Key1` records and pointing to a symbol sequence different from K1. Let `TEMP` be a temporary pointer, `FirstRecord` be `Key1` first record `NEXT[x]` be the `Key1` record next to `x` and `UNDERFLOW` the subroutine called whenever we reach the `Key1` last record third pointer, see also the DLDS writing access discussed in the chapter 3. 4. 1..

`RETRIEVEitem[¬K1]`
First, set `TEMP = FF[Key1, FirstRecord];`
second, `READ[TEMP];`
third, if the just read key is different from K1, then store the item;
fourth, set `TEMP = NEXT[TEMP];`
fifth, if `TP[TEMP]` ≠ `UNDERFLOW` then go to the second step;
sixth, `END` the algorithm.

Then the searched for items can be read from the store.

The cost to retrieve and to read an item characterized by the logical negation of one key depends on the number N1 of items having a key different from K1, where K1 is represented by a sequence of n_{K1} symbols. To read a key we must follow in the order of $2n_k$ pointers, see the chapter 3.3.5.: in our case $2n_{K1}$ pointers. To read a previously retrieved item we must follow in the order of K" pointers, see the chapter 3. 3. 6. 1.. Moreover, every time we read a key, or an item, we must adjourn the pointer `TEMP`. Accordingly, to retrieve and read an item characterized by the logical negation of one given key, we must follow in the order of N1*K" pointers, namely those pointing to the keys different from K1, plus in the order of (H – N1)*$2n_{K1}$ pointers, namely those pointing to the keys equal to K1, and set H times the pointer `TIME`. Eventually, the cost to retrieve and read an item characterized by the negations of two or more given keys depends on the number of connectives defining the item, ie, it is the sum of the costs of retrieving and reading the negation of a single key.

3.4. The writing access.

We start with an empty DLDS, ie, we start with one first vector and one second vector. In the former each component holds one of the symbols belonging to the keyboard alphabet and it has as many components as the alphabet cardinality. In the second vector each component holds one 2-word and the number of its components is the square of the alphabet cardinality.

As we want to store, in the DLDS, H items each composed by K keys, we need K key structures with H components each, ie, either K key vectors with H components or K key red-black trees with H nodes. Initially, each component holds a pointers to EOJ, the End Of Job routine. Let us call `KEY` the generic key structure and `PREV[KEY]`, `NEXT[KEY]`, the key structure preceding, following, `KEY` in the keys circular order.

3.4.1. *Inserting the first item in an Empty DLDS.*

To discuss the first item insertion, let us call PREV[generic], generic and NEXT[generic] three of the first item K keys following each other in the circular order. We have to insert generic, NEXT[generic], ..., PREV[generic], in the key structures KEY, NEXT[KEY], ..., PREV[KEY], respectively. Being the first item to insert, the choice of the key vector three-record in which to insert the first key is quite arbitrary. In the key vector case the choice depends on the order on which we want to sort each key structure. In the red-black tree case the insertion algorithm solves the said ambiguity. Given a record belonging to the key structure KEY, NEXT[KEY], ..., PREV[KEY], ie, given the record having as first fields, respectively, FF[KEY, Generic], FF[NEXT[KEY], NEXT[Generic]], ..., FF[PREV[KEY], PREV[Generic]], we have to set its three pointers. With the notation:

```
VAL[x] - a
```

we mean that the value of the pointer x is set to a, instead with the notation:

```
VAL[x] ≡ a
```

we mean that the value of the pointer x is a.

First, we set the third pointers:

```
VAL[TP[KEY, Generic]] - FF[NEXT[KEY], NEXT[Generic]]]
VAL[TP[NEXT[KEY], NEXT[Generic]]] -
                FF[NEXT[NEXT[KEY], NEXT[NEXT[Generic]]]]
```

and go on putting NEXT[KEY] in place of KEY and NEXT[Generic] in place of Generic, until we set:

```
VAL[TP[PREV[KEY], PREV[Generic]]] - FF[KEY, Generic]
```

ie, until we have set all the K third pointers.

Second, to set the key records first two pointers, let us call N the longest key length. Accordingly, we have to set up $\log_2 N$ linked lists, containing at the least one three-record each. That is, whereas we set to EOJ the first and the second pointer of $\log_2 N$ three-records, we set each of their third pointers either to the linked list next three-record first field address or to the UNDERFLOW routine, whichever is the case.

Third, we set the third linked list first birecord first field content to gen:

```
CONT[FP[three, gen]] - gen
```

Fourth, in the structure KEY we set the record Generic first pointer to the just set birecord holding the sequence gen and belonging to the third linked list:

```
VAL[FP[KEY, Generic]] - FF[three, gen]
```

Fifth, we set the third linked list second birecord first field content to eri:

```
CONT[FP[three, eri]] - eri
```

Sixth, we set the fourth linked list first three-record first pointer to the third linked list second birecord first field:

```
VAL[FP[four, eric]] = FF[three, eri]
```

where `FP[four, eric]` is the first field of a three-record from the `AVAILABLE` list.

Seventh, we set the fourth linked list first three-record second pointer to the first vector monorecord holding the symbol c:

```
VAL[SP[four, eric]] = FF[one, c]
```

Eight, in the key structure `KEY`, we set `Generic` second pointer to the three-record of the fourth linked list holding the pointer to the sequence eric:

```
VAL[SP[KEY, Generic]] = FF[fourth, eric]
```

Once we have set the three-record `Generic` firsts two pointers, we can set the firsts two pointers of the three-records `NEXT[Generic]`, `NEXT[NEXT[Generic]]`, and so on until we have set the first two pointers of the three-record `PREV[Generic]`.

To conclude, to store in a DLDS the first key of the first item we have to insert the two symbol sequences gen and eri in the third linked list and to set four pointers to represent the sequence generic first occurrence. However, from its second occurrence onward we have to set just two pointers per each new key.

3.4.1.1. *The cost of inserting the first item in an empty DLDS.*

Let us consider an item composed by K keys, where the key k is represented by an m_k-word, with $\lceil n_k - 1 \rceil MAX < m_k \leq n_k MAX$. To insert the item into a DLDS, once we have set up its first and second vector, the linked lists and the K key structures, we must set the 2K pointers held by the key structures, plus the $P \leq \Sigma_{0 < k \leq K} 2\lceil n_k - 1 \rceil$ pointers, held by the linked lists three-records, if any.

3.4.2. *Inserting an item in a non-empty DLDS.*

Let us consider a non-empty DLDS, ie, a DLDS composed by the first vector, the second vector, some linked lists and the key structures, as discussed in the chapter 3.4.1..

We want to insert in the DLDS a new item represented by the K keys: newitem, next[newitem], ..., prev[newitem]. That is, we want to insert newitem in the record `Newitem`, belonging to the key structure `KEY`, next[newitem] in the record `Next[Newitem]`, belonging to the key structure `NEXT[KEY]`, ..., prev[newitem] in the record `Prev[Newitem]`, belonging to the key structure `PREV[KEY]`. To do that, we must actually retrieve as many empty records.

That is, we must apply a searching algorithm to each key structure. Eg, we can apply the standard search algorithm, see [13, pag.265], to each red-black tree or the binary search algorithm, see [12, pag. 411] to each key vector, whichever is applicable. We discuss the successful case, ie, if the searching procedure terminates on the empty record Newitem, in the chapter 3.4.2.1.. Otherwise, ie, if the searching procedure terminates unsuccessfully, then, we must apply the procedure discussed in the chapter 3.4.2.2..

If there are different items with equal keys, we must find how to sort the new and the old keys. For the moment being, we discuss the insertion of a new key once found a record to hold it. We shall discuss the case of different items having the same keys in the chapter 4. 5..

3.4.2.1. *The searching procedure terminates on an empty record.*

The search procedure applied to the key structure KEY resulted in the empty record we have called Newitem, ie, we have:

```
VAL[FP[KEY, Newitem]] ≡ EOJ
VAL[SP[KEY, Newitem]] ≡ EOJ
VAL[TP[KEY, Newitem]] ≡ EOJ
```

We start by setting the first two pointers of the three-record Newitem:

first, we set:

```
CONT[FP[three, new]] = new
VAL[FP[KEY, Newitem]] = FF[three, new]
```

where either new is a birecord from the AVAILABLE list, to be set and then inserted into the third linked list, or it is one previously inserted in the latter linked list, in which case we just set the pointer FP[KEY, Newitem];

second, we set:

```
CONT[FP[one, m]] ≡ m
CONT[FP[three, ite]] = ite
VAL[FP[five, ITEM]] = FF[three, ite]
VAL[SP[five, ITEM]] = FF[one, m]
VAL[SP[KEY, Newitem]] = FF[five, ITEM]
```

where m is the first vector monorecord holding the symbol m and ite, ITEM, is either a new birecord, three-record, from the AVAILABLE list, to be set and then inserted into the linked list three, five, or it is one previously inserted in the liked list three, five, in which case we just set the pointers FP[five, ITEM], SP[five, ITEM], SP[KEY, Newitem], VAL[SP[KEY, Newitem]];

To set Newitem third pointer we must find the record Next[Newitem], belonging to the key structure NEXT[KEY]. That is, we must apply the searching procedure to NEXT[KEY]. Once found Next[Newitem],

first, we set:

```
VAL[TP[KEY, Newitem]] = FF[NEXT[KEY], Next[Newitem]]
```

second, we put NEXT[KEY] in place of KEY and Next[Newitem] in place of Newitem, for K - 1 times, ie, until we set:

```
VAL[TP[PREV[KEY], Prev[Newitem]]] = FF[KEY, Newitem]
```

ie, until we have set all the third pointers.

In the case of different items with equal keys, to set Newitem third pointer, we must apply the rules discussed in the chapter 3. 4. 5.

Once we have set the pointers of the three-record Newitem, we set the pointers of the three-records Next[Newitem], belonging to the key vector NEXT[KEY], ..., and so

on substituting Next[Newitem] in place of Newitem and NEXT[KEY] in place of KEY for K times, namely until we have set the first two pointers of the three-record Prev[Newitem], belonging to the key vector PREV[KEY].

Mutatis mutandis, the same considerations apply to the sequence newitem as those given at the end of the chapter 3.4.1.1. for the sequences generic.

3.4.2.2. *The searching procedure terminates on a non-empty record.*

If the DLDS is organized with key red-black trees, we call the insertion algorithm for red-black trees, see [13], and, once the said algorithm stops, a new, empty, node, has been inserted and we can apply the algorithm discussed in the chapter 3.4.2.1.. Otherwise, namely if the DLDS is organized with key vectors, we apply the following algorithm.

The search procedure applied to the key vector KEY results in the three-record Occupied, ie, we have:

CONT[FP[KEY, Occupied]] ≠ EOJ
CONT[SP[KEY, Occupied]] ≠ EOJ

For simplicity sake, we say Occupied, ie, the key vector KEY j-th three-record first pointer points to a sequence preceding newitem in the KEY vector order. In the contrary case, namely if Occupied first pointer points to a sequence following newitem in the KEY order, it is sufficient to consider the (j - 1)-th three-record of the key vector KEY instead of Occupied.

To have an empty three-record to store newitem, let T be an auxiliary bi-component vector and Top and Bottom its two three-record components.

First, we set:

```
VAL[FP[T, Top]] = VAL[FP[KEY, Occupied]]
VAL[SP[T, Top]] = VAL[SP[KEY, Occupied]]
VAL[TP[T, Top]] = VAL[TP[KEY, Occupied]]
VAL[FP[KEY, Occupied]] = FF[T, Top]
VAL[SP[KEY, Occupied]] = FF[T, Top]
VAL[TP[KEY, Occupied]] = FF[T, Top]
```

second, in VAL[SP[KEY, Occupied]] we put NEXT[KEY] in place of KEY and Next[Occupied] in place of Occupied, for K - 1 times, ie, until we have:

VAL[TP[PREV[KEY], Prev[Occupied]]] ≡ FF[KEY, Occupied]

so we can set:

VAL[TP[PREV[KEY], Prev[Occupied]]] = FF[T, Top]

third, we apply to the empty three-record Bottom the algorithm discussed in the chapter 4. 2. 1.. However, to set Newitem third pointer we must find the three-record Next[Newitem], belonging to the vector NEXT[KEY], ie, we must apply the searching procedure to NEXT[KEY]. If the latter searching procedure results in a non-empty three-record we apply to the vector NEXT[KEY] the algorithm discussed in the present chapter.

In the case of different items with equal keys, ie, if `Top` and `Bottom` end with the same firsts and seconds pointers, to set `Bottom` third pointer, it is convenient to apply the results discussed in the chapter 3.4.5.

It is worth to note that the names `T`, `Top` and `Bottom` can be reused again and again as the auxiliary bi-component vector and the two three-records now are identified by the just set pointers.

If at the least one auxiliary `T` vector is pointed to by the pointers held by a three-record belonging to a key vector, we call the latter a *dirty*, key, vector.

3.4.2.3. *The cost of inserting an item in a non-empty DLDS.*

Let us consider a Newitem composed by K keys, with the key k represented by an m_k-word, where $\lceil n_k \cdot UMAX < m_k \leq n_k MAX$, to insert in a DLDS holding H items.

To find the K records which have to hold the Newitem K keys, we have to apply K times a searching procedure to the K sorted key structures, each time yielding a result in the order of $\log_2 H$ steps. That is, to find where to insert all the Newitem K keys, we have to follow in the order of $\Sigma_{0<k\leq K} 2n_k \log_2 H$ pointers, see the chapter 3.3.5..

Once we have found the K records, we have to insert in each of them one of the Newitem K keys. Depending on whether or not the searching procedure has found empty records and the DLDS holds pointers to the symbol subsequences representing the current keys, we can have very different access times. Accordingly, we discuss the most favorable and the most unfavorable case.

In any case, we must add the cost to find the proper position for the record if the item to insert has some keys equal to some of the previously inserted keys, see the chapter 3.4.5..

3.4.2.3.1. *The most favorable case.*

The search procedure applied to the key structure has found both the empty records and the three-records holding pointers to the subsequences present in the Newitem K keys: let m_k be the k-th key length, with $\lceil n_k \cdot UMAX < m_k \leq n_k MAX$.

To insert the Newitem keys, first we must set the 2K pointers belonging to the key structures.

To read the sequence of symbols pointed to by the pointers held in a three-record belonging to the m_k-th linked list, we must follow $2n_k$ pointers, see the chapter 3.3.5.. If

$$n_{max} = \max_{0<k\leq K} \{n_k\},$$

the total number of symbols for key, or TNSK, is no greater than $MAX^* n_{max}$, ie:

$$TNSK < MAX^* n_{max}.$$

In turn, the total number of symbols for item, TNSI, is no greater than $K^* MAX^* n_{max}$, ie:

$$TNSI < K^* TNSK.$$

As H is the number of items stored into the DLDS, the total number of symbols, or TNS, stored into the DLDS is no greater than $H^* TNSI$, ie:

$$TNS < H^* TNSI.$$

At the most, we have to retrive a 3-word in a linked list containing no more than TNS/3 three-records, a 4-word in a linked list containing no more than TNS/4 three-records, ..., a TNSK-word in a linked list containing no more than TNS/TNSK three-records.

Accordingly, to find the three-records holding the pointers to the sequence of symbols representing one of Newitem K keys, on average, we must follow and test the symbol sequences pointed to by less than $2^*\Sigma_{3 \le tnsk \le TNSK}$ TNS/tnsk pointers. That is, to insert one item in a non-empty DLDS, we must follow $2^*K^*\Sigma_{3 \le tnsk \le TNSK}$ TNS/tnsk pointers, in the most favorable case. Plus the $\Sigma_{0 < k \le K} 2n_k \log_2 H$ pointers discussed in the chapter 4. 2. 3..

To conclude, in the most favorable case, we must follow less than:

$$2K^*\Sigma_{3 \le tnsk \le TNSK} \text{TNS/tnsk} + \Sigma_{0 < k \le K} 2n_k \log_2 H$$

pointers and set 3^*K pointers.

3.4.2.3.2. *The most unfavorable case.*

Let us consider the most unfavorable case, ie, we have found no empty three-records at the right places in the key structures and no intermediate three-records holding pointers to the subsequences of symbols representing the keys. Then, first, we need one T vector for key, second, we must insert all the intermediate records holding the pointers to the symbol sequences representing the K keys and having the length of $\Sigma_{0 < k \le K} m_k$ symbols, third, we must set all the latter pointers.

To insert one Newitem key in one key vector, we have to set the seven pointers discussed in the chapter 3.4.2.2.. That is, to insert all the Newitem K keys in the K key vectors, we have to set 7K pointers and we have to follow the $\Sigma_{0 < k \le K} 2n_k \log_2 H$ pointers discussed in the chapter 4. 2. 3..

Now in each sorted linked list we must insert and set one three-record. To find where to insert the new three-records, for each key, we must follow and test the symbol sequences pointed to by less than $2^*\Sigma_{3 \le tnsk \le TNSK}$ TNS/tnsk pointers. That is, to find where to insert the new three-records, we must follow $2^*K^*\Sigma_{3 \le tnsk \le TNSK}$ TNS/tnsk pointers. To set the new three-record pointers, we must set four pointer for each three-record, ie, we must set, for each three-record, the first and the second pointer, plus one old and one new third pointer. As there are up to $m_k - 3$ linked lists holding pointers, we must set at the most $4(m_k - 3)$ pointers.

In all, to insert a new item in a DLDS, we must follow less than:

$$2^*K^*\Sigma_{3 \le tnsk \le TNSK} \text{TNS/tnsk} + \Sigma_{0 < k \le K} 2n_k \log_2 H$$

pointers and we must set no more than $7^*K + 4(m_k - 3)$ pointers.

3.4.3. *Reordering a dirty key vector.*

Obviously, in the present, and in the following chapters up to the 4.4., the DLDS is presumed to be organized with key vectors, as the red-black trees cannot give rise to dirty key structures.

To *reorder* a dirty vector, see the chapter 4.2.2., ie, to eliminate all the T vectors pointed to by its three-records, we need to merge the dirty and the T vectors content into a new vector.

To fix the ideas, let us consider a DLDS containing the H items ITEM[1],..., ITEM[h], ...ITEM[H], the h-th being composed by the K keys KEY[K][1][h],...,

KEY[K[k]][h], ..., KEY[K[K]][h]. That is, we are considering the K key vectors K[1], ..., K[k], ..., K[K], where FP[K[k], j] is the name of the first pointer held by the j-th component of the vector K[k], namely FP[K[k], j] points to the sequence of symbols representing the key KEY[K[k]][j].

If we apply the search algorithm to the key vector K[k] pointing to a number of T vectors, we must take one or another train of actions depending on the current three-record content. That is, whether or not it holds three pointers with the same value. Moreover, for each T vector there are four extra pointers. So to avoid to degrade progressively the searching procedure efficiency and to control the DLDS storage space requirements, it is convenient to eliminate the T vectors as soon as possible. To this end, we must merge, into a new vector NK, both the key vector K[k] and the T vectors it is pointing to. And we must adjourn the second field contents of the three-records belonging to the key vector K[k - 1], ie, to the key vector preceding the present one in the keys circular order. Initially, NK pointers are set to EOJ.

Because the key vectors are sorted one independently of the others, from KEY[K[k]][h] precedes KEY[K[k]][h + 1] we cannot conclude that KEY[K[k + 1]][h] precedes KEY[K[k + 1]][h + 1], on the contrary, it could be that KEY[K[k + 1]][h] follows KEY[K[k + 1]][h + 1]. Accordingly, to save time, we must interleaves the merge and the adjournment processes.

We call FP[K[k], h[K[k]]], SP[K[k], h[K[k]]], TP[K[k], h[K[k]]], the first, second, third, current pointer held by the key vector K[k] current three-record h[K[k]]. We call FP[NK, h[NK]], SP[NK, h[NK]], TP[NK, h[NK]], the first, second, third, pointer held by the new vector NK current three-record h[NK]. Actually, FP[NK, h[NK]] and SP[NK, h[NK]] are the pointers to the sequence of symbols representing the key KEYNK[h[NK]]. Accordingly, with the notation FF[PREV[NK], PREV[h[NK]]] we represent the address of the first field of the three-record holding the pointer to the sequence of symbols representing the key preceding KEYNK[h[NK]], in the keys circular order.

To reorder the key vector K[k]:

first, we set C = 0;

second, we test the vector K[k] current three-record content to see whether it is holding three pointers having different values, ie, if VAL[FP[K[k], h[K[k]]]] ≠ VAL[SP[K[k], h[K[k]]]] ≠ VAL[TP[K[k], h[K[k]]]];

third, we set NK current three-record h[NK] first, second, third, pointer equal to the K[k] current three-record h[K[k]] first, second, third, pointer, namely we set:

```
VAL[FP[NK, h[NK]]] = VAL[FP[K[k], h[K[k]]]]
VAL[SP[NK, h[NK]]] = VAL[SP[K[k], h[K[k]]]]
VAL[TP[NK, h[NK]]] = VAL[TP[K[k], h[K[k]]]]
```

COMMENT Now we are going to find and adjourn TP[K[k - 1], h[K[k - 1]]], as its present value is FF[K[k], h[K[k]]] and it has to become FF[NK, h[NK]]. **COMMENT END**;

fourth, we nest K – 1 times NEXT[... NEXT[TP[K[k], h[K[k]]]] ...] so to obtain the third pointer TP[K[k - 1], h[K[k - 1]]], at present pointing to FF[K[k], h[K[k]]], that is, because at present we have:

```
NEXT[ ... NEXT[TP[K[k], h[K[k]]]] ... ] ≡ TP[K[k - 1], h[K[k - 1]]]
```

we set:

```
VAL[NEXT[ ... NEXT[TP[K[k], h[K[k]]]] ... ]] ≡
                        VAL[TP[K[k - 1], h[K[k - 1]]]] =
                                              FF[NK, h[NK]]
```

fifth, we must adjourn the current three-record, ie, `FF[K[k], h[K[k]] + 1]`, `FF[NK, h[NK] + 1]`, becomes the key vector `K[k]`, `NK`, current three-record;

sixth, we set C = C + 1 and test whether or not C is greater than H;

seventh, if C < H we go back to the second step, otherwise we return `K[k]` to the `AvailableVector` list, then `END` the algorithm;

second, otherwise, ie, the vector `K[k]` current three-record holds three pointers with the same value, ie, we have:

```
VAL[FP[K[k], h[K[k]]]] ≡ VAL[SP[K[k], h[K[k]]]] ≡
                        VAL[TP[K[k], h[K[k]]]] ≡ CommonValue
```

third, we test whether CommonValue is EOJ;

fourth, we copy EOJ into the new vector current three-record first and second fields, ie, we set:

```
VAL[FP[NK, h[NK]]] = EOJ
VAL[SP[NK, h[NK]]] = EOJ
VAL[TP[NK, h[NK]]] = EOJ
```

fifth, we consider the vector `K[k]`, `NK`, next three-record `FF[K[k], h[K[k]] + 1]`, `FF[NK, h[NK] + 1]`, as the current three-record;

sixth, we set C = C + 1 and test whether or not C is greater than H;

seventh, if C < H we go back to the second step, otherwise we return `K[k]` to the `AvailableVector` list, then `END` the algorithm;

third, otherwise, ie, if CommonValue is different from EOJ, then, it is assumed to be the address of `Top`, the temporary vector `T` first three-record;

fourth, we set:

```
VAL[FP[NK, h[NK]]] = VAL[FP[T, Top]]
VAL[SP[NK, h[NK]]] = VAL[SP[T, Top]]
VAL[TP[NK, h[NK]]] = VAL[TP[T, Top]]
VAL[FP[NK, h[NK] + 1]] = VAL[FP[T, Bottom]]
VAL[SP[NK, h[NK] + 1]] = VAL[SP[T, Bottom]]
VAL[TP[NK, h[NK] + 1]] = VAL[TP[T, Bottom]]
```

fifth, in `NEXT[TP[NK, h[NK]]]`, we put `NEXT[TP[NK, h[NK]]]` in place of `TP[NK, h[NK]]`, for K – 1 times, ie, until we have

```
NEXT[ ... NEXT[TP[NK], h[NK]]] ... ] ≡ TP[PREV[NK], PREV[h[NK]]]
```

and we can set:

```
VAL[NEXT[ ... NEXT[SP[NK, h[NK]]] ... ]] ≡
VAL[SP[PREV[NK], PREV[h[NK]]]] = FF[NK, h[NK]]
```

sixth, in `NEXT[SP[NK, h[NK] + 1]]`, we put `NEXT[SP[NK, h[NK] + 1]]` in place of `SP[NK, h[NK] + 1]`, for K – 1 times, ie, until we have

```
NEXT[ ... NEXT[SP[NK, h[NK] + 1]] ... ] ≡
                SP[PREV[NK], PREV[h[NK] + 1]]
```

and we can set:

```
VAL[NEXT[ ... NEXT[TP[NK, h[NK] + 1]] ... ]] ≡
                VAL[TP[PREV[NK], PREV[h[NK] + 1]]] =
                                    FF[NK, h[NK] + 1]]
```

seventh, we return to the AvailableVector list the temporary vector T;

eight, we consider the vector K[k], NK, next three-record FF[K[k], h[K[k]] + 1], FF[NK, h[NK] + 2], as the current three-record;

ninth, we set C = C + 1 and test whether or not C is greater than H;

tenth, if C < H we go back to the second step, otherwise we return K[k] to the AvailableVector list, then END the algorithm.

3.4.3.1. *The cost of reordering a dirty key vector.*

If the DLDS holds H items, to reorder one dirty key vector we need the new vector NK holding H three-records, ie, we have to set 3H pointers. However, we have to adjourn also the third pointers held by the previous three-records, ie, by the three-records belonging to the key vector preceding the present one, in the key circular order. To set the latter third pointers we have to follow K pointers per item: eventually, to copy the old into the NK vector we have to follow H*K and to set 4H pointers.

3.4.4. *The reordering of a DLDS and its cost.*

To reorder a DLDS with more than one dirty key vector, we have to copy each dirty vector in a new vector. In other words, in the algorithm given in the chapter 3.4.3. we must insert a zeroth step:

zeroth, we set k = 1;

and the tenth and both the seventh steps holding the otherwise clause terminating with the "then END the algorithm" become:

if C < H we go back to the second step, otherwise we return K[k] to the AvailableVector list, then, we set k = k + 1 and if k ≤ K we set h[k] = 1 and go back to the first step, otherwise END the algorithm.

Whereas we must rearrange the traditional data structures, eg rebalance the balanced trees, as soon as a new item insertion requires it, we do not need to rearrange the DLDS organized with the key vectors every time we introduce a T vector. Actually, we can batch all the DLDS reordering accesses together and execute them at a later time. In any case, as the DLDS is a multisorted data structure, to reorder it we must set a number of pointers no smaller than two times the number of each item keys.

In the most unfavorable case to reorder a DLDS we must reorder all its K key vectors. That is, we must set 4H*K and follow H*K^2 pointers.

3.4.5. *The case of equal keys.*

Let us consider a DLDS holding the item A represented as sequences of symbols by the keys: KEY[1], KEY[2], ..., KeY[k], KEY[k + 1], ..., KEY[K] pointed to, respectively, by the first two pointers stored in the records `key[1]`, `key[2]`, ..., `key[k]`, `key[k + 1]`, ..., `key[K]` in turn, held in the key structures `Key[1]`, `Key[2]`, ..., `Key[k]`, `Key[k + 1]`, ..., `Key[K]`. We want to insert in the said DLDS the item B represented as sequences of symbols by the keys: KEY[1], KEY[2], ..., kEy[k], KEY[k + 1], ..., Key[K] to be pointed to, respectively, by the first two pointers stored in the records `NEWkey[1]`, `NEWkey[2]`, ..., `NEWkey[k]`, `NEWkey[k + 1]`, ..., `NEWkey[K]` in turn, to be inserted, respectively, in the key structures `Key[1]`, `Key[2]`, ..., `Key[k]`, `Key[k + 1]`, ..., `Key[K]`. As the two items have K – 2 identical keys, they are characterized by their k-th and K-th keys, we must decide the order in which to insert each key in the appropriate record. As a matter of fact, two items must differ at the least in one key: two items with identical keys are one and the same item.

In general, it is more efficient to insert the key in accordance with their lexicographic order, however, in the equal keys case the rule does not tell us nothing. Accordingly, see also the chapter 3.6., we insert two equal keys with the same order of the two keys preceding them, in the keys circular order. That is, in the [k + 1]-th key structure, two equal keys are inserted with the order the keys preceding them, in the key circular order, have been inserted in the k-th key structure. Actually, instead of the key structure preceding the present one, we could have chosen the following one. However, as in the key records the third pointer points to the key record belonging to the next key structure, the chosen solution is simpler to implement.

To fix the ideas, we consider the sequence Key[K] preceding the sequence KEY[K] and the sequence KeY[k] preceding the sequence kEy[k], in the lexicographic order. That is, the key record `NEWkey[K]` has to precede the key record `key[K]` and the key record `key[k]` has to precede the key record `NEWkey[k]`. Accordingly, the key records `key[1]`, `key[2]`, ..., `key[k - 1]` have to follow, respectively, the key records `NEWkey[1]`, `NEWkey[2]`, ..., `NEWkey[k - 1]` and the key records `NEWkey[k + 1]`, ..., `NEWkey[K - 1]` have to follow, respectively, the key records `key[k + 1]`, ..., `key[K - 1]`, in the respective key structures. It is worth to note that to precede or to follow does not mean the records have to be contiguous.

To insert an item we must search for an empty record, as we have discussed in the chapter 4.2.. As we must read the previously inserted keys, it is easy to verify whether or not the latter and the new item to insert have equal keys. If they have equal keys, then, it is convenient to insert the new item starting with one of the keys characterizing the new item as different from the previously inserted item. To this end, let `buf1[k]` and `buf2[k]` be two buffer stores having the capacity to hold K key records each, ie, initially they have to store the searching algorithm last two read key records in the k-th key structure. If in either `buf1[k]` and/or `buf2[k]` at the least one of the first two pointers are equal to the new key to store, ie, if at the least one of them points to the same symbol sequences representing the new key to store, we store in `buf1[k]` and `buf2[k]`, respectively, the records pointed to by the third pointers held in the very same `buf1[k]` and `buf2[k]`. We repeat again and again the same step until both the two buffer stores contain first pointers different between them and pointing to symbol sequences different from those representing the present new key to store. Let `KF` be the key structure holding the records last stored in the buffer stores and containing different first and second pointers. Every time, after the first time, the buffer stores held different first and second pointers and pointing to symbol sequences different from those representing the present new key to store, we apply the searching algorithm to the relative key structure, at present the `KF` structure. It is worth to note that in any case we should have searched the key structures for the empty records to allocate for the new keys to insert: the present procedure save us to search in the case of different items with equal keys. As the search termi-

nates, we start the carousel again saving the pointers held in the two last read key records in buf1[k] and buf2[k], respectively. The algorithm terminates as we have copied in the two buffer stores the 3K pointers held in all the pairs of key records: for each pair, one triplet of pointers preceding and one triplet of pointers following, in the lexicographic order, the new keys to insert.

Let FindPosition[NewItem] be the calling sequence for the procedure to find the records, one for key structure, to allocate to the new item to insert. Let Search[Key[C], KEY[C]] be the calling sequence for the procedure to search the key structure Key[C] for an empty record to allocate to the key KEY[C]. Let FP[LastRead], FP[LastButOne], SP[LastRead], SP[LastButOne] be the last two first and second pointers read by the search procedure to find out whether there is an empty record to hold the key to insert. Let CONT[NEWkey[C]], CONT[NEXT[NEWkey[C]]], represent the content of the first, second, field of the key record NEWkey[C], once the KEY[C] have been stored in. Eventually, let TP[buf1[C]], TP[buf2[C]], represent the third pointer stored in the buf1[C], buf2[C], buffer store

FindPosition[NewItem]
First, set C = 1;
second, call Search[Key[C], KEY[C]];
third, store LastRead in buf1[C] and LastButOne in buf2[C];
fourth, if FP[LastRead] = CONT[NEWkey[C]] v
FP[LastButOne] = CONT[NEWkey[C]] v
SP[LastRead] = CONT[NEXT[NEWkey[C]]] v
SP[LastButOne] = CONT[NEXT[NEWkey[C]]]
then
fourth, first, set C = C + 1;
fourth, second, if C > K then END the algorithm.
fourth, third, store CONT[TP[buf1[C - 1]]] in buf1[C]
and
CONT[TP[buf2[C - 1]]] in buf2[C];
fourth, fourth, go to the fourth step;
otherwise
fourth, first, set C = C + 1;
fourth, second, if C > K then END the algorithm.
fourth, third, go to the second step;

To retrieve the proper position of the record in which to store a new key, in the case of different items having the same keys, we must copy 6K pointers in the buffer stores buf1[k] and buf2[k] and we must call the searching procedure as many times as the item to store has keys different from the previously inserted ones. That is, the said cost is on top to the cost given in the chapter 3.4.2.3..

3.5. The erasing access.

As in an intermediate linked list a three-record can be pointed to by many pointers held in many different records, it is not a viable proposition to erase the intermediate three-records.

Accordingly, we consider a DLDS with the first, the second vectors and K key structures. We call Erasable a pointer to the current three-record first field, ie, FF[Erasable] is the address of the three-record holding the current key of the item to be erased. And we call NEXT[Erasable] the three-record pointed to by TP[Erasable]. In a DLDS organized with red-black tress, we have simply to call the

erasing algorithm, see [131, for every item key and the relative algorithm is discussed in the chapter 3.5.2.. Instead, to erase from a DLDS organized with key vectors the item characterized by the key held in the current three-record, the algorithm is the following.

3.5.1. *Erasing an item from a DLDS organized with vectors.*

To erase the item characterized by the key stored into `FF[Erasable]` belonging to the key vector `KEY`, we need the temporary pointer `TEMP` and the counter C, initially set to zero. The erasing algorithm is the following:

first, if C > K, we set C = 0 and `END` the algorithm;
first, otherwise,
second, if

```
VAL[FP[NEXT[Erasable]]] ≡
      VAL[SP[NEXT[Erasable]]] ≡
            VAL[TP[NEXT[Erasable]]] ≡ EOJ
```

third, we set

C = K + 1;

fourth, we go back to the first step;
second, if

```
VAL[FP[NEXT[Erasable]]] ≡
      VAL[SP[NEXT[Erasable]]] ≡
            VAL[TP[NEXT[Erasable]]] ≡
                  CommonValue ≠ EOJ,
```

ie, we take `CommonValue` as the T vector `Top` three-record address and we set:

```
TEMP = VAL[TP[T, Top]]
VAL[FP[T, Top]] = VAL[SP[T, Top]] = VAL[TP[T, Top]] = EOJ
```

third, to erase the next key, we put `TEMP` in place of `NEXT[Erasable]`;
fourth, we set

C = C + 1;

fifth, we go back to the first step;
second, if

```
VAL[FP[NEXT[Erasable]]] ≠
      VAL[SP[NEXT[Erasable]]],
```

we set

```
TEMP = VAL[TP[NEXT[Erasable]]]
VAL[FP[NEXT[Erasable]]] = EOJ
VAL[SP[NEXT[Erasable]]] = EOJ
```

VAL[TP[NEXT[Erasable]]] - **EOJ**

third, we put TEMP in place of NEXT[Erasable];
fourth, we set

C = C + 1;

fifth, we go back to the first step.

3.5.2. *Erasing an item from a DLDS organized with red-black trees.*

To erase the item characterized by the key stored into FF[Erasable] belonging to the key red-black tree KEY, we need the temporary pointer TEMP and the counter C, initially set to zero. The erasing algorithm is the following:

first, if C > K, we set C = 0 and END the algorithm;
first, otherwise, we set

TEMP = VAL[TP[NEXT[Erasable]]]

second, we apply the red-black trees erasing algorithm to the node NEXT[Erasable] of the red-black tree KEY;
third, we put TEMP in place of NEXT[Erasable];
fourth, we set

C = C + 1;

fifth, we go back to the first step.

3.5.3. *The cost of erasing an item from a DLDS.*

To erase one key from a DLDS organized with vectors, we must set three pointers plus a temporary one, accordingly, to erase an item we must set 4K pointers.

To erase one key from a DLDS organized with red-black trees, we must set one temporary pointer plus the cost of the erasing algorithm for red-black trees. Accordingly, to erase an item from a DLDS organized with red-black trees, we must set K pointers plus K times the cost of the erasing algorithm for red-black trees.

3.6. To insert or to erase keys from all the items.

In the following we are considering a DLDS with the keys held either in red-black trees or in vectors. That is, we do not discuss and ignore the cost to the set up either data structures, instead, we present the algorithm to insert, erase, the new keys in either of them.

3.6.1. *Inserting a new key in all the items.*

For simplicity sake, we consider the new keys sorted, eg according to their lexicographic order, and a one to one correspondence between the items previously inserted into the DLDS and the new keys to insert. In other words we do know which item corresponds

to any given new key, and vice versa. Actually, the one to one correspondence exists between the key which has to precede the new one to insert and the very latter.

Let us call NewKey the current new key we have to insert in the newKey record, namely either a six-record or a three-record, belonging to NEWKEY, the new key structure we want to insert in a DLDS containing H items. We call PREVKEY and NEXTKEY the two key structures, respectively, which have to precede and to follow, in the key circular order, NEWKEY, ie, the latter has to be inserted between the former two. We call new-record, prev-record and next-record the records, the first belonging to NEWKEY, the second to PREVKEY and the third to NEXTKEY, holding the pointers to the symbol sequences representing the keys of the very same item, with the first preceding and the second following NewKey, in the keys circular order. That is, initially TP[PREVKEY, prev-record] points to FF[NEXT, next-record]. Eventually, let Search[PREVKEY, prev-record] be the calling sequence for the procedure to search the key structure PREVKEY for the prev-record.

To insert the new key in the three-record newKey, we have to set, first, FP[NEWKEY, new-record] to the third vector birecord holding the first three symbols of the sequence NewKey, ie, holding the 3-word New; second, SP[NEWKEY, new-record] to the third vector birecord holding the last symbols of the sequence representing NewKey, ie, in our case holding the 3-word Key; third, TP[NEWKEY, new-record] to FF[NEXT, next-record]; fourth, TP[PREVKEY, prev-record] to FF[NEWKEY, new-record]. That is, we have:

first, we set C - 1

second, as the new keys are sorted, we take the C-th record as the new-record. In the red-black tree case we call the insertion algorithm to insert a new six-record. In any case, we call the searching algorithm to retrieve in the PREVKEY and NEXTKEY structures the records to hold the pointers to the keys preceding and following the new current key. That is, we call the search procedures Search[PREVKEY, prev-record] and Search[NEXTKEY, next-record], see also the chapter 4.5.;

third, we set, in the order:

```
VAL[FP[NEWKEY, new-record]] - FF[three, New]
VAL[SP[NEWKEY, new-record]] - FF[three, Key]
VAL[TP[NEWKEY, new-record]] - FF[NEXTKEY, next-record]
VAL[TP[PREVKEY, prev-record]] - FF[NEWKEY, new-record]
```

with

```
CONT[FP[three, New]] - New
CONT[FP[three, Key]] - Key;
```

fourth, we set C - C + 1;

fifth, if C < H we go back to the second step; otherwise, we set C - 0 and END the algorithm.

3.6.2. *The cost of inserting a new key.*

To retrieve the prev-record, the searching procedure yields a result in $\log_2 H$ steps. To retrieve the next-record, the searching procedure yields a result in $\log_2 H$ steps. Then we have to set four pointers, ie, in total we must follow $2\log_2 H$ pointers and set 4H pointers, at the least. To these costs we must add the cost of setting the pointers to the intermediate linked lists, if any: obviously, their number depends on the new keys length. Eventually, to these costs we must add the cost to set up either a vector or a red-black tree.

3.6.3. *Erasing a key from all the items.*

To erase the same type of key from all the items in a DLDS it is sufficient to set the third pointer of the key preceding the one to erase, in the keys circular order, equal to the third pointer of the key to erase. Let us call KEY, PREV[KEY], respectively, the structure holding the key to erase, the one preceding it. Let `Erasable` be the pointer to the current record to erase, and let `Prev[Erasable]`, be the pointer to the record holding the same item preceding key, in the keys circular order. The erasing algorithm is the following:

first, we set C = 1;
second, we consider the C-th record as the current `Erasable` record and search in the `PREV[KEY]` for the record `Prev[Erasable]`;
third, we set, in the order:

```
VAL[FP[Erasable]] = EOJ
VAL[SP[Erasable]] = EOJ
VAL[TP[Prev[Erasable]]] = VAL[TP[Erasable]]
VAL[TP[Erasable]] = EOJ
```

fourth, we set C = C + 1;
fifth, if C < H we go back to the second step; otherwise, we return `KEY` to the `AvailableStructure` list, set C = 0 and `END` the algorithm.

3.6.4. *The Cost of erasing a key.*

In the algorithm second step, to retrieve the record `Prev[Erasable]` in the `PREV[KEY]` structure we have to follow the K third pointers and set 4 pointers, ie, to erase `KEY` we must follow K*H pointers and set 4*H pointers.

3.7. Conclusions.

We have discussed a new way to store the data in some traditional data structures, actually exploiting the redundancy present in the data when represented as symbol sequences.

We have presented two DLDS versions: in both each key is represented by three pointers. Obviously, if in our data there are more single than multiple occurrences of b-words, then, the proposed key representation is an efficient way to store the data. By contrast, if the multiple occurrences of b-words outnumber the single occurrences, then, it is more efficient to represent the keys by means of two pointers. That is, with records holding a second pointer pointing to the next key holding structure and a first pointer pointing to an intermediate three-record holding, in turn, a first pointer pointing to the key first three symbols, a second pointer pointing to the key remaining symbols and a third pointer pointing to the next three-record in the same intermediate structure.

Apart from the storage space saving, confirmed in our tests, on a DLDS we can execute operations the more traditional data structures do not allow. Eg, in a DLDS it is easy to modify the number of keys per item or to retrieve an item by any number of its keys, as the latter are independently sorted and stored and the data are held as pointers.

Chapter 4

THE BNN AND THE HEBBIAN RULE

Up to now we have considered BNNs characterized by constant coupling coefficients, the $C_{h,k,d}$. That is, in our BNN the $C_{h,k,d}$ have been defined as delay dependent and time independent constants. However, a neural network should be programmable by examples, ie, it should be a trainable system. Accordingly, first, we redefine the coupling coefficients as time, as well as delay, dependent quantities. Second, we shall discuss how to assign a Boolean neural net, namely its coupling coefficient matrix, so to obtain a system autonomously adaptable to its unknown and unpredictably evolving environment. Actually, in the following chapters we shall characterize some coupling coefficient matrices containing both time independent and time dependent elements, namely both constant elements and elements satisfying a Hebbian rule, see also [3 pag. 65], so to have autonomously auto-trainable BNNs.

Sometime in the neural network literature the two locutions *Hebbian rule* and *Hebbian learning* are used as each other synonym. For us the locution Hebbian rule denotes an algorithm to compute the value, at the time $t + \tau$, of the time dependent coupling coefficient associated to the d-delayed connection between the two nodes h and k, in function both of the very same time dependent coupling coefficient present value, namely its value at the time t, and of the two nodes state, respectively at the time t and $t - d\tau$. In other words, here the Hebbian rule is the name of an algorithm deterministically dependent on the states of just two nodes: it does depend neither on the entire net nor on the output state. To sum it up, for us the Hebbian rule is a deterministic and locally executed algorithm dependent on local variables having only an implicit and indirect dependence on the learning procedures, eg on the Hebbian learning as defined in [4], or on the environment events. By contrast, in more traditional ANNs the error backpropagation algorithm depends on the net output state and the Hebbian learning has a statistical dependence on the entire net state. Eventually, we are not interested in the Hebbian learning and our Hebbian rules must not be confused with it. Eg, as we shall discuss later on, see the chapter 5. 6., in a neural net there could be different sets of nodes connected by time dependent connections with different Hebbian rules, one for each set. Taking advantage of the Boolean neural net universality, we shall implement, in the chapter 5., some deterministic algorithms to control both the activity of each Hebbian-out node, one independently of the others, and the activity of each Hebbian-in node, one independently of the others, so to govern the activity of each time dependent coupling coefficient and, hence, its value, one independently of the others.

4.1. The facilitatory connection case.

In the literature, e. g. see [3, 4], we find a number of rules to obtain a trainable system. Mainly, not exclusively, it seems as if a neural net purpose is the input patterns classification or recognition thanks to their statistical properties. Different learning rules, as distinct from the various rules used to adjust the net weights, consent to control the nets behavior. Usually, in the neural nets the learning comes in one of two flavors: either supervised or unsupervised. The former requires the presence of an omniscient trainer, or teacher, able to adjust the net weights, in function of the net output response to each input pattern. Hopefully, the trainer ability should lead the net input-output relationships to converge to some desired goal. In the unsupervised learning, instead, the input set statistical properties should hopefully drive the net toward the desired classification goal.

By contrast, we are interested in the autonomously trainable nets, ie, in nets trainable independently both of the presence, or absence, of a teacher and of the input statistical properties. For the nets of interest to us, whereas the teacher presence can be useful as it reduces the number of training sessions, it is by no means necessary. In other words, the trainer presence, or absence, during the training period does not affect our net behavior, once the training period has been completed. The teacher presence, or absence, has the only effect to reduce, or to lengthen, our net training period. To train the autonomous trainable nets, we require the environment to feed the neural net sensors with input sequences containing a, non-necessarily aperiodic, repetition of subsequences. Trivially, any sufficiently long sequence, composed by symbols belonging to a finite alphabet, contains repetitions of subsequences.

To obtain the autonomous trainable nets, we shall implement in a BNN a Hebbian rule. That is, a version of the rule discussed by Hebb in [5 pag. 62]:

"*When an axon of a cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.*"

and in [5 pag. 70]:

"*The general idea is an old one, that every two cells or systems of cells are repeatedly active at the same time will tend to become "associated", so that the activity in one facilitates activity in the other.*"

However, there are some problems in implementing the Hebb rule as it has been stated. In a BNN with the time dependent coupling coefficients , see the chapter 4. 1. 1., adjusting their values as ruled, namely without any upper bound, very soon all the coupling coefficients becomes greater than the node threshold. That is, it becomes a BNN such that a single node activity induces the whole net activity, for ever. Because the coupling coefficients can be only increased and in no circumstance whatsoever they are decreased, we arrive to the same quite uninteresting result, if the BNN cannot forget.

4.1.1. *The Hebbian rules.*

A delay dependent positive coupling coefficient can be either time independent, as the one introduced in the chapter 1. 3, or time dependent, to be represented as $C_{h,\,k,\,d}[t]$. Let us call *Hebbian* the excitatory connection associated to a delay and time dependent positive coupling coefficient $C_{h,\,k,\,d}[t]$. Let us call *non-Hebbian* the connection associated to the delay dependent and time independent $C_{h,\,k,\,d}$.

We call *Hebbian-out, Hebbian-in,* the nodes receiving, sending, at least one Hebbian connection, e. g. the nodes plying the node A, B, role in the above given Hebb rule. That is, for us a Hebbian-out node transmits its activity to a Hebbian-in node and the former becomes more effective in inducing the latter activity, the more frequently both become active, with the given delay. We call non-Hebbian a node receiving at least one non-Hebbian connection. Obviously, as a node can receive, and send, both Hebbian and non-Hebbian connections, it can be at the same time Hebbian-in, Hebbian-out and non-Hebbian. For each h and d, we call $C_{h,\,k',\,d}[t]$ the time dependent coupling coefficients associated to the Hebbian connections impinging, with the delay d, upon the Hebbian-in node h and stemming out of the Hebbian-out node k'. In other words, the node k' is the generic element of the h dependent set of all the Hebbian-out nodes sending Hebbian connections to the Hebbian-in node h. In a net, the presence of a Hebbian-in node implies the presence both of a Hebbian-out node and of a Hebbian connection, and vice versa.

Let us call *Hebbian threshold* the upper bound we are going to introduce for the time dependent coupling coefficient values. Once a time dependent coupling coefficient becomes no smaller than the Hebbian threshold, it becomes a time independent constant: we call it a *frozen* coupling coefficient. If the Hebbian threshold is no smaller than the node threshold, a Hebbian-in node can be trained to compute its inputs inclusive disjunction, at the most. On the contrary, if the Hebbian threshold is smaller than the node threshold, a Hebbian-in node can be trained to compute its inputs conjunction, at the most. Eventually, if the Hebbian threshold is equal to the node threshold, we can train the Hebbian-in nodes in either connectives. Accordingly, in our version of the Hebbian rule, we shall freeze the time dependent coupling coefficients associated to the active connections, see the chapter 1. 3, terminating onto a Hebbian-in node, as soon as their sum becomes no smaller than the Hebbian-in node threshold. For the time dependent coupling coefficients associated to the inactive connections, see the chapters 4.1.1.1. and 4.1.1.2.

The Hebb rule tell us something on the time dependency of the $C_{h,\,k',\,d}[t]$ associated to the d delayed Hebbian connection going from the Hebbian-out node k' to the Hebbian-in node h, when they are active, in the order, with the delay d. It says nothing on the time dependency of the coupling coefficients $C_{h,\,k',\,d}[t]$ when either h or k' or both are inactive. In other words, if we want to extend the rule to the inactive node cases, we can do it in many different ways. If the node h is inactive and the node k' is active, in our version of the Hebbian rule the $C_{h,\,k',\,d}[t]$ is decreased. Symmetrically, the $C_{h,\,k',\,d}[t]$ is decreased if the node k' is inactive and the node h is active. How we must choose the relative amounts? And what the new rule should say if both the nodes are inactive?

Moreover, the Hebb rule tell us nothing when we have many Hebbian-out nodes sending time dependent connections to many Hebbian-in nodes. As we cannot assume to have all the time dependent coupling coefficients with the same value at the same time, we must consider the case of some connections associated to time dependent coupling coefficients greater than the others. In other words we must consider the case with frozen and not yet frozen Hebbian connections.

Accordingly, we give two parametric version for the Hebbian rule. In one we consider the Hebbian-in node to be no more Hebbian once at least one of its incoming Hebbian connections is frozen, ie, we set to zero all its non frozen incoming Hebbian connections. In the second version, we consider the Hebbian-out node to be no more Hebbian once at least one of its outgoing Hebbian connections is frozen, ie, we set to zero all its non frozen outgoing Hebbian connections. In the following we call *first Hebbian rule,* respectively *second Hebbian rule* the first, second, Hebbian rule version. We shall say *Hebbian rule* whenever it is indifferent which of the two versions we are discussing.

In the same neural net we can have Hebbian nodes satisfying either the first or the second Hebbian rule, not both. To this end it is sufficient that each Hebbian node belong to just one *Hebbian set,* ie, to a set of Hebbian nodes such that: *first, it contains all the Hebbian-in, the Hebbian-out, nodes receiving, sending, the Hebbian connections coming*

out of, impinging onto, the Hebbian-out, the Hebbian-in, nodes belonging to the same set, second, the nodes belonging to the set do not receive, send, Hebbian connections to the nodes belonging to a different Hebbian set.

Eventually, let us call *Hebbian parameters* the positive constants a, b, e, g. The Hebbian parameter absolute values determine the *cadence*, namely the number of times relative to the time interval under consideration, the Hebbian node h and the Hebbian node k have to be active, with the delay d, to freeze the coupling coefficient associated to the Hebbian connection carrying the signal from the latter to the former with the delay d. Actually, if we choose a sufficiently high value for the Hebbian parameter g, relative to the values we choose for a and b and e, the $C_{h,k',d}[t]$ can be frozen in as little as two τs.

4.1.1.1. *The first Hebbian rule.*

To simplify the presentation, we call $C_{h,\,k'',\,d}[t]$ the time dependent coupling coefficient associated to the connection carrying a signal with a delay d from the active Hebbian-out node k" to the Hebbian-in node h. We call $C_{h,\,k''',\,d}[t]$ the time dependent coupling coefficient associated to the connection carrying a signal with a delay d from the inactive Hebbian-out node k"' to the Hebbian-in node h. The time dependent coupling coefficients are computed every τ by applying the first Hebbian rule:

$C_{h,\,k',\,d}[t+\tau] = C_{h,\,k',\,d}[t] - a$ if $f[h, t] = f[k', t - d*\tau] = 0$ and $0 \le C_{h,\,k',\,d}[t] - a$,

$C_{h,\,k',\,d}[t+\tau] = -[C_{h,\,k',\,d}[t] - a]$ if $f[h, t] = f[k', t - d*\tau] = 0$ and $C_{h,\,k',\,d}[t] - a < 0$,

$C_{h,\,k',\,d}[t+\tau] = C_{h,\,k',\,d}[t] - e$ if $f[h, t] = 1 - f[k', t - d*\tau] = 1$ and $0 \le C_{h,\,k',\,d}[t] - e$ and $\sum_{k''} C_{h,\,k'',\,d}[t] * f[k'', t - d*\tau] < T$,

$C_{h,\,k',\,d}[t+\tau] = -[C_{h,\,k',\,d}[t] - e]$ if $f[h, t] = 1 - f[k', t - d*\tau] = 1$ and $C_{h,\,k',\,d}[t] - e < 0$ and $\sum_{k''} C_{h,\,k'',\,d}[t] * f[k'', t - d*\tau] < T$,

$C_{h,\,k',\,d}[t+\tau] = C_{h,\,k',\,d}[t] - b$ if $f[h, t] = 1 - f[k', t - d*\tau] = 0$ and $0 \le C_{h,\,k',\,d}[t] - b$,

$C_{h,\,k',\,d}[t+\tau] = -[C_{h,\,k',\,d}[t] - b]$ if $f[h, t] = 1 - f[k', t - d*\tau] = 0$ and $C_{h,\,k',\,d}[t] - b < 0$,

$C_{h,\,k',\,d}[t+\tau] = C_{h,\,k',\,d}[t] + g$ if $f[h, t] = f[k', t - d*\tau] = 1$ and $\sum_{k''} C_{h,\,k'',\,d}[t] * f[k'', t - d*\tau] < T$,

$C_{h,\,k'',\,d}[t+\tau] = C_{h,\,k'',\,d}[t] = C_{h,\,k'',\,d}$
and $C_{h,\,k''',\,d}[t+\tau] = C_{h,\,k''',\,d} = 0$,

if $f[h, t] = f[k'', t - d*\tau] = 1$ and $f[h, t] = 1 - f[k''', t - d*\tau] = 1$ and $\sum_{k''} C_{h,\,k'',\,d}[t] * f[k'', t - d*\tau] \ge T$, ie, the node h is no more a Hebbian-in node.

To sum it up, if either the Hebbian-in node h, or the Hebbian-out node k', or both, are inactive, with the delay d, the coupling coefficient associated to the Hebbian connection between them is decreased. In any case, the coupling coefficient associated to a Hebbian connection is decreased as long as it stays non negative. That is, it does never assume

negative values. If both the Hebbian-in node h is active at the time t and the Hebbian-out node k" has been active at the time $t - d^*\tau$ and if the sum of the time dependent coupling coefficients associated to the Hebbian connections stemming from the said active Hebbian-out nodes k" and impinging onto the Hebbian-in node h is smaller than the node threshold T, then the coupling coefficient $C_{h, k", d}[t]$ associated to the Hebbian connection between them is increased. Otherwise, namely if both the node h is active at the time t and the node k" has been active at time $t - d^*\tau$ and if the latter sum is no smaller than the node threshold T, the coupling coefficients $C_{h, k", d}[t]$ become the frozen, ie, the time independent constants $C_{h, k", d}$, from the next τ on, and all the remaining Hebbian connections with an associated coupling coefficient $C_{h, k"', d}[t + \tau]$ stemming from the inactive, at the time $t - d^*\tau$, Hebbian-out nodes k"' and impinging onto the Hebbian-in node h are set to zero. In particular, the connections between the nodes k' and the node h are no longer Hebbian, see also 4. 3. In the last clause we have set to zero the time dependent coupling coefficients associated to the inactive connections, namely to the connections coming out of an inactive Hebbian-out node and terminating onto the active Hebbian-in node h.

To have frozen them at their greater than zero unknown values could lead to unwanted as well unforeseeable consequences. To see why, let us consider the three sets A, B and C of Hebbian-out nodes sending connections onto the Hebbian-in node h. If, before the freezing of the connections coming out of the nodes belonging to the set C, the three sets A, B and C are active independently of each other, then, once the connections have been frozen, the node h could becomes active either because the nodes belonging to the set C are active or because the nodes belonging to both the set A and the set B are active with suitable delays. In other words, the freezing of the Hebbian connections to greater than zero unknown values can lead to the uncontrollable implementation of functions in the net. By contrast, if the connections coming out of the nodes belonging to the set C are frozen and, at the same time, the connections coming out of the nodes belonging to both sets A and B are set to zero, then the node h becomes active only when the nodes belonging to the set C become active. That is, the node h activity, ie, the function computed by the node h does not depends on the activity of the nodes belonging to the sets A and B.

4.1.1.2. *The second Hebbian rule.*

The time dependent coupling coefficients are computed every τ by applying the second Hebbian rule:

$C_{h, k', d}[t + \tau] = C_{h, k', d}[t] - a$ if $f[h, t] = f[k', t - d^*\tau] = 0$ and $0 \leq C_{h, k', d}[t] - a$,

$C_{h, k', d}[t + \tau] = -[C_{h, k', d}[t] - a]$ if $f[h, t] = f[k', t - d^*\tau] = 0$ and $C_{h, k', d}[t] - a < 0$,

$C_{h, k', d}[t + \tau] = C_{h, k', d}[t] - b$ if $f[h, t] = 1 - f[k', t - d^*\tau] = 0$ and $0 \leq C_{h, k', d}[t] - b$ and $\sum_{k'} C_{h, k', d}[t] * f[k', t - d^*\tau] < T$,

$C_{h, k', d}[t + \tau] = -[C_{h, k', d}[t] - b]$ if $f[h, t] = 1 - f[k', t - d^*\tau] = 0$ and $C_{h, k', d}[t] - b < 0$ and $\sum_{k'} C_{h, k', d}[t] * f[k', t - d^*\tau] < T$,

$C_{h, k', d}[t + \tau] = C_{h, k', d}[t] - e$ if $f[h, t] = 1 - f[k', t - d^*\tau] = 1$ and $0 \leq C_{h, k', d}[t] - e$,

$C_{h, k', d}[t + \tau] = -[C_{h, k', d}[t] - e]$ if $f[h, t] = 1 - f[k', t - d^*\tau] = 1$ and $C_{h, k', d}[t] - e < 0$,

$C_{h,k',d}[t+\tau] = C_{h,k',d}[t] + g$ if $f[h,t] = f[k', t - d^*\tau] = 1$ and $\sum_{k'} C_{h,k',d}[t] * f[k', t - d^*\tau] < T$,

$C_{h,k',d}[t+\tau] = C_{h,k',d}[t] - C_{h,k',d}$
and $C_{h',k',d'}[t+\tau] = C_{h',k',d'} = 0$, for $\forall d' \neq d$ and for $\forall h' \neq h$,

if $f[h,t] = f[k', t - d^*\tau] = 1$
and $f[h', t] = 1 - f[k', t - d'^*\tau] = 0$
and $\sum_{k'} C_{h,k',d}[t] * f[k', t - d^*\tau] \geq T$,
ie, the node k' is no more a Hebbian-out node.

To sum it up, if either the Hebbian-in node h, or the Hebbian-out node k', or both, are inactive, with the delay d, the coupling coefficient associated to the Hebbian connection between them is decreased. In any case, the coupling coefficient associated to a Hebbian connection is decreased as long as it stays non negative. That is, it does never assume negative values. If both the Hebbian-in node h is active at the time t and the Hebbian-out node k' has been active at the time $t - d^*\tau$ and if the sum of the time dependent coupling coefficients associated to the Hebbian connections stemming from the said active Hebbian-out nodes k' and impinging onto the Hebbian-in node h is smaller than the node threshold T, then the time dependent coupling coefficient $C_{h,k',d}[t]$ associated to the Hebbian connection between them is increased. Otherwise, namely if both the node h is active at the time t and the node k' has been active at the time $t - d^*\tau$ and if the latter sum is no smaller than the node threshold T, then, the coupling coefficients $C_{h,k',d}[t]$ become the frozen, ie, the time independent constants $C_{h,k',d}$, from the next τ on, and all the remaining Hebbian connections stemming from the active, at the time $t - d^*\tau$, Hebbian-out nodes k' and impinging onto the inactive Hebbian-in node h' are set to zero, for any delay d' different from d and any Hebbian-in node h' different from h. In particular, the connections between the nodes k' and the node h are no longer Hebbian, see also 4. 3..

In the last clause we have set to zero the time dependent coupling coefficients associated to the connections coming out of an active Hebbian-out node and terminating onto the inactive Hebbian-in nodes. By contrast, if they are frozen at their greater than zero unknown values we could have, depending on the said unknown values, unpredictable results. E. g., given the Hebbian-out nodes k' and k" and the Hebbian-in nodes h', h" and h'", if for a sufficient number of t values and for suitable d' and d" we have

$$f[h', t] = f[k', t - d'^*\tau] = 1$$
$$f[h", t] = f[k", t - d"^*\tau] = 1$$

then, because of the second Hebbian rule, we have the greater than the threshold frozen $C_{h',k',d'}$ and $C_{h",k",d"}$ values. However, if we do not set to zero the remaining coupling coefficients, we could have as well $C_{h'",k',d'"} + C_{h'",k",d""} > T$ frozen, even if not once we had

$$f[k', t - d'"^*\tau] = f[k", t - d""^*\tau] = 1.$$

That is, we end with the node h'" computing the logical conjunctions of the nodes k' and k" activity, even if in the past not once it has ever computed such a function. Because such result depends on the $C_{h'",k',d'"}$ and $C_{h'",k",d""}$ unknown values, to have Boolean nets with predictable behaviors, we chose the given second Hebbian rule version.

4.1.2. *The Hebbian rule and the logical connectives.*

The given Hebbian rule consents us to freeze the time dependent coupling coefficients so to obtain a net computing any combination whatever, depending on the net activity in the past, of the conjunctions and/or of the inclusive disjunctions of the Hebbian-in nodes input. In other words, let us consider a Boolean net containing an output node receiving, first, a non-Hebbian connection greater than the threshold; second, N Hebbian connections. That is, we are considering a Boolean net having one output node both non-Hebbian and Hebbian-in, see the Figure 4.1.

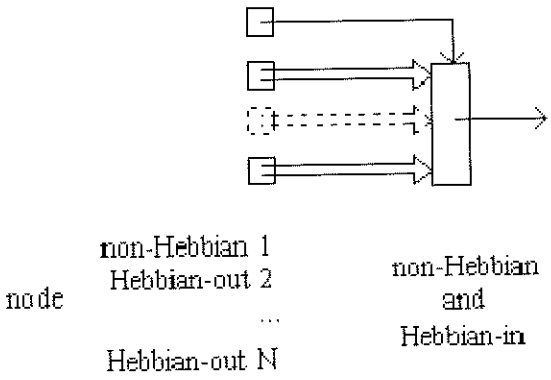


Fig. 4.1. *The Hebbian connection freezing and the logical connectives.* The empty arrows represent the Hebbian connections. The arrow connecting the node non-Hebbian 1 to the node Hebbian-in represents one facilitatory connection greater than the threshold. Initially, the output node is active at the time t if and only if the node non-Hebbian 1 is active at the time t - τ. The Hebbian connections are frozen by repeatedly activating the node non-Hebbian 1 and, after one τ, at least one of the Hebbian-out nodes. The activation cadence needed to freeze the Hebbian connections depends on the Hebbian parameter values. Our Hebbian rule consents to freeze either one or two or ... or N time dependent coupling coefficients. In other words, once some Hebbian connections have been frozen, the output node is active at the time t because the right combination of its input nodes have been active at the time t - τ. That is, once the Hebbian connections have been frozen, the net computes a combination of its input conjunctions and/or inclusive disjunctions, the actual combination depending on the input history.

Initially, the net output node computes the inclusive disjunction of its non-Hebbian connection, only. Because of our Hebbian rule, it is possible, always, to freeze the time dependent coupling coefficients, namely the output node Hebbian-in connections. Once they have been frozen, the Boolean net computes one of any given combination of its input conjunctions and/or inclusive disjunctions. That is, once the time dependent coupling coefficients have been frozen, the actual function computed by the net output node depends on the input history. And the time required to freeze the time dependent coupling coefficients depends on the Hebbian parameters and on the relevant input repetion cadence.

4.2. The inhibitory connection case.

As we are considering Boolean neural nets with absolute inhibition, it is meaningless to extend the Hebbian rule to the time dependent negative coupling coefficients, see also 16 cap. 41. Apart from particular cases, e. g. for the two node BNNs, there is no known way to extend meaningfully a Hebbian rule to the time dependent negative coupling coefficients, be them associated to the absolute or to the relative inhibitory connections. To see the time dependent negative coupling coefficient uselessness, let us consider the net represented in the Figure 4.2..

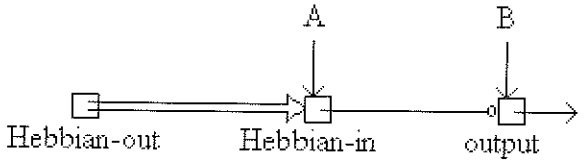


Fig. 4.2.. *The Hebbian connection freezing and the inhibition.* The empty arrow represents a Hebbian connection. Each of the remaining arrows represents one facilitatory connection greater than the threshold. The empty circle represents an absolute inhibitory connection. Initially, the output node is active at the time t whenever both the input B is active at the time t - τ and the input A is not active at the time t - 2τ. The Hebbian connection is frozen by repeatedly activating the input A and, after one tau, the node Hebbian-out. The activation cadence needed to freeze the Hebbian connection depends on the Hebbian parameter values. Once the time dependent coupling coefficient is frozen, the output node is active at the time t whenever both the input B is active at the time t - τ and neither the input A nor the node Hebbian-out nor both are active at the time t - 2τ. That is, a net output can be inhibited by varying the facilitatory connection efficacies and leaving unchanged its non-zero inhibitory connection efficacies.

Initially, the represented net output node is active at the time t whenever both the input B is active at the time t - τ and the input A is not active at the time t - 2τ. To train the net, we activate repeatedly both the input A and, after one tau, the Hebbian-out node, with the net activation cadence depending on the Hebbian parameter values. Once we have completed the training, the output node is active at the time t whenever both the input B is active at the time t - τ and neither the input A nor the Hebbian-out node is active at the time t - 2τ. That is, whereas initially the net output was inhibited by the input A activity, alone, once the training has been completed, the net output is inhibited either by the input A or by the Hebbian-out node or by both. In other words we have obtained the inhibition of the net output by varying a facilitatory connection efficacy. Its non-zero inhibitory connection efficacy has been left unchanged.

4.3. A necessary condition for the implementation of the Hebbian rule in a BNN.

We are interested in implementing our Hebbian rule in a BNN with a finite output subvector. However, once implemented the Hebbian rule, ie, once implemented a rules to modify the algorithms a BNN computes, we must guaranty the BNN proper functioning. That is, the net states, in particular its output states, have to continue to satisfy the initial conditions. E. g., the Hebbian rule implementation must be done so to avoid the possibility to train the net, inadvertently, to activate all its output nodes, for ever. To this end, let us discuss how to choose the Hebbian connections in order to guaranty an upper bound on the number L of the simultaneously active output nodes. Obviously, we can satisfy the said condition when drawing the net blueprint, for the non-Hebbian connections. Instead, for the Hebbian connections we must take into account the consequences the training can have, also.

If in a neural net there is a Hebbian set containing Hebbian nodes satisfying the first Hebbian rule, whether or not the net contains a second Hebbian set, to have no more than L simultaneously active output nodes, no node can be at the same time Hebbian-in and Hebbian-out and each pattern of activity of the Hebbian-out nodes has to corresponds to one, at the most, active Hebbian-in node.

To guarantee no more than L output nodes simultaneously active, at any given time, the activity of each Hebbian-in node has to induce the activity in L output nodes, at the most. So, once a Hebbian connection has been frozen the condition is certainly satisfied. However, if the Hebbian nodes obey the second Hebbian rule and we are interested in

unbounded *incremental learning*, i.e., in concatenating some of the previously learned tasks, without limitations. That is, let us call S_1 the set of Hebbian-out nodes trained, previously, to control the task s_1, S_2 the set of Hebbian-out nodes trained, previously, to control the task s_2. As we are interested in incremental learning, let us call S_3 the set of Hebbian-out nodes we have to train to control the task s_3, ie, the task obtained by concatenating s_1 with s_2, ie, the task sequence s_1s_2. More precisely, s_3 is the task obtained by executing the task s_1 followed by the task s_2. Accordingly, to have unbounded incremental learning, the number of output nodes $O(S_i, d)$ simultaneously activated by the Hebbian-in nodes trainable, with a delay d, by the Hebbian-out nodes belonging to each set S_i must be no greater than L. To sum it up, *to implement a Hebbian rule in a Boolean neural net, at the same time guaranteeing both unbounded incremental learning and no more than L simultaneously active output nodes, we must choose the net coupling coefficients so to satisfy the following necessary condition.*

$$\forall d \sum_i O(S_i, d) \leq L$$

In our Hebbian rules, we have formulated the last, namely the otherwise, clause so to simplify the compliance with the given necessary condition. Actually, in the said clause we have introduced an element lacking in the Hebb's formulation. Namely, once one of the Hebbian connections stemming out of the active Hebbian-out node H has been frozen to a value greater than zero, we have set to zero all the remaining Hebbian connections impinging onto the presently active Hebbian-in node, stemming out of the very same H. In other words, let us consider the contrary case, ie, a net in which we have implemented the second Hebbian rule, first, requiring the freezing of the Hebbian connections stemming out of a Hebbian node H once the time dependent coupling coefficient sum becomes no smaller than the threshold, second, telling nothing about the remaining Hebbian connections stemming out of H. Let S_b and S_c be two sets of Hebbian-out nodes belonging to the said net. The former set is activated whenever the net input is b and the latter set is activated whenever the net input is c. Let the net be trained to execute the task b, requiring the activation of L output nodes, whenever the input is the sequence bc. That is, the task b is the net response to the delayed conjunction of the input b followed by the input c. If the second Hebbian rule tells nothing on the set S_c remaining Hebbian connections once some of them have been frozen, we can train the nodes belonging to the set S_c to control the execution of the task c, whenever the input is c. Even if the task c execution requires activity in less than L output nodes, once the second training period has been completed, the input sequence bc shall activate the simultaneous execution of the tasks b and c, ie, the input sequence bc shall activate more than L output nodes. Accordingly, the necessary condition is not satisfied.

However, the necessary condition does not guarantee a loop free BNN, once the Hebbian connections have been frozen. As we shall see in the chapter 5, in some instances a BNN free from the loops resulting from the Hebbian connections freezing depends on the *example presentation* sequence, ie, on the presence of a teacher. Instead, for the *general assembly*, see the chapter 5. 6., a BNN free from the loops resulting from the Hebbian connections freezing is the consequence of the net wiring and of the second Hebbian rule last clause.

We have called primitive a Boolean net controlling the *primitive tasks*, eg see the chapter 2.2.1.. Let us call *composite* any task obtained by concatenating two, or more, primitive tasks. Whereas in the chapter 2 we have discussed a BNN wired so to control both the primitive and the composite tasks, now we are interested in a net wired so to control the primitives tasks initially and the composite tasks only once it has accepted some suitable inputs. In any case the latter net has a finite output subvector, ie, it has been designed to control finitely many primitives.

Let us consider one of the latter nets, if we train S_{one}, namely its first set of Hebbian-out nodes, to control a sequence composed by less than L, e. g. by two, primitives, we have satisfied the above given necessary condition. Then, if we train S_{two}, namely its second set of Hebbian-out nodes, to control either the sequence composed by S_{one} and one sequence composed by less than L, e. g. by two, primitives or two sequences each composed by less than L, e. g. by two, primitives, we have satisfied the above given necessary condition, again. Once we have trained the Hebbian-out nodes belonging to the sets S_{one}, S_{two}, ..., S_h, ..., S_k, ..., S_{i-1}, satisfying the above given necessary condition, if we train S_i, namely the i-th set of Hebbian-out nodes, to control either the sequence S_hS_k, or the sequence composed by S_h and one sequence composed by less than L, e. g. by two, primitives or two sequences each composed by less than L, e. g. by two, primitives, we continue to satisfy the above given necessary condition. That is, the above given necessary condition is satisfied as we have implemented incremental learning in our net. On the other hand, a BNN must have a finite binary output subvector. Accordingly, *if we satisfy the above given necessary condition in a BNN, we can implement incremental learning in it.*

We have proposed a BNN organized in *assemblies*, see the chapter 5 and [6], as a sufficient condition on the coupling coefficients, for the implementation of our Hebbian rule in a BNN. The proposed sufficient condition is compatible with the above given necessary condition and guarantees both the unbounded incremental learning and the no more than L simultaneously active output nodes. In other words, instead of a N node BNN learning to compute a given function f, we have proposed to consider N smaller nets, the assemblies, connected so to obtain a larger net both satisfying the given necessary condition and guaranteeing both the unbounded incremental learning and the no more than L simultaneously active output nodes.

Chapter 5

THE ADAPTABLE BNN, OR ABNN

5.1. Introduction.

In accordance with the definition given in the chapter 2. 3., we call *Adaptable Boolean Neural Network*, or *ABNN, a BNN, first, having some Hebbian connections, second, initially computing only the primitives, third, once it has been trained to compute the function g, see below, for a suitable number of the independent variable values, computing both the very same primitives and at least the function g wherever it is defined.* In the following, whenever an ABNN computes autonomously a function g for a suitable number of its independent variable values, we shall say *the ABNN has been trained*, or *has learned, to compute the function g.* Let us call g_1 and g_2 two functions the ABNN computes and let g be a function computable as their functional product g_2g_1 for some of the g_1 independent variable values, at the least. We call *example* the ordered input triple l(independent variable values, g), g_1, g_2l, where g_1 is not necessarily the input immediate successor of the input (independent variable values, g) and g_2 is not necessarily the input immediate successor of the input g_1. We say the ABNN *accepts the example* E if *the ABNN has been trained to compute the function g* because of the E repetitive presentation as ABNN inputs. Some instances of the functions g_1 and g_2 can be found in the chapter 2. 2. 2, see the Table 2. 1., where we have characterized by means of their input/output correspondences the BNNs computing the Pythagorean table for the addition. Some instances of the function g can be found in the chapter 6., eg in 6.3. we discuss a procedure to train an ABNN to add numbers represented by numerals with an unbounded number of digits

To define an ABNN we have introduced the two locutions: *to compute the function g for a suitable number of the independent variable values* and *the ABNN has been trained.* In the next chapter 5. 2., we shall deal with the latter, spelling out formally what we mean by it. The former we discuss right now, instead.

To start with, let us set some nomenclature questions about the nets with time dependent coupling coefficients. Usually, in the neural network literature learning is qualified as supervised when the net output is measured by a tutor, against some known and desired goal. The very same tutor somehow adjusts the net weights accordingly to the training session results, until the desired output is obtained. By contrast, in the unsupervised learning the net weights are adjusted until some clustering of the input vectors is detected, i.e., until similarities in the latter are recorded in the output vector values. E. g., the unsupervised learning is called Hebbian whenever the weights are adjusted by applying a Hebbian rule. In any case the learning process can be seen as a search for the weight values classifying the input vector sets. With the desired output vector sets known in the supervised case and unknown in the unsupervised case. Eventually, the neural

network have been studied also as non linear dynamical systems, see [4]. In this case, the training is the search for the weights associating a particular input perturbation, the key in the retrieval phase, to a particular steady state, or to the desired output.

By contrast, the learning paradigm we are discussing here, see also [6, 17], is different from and must not be confused with the Hebbian learning studied in the classical paradigm. Here, we are interested in an incremental learning paradigm based on the Hebbian rules given in the chapter 4.1.1., as distinct from the unsupervised Hebbian learning. We have implemented in a Boolean neural net both Hebbian rules thanks to the Caianiello formalism, see the chapter 4.1.. Now our aim is to assign a coupling coefficient matrix characterizing an autonomously adaptable Boolean neural net behaving like a filter in the time domain. Actually, the ABNN selects the input sequences on the basis of their presentation properties, ie, without outside intervention or help. In other words, the ABNN selects autonomously the examples, ie, the sequence composed by no more than two or three, respectively the conditional and the unconditional, input subvectors repeated a sufficient number of times. Here the sufficient number is determined both by the values chosen for the Hebbian parameters and by the very same examples repetition cadence, i.e., by how often the training sessions follow each other. Actually, with the words "the net selects autonomously the examples" we mean that the net leaves in, ie, stores only the inputs composed by the two or three input sequences separated by the delays satisfying the conditions implicitly defined by the net wiring. Accordingly, once the training has been successfully completed, the net generates the desired output vector sequence every time the conditional input vector is present at its sensory input, whether or not it is followed by the unconditional input vectors.

Implicitly, the input vector sequences define a clock ticking on, as the net ruminates its input. As the time passes by the dependence of the coupling coefficient variations on the training sessions repetion cadence becomes apparent, independently of any teacher presence. Accordingly, in our net most general version, we can dispose of the teacher as its role is, at most, to shorten the learning process by letting in, at intervals as short as possible, the very same input vector sequences so to frozen as fast as possible the relevant Hebbian connections. As usual, the time intervals depend on the times required for the output task executions. Instead, the time required to freeze the relevant Hebbian connections depend on the values we have chosen for the Hebbian parameters. In our simulations, see [6, 18, 19], the *suitable number*, or *suitable cadence*, of the same example presentations can be as low as two or three and depends on the values chosen for the Hebbian parameters. The *number of the same example presentations* increases with the increase of the noise to signal ratio, ie, of the ratio between unwanted and wanted input vector sequences. By contrast, the *number of distinct and different examples needed to train the ABNN to compute the function g for a suitable number of independent variable values* depends, first, on the set of the ABNN primitives, second, on the number of functions the ABNN has been previously trained on and, third, on how g can be expressed as their functional product.

For some particular wiring diagrams the teacher role is more pregnant than in others. Eg for the net discussed in the chapter 5. 4. the examples must be broken down in sequences of primitives. Instead, in the net discussed in the chapter 5. 6. the teacher unessential role is simply to speed up the training process, ie, the teacher can be disposed of without hampering the net trainability. Accordingly, to avoid ambiguity with the classical learning paradigms, instead of supervised and unsupervised learning we speak about assisted learning and unassisted learning, or autodidactic learning.

In the chapter 4, we have given a necessary condition to implement the Hebbian rules in a BNN, now we are going to discuss, first, some of the difficulties arising when we try to implement a Hebbian rule in a BNN, second, some sufficient conditions for the wiring diagrams guaranteeing a properly working ABNN, ie, an ABNN learning incrementally the tasks itself selects autonomously from the continuos and unpredictable input stream generated by the environment.

5.2. Some problems arising with a hebbian rule implementation.

5.2.1. *An informal discussion.*

Whereas in an ANN the supervised as well as the unsupervised learning algorithms role is to monitor the global learning processes going on in the entire net, ie, they empower something like a central control system, a GABNN has no central control, namely in a GABNN there is not a global algorithm monitoring all the Hebbian connection states. Actually, a Hebbian rule effect is to change the value of a coupling coefficient associated to a Hebbian connection, in function of the activity of the two connected nodes. Accordingly, the Hebbian connection values depend implicitly on the environment events and more explicitly on the activity going on within the GABNN. As we are interested to store in the net coupling coefficients the former events only, we must implement an independent control for the Hebbian nodes activities, ie, we must shield the GABNN time dependent connections freezing from the remaining node activities.

First of all, let us consider a net composed by connections with the maximum delay D, greater than two. Let us consider a net to be trained to compute the function g(A), where the very name g(A) is represented by the sequence 1, 2, 3, ..., N, with N greater than D. Here we have clearly a problem: for N smaller than D, we could have arranged the connections so to train the delayed Hebbian connections to store the different elements of the given sequences. No easy solution exists for N greater than D. Obviously, we could solve the problem if the net support incremental learning. In the latter case, we could train the net to compute, first, the function $g(A_1)$ represented by the sequence [1, 2]; second, the function $g(A_2)$ represented by the sequence [$g(A_1)$, 3]; third, the function $g(A_3)$ represented by the sequence [$g(A_2)$, 4]; ...; eventually, the desired function g(A) is now represented by the two components sequence [$g(A_{N-2})$, N]. Obviously, as the last sequence is composed by two components and D is greater than two, the net can be trained to compute the function g(A).

Now we are going to discuss some of the problems arising when we try to implement a Hebbian rule in a Boolean net. Once the rule has been implemented coherently and meaningfully, by no means an easy task, the learning procedures themselves can be a source of trouble. To this end, let us consider a Boolean neural net with the input composed by two input channels accepting binary vectors. That is, our net has a first input represented by the symbols a, b, c, ..., N, we call it the literal input channel, and a second input represented by the symbols 0, 1, 2, 3, ..., 9, we call it the numeral input channel. Eventually, the net output is represented by the symbols 0, 1, 2, 3, ..., 9 and each of them is the net effector response to the corresponding numeral input.

As an illustrative example, let us train the net in associating the literal input sequence z with the numeral input sequence 123435. That is, after a training period composed by a sequence of example presentations, we want the net to respond with the output sequence 123435 to the literal input z Whereas with a careful implementation of the second Hebbian rule in the Boolean neural net we can train the net so it can exhibit the said behavior, if the second Hebbian rule has been naively implemented, then, the most likely outcome is a net exhibiting the endlessly repetitive behavior 12343434... . Obviously, the said problems are easily solved in nets allowing the incremental learning, eg if we can associate the literal x to the numeral sequence 123, the literal y to the numeral sequence 435 and, then, the literal z to the sequence xy. We can also associate as many different literal names to the different numeral sequences composed by 1 followed by a finite number of numeral inputs. Similarly, if we want to train the net in associating, first, the literal input z with the numerical sequence 12 and, second, the literal input x with the numerical sequence 13. In case the second Hebbian rule has not been implemented with due care, we can obtain a net learning to associate both z and x to the sequence 12. That is, a net such that once it has been trained to associate an input sequence, eg z, to a second input

sequence, eg 12, then, if we try to train it to associate a different pair of input sequences having a common element, eg x with 13, we end with a net associating both first sequences, in our examples z and x, with the same sequence 12. In other words, apart a coherent and meaningful implementation of the Hebbian rules, the Boolean neural network must allow some kind of unlimited incremental learning, also.

There is another problem arising with the Hebbian rules implementation. In general, the net has a number of effectors and each of them has its own response time. E. g., for the BNN discussed in the chapter 2.2., the time to write the digit 1 is different from the time to write the digit 0. So we need a system working independently of the time required to output any given response. On the contrary case, as the maximum delay between the responses must be D, we can train the net on sequences with response times no greater than D.

5.2.2. *A formal approach.*

Formally, a BNN can be represented as

$$F[t] = \Pi(C * F[t - \tau] - T),$$

where F[t] is the state vector, C is the coupling coefficient matrix and T is the nodes common threshold. Given C and the BNN initial state F[0], actually, the given formalization is a recurrent relation we can apply again and again to obtain the succession:

$$F[0], F[1], F[2], ..., F[t], ...$$

Without any loss of generality, we can consider it as representing the BNN state vector evolution in the time domain, ie, the given succession represents the computations executed by the BNN.

Analogously, an ABNN can be formally represented by the relation:

$$F[t] = \Pi(C[t - \tau] * F[t - \tau] - T),$$

where, as in the BNN, F[t] is the state vector, T is the common threshold. By contrast, in an ABNN the coupling coefficient matrix C[t - τ] contains at least one time dependent element associated to a Hebbian connection. Again, if C[0] and the ABNN initial state F[0] are known, the given formalization is a recurrent relation we can apply to obtain the succession:

$$F[0], F[1], F[2], ..., F[t], ...,$$

representing the ABNN state vector evolution in the time domain. However, given C[0] and F[0], once we know the ABNN next state vector, thanks to the Hebbian rules, we can generate also its next coupling coefficient matrix, ie, we can generate the succession:

$$C[0], C[1], C[2], ..., C[t], ...$$

Whereas the ABNN state vector succession represents, as in the BNN case, the executed computations, the coupling coefficient matrix succession represents the succession of the functions the ABNN has been trained to compute. To implement the Hebbian rules in a BNN, ie, to obtain an ABNN, the initial coupling coefficient matrix C[0] must satisfy the necessary condition we have discussed in the chapter 4.3.

Let $g_1[x]$ and $g_2[x]$ be two functions the ABNN computes: eventually, either $g_1[x]$ or $g_2[x]$ or both can belong to the ABNN primitive set. In accordance with the definition given in the chapter 2.3., we shall say: *the ABNN has been trained,* or *the ABNN has learned, to compute,* or *converges to the constant* $g[x_c] = g_2[g_1[x_c]]$ *if* $g[x]$ *is defined on* x_c *and there exists a positive integer* t_c *such that both the input subvector succession components assume as value the sequence* $[(x_c, g), g_1, g_2]$ *and the output subvector succession components assumes, after a suitable delay, as value the sequence* $g[x_c]$ *a number of times no greater than* t_c *sufficient to freeze the relevant Hebbian connections.*

We shall say: *the ABNN has been trained,* or *has learned, to compute,* or *converges to, the function* $g[x] = g_2[g_1[x]]$ *if* $g[x]$ *is defined on* $x_1, x_2, ..., x_n$ *and there exists a positive integer* t_n *such that the ABNN converges, first, to the constant* $g[x_1]$*, second, to the constant* $g[x_2]$*, third, ..., n-th, to the constant* $g[x_n]$*, and, for* $t_n < t$*, the ABNN computes* $g[x]$*, wherever it is defined and whenever its input subvector components assume as value the sequence* $[x, g[x]]$.

The number of different functions on which an ABNN can converge depends on the coupling coefficient matrix size.

To freeze the Hebbian connections, the number of times the example $[(x_i, g), g_1, g_2]$ have to be presented, namely the number of times no greater than t_i the ABNN have to compute $g[x_i]$, depends on the Hebbian parameters, see the chapter 4. 1, and on the example presentation repetition cadence, eg on whether the learning is assisted or unassisted. On the other hand, the number of different examples needed for its training depends on the g_1 and g_2 functions, ie, on the ABNN input history. To see why, let us consider two different algorithms computing the sum of two integers.

In the chapter 2.2.2, we have implemented in a BNN an algorithm to compute the Pythagorean table for the sum, ie, the ten steps corresponding to the ten possible environment vectors represented in the Table 2. 2.. A different algorithm is inspired to the one used to teach, sometime, the sum to the six or seven year olds. Let us consider the environment vector (+, v, u, w, b), where b is a blank, v is either a blank or a one, u is either a blank or a digit and w is either a blank or a digit. That is, if we are dealing with number represented by numerals in the binary base, both u and w are, independently of one another, either a blank or a one or a zero. Then, the algorithm to sum two numbers is: first, let us call V the u-th successor of v; second, let us call W the w-th successor of V; third, write the W rightmost digit in the environment vector bottom component, ie, in the component holding the blank; fourth, write the W leftmost digit, either a one or a blank, in the next environment vector second, from the top, component; fifth, erase the present environment vector first two, from the top, components; sixth, write a + sign in the next environment vector topmost component. Obviously, the training on the new algorithm depends on whether the ABNN has to be trained to compute the k-th successor in the past, or not. Accordingly, the number of examples needed to train an ABNN to compute the sum is different if it has to be trained to execute the first or the second algorithm.

Eventually, the t_n value depends on the algorithms the ABNN has been previously trained on, on the number of different examples needed to characterize the algorithm chosen to compute the function g, on the number of each example presentations and on the time the ABNN needs to compute each example. To sum it up, t_n depends on the Hebbian parameters, on the teacher presence, ie, whether the learning is assisted or unassisted, on the algorithm A chosen to compute g and, if A is the name of the sequence of algorithms BCD...E, on whether or not the ABNN has been previously trained to compute the algorithm B and the algorithm C and the algorithm D and ... and the algorithm E.

5.3. On the sufficient conditions for an ABNN.

To set the conditions needed to implement the Hebbian rules in a Boolean neural net, in the most general way, we must discuss how to assign both the non-Hebbian and

the Hebbian connections, namely their associated coupling coefficients, so to guarantee a properly working net. Both in the Hebb original formulation and in the subsequently proposed Hebbian rules, the coupling coefficient adjournment depends both on the activity of the, single, transmitting node and on the delayed activity of the, single, receiving node, as it ought to be. However, we are interested in nets containing more than one [Hebbian-out, Hebbian-in] node pair, besides the non-Hebbian nodes. That is, either we consider a rule taking into account the activities of the remaining nodes: an unrealistic and unpractical solution, we do not investigate further. Or we give sufficient conditions for the time dependent as well as for the time independent coupling coefficients, ie, we assign a suitably chosen [10]. Our solution is to define the Iterable Boolean Neural Subnet, or IBNS, and to connect many of them so to obtain the desired ABNN.

As we want the desired ABNN to be trained by the environment events, ie, by the ABNN input patterns, the IBNSs must be activable by the ABNN inputs and their wiring must shelter the relevant Hebbian connections from the activity going on in the ABNN remaining Hebbian nodes.

5.3.1. *The IBNS activity and the ABNN input.*

To satisfy the first requirement, we connect the ABNN inputs to the IBNSs so that each IBNS corresponds to just one input patterns and one input pattern corresponds at the most to one IBNS. As we shall see in the chapters 5. 4. and 5. 5., a solution is to connect the ABNN inputs to the IBNSs with a characteristic wiring, ie, with a wiring characterizing each IBNS. Eg, let the ASCII code for j be the j-th IBNS *identifier*. Then, we wire the j-th IBNS to the ABNN input with connections such that every time j is the ABNN input subvector value, the j-th IBNS input node becomes active. However, we must consider also the case of an input inhibiting an IBNS. That is, we define an IBNS *inhibitory identifier*, or *i-identifier*, as the ABNN input subvector value inhibiting the IBNS characterized by the very same i-identifier. Let us consider the ABNN input subvector value assuming the identifier or the i-identifier value j, ie, the one characterizing the IBNS j. Whereas either the identifier or the i-identifier can have a cadence sufficient to freeze one of the j incoming Hebbian connections, the j outgoing Hebbian connections are frozen if the ABNN input subvector assumes the identifier value, only. That is, they cannot be frozen if the ABNN input subvector assumes the i-identifier value. This should not come as a surprise: if the ABNN input subvector assumes the j i-identifier value, j has been inhibited. That is, it cannot transmit any activity to any other IBNS, accordingly.

However, in the chapter 5.6. we present a different procedure to connect an IBNS to the ABNN inputs. There we discuss an IBNS with Hebbian input connections such that, in the resulting ABNN, there is at the most one IBNS trainable to become active every time just one particular pattern is the very same ABNN input. In turn, just one IBNS at the times is trainable to become active as the ABNN input patterns follow each other. Accordingly, as the input pattern, we call it the *calling sequence* or *i-calling sequence*, identifying the IBNS j is unknown at the blueprint stage, we reserve the name identifier, i-identifier, to the value j.

As we have seen, the input subvector components can assume two different identifier values for the same IBNS. We can extend the notion, ie, we can consider an ABNN input subvector assuming a finite number of different identifier values for the same IBNS. In other words, we can consider the ABNN input subvector composed by a finite number of mutually exclusive input sub-subvectors. In this case we shall say we are considering an ABNN with *multiple input modalities*. E. g., the ABNN with an input subvector assuming both the identifier, calling sequence, and, separately, the i-identifier, i-calling sequence, value for the same IBNS is an ABNN with a *duple input modality*. For simplicity sake, we consider the input modalities and the input sub-subvectors to be in a one-to-one correspondence, ie, we consider each input modality associated to just one in-

put sub-subvector, and vice versa. Accordingly, we shall speak of the first input modality and of the first IBNS identifier, calling sequence, of the second input modality and of the second IBNS identifier, calling sequence, and so on.

If we consider an ABNN with multiple input modalities, between the ABNN and its sensors, there is a *pre-input BNN* such that no more than one identifier, calling sequence, at the times is present at the ABNN input. A possible algorithm implemented in the pre-input BNN could be based on a correspondence between the input signals intensity and the number of the pre-input BNN contemporaneously active input nodes. That is, at each time the pre-input BNN could output its, eg loudest, input, ie, the input represented by the greatest number of contemporaneously active nodes, only. Or we could sort in order, ie, give a priority to the various input modalities. That is, at each time the pre-input BNN could output, among all the identifiers, or calling sequences, present at the input, only the input associated to the input modality with the highest priority. A third possibility is a combination of the two, eg, to solve the possible ambiguities. As there are many possible choice for the algorithm computed by the pre-input BNN, we leave to the interested reader to develop them. In any case we shall assume *an unrestricted ABNN environment and no more than one IBNS identifier*, or *calling sequence, present at the ABNN input, at any given time.*

Actually, we do not define explicitly the ABNN environment, ie, we consider the ABNN environment to be just a rule to generate an unending succession of the IBNS identifiers, namely of the input subvector states. In other words, for us the ABNN environment is represented by an unending succession of symbols drawn from an alphabet, the input alphabet, implicitly defined by the, either Hebbian or non-Hebbian, connections linking, in any ABNN, the sensor outputs to each IBNS input. That is, the wiring diagram specifying the, Hebbian or non-Hebbian, connections going from the pre-input BNN to the IBNS inputs, in any given ABNN, defines the input alphabet, ie, the complexity of the input patterns the very same ABNN can be trained on. Thus, if we want an ABNN trainable to see only the instantaneous input patterns, namely the input patterns represented by just one input subvector state, the connections between the ABNN sensors and the very same IBNS inputs must have just a delay of one tau. Instead, if we want an ABNN trainable to see a succession of input patterns, namely the input patterns represented by more than just one input subvector state, the connections between the ABNN sensors and the very same IBNS inputs must have a delay of more than just one tau, the actual delay depending on the particular succession we want the ABNN to be trained on. As a numeral in the base two is always a numeral in an higher base, of course representing different numbers depending on the base, so an ABNN trainable to see a succession of input patterns is also trainable to see simpler, eg instantaneous, input patterns, always.

5.3.2. *The Hebbian sets.*

We can shelter each Hebbian connection from the activity going on in the ABNN remaining Hebbian nodes by wiring the Hebbian-out and the Hebbian-in nodes so they belong to disjunct sets, the *Hebbian sets*, defined as follows. *First, the Hebbian-out nodes belonging to the h-th Hebbian set send their Hebbian connections to the Hebbian-in nodes belonging to the very same h-th Hebbian set, only. Second, the Hebbian-in nodes belonging to the h-th Hebbian set receive the Hebbian connections from the Hebbian-out nodes belonging to the very same h-th Hebbian set, only.*

If the ABNN input activate, inhibit, just one, at the most, IBNS input node at the time, ie, if the ABNN input carries just one, at the most, IBNS identifier, calling sequence, at the time, the ABNN can converge to the conjunction of more inputs having a one τ delay from one another, at least.

Eventually, as we shall discuss in the next chapters, we can wire the ABNN so both its Hebbian nodes belong to a Hebbian set and the necessary condition given in the chapter 4. 3 are satisfied. Actually, in an ABNN we need IBNSs of two kinds, namely the assemblies and the primitives. Whereas the assemblies are IBNS playing the role of both the input and the hidden nodes in a BNN, the primitive output nodes activity is the ABNN output, ie, they are in a one-to-one correspondence with the ABNN effectors and play the role of the output nodes in a BNN.

5.3.3. *The parent and the offspring assemblies.*

As an ABNN can be trained to compute the new sequence S ≡ S'S", ie, the sequence S composed by the sequence S' followed by the sequence S", where both S' and S" are finite sequences of primitive tasks the very same ABNN has been trained to compute in the past. As S' and S" are sequences of primitives, one of them or both can be composed by just one primitive. Let us call A the assembly controlling the S execution, A' the IBNS controlling the S' execution and A" the identifier of the IBNS controlling the S" execution. We call A the *father*, or *parent*, assembly, A' the *first son*, or *first offspring*, A" the *second son*, or *second offspring*, and either of them is an A *offspring*. Whereas both A' and A" must be different from A, they can compute the very same sequence S', ie, it can be S' ≡ S".

5.4. The six node IBNS.

5.4.1. *The six node assembly blueprint.*

In *a six node assembly*, see the Figure 5.1. and also [20], the nodes number one and two are the Hebbian-in input nodes. They receive the Hebbian connections from each of the six node assembly node number six and the non-Hebbian connections, with associated coupling coefficients greater than the threshold, from the ABNN input nodes. Both the six node assembly Hebbian-in input nodes send non-Hebbian connections to the node number three: one is a facilitatory and the other an inhibitory connection. The node number three sends a non-Hebbian connection to the node number five and, with a two τ delay, to the node number four. Whereas the first non-Hebbian connection is associated to a coupling coefficient greater than the threshold, the second is associated to a coupling coefficient equal to one half the threshold. That is, the node number five becomes active one τ after the node number three. The node number four is a Hebbian-out node sending its Hebbian connection to the ABNN primitives, see the chapter 5.4.2. The node number five, once activated by the node number three, computes the existential quantifier in the time domain, see also the chapter 1.3.1.3. It is inhibited by the mask, see the chapter 5.4.3, and by the end of the activity of the primitives. In turn, first it inhibits, then, with a two τ delay, activates the node number six. That is, as it is inhibited, either by the mask or by the end of the activity signal of a primitive, the node number five activates, with a one τ delay, the node number six. Eventually, the latter is the assembly second Hebbian-out node sending a Hebbian connection to all the nodes one and two of the remaining IBNSs and a non-Hebbian connection to the mask, see the chapter 5.4.3. and the Figure 5.3.. Every time a six node assembly activates a primitive, the latter, at the end of its activity, inhibits the assembly node five. This signal enables the present assembly to activate the next IBNS, either a different six node assembly or a primitive.

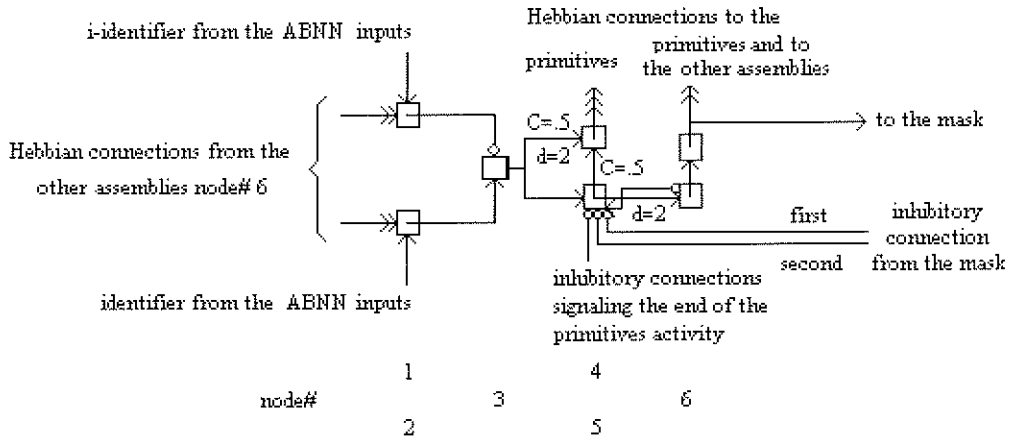


Fig. 5.1.. *The six node assembly.* The double and the tripled barbed arrows represent the Hebbian connections. The empty circles represent the absolute inhibitory connections. The single barbed arrows represent the non-Hebbian excitatory connections. Their associated coupling coefficient c is greater than the threshold and their delay d is one τ, unless otherwise specified.

With the six node assembly we have presented a two Hebbian-out node IBNS. It is possible to find IBNSs with more nodes having, also, more Hebbian-out nodes. However, greater the number of Hebbian-out nodes, greater the time required to send out their activity on their Hebbian-out connections. Accordingly, the number of example presentations needed to freeze their time dependent coupling coefficients increases with the number of Hebbian-out nodes present in an IBNS. Eventually, for given Hebbian parameters, if the number of Hebbian-out nodes becomes too large, the Hebbian connections cannot freeze, no matter how large is the number of training examples.

5.4.2. *The six node assembly functioning.*

The six node assembly node number one, two, are activated by the ABNN input subvector when it assumes the very same assembly i-identifier, identifier, value. The node one, two, are activated also by the Hebbian connections stemming out of the other assemblies node six, once they have been frozen. For simplicity sake we call the latter *the Hebbian-in connections.* The node number three is activated, after one tau, by the node number two activity and is inhibited, after one tau, by the node number one activity. The node number four is activated two taus after the none number three and one tau after the activity of the node number five. That is, if the latter has not been inhibited. The node number four is Hebbian-out and sends its Hebbian connection onto the primitives. We call the latter the six node assembly *first Hebbian-out connection.* If an assembly node number four activity is followed, the next tau, by the ABNN input subvector value being a primitive identifier, then, the coupling coefficient associated to the node number four Hebbian-out connection is increased. In turn, the node number five is activated, after one tau, by the node number three activity, if both the mask and no primitive are active at the same time as the node number three. Once activated, the node number five remains active until it is inhibited. Two taus after the node number five is inhibited, the node number six becomes active. The node number six is Hebbian-out and sends its Hebbian connection onto the ABNN remaining IBNSs. We call the latter the six node assembly *second Hebbian-out connection.* If an assembly node number six activity is followed, the next tau, by the ABNN input subvector value being an IBNS identifier, different from the

very same assembly identifier, then, the coupling coefficient associated to the node number six Hebbian-out connection is increased. We say a *six node assembly is active* either when its node number one is active or in the time interval comprised between the activity of its node number two and the inhibition of its node number five. Because a six node assembly first Hebbian-out connection terminates on the primitives, see 5.4.3. and the Figure 5.2., and its second Hebbian out connection becomes active one tau after the assembly has terminated its own activity, in an ABNN organized in six node assemblies just one assembly can be active at any given time. Accordingly, every time the node number six of a primitive becomes active only the assembly which activated the said primitive is the active one, ie, is the one which have to terminate its activity.

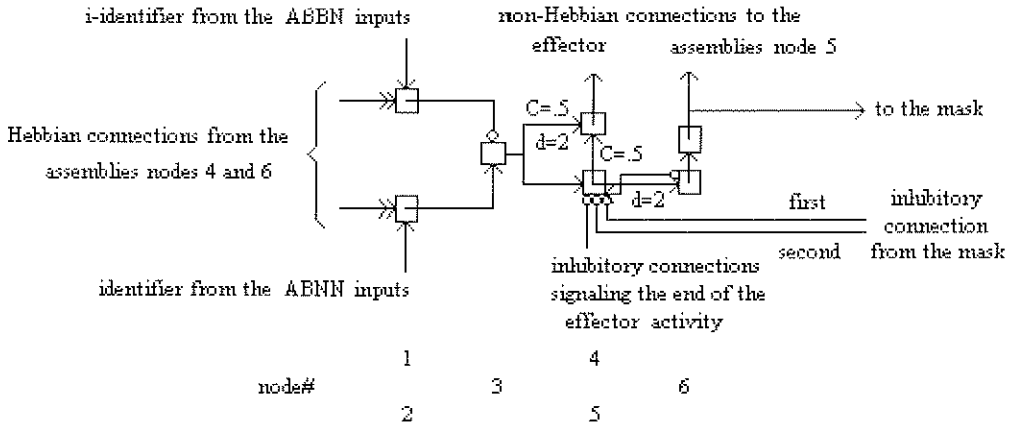


Fig. 5.2.. *The six node primitive.* The double barbed arrow represents the Hebbian connections. The empty circles represent the absolute inhibitory connections. The single barbed arrows without labels represent non-Hebbian excitatory connections. Their associated coupling coefficient is greater than the threshold and their delay is one τ.

An ABNN organized in six node assemblies can be trained to compute the new primitive sequence SP by incrementing, with the primitive task P, the sequence S of primitive tasks it has been trained to compute in the past. Let us call A the identifier of the assembly controlling the S execution, P the identifier of the IBNS controlling the primitive task P execution and A' the identifier of an assembly different from A. The ABNN can be trained to compute the sequence PA by freezing the assembly A' Hebbian-out connections so to activate first the IBNS P and, once P has terminated to control the task P, the assembly A. That is, the six node assembly filters from the ABNN input stream the sequences composed by a primitive identifier followed, with the right delay, by an assembly identifier, ie, the sequence PA. Obviously, as any command sequence can be always presented as the sequence composed by one elementary command followed by a sequence of commands, then the nets organized in six node assemblies can be universally trained. However, an ABNN organized in six node assemblies learns, either unassisted or assisted, only the elementary command sequences presented to its sensory inputs as a primitive identifier followed by an assembly identifier and ignores any different sequence presentation. In other words, whereas an ABNN organized in six node assemblies works properly unassisted in an environment with the said limitations, in the assisted learning, the teacher can always organize the examples presentation as a sequence composed by one primitive followed by a previously learned sequence of tasks.

5.4.3. *The six node primitive.*

In a six node primitive, see the Figure 5.2., the nodes one and two are the Hebbian-in input nodes. They receive a Hebbian connection from each of the six node assembly nodes number four and six and a non-Hebbian connection, with an associated coupling coefficient greater than the threshold, from the ABNN input nodes. The node number one sends an inhibitory connection and the node number two a facilitatory non-Hebbian connection with an associated coupling coefficient greater than the threshold to the node number three. In turn, the latter sends a non-Hebbian connection to the node number five and, with a two taus delay, to the node number four. The first is associated to a coupling coefficient greater than the threshold and the second is associated to a coupling coefficient equal to one half the threshold. That is, the node four computes the logical conjunction of the node three previous activity and of the node five present activity. The node four activity starts the effector associated to the very same primitive, see the chapter 5.3.. In turn, the node number five becomes active one tau after the node number three. The node number five, once activated by the node number three, computes the existential quantifier in the time domain, see also the chapter 1.3.1.3.. It is inhibited by the mask, see the chapter 5.4.3., and by the end of the activity signal sent by the effector it has previously activated. In turn, first it inhibits, then, with a two taus delay, activates the node number six. That is, as it is inhibited, either by the mask or by the end of the activity signal of its associated effector, the node number five activates, with a one tau delay, the node number six. Eventually, the latter is the primitive non-Hebbian node sending a non-Hebbian connection both to all the assemblies and to the mask.

That is, the primitive node six activity is the signal inhibiting the node five activity of all the assemblies. Every time the node four of a six node primitive activates an effector, the latter, at the end of its activity, inhibits the node five of the very same primitive.

5.4.4. *The six node primitive internal activity.*

The six node primitive nodes number one, respectively two, are activated by the very same primitive i-identifier, respectively identifier, once they are the ABNN input subvector value and by the Hebbian connections stemming out of the other assemblies nodes four and six, once the latter connections have been frozen. The node number three is activated, after one tau, by the node number two activity and is inhibited, after one tau, by the node number one activity. The node number four is activated two taus after the node number three and one tau after the activity of the node number five. That is, if the latter has not been inhibited. The node number four is non-Hebbian and its activity is the signal to activate the effector associated to the primitive. The node number five is activated, after one tau, by the node number three activity, if both the mask is inactive and there is not an end of the activity signal incoming from the effector associated to the primitive. Once activated, the node number five remains active until it is inhibited. Two taus after the node number five is inhibited, the node number six becomes active. The node number six activity is the primitive end of activity signal to the ABNN remaining IBNSs. We say a *six node primitive is active* either when its node number one is active or in the time interval comprised between the activity of its node number two and the activity of its node number six.

5.4.5. *The mask for the ABNN organized in six node IBNSs.*

To help during and speed up the training sessions, an ABNN organized in six node assemblies contains a *mask*, namely the subnet represented in the Figure 5.3.. To under-

stand its role, let us consider an ABNN organized in *six node assembly*, for short the six node ABNN, trained to compute the task represented by the sequence S of primitive tasks. Let us call P the effector controlled by the primitive P and t_S the time needed to execute S. To train the six node ABNN in the task S' represented by the sequence PS, the six node ABNN input subvector content must be P followed by S, a number of time dependent on the Hebbian parameter values. Let us call t_{PS} the smallest repetion cadence for the presentation of the sequence PS needed to freeze the relevant time dependent coupling coefficients. Until t_S is smaller than t_{PS}, there is no problem. Eventually, for given Hebbian parameters, as the six node ABNN autonomously learns to compute a longer sequence S", $t_{S"}$ becomes greater than t_{PS}. Accordingly, as the Hebbian parameters have been given at manufacturing time, either we limit the length of the sequences a six node ABNN can be trained to compute, or we find a way to inhibit the execution of the previously learned primitive sequences during the training sessions. As we find the first alternative of no interest, we shall discuss the second.

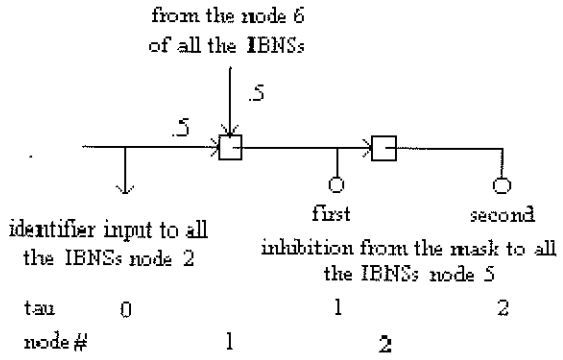


Fig. 5.3. *The mask for an ABNN organized in six node assemblies.* The singled barbed arrows represent the non-Hebbian connections: the unlabelled represent connections with associated coupling coefficients greater than the threshold. The labeled represent connections with associated coupling coefficients one half of the threshold. The empty circles represent the absolute inhibitory connections. The node one computes the logical conjunction of the node six of each assembly and of the six node ABNN sensory input, if the latter is an assembly identifier.

An obvious solution is to have a teacher inhibiting the tasks executions whenever the six node ABNN is in a training session. Instead, our solution is to find, first, some activation subsequence characterizing the training sessions. Second, to wire suitably the six node ABNN so it can be trained unassisted. For a six node ABNN organized in six node assemblies the activation sequence characterizing the training sessions is the sequence composed by an IBNS node six activation followed, after one τ, by the input representing an IBNS identifier. We have devised the subnet mask so wired to respond to such activation sequence. The mask sends two inhibitory signals to all the six node assemblies node number five, the first with a one τ and the second with a two τ delay after both the node number six of the assembly A is active and, simultaneously, the identifier A', different from A, is present at the six node ABNN sensory input. On the contrary case, namely if at the six node ABNN input is present the IBNS identifier I and, after one τ, there is an IBNS node six activation, the mask is activated. However, in this case the mask activation helps in dumping the I activity, without unduly interfering with the six node ABNN proper functioning.

5.4.6. *The six node IBNS pitfalls.*

To discuss the limitations of a six node ΛBNN, let us examine the constraints on the input sequences accepted by the six node ABNN. To freeze the relevant Hebbian connections, the six node ABNN input subvector content sequence must be of the type:

$$\Lambda, \tau, P, \tau, \tau, \tau, \tau, \tau, \tau, \tau, \tau, \tau, \Lambda', ...,$$

where the symbols Λ, Λ', P represent the six node ΛBNN input subvector given content at successive taus, τ represents the input subvector generic content at successive taus, Λ and Λ' represent different assemblies identifiers and P represents a primitive identifier. The Λ and Λ' identifiers must represent different assemblies, so to avoid training the six node ABNN in the execution of a loop, ie, in a circular behavior. In the following, we call Λ the *leading* identifier and Λ' the *trailing* identifier.

The second limitation the six node ABNN is subject to depends on the particular sequences the input subvector contents have to assume. That is, the six node ΛBNN accepts input sequences of the above given particular type, namely input sequences chosen in function of the six node assemblies wiring diagrams. Accordingly, an ABNN organized in six node assemblies does not have a convergence dependent more on the Hebbian parameter values and less on the component IBNS wiring diagrams. As it is the case for the ABNN organized in assemblies of greater complexity, we shall discuss in the chapter 5.6..

To present a third more fundamental limitation, let us consider the two primitive identifiers P and P'' and the leading Λ and trailing Λ' identifiers. To avoid endless loops in the six node ΛBNN behavior, it must be $\Lambda \neq \Lambda'$. However, let us discuss what happens when the Hebbian connection stemming out of the Λ node number six is frozen and the Hebbian connection stemming out of the Λ' node number six is not yet frozen. That is, let us consider an ABNN with both a frozen Hebbian connection between the Λ Hebbian-out node number six and the Λ' Hebbian-in node number one and a not yet frozen Hebbian connection stemming out of the Λ' node number six. First, the ABNN can be trained to accept the sequence:

$$\Lambda, \tau, P, \tau, \tau, \tau, \tau, \tau, \tau, \tau, \tau, \tau, \Lambda'.$$

Then, once it has been trained in the given sequence, it can be trained to accept the sequence:

$$\Lambda', \tau, P'', \tau, \tau, \tau, \tau, \tau, \tau, \tau, \tau, \tau, \Lambda,$$

ie, the ΛBNN can be trained to compute an endless loop. To avoid it, we need either a teacher, or a special purpose ΛBNN environment. Or, as we shall see in the chapter 5. 6., we have to implement in the assemblies a rule of the type: *if the Hebbian-in nodes belonging both to the assembly Λ and to the Hebbian set H receives the Hebbian connections coming out of the Hebbian-out nodes belonging both to the assembly B ≠ Λ and to the same Hebbian set H, then, the latter Hebbian connections, namely those terminating on the Hebbian-in nodes belonging both the assembly Λ and to the Hebbian set H, cannot be frozen before the Hebbian connections coming out of the Hebbian-out nodes belonging both to the same assembly Λ and to the same Hebbian set H.* Actually, whereas the former connections are, possibly some of, the assembly Λ Hebbian input connections, the latter are, possibly some of, its Hebbian output connections.

5.5. The sixteen node IBNS.

5.5.1. *The sixteen node assembly.*

A more complex IBNS is the *sixteen node assembly*: in the Figure 5.4. we give its wiring diagram and in the Table 5.1. its internal activity sequence. As the six node assembly, a sixteen node assembly is a subnet composed by a constant number of nodes. That is, the number of its nodes does not depend on the number of assemblies we connect to obtain an ABNN. In an ABNN organized with sixteen node assemblies, for short a *sixteen node ABNN*, the Hebbian connections stemming out of the Hebbian-out node sixteen of each assembly, or stemming out of the Hebbian-out node nine of each primitive, see 5.5.3., and impinging onto the Hebbian-in node nine and fifteen of the remaining assemblies are the ones trainable to control the sixteen node ABNN proper working. Whereas such Hebbian-in nodes guarantee the total number of nodes constancy in a sixteen node assembly, they are the reason of one of its weakness, see 5.4.4.. By contrast, the Hebbian connections stemming out of the Hebbian-out nodes four and ten of each assembly and impinging both onto the Hebbian-in nodes two and four of the remaining assemblies and onto the Hebbian-in nodes three and four of the primitives are the input trainable nodes. By increasing their number, a sixteen node assembly can have more than two Hebbian-out nodes impinging on the nodes two and four of the other assemblies, or on the nodes three and four of the primitives. The upper bound on the number of such Hebbian-out nodes depends on the Hebbian parameter values, see the discussion in 5.4.1..

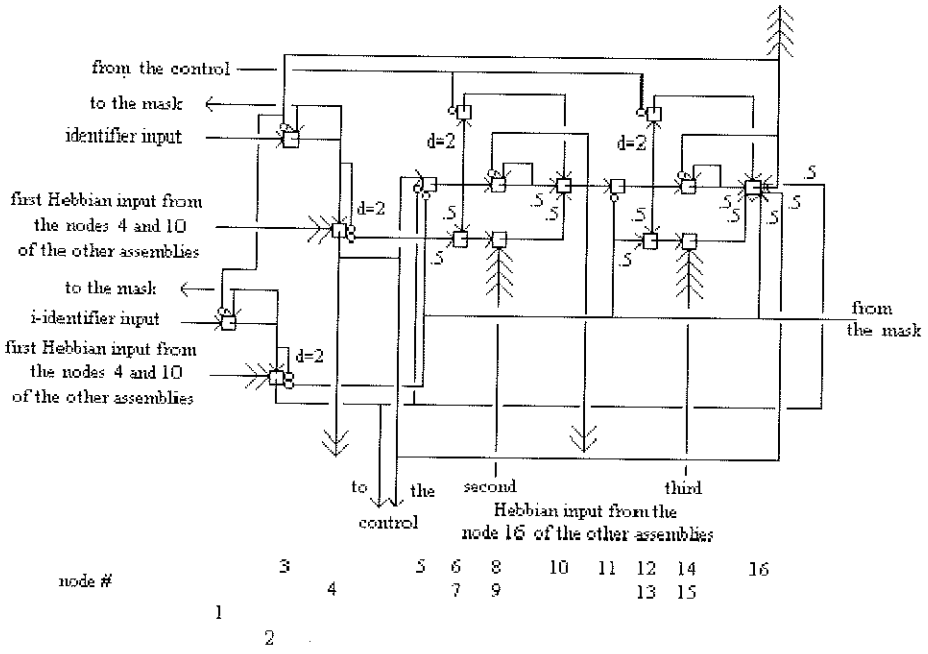


Fig. 5.4.. *The sixteen node assembly.* The empty circles represent the absolute inhibitory connections; the single barbed arrows represent the facilitatory connections; the double barbed arrows represent the Hebbian connections connecting the Hebbian-out nodes four and ten to the Hebbian-in nodes two and four; the quadruple barbed arrows represent the Hebbian connections connecting the Hebbian-out node sixteen to the Hebbian-in nodes nine and fifteen; the label $d = 2$ is associated to the connections with a two τs delay. The unlabelled connections have a one τ delay and the associated coupling coefficient is no smaller than the node threshold. The .5 label represents a coupling coefficient with a value one half of the threshold value. The node four is Hebbian-in and Hebbian-out; the nodes two, nine and fifteen are Hebbian-in; the nodes ten and sixteen are Hebbian-out. In the Table III is given the activation sequence of the sixteen nodes assembly. To speed up the sixteen node ABNN training, the net obtained by connecting together more sixteen node assemblies should contain two special purpose subnets: the mask, see the Figure 5.5., and the control, see the Figure 5.6.. To avoid too many lines in the figure, we have collapsed in one the two connections incoming from the mask.

In the Table 5.1., we have represented both the activity going on in an assembly with frozen coefficients as well as the activity going on in an assembly with not yet frozen coefficients. That is, we consider the activity of the nodes belonging to the three sixteen node assemblies A, A' and A", where A' and A" are the *offsprings* of A, the A assembly Hebbian-out connections are not yet frozen and the A' and A" Hebbian-out connections are frozen. Whereas the input signals are received by all the assemblies in the sixteen node ABNN, for simplicity sake, in the table last six rows we mention A, A' and A", only. Eg, in the row labeled *sensory input* to, only the assembly characterized by the identifier equal to the input subvector has been indicated. All the remaining assemblies have been omitted as the said input does not activate their input nodes. Together with the node and the input identifiers we give their Hebbian properties, also. In any assembly the nodes are activated as follows. One τ after the net receives either the i-identifier i-A or the identifier A, ie, one τ after the A node number one, or three, is active, the A' input arrives. That is, either there is an assembly with A' as identifier or, after some transient activity in some of the assembly A nodes, everything goes back to sleep. Two τ after the A input, the A node number two, or four, is active and broadcasts, to all the other assemblies, a signal through its Hebbian output connection and sends, through a non-Hebbian connection, a signal to the control subnet, see the Figure 5. 6.. Three τ after the A input, the A' first Hebbian-out output node, ie, either the A' node number two, or the A' node number four, is active, so reinforcing the Hebbian input connection stemming-out of the first Hebbian-out A node. At the same time the mask, see the Figure 5.5., is active, so inhibiting the node number two and four in all the remaining assemblies. In particular, inhibiting the node number two, or four, belonging to the assembly B, if there is one such assembly, connected to the A' assembly node number two, or node number four, by a frozen connection, so in effect inhibiting the B activity. If at the time zero the input subvector value was the i-identifier i-A, the A node number five is inhibited. Instead if at the time zero the input subvector value was the identifier A, both the A node number five, the mask and the control are active, in turn causing the A node number seven to become active at the time four and inhibiting the node number six. Both if at the time one the subvector value was either the i-identifier A' or the identifier A', at the time four, also the A' node number sixteen is active, so both terminating A' activity and reinforcing the connection between the A' node sixteen and the A node number nine, if the latter was the subvector value at the time zero. As the A nodes eight and nine are active simultaneously, namely at the τ five, the A node number ten becomes active at the time six. At the τ five arrives the A" input which activates, at the τ six, either the A" node number two, or four, and the carousel starts anew with the A" in place of A'. That is, either there is an assembly with A" as identifier or, after some transient activity in some of the assembly A nodes, everything goes back to sleep. After six τs the A" first Hebbian-out output node, ie, either the A" node number two, or the A" node number four, is active, so reinforcing the Hebbian input connection stemming out of the A node number ten, if at the time zero the input subvector value was the identifier A, and activating the control subnet, the next τ. At the same time, the mask is active, so inhibiting the node number two, or four, of the assembly B', if there is one such assembly, connected to the assembly A" node number two, or four, by a frozen connection, so in effect inhibiting the B' activity. At the same time, both the A node number eleven and the mask are active, in turn causing the A node number thirteen to become active eight τs after the A input. At the very same τ also the A" node number sixteen is active, so both terminating A" activity and reinforcing the Hebbian connection between the very same A" node number sixteen and the A node fifteen, as the latter becomes active the next τ, ie, at the τ nine. As the A nodes fourteen and fifteen are active simultaneously, namely at the τ nine, the A node sixteen becomes active at the τ ten, so terminating A activity and signaling it to all the remaining assemblies. The nodes six and twelve becomes active only if the A input is not followed, with the right delay, by

either the A' or the A" input and the respective Hebbian connections are not frozen. Actually, their purpose is to terminate the A activity whenever the said conditions are not satisfied.

TABLE 5. 1.

TIME NODE #	0	1	2	3	4	5	6	7	8	9	10
either 1 or 3 non-Hebbian	-	A	A,A'	A,A'	A,A'	A	A,A"	A,A"	A,A"	A	A
either 2 Hebbian-in or 4 Hebbian-in and Hebbian-out	-	-	A	A'	-	-	-	A"	-	-	-
5 non-Hebbian	-	-	-	A	-	-	-	-	-	-	-
6 non-Hebbian	-	-	-	-	-	-	-	-	-	-	-
7 non-Hebbian	-	-	-	-	A	-	-	-	-	-	-
8 non-Hebbian	-	-	-	-	-	A	A	-	-	-	-
9 Hebbian-in	-	-	-	-	-	A	-	-	-	-	-
10 Hebbian-out	-	-	-	-	-	-	A	-	-	-	-
11 non-Hebbian	-	-	-	-	-	-	-	A	-	-	-
12 non-Hebbian	-	-	-	-	-	-	-	-	-	-	-
13 non-Hebbian	-	-	-	-	-	-	-	-	A	-	-
14 non-Hebbian	-	-	-	-	-	-	-	-	-	A	A
15 Hebbian-in	-	-	-	-	-	-	-	-	-	A	-
16 Hebbian-out	-	-	-	-	A'	-	-	-	A"	-	A
TIME INPUT	0	1	2	3	4	5	6	7	8	9	10
sensory input to	A	A'	-	-	-	A"	-	-	-	-	-
from the mask to	-	-	-	A,A',A"	-	-	-	A,A',A"	-	-	-
from the control to	-	-	-	A,A',A"	A,A',A"	-	-	-	A,A',A"	-	-
Hebbian input to the node 2 or 4	-	-	A', A"	A,A"	-	-	A', A"	A,A'	-	-	-
Hebbian input to the node 9	-	-	-	-	A,A"	-	A,A'	-	-	-	A', A"
Hebbian input to the node 15	-	-	-	-	A,A"	-	A,A'	-	-	-	A', A"

Table 5.1.. *The sixteen node assembly execution sequence.* We have represented the activity of the nodes belonging to the three sixteen node assemblies A, A' and A", for the times given in the first and fifteen rows. We call the assemblies A' and A" the *offsprings* of A. We consider the A assembly Hebbian-out connections not yet frozen and the A' and A" Hebbian-out connections previously frozen.

Let us consider the sequence S ≡ S'S", ie, the sequence S composed by the sequence S' followed by the sequence S", where both S' and S" are finite sequences of primitive tasks the ABNN has been trained to compute in the past. The ABNN can be trained to compute the sequence S, first, by freezing the assembly A first Hebbian-out connection, namely the one stemming out of the A assembly node number two, and impinging onto the A' node number two. Second, by freezing the assembly A second Hebbian-out connection, namely the one stemming out of the A assembly node number eight, and impinging onto the A" node number two. Once the two said Hebbian connections have been frozen, every time the assembly A becomes active, either because it has been activated by another assembly or because its identifier has been the ABNN input subvector content, its activity is followed, first, by the A' activity, second, by the A" activity, once A' has terminated its

own activity. In turn, the assembly Λ terminate its own activity as soon as Λ" has terminated its own activity. In other words, the ABNN can be trained to control the task S execution, first, by training it to control the task S' execution, second, ie, once S' has been terminated, by training it to control the task S" execution.

The sixteen node ABNN filters from its input stream the sequences composed by an identifier followed, with the right delay, by an IBNS identifier, in turn followed, with the right delay, by an IBNS identifier. That is, the ABNN filters the sequences of the type ΛΛ'τττΛ", where the symbol Λ, Λ', Λ" and τ represent the sixteen node ABNN input subvector content at successive times. Obviously, as any command sequence can be always so presented, the nets organized in sixteen node assemblies can be universally trained. However, a sixteen node ABNN learns only the elementary command sequences so presented to its sensory inputs. In other words, whereas a sixteen node ABNN works properly unassisted in an environment with the said limitations, in the assisted learning, the teacher can always organize the examples presentation as a sequence of the type ΛΛ'τττΛ".

5.5.2. *The mask and the control for a sixteen node ABNN.*

To help during and speed up the training sessions, an ABNN organized in sixteen node assemblies contains a mask, namely the subnet represented in the Figure 5.5.. To understand its role, let us consider, again, a sixteen node ABNN trained to compute the task represented by the primitive task sequence S ≡ S'S", where both S' and S" are finite sequences of primitive tasks the ABNN has been trained to compute in the past. Let us call Λ' the IBNS controlling the execution of the task sequence S' and $t_{S'}$ the time needed to execute S'. To train the ABNN in the task S, the ABNN input subvector content must be the sequence ΛΛ', a number of times dependent on the Hebbian parameter values. Let us call $T_{\Lambda\Lambda'}$ the smallest repetion cadence for the presentation of the sequence ΛΛ' needed to freeze the relevant time dependent coupling coefficients. Until $t_{S'}$ is smaller than $T_{\Lambda\Lambda'}$, there is no problem. However, once given the Hebbian parameters, as we train the ABNN to compute the longer sequence S"', $t_{S'''}$ becomes greater than $t_{\Lambda\Lambda'}$. Accordingly, as the Hebbian parameters have been given at manufacturing time, either we limit the length of the sequences an ABNN can be trained to compute, or we find a way to inhibit the execution of the previously learned primitive sequences during the training sessions. As we find the first alternative of little interest, we shall discuss the second.

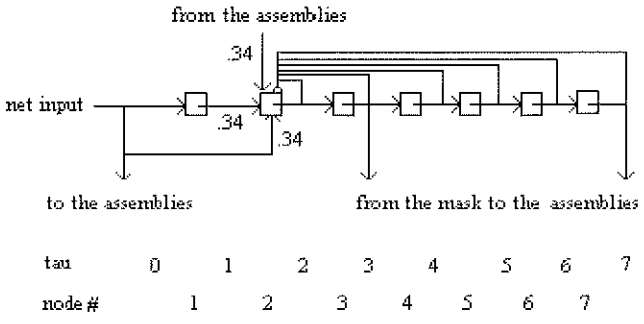


Fig. 5.5.. *The mask for an ABNN organized in sixteen node assemblies.* The empty circle represents the absolute inhibitory connection; the single barbed arrows represent the facilitatory connections; when there is no label near a connection, the associated coupling coefficient is no smaller than the threshold; when there is a 1/3 label, the coupling coefficient associated to the connection is one third of the threshold value. The mask node two becomes active if and only if it receives a signal from the net input and either from an assembly node one, or from an assembly node three, or from a primitive node one, or from a primitive node two. As explained in the text, the mask has two functions: first, it facilitates the training process by consenting its speed-up and, second, it guarantees the net functionality as it terminates the activity in the assemblies whenever the input conditions are not satisfied.

As we have seen for the sixteen node ABNN, see 5.4.3., the banal solution is to have a teacher inhibiting the tasks executions whenever the sixteen node ABNN is in a training session. Instead, our solution is to find, first, an activation subsequence characterizing the training sessions. Second, to wire suitably the sixteen node ABNN so it can be trained unassisted. For a sixteen node ABNN the activation sequence characterizing the training sessions is the sequence composed by either an assembly node one, or an assembly node three, or a primitive node one, or a primitive node two activation followed, after one τ, by the input representing an IBNS identifier. We call mask the subnet wired to respond to such activation sequence. The mask sends to all the sixteen node assemblies both two inhibitory signals to the node number two, four, five and eleven and two facilitatory signals to the node number seven, thirteen and sixteen. Moreover, the mask sends to all the sixteen node primitives both two inhibitory signals to the node number three, four and five and two facilitatory signals to the node number six.

The mask first signal has a one τ and the second has a two τ delay after both the node number either one or three of the assembly A, or the node number one or two of a primitive, is active and, simultaneously, the identifier A', different from A, is present at the sixteen node ABNN sensory input. On the contrary case, namely if at the sixteen node ABNN input is present the IBNS identifier I and, after one τ, there is an IBNS node two activation, the mask is activated. However, in this case the mask activation helps in terminating the I activity, without unduly interfering with the sixteen node ABNN proper functioning.

We give in the Figure 5. 6. the *control* subnet wiring diagram. The control task is to guarantee an assembly correct behavior once its training has been completed for only some, not all, of its Hebbian-out connections.



Fig. 5.6.. *The control.* The empty circle represents the absolute inhibitory connection; the single barbed arrow represents one facilitatory connection associated to a greater than the threshold coupling coefficient.

5.5.3. *The sixteen node ABNN primitive.*

A sixteen node ABNN primitive, see the Figure 5.7., has seven nodes. As a primitive is easier to control than a sixteen node assembly, it requires a smaller number of nodes.

The node number three and four are the primitive Hebbian-in input node. A primitive receives a Hebbian connection from each of the sixteen node assembly nodes number four and ten and a non-Hebbian connection from the sixteen node ABNN input nodes. A primitive input node one, two, sends a facilitatory non-Hebbian connection to the node number three, four, and to the mask. To these connections are associated coupling coefficients greater than the threshold. That is, the node number three, four, is activated by the node number one, two. The node four sends a facilitatory connection both to its associated effector, see the chapter 5.3., and to the node five, with an associated coupling coefficient greater than the threshold. The node three sends an absolute inhibitory connection to the node five. The node three and four are inhibited by the mask. In turn, the none number five becomes active two τ after the node number two. The node number five sends its non-Hebbian connections to the node number six and seven. The node six computes the conjunction of the node five and of the mask. The node seven computes the existential quantifier in the time domain, see also the chapter 1.3.1.3., is activated by the node five and is inhibited by the node nine. The Hebbian-out node number nine computes the conjunction of the node number seven and eight and inhibits the node seven, so resetting the primitive activity, moreover, it sends its Hebbian connections to the Hebbian-in node nine and fifteen of all the assemblies. The node eight computes the inclusive disjunction of the effec-

tor end of activity signal and of the node number six activity. That is, every time a seven node primitive activates an effector, the latter, at the end of its activity, activates the primitive node eight. In turn, the node number six computes the conjunction of the mask signal and of the node number five activity. The mask inhibits the nodes number three, four and five of a primitive.

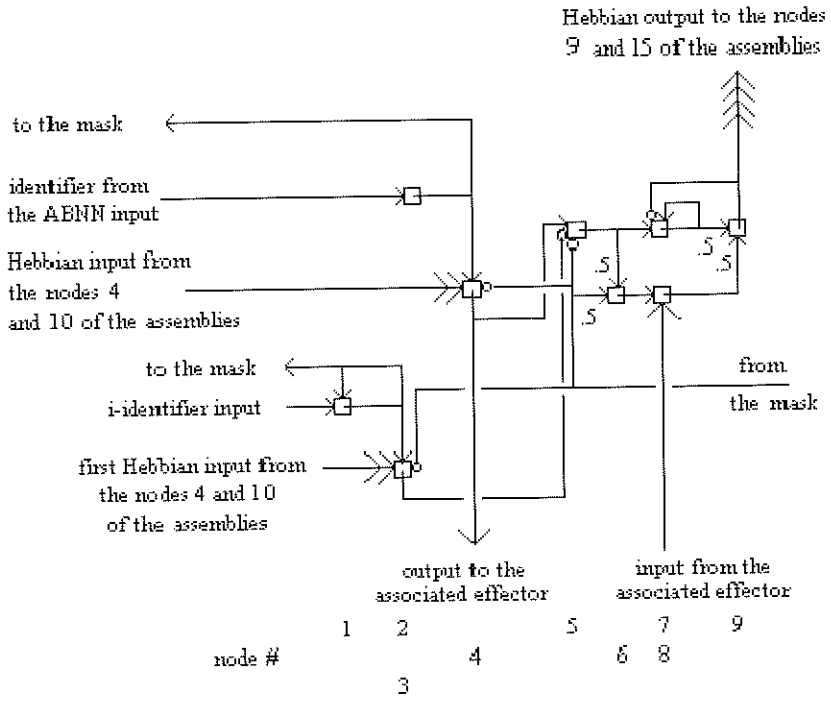


Fig. 5.7.. *The sixteen node ABNN primitive.* The multiple barbed arrows represent the Hebbian connections. The empty circles represent the absolute inhibitory connections. The single barbed arrows without labels represent the non-Hebbian excitatory connections with associated coupling coefficients greater than the threshold and one τ delay. The single barbed arrows with a .5 label represent connections with the associated coupling coefficients equal to one half of the threshold. The node three, four, receives a Hebbian connection from each assembly node four and ten, a facilitatory non-Hebbian connection from the node one, two, and it is inhibited by the mask activity.

5.5.4. *The sixteen node IBNS pitfalls.*

To discuss the limitations of a sixteen node ABNN, let us examine the constraints on the input sequences it accepts. To freeze the relevant Hebbian connections, the sixteen node ABNN input subvector content sequence must be of the type:

$$A, A', \tau, \tau, \tau, A'', ...,$$

where the symbols A, A', A" represent the sixteen node ABNN input subvector given content at successive taus and τ represents the sixteen node ABNN input subvector generic content at successive taus; A, A' and A" represent IBMS identifiers, or i-identifiers, and τ represents the input subvector generic content. If A, A' and A" are assembly identifiers, then, A and both A' and A" must represent different assemblies to avoid training the sixteen node ABNN in the execution of a loop, ie, in a circular behavior. By contrast, it can be A' $\equiv$ A", ie, A' and A" can represent the same IBNS. In conclusion, an assembly must be different from its two offsprings, see the chapter 5.3., and can have the very same as-

sembly both as first and as second son. In the following, we call A the *leading* and both A' and A" the trailing identifiers.

The last observation allows us to see the first limitation of an ABNN organized with sixteen node IBNSs. Namely, to train properly the sixteen node ABNN it is needed a sequence of examples, ie, an input sequence which does not contain two, or more, different examples having either the same leading or the same trailing identifiers. In other words, the environment events sequence generating the said input sequence must contains two, or more, times the very same events.

However, in general, the sixteen node ABNN input sequence shall contain more than once the same [leading, trailing] assembly identifiers. As the sixteen node ABNN input sequence cannot contain the very same leading assembly identifier followed by different trailing assembly identifiers, to be trained properly, the sixteen node ABNN cannot accept an input subvector sequence of the type:

$$A, A', ..., A", ..., A, A', ..., A", ... A, A', ..., A"', ...,$$

or of the type:

$$A, A', ..., A", ..., A, A"', ..., A", ... A, A', ..., A", ...$$

Neither the sixteen node ABNN can be trained to execute the sequence having the leading assembly identifier identical to the assembly identifier either of one of the sons of one its sons, or of one of the sons of one of the sons of one its sons, or ... . Actually, the difficulty is more apparent than real, as the very same environment event can be represented by many different assembly identifiers. That is, the sixteen node ABNN has learnes to compute one and the same function g identified either as g' or as g" or as ... . Then, whenever the sixteen node ABNN has to be trained in the computations of the function G, containing more than once the computation of the function g, it can be trained to compute G where the first instance of the function g is computed as g', the second instance is computed as g", and so on. In other words, the very same function g computation can be repeated many times, without any danger of the sixteen node ABNN entering in a loop. To this end, it is sufficient to employ a different name for each instance of the function g. Actually, this is possible because of the data representation in the sixteen node ABNN.

A second different limitation a sixteen node ABNN is subject to depends on the particular sequences the input subvector contents have to assume. That is, the sixteen node ABNN accepts input sequences of a particular type, only. In other words, an ABNN organized in sixteen node assemblies converges to the function g computation, if the examples arrive in the previously given sequence, only. A sixteen node ABNN does not have a convergence more dependent on the Hebbian parameters and less dependent on the assembly wiring diagram. As it is the case for ABNN organized in assemblies of greater complexity, we shall discuss in a few chapters.

As was the case for the six node ABNN, see 5.4.6., also the sixteen node ABNN has a far more fundamental limitation. Always, they must satisfy the rule: *the connections stemming out of the Hebbian-out nodes belonging both to the ABNN A and to the Hebbian set H must be frozen before of the connections terminating on the Hebbian-in nodes belonging both to the very same ABNN A and to the very same Hebbian set H.* The given rule can be satisfied either by following suitable training sequences or by implementing the very same rule in the assembly wiring diagram, for an example of the latter see 5.6.5. and the Figure 5.9.**B)**.

5.6. The general assembly.

5.6.1. *Introduction.*

The *general assembly*, see [6] and the Figure 5.8., is a sufficient condition to implement a Hebbian rule in a Boolean neural net, so to obtain an uniformly convergent and coherent ABNN, for short a *General ABNN*, or *GABNN*. The GABNN primitive, see [6], is given in the Figure 5.16..

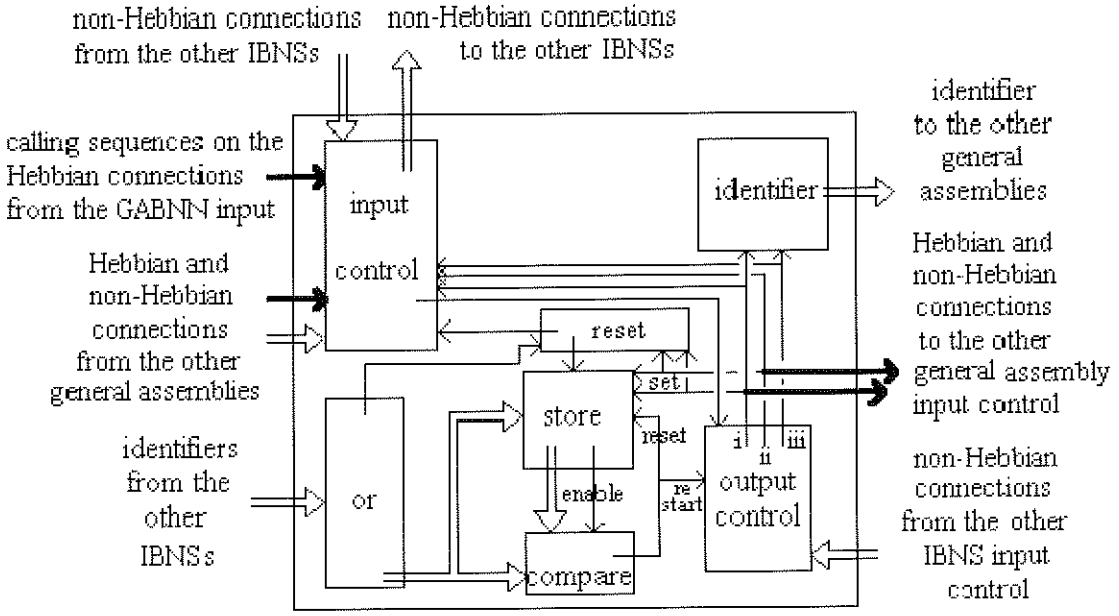


Fig. 5.8.. *The general assembly.* The light arrows represent the single non-Hebbian intra-general assembly facilitatory connections. The heavy arrows represent the Hebbian inter-general assembly connections, both those belonging to the first and to the second Hebbian set, ie, both those obeying the first and the second Hebbian rule. The hollow arrows represent more non-Hebbian inter-general assembly facilitatory and inhibitory connections.

Whereas in the assemblies so far discussed the input identifiers are wired-in at manufacturing time, with all the problems a solution of this kind entails, a general assembly sensory input identifier, actually a general assembly *calling sequence*, is the result of a training process. In other words, the connections embodying the general assemblies calling sequences are frozen at run time, ie, as the GABNN is trained to execute its various tasks. However, for a coherent GABNN, we must consider the number M of general assemblies present in the GABNN and the number n of components in the GABNN input subvector. Obviously, if 2^n is smaller than M, there are more general assemblies identified by the same calling sequence. In other words, to have a one-to-one correspondence between the general assemblies and the 2^n input subvector states, we must choose M no greater than 2^n. That is, if M is no greater than 2^n, we can have either a GABNN with both n Hebbian-out nodes corresponding to the input bitmap and n Hebbian-in nodes per general assembly, ie, either a GABNN with n + nM Hebbian nodes to store the general assemblies identifiers. Or a GABNN with both 2^n Hebbian-out nodes corresponding to the input bitmap and just one Hebbian-in node per general assembly, ie, or a GABNN with 2^n + M Hebbian nodes to store the general assemblies identifiers. To sum it up, for n greater than one, if

$M < [2^n - n][n - 1]^{-1}$ we have $n + nM < 2^n + M$: we call it the first case,
$M > [2^n - n][n - 1]^{-1}$ we have $n + nM > 2^n + M$: we call it the second case.

Accordingly, to have a GABNN with the smallest possible number of Hebbian nodes dedicated to store the general assemblies identifiers we must distinguish the two cases. As we shall see, if M is greater than $[2^n - n][n - 1]^{-1}$ the second Hebbian rule is sufficient, otherwise, ie, in the first case we need both the first and the second Hebbian rule, see the chapter 4..

As in both the six and sixteen node ABNNs, also in a GABNN the intra-IBNS connections are always non-Hebbian. However, in both the six and sixteen node assemblies the number of nodes does not depend on the number of IBNSs present in the ABNN. By contrast, in a GABNN the number of nodes depends on the number of IBNSs, as each general assembly has to store both its own and another IBNS identifier. That is, in a GABNN the number of nodes of each general assembly increases with the number of functions we want the very same GABNN to be trainable to compute. Whereas both the six and the sixteen node ABNNs accept only examples with structures in the time domain dependent on the assembly inner wiring, the GABNN accepts examples with a structure in the time domain dependent on the Hebbian parameters and on the maximum delay D. Eventually, in a GABNN the general assemblies are sorted in a total order. That is, apart the first and the last, in a GABNN the present general assembly has exactly one, the previous, general assembly preceding it and exactly one, the next, general assembly following it. The ordering is implicitly defined by some non-Hebbian connections between the IBNSs input control subnets, see the chapter 5.6.5.4. and the Figure 5.9.**D)**.

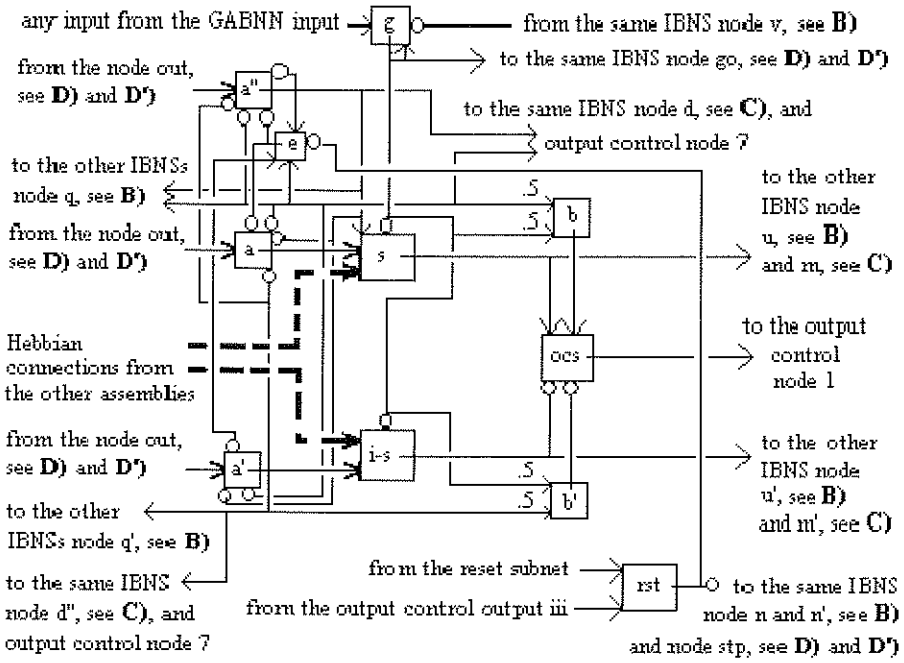
5.6.2. *The general assembly.*

A general assembly, see the Figure 5.8. for its wiring diagram and the chapter 5.6.3. for a discussion on its execution sequence, is a Boolean net composed by a number of nodes dependent on the total number of IBNSs we want to connect to form a GABNN. Because we need to associate an identifier, and an i-identifier, to each IBNS belonging to a GABNN and each general assembly has a store to hold one identifier, or i-identifier, the number of nodes per general assembly depends on the number of symbols, digits in particular, we need for each to identify an IBNS. That is, on the total number of IBNSs we want the ABNN to be composed of. Moreover, as a general assembly has to be trained to store its calling sequence, again the number of nodes needed for such a task depends on the number of general assemblies we want the GABNN to be composed of.

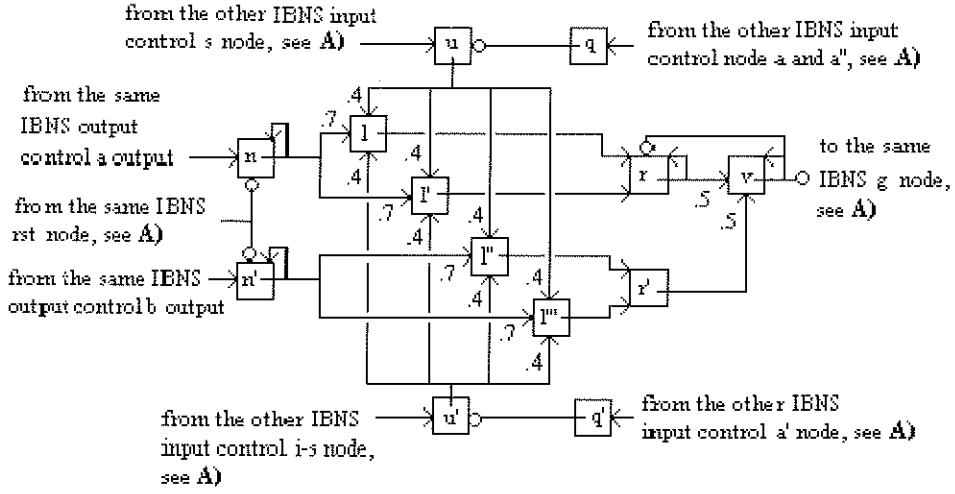
In the Figure 5.8. we have depicted a general assembly wiring diagram. Once it has received either the very same general assembly calling sequence from the net sensory inputs, through the Hebbian-in connections belonging to the first Hebbian set and obeying the first Hebbian rule, or a signal through one of its Hebbian-in connections belonging to the second Hebbian set, or both, a general assembly input control subnet, see 5.6.5. and the Figure 5.9., sends a start signal to the same general assembly output control subnet, see 5.6.6. and the Figure 5.10.. If the general assembly has both the Hebbian-out and the Hebbian-in connections, both belonging to the second Hebbian set, either frozen or not yet frozen, the output control i) sets the same general assembly store subnet, see 5.6.7. and the Figure 5.11.; ii) sends a signal to the same general assembly identifier subnet: however, as the latter wiring diagram depends on the algorithm generating the very same general assembly identifier it cannot be shown explicitly; iii) broadcasts a signal on the first Hebbian-out connection belonging to the second Hebbian set and obeying the second Hebbian rule, the i connection in the figure, to all the other IBNSs Hebbian-in input nodes belonging to the second Hebbian set and obeying the second Hebbian rule; v) starts the "reset" subnet, see 5.6.10. and the Figure 5.15.. Whenever the identifier subnet receives

a signal from the output control, it broadcasts the very same general assembly identifier to all the other general assemblies "or" subnets, see 5.6.8. and the Figure 5.12. As a general assembly "or" subnet receives an identifier, both it sends it both to the store and to the compare subnet, see 5.6.9. the Figure 5.13., and sends an inhibitory signal to the reset subnet, see 5.6.10. and the Figure 5.15. The first time the store receives an identifier, it memorizes it and sends an enabling signal to the compare subnet. Then, until it is not reset, each times it receives a new identifier, it sends the identifier it has memorized to the compare subnet. As the latter receives two identical identifiers, one from the or subnet and the other from the store subnet, it resets the store and restarts the output control. Otherwise nothing happens. The first time the output control is restarted by the compare subnet, it i) sets the store; ii) broadcasts a signal on the general assembly second Hebbian-out connection belonging to the second Hebbian set and obeying the second Hebbian rule, the ii connection in the figure, to all the other IBNSs Hebbian-in input nodes belonging to the second Hebbian set and obeying the second Hebbian rule; iii) starts anew the reset subnet. The second time the output control is restarted by the compare subnet, both it sends a signal to the identifier subnet and resets the input control, through the connection iii of the figure, so terminating the very same general assembly activity. Once it has been set on by the output control, by either its i or ii output connection, and unless the or subnet sends it an inhibitory signal within a constant time delay, dependent on the actual maximum delay of the inter-general assembly Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule, the reset subnet terminates the general assembly activity by inhibiting both the input control and the store subnet activity. If the general assembly has both frozen Hebbian-out connections and not yet frozen Hebbian-in connections belonging to the second Hebbian set and obeying the second Hebbian rule it computes the mask algorithm. That is, its input control, see the chapter 5.6.3.2. and the Figure 5.9.**C)**, activates the output control of the general assembly sons, see 5.6.6. and the Figure 5.10.**B)**. In other words, first, it inhibits its sons Hebbian-out connections belonging to the second Hebbian set and obeying the second Hebbian rule, second, terminates the sons activity.

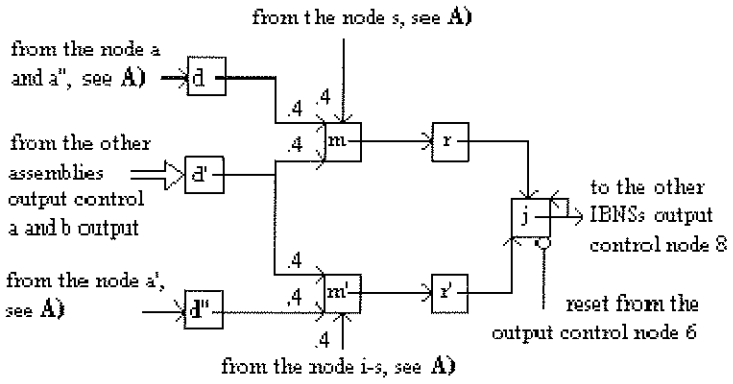
In a general assembly there are nodes belonging to two Hebbian sets: the *first Hebbian set is composed by the Hebbian-out nodes belonging to the GABNN input net and by the Hebbian-in nodes belonging to each general assembly*. The latter Hebbian connections are deputed to store the very same general assembly calling sequence, ie, the one the GABNN has to be trained on for each general assembly. The *second Hebbian set is composed by Hebbian-out nodes belonging to one general assembly and Hebbian-in nodes belonging to a different general assembly*, ie, having the Hebbian connections going from one to a different general assembly. The latter Hebbian connections are deputed to store the procedures the GABNN has to be trained to execute. For simplicity sake we call *Hebbian-in connections* and, respectively, *Hebbian-out connections* the Hebbian connections terminating on the Hebbian-in nodes belonging to the second Hebbian set, ie, stemming out of the Hebbian-out nodes belonging to second Hebbian set. As in each general assembly there are two Hebbian-out nodes belonging to the second Hebbian set, the Hebbian-out connections come in two subsets: the *first Hebbian-out connections* and the *second Hebbian-out connections*. Eventually, *a Hebbian connection belongs to the first, second, Hebbian set if it connects two Hebbian nodes belonging to the first, second, Hebbian set.*


any input from the GABNN input
g
from the same IBNS node v, see B)
to the same IBNS node go, see D) and D')
from the node out, see D) and D')
a''
e
to the same IBNS node d, see C), and output control node 7
to the other IBNSs node q, see B)
from the node out, see D) and D')
a
s
.5
.5
b
to the other IBNS node u, see B) and m, see C)
Hebbian connections from the other assemblies
ocs
to the output control node 1
from the node out, see D) and D')
a'
i-s
.5
.5
b'
to the other IBNS node u', see B) and m', see C)
to the other IBNSs node q', see B)
to the same IBNS node d'', see C), and output control node 7
from the reset subnet
from the output control output iii
rst
to the same IBNS node n and n', see B) and node stp, see D) and D')

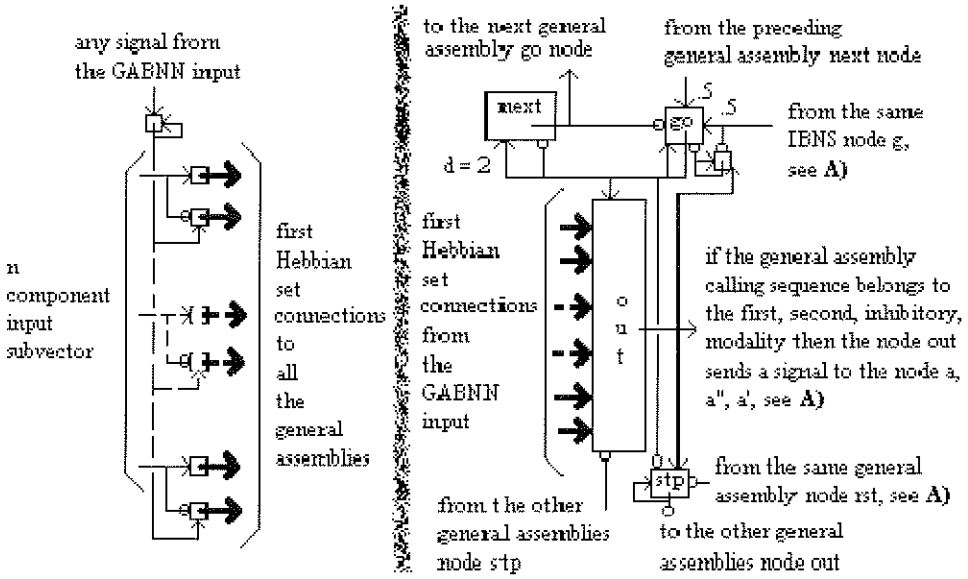
A)


from the other IBNS input control s node, see A)
u
q
from the other IBNS input control node a and a'', see A)
from the same IBNS output control a output
n
.7
.4
l
.4
.4
l'
.7
.4
.4
l''
.7
.4
.4
l'''
.7
.4
r
v
.5
.5
to the same IBNS g node, see A)
r'
from the same IBNS rst node, see A)
from the same IBNS output control b output
n'
from the other IBNS input control i-s node, see A)
u'
q'
from the other IBNS input control a' node, see A)

B)


from the node s, see A)
from the node a and a", see A)
d
.4
.4
.4
m
r
from the other assemblies output control a and b output
d'
to the other IBNSs output control node 8
j
.4
from the node a', see A)
d"
.4
m'
r'
reset from the output control node 6
.4
from the node i-s, see A)

C)


any signal from the GABNN input
n component input subvector
first Hebbian set connections to all the general assemblies
to the next general assembly go node
from the preceding general assembly next node
next
go
.5
.5
d = 2
from the same IBNS node g, see A)
first Hebbian set connections from the GABNN input
o u t
if the general assembly calling sequence belongs to the first, second, inhibitory, modality then the node out sends a signal to the node a, a", a', see A)
stp
from the same general assembly node rst, see A)
from the other general assemblies node stp
to the other general assemblies node out

D)

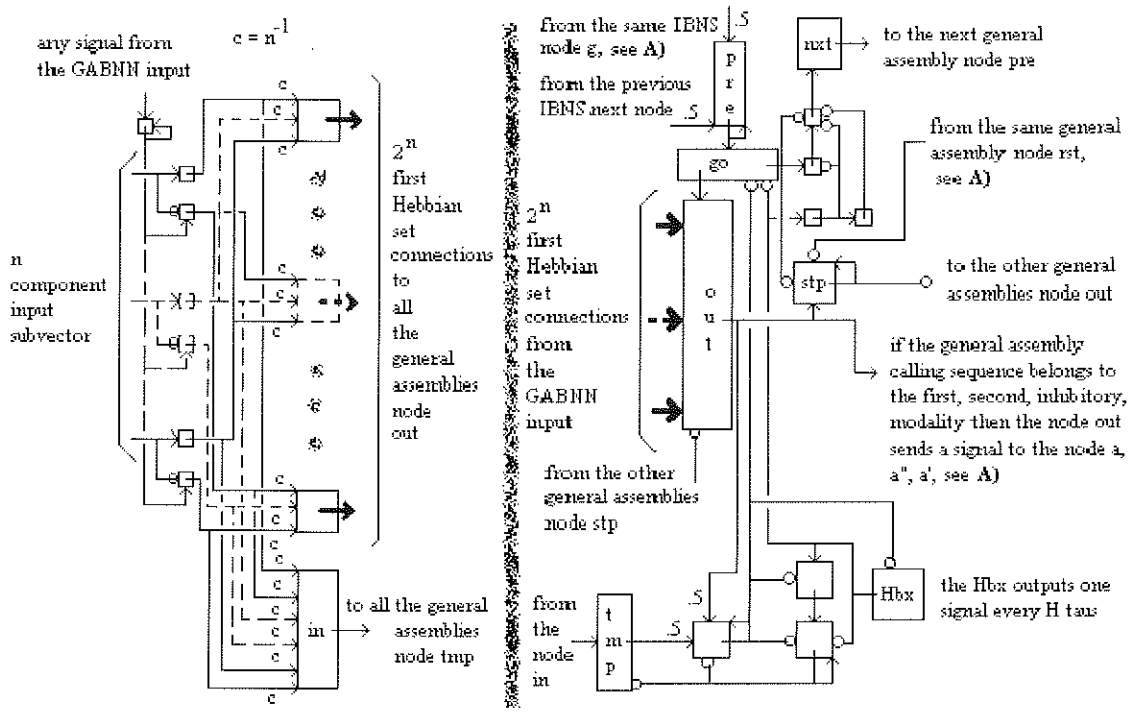


Fig. 5.9.. *The general assembly input control subnet.* The symbols within the nodes are the node identifiers. The heavy arrows represent the Hebbian connections both those belonging to the first and to the second Hebbian set, ie, both those obeying the first and the second Hebbian rule. The empty circles represent the absolute inhibitory connections; the single barbed light arrows represent the facilitatory connections: unless otherwise specified their associated coupling coefficients are greater than the node threshold and the delay is one tau; the heavy broken lines in **A)** represent the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule, they come in from the other general assemblies output control Hebbian-out nodes, see the Figure 5. 10.; the heavy lines in **D)** represent the Hebbian connections belonging to the first Hebbian set and obeying the first Hebbian rule.

In the Figure 5.9.**A)** we represent a general assembly input control box with just two Hebbian-in nodes, the s and the i-s nodes, belonging to the second Hebbian set. Each general assembly receives the first, the second and the inhibitory input modalities, see the chapter 5.3.. Analogously, whereas we could have given a general assembly with finitely many Hebbian-out connections belonging to the second Hebbian set, their number being bounded by the values we choose for the Hebbian parameters, here we discuss and in the Figure 5.8. we represent a general assembly with just two Hebbian-out connections belonging to the second Hebbian set. To increase the number of the Hebbian-out connections between the Hebbian nodes belonging to the second Hebbian set, it is sufficient to increase the number of its ii output connections in the output control subnet, see the Figure 5.10.**A)**. See also the chapters 5.4.1. and 5.5.1..

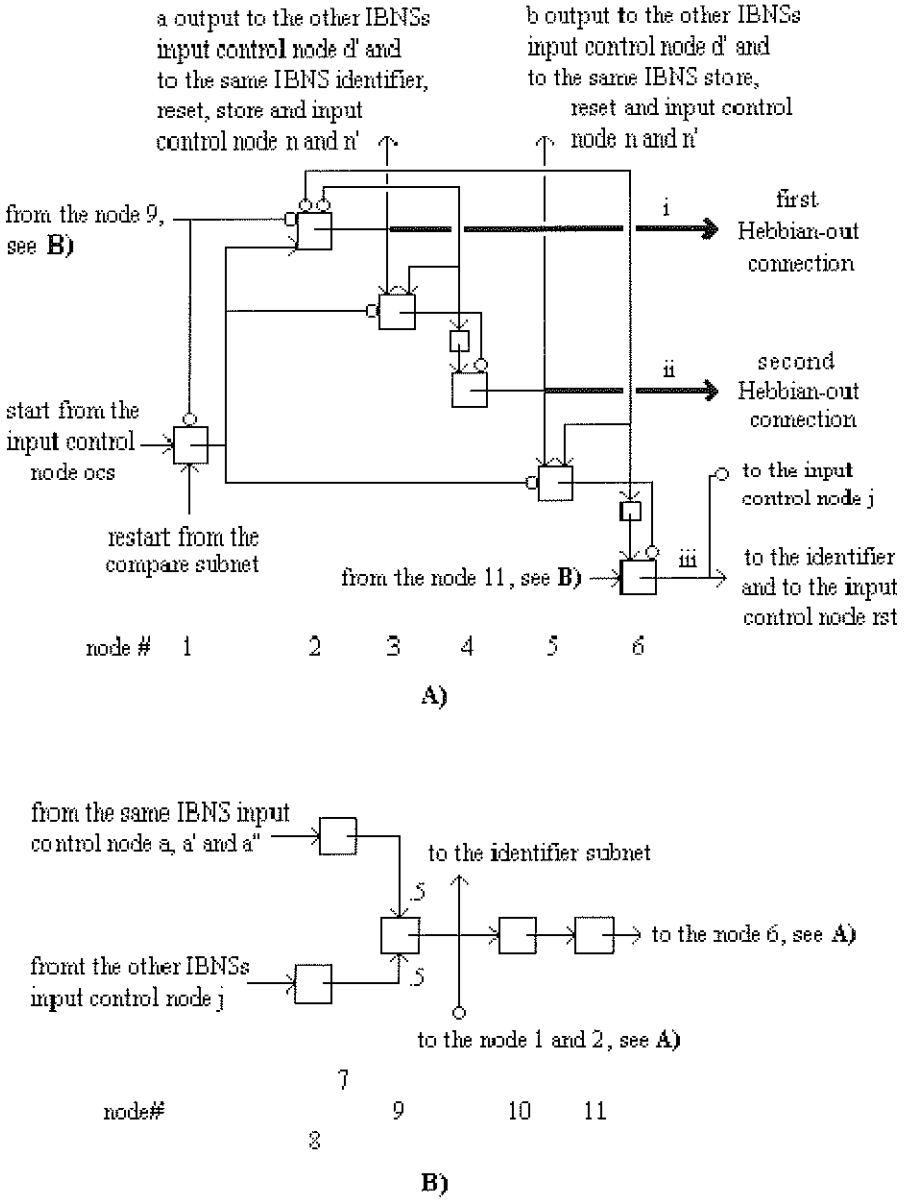


Fig. 5.10.. *The general assembly output control subnet.* The empty circles represent the absolute inhibitory connections; the single barbed arrows represent the facilitatory connections. Unless otherwise stated, the latter have associated coupling coefficients no smaller than the threshold. The heavy arrows represent the Hebbian-out connections belonging to the second Hebbian set and obeying the second Hebbian rule.

Eventually, to obtain a GABNN, we connect, first, each general assembly Hebbian-out nodes belonging to the second Hebbian set to all the other IBNSs Hebbian-in nodes belonging to the same second Hebbian set; second, the identifier box output of each IBNS to the input of all the general assemblies or boxes; third, the output of each general assembly input control box to the other IBNSs output control boxes; fourth, because just one general assembly at the time can be trained to store its own calling sequence, the general assemblies must be sorted and there must be a non-Hebbian connection between successive general assemblies to train them one at the time; fifth, we connect each general assembly Hebbian-in nodes belonging to the first Hebbian set to the GABNN input net Hebbian-out nodes, ie, to the only Hebbian-out nodes belonging to the first Hebbian set.

5.6.2.1. *The first case.*

In a general assembly first case the nodes belonging to the first, second, Hebbian set obey the first, second, Hebbian rule. That is, once a Hebbian connection belonging to the first Hebbian set has been frozen, the Hebbian-in node is no more a Hebbian node. By contrast, once a Hebbian connection belonging to the second Hebbian set has been frozen, the Hebbian-out node is no more a Hebbian node.

In the Figure 5.9.**D)** the node out represents a general assembly Hebbian-in node belonging to the first Hebbian set and deputed to store the very same general assembly identifier.

5.6.2.2. *The second case.*

In a general assembly second case the nodes belonging to the first and to the second Hebbian set obey the second Hebbian rule. That is, once a Hebbian connection belonging to either the first or the second Hebbian set has been frozen, the Hebbian-out node is no more a Hebbian node.

In the Figure 5.9.**D')** the node out represents a general assembly Hebbian-in node belonging to the second Hebbian set and deputed to store the very same general assembly identifier.

5.6.3. *The general assembly functioning.*

In a general assembly functioning we must distinguish two phases. To guarantee the respect of the necessary condition for the coherent implementation of the second Hebbian rule, see 4.3., a general assembly Hebbian-out connections belonging to the second Hebbian set have to be frozen before the same general assembly Hebbian-in connections belonging to the same second Hebbian set. Accordingly, if we want a coherent GABNN, the general assembly functioning has to change as soon as its Hebbian-out connections belonging to the second Hebbian set are frozen. To start with, as the GABNN is set on, ie, as just one component of its input subvector becomes active, in all the GABNN general assemblies the node g, see the Figure 5.9.**A)**, becomes active. The node g both computes the existential quantifier in the time domain, see 1.3.1.3., and inhibits the general assembly Hebbian-in nodes belonging to the second Hebbian set. That is, as long as the node g is active, the Hebbian-in nodes belonging to the second Hebbian set cannot be activated and the general assembly Hebbian-in connections belonging to the second Hebbian set cannot be frozen. To understand how to inhibit a general assembly node g, we must discuss the algorithms computed by a general assembly. First, the one computed when the Hebbian-out connections belonging to the second Hebbian set are not yet frozen, namely the *first phase algorithm.* Second, the one computed when the said Hebbian connections have been frozen, namely the *second phase algorithm.*

In any case, the general assembly has to be trained to store its own calling sequence, ie, to store the GABNN first input pattern, after the last one was stored in the preceding general assembly input control connections, having a repetition cadence compatible with the Hebbian parameter values. To this end, in each general assembly, apart the first and the last, the node next has one non-Hebbian connection to the node go of the next general assembly. Whereas the first general assembly node go is activated by the GABNN initial input activity, the last general assembly node next lacks the non-Hebbian output connection. Accordingly, just one Hebbian connection, at the time, from the first Hebbian set and just one Hebbian connection, at the time, from the second Hebbian set can be frozen. That is, just one general assembly at the time can be trained to store both a calling sequence and a logical connective to be executed by the GABNN.

5.6.3.1. *The first phase algorithm.*

When its Hebbian-out connections belonging to the second Hebbian set are not yet frozen, a general assembly computes the first phase algorithm. If at the time t either the first or the second modality calling sequence, for short the *first calling sequence* or the *second calling sequence* or the i-calling sequence, of the general assembly A is the GABNN input subvector value, at the next tau, ie, at the time t plus tau either the node a or a'', or a', belonging to A becomes active, see the Figure 5.9.**A)**. Because each general assembly is trained to store its own identifier, there is a one to one correspondence between the calling sequences, or the i-calling sequences, and the IBNSs and no more than one calling sequence at the time can be the GABNN input subvector value, see 5.3., no more than either one node a or a'', or a', can become active at any given time. Accordingly, the general assembly A executes the following first phase algorithm:

a) it inhibits its own input nodes, see the Figure 5.9.**A)**;

Comment. Actually, in the Figure 5.9.**A)** the node a, a'' and a' are mutually inhibiting each other. However the very same inhibitory signal can be sent to the GABNN input nodes, ie, to the GABNN nodes represented by the input subvector components. If we are considering a GABNN with multiple input modalities, see 5.3., it is sufficient to inhibit the GABNN input nodes belonging to the modality which activated A, ie, the nodes represented by the input subvector components associated to the said modality.

b) via the output control, see 5.6.6. and the Figure 5.10.**A)**, first, it predisposes its own store, see 5.6.7. and the Figure 5.11., to accept an IBNS identifier, second, it sets on the resets subnet, see 5.6.10. and the Figure 5.15., third, it sets on the identifier subnet;

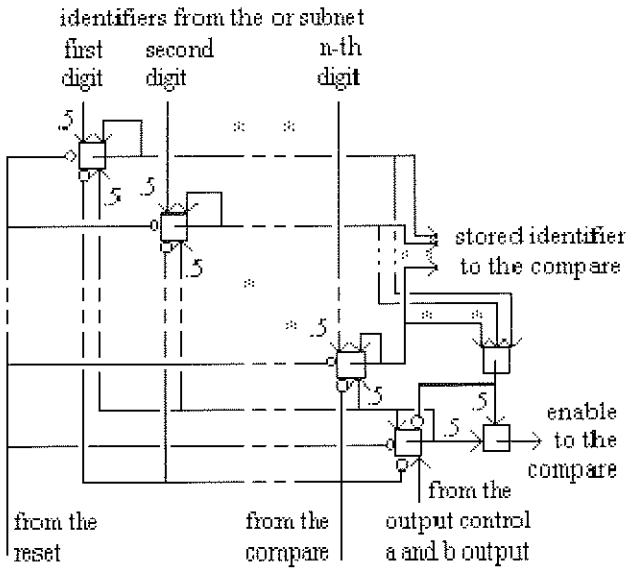


Fig. 5.11.. *The store subnet.* The empty circles represent the absolute inhibitory connections; the single barbed arrows represent the non-Hebbian facilitatory connections. Unless otherwise stated, the facilitatory connections have associated coupling coefficients no smaller than the threshold; the .5 label means the associated coupling coefficient is one half the threshold. The store subnet task is to store the identifiers broadcasted by the other general assemblies. It is enabled by the signal incoming from the output control subnet either a or b output, see the Figure 5.10.**A)**, and is reset by the signal incoming from either the compare subnet, see the Figure 5. 13., or the reset subnet, see the Figure 5. 15.. Both its outputs, the identifier it is storing and an enabling signal, go to the compare subnet.

c) its identifier subnet, see the Figure 5.8., broadcasts the A identifier to all the other general assemblies;

d) its first Hebbian-out connection, belonging to the second Hebbian set and stemming out of the output control, see 5.6.6. and the Figure 5.10.**A)**, becomes active;

e) after a constant delay, namely the delay wired in the reset subnet, e. g. a delay equal to the greatest delay D, see the chapter 1.3., if the general assembly A "or" subnet, see 5.6.8. and the Figure 5.12., does not receive another general assembly identifier, then A activity is terminated. Otherwise, if A receives the general assembly A' identifier, it stores it in its store subnet, see 5.6.7. and the Figure 5.11..

Comment. Because of the step b), in a GABNN the general assembly A is the only general assembly storing another general assembly A' identifier. Once it has done so, A waits until A' broadcasts its own identifier for the second time, ie, the parent waits until the offspring is reset to the initial conditions. By definition, we characterize the first offspring A' of a general assembly A as: *A' identifier is the first identifier received by the general assembly A no less than five and no more than D taus after its node a, or a', has been activated.*

f) all the time the general assemblies accept as their "or" subnet inputs, see 5.6.8. and the Figure 5.12., the identifiers broadcast by the primitives and by all the other general assemblies. As they receive the said identifiers, they compare them, see 5.6.9. and the Figures 5.13. and 5. 4.14., to the one they have stored in their respective store subnets. As a general assembly A receives an identifier equal to the one it has stored, i.e., as the parent receives for the second time the first offspring identifier, then A executes the following algorithm:

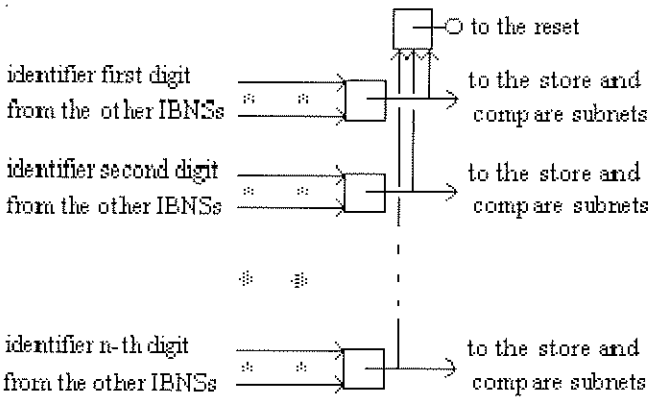


Fig. 5.12.. *The "or" subnet.* The single barbed arrows represent the non-Hebbian facilitatory connections; their associated coupling coefficients are no smaller than the threshold. We have assumed an identifier to be an n digit binary numeral.

a') it resets its own output control subnet, see 5.6.6. and the Figure 5.10.**A)**, and again, first, it predisposes its own store, see 5.6.7. and the Figure 5.11., to accept an IBNS identifier, second, it sets on the resets subnet, see 5.6.10. and the Figure 5.15.;

b') its second Hebbian-out connection belonging to the second Hebbian set and stemming out of the output control becomes active;

c') it executes once more the above given steps e) and f).

Comment. In a GABNN, the general assembly A" characterized by the identifier the general assembly A stores after it has reset, for the first time, its own output control subnet is A second offspring, see the chapter 5.3.. Because of the step c'), A is again the

only general assembly ready to accept and store a new, namely the second offspring, identifier. After that, A waits until A" broadcasts its own identifier for the second time, ie, A waits until its second offspring is reset to the initial conditions. By definition, we characterize the second offspring A" of a general assembly A as: *A" identifier is the first identifier received by the general assembly A no less than six and no more than D taus after the first offspring of A has broadcast, for the second time, its own identifier, ie, has terminated its own activity.*

d') All the time the general assemblies accept as their "or" subnet inputs, see 5.6.8. and the Figure 5.12., the identifiers broadcast by all the other IBNSs. As they receive the said identifiers, they compare them to the one they have stored in they respective store subnets, see the chapter 5.6.9. and the compare and n-xor subnet in the Figure 5.13. and 5.14., respectively. As A receives an identifier equal to the new one it has stored, i.e., as the parent receives the second offspring identifier, then A executes the following algorithm:

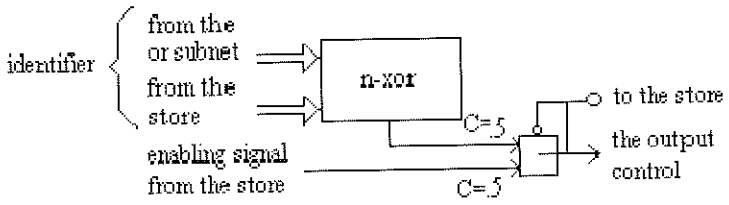


Fig. 5.13.. *The compare subnet.* Both the single barbed and the empty arrows represent facilitatory non-Hebbian connections, the latter represent the connections carrying the n digit identifiers, see the Figures 5.11. and 5.12; a .5 label means the associated coupling coefficient is one half of the threshold value; the empty circles represent the absolute inhibitory connections; the n-xor subnet wiring diagram is given in the Figure 5.14.. Almost continuously, the compare subnet receives the identifiers from the primitives and all the other general assemblies, represented in the figure by the upper empty arrow. As it receives both the identifier stored in the store subnet and the enabling signal, it compares the two identifiers. If they are equal and the store subnet is not empty, it sends the restarting signal to the output control node one, see the chapter 5.6.6. and the Figure 5.9.**A)**. Otherwise nothing happens.

a") the output control iii output connection, see 5.6.6. and the Figure 5.10.**A)**, becomes active. That is, first, it terminates the store subnet, see 5.6.7. and the Figure 5.11., activity; second, it activates the identifier subnet, see the Figure 5.8., so broadcasting the A identifier to all the other general assemblies; third, it sends a resetting signal to the input control subnet, see 5.6.5. and the Figure 5.9.**A)**, so resetting A to the initial conditions.

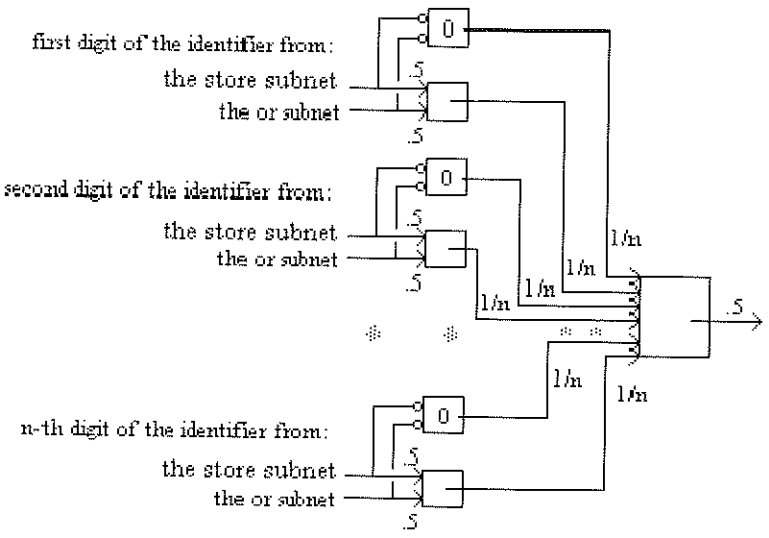


Fig. 5.14.. *The n-xor subnet.* The single barbed arrows represent the non-Hebbian facilitatory connections; the empty circles represent the absolute inhibitory connections; the labels near the connections give the associated coupling coefficient values. The nodes with a zero label have a zero threshold, ie, they are always active unless they are inhibited. They could have been substituted by nodes computing the logical negation of the output control node six activity, ie, of the node resetting the very same general assembly to the initial conditions, see the Figure 5.10.**A)**. The n-xor has an output signal if the identifier sent by the other IBNSs is identical to the one stored into the store subnet, see the Figure 5.1.., even if there is no incoming identifier and the store subnet is empty. Otherwise it has no output. Together with the enabling signal incoming from the store subnet, namely if the store subnet is not empty, the n-xor output activates the compare output node, see the Figure 5.13., which, in turn, sends the restart signal to the output control subnet, see the Figure 5.10.. Otherwise the n-xor output is ineffective.

5.6.3.2. *The second phase algorithm.*

When both its Hebbian-out connections belonging to the second Hebbian set are frozen, a general assembly computes the second phase algorithm. If the activity of each Hebbian-out connection belonging both to A and to the second Hebbian set is followed by the inactivity of the a, a'' and a' nodes of the remaining general assemblies and by the activity of one Hebbian-in node belonging to a general assembly different from A and to the second Hebbian set, then, the second phase algorithm initial transient, implemented in the net represented in the Figure 5.9.**B)**, is automatically computed by the general assembly A. To start with, in A the input control node v, see the Figure 5.9.**B)**, becomes active. Accordingly, in A whereas the input control node g is and shall remain inhibited, from the next tau on, the two Hebbian-in nodes s and i-s belonging to the second Hebbian set are no longer inhibited. Once A input control node g becomes inhibited, there are exactly two general assemblies, let us call them A' and A'', such that the first Hebbian-out node of A belonging to the second Hebbian set has its only frozen Hebbian-out connection to the Hebbian-in node of A' belonging to the second Hebbian set and the second Hebbian-out node of A has its only frozen Hebbian connection to the Hebbian-in node of A'' belonging to the second Hebbian set.

Comment. Once in the general assembly A the input control node g is inhibited, the Hebbian-in nodes of A' and A'' and the Hebbian-out node of A, all belonging to the second Hebbian set, as well as the connection linking them, are no longer Hebbian. Still, we continue to call them Hebbian, as they have been so identified and notwithstanding they are no longer time dependent entities. As said in 5.3., A' and A'' can compute the very same task sequence, however, both of them must be different from A.

Comment. The conditions given for A to compute the second phase algorithm initial transient could be a little relaxed. That is, we could consider the case of the node a, or a', of either A' or A'' being active after the activity in one of the Hebbian-out nodes of A, presently frozen, connections and one tau before the activity of one of the Hebbian-in nodes of either A' or A'', all belonging to the second Hebbian set. However, once in a general assembly the Hebbian-out and the Hebbian-in connections are frozen, the said behavior is exceptional.

Once the general assembly A two Hebbian-in nodes belonging to the second Hebbian set are no more inhibited, the coupling coefficients associated to its Hebbian-in connections start to vary, because of the Hebbian rule. Accordingly, to avoid to implement the mask subnet in the GABNN, we introduce two different algorithms. First A computes the mask algorithm until its Hebbian-in connections belonging to the second Hebbian set becomes frozen. Then it computes the steady state algorithm, once its Hebbian-in connections belonging to the second Hebbian set are frozen. Thanks to the mask algorithm, every time the calling sequence of A is the GABNN input subvector value, the Hebbian-out connections belonging both to the general assemblies different from A and to the second set and having their calling sequence as the GANN input subvector values, in particular its two offsprings Hebbian-out connections belonging to the second Hebbian set, are inhibited, to speed up the training sessions and guarantee unrestricted incremental learning, see the chapter 5.2.1. last paragraph and the next chapter 5.6.5.1.. The inhibition is terminated every time A sends out its identifier for the second time, ie, every time it is reset to the initial conditions. Instead, as soon as A Hebbian-in connections belonging to the second Hebbian set are frozen, A computes the steady state algorithm and does not inhibit its two offsprings Hebbian-out connections belonging to the second Hebbian set.

Now let us discuss the second phase algorithm, ie, the algorithm computed by a general assembly with frozen Hebbian-out connections belonging to the second Hebbian set, ie, with an inhibited g node. If at the time t either the first calling sequence or the second calling sequence, the i-calling sequence, of the general assembly A is the GABNN input subvector value, then, at the time t plus tau either the node a or the node a'', the node a', see the Figure 5.9.**A)**, belonging to A becomes active. Because there is a one to one correspondence between the calling sequences, or the i-calling sequences, and the IBNSs and no more than one calling sequence at the time can be present at the GABNN input subvector value, see 5.3., no more than either one node a or a'', or a', can become active at any given time. Then, the general assembly A executes the following second phase algorithm:

a) both it inhibits its own input nodes, see the Figure 5.9.**A)**, and if there is at least one active Hebbian-out connection belonging to the second Hebbian set, see the Figure 5.9.**C)**, coming-in from at least one general assembly A'''' $\neq$ A, we call it the *learning alternative*, after five taus A inhibits the Hebbian-out nodes belonging to the second Hebbian set of the other general assemblies;

Comment. Actually, in the Figure 5.9.**A)** the node a, a'' and a' are inhibiting each other. However the very same inhibitory signal can be sent to the GABNN input nodes, ie, to the GABNN nodes represented by the input sub-subvector components. If we are considering a GABNN with multiple input modalities, see 5.3., it is sufficient to inhibit the GABNN input nodes belonging to the modality which activated A, ie, the nodes represented by the input sub-subvector components associated to the said modality.

Comment. The A generated inhibition of the Hebbian-out connections belonging to the second Hebbian set in the other general assemblies is terminated as soon as A both is reset to the initial conditions and sends out, for the second time, its own identifier.

b) due to the learning alternative outcome, b) can be one of two different steps. Either the learning alternative is true for at least one general assembly different from A, ie, both at least one node a, or a', and one Hebbian-in connection, both belonging to the same general assembly different from A and to the second Hebbian set, are active simultaneously. Or either the learning alternative is false or it is true for A alone.

If the learning alternative is true for a general assembly different from A, then A executes the mask algorithm, see the Figure 5.10.**B**). That is, the A output control inhibits its own Hebbian-out nodes belonging to the second Hebbian set and sends one signal to the A identifier subnet and, after two taus, executes the first phase algorithm resetting step:

a'') the output control iii output connection, see 5.6.6. and the Figure 5.10.**A**), becomes active. That is, first, it terminates the store subnet, see 5.6.7. and the Figure 5.11., activity, second, it activates the identifier subnet, see the Figure 5.8., so broadcasting the A identifier to all the other general assemblies, third, it sends a resetting signal to the input control subnet, see 5.6.5. and the Figure 5.9.**A**), so resetting A to the initial conditions;

If either the learning alternative is false or it is true for A alone, see the Figure 5.10.**A**), A executes the steady state algorithm. That is, via the output control, see 5.6.6. and the Figure 5.10.**A**), first, A predisposes its own store, see 5.6.7. and the Figure 5.11., to accept an IBNS identifier, second, it sets on the resets subnet, see the chapter 5.6.10. and the Figure 5.15., third, it sets on its identifier subnet;

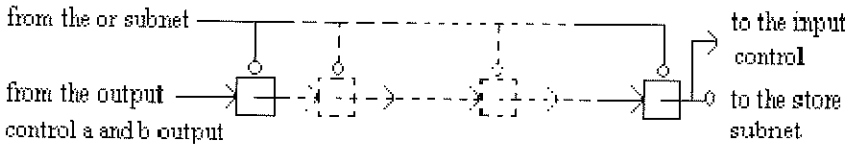


Fig. 5.15.. *The reset subnet.* The single barbed arrows represent the non-Hebbian facilitatory connections with associated coupling coefficients no smaller than the threshold; the empty circles represent the absolute inhibitory connections. The reset subnet is a delay line composed by as many nodes as the actual delay D of the general assembly Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule.

c) A identifier subnet, see the Figure 5.8., broadcasts the A identifier to all the other general assemblies;

d) A first Hebbian-out output connection both belonging to the second Hebbian set and stemming out of the output control, see 5.6.6. and the Figure 5.10.**A**), becomes active;

e) after a constant delay, namely the delay wired in the reset subnet, e. g. a delay equal to the greatest delay D, see 1.3., if the general assembly A "or" subnet, see 5.6.8. and the Figure 5.12., does not receive another general assembly identifier, then A is reset to the initial conditions. Otherwise, ie, if A receives the general assembly A' identifier, it stores it in its store subnet, see 5.6.7. and the Figure 5.11..

Comment. Because of the step b), in the GABNN the general assembly A is the only general assembly storing another general assembly A' identifier and, once it has done so, A waits until A' broadcasts its own identifier for the second time, ie, the parent waits until the offspring ends its own activity. By definition, we characterize the first offspring A' of a general assembly A as: *A' identifier is the first identifier received by the general assembly A no less than five and no more than D taus after its node a, or a', has been activated.*

f) all the time the general assemblies accept as their "or" subnet inputs, see 5.6.8. and the Figure 5.12., the identifiers broadcast by the primitives and by all the other general assemblies. As they receive the said identifiers, they compare them, see 5.6.9. and the

Figures 5.13. and 5.14., to the one they have stored in their respective store subnets. As a general assembly A receives an identifier equal to the one it has stored, i.e., as the parent receives for the second time the first offspring identifier, then A executes the following algorithm:

a') it resets its own output control subnet, see 5.6.6. and the Figure 5.10.**A)**, and again, first, it predisposes its own store, see 5.6.7. and the Figure 5.11., to accept an IBNS identifier, second, it sets on the resets subnet, see 5.6.10. and the Figure 5.15.;

b') its second Hebbian-out output connection both belonging to the second Hebbian set and stemming out of the output control becomes active;

c') it executes once more the above given steps e) and f).

Comment. In the GABNN the general assembly A" characterized by the identifier the general assembly A stores after it has reset, for the first time, its own output control subnet is A second offspring, see the chapter 5. 3.. Because of the step c'), A is again the only general assembly ready to accept and store a new, namely the second offspring, identifier. After that, A waits until A" broadcasts its own identifier for the second time, ie, A waits until its second offspring is reset to the initial conditions. By definition, we characterize the second offspring A" of a general assembly A as: *A" identifier is the first identifier received by the general assembly A no less than six and no more than D taus after the general assembly A first offspring has broadcast, for the second time, its own identifier, ie, has terminated its own activity.*

d') All the time the general assemblies accept as their "or" subnet inputs, see 5.6.8. and the Figure 5.12., the identifiers broadcast by all the other IBNSs and compare them to the one they have stored in they respective store subnets, see 5.6.9. and the compare and n-xor subnet in the Figure 5.13. and 5.4.14., respectively. As A receives an identifier equal to the new one it has stored, i.e., as the parent receives the second offspring identifier, then A executes the following algorithm:

a") the output control iii output connection, see 5.6.6. and the Figure 5.10.**A)**, becomes active. That is, first, its store subnet, see 5.6.7. and the Figure 5.11., activity is terminated; second, it activates the identifier subnet, see the Figure 5.8., to broadcast the A identifier to all the other general assemblies; third, it sends a resetting signal to the input control subnet node rst and node j, see 5.6.5. and the Figure 5.9.**A)** and **C)**, so resetting A to the initial conditions.

As it is easy to verify, in a GABNN, the activity of one Hebbian-out node belonging to the second Hebbian set is followed, at most, by the activity of one Hebbian-in node belonging to the same second Hebbian set. And they are connected by a Hebbian connection. Accordingly, once the training has been suitably completed, i.e., once some of the Hebbian connections have been frozen, just one primitive at the time can be active.

It is quite simple to train an adaptable net on the illustrative example given in the chapter 5.2.1., and so to verify its correct behavior as predicted, i.e., to verify it is learning to respond, to the literal commands, with the correct numerical strings without entering in loops. As we shall discuss with more details later, it is a very elementary, perhaps at the time tedious, task to train the net to control the execution of very long sequences of primitive.

In our simulations, see [18, 19] we have verified the adaptable Boolean neural net behavior under different input conditions, as when it interacts with environments characterized by different noise to signal ratios, in other words, with different mixes of random and correlated events, where the correlated events are the ones the net has to learn. Actually, the number of training sessions increases exponentially as the frequency of the random events increases. That is, for input streams composed by up to 80% of random events and 20% of correlated events, the number of training sessions increases up to three times, with respect to the number of training sessions necessary when the net input is

composed of the correlated events, only. Actually, this result illustrate quantitatively the difference between assisted and unassisted learning.

The adaptable Boolean neural net simulation is quite heavy as its coupling coefficient matrix contains both the time independent and the time dependent elements, where the latter have to be adjourned at each tau. Moreover, the coupling coefficient matrix has in the order of one hundred times more elements than the matrix of a non adaptable Boolean neural net computing the same functions, because of the greater number of connections we need to define a general assembly.

5.6.4. *The general assembly wiring diagram.*

The general assembly blueprint, see the Figure 5.8., is composed by seven modules: the *input control* subnet, see the Figure 5.9., the *identifier* subnet, the *or* subnet, see the Figure 5.12., the *store* subnet, see the Figure 5.11., the *compare* subnet, see the Figures 5. 13. and 5.14., the *output control* subnet, see the Figures 5.10.**A)** and 5.10.**B)**, and the *reset* subnet, see the Figure 5.15..

The input control subnet receives i) the GABNN inputs carrying the first calling sequence, the second calling sequence and the i-calling sequence characterizing the general assembly, actually the said calling sequences are received by the general assembly Hebbian-in nodes belonging to the first Hebbian set, ie, by the Hebbian-in nodes obeying the first Hebbian rule; ii) the Hebbian-in connections belonging to the second Hebbian set from the other IBNSs, ie, the Hebbian-in connections obeying the second Hebbian rule; iii) a number of non-Hebbian connections, from the other IBNSs, to control the freezing of the general assembly Hebbian-in connections belonging to the second Hebbian set, ie, to guarantee the latter are frozen only after the very same general assembly Hebbian-out connections belonging to the second Hebbian set have been frozen; iv) one facilitatory and one inhibitory non-Hebbian connection from the output control node number six; v) one non-Hebbian connection from the reset subnet; vi) inhibitory non-Hebbian connections coming in from the other general assemblies to control the proper freezing of the first Hebbian connections. In turn, the input control sends a number of non-Hebbian connections to the same and to the other IBNS output control subnets. As we can see in the Figure 5.9.**A)**, the node s activates the node ocs, the node i-s inhibits it. Once the very same general assembly Hebbian-out connections belonging to the second Hebbian set are frozen, to freeze the general assembly Hebbian-in connections belonging to the second Hebbian set and terminating on the node s, i-s, either the node a or the node a'', or the node a', activates the s node, the i-s node, namely one of the general assembly Hebbian-in nodes belonging to the second Hebbian set. The node g computes the existential quantifier of the GABNN input subvector values, ie, it is activated the first time one of the GABNN input subvector components assumes the value one. It is inhibited, once the general assembly Hebbian-out connections belonging to the second Hebbian set have been frozen, by the node v belonging to the subnet given in **B)**. That is, the node g is inhibited once the activity of both the general assembly Hebbian-out connections belonging to the second Hebbian set is followed by the activity of either the a, or a'', or a', nodes of at least one of the remaining IBNSs and, the next tau, by the activity of the s, or i-s, nodes of at least one of the remaining IBNSs. In the Figure 5.9.**C)**, we give the BNN computing the conditions to activate the mask algorithm, see the Figure 5.10.**B)**. That is, if the activity of either the input control node a, or a'', or a', belonging to the general assembly A and the activity of the non-Hebbian output control a and b output belonging to the remaining general assemblies is followed, the next tau, by the activity of the input control node s, or i-s, belonging to A, then, the node j sends a signal to the other general assemblies output control node number eight to inhibit their Hebbian-out nodes belonging to the second Hebbian set and obeying the second Hebbian rule.

The two wiring diagrams **D)** and **D')** are alternative and correspond to the first and to the second case, respectively, see the chapter 5.6.1.. Let us call M the number of general assemblies present in the GABNN and n the number of the input subvector components, the first case should be chosen for $M < [2^n - n]ln - 1]^{-1}$ and the second case should be chosen for $M > [2^n - n]ln - 1]^{-1}$, so to optimize the number of the first Hebbian set nodes, ie, the number of Hebbian nodes storing the general assemblies identifiers. In the Figure 5.9.**D)**, at the left, we have represented the GABNN input net, ie, a net transforming the input bitmap of the appropriate modality in the general assemblies calling sequences. It contains the n Hebbian-out nodes belonging to the first Hebbian set and obeying the first Hebbian rule. The said Hebbian-out nodes are activated in a one-to-one correspondence by the GABNN input nodes of the appropriate input modality. They send their Hebbian connections obeying the first Hebbian rule to all the general assemblies Hebbian-in node "out" belonging to the first Hebbian set. At the right, we have represented a generic general assembly input subnet, ie, the subnet trainable to store the very same general assembly calling sequence. The input subnet contains the general assembly Hebbian-in node, the node out, belonging to the first Hebbian set and obeying the first Hebbian rule. The input subnets as well as nodes out corresponding to the different input modalities have been collapsed into one for simplicity sake. The connections belonging to the first Hebbian set, represented by the heavy lines, are trainable to store the general assembly calling sequence, ie, to store the GABNN first incoming input pattern with a repetition cadence compatible with the Hebbian parameters. Actually, the connections belonging to the first Hebbian set connect the GABNN input net to each general assembly input subnet and obey the first Hebbian rule. The non-Hebbian connections coming out of the node next of a general assembly input box and terminating on the node go of a different general assembly input box are set at manufacturing time and induce a total order among the general assemblies. Once the present node out training has been completed and the present general assembly identifier is the GABNN input, the node stp inhibits the nodes out of all the remaining general assemblies, to prevent a second general assembly to be trained to respond to the very same calling sequence the present general assembly has been trained on. Eventually, the input control box activity is reset from the same general assembly output control node number six, ie, when the very same general assembly is reset to the initial conditions.

In the Figure 5.9.**D')**, at the left, we have represented the GABNN input net, ie, a net transforming the input bitmap of the appropriate modality in the general assemblies calling sequences. It contains the 2^n Hebbian-out nodes belonging to the first Hebbian set and obeying the second Hebbian rule. The activity of one of the said Hebbian-out nodes corresponds to one of the GABNN input subvector activity configurations. Each of them sends its Hebbian connections, obeying the second Hebbian rule, to all the general assemblies Hebbian-in node "out" belonging to the first Hebbian set. That is, each node out receives 2^n Hebbian-in connections. At the right, we have represented a generic general assembly input subnet, ie, the subnet trainable to store the very same general assembly calling sequence. The input subnet contains the general assembly Hebbian-in node, the node out, belonging to the first Hebbian set and obeying the second Hebbian rule. The input subnets as well as the nodes out corresponding to the different input modalities have been collapsed into one for simplicity sake. The connections belonging to the first Hebbian set, represented by the heavy lines, are trainable to store the general assembly calling sequence, ie, to store the GABNN first incoming input pattern with a repetition cadence compatible with the Hebbian parameters. Actually, the connections belonging to the first Hebbian set connect the GABNN input net to each general assembly input subnet and obey the second Hebbian rule. The non-Hebbian connections coming out of the node next of a general assembly input box and terminating on the node pre of a different general assembly input box are set at manufacturing time and induce a total order among the general assemblies. The box Hbx has an output signal every H taus, where H depends on the

values chosen for the Hebbian parameters. Every time the box Hbx has an output signal, thanks to the node tmp, the net automatically checks whether or not one of the Hebbian-in connections on the node out has been frozen. That is, whether or not the very same general assembly has stored its own identifier. Once the present node out training has been completed and the present general assembly identifier is the GABNN input, the node stp inhibits the nodes out of all the remaining general assemblies, to prevent a second general assembly to be trained to respond to the very same calling sequence the present general assembly has been trained on. Eventually, the input control box activity is reset from the same general assembly output control node number six, ie, when the very same general assembly is reset to the initial conditions.

The identifier subnet receives three non-Hebbian connections from the output control subnet: one from the node number two, one from the node number six and the third from the node number nine. A fixed number of taus after one of them is active, the identifier subnet broadcasts the general assembly identifier to the or subnet of all the GABNN remaining IBNSs. The fixed delay required by the identifier subnet to broadcast the general assembly identifier depends on its wiring diagram. In other words the identifier output is a fixed set of signals: they are outputted every time the identifier subnet receives one of its input signals. As it has many equivalent wiring diagrams, we leave it to the interested reader to draw one.

The or subnet inputs are the identifier subnet outputs from all the GABNN remaining IBNSs. Every time the or subnet receives one identifier it both sends the received identifier both to the store and to the compare subnet and one inhibitory signal to the reset subnet, so resetting it.

The store subnet is reset by the signal incoming from the reset and, independently, from the compare subnet. One independently of the others, the node number two, the node number four and the node number nine of the output control subnet enable the store subnet to store just one identifier. Namely the first identifier sent by the GABNN remaining IBNS or subnets after the said output control nodes activity. In turn, the store subnet both enables and sends the stored identifier to the compare subnet.

The compare subnet receives the IBNS identifiers from both the or and the store subnets. If it is enabled by the store subnet, it sends a resetting signal to the latter and the restarting signal to the output control subnet.

When a general assembly computes either the first phase or the steady state algorithms its output control subnet is started by the input control and restarted by the compare subnet. Both its nodes number two and four send non-Hebbian connections to the same general assembly reset and store subnets and to the other general assemblies input control subnet. Both its nodes number two and four send Hebbian connections belonging to the second Hebbian set to the GABNN remaining IBNS or subnets. That is its nodes number two and four are both non-Hebbian and Hebbian-out. Its node number nine sends one non-Hebbian connection to the input control subnet. Its node number six sends one non-Hebbian connection to the identifier subnet and to the input control subnet. The node six activity reset to the initial conditions the general assembly. When a general assembly computes the mask algorithm its output control subnet is started by the input control subnet. Then the control subnet does not activate its Hebbian-out nodes belonging to the second Hebbian set, ie, the ones obeying the second Hebbian rule. Instead, first it activates the identifier subnet, second, the node number six, so resetting to the initial conditions the general assembly.

As we can see in the Figure 5.10.**A**), in the output control subnet the node one input is the output of the input control subnet node ocs, see the Figure 5.9.**A**); the restart input is the output of the compare subnet, see the Figure 5.13.. From the first and from the second Hebbian-out node belonging to the second Hebbian set and obeying the second Hebbian rule stem the general assembly Hebbian-out connections. Their delays have given maximum values: the greatest of the latter determines the time the general assembly reset

subnet waits for an incoming identifier, see the Figure 5.15.. The a and b outputs go to the store subnet, see the Figure 5.11., to the reset subnet, see the Figure 5.15., and both to the same and to the other IBNS input control subnet, see the Figure 5.9.. Moreover, the a output goes to the same assembly identifier subnet. As we have represented in the Figure **5.10.A)**, once it becomes active, the node number one both activates the first Hebbian-out node belonging to the second Hebbian set, ie, activates a node obeying the second Hebbian rule and inhibits the nodes number three and five. The next tau , the first Hebbian-out node belonging to the second Hebbian set and obeying the second Hebbian rule: i) activates the node three, ii) sends a signal on a non-Hebbian connection to both the store, the identifier and the input control subnets of the very same general assembly and to the input control subnet of the other IBNSs, iii) sends a signal on the Hebbian-out connections belonging to the second Hebbian set and obeying the second Hebbian rule to the GABNN remaining IBNSs. The next tau, ie, three taus after the start signal arrives, the node three becomes active and remains active until a new signal from the node one, ie, from the compare subnet arrives. As the node number three becomes active, i) it inhibits the first and the second Hebbian-out nodes belonging to the second Hebbian set and obeying the second Hebbian rule, ii) it sends, with a two taus delay, a facilitatory signal only to the second Hebbian-out node, namely the one belonging to the second Hebbian set and obeying the second Hebbian rule. As the node three is inhibited by the node number one new activity, ie, three taus after the new signal arrives upon the node one, the second Hebbian-out node belonging to the second Hebbian set and obeying the second Hebbian rule becomes active. The next tau, the second Hebbian-out node belonging to the second Hebbian set and obeying the second Hebbian rule both activates the node five and sends a signal to the store subnet of the very same general assembly and, through the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule, to the primitives and the other general assemblies belonging to the GABNN. The next tau, ie, five taus after the new signal arrives upon the node number one, the node five becomes active and remains active until a new signal from the node number one, ie, from the compare subnet arrives. As the node number five becomes active both it inhibits both the first Hebbian-out node belonging to the second Hebbian set, namely the node obeying the second Hebbian rule, and the node six and sends, with a two taus delay, a facilitatory signal to the node six. As the node five is inhibited by the node one new activity, ie, three taus after a new signal arrives, for the second time, upon the node one, the node six becomes active. As the node six becomes active, it sends both a facilitatory signal to both the identifier and the input control subnets, so resetting them and resetting to the initial conditions the general assembly, and an inhibitory signal to the same general assembly input control subnet node j, see the Figure 5.9.**C)**. The BNN given in the Figure 5.10.**B)** computes the mask algorithm, see the chapter 5.6.5.2. and the Figure 5.9.**C)**. That is, if both the same general assembly node a or a'' or a' and at least another general assembly node j are simultaneously active, then, after one tau, the same general assembly output control nodes one and two are inhibited and the same general assembly identifier subnet is activated. And after two taus, the output control node six is activated, so resetting the same general assembly to the initial conditions without ever activating its sons.

The reset subnet receives a non-Hebbian facilitatory connection from both the node two and four of the output control subnet and a non-Hebbian inhibitory connection from the or subnet. It sends non-Hebbian connections to the store and to the input control subnets. The latter connections activity reset to the initial conditions the general assembly.

Instead of storing the general assembly identifiers we could have implemented the following alternative solution. In each general assembly we could have eliminated the identifier, the control and the or boxes, see the Figure 5.8., and substituted the store box, see the chapter 5.6.7. and the Figures 5.8. and 5.11., with a box containing as many nodes computing the existential quantifier in the time domain as general assemblies there are in the GABNN. In other words, in each general assembly there should be one node both

computing the existential quantifier in the time domain and corresponding to just one, well identified, different general assembly, for all the general assemblies of the GABNN. Once the present general assembly A input control output node has been activated, see the chapter 5.6.5. and the Figures 5.8. and 5.9., it sends one signal to the all the nodes corresponding to itself and both computing the existential quantifier and belonging to all the remaining general assemblies. As soon as A has terminated its own activity, ie, as soon as A has reset its input control box to the initial conditions, it sends an inhibiting signal to the nodes computing the existential quantifiers both corresponding to A and belong to all the remaining general assemblies. In other words, in the general assembly A the output node of the input control box sends both a predisposing signal to all the A nodes computing the existential quantifiers in the time domain, or A existential nodes, and activates its Hebbian-out first node belonging to the second Hebbian set and obeying the second Hebbian rule. From that moment on, the A existential nodes are first activated and then inhibited by the signal incoming from the other IBNSs. As they are all inhibited, the A output control box both predisposes them to accept again the signals from the other IBNSs and activates the A Hebbian-out second node belonging to the second Hebbian set and obeying the second Hebbian rule. As soon as A existential nodes are all inhibited for the second time, A output control box sends a resetting signal both to the A input control box and to the existential quantifier of all the general assemblies corresponding to itself. Whereas in this way the number of nodes per general assembly depends always on the number of general assemblies present in a GABNN, the number of output connections per general assembly decreases.

5.6.5. *The general assembly input control.*

The input control subnet has four tasks. The first is to make sure the general assembly Hebbian-out connections belonging to the second Hebbian set are frozen before its Hebbian-in connections belonging to the second Hebbian set, so to guarantee the Hebbian connections freezing process obeying the second Hebbian rule does not implement loops in the GANN. The second is to verify the conditions for the computation of the mask algorithm, so to guarantee even for the incremental learning case, see the chapter 5. 2. 1. last paragraph, the freezing of the Hebbian-in connections belonging to the second Hebbian set, ie, terminating onto the general assembly Hebbian-in nodes obeying the second Hebbian rule. The third is to start and then reset the general assembly computation of the said transients and of the steady state algorithm. The fourth task is, first, to store the present general assembly calling sequence, ie, to store the GABNN first incoming input pattern, after the one just stored into the previous general assembly, which satisfy the Hebbian parameter values; second, to enable the next general assembly; third, to prevent a second, ie, a different general assembly to store the very same calling sequence the present general assembly has just stored.

5.6.5.1. *The input control first task..*

To complete the first task, there is the input control node g, see the Figure 5.9.**A)**, inhibiting the input control s, and i-s, Hebbian-in nodes belonging to the second Hebbian set, ie, obeying the second Hebbian rule. As the node g computes the existential quantifier of the GABNN input, ie, as the node g becomes active as soon as there is a one in the GABNN input subvector, the general assembly Hebbian-in nodes belonging to the second Hebbian set and obeying the second Hebbian rule cannot become active, initially. Hence, they cannot be frozen as long as the node g is active. The node g remains active until it is inhibited by the activity of the node v, see the Figure 5.9.**B)**. To understand the argument

of the existential quantifier computed by the node v, we must recall the property characterizing the first and the second sons of a general assembly, see 5.6.3.1. and 5.6.3.2.. As usual, let us call A the latter general assembly, A' its first son and A'' its second son, with A different from both A' and A''. Respectively, A', A'', is the general assembly represented by the first identifier arriving at the A or subnet after the activity of the A output control node two, four, namely of its a, b, output, see the Figure 5.10.**A)**. Consequently, it is sufficient for the node v to be active because of the activity, first, of the A output control a output, second, followed both by another general assembly node s, or i-s, activity and by the inactivity of, namely the inhibition generated by, another general assembly input control node a, a'', or a', fourth, of the A output control b output, fifth, followed both by another general assembly node s, or i-s, activity and by the inactivity of, namely the inhibition generated by, another general assembly input control node a, a'', or a'. As the delay after the activity of a general assembly output control a or b outputs and the activity of its sons input control s, or i-s, nodes is variable, the n and n' nodes computes the existential quantifiers of their inputs. Accordingly, they have to be reset, ie, inhibited every time the general assembly they belong to is reset to its initial conditions, ie, every time it becomes active the node rst belonging to the very same general assembly.

5.6.5.2. *The input control second task.*

To complete the second task, there is the input control node j, see the Figure 5.9.**C)**, with a greater than the threshold non-Hebbian connection to the other IBNSs output control node eight. Once the general assembly A Hebbian-out connections belonging to the second Hebbian set and obeying the second Hebbian rule has been frozen, its Hebbian-in s, and i-s, nodes belonging to the second Hebbian set and again obeying the second Hebbian rule are no longer inhibited. Because of the incremental learning, every time a Hebbian connection belonging to the second Hebbian set is frozen, the time to execute the primitive output sequence increases, as at least one more primitive output must be executed. Accordingly, either we are satisfied with a GABNN trainable to execute output sequences having a maximum length dependent on D, the maximum delay associated to the connections in between the nodes. Or we must find a way to shorten the GABNN responses during the training sessions. One solution is to inhibit A sons Hebbian-out nodes belonging to the second Hebbian set and obeying the second Hebbian rule, during A training sessions, ie, when the GABNN is trained to freeze the A Hebbian-in connections belonging to the second Hebbian set and obeying the second Hebbian rule. In other words, to understand the algorithm computed by the BNN represented in the Figure 5.9.**C)**, we must consider both A and its two sons, ie, the general assemblies A' and A''. Actually, the node j computes the existential quantifier of the inclusive disjunction either, first, of the logical conjunction of the three inputs: i) both the remaining IBNSs output control a and b outputs, ii) the same general assembly input control a node, iii) followed the next tau by the same assembly input control s node; or, second, of the logical conjunction of the three inputs: i) both the remaining IBNSs output control a and b outputs, ii) the same general assembly input control a node, iii) followed the next tau by the same assembly input control s node. In turn, the input control node j is inhibited by the resetting activity of the A output control node six, as the latter is activated to reset the general assembly to its initial conditions. The node j output activates the other general assemblies output control node eight, see the Figure 5.10.**B)**. As the general assembly A', A'', is uniquely determined, actually, its output control a and b outputs are the ones mentioned in the above given inclusive disjunction arguments, namely the i), both of the first and of the second logical conjunction.

5.6.5.3. *The input control third task.*

To complete the third task, there are the input control nodes ocs and rst, see the Figure 5.9.**A)**. The node ocs sends a greater than the threshold non-Hebbian connection to the same general assembly output control node one. The node rst resets the same input control subnet to the initial conditions, once the identifier subnet has outputted the general assembly identifier for the second time. The node rst is activated by the reset subnet if no identifier has been the input of the or subnet in the given interval of time. The node ocs is activated either by the node s or, with a one tau delay, by either the node a, or a'', or by both. The node ocs is inhibited either by the node i-s or, with a one tau delay, by the node a', or by both. The node a, a'', or a', is activated by the GABNN input, when the latter value is the general assembly first, second, calling sequence, or i-calling sequence. The nodes a and a'' inhibit the node a' and the node a' inhibits both the node a and the node a''. The nodes a and a'' activate the node e, computing the existential quantifier. In turn, the node e inhibits the node a, a'' and a' until is itself inhibited by the node rst, at reset time. Both the node a and the node a'', the node a', have a greater than the threshold connection to the node s, to the node i-s. The nodes s and i-s are inhibited by the node g, until the general assembly Hebbian-out connections belonging to the second Hebbian set become frozen.

5.6.5.4. *The input control fourth task.*

To understand a GABNN input control fourth task we must introduce the general assemblies total ordering. Apart from the first and the last, in a GABNN each general assembly is preceded by exactly one previous general assembly and followed by exactly one next general assembly. The ordering is implicitly defined by a facilitatory non-Hebbian connection starting from the input control node next, see the Figure 5.9.**D)** and **D')**, of the present general assembly and terminating onto the input control node go, see the Figure 5.9.**D)**, node pre, see the Figure 5.9.**D')**, of the next general assembly. Whereas in the first IBNS the node go is activated as the GABNN is started, in the last IBNS the node next lacks the output connection.

To discuss the input control fourth task we must distinguish two cases: the first and the second case, respectively, see the chapter 5.6.1.. Let us call M the number of general assemblies present in the GABNN and n the number of the input subvector components, the first case should be chosen for $M < [2^n - n][n - 1]^{-1}$ and the second case should be chosen for $M > [2^n - n][n - 1]^{-1}$, so to optimize the number of the first Hebbian set nodes, ie, the number of Hebbian nodes storing the general assemblies identifiers.

In the first case, to execute the fourth task there are as many first Hebbian sets of nodes, see the Figure 5.9.**D)**, as the number of the GABNN input modalities. The node g, see the Figure 5.9.**A)**, sends a non-Hebbian connection, with an associated coupling coefficient equal to one half of the threshold, onto the node go. The latter receives also a connection, with an associated coupling coefficient equal to one half of the threshold, from the node next belonging to the preceding general assembly. Once the latter becomes active, ie, once the preceding general assembly has stored its own calling sequence, the node go of the present general assembly becomes active. As the node g computes the existential quantifier in the time domain of the GABNN input and the preceding general assembly next node sends out a continuous stream of signals, also the present general assembly node go sends out a continuous stream of signals to the Hebbian-in node out belonging to the first Hebbian set, ie, obeying the first Hebbian rule. As the GABNN receives the input patterns, the general assembly first Hebbian set node out receives the corresponding set of signals on the incoming Hebbian connections belonging to the first Hebbian set, ie, obeying the first Hebbian rule. As the Hebbian-out connections of the present general assem-

bly belonging to the second Hebbian set and obeying the second Hebbian rule are frozen, the node g is inhibited and the node go does not send any more a signal to the node out belonging to the first Hebbian set and obeying the first Hebbian rule. Accordingly, once frozen the connections belonging to the second Hebbian set, first, the Hebbian-in node out belonging to the first Hebbian set and obeying the first Hebbian rule inhibits the node go, second, activate the same general assembly next node. So the carousel starts anew with the next general assembly.

The node stp task is to prevent the training of a second different general assembly to respond to the same calling sequence, once one general assembly node out has been trained to do that. To this end, the node go inhibits the node stp, until the node out training has been completed, ie, until the node g is inhibited, see the chapter 5.6.5.1. and the Figure 5.9.**A)**. Once the training of the node out has been completed, the node stp is activated by the same node out activity. Then, as the present general assembly is reset once its task has been completed, see the chapter 5.6.5.2. and the Figure 5.9.**A)**, the node stp is inhibited by the rst node. That is, the node stp output inhibits the nodes out of all the remaining general assemblies and this happens when the present general assembly node out is activated only by an incoming calling sequence. Accordingly, the node stp prevents the same calling sequence, namely the one previously stored in the present general assembly Hebbian-in connections terminating onto the node out, to freeze a second general assembly Hebbian-in connections belonging to the first Hebbian set and obeying the first Hebbian rule.

In the second case, to execute the fourth task the GABNN input net is the net transforming the input bitmap of the appropriate modality in the general assemblies calling sequences. It contains the 2^n Hebbian-out nodes belonging to the first Hebbian set and obeying the second Hebbian rule. The activity of one of the said Hebbian-out nodes corresponds to one of the GABNN input subvector activity configurations. Each of them sends its Hebbian connections, obeying the second Hebbian rule, to all the general assemblies Hebbian-in node "out" belonging to the first Hebbian set. That is, each node out receives 2^n Hebbian-in connections.

The general assembly input subnet is the subnet trainable to store the very same general assembly calling sequence. The input subnet contains the general assembly Hebbian-in node, the node out, belonging to the first Hebbian set, obeying the second Hebbian rule ad receiving 2^n Hebbian-in connections. The connections belonging to the first Hebbian set, represented in the Figure 5.9.**D')** by the heavy lines, are trainable to store the general assembly calling sequence, ie, to store the GABNN input pattern with a repetition cadence compatible with the Hebbian parameters and arriving once the node pre has been activated. That is, the node pre computes the existential quantifier of the node next of the previous general assembly. Once the node pre has been activated, the node go and the node out, in the order, are activated. In turn, the node go is inhibited by the activity of the box Hbx every H taus, where H depends on the Hebbian parameter values. As the node go is inhibited, thanks to the node tmp the net tests whether at the least one of the Hebbian-in connections terminating onto the node out has been frozen. In the positive case, the node go and the box Hbx output are definitively inhibited, so the node next is activated and the carousel starts anew in the next general assembly. Otherwise, the node go activates the node out and everything goes on as before.

The node stp task is to prevent the training of a second different general assembly to respond to the same calling sequence, once one general assembly node out has been trained to do that. To this end, the node go inhibits the node stp, until the node out training has been completed, ie, until the node g is inhibited, see the chapter 5.6.5.1. and the Figure 5.9.**A)**. Once the training of the node out has been completed, the node stp is activated by the same node out activity. Then, as the present general assembly is reset once its task has been completed, see the chapter 5.6.5.2. and the Figure 5.9.**A)**, the node stp is inhibited by the rst node. That is, the node stp output inhibits the nodes out of all the re-

maining general assemblies and this happens when the present general assembly node out is activated only by an incoming calling sequence. Accordingly, the node stp prevents the same calling sequence, namely the one previously stored in the present general assembly Hebbian-in connections terminating onto the node out, to freeze a second general assembly Hebbian-in connections belonging to the first Hebbian set and obeying the second Hebbian rule.

5.6.6. *The general assembly output control.*

The output control subnet has two tasks. The first is to guarantee a proper activation of the general assembly first and second Hebbian-out connection belonging to the second Hebbian set and obeying the first Hebbian rule, see the Figure 5.10.**A)**, in the order and one independently of the other. The second is to execute the mask algorithm, see the Figure 5.10.**B)**, once in another general assembly the BNN discussed in 5.6.5.2., see also the Figure 5.9.**B)**, has verified the conditions for its execution, ie, once in another general assembly the input control node j is active.

5.6.6.1. *The general assembly output control first task.*

For the second Hebbian rule to work properly, the Hebbian connections belonging to the second Hebbian set, ie, the ones obeying the second Hebbian rule must be active no more than D taus before the activity of the Hebbian-in node belonging to the second Hebbian set on which they terminate. In other words, a general assembly Hebbian-out connections belonging to the second Hebbian set, namely the connections stemming out of its Hebbian-out nodes belonging to the second Hebbian set, must be active at some well defined times, if we want both an efficient self-training process and a proper working GABNN once it has been trained. That is, in a trained general assembly the Hebbian-out nodes belonging to the second Hebbian set must be active one after the other, always in the same order, and the next must be activated only after the present one has terminated its own task. As usual we present general assemblies with two Hebbian-out nodes belonging to the second Hebbian set and obeying the second Hebbian rule. To have general assemblies with a greater number of Hebbian-out nodes belonging to the second Hebbian set and obeying the second Hebbian rule, it is sufficient to increase the number of node pairs like the two-three or four-five, see the Figure 5.10.**A)**. However the upper bound on the number of such Hebbian-out nodes depends on the Hebbian parameter values, see the discussion in the chapter 5.4.1. and 5.5.1..

As the input control node ocs has a greater than the threshold non-Hebbian connection to the output control node one, see the Figure 5.10.**A)**, the activity of the former is followed by the activity of the latter, after one tau. That is, two taus after the activity of the input control node ocs, the first Hebbian-out node belonging to the second Hebbian set and obeying the second Hebbian rule becomes active and, simultaneously, are inhibited the output control node three and five. As the node two has a greater than the threshold non-Hebbian connection terminating on the node three, the next tau the latter becomes active. The node three computes the existential quantifier of the node two activity, so it remains active until the node one, restarted by the compare subnet, inhibits it. Because the node three output inhibits the node two, when the node one is restarted, the node two remains inactive. By contrast, the node three both inhibits the node four, with a one tau delay, and has a greater than the threshold non-Hebbian connection terminating on the node four, with a two taus delay. Accordingly, when the node one is restarted and the next tau the node three is inhibited, the node four becomes active after two taus. That is, three taus after the restarting of the output control node one, the general assembly sec-

ond Hebbian-out node belonging to the second Hebbian set and obeying the second Hebbian rule becomes active and, simultaneously, are inhibited the output control node three and five. As the node four has a greater than the threshold non-Hebbian connection terminating on the node five, the next tau the latter becomes active. The node five computes the existential quantifier of the node four activity, so it remains active until the node one, restarted by the compare subnet, inhibits it. Because the node five output inhibits the node two, when the node one is restarted, the node two remains inactive. Moreover, the node five both inhibits the node six, with a one tau delay, and has a greater than the threshold non-Hebbian connection terminating on the node six, with a two taus delay. Accordingly, when the node one is restarted and the next tau both the node three and the node five are inhibited, the node six becomes active after two taus. Eventually, the node six activity is the signal for both the identifier subnet to broadcast the general assembly identifier and the input control subnet to reset the general assembly to the initial conditions.

5.6.6.2. *The general assembly output control second task.*

If both in the general assembly A either the node a or the node a'' or the node a' is active and in the general assembly A', with A different from A', the input control node j is active, then, in the general assembly A the output control subnet computes the mask algorithm, see the Figure 5.10.**B)**. Because the input control node j computes the existential quantifier of its facilitatory inputs, as either the node a or the node a'' or the node a' of A is active, the output control node nine of A is active, the next tau. That is, the next tau in the general assembly A both the output control nodes one and two are inhibited and the identifier subnet is activated. And after two taus, the output control node six is activated, so resetting A to the initial conditions without ever activating A sons.

5.6.7. *The general assembly store subnet.*

As the store subnet receives a signal from either the a or b output of the same general assembly output control subnet, see the chapter 5.6.6. and the Figure 5.10.**A)**, it is enabled to accept another IBNS identifier from the or subnet. Here we assume an identifier to be represented by an n digit binary numeral. However, as a GABNN can be composed by many IBNSs, the number of identifiers depends on the number of IBNSs. That is, the number of nodes needed to store the identifiers and, a fortiori, the number of nodes for a general assembly is not a constant. It depends on the number of IBNSs present in the GABNN and on the identifier representation in the store BNN.

Once it has stored an identifier the store subnet enabling node, the one activated by the same general assembly output control either a or b outputs, is inhibited, so later coming identifiers cannot interfere with the already stored one. Both the store subnet outputs, ie, an enabling signal and the very identifier it is storing are sent to the compare subnet, see the chapter 5.6.9. and the Figure 5.13.. If the store subnet is empty, ie, if it does not store an identifier, both its output connections are inactive, so in effect representing the null identifier. The store subnet is reset to the initial conditions either by the compare or by the store subnet.

5.6.8. *The general assembly "or" subnet.*

The or subnet computes the inclusive disjunction of the m-th digit of the identifiers incoming from the other IBNSs, m being a positive integer no greater than n. An identifier is assumed to be an n digit binary numeral, see the Figure 5.12.. The or subnet out-

puts go to the store subnet, see the chapter 5.6.7. and the Figure 5.11., to the compare subnet, see the chapter 5.6.9. and the Figure 5.13., and to the reset subnet, see the chapter 5.6.10. and the Figure 5.15..

5.6.9. *The general assembly compare subnet.*

The compare subnet task is to restart the output control, see the chapter 5.6.6. and the Figure 5.10. **A)**, whenever the store subnet content is the same as the identifier incoming from the other IBNSs, through the or subnet. However, as an empty store content is the same as no incoming identifier, both are represented by n inactive connections. Accordingly, the compare subnet output connection becomes active only if there is the enabling signal sent by the store subnet. In turn, the latter sends the enabling signal only if it is storing a non null identifier, ie, if it is not empty.

The n-xor subnet has an output signal every time the store and the or subnet have the same pattern of activity, both if the pattern of activity represents the same general assembly identifier or if they are both inactive. For simplicity sake, we have given the n-xor subnet, see the Figure 5.14., with zero threshold nodes, ie, with always active nodes. They could have been substituted with nodes computing the logical negation of a node, eg of the output control node six, surely inactive when the n-xor subnet is active.

5.6.10. *The general assembly reset subnet.*

The reset subnet task is to reset the general assembly to the initial conditions if an identifier does not arrive within a time interval equal to the maximum delay D of the general assembly Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule. Actually it is just a delay line composed by D nodes. It is inhibited by the or subnet, see the chapter 5.6.8. and the Figure 5.12.. Its output is a facilitatory connection to the input control node rst, see the chapter 5.6.5.3. and the Figure 5.9.**A)**, and an inhibitory connection to the store subnet.

5.6.11. *The GABNN primitive subnet.*

As any other general assembly, the primitives, see the Figure 5. 16., have Hebbian-in nodes belonging to the second Hebbian set and obeying the second Hebbian rule. By contrast, they lack the Hebbian-out connections belonging to the second Hebbian set, namely the ones obeying the second Hebbian rule, and the store, the compare, the reset and the "or" subnets, see the Figures 5.11., 5.12., 5.13. and 5.15., respectively. Actually, the primitive p output control, see 5.6.12. and the Figure 5.18., has just two non-Hebbian output connections. They correspond to the connections labeled i and iii in the Figures 5.10. and 5.16.. A signal on the p-th primitive output control connection i activates, first, just one, always the same, effector: the GABNN p-th effector. Second, it activates the identifier subnet and, consequently, the primitive broadcasts its own identifier to all the general assemblies. Once the effector p has terminated its own activity, it sends a restart signal to the primitive p output control subnet. As the primitive p receives the restart signal from the p-th effector, its output control iii connection becomes active and, consequently, the primitive broadcasts its own identifier to all the general assemblies. That is, a primitive signals to all the general assemblies both the beginning and the end of its activity by broadcasting its own identifier, exactly as any general assembly does.

As we have depicted in the Figure 5.16., a primitive, as any general assembly, input control subnet, see the Figure 5.9., accepts its own identifier, only. As the primitive p has

received its identifier, it sends a start signal to the same primitive output control subnet. As the latter receives the start signal from the input control, it sends a signal both to the same primitive identifier subnet and, on a non-Hebbian output connection, to the p-th net effector. Whenever the primitive identifier subnet receives a signal from the output control, it broadcasts the very same primitive identifier to all the GABNN general assemblies "or" subnets. As the p-th primitive output control subnet is restarted by the signal incoming from the p-th net effector, both it sends a signal to the input control and to the identifier subnet. The first resets the input control subnet and, because of the second, the primitive broadcasts its own identifier to all the general assemblies. Accordingly, the primitive has been reset to the initial conditions.

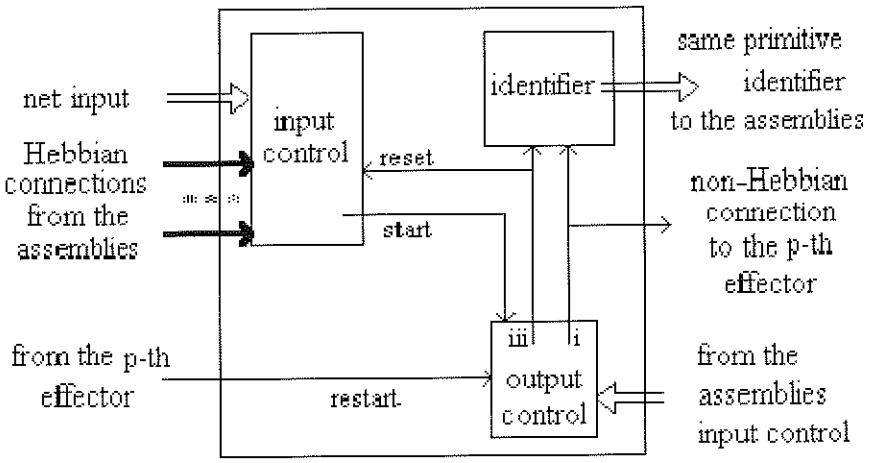


Fig. 5.16.. *The p-th primitive subnet.* The light arrows represent the single non-Hebbian connections, the heavy arrows the Hebbian connections both belonging to the second Hebbian set and obeying the second Hebbian rule. Eventually, the hollow arrows represent a number, dependent on the identi fier magnitude, of non-Hebbian connections.

5.6.12. *The primitive input control subnet.*

The input control task is to start and then reset the primitive to the initial conditions. To complete its task, there are the input control nodes ocs and rst, see the Figure 5. 17.. The node ocs sends a greater than the threshold non-Hebbian connection to the same primitive output control node one. The node ocs is activated by the node s and is inhibited by the node i-s. The node a, a", or a', is activated by the GABNN subvector input, when its value is the first, second, primitive identifier, or i-identifier, see the chapter 5. 3.. The node a, a" and a' inhibit each other, mutually. The nodes a and a" activate the existential quantifier node e, which inhibits a and a" and a' until is itself inhibited by the node rst, at reset time. The nodes a, a" and a', have a greater than the threshold connection to the node s, i-s. The node rst resets the same input control subnet to the initial conditions, once the identifier subnet has broadcast the primitive identifier for the second time. As a primitive has only non-Hebbian output connections, it does not need to inhibit its Hebbian-in nodes belonging to the second Hebbian set and obeying the second Hebbian rule. As the son of a primitive is an effector, not an IBNS, the primitives do not have to verify the conditions for the mask algorithm, see the chapter 5.6.5.2..

As we have depicted in the Figure 5.17., whereas the node s activates the node ocs, the node i-s inhibits it. To help to freeze some of the primitive Hebbian-in connections belonging to the second Hebbian set and obeying the second Hebbian rule, the nodes a and a" activate the s node, namely one of the general assembly Hebbian-in node belonging to the second Hebbian set and obeying the second Hebbian rule. To help to freeze some of the primitive Hebbian-in connections belonging to the second Hebbian set and

obeying the second Hebbian rule, the node a' activates the i-s node, namely the primitive remaining Hebbian-in node belonging to the second Hebbian set and obeying the second Hebbian rule. To guarantee a proper working of the primitive, once activated the nodes a, a'' and a' inhibit each other. The primitive is reset to the initial conditions, by the activity of the node rst, in turn activated by the output control terminating signal on the iii output.

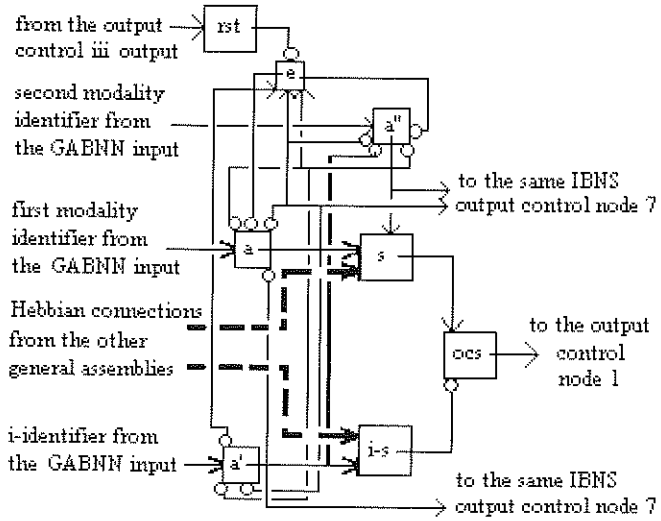


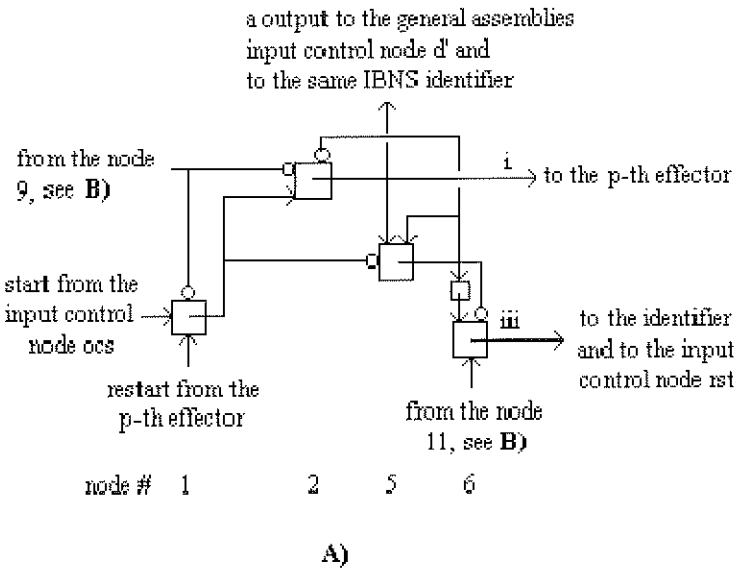
Fig. 5.17.. *The primitive input control subnet.* The symbols within the nodes are the node identifiers. The empty circles represent the absolute inhibitory connections; the single barbed light arrows represent the facilitatory connections: unless otherwise specified their associated coupling coefficients are greater than the node threshold; the heavy broken lines represent the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule coming in from the general assemblies output control Hebbian-out nodes belonging to the second Hebbian set and obeying the second Hebbian rule, see the Figure 5.10..

5.6.13. *The primitive output control subnet.*

The output control subnet of the primitive p has two tasks. The first is to guarantee a proper activation of the primitive non-Hebbian output connection to the effector p, see the Figure 5. 18. **A)**. The second is to execute the mask algorithm, see the Figure 5.18.**B)**, once in a general assembly the BNN discussed in 5.6.5.2., see also the Figure 5.9.**B)**, has verified the conditions for its execution, ie, the said BNN output node j is active. To simplify the confront with the BNN discussed in 5.6.6., the nodes have the same numeration given in the Figure 5.10..

As we have represented in the Figure 5.18.A), the primitive output control subnet node one input is the output of the input control subnet node ocs, see the Figure 5.17.; the primitive p restarting input is the output of the effector p. The a output goes to the input control subnet of all the general assemblies, see the Figure 5.9.**B)** and **C)**. Once it becomes active, the node number one both activates the node number two, ie, the output control output i, and inhibits the node number five. The next tau, the node two: i) activates the node five, ii) sends a signal on a non-Hebbian connection to the input control subnet of the other general assemblies, iii) sends a signal on the non-Hebbian connection to the effector p. The next tau, ie, three taus after the start signal arrives, the node five becomes active and remains active until a new signal from the node one, ie, from the effector p arrives. As the node number five becomes active, i) it inhibits the nodes two and six, ii) it sends, with a two taus delay, a facilitatory signal to the node six. As the node five is inhibited by the node one new activity, ie, three taus after a new signal arrives, for the second time, upon the node one, the node six becomes active. As the node six becomes active,

it sends both a facilitatory signal to both the identifier and the input control subnets, so resetting them and resetting to the initial conditions the primitive. The BNN given in the Figure 5.18.**B)** computes the mask algorithm, see the chapter 5.6.5.2.. That is, if both the same primitive node a, or a'' or a', and at least another general assembly node j are simultaneously active, then, after one tau, the same primitive output control nodes one and two are inhibited and the same primitive identifier subnet is activated. And after two taus, the output control node six is activated, so resetting the same primitive to the initial conditions without ever activating the effector p.



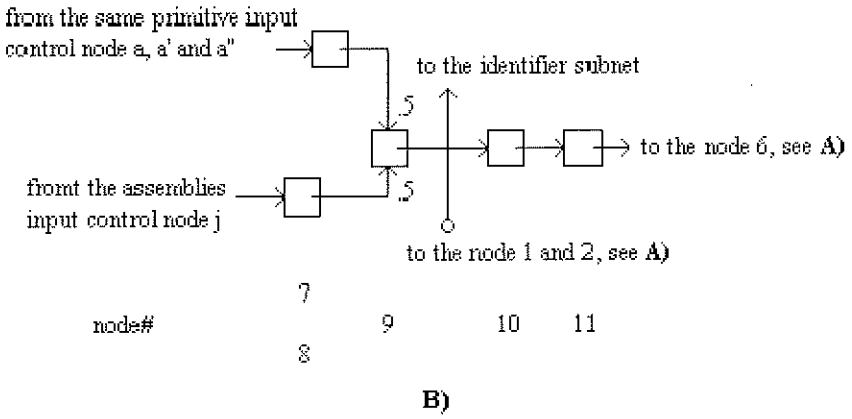


Fig. 518.. *The primitive output control subnet.* The empty circles represent the absolute inhibitory connections; the single barbed arrows represent the facilitatory connections. Unless otherwise stated, the latter have associated coupling coefficients no smaller than the threshold. To simplify the confront, the nodes have the same numeration given in the Figure 5.10..

5.6.13.1. *The primitive output control first task.*

As the input control node ocs has a greater than the threshold non-Hebbian connection to the output control node one, see the Figure 5.18.**A)**, the activity of the former is followed by the activity of the latter, after one tau. That is, two taus after the input control node ocs becomes active also the node one becomes active and, simultaneously, is in-

hibited the output control node five. As the node two has a greater than the threshold non-Hebbian connection terminating on the node five, the next tau the latter becomes active. The node five computes the existential quantifier of the node two activity. In other words, it remains active until it is inhibited by the node one, ie, as the latter is restarted by the effector p as it has completed its task. Because the node five output inhibits the node two, when the node one is restarted, the node two remains inactive. Moreover, the node five both inhibits the node six, with a one tau delay, and has a greater than the threshold non-Hebbian connection terminating on the node six, with a two taus delay. Accordingly, when the node one is restarted and the next tau the node five is inhibited, the node six becomes active, actually with a two tau delay in respect to the node one activity. Eventually, the node six activity is the signal both for the identifier subnet to broadcast the primitive identifier and for the input control subnet to reset the primitive to the initial conditions.

5.6.13.2. *The primitive output control second task.*

If in the general assembly A the input control node j is active, see the chapter 5.6.5.2. and the Figure 5.9.**C)**, and in the primitive p the input control node a, or a'' or a', is active, the primitive p output control subnet computes the mask algorithm, see the Figure 5.18.**B)**. Because both in the general assemblies the input control node j computes the existential quantifier of its inputs and as either the node a or the node a'' or the node a' of p is active, the output control node nine of p is active, the next tau. That is, the next tau in the primitive p both the output control nodes one and two are inhibited and the identifier subnet is activated. And after two taus, the output control node six is activated, so resetting p to the initial conditions without ever activating the effector p.

5.6.14. *The general assembly input limitations.*

To discuss the limitations of a general assembly, let us examine the constraints on the input sequences accepted by the GABNN. As usual, let A, A' and A'' be three IBNS identifiers, with A a general assembly identifier and with A', the first son, and A'', the second son, different from A. Because to freeze the relevant Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule the three IBNSs must compute the mask algorithm, the GABNN input subvector content sequence must reproduce a schema of the type:

, ..., A, D(D', d', A'), D(D'', d'', A''), ...,

where d' < D', d'' < D'', are constants and D(D', d', A'), D(D'', d'', A''), represents a sequence of D', D'', consecutive input subvector contents containing A', A'', at the place d', d'', as the first occurrence of an IBNS identifier, if we read the sequence D(D', d', A'), D(D'', d'', A''), going from the past toward the present. Actually, to freeze the relevant Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule the very same sequence A, D(D', d', A'), D(D'', d'', A'') does not have to be repeated again and again, as is the case for the six and sixteen node assembly ABNN. Instead, in the GABNN case, it is sufficient to repeat the triple [A, A', A''], with the same delays as in the sequence ..., A, D(D', d', A'), D(D'', d'', A''), ...

In other words, in contrast with the six and the sixteen node assembly ABNN, the GABNN input subvector contents leading to the Hebbian connection belonging to the second Hebbian set freezing in accordance with the second Hebbian rule have not to assume just one particular sequence. In the GABNN case, the sequence of the said input subvec-

tor contents belong to a set containing finitely many sequences. Actually, as in the six and sixteen node ABNNs, in a GABNN to freeze a given Hebbian connection belonging to the second Hebbian set and obeying the second Hebbian rule, the very same sequence have to be the GABNN input subvector content finitely many times with a suitable repetion cadence. However, whereas in the six and sixteen node ABNNs the said input subvector sequence is just of the type given in the chapter 5.4.6. or, respectively, 5.4.4., in the GABNN it is one out of many different sequences. That is, as only one identifier at the time can be GABNN input, see the chapter 5.3., the triple [A, A', A"] shall be the one stored in the frozen Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule, if and only if the triple [A, A', A"] is the first one repeated, with the same delays as in the first occurrence of the sequence ..., A, D(D', d', A'), D(D", d", A"), ..., a suitable number of times, the number depending on the given Hebbian parameter values, see the chapter 4.1.1. .

Because in a general assembly the Hebbian-in connections belonging to the second Hebbian set and obeying the second Hebbian rule cannot be frozen before its Hebbian-out connections belonging to the second Hebbian set and obeying the second Hebbian rule, see the chapter 5.6.5.1. and the Figure 5.9.**B)**, a GABNN cannot be trained to freeze its general assemblies in loops. However, we are wrong if we conclude that a GABNN can compute only the primitive recursive functions. As we have seen in the chapter 2.2.2., the Adder is an example of a BNN repeating again and again some very elementary tasks chosen from a finite set, until a condition in its environment is verified. Specifically, as we shall discuss in the chapter six, by interacting with its environment a GABNN can be trained, either assisted or unassisted, to compute the recursive functions.

Least but not the last, a general assembly, thanks to the nodes belonging to the first Hebbian set, can be trained to store its own calling sequence. So we do not have to wire its identifier onto its input nodes, as is the case for the other assembly previously discussed,

5.7. A formal characterization.

To characterize formally a GABNN, let us consider one composed by N general assemblies and P primitives. To fix the ideas, let the general assembly identifiers be the integers 1, 2, ..., N; the primitive identifiers be the integers N + 1, N + 2, ..., N + P < K; the GABNN input subvector content i be a positive integer no greater than K, K being the integer represented by the input subvector content when all its components have the value one; the GABNN output subvector content o_0, o_1, ..., o_p, ..., o_P be the integers 0, N + 1, ..., N + p, ..., N + P, respectively. Initially, for $N < i \leq N + P$ let the GABNN input subvector content correspond to the GABNN output subvector content o_{i-N}; to o_0, otherwise.

5.7.1. *The finite state machine immanent in a GABNN.*

Given M, N, P, N + P < K, T, $m_{q-1} < m_q$, $T_{q-1} < T_q$, $0 < m_q < T_q < T$ and $0 < q \leq M$, let us consider the sets $\{T_q\}$; $\{m_q\}$; $\{i_t\}$, $0 < t \leq T$, $0 < i_t < K$; the input set $I \equiv \{i_T \ldots i_t \ldots i_2 i_1\}$ where, by convention, the left to right direction represents the past to present input sequence; $\{o_p\}$, $0 < p \leq P$; the output set O having as elements both o_0 and the sequences of $\{o_p\}$ elements of length one, two, ..., 2^N; the state set $S \equiv \{[s_1, s_2, \ldots, s_{q''}, \ldots, s_N; s'_1, s'_2, \ldots, s'_{q'}, \ldots, s'_N]\}$, where both $s_{q''}$ and $s'_{q'}$ are variables on the non negative integers. Whereas initially $s_{q''}$ and $s'_{q'}$ are set to zero, ie, initially the state is a couple of N-tuples of zeros [0, 0, ..., 0; 0, 0, ..., 0], they assume a value no greater than N + P, see below the next state function definition. We are going to represent a GABNN as the formal system character-

ized by the quintuples {I, O, S, Σ, Ω}, where Σ is the next state function and Ω is the next output function, see below.

5.7.1.1. *The next state function.*

To simplify the presentation, we introduce a suitable notation for the input sequences to be read, as usual, in the left to right direction. Given the integers $0 < d' \leq D'$, $0 < d'' \leq D''$, with the notation $D(D', d', i_2)$, $D(D'', d'', i_1)$, we represent a sequence containing D', D'', symbols belonging to $\{i_t\}$ in which i_2, i_1, both is the first occurrence of an i_t no greater than N + P and occurs at the place d', d''.

The next state function $\Sigma[[i_T \ldots i_3 i_2 i_1], [s_1, s_2, \ldots, s_N; s'_1, s'_2, \ldots, s'_N]]$ is defined by the following two step rule, to apply in the order:

first step) if the sequence $i_3 D(D', d', i_2)$ is present at least m_q times in the sequence $i_T \ldots i_3 D(D', d', i_2) D(D'', d'', i_1)$, for at least one q, and $0 < i_3 \leq N$ and $s_{i3} = 0$ and either $0 < i_2 \leq N$ and $s_{i2} > 0$ and $s'_{i2} > 0$ or $N < i_2 \leq N + P$, ie, if we have:

$$\Sigma[[i_T \ldots i_3 D(D', d', i_2)\, D(D'', d'', i_1)], [s_1, s_2, \ldots, s_{i3} = 0, \ldots, s_N; s'_1, s'_2, \ldots, s'_N]]$$

then, the next state becomes:

$$[s_1, s_2, \ldots, s_{i3} = i_2, \ldots, s_N; s'_1, s'_2, \ldots, s'_N]$$

otherwise it is left unchanged, ie:

$$[s_1, s_2, \ldots, s_{i3} = 0, \ldots, s_N; s'_1, s'_2, \ldots, s'_N]$$

second step) if the sequence $i_3 D(D', d', i_2) D(D'', d'', i_1)$ is present at least $m_{q'}$ times in the sequence $i_T \ldots i_3 D(D', d', i_2) D(D'', d'', i_1)$, for at least one q', and $0 < i_3 \leq N$ and $s_{i3} = i_2$ and $s'_{i3} = 0$ and either $0 < i_1 \leq N$ and $s_{i1} > 0$ and $s'_{i1} > 0$ or $N < i_1 \leq N + P$, ie, if we have:

$$\Sigma[[i_T \ldots i_3 D(D', d', i_2) D(D'', d'', i_1)], [s_1, s_2, \ldots, s_{i3} = i_2, \ldots, s_N; s'_1, s'_2, \ldots, s'_{i3} = 0, \ldots, s'_N]]$$

then, the next state becomes:

$$[s_1, s_2, \ldots, s_{i3} = i_2, \ldots, s_N; s'_1, s'_2, \ldots, s'_{i3} = i_1, \ldots, s'_N]$$

otherwise it is left unchanged, ie:

$$[s_1, s_2, \ldots, s_{i3} = i_2, \ldots, s_N; s'_1, s'_2, \ldots, s'_{i3} = 0, \ldots, s'_N].$$

5.7.1.2. *The next output function.*

To define recursively the next output function we need the concatenation operator ⊕: it is defined and assumes values on the set O. Given the two sequences $Q = o_a o_b$ and $Q' = o'_a o'_b$, if we apply to them the concatenation operator, we obtain the new sequence $Q'' = Q \oplus Q' = o_a o_b o'_a o'_b$, i.e., the sequence Q followed by the sequence Q'. Because ⊕ has the associative property, the definition holds for any pair of sequences.

As the next output function depends on i_1, only, for simplicity sake we write its input variable as a i without a subscript. Then, the next output function $\Omega[[i], [s_1, s_2, ..., s_N; s'_1, s'_2, ..., s'_N]]$ is defined as:

a) if $N < i \leq N + P$ and we set $j = i - N$, then:

$\Omega[[i], [s_1, s_2, ..., s_N; s'_1, s'_2, ..., s'_N]] = o_j$

b) if either $0 < i \leq N$ and $s_{i1} = s'_{i1} = 0$ or either $i = 0$ or $i > N + P$, then:

$\Omega[[i], [s_1, s_2, ..., s_N; s'_1, s'_2, ..., s'_N]] = o_0$

c) if $0 < i \leq N$ and $s_{i1} > 0$ and $s'_{i1} = 0$, then:

$\Omega[[i], [s_1, s_2, ..., s_N; s'_1, s'_2, ..., s'_N]] = \Omega[[s_{i1}], [s_1, s_2, ..., s_N; s'_1, s'_2, ..., s'_N]]$

d) otherwise:

$\Omega[[i], [s_1, s_2, ..., s_N; s'_1, s'_2, ..., s'_N]] =$
$\Omega[[s_{i1}], [s_1, s_2, ..., s_N; s'_1, s'_2, ..., s'_N]] \oplus \Omega[[s'_{i1}], [s_1, s_2, ..., s_N; s'_1, s'_2, ..., s'_N]].$

5.7.1.3. *The GABNN formal characterization at work: an illustrative example.*

To see the formal system at work, let one be initially defined by: $i = 2, 3, 4, 5, 6, 7$; $O \equiv \{o_0, o\}$; $S \equiv \{[0, 0, 0; 0, 0, 0]\}$, $N = 3$ and $P = 1$; m_q is chosen so to be the L% of T_q, with L a positive integer smaller than one hundred. In other words, initially the system responds with the sequence composed by just one symbol, namely o, to the input 5 and with o_0 to any other input, eg to the input 2. To train it to respond oo to the input 2, ooo to the input 3 and oooo to the input 4, its input sequences have to contain the proper sequences of primitive identifiers at least L% of the times among all the inputs actually entering in the formal system. To simplify the presentation, in the input sequences we shall use the variables: $x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16} \in \{6, 7\}$ and $y_1, y_2, y_3, y_4, y_5, y_6 \in \{2, 3, 4, 5, 6, 7\}$.

Accordingly, if $d' = 3$, $D' = 5$, $d'' = 4$, $D'' = 4$, and if an input sequence like $2x_1x_2 5y_1y_2x_3x_4x_5 5$ enters in the system and at least L% of the times among the least T input symbols there has been an occurrence of the input 2 followed by two input symbols, belonging to the set {6, 7}, followed by the input 5, in turn, followed by two input symbols, belonging to the set {2, 3, 4, 5, 6, 7}, followed by three input symbols, belonging to the set {6, 7}, followed by the input 5, then, the system state becomes [5, 0, 0; 5, 0, 0], ie, the input 2 has been associated to the output sequence oo. Second, if $d' = 4$, $D' = 6$, $d'' = 3$, $D'' = 4$ and if another of the possible input sequences, eg $3x_6x_7x_8 2y_3y_4x_9x_{10} 5y_5$, enters in the system and at least L% of the times among the least T input symbols there has been an occurrence of the input 3 followed by three input symbols, belonging to the set {6, 7}, followed by the input 2, in turn, followed by two input symbols, belonging to the set {2, 3, 4, 5, 6, 7}, followed by two input symbols, belonging to the set {6, 7}, followed by the input 5, in turn, followed by one input symbol, belonging to the set {2, 3, 4, 5, 6, 7}, then, after a while the state becomes [5, 5, 0; 5, 2, 0]. That is, the input 3 has been associated to the output sequence ooo. Third, if $d' = 5$, $D' = 5$, $d'' = 3$, $D'' = 3$, and if the sequence like $4x_{11}x_{12}x_{13}x_{14} 3x_{15}x_{16} 5$ enters the system and as before at least L% of the times among the least T input symbols there has been an occurrence of the input 4 followed by four input symbols, belonging to the set {6, 7}, followed by the input 3, in turn, followed by two input

symbols, belonging to the set {6, 7}, followed by the input 5, then, after a while the state becomes [5, 5, 3; 5, 2, 5]. That is, the input 4 has been associated to the output sequence oooo. Depending on the sequences actually presented, a different possible outcome for the final state could have been [5, 5, 2; 5, 2, 2] or, equivalently, [5, 2, 5; 5, 5, 3].

Chapter 6

A GABNN AS AN AUTONOMOUSLY SELF-TRAINABLE CONTROL SYSTEM.

6.1. The neural network applications.

In general both the ANN and the BNN, being functional models of the mammalian nervous systems, can be applied to any situation in which there is something to control. In their own specific approaches, both are programmable systems in which we can implement particular sequences of commands. However, we are primarily interested in the neural networks adaptability, ie, in the possibility to train them, namely to implement in them the possibility, to output command sequences, e. g. by chaining them. The ANNs adaptability can be obtained by means of a number of different learning algorithms, e. g. the error backpropagation, in any case leaving them as special purpose adaptable systems. By contrast, our implementation of the Hebbian rule in a BNN by means of the, general, assemblies preserves the adaptable system, ie, the GABNN universality. In other words, our deterministic implementation of the Hebbian rule preserves the possibility to concatenate any number of the command sequences the GABNN can be trained to control. A feature the traditional ANNs do not share with the ABNNs. Accordingly, now we are going to discuss some of the possible GABNN applications, in order to find one to prove the system training universality.

6.1.1. *The GABNN applications.*

In [6, chapter 2] we have presented, as a BNN application, a multitasking architecture having a BNN as its diffuse control. In [6, chapter 6] we have presented, as an Adaptable BNN application, the said multitasking architecture having the ABNN as its diffuse control. Like a concurrent general purpose von Neumann architecture, the former system executes any program written in its associated language. By contrast, the latter system is automatically trained to increase its associated language with the macros present at its input channel with a repetion cadence sufficient to freeze the subset of its Hebbian connections needed to store the said macros. Obviously, we could have presented here the GABNN controlling an equivalent multitasking architecture so proving its training universality.

An alternative application has been discussed in [27, 28, 29], ie, the Caianiello-Hebb, or CH, algebraic transform. To have a CH algebraic transform we have to implement in the GABNN the algorithms needed to simulate a Turing Machine, together with

its unbounded tape, and the algorithms needed to copy the computation results, from the tape, into the very same GABNN output subvector. In other words, given the function f(x) defined in x*, a CH algebraic transform can be trained to take as the independent variable value, namely as x*, and as the name of the function to be computed, namely as f(), its input subvector values and to assume as its output subvector values the f(x*) value, in a finite succession of the GABNN internal states. In so doing, we can define the CH algebraic transform learning as a property definitively acquired by its internal state succession.

Because the said applications have been discussed elsewhere, in particular as ABNN applications, we are interested here to present a new application. To this end, in both applications, the Boolean neural net has both a finite input subvector and a finite output subvector together with the codes projecting both the input and the output alphabets into two distinct sets of words. Moreover, in both the applications the input alphabets have a subset, let us call it P, containing the input values corresponding to the output "primitives", ie, to each of the net initial alphabet symbols. In both applications the universal training requires that any input value not belonging to P have to be made to correspond to just one finite sequences of output primitives, characteristically depending on the said input value. To avoid a too cumbersome discussion of the GABNN environment, we have opted for an application with as few as possible constraints in an easy to present framework. In other words as the elementary arithmetic operations are common elementary knowledge, we shall discuss a GABNN as the adaptable control system for a writing robot, as it is quite easy to understand its I/O without the need to spend too much time on the presentation of the net-environment interactions and yet satisfying the said conditions for an universal training system.

6.1.2. *The GABNN and the writing robot.*

Let us consider a GABNN composed by a sufficient number of general assemblies to be trained in the tasks we are going to discuss. Our GABNN has three input subvectors: two are the first and the second modality inputs. The third is the inhibitory modality input, ie, the input subvector components assuming the i-calling sequence values. With respect to the IBNSs discussed in the chapter 5.6., whereas the signals belonging to the two facilitatory input modalities activate the input control nodes a and a'', see the Figure 5.9., those belonging to the inhibitory modality input activate the nodes a'. The first and the second IBNS calling sequences are respectively associated to the two facilitatory input modalities. In the GABNN, they are respectively represented by the values of the first and of the second input subsubvector components. We call inhibitory the third modality input.

By contrast, the GABNN has one output subvector to control the corresponding effector responses. Actually, the latter are executed as soon as the subvector components assume one of the possible combinations of ones and zeros, for short as soon as the output nodes are activated.

Because to each calling sequence corresponds just one IBN, a particular combination of activity in the components belonging either to the first or to the second input subsubvector is, in computer parlance, a pointer to a particular IBNS. To avoid keeping a table of the correspondences between the actual calling sequences of an IBNS and the role it has been trained on, we call *name* of a trained general assembly a mnemonic as the sequence of its two son names, in the order they are activated. Eventually, the name of a trained general assembly is the sequence of the names of the two primitives to be activated in the order, whenever the said general assembly calling sequence is the GABNN input subvector component value.

To fix the ideas, we shall present the training procedures for the general assemblies. However, mutatis mutandis, they could have been given for any kind of assembly. Actu-

ally, the main difference is relative to the training example sequences. They are fixed for the six and sixteen node assembly, variables for the general assemblies, see the chapter 5. 6. 14.. Accordingly, *with the notation [A, A', A"], with A different from both A' and A", we represent a training triple of IBNS names, or calling sequences, present as the GABN input subvector component values with delays satisfying the constraints discussed in the chapter 5. 6. 14.*.

We consider the GABNN as the control for a writing robot with six primitive unitary motor responses: left, right, up, down, write, erase, or, in short, `l`, `r`, `u`, `d`, `w`, `e`, respectively, similar to the one discussed in the chapter 2. 2. 1. and in [21, 22]. Without danger of confusion, we call both the GABNN output nodes and the primitive movements they control left, right, up, down, write, erase or `l`, `r`, `u`, `d`, `w`, `e`, for short. The `l`, `r`, `u`, `d`, `w` and `e` output nodes activity are the primitive commands issued by the primitive GABNN to control the robot primitive actions. The first four control the robot primitive movements parallel to the writing plane, the last two control the robot in its writing and erasing capabilities. Initially these are the GABNN entire repertoire of input-output commands, ie, in the GABNN no other input subvector value has a corresponding output subvector value, see also the finite machine introduced in the chapter 5.7.1..

We shall discuss the procedures needed to train the GABNN to control the given robot to compute the sum, the product, the delta function together with the substitution and the minimalization schemata.

6.2. Some auxiliary training procedures.

As discussed in [21], we start by training the GABNN to control the robot to write, or to erase, the digits one and zero, the arithmetic operators like the plus or the times signs, to move around so to reach an assigned position, to copy an intermediate result in the equivalent of a temporary memory location of a von Neumann architecture. In other words, first, we define the robot environment and the effects, as seen by the robot sensors, ie, by the GABNN input subvector component values, of the robot actions, second, we give the *training procedures* for some clerical tasks, to be recalled later. Together with each *training procedure* we give also the number of steps it is composed of, ie, the number of general assemblies needed to store it.

6.2.1. *The GABNN and the controlled robot environment.*

As discussed in the chapter 2.2.2., to control a writing robot the GABNN first modality input subvector component content can be represented as a binary numeral containing one hundred and twenty-five digits, at least, ie, can be represented as the content of a five component environment vector, see the Figure 6.1.. By contrast, the second modality input represents a broader as well as an unbounded environment portion. In any case, also the latter is representable as a one hundred and twenty five digit binary numeral.

In other words, both the GABNN input subvector representing the first modality input and the GABNN input subvector representing the second modality input hold the binary numerals composed by one hundred and twenty five ones and zeros, at least. In the first modality case they represent the state of the five components environment vector, see the chapter 2.2.2. and the Figure 2.1.. Each component is composed by twenty five unitary squares, ie, by twenty five squares having the side length equal to one of the robot primitive movements. To sum it up, whereas to talk about the environment vector component contents means to talk about the environment state, the input subvector component values are the GABNN input values.

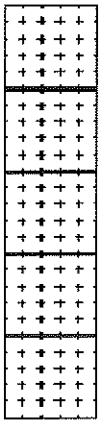


Fig. 6.1.. *The environment vector*: The GABNN first modality input is an input subvector with one hundred and twenty five components corresponding to a five component environment vector. As each input subvector component can assume one of two values, the GABNN input space has 2^{125} distinct points. That is, with such a choice for the input subvector dimensions, in the given GABNN there are less than 2^{125} IBNSs. To each environment vector component corresponds twenty five input subvector components, ie, 2^{25} distinct points in the GABNN input space.

To normalize the control of the robot movements, so to facilitate their concatenation, all the robot journeys start and terminate on the central unitary square of the middle component of the environment vector. Accordingly, all the robot journeys relative to just one environment vector must be composed by as many up, or left, as down, or right, primitive movements. That is, if we associate a plus one to both the `u` and `l` commands and a minus one to both the `d` and `r` commands, it must be zero the sum of all the values associated to the sequence of moving commands controlling any of the robot possible journeys on the same environment vector.

An environment vector unitary square state can be either blank or written on. If the robot writes on a previously written unitary square, it remains written. If the robot erases a blank unitary square, it remains blank. If the robot moves on the unitary square S without executing neither the `w` nor the `e` command, then, the S state remains unchanged. That is, if S was a written, blank, unitary square before the robot started to move onto it, then, S remains written, blank, once the robot as terminated the moving command execution.

In contrast to the BNN discussed in the chapter 2.2.1., the GABNN needs a way to test the robot environment. Actually, whereas the former is a system engineered to control any tasks we know how to broke down in sequences of its primitive commands, the latter is a system engineered to be trained to control any task, namely any sequences of primitives, its unknowable and unpredictably evolving environment requires it to perform. To this end, the GABNN tests the state of the presently observed unitary square, ie, tests whether the central square of the central component of the presently observed environment vector is written or erased. That is, in the GABNN the first input subsubvector component number sixty three value is taken as the test of the environment state. As usual, the latter value is one if the unitary square is written on and zero otherwise.

Actually, to present a *training procedure* we give the relative general assembly calling sequence, specifying either the environment vector or the first modality input subvector content, and the corresponding output subvector content. We represent the environment vector, the input subvector content, the output subvector content and the general assembly calling sequence by means of capital letters belonging to the Latin alphabet or as sequences of the latter or combination of the latter and Greek letters. To simplify the presentation, ie, to avoid the need to keep and update a table of the correspondence between the input names and the corresponding output sequences, ie, the corresponding ac-

tions, we call with the same names both the input and the output sequences, as long as the latter are given and are constants. Eg, L is both the name of the general assembly trained to control the output subvector sequence 11111 and the name of the latter sequence. To distinguish a sequence of general assembly calling sequence from one calling sequence composed by a sequence of capital letters, we bracket the latter within the curved parenthesis (). If an output subvector component content sequence is a variable, the relative general assembly calling sequence is given as a constant name, possibly enclosed within curved parenthesis, followed by a generic calling sequence name enclosed within the curly brackets {}. In other words, the sequence W{G} is a calling sequence associated to a set of general assemblies: once G is specified, W{G} becomes the calling sequence of the so pinpointed general assembly. Eg, if G ≡ **0**, W{0} is the calling sequence of the general assembly trained to control the output subvector sequence controlling the robot to write the digit zero. As we shall discuss later on, the sequence EC is the calling sequence of the general assembly trained to erase the present environment vector central component. However, we can modify it, ie, we can train the GΑBNN to generate another output subvector sequence every time its first, second or both, input subvectors component content is (EC){G}, with G an input to specify at some later time. Eg, if G ≡ W{+}, the sequence (EC){G}, ie, (EC){W{+}} is the calling sequence of the general assembly trained to control the robot, first, to erase the present central component, second, to write the symbol +.

6.2.2. *The primitive pair training.*

As we shall find many times sequences composed by the same pairs of primitives, we start with their common training procedures. That is, now we are going to discuss the training procedures relative to pairs of primitives. Because sequences like ew or lr or ww leave the robot and its environment exactly as they were before of their execution, we shall ignore them. Moreover, to simplify the presentation we shall discuss a few of them, leaving the rest to the interested reader. As before said, we identify the general assembly to be trained with the very same output subvector component sequence values its activity generates. In other words, we are going to discuss the training procedure for the GΑBNN output subvector components to assume the sequences of values needed to control the robot to execute the commands ll, rr, uu, dd, lu, ld, ul, ur, wl, el, rw, re, wu, eu, dw, de, every time the GΑBNN first modality input is ll, rr, uu, dd, lu, ld, ul, ur, wl, el, rw, re, wu, eu, dw, de. For each input-output calling sequence pair, the one step training procedure is the following:

I) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly ll, rr, uu, dd, lu, ld, ul, ur, wl, el, rw, re, wu, eu, dw, de, whichever is the case, and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the primitive l, r, u, d, l, l, u, u, w, e, r, r, w, e, d, d, calling sequence. As the primitive l, r, u, d, w, e, has terminated to control the task l, r, u, d, w, e, the second modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the primitive l, r, u, d, u, d, l, r, l, l, w, e, u, u, w, e, calling sequence. If the given training triplet [ll, l, l], [rr, r, r], [uu, u, u], [dd, d, d], [lu, l, u], [ld, l, d], [ul, u, l], [ur, u, r], [wl, w, l], [el, e, l], [rw, r, w], [re, r, e], [wu, w, u], [eu, e, u], [dw, d, w], [de, d, e], is the GΑBNN input subvector component value, with delays satisfying the constraints discussed in 5.6.14., a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule and obeying the second Hebbian rule connections belonging to the second Hebbian set between the IBNSs

identified by the triplet symbols, then, the GABNN has been trained to output the sequence ll, rr, uu, dd, lu, ld, ul, ur, wl, el, rw, re, wu, eu, dw, de, every time its first modality input is ll, rr, uu, dd, lu, ld, ul, ur, wl, el, rw, re, wu, eu, dw, de.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

6.2.3. *The macro training.*

To move from an environment vector component to the next it is convenient to train, either assisted or unassisted, the GABNN to output the sequences of five, or twenty-five, identical primitive moving commands, see also the chapter 2. 2. 2. and the Figure 2.2.. Similarly, it is convenient to have macros to write, or erase, the digits one and zero as well as the plus and the times signs and to have the macros to copy, in temporary environment vectors, the intermediate results.

6.2.3.1.1. *Training the GABNN to control the robot large movements.*

Let us discuss a procedure to train the GABNN output subvector components to assume the sequence of values needed to control the robot to execute the command LEFT, RIGHT, UP, DOWN, for short L, R, U, D, see also the macro already introduced in the chapter 2.2.2.. That is, we are going to discuss a procedure to train the GABNN to execute the primitive command sequence lllll, rrrrr, uuuuu, ddddd, respectively, every time its first modality input is L, R, U, D. Of the given four command sequences, the first two control the robot to move over the next environment vector at the left, right, of the present one. The second two move the robot writing hand on the next up, down, component. By convention, the next of the topmost, bottommost, component of a given environment vector is the bottommost, topmost, component of the next environment vector in the upward, downward, direction. As it easy to verify, the execution of the L, R, U, D commands leaves the robot over the unitary square homologous of the one it started on. For each input-output calling sequence pair, the two steps training procedure is the following:

I) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly L', R', U', D', whichever is the case, and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the primitive ll, rr, uu, dd, calling sequence. As the primitive l, r, u, d, has terminated to control the task l, r, u, d, for the second time, the first modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the general assembly ll, rr, uu, dd, calling sequence. If the input triplet [L', ll, lll, [R', rr, rrl, [U', uu, uul, [D', dd, ddl, is entered a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence llll, rrrr, uuuu, dddd, every time its second modality input is L', R', U', D'.

II) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly L, R, U, D, whichever is the case, and, with a delay no greater than the maximum delay D, the second modality input assumes the value L', R', U', D'. As the primitive l, r, u, d, has terminated to control the task l, r, u, d, for the second time, the second modality input assumes, again with a delay no greater

than the maximum delay D, the value l, r, u, d. If the input triplet [L, L', l], [R, R', r], [U, U', u], [D, D', d], is entered a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence lllll, rrrrr, uuuuu, ddddd, every time its first modality input is L, R, U, D.

As the names L, R, U, D, L', R', U', D', l, r, u, d, identify different IBNSs, actually, the GABNN training result in as many different behaviors. And each of them can be recalled independently of the others.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

In the given training procedure we modify the first step as follows:

I bis) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly L'", R'", U'", D'", whichever is the case, and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the primitive ll, rr, uu, dd, calling sequence. As the primitive l, r, u, d, has terminated to control the task l, r, u, d, for the second time, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly l, r, u, d, calling sequence. If the input triplet [L'", ll, l], [R'", rr, r], [U'", uu, u], [D'", dd, d], is entered a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence lll, rrr, uuu, ddd, every time its second modality input is L'", R'", U'", D'".

6.2.3.1.2. *Training the GABNN to control the robot extra large movements.*

Let us discuss a procedure to train the GABNN output subvector components to assume the sequence of values needed to control the robot to execute the movement λ, ρ, υ, δ, ie, to execute the command sequence LLLLL, RRRRR, UUUUU, DDDDD, respectively, every time the GABNN first modality input is λ, ρ, υ, δ. Of the given four command sequences, the first two control the robot movements toward the fifth environment vector at the left, right, of the present one. The second two control the robot movements toward the next up, down, environment vector. As it easy to verify, the execution of the λ, ρ, υ, δ, commands leaves the robot over the unitary square homologous of the one it started on. For each input-output calling sequence pair, the relative three steps training procedure is the following:

I) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly λ', ρ', υ', δ', whichever is the case, and, with a delay no greater than the maximum delay D, the second modality input assumes the value L, R, U, D. As the primitive l, r, u, d, has terminated to control the task l, r, u, d, for the fifth time, the second modality input assumes again, and again with a delay no greater than the maximum delay D, the value L, R, U, D. If the input triplet [λ', L, L], [ρ', R, R], [υ', U, U], [δ', D, D], is entered a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence lllllllll, rrrrrrrrr, uuuuuuuuu, ddddddddd, ie, LL, RR, UU, DD, every time its first modality input is λ', ρ', υ', δ'.

II) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly λ'', ρ'', υ'', δ'', whichever is the case, and, with a delay no greater than the maximum delay D, the first modality input assumes the value λ', ρ', υ', δ'. As the primitive l, r, u, d, has terminated to control the task l, r, u, d, for the tenth time, the first modality input assumes again, and again with a delay no greater than the maximum delay D, the value λ', ρ', υ', δ'. If the input triplet [λ'', λ', λ'], [ρ'', ρ', ρ'], [υ'', υ', υ'], [δ'', δ', δ'], is entered a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence lllllllllllllllllll, rrrrrrrrrrrrrrrrrrr, uuuuuuuuuuuuuuuuuuu, ddddddddddddddddddddd, ie, LLLL, RRRR, UUUU, DDDD, every time its second modality input is λ'', ρ'', υ'', δ''.

III) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly λ, ρ, υ, δ, whichever is the case, and, with a delay no greater than the maximum delay D, the second modality input assumes the value λ'', ρ'', υ'', δ''. As the primitive l, r, u, d, has terminated to control the task l, r, u, d, for the twentieth time, the second modality input assumes, again with a delay no greater than the maximum delay D, the value L, R, U, D. If the input triplet [λ, λ'', L], [ρ, ρ'', R], [υ, υ'', U], [δ, δ'', D], is entered a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence LLLLL, RRRRR, UUUUU, DDDDD, ie, lllllllllllllllllllllll, rrrrrrrrrrrrrrrrrrrrrrr, uuuuuuuuuuuuuuuuuuuuuuu, ddddddddddddddddddddddd, every time its first modality input is λ, ρ, υ, δ.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

6.2.3.1.3. *Training the GABNN to control the robot megamovements.*

Let us discuss a procedure to train the GABNN output subvector components to assume the sequence of values needed to control the robot to execute the movement Mλ, Mρ, Mυ, Mδ, ie, to execute the command sequence λλλλλ, ρρρρρ, υυυυυ, δδδδδ respectively, every time the GABNN first modality input is Mλ, Mρ, Mυ, Mδ. Of the given four command sequences, the first two control the robot movements toward the twenty fifth environment vector at the left, right, of the present one. The second two control the robot movements toward the fifth next up, down, environment vector. As it easy to verify, the execution of the Mλ, Mρ, Mυ, Mδ, commands leaves the robot over the unitary square homologous of the one it started on. For each input-output calling sequence pair, the relative three steps training procedure is the following:

I) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly Mλ', Mρ', Mυ', Mδ', whichever is the case, and, with a delay no greater than the maximum delay D, the second modality input assumes the value λ, ρ, υ, δ. As the primitive l, r, u, d, has terminated to control the last instance of the task l, r, u, d, the second modality input assumes again, and again with a delay no greater than the maximum delay D, the value λ, ρ, υ, δ. If the input triplet [Mλ', λ, λ], [Mρ', ρ, ρ], [Mυ', υ, υ], [Mδ', δ, δ], is entered a number of

times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence LLLLLLLLLL, RRRRRRRRRR, UUUUUUUUUU, DDDDDDDDDD, ie, λλ, ρρ, υυ, δδ, every time its first modality input is Mλ', Mρ', Mυ', Mδ'.

II) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly Mλ'', Mρ'', Mυ'', Mδ'', whichever is the case, and, with a delay no greater than the maximum delay D, the first modality input assumes the value Mλ', Mρ', Mυ', Mδ'. As the primitive l, r, u, d, has terminated to control the last instance of the task l, r, u, d, the first modality input assumes again, and again with a delay no greater than the maximum delay D, the value Mλ', Mρ', Mυ', Mδ'. If the input triplet [Mλ'', Mλ', Mλ'], [Mρ'', Mρ', Mρ'], [Mυ'', Mυ', Mυ'], [Mδ'', Mδ', Mδ'], is entered a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence LLLLLLLLLLLLLLLLLLLL, RRRRRRRRRRRRRRRRRRRR, UUUUUUUUUUUUUUUUUUUU, DDDDDDDDDDDDDDDDDDDD, ie, λλλλ, ρρρρ, υυυυ, δδδδ, every time its second modality input is Mλ'', Mρ'', Mυ'', Mδ''.

III) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly Mλ, Mρ, Mυ, Mδ, whichever is the case, and, with a delay no greater than the maximum delay D, the second modality input assumes the value Mλ'', Mρ'', Mυ'', Mδ''. As the primitive l, r, u, d, has terminated to control the last instance of the task l, r, u, d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value λ, ρ, υ, δ. If the input triplet [Mλ, Mλ'', λ], [Mρ, Mρ'', ρ], [Mυ, Mυ'', υ], [Mδ, Mδ'', δ], is entered a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence
LLLLLLLLLLLLLLLLLLLLLLLLLLLLLL,
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRR,
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUU,
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDD, ie, λλλλλ, ρρρρρ, υυυυυ, δδδδδ, every time its first modality input is Mλ, Mρ, Mυ, Mδ.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

Because of the incremental learning, the procedure given is composed by just one step thanks to the training discussed in the chapters 6.2.3.1.1. and 6.2.3.1.2.. Otherwise, the number of steps increases.

6.2.3.1.4. *Training the GABNN to control the robot almost megamovements.*

Let us discuss a procedure to train the GABNN output subvector components to assume the sequence of values needed to control the robot to execute the movement Mλ''LLLL, Mρ''RRRR, Mυ''RRRR, Mδ''DDDD, every time the GABNN first modality input is Mλ''', Mρ''', Mυ''', Mδ'''. Of the given four command sequences, the first two control the robot movements toward the twenty fourth environment vector at the left, right, of the present one. The second two control the robot movements toward the fourth environment vector plus four components up, down. As it easy to verify,

the execution of the Mλ, Mρ, Mυ, Mδ, commands leaves the robot over the unitary square homologous of the one it started on. For each input-output calling sequence pair, the relative training procedure is the following:

I) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly Mλ''', Mρ''', Mυ''', Mδ''', whichever is the case, and, with a delay no greater than the maximum delay D, the second modality input assumes the value Mλ'', Mρ'', M''υ, Mδ''. As the primitive l, r, u, d, has terminated to control the last instance of the task l, r, u, d, the second modality input assumes again, and again with a delay no greater than the maximum delay D, the value λ'', ρ'', υ'', δ''. If the input triplet [Mλ''', Mλ'', λ''], [Mρ''', Mρ'', ρ''], [Mυ''', Mυ'', υ''], [Mδ''', Mδ'', δ''], is entered a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence Mλ''λ'', Mρ''ρ'', Mυ''υ'', Mδ''δ'', every time its first modality input is Mλ''', Mρ''', Mυ''', Mδ'''.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

Because of the incremental learning, the procedure given is composed by just one step thanks to the training discussed in the chapters 6.2.3.1.1., 6.2.3.1.2. and 6.2.3.1.3.. Otherwise, the number of steps increases.

We can change the given procedure as follows:

I) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly Mλ''''', Mρ''''', Mυ''''', Mδ''''', whichever is the case, and, with a delay no greater than the maximum delay D, the second modality input assumes the value Mλ'', Mρ'', M''υ, Mδ''. As the primitive l, r, u, d, has terminated to control the last instance of the task l, r, u, d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value L''', R''', U''', D'''. If the input triplet [Mλ''''', Mλ'', L'''], [Mρ''''', Mρ'', R'''], [Mυ''''', Mυ", U'''], [Mδ''''', Mδ'', D'''], is entered a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence Mλ''L''', Mρ''R''', Mυ''U''', Mδ''D''', every time its first modality input is Mλ''''', Mρ''''', Mυ''''', Mδ'''''.

Of the given four command sequences, the first two control the robot movements toward the twenty third environment vector at the left, right, of the present one. The second two control the robot movements toward the fourth environment vector plus three components up, down.

6.2.3.2. *Training the GABNN to output the command write-one.*

To write the digit one on the present environment vector component, starting from its central square, the GABNN output subvector components have to be trained to assume

the sequence of values: uwdwdwu, every time its first modality input is write-one, or for short W{1}. The three steps training procedure is the following:

I) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $W\{1\}_1$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly uw calling sequence. As the primitive w has terminated to control the task w, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly dw calling sequence. If the input triplet $[W\{1\}_1$, uw, dw] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence uwdw every time its second modality input is $W\{1\}_1$.

II) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $W\{1\}_2$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly dw calling sequence. As the primitive w has terminated to control the task w, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the primitive u calling sequence. If the input triplet $[W\{1\}_2$, dw, u] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence dwu every time its second modality input is $W\{1\}_2$.

III) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly W{1} and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $W\{1\}_1$ calling sequence. As the primitive w has terminated to control the task w, for the second time, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly $W\{1\}_2$ calling sequence. If the input triplet $[W\{1\}, W\{1\}_1, W\{1\}_2]$ is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence uwdwdwu every time its first modality input is W{1}.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

6.2.3.3. *Training the GABNN to output the command erase-one.*

To erase the digit one from the present environment vector component, starting from its central square, the GABNN output subvector components have to be trained to assume the sequence of values: uededeu, every time its first modality input is erase-one, or for short E{1}. The three steps training procedure is the following:

I) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $E\{1\}_1$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly ue calling sequence. As the primitive e has terminated to control the task e, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly de calling sequence. If the input triplet $[E\{1\}_1$, ue, de] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian

connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence uede every time its second modality input is E{1}$_1$.

II) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly E{1}$_2$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly de calling sequence. As the primitive e has terminated to control the task e, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the primitive u calling sequence. If the input triplet [E{1}$_2$, de, ul is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence deu every time its second modality input is E{1}$_2$.

III) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly E{1} and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly E{1}$_1$ calling sequence. As the primitive e has terminated to control the task e, for the second time, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly E{1}$_2$ calling sequence. If the input triplet [E{1}, E{1}$_1$, E{1}$_2$] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence uededeu every time its first modality input is E{1}.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

6.2.3.4. *Training the GABNN to output the command write-zero.*

To write the digit zero on the present environment vector component, starting from its central square, the GABNN output subvector components have to be trained to assume the sequence of values: uwrwdwdwlwlwuwuwrd, every time its first modality input is write-zero, or for short W{0}. The eight steps training procedure is the following:

I) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly W{0}$_1$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly uw calling sequence. As the primitive w has terminated to control the task w, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly rw calling sequence. If the input triplet [W{0}$_1$, uw, rw] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence uwrw every time its second modality input is W{0}$_1$.

II) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly W{0}$_2$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly dw calling sequence. As the primitive w has terminated to control the task w, the first modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the general assembly dw calling sequence. If the input triplet [W{0}$_2$, dw, dwl is the value assumed by the GABNN input subvector a number of times sufficient to freeze the

Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence dwdw every time its second modality input is $W\{0\}_2$.

III) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $W\{0\}_3$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly lw calling sequence. As the primitive w has terminated to control the task w, the first modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the general assembly lw calling sequence. If the input triplet [$W\{0\}_3$, lw, lw] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence lwlw every time its second modality input is $W\{0\}_3$.

IV) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $W\{0\}_4$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly uw calling sequence. As the primitive w has terminated to control the task w, the first modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the general assembly uw calling sequence. If the input triplet [$W\{0\}_4$, uw, uw] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence uwuw every time its second modality input is $W\{0\}_4$.

V) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $W\{0\}_5$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $W\{0\}_1$ calling sequence. As the primitive w has terminated to control the task w, for the second time, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly $W\{0\}_2$ calling sequence. If the input triplet [$W\{0\}_5$, $W\{0\}_1$, $W\{0\}_2$] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence uwrwdwdw every time its first modality input is $W\{0\}_5$.

VI) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $W\{0\}_6$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $W\{0\}_3$ calling sequence. As the primitive w has terminated to control the task w, for the second time, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly $W\{0\}_4$ calling sequence. If the input triplet [$W\{0\}_6$, $W\{0\}_3$, $W\{0\}_4$] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence lwlwuwuw every time its first modality input is $W\{0\}_6$.

VII) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $W\{0\}_7$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $W\{0\}_5$ calling sequence. As the primitive w has terminated to control the task w, for the fourth time, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly $W\{0\}_6$ calling sequence. If the input triplet [$W\{0\}_7$, $W\{0\}_5$, $W\{0\}_6$] is the value assumed by the GABNN input subvector a number

of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence uwrwdwdwlwlwuwuw every time its second modality input is $W\{0\}_7$.

VIII) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly W{0} and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $W\{0\}_7$ calling sequence. As the primitive w has terminated to control the task w, for the eighth time, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly rd calling sequence. If the input triplet [W{0}, $W\{0\}_7$, rd] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence uwrwdwdwlwlwuwuwrd every time its first modality input is W{0}.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

6.2.3.5. *Training the GABNN to output the command erase-zero.*

To erase the digit zero from the present environment vector component, starting from its central square, the GABNN output subvector components have to be trained to assume the sequence of values: uerededeleleueuerd, every time its first modality input is erase-zero, or for short E{0}. The eight steps training procedure is the following:

I) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $E\{0\}_1$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly ue calling sequence. As the primitive e has terminated to control the task e, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly re calling sequence. If the input triplet [$E\{0\}_1$, ue, rel is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence uere every time its second modality input is $E\{0\}_1$.

II) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $E\{0\}_2$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly de calling sequence. As the primitive e has terminated to control the task e, the first modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the general assembly de calling sequence. If the input triplet [$E\{0\}_2$, de, del is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence dede every time its second modality input is $E\{0\}_2$.

III) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $E\{0\}_3$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly le calling sequence. As the primitive e has terminated to control the task e, the first modality input assumes again, and again with a delay no greater than the maximum delay D, the

value of the general assembly le calling sequence. If the input triplet [E{0}$_3$, le, le] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence lele every time its second modality input is E{0}$_3$.

IV) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly E{0}$_4$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly ue calling sequence. As the primitive e has terminated to control the task e, the first modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the general assembly ue calling sequence. If the input triplet [E{0}$_4$, ue, ue] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence ueue every time its second modality input is E{0}$_4$.

V) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly E{0}$_5$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly E{0}$_1$ calling sequence. As the primitive e has terminated to control the task e, for the second time, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly E{0}$_2$ calling sequence. If the input triplet [E{0}$_5$, E{0}$_1$, E{0}$_2$] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence uerededе every time its first modality input is E{0}$_5$.

VI) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly E{0}$_6$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly E{0}$_3$ calling sequence. As the primitive e has terminated to control the task e, for the second time, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly E{0}$_4$ calling sequence. If the input triplet [E{0}$_6$, E{0}$_3$, E{0}$_4$] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence leleueue every time its first modality input is E{0}$_6$.

VII) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly E{0}$_7$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly E{0}$_5$ calling sequence. As the primitive e has terminated to control the task e, for the fourth time, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly E{0}$_6$ calling sequence. If the input triplet [E{0}$_7$, E{0}$_5$, E{0}$_6$] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence uerededeleleueue every time its second modality input is E{0}$_7$.

VIII) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly E{0} and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly E{0}$_7$ calling sequence. As the primitive e has terminated to control the task e, for the eighth time,

the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly rd calling sequence. If the input triplet [E{0}, $E\{0\}_7$, rd] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence ueredededeleleueuerd every time its first modality input is E{0}.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

6.2.3.6. *Training the GABNN to output the command write-+.*

To write the plus sign + on the present environment vector component, starting from its central square, the GABNN output subvector components have to be trained to assume the sequence of values: wlwrrwldwuuwd, every time its first modality input is write-+, or for short W{+}. The six steps training procedure is the following:

I) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $W\{+\}_1$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly wl calling sequence. As the primitive l has terminated to control the task l, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly wr calling sequence. If the input triplet [$W\{+\}_1$, wl, wr] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence wlwr every time its first modality input is $W\{+\}_1$.

II) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $W\{+\}_2$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly rw calling sequence. As the primitive w has terminated to control the task w, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly ld calling sequence. If the input triplet [$W\{+\}_2$, rw, ld] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence rwld every time its first modality input is $W\{+\}_2$.

III) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $W\{+\}_3$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly wu calling sequence. As the primitive u has terminated to control the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly uw calling sequence. If the input triplet [$W\{+\}_3$, wu, uw] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence wuuw every time its first modality input is $W\{+\}_3$.

IV) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $W\{+\}_4$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $W\{+\}_3$ calling sequence. As the primitive w has terminated to control the task w, for the second

time, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the primitive d calling sequence. If the input triplet [W{+}$_4$, W{+}$_3$, dl is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence wuuwd every time its second modality input is W{+}$_4$.

V) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly W{+}$_5$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly W{+}$_1$ calling sequence. As the primitive r has terminated to control the task r, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly W{+}$_2$ calling sequence. If the input triplet [W{+}$_5$, W{+}$_1$, W{+}$_2$] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence wlwrrwld every time its second modality input is W{+}$_5$.

VI) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly W{+} and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly W{+}$_5$ calling sequence. As the primitive d has terminated to control the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly W{+}$_4$ calling sequence. If the input triplet [W{+}, W{+}$_5$, W{+}$_4$] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence wlwrrwldwuuwd every time its first modality input is W{+}.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

6.2.3.7. *Training the GABNN to output the command erase-+.*

To erase the plus sign + from the present environment vector component, starting from its central square, the GABNN output subvector components have to be trained to assume the sequence of values: elerreldeuued, every time its first modality input is erase-+, or for short E{+}. The six steps training procedure is the following:

I) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly E{+}$_1$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly el calling sequence. As the primitive l has terminated to control the task l, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly er calling sequence. If the input triplet [E{+}$_1$, el, erl is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence eler every time its first modality input is E{+}$_1$.

II) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly E{+}$_2$ and, with a delay no greater than the maximum

delay D, the second modality input assumes the value of the general assembly re calling sequence. As the primitive e has terminated to control the task e, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly ld calling sequence. If the input triplet [E{+}$_2$, re, ld] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence reld every time its first modality input is E{+}$_2$.

III) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly E{+}$_3$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly eu calling sequence. As the primitive u has terminated to control the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly ue calling sequence. If the input triplet [E{+}$_3$, eu, ue] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence euue every time its first modality input is E{+}$_3$.

IV) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly E{+}$_4$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly E{+}$_3$ calling sequence. As the primitive e has terminated to control the task e, for the second time, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the primitive d calling sequence. If the input triplet [E{+}$_4$, E{+}$_3$, dl is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence euued every time its second modality input is E{+}$_4$.

V) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly E{+}$_5$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly E{+}$_1$ calling sequence. As the primitive r has terminated to control the task r, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly E{+}$_2$ calling sequence. If the input triplet [E{+}$_5$, E{+}$_3$, E{+}$_7$] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence elerreld every time its second modality input is E{+}$_5$.

VI) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly E{+} and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly E{+}$_5$ calling sequence. As the primitive d has terminated to control the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly E{+}$_4$ calling sequence. If the input triplet [E{+}, E{+}$_5$, E{+}$_4$] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence elerreldeuued every time its first modality input is E{+}.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

6.2.3.8. *Training the GABNN to output the command write-x.*

To write the times sign x on the present environment vector component, starting from its central square, the GABNN output subvector components have to be trained to assume the sequence of values: `wluwrrwddwllwur`, every time its first modality input is write-x, or for short `W{x}`. The seven steps training procedure is the following:

I) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $W\{x\}_1$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly `wl` calling sequence. As the primitive `l` has terminated to control the task `l`, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly `uw` calling sequence. If the input triplet [$W\{x\}_1$, `wl`, `uwl` is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence `wluw` every time its first modality input is $W\{x\}_1$.

II) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $W\{x\}_2$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly `rr` calling sequence. As the primitive `r` has terminated to control the task `r`, for the second time, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly `wd` calling sequence. If the input triplet [$W\{x\}_2$, `rr`, `wdl` is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence `rrwd` every time its first modality input is $W\{x\}_2$.

III) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $W\{x\}_3$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly `dw` calling sequence. As the primitive `w` has terminated to control the task `w`, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly `ll` calling sequence. If the input triplet [$W\{x\}_3$, `dw`, `lll` is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence `dwll` every time its first modality input is $W\{x\}_3$.

IV) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $W\{x\}_4$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly `wu` calling sequence. As the primitive `u` has terminated to control the task `u`, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the primitive `r` calling sequence. If the input triplet [$W\{x\}_4$, `wu`, `rl` is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence `wur` every time its first modality input is $W\{x\}_4$.

V) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $W\{x\}_5$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $W\{x\}_1$ calling sequence. As the primitive w has terminated to control the task w, for the second time, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly $W\{x\}_2$ calling sequence. If the input triplet [$W\{x\}_5$, $W\{x\}_1$, $W\{x\}_2$] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence wluwrrwd every time its second modality input is $W\{x\}_5$.

VI) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $W\{x\}_6$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $W\{x\}_3$ calling sequence. As the primitive l has terminated to control the task l, for the second time, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly $W\{x\}_4$ calling sequence. If the input triplet [$W\{x\}_6$, $W\{x\}_3$, $W\{x\}_4$] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence dwllwur every time its second modality input is $W\{x\}_6$.

VII) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $W\{x\}$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $W\{x\}_5$ calling sequence. As the primitive d has terminated to control the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly $W\{x\}_6$ calling sequence. If the input triplet [$W\{x\}$, $W\{x\}_5$, $W\{x\}_6$] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence wluwrrwddwllwur every time its first modality input is $W\{x\}$.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

6.2.3.9. *Training the GABNN to output the command erase-x.*

To erase the times sign x from the present environment vector component, starting from its central square, the GABNN output subvector components have to be trained to assume the sequence of values: eluerreddelleur, every time its first modality input is erase-x, or for short $E\{x\}$. The seven steps training procedure is the following:

I) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $E\{x\}_1$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly el calling sequence. As the primitive l has terminated to control the task l, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly ue calling sequence. If the input triplet [$E\{x\}_1$, el, uel is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Heb-

bian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence elue every time its first modality input is $E\{x\}_1$.

II) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $E\{x\}_2$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly rr calling sequence. As the primitive r has terminated to control the task r, for the second time, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly ed calling sequence. If the input triplet [$E\{x\}_2$, rr, ed] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence rred every time its first modality input is $E\{x\}_2$.

III) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $E\{x\}_3$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly de calling sequence. As the primitive e has terminated to control the task e, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly ll calling sequence. If the input triplet [$E\{x\}_3$, de, ll] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence dell every time its first modality input is $E\{x\}_3$.

IV) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $E\{x\}_4$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly eu calling sequence. As the primitive u has terminated to control the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the primitive r calling sequence. If the input triplet [$E\{x\}_4$, eu, r] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence eur every time its first modality input is $E\{x\}_4$.

V) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $E\{x\}_5$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $E\{x\}_1$ calling sequence. As the primitive e has terminated to control the task e, for the second time, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly $E\{x\}_2$ calling sequence. If the input triplet [$E\{x\}_5$, $E\{x\}_1$, $E\{x\}_2$] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence eluerred every time its second modality input is $E\{x\}_5$.

VI) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $E\{x\}_6$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $E\{x\}_3$ calling sequence. As the primitive l has terminated to control the task l, for the second time, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly $E\{x\}_4$ calling sequence. If the input triplet [$E\{x\}_6$, $E\{x\}_3$, $E\{x\}_4$] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet sym-

bols, then, the GABNN has been trained to output the sequence delleur every time its second modality input is $E\{x\}_6$.

VII) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $E\{x\}$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $E\{x\}_5$ calling sequence. As the primitive d has terminated to control the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly $E\{x\}_6$ calling sequence. If the input triplet [$E\{x\}$, $E\{x\}_5$, $E\{x\}_6$] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence eluerreddelleur every time its first modality input is $E\{x\}$.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

6.2.3.10. *Training the GABNN to mark an environment vector component.*

If there are a number of environment vectors the robot has to operate on, it is useful for the GABNN to mark the component of the one it is presently working on. By convention, the robot marks. ie, writes on the lower left unitary square belonging to the bottommost component of the presently observed environment vector. For simplicity sake, we give the procedure for the lower component: to obtain the procedure for the remaining components, it is sufficient to change the first, and the last, two general assembly names. To mark the environment vector bottommost component, the GABNN has to learn the sequence: DDddllwrruuUU, every time its first modality input is MEV. The six steps training procedure is the following:

I) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly DWN_1 and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly dd calling sequence. As the primitive d has terminated to control the task d, for the second time, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly ll calling sequence. If the input triplet [DWN_1, dd, ll] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence ddll every time its second modality input is DWN_1.

II) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly DWN_2 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly δ' calling sequence, see the chapter 6. 2. 3. 2.. As the primitive d has terminated to control the task d, for the tenth time, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly DWN_1 calling sequence. If the input triplet [DWN_2, δ', DWN_1] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DDddll every time its first modality input is DWN_2.

III) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly UP_1 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly rr calling sequence. As the primitive r has terminated to control the task r, for the second time, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly uu calling sequence. If the input triplet [UP_1, rr, uu] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence rruu every time its first modality input is UP_1.

IV) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly UP_2 and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly UP_1 calling sequence. As the primitive u has terminated to control the task u, for the second time, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly υ' calling sequence, see the chapter 6. 2. 3. 2.. If the input triplet [UP_2, UP_1, υ'] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence rruuUU every time its second modality input is UP_2.

V) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly MEV_1 and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly DWN_2 calling sequence. As the primitive l has terminated to control the task l, for the second time, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the primitive w calling sequence. If the input triplet [MEV_1, DWN_2, w] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DDddd111w every time its second modality input is MEV_1.

VI) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly MEV and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly MEV_1 calling sequence. As the primitive w has terminated to control the task w, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly UP_2 calling sequence. If the input triplet [MEV, MEV_1, UP_2] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DDdd11wrruuUU every time its first modality input is MEV.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

Because of the incremental learning, the one given is a six steps procedure thanks to the training discussed in the chapter 6. 2. 3. 1.. Otherwise, the number of steps increases.

To mark the present component the GABNN has to learn the sequence dd11wrruu, every time its first modality input is MPC.

To mark the second from the top component the GABNN has to learn the sequence Uddllwrruu$D$, every time its first modality input is MSC.

6.2.3.11. *Training the GABNN to erase the mark from an environment vector component.*

To erase just the mark from the marked environment vector bottommost component, the GABNN has to learn the sequence: DDddllerruuUU, every time its first modality input is EEV. The procedure to erase the mark from another component is easily obtained, starting from the one given here, by changing the first, and the last, two general assembly names. The two steps training procedure is the following:

I) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly EV_1 and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly DWN_2 calling sequence, see the chapter 6.2.3.11.. As the primitive l has terminated to control the task l, for the second time, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the primitive e calling sequence. If the input triplet [EV_1, DWN_2, el is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DDddlle every time its second modality input is EV1.

II) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly EEV and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly EV1 calling sequence. As the primitive e has terminated to control the task e, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly UP2 calling sequence, see the chapter 6.2.3.11.. If the input triplet [EEV, EV1, UP2] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DDddllerruuUU every time its first modality input is EEV.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

Because of the incremental learning, the given is a two steps procedure thanks to the training discussed in the chapter 6.2.3.1. and 6.2.3.10.. Otherwise, the number of steps increases.

To erase the mark from the present component the GABNN has to learn the sequence ddllerruu, every time its first modality input is EPC.

To erase the mark from the second from the top component the GABNN has to learn the sequence Uddllerruu$D$, every time its first modality input is ESC.

6.2.3.12. *Training the GABNN to erase an environment vector component.*

To erase an environment vector component, the GABNN has to learn the sequence: ddllererererereuelelel

as $d^2l^2[erl^4eul^4eul^2[erl^4ed^2l^2$. The thirteen steps training procedure is the following:

I) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly ER_1 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly er calling sequence. As the primitive r has terminated to control the task r, the second modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the general assembly er calling sequence. If the input triplet [ER_1, er, erl is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence erer every time its first modality input is ER_1.

II) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly ER_2 and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly ER_1 calling sequence. As the primitive r has terminated to control the task r, for the second time, the first modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the general assembly ER_1 calling sequence. If the input triplet [ER_2, ER_1, ER_1] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence erererer every time its second modality input is ER_2.

III) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly ER_3 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly ER_2 calling sequence. As the primitive r has terminated to control the task r, for the fourth time, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the primitive e calling sequence. If the input triplet [ER_3, ER_2, el is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence erererere every time its first modality input is ER_3.

IV) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly ER_4 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly ER_2 calling sequence. As the primitive r has terminated to control the task r, for the fourth time, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly eu calling sequence. If the input triplet [ER_4, ER_2, eul is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence erererereu every time its first modality input is ER_4.

V) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly EL_1 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly el calling sequence. As the primitive l has terminated to control the task l, the second modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the general assembly el calling sequence. If the input triplet [EL_1, el, ell is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second

Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence elel every time its first modality input is EL_1.

VI) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly EL_2 and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly EL_1 calling sequence. As the primitive l has terminated to control the task l, for the second time, the first modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the general assembly EL_1 calling sequence. If the input triplet [EL_2, EL_1, EL_1] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence elelelel every time its second modality input is EL_2.

VIII) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly EL_3 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly EL_2 calling sequence. As the primitive l has terminated to control the task l, for the fourth time, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly eu calling sequence. If the input triplet [EL_3, EL_2,eul is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence eleleleleu every time its first modality input is EL_3.

IX) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly EC_1 and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly ER_4 calling sequence. As the primitive u has terminated to control the task u, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly EL_3 calling sequence. If the input triplet [EC_1, ER_4, EL_3] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence erererereueleleleleleu every time its second modality input is EC_1.

X) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly EC_2 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly EC_1 calling sequence. As the primitive u has terminated to control the task u, for the second time, the second modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the general assembly EC_1 calling sequence. If the input triplet [EC_2, EC_1, EC_1] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence erererereueleleleleleuererererereueleleleleleu every time its first modality input is EC_2.

XI) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly EC_3 and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly ER_3 calling sequence. As the primitive e has terminated to control the task e, for the fifth time, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly ddll, ie, DWN_1 calling sequence, see the chapter 6.2.3.11.. If the input triplet [EC_3, ER_3, DWN_1] is the value assumed by the GABNN in-

put subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence erererereddll every time its second modality input is EC_3.

XII) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly EC_4 and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly ddll, ie, DWN_1 calling sequence, see the chapter 6.2.3.11.. As the primitive l has terminated to control the task l, for the second time, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly EC_2 calling sequence. If the input triplet [EC_4, DWN_1, EC_2] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence ddllererereereueleleleleueererererereueleleleleu every time its second modality input is EC_4.

XIII) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly EC and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly EC_4 calling sequence. As the primitive u has terminated to control the task u, for the fourth time, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly EC_3 calling sequence. If the input triplet [EC, EC_4, EC_3] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, every time its first modality input is EC, the GABNN has been trained to output the sequence ddllererererereueleleleleueererererereueleleleleuererereddll.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

6.2.3.13. *Training the GABNN to erase an environment vector.*

To erase an environment vector, every time its first modality input is EV, the GABNN has to learn the sequence: DD[(EC)U]^4(EC)DD, where (EC) is the general assembly introduced in 6. 2. 3. 13. and the fourth power means the base has to be repeated four times, ie, it is the same as (EC)U(EC)U(EC)U(EC)U. The six steps training procedure is the following:

I) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly EV_1 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly EC calling sequence, see the chapter 6.2.3.13.. As the primitive l has terminated to control the task l, for the twelfth time, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly U calling sequence. If the input triplet [EV_1, EC, U] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence (EC)U every time its first modality input is EV_1.

II) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly EV_2 and, with a delay no greater than the maximum de-

lay D, the first modality input assumes the value of the general assembly EV_1 calling sequence. As the primitive u has terminated to control the last instance of the task u, the first modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the general assembly EV_1 calling sequence. If the input triplet [EV_2, EV_1, EV_1] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence (EC)U(EC)U every time its second modality input is EV_2.

III) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly EV_3 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly EV_2 calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the primitive EV_2 calling sequence. If the input triplet [EV_3, EV_2, EV_2] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence (EC)U(EC)U(EC)U(EC)U every time its first modality input is EV_3.

IV) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly EV_4 and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly EC calling sequence. As the primitive l has terminated to control the last instance of the task l, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly δ' calling sequence, see the chapter 6. 2. 3. 2.. If the input triplet [EV_4, EC, δ'] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence (EC)δ' every time its second modality input is EV_4.

V) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly EV_5 and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly δ' calling sequence. As the primitive d has terminated to control the last instance of the task d, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the primitive EV_3 calling sequence. If the input triplet [EV_5, δ', EV_3] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence δ'(EC)U(EC)U(EC)U(EC)U every time its second modality input is EV_5.

VI) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly EV and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly EV_5 calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the primitive EV_4 calling sequence. If the input triplet [EV, EV_5, EV_4] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence δ'(EC)U(EC)U(EC)U(EC)U(EC)δ', ie, DD[(EC)U]^4(EC)DD every time its first modality input is EV.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

6.2.3.14. *Training the GABNN to copy an environment vector component unitary square.*

Because we are interested in the GABNN as an autonomously self-trainable unassisted system, to copy the content of an unitary square, and a fortiori of an environment vector or of one of its components, it must have the possibility to test its content. That is, the GABNN output subvector component values must control either the primitive w or the primitive e, depending on the observed square state. In the chapter 6.1., we have defined the test of the environment state as the value assumed by the sixty-third component belonging to the GABNN first input subsubvector. The said test assumes the value one if the observed square is written on and zero otherwise. In other words, to implement the desired procedure we need two general assemblies, the first associated to a calling sequence composed by the conjunction of the name CS and of the first input subsubvector sixty-third component state. The second associated to a calling sequence composed by the conjunction of the name CS and of the logical negation of the first input subsubvector sixty-third component state. The training procedure to test a different unitary square is obtained by prefixing and suffixing, to the procedure given here, the general assembly names needed to control the movement from the presently observed unitary square to the one to test, and vice versa. Once tested, the unitary square content has to be copied elsewhere, then, the system has to return to the initial position.

That is, the GABNN output vector components must be trained to assume as value a suitable sequence of moving primitives both before and after the w or e values, ie, of the value needed to copy the unitary square content. Actually, the two said sequences are symmetric, ie, to each l, r, u, d, in the first of them it has to correspond r, l, d, u, in the second.

6.2.3.14.1. *Training the GABNN to copy an unitary square one and five vectors below, up.*

To fix the ideas, we present the training procedure to copy the presently observed unitary square content, ie, the source, in the central unitary square of the central component belonging to the environment vector five environment vectors below the present one, ie, in the destination. As the source is either written on or erased, the GABNN output vector components must be trained to assume either the value w or e, in correspondence of the destination. That is, the GABNN output vector components must be trained to assume the sequence of values MδwMυ, every time its first input subsubvector last twenty five components have the value CS and the sixty third component has the value one. Instead, the GABNN output vector components must be trained to assume the sequence of values MδeMυ, every time its first input subsubvector last twenty five components have the value CS and the sixty third component has the value zero. For each input-output calling sequence pair, the two steps training procedure is the following:

I) Both if the first modality input last twenty five components assume the value of the calling sequence corresponding to the general assembly CS_1 and if the first modality input component sixty three has the value one, then , with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly Mδ calling sequence, see the chapter 6.2.3.1.3.. As the primitive d has terminated to control the last instance of the task d, the second modality input assumes, again with a delay

no greater than the maximum delay D, the value of the general assembly w calling sequence. Otherwise, ie, both if the first modality input last twenty five components assume the value of the calling sequence corresponding to the general assembly CS_1 and if the first modality input component sixty three has the value zero, then, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly Mδ calling sequence, see the chapter 6.2.3.1.3.. As the primitive d has terminated to control the last instance of the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly e calling sequence. If both the input triplets [CS_1, Mδ, w] and [CS_1, Mδ, e] are the values assumed by the GΑBNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence Mδw, Mδe, every time its first modality input last twenty five components have the value CS_1 and the first modality input component sixty three has the value one, zero.

II) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly CS and, with a delay no greater than the maximum delay D, the first modality input assumes the value CS_1. As the primitive w, or e, has terminated to control the task w, or e, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly Mυ calling sequence. If the input triplet [CS, CS_1, Mυ] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GΑBNN has been trained to output the sequence MδwMυ, MδeMυ, every time its second modality input is CS and its first modality input component sixty three has the value one, zero.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

Because the value of any of the GABNN input components is either zero or one, once the training has been completed it is sufficient the name CS to copy the central unitary square content.

To train the GΑBNN to copy one unitary square into the central square of the fifth vector up, with respect the present unitary square, it is sufficient to interchange the υ with the δ, and vice versa, in the given procedure. The corresponding first modality input is the name CSU.

To train the GΑBNN to copy one unitary square into the central square of the next vector up, down, it is sufficient to substitute υ to Mυ, δ to Mδ, in the given procedures. The corresponding first modality input is the name CS1VU, CS1VD.

To train the GABNN to copy one unitary square into the central square of the second vector up, down, it is sufficient to substitute υυ to Mυ, δδ to Mδ, in the given procedures. The corresponding first modality input is the name CS2VU, CS2VD.

6.2.3.14.2. *Training the GABNN to copy one unitary square five vectors less one component below, up.*

We present the training procedure to copy the presently observed unitary square content, ie, the source, in the central unitary square of the central component belonging to the environment vector five environment vectors less one component below, up, the present unitary square, ie, in the destination. For the same reason given in the chapter 6.2.3.14.1., the GABNN output vector components must be trained to assume the sequence

of values Mδ'''wMυ''' every time its first modality input subsubvector last twenty five components have the value of the calling sequence CS-1 and the sixty third component has the value one. Instead, the GABNN output vector components must be trained to assume the sequence of values Mδ'''eMυ''', every time its first input subsubvector last twenty five components have the value of the calling sequence CS-1 and the sixty third component has the value zero. For each input-output calling sequence pair, the two steps training procedure is the following:

I) Both if the first modality input last twenty five components assume the value of the calling sequence corresponding to the general assembly CS_1-1 and if the first modality input component sixty three has the value one, then, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly Mδ''' calling sequence, see the chapter 6.2.3.1.4.. As the primitive d has terminated to control the last instance of the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly w calling sequence. Otherwise, ie, both if the first modality input last twenty five components assume the value of the calling sequence corresponding to the general assembly CS_1-1 and if the first modality input component sixty three has the value zero, then, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly Mδ''' calling sequence, see the chapter 6.2.3.1.4.. As the primitive d has terminated to control the last instance of the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly e calling sequence. If both the input triplets [CS_1-1, Mδ''', w] and [CS_1-1, Mδ''', e] are the values assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence Mδ'''w, Mδ'''e, every time its first modality input last twenty five components have the value CS_1 and the first modality input component sixty three has the value one, zero.

II) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly CS-1 and, with a delay no greater than the maximum delay D, the first modality input assumes the value CS_1-1. As the primitive w, e, has terminated to control the task w, e, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly Mυ''' calling sequence. If the input triplet [CS-1, CS_1-1, Mυ'''] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence Mδ'''wMυ''', Mδ'''eMυ''', every time its second modality input is CS-1 and its first modality input component sixty three has the value one, zero.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

Because the value of any of the GABNN input components is either zero or one, once the training has been completed it is sufficient the name CS-1 to copy the central unitary square content.

To train the GABNN to copy one unitary square into the central square of the fifth vector less one component up, with respect the present unitary square, it is sufficient to interchange the Mυ'" with the Mδ'", and vice versa, in the given procedure. The corresponding first modality input last twenty five components have as value the name CSU-1.

To train the GABNN to copy one unitary square into the central square of the fifth vector less two components down, up, with respect the present unitary square, it is sufficient to substitute both Mυ'" with Mυ"", Mδ''", and Mδ'" with Mδ''", Mυ"", in the given procedure. The corresponding first modality input last twenty five components have value the name CSD-2, CSU-2.

6.2.3.14.3. *Training the GABNN to copy one unitary square on the next component at the left.*

We present the training procedure to copy the presently observed unitary square content, ie, the source, in the central unitary square of the central component belonging to the next environment vector at the left of the present one, ie, in the destination. As the source is either written on or erased, the GABNN output vector components must be trained to assume either the value w or e, in correspondence of the destination. That is, the GABNN output vector components must be trained to assume the sequence of values LwR, every time its first input subsubvector last twenty five components have the value CSL and the sixty third component has the value one. Instead, the GABNN output vector components must be trained to assume the sequence of values LeR, every time its first input subsubvector last twenty five components have the value CSL and the sixty third component has the value zero.

In conclusion, to train the GABNN to control the robot movements depending on the environment state it has to learn what to do in both conditions. Accordingly, it has to trained, first, in a two steps procedure and, second, again in one composed by two steps, thanks to the incremental learning properties. The two training procedures are the followings:

A) First Procedure

I) Both the first modality input last twenty five components assume the value of the calling sequence corresponding to the general assembly CSL_1 and the first modality input component sixty three has the value one, then , with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly L calling sequence. As the primitive l has terminated to control the last instance of the task l, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly w calling sequence. If the input triplet [CSL_1, L, wl is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence Lw every time its first modality input last twenty five components have the value CSL_1 and the first modality input component sixty three has the value one.

II) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly CSL and, with a delay no greater than the maximum delay D, the first modality input assumes the value CSL_1. As the primitive w has terminated to control the task w the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly R calling sequence. If the input triplet [CSL, CSL_1, Rl is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence LwR every time its second modality input is CSL and its first modality input component sixty three has the value one.

B) Second Procedure

I) Both the first modality input last twenty five components assume the value of the calling sequence corresponding to the general assembly CSL_2 and the first modality input component sixty three has the value zero, then, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly L calling sequence. As the primitive l has terminated to control the last instance of the task l, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly e calling sequence. If the input triplet [CSL_2, L, e] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence e every time its first modality input last twenty five components have the value CSL_2 and the first modality input component sixty three has the value zero.

II) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly CSL and, with a delay no greater than the maximum delay D, the first modality input assumes the value CSL_2. As the primitive e has terminated to control the task e the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly R calling sequence. If the input triplet [CSL, CSL_2, R] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence LeR, every time its second modality input is CSL and its first modality input component sixty three has the value one, zero.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

Because the value of any of the GABNN input components is either zero or one, once the training has been completed it is sufficient the name CSL to copy the central unitary square content.

To train the GABNN to copy one unitary square into the central square of the next vector at the right, it is sufficient to interchange L with R, and vice versa, in the given procedure. The corresponding first modality input is the name CSR.

To train the GABNN to copy one unitary square into the central square of the component below the present one, it is sufficient to substitute D, U, to L, R, in the given procedure. The corresponding first modality input is the name C1D.

To train the GABNN to copy one unitary square into the central square of the component on top of the present one, it is sufficient to substitute U, D, to L, R, in the given procedure. The corresponding first modality input is the name C1U.

To train the GABNN to copy one unitary square into the central square of the third component below the present one, it is sufficient to substitute DDD, UUU, to L, R, in the given procedure. The corresponding first modality input is the name C3D.

To train the GABNN to copy one unitary square into the central square of the third component on top of the present one, it is sufficient to substitute UUU, DDD, to L, R, in the given procedure. The corresponding first modality input is the name C3U.

To train the GABNN to copy one unitary square into the central square of the fourth component below the present one, it is sufficient to substitute DDDD, UUUU, to L, R, in the given procedure. The corresponding first modality input is the name C4D.

To train the GABNN to copy one unitary square into the central square of the fourth component on top of the present one, it is sufficient to substitute UUUU, DDDD, to L, R, in the given procedure. The corresponding first modality input is the name C4U.

6.2.3.15. *Training the GABNN to copy an environment vector component.*

To complement the procedure discussed in the chapter 6.2.3.14., we present the training procedure for the control of the robot, first, to copy the content of the source, namely the content of one component of the presently observed environment vector, into the destination, namely into the corresponding component belonging to the environment vector five environment vectors below the present one, second, to return in the initial position. To fix the ideas, let the source be the present environment vector central component. To copy the content from a different component we need both to prefix, to the procedure given below, as many moving instruction as needed to move from the central component to the new component. And suffix, to the very same procedure, as many moving instruction as needed to move to the central component from the very same new component.

Every time the first input subsubvector components assume the value CC, the GABNN output vector components must be trained to assume the sequence of values $\mathtt{ddll[(CS)r]^4(CS)ul(CS)ll^4(CS)ul^2(CS)rl^4(CS)ddll}$, for the notation see the chapters 6.2.3.12. and 6.2.3.13.. The sixteen steps training procedure is the following:

I) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly CR_1 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly CS calling sequence. As the primitive u has terminated to control the task u, for the twenty fifth time, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly r calling sequence. If the input triplet $[CR_1\ CS\ rl$ is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence (CS)r every time its first modality input is CR_1.

II) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly CR_2 and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly CR_1 calling sequence. As the primitive r has terminated to control the task r, the first modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the general assembly CR_1 calling sequence. If the input triplet $[CR_2\ CR_1\ CR_1]$ is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence (CS)r(CS)r every time its second modality input is CR_2.

III) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly CR_3 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly CR_2 calling sequence. As the primitive r has terminated to control the task r, for the second time, the second modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the primitive CR_2 calling sequence. If the input triplet $[CR_3\ CR_2\ CR_2]$ is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence (CS)r(CS)r(CS)r(CS)r, ie, $[(CS)r]^4$ every time its first modality input is CR_3.

IV) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly CR_4 and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly CR_3 calling sequence. As the primitive r has terminated to control the task r, for the fourth time, the first modality input assumes, again with a delay no greater than the maximum delay D,

the value of the general assembly CS calling sequence. If the input triplet [CR_4 CR_3 CS] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence [(CS)r]^4(CS) every time its second modality input is CR_4.

V) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly CR_5 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly CR_4 calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly u calling sequence. If the input triplet [CR_5 CR_4 u] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence [(CS)r]^4(CS)u every time its first modality input is CR_5.

VI) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly CL_1 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly CS calling sequence. As the primitive u has terminated to control the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly 1 calling sequence. If the input triplet [CL_1 CS 1] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence (CS)1 every time its first modality input is CL_1.

VII) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly CL_2 and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly CL_1 calling sequence. As the primitive 1 has terminated to control the task 1, the first modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the general assembly CL_1 calling sequence. If the input triplet [CL_2 CL_1 CL_1] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence (CS)1(CS1) every time its second modality input is CL_2.

VIII) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly CL_3 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly CL_2 calling sequence. As the primitive 1 has terminated to control the task 1, for the second time, the second modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the general assembly CL_2 calling sequence. If the input triplet [CL_3 CL_2 CL_2] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence (CS)1(CS)1(CS)1(CS)1, ie, [(CS)1]^4 every time its first modality input is CL_3.

IX) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly CL_4 and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly CL_3 calling sequence. As the primitive 1 has terminated to control the task 1, for the fourth time, the first modality input assumes, again with a delay no greater than the maximum delay D,

the value of the general assembly CS calling sequence. If the input triplet [CL_4 CL_3 CS] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence [(CS)l]^{4}CS every time its second modality input is CL_4.

X) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly CL_5 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly CL_4 calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly u calling sequence. If the input triplet [CL_5 CL_4 u] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence [(CS)l]^4(CS)u every time its first modality input is CL_5.

XI) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly CC_1 and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly CR_5 calling sequence. As the primitive u has terminated to control the last instance of the task u, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly CL_5 calling sequence. If the input triplet [CC_1 CR_5 CL_5] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence [(CS)r]^4(CS)u[(CS)l]^4(CS)u every time its second modality input is CC_1.

XII) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly CC_2 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly CC_1 calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the general assembly CC_1 calling sequence. If the input triplet [CC_2 CC_1 CC_1] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence [[(CS)r]^4(CS)u[(CS)l]^4(CS)u]^2 every time its first modality input is CC_2.

XIII) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly CC_3 and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly CC_2 calling sequence. As the primitive u has terminated to control the last instance of the task u, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly CR_3 calling sequence. If the input triplet [CC_3 CC_2 CR_3] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, every time its second modality input is CC_3, the GABNN has been trained to output the sequence [[(CS)r]^4(CS)u[(CS)l]^4(CS)u]^2[(CS)r]^4.

XIV) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly CC_4 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly ddll, ie, DWN_1 calling sequence, see the chapter 6. 2. 3. 11.. As the primitive l has terminated to

control the task l, for the second time, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly CC_3 calling sequence. If the input triplet [CC_4 DWN_1 CC_3] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, every time its first modality input is CC_4, the GABNN has been trained to output the sequence ddll[[(CS)r]^4(CS)u[(CS)l]^4(CS)u]^2[(CS)r]^4.

XV) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly CC_5 and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly CC_4 calling sequence. As the primitive r has terminated to control the last instance of the task r, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly CS calling sequence. If the input triplet [CC_5 CC_4 CS] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence ddll[[(CS)r]^4(CS)u[(CS)l]^4(CS)u]^2[(CS)r]^4(CS) every time its second modality input is CC_5.

XVI) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly CC and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly CC_5 calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly DWN_1 calling sequence. If the input triplet [CC_5 CC_4 DWN_1] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence ddll[[(CS)r]^4(CS)u[(CS)l]^4(CS)u]^2[(CS)r]^4(CS)ddll every time its first modality input is CC.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

To train the GABNN to copy the present component content into the next component below, on top of, it it is sufficient to substitute CS with C1D C1U, in the given procedure. The corresponding first modality input last twenty five components have as value the name CC1D CC1U.

To train the GABNN to copy the present component content into the third component below, on top of, it it is sufficient to substitute CS with C3D C3U, in the given procedure. The corresponding first modality input last twenty five components have as value the name CC3D CC3U.

To train the GABNN to copy the present component content into the fourth component below, on top of, it it is sufficient to substitute CS with C4D C4U, in the given procedure. The corresponding first modality input components comprised between the twenty sixth and the fiftieth have as value the name CC4D CC4U.

The copy procedures so far introduced stop on the original component. To have them terminating on a different component, it is sufficient to juxtapose to them the right instructions. Eg, once copied a component into the fourth component up, to erase the calling sequence and then to stop on the third component up at the right it is sufficient the sequence UUUR. So the above given output sequence ddll[[(CC4U)r]^4(CC4U)u[(CC4U)l]^4(CC4U)u]^2[(CC4U)r]^4(CC4U)ddll

now becomes dd11[[(CC4U)r]^4(CC4U)u[(CC4U)l]^4(CC4U)u]^2[(CC4U)r]^4 (CC4U)dd11U(EC)UUR, ie, the new sequence the GABNN has to generate every time its input subvector components comprised between the twenty sixth and the fiftieth have as value the calling sequence CC4UR3U.

To train the GABNN to copy the present component content into the component one vector below, up, it is sufficient to substitute CS with CS1VD, CS1VU, in the given procedures. The corresponding first modality input last twenty five components have as value the name CC1VD, CC1VU.

To train the GABNN to copy the present component content into the component two vectors below, up, it is sufficient to substitute CS with CS2VD, CS2VU, in the given procedures. The corresponding first modality input last twenty five components have as value the name CC2VD, CC2VU.

To train the GABNN to copy the present component content into the component four vectors and three components below, up, it is sufficient to substitute CS with CSD-2, CSU-2, in the given procedure. The corresponding first modality input last twenty five components have as content the name CCD-2, CCU-2.

To train the GABNN to copy the present component content into the component four vectors and four components below, up, it is sufficient to substitute CS with CS-1, CSU-1, in the given procedure. The corresponding first modality input last twenty five components have as content value the name CC-1, CCU-1.

To train the GABNN to copy the present component content five vector up, with respect the present component, it is sufficient to substitute CS with CSU, in the given procedure. The corresponding first modality input last twenty five components have as content value the name CCU.

To train the GABNN to copy the present component content in the next component at the left, right, of the present one, it is sufficient to substitute CS with CSL, CSR, in the given procedure. The corresponding first modality input last twenty five components have as content value the name CCL, CCR.

6.2.3.16. *Training the GABNN to copy an environment vector.*

Let us call source the present environment vector and destination the fifth environment vector below the present one. To copy the source into the destination, every time its first modality input is CV, the GABNN has to learn the sequence: DD[(CC)U]^4(CC)DD, where CC is the general assembly introduced in 6.2.3.15. and the fourth power means the base has to be repeated four times, ie, [(CC)U]^4 is the same as (CC)U(CC)U(CC)U(CC)U. The six steps training procedure is the following:

I) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly CV_1 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly CC calling sequence, see the chapter 6.2.3.16.. As the primitive 1 has terminated to control the last instance of the task 1, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly U calling sequence. If the input triplet [CV_1 CC U] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence (CC)U every time its first modality input is CV_1.

II) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly CV_2 and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly CV_1 calling sequence. As the primitive u has terminated to control the last instance of the task u, the

first modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the general assembly CV_1 calling sequence. If the input triplet $[CV_2\ CV_1\ CV_1]$ is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence $[(CC)U]^2$ every time its second modality input is CV_2.

III) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly CV_3 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly CV_2 calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the primitive CV_2 calling sequence. If the input triplet $[CV_3\ CV_2\ CV_2]$ is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence $[(CC)U]^4$ every time its first modality input is CV_3.

IV) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly CV_4 and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly CC calling sequence. As the primitive 1 has terminated to control the last instance of the task 1, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly δ' calling sequence, see the chapter 6. 2. 3. 1.. If the input triplet $[CV_4\ CC\ \delta']$ is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence $(CC)\delta'$ every time its second modality input is CV_4.

V) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly CV_5 and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly δ' calling sequence. As the primitive d has terminated to control the last instance of the task d, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the primitive CV_3 calling sequence. If the input triplet $[CV_5,\ \delta',\ CV_3]$ is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence $\delta'[(CC)U]^4$ every time its second modality input is CV_5.

VI) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly CV and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly CV_6 calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the primitive CV_4 calling sequence. If the input triplet $[CV_6,\ CV_5,\ CV_4]$ is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence $\delta'(CC)U(CC)U(CC)U(CC)U(CC)\delta'$, ie, $DD[(CC)U]^4(CC)DD$ every time its first modality input is CV.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

6.2.3.17. *The training procedures implemented.*

As an help for the reader, we give the names of the general assemblies introduced in the training procedures. In the Table 6.1., first, for each entry we give the name of the general assembly, ie, the GABNN input subsubvector component values, for the output subvector to output, after a delay, the Table 6.1. each entry second component, ie, the sequence of primitive and general assembly names controlling the action described as the Table 6.1. each entry third component. The chapter where the relative training procedure has been described is given as the Table 6. 1. each entry last element.

TABLE 6. 1.

C1D & the input subvector sixty third component is one ⇔ DwU ⇔ the present square is copied in the component below, 6.2.3.14.3.

C1D & the input subvector sixty third component is zero ⇔ DeU ⇔ the present square is copied in the component below, 6.2.3.14.3.

C1U & the input subvector sixty third component is one ⇔ UwD ⇔ the present square is copied in the next component up, 6.2.3.14.3.

C1U & the input subvector sixty third component is zero ⇔ UeD ⇔ the present square is copied in the next component up, 6.2.3.14.3.

C3D & the input subvector sixty third component is one ⇔ DDDwUUU ⇔ the present square is copied three components below, 6.2.3.14.3.

C3D & the input subvector sixty third component is zero ⇔ DDDeUUU ⇔ the present square is copied three components below, 6.2.3.14.3.

C3U & the input subvector sixty third component is one ⇔ UUUwDDD ⇔ the present square is copied three components up, 6.2.3.14.3.

C3U & the input subvector sixty third component is zero ⇔ UUUeDDD ⇔ the present square is copied three components up, 6.2.3.14.3.

C4D & the input subvector sixty third component is one ⇔ DDDDwUUUU ⇔ the present square is copied four components below, 6.2.3.14.3.

C4D & the input subvector sixty third component is zero ⇔ DDDDeUUUU ⇔ the present square is copied four components below, 6.2.3.14.3.

C4U & the input subvector sixty third component is one ⇔ UUUUwDDDD ⇔ the present square is copied four components up, 6.2.3.14.3.

C4U & the input subvector sixty third component is zero ⇔ UUUUeDDDD ⇔ the present square is copied four components up, 6.2.3.14.3.

CC ⇔ $ddl[[(CS)r]^4(CS)u[(CS)l]^4(CS)u]^2[(CS)r]^4(CS)ddl$ ⇔ the presently observed central component is copied five vectors down, 6.2.3.15.

CCL ⇔ $ddl[[(CSL)r]^4(CSL)u[(CSL)l]^4(CSL)u]^2[(CSL)r]^4(CSL)ddl$ ⇔ the presently observed central component is copied at the left, 6.2.3.15.

CCR ⇔ $ddl[[(CSR)r]^4(CSR)u[(CSR)l]^4(CSR)u]^2[(CSR)r]^4(CSR)ddl$ ⇔ the presently observed central component is copied at the right, 6.2.3.15.

CCU ⇔ $ddl[[(CSU)r]^4(CSU)u[(CSU)l]^4(CSU)u]^2[(CSU)r]^4(CSU)ddl$ ⇔ the presently observed central component is copied five vectors up, 6.2.3.15.

CC1D ⇔ dd11[[(C1D)r]⁴(C1D)u[(C1D)l]⁴(C1D)u]²[(C1D)r]⁴(C1D)dd11 ⇔ the presently observed central component is copied one component down, 6.2.3.15.

CC1U ⇔ dd11[[(C1U)r]⁴(C1U)u[(C1U)l]⁴(C1U)u]²[(C1U)r]⁴(C1U)dd11 ⇔ the presently observed central component is copied one component up, 6.2.3.15.

CC1VD⇔dd11[[(CS1VD)r]⁴(CS1VD)u[(CS1VD)l]⁴(CS1VD)u]²[(CS1VD)r]⁴(CS1VD)dd11 ⇔ the presently observed central component is copied one vector down, 6.2.3.15.

CC1VU⇔dd11[[(CS1VU)r]⁴(CS1VU)u[(CS1VU)l]⁴(CS1VU)u]²[(CS1VU)r]⁴(CS1VU)dd11 ⇔ the presently observed central component is copied one vector up, 6.2.3.15.

CC2VD⇔dd11[[(CS2VD)r]⁴(CS2VD)u[(CS2VD)l]⁴(CS2VD)u]²[(CS2VD)r]⁴(CS2VD)dd11 ⇔ the presently observed central component is copied one vector down, 6.2.3.15.

CC2VU⇔dd11[[(CS2VU)r]⁴(CS2VU)u[(CS2VU)l]⁴(CS2VU)u]²[(CS2VU)r]⁴(CS2VU)dd11 ⇔ the presently observed central component is copied one vector up, 6.2.3.15.

CC3D ⇔ dd11[[(C3D)r]⁴(C3D)u[(C3D)l]⁴(C3D)u]²[(C3D)r]⁴(C3D)dd11 ⇔ the presently observed central component three components down, 6.2.3.15.

CC3U ⇔ dd11[[(C3U)r]⁴(C3U)u[(C3U)l]⁴(C3U)u]²[(C3U)r]⁴(C3U)dd11 ⇔ the presently observed central component is copied three components up, 6.2.3.15.

CC4D ⇔ dd11[[(C4D)r]⁴(C4D)u[(C4D)l]⁴(C4D)u]²[(C4D)r]⁴(C4D)dd11 ⇔ the presently observed central component is copied four components down, 6.2.3.15.

CC4U ⇔ dd11[[(C4U)r]⁴(C4U)u[(C4U)l]⁴(C4U)u]²[(C4U)r]⁴(C4U)dd11 ⇔ the presently observed central component is copied four components up, 6.2.3.15.

CC4UR3U⇔dd11[[(CC4U)r]⁴(CC4U)u[(CC4U)l]⁴(CC4U)u]²[(CC4U)r]⁴(CC4U)dd11UUUR ⇔ the presently observed central component is copied four components up and stops on the third component up at the right, 6.2.3.15.

CC-1 ⇔ dd11[[(CS-1)r]⁴(CS-1)u[(CS-1)l]⁴(CS-1)u]²[(CS-1)r]⁴(CS-1)dd11 ⇔ the presently observed central component is copied five vectors less one component down, 6.2.3.15.

CCD-2 ⇔ dd11[[(CSD-2)r]⁴(CSD-2)u[(CSD-2)l]⁴(CSD-2)u]²[(CSD-2)r]⁴(CSD-2)dd11 ⇔ the presently observed central component is copied five vectors less two components down, 6.2.3.15.

CCU-1 ⇔ dd11[[(CSU-1)r]⁴(CSU-1)u[(CSU-1)l]⁴(CSU-1)u]²[(CSU-1)r]⁴(CSU-1)dd11 ⇔ the presently observed central component is copied five vectors less one component up, 6.2.3.15.

CCU-2 ⇔ dd11[[(CSU-2)r]⁴(CSU-2)u[(CSU-2)l]⁴(CSU-2)u]²[(CSU-2)r]⁴(CSU-2)dd11 ⇔ the presently observed central component is copied five vectors less two components up, 6.2.3.15.

CS & the input subvector sixty third component is one ⇔ MδwMυ ⇔ the central square is copied five vectors down, 6.2.3.14.1.

CS & the input subvector sixty third component is zero ⇔ MδeMυ ⇔ the central square is copied five vectors down, 6.2.3.14.1.

CSD-2 & the input subvector sixty third component is one ⇔ Mδ''''wMυ'''' ⇔ the central square is copied twenty three components down, 6.2.3.14.3.

CSD-2 & the input subvector sixty third component is zero ⇔ Mδ'''eMυ'''' ⇔ the central square is copied twenty three components down, 6.2.3.14.3.

CSL & the input subvector sixty third component is one ⇔ LwR ⇔ the central square is copied at the left, 6.2.3.14.3.

CSL & the input subvector sixty third component is zero ⇔ LeR ⇔ the central square is copied at the left, 6.2.3.14.3.

CSR & the input subvector sixty third component is one ⇔ RwL ⇔ the central square is copied at the right, 6.2.3.14.3.

CSR & the input subvector sixty third component is zero ⇔ ReL ⇔ the central square is copied at the right, 6.2.3.14.3.

CS1VD & the input subvector sixty third component is one ⇔ δwυ ⇔ the central square is copied one vector down, 6.2.3.14.1.

CS1VD & the input subvector sixty third component is zero ⇔ δeυ ⇔ the central square is copied one vector down, 6.2.3.14.1.

CS1VU & the input subvector sixty third component is one ⇔ υwδ ⇔ the central square is copied one vector up, 6.2.3.14.1.

CS1VU & the input subvector sixty third component is zero ⇔ υeδ ⇔ the central square is copied one vector up, 6.2.3.14.1

CS2VU & the input subvector sixty third component is one ⇔ υυwδδ ⇔ the central square is copied two vectors up, 6.2.3.14.1.

CS2VU & the input subvector sixty third component is zero ⇔ υυeδδ ⇔ the central square is copied two vectors up, 6.2.3.14.1

CS2VD & the input subvector sixty third component is one ⇔ δδwυυ ⇔ the central square is copied two vectors down, 6.2.3.14.1.

CS2VD & the input subvector sixty third component is zero ⇔ δδeυυ ⇔ the central square is copied two vectors down, 6.2.3.14.1.

CSU & the input subvector sixty third component is one ⇔ MυwMδ ⇔ the central square is copied five vectors up, 6.2.3.14.1.

CSU & the input subvector sixty third component is zero ⇔ MυeMδ ⇔ the central square is copied five vectors up, 6.2.3.14.1.

CS-1 & the input subvector sixty third component is one ⇔ Mδ'''wMυ''' ⇔ the central square is copied twenty four components down, 6.2.3.14.3.

CS-1 & the input subvector sixty third component is zero ⇔ Mδ'''eMυ''' ⇔ the central square is copied twenty four components down, 6.2.3.14.3.

CSU-1 & the input subvector sixty third component is one ⇔ Mυ'''wMδ''' ⇔ the central square is copied twenty four components up, 6.2.3.14.3.

CSU-1 & the input subvector sixty third component is zero ⇔ Mυ'''eMδ''' ⇔ the central square is copied twenty four components up, 6.2.3.14.3.

CSU-2 & the input subvector sixty third component is one ⇔ Mυ''''wMδ'''' ⇔ the central square is copied twenty three components up, 6.2.3.14.3.

CSU-2 & the input subvector sixty third component is zero ⇔ Mυ'' '''eMδ ''' ⇔ the central square is copied twenty three components up, 6.2.3.14.3.

CV ⇔ DD[(CC)U]^4(CC)DD ⇔ the vector is copied down, 6.2.3.16.

D ⇔ ddddd ⇔ to move down on the next component, 6.2.3.1.1.

D'" ⇔ ddd ⇔ to move down three squares, 6.2.3.1.1.

D' ⇔ dddd ⇔ to move down four squares, 6.2.3.1.1.

δ' ⇔ DD ⇔ to move down on the second component, 6.2.3.1.2.

δ'' ⇔ δ'δ' ⇔ to move down on the fourth component, 6.2.3.1.2.

δ ⇔ DDDDD ⇔ to move down on the next vector, 6.2.3.1.2.

EC ⇔ $d^2l^2[[er]^4eu[el]^4eu]^2[er]^4ed^2l^2$ ⇔ to erase the component, 6.2.3.12.

EEV ⇔ DDddllerruuUU ⇔ to erase the mark from the bottommost component, 6.2.3.11.

EPC ⇔ ddllerruu ⇔ to erase the mark from the present component, 6.2.3.11.

ESC ⇔ UddllerruuD ⇔ to erase the mark from the second component from the top, 6.2.3.11.

EV ⇔ $DD[(EC)U]^4(EC)DD$ ⇔ to erase the vector, 6.2.3.13.

E{x} ⇔ eluerreddelleur ⇔ to erase the times sign, 6.2.3.9.

E{+} ⇔ elerreldeuued ⇔ to erase the plus sign, 6.2.3.7.

E{0} ⇔ ueredede leleueuerd ⇔ to erase zero, 6.2.3.5.

E{1} ⇔ uededeu ⇔ to erase one, 6.2.3.3.

L ⇔ lllll ⇔ to move left on the next vector, 6.2.3.1.1.

L'" ⇔ lll ⇔ to move left three squares, 6.2.3.1.1.

L' ⇔ llll ⇔ to move left four squares, 6.2.3.1.1.

λ' ⇔ LL ⇔ to move left on the second component, 6.2.3.1.2.

λ'' ⇔ λ'λ' ⇔ to move left on the fourth component, 6.2.3.1.2.

λ ⇔ LLLLL ⇔ to move left on the fifth vector, 6.2.3.1.2.

ℑ & in the input subvector there is at least one component different from zero, between the fifty first and the seventy fifth ⇔ UU(E{+})(E{x})L(W{+})(W{x})DD ⇔ to move on the component at the left, 6.4.1.

ℑ & all the input subvector components between the fifty first and the seventy fifth are zero ⇔ UU(E{0})ll(E{1})DD ⇔ to terminate the leftward movement, 6.4.1.

Mδ ⇔ δδδδδ ⇔ to move down on the fifth vector, 6.2.3.1.3.

Mδ' ⇔ δδ ⇔ to move down on the second vector, 6.2.3.1.3.

Mδ'' ⇔ δδδδ ⇔ to move down on the fourth vector, 6.2.3.1.3.

Mδ'''' ⇔ Mδ''D'''' ⇔ to move down on the twenty third component, 6.2.3.1.4.

Mδ''' ⇔ Mδ''DDDD ⇔ to move down on the twenty fourth component, 6.2.3.1.4.

MEV ⇔ DDddllwrruuUU ⇔ to mark the environment vector bottommost component, 6.2.3.10.

Mλ ⇔ λλλλλ ⇔ to move left on the twenty fifth vector, 6.2.3.1.3.

Mλ' ⇔ λλ ⇔ to move left on the tenth vector, 6.2.3.1.3.

Mλ'' ⇔ λλλλ ⇔ to move left on the twentieth vector, 6.2.3.1.3.

Mλ'''' ⇔ Mλ''L'''' ⇔ to move left on the twenty third component, 6.2.3.1.4.

Mλ''' ⇔ Mλ''LLLL ⇔ to move left on the twenty fourth component, 6.2.3.1.4.

MPC ⇔ ddllerruu ⇔ to mark the present component, 6.2.3.11.

Mρ ⇔ ρρρρρ ⇔ to move right on the twenty fifth vector, 6.2.3.1.3.

Mρ' ⇔ ρρ ⇔ to move right on the tenth vector, 6.2.3.1.3.

Mρ'' ⇔ ρρρρ ⇔ to move right on the twentieth vector, 6.2.3.1.3.

Mρ'''' ⇔ Mρ''R'''' ⇔ to move right on the twenty third component, 6.2.3.1.4.

Mρ''' ⇔ Mρ''RRRR ⇔ to move right on the twenty fourth component, 6.2.3.1.4.

MSC ⇔ UddllerruuD ⇔ to mark the second component from the top, 6.2.3.11.

Mυ ⇔ υυυυυ ⇔ to move up on the fifth vector, 6.2.3.1.3.

Mυ' ⇔ υυ ⇔ to move up on the second vector, 6.2.3.1.3.

Mυ'' ⇔ υυυυ ⇔ to move up on the fourth vector, 6.2.3.1.3.

Mυ''U'''' ⇔ Mυ'''' ⇔ to move up on the twenty third component, 6.2.3.1.4.

Mυ''' ⇔ Mυ''UUUU ⇔ to move up on the twenty fourth component, 6.2.3.1.4.

R ⇔ rrrrr ⇔ to move right on the next vector, 6.2.3.1.1.

R'" ⇔ rrr ⇔ to move right three squares, 6.2.3.1.1.

R' ⇔ rrrr ⇔ to move right four squares, 6.2.3.1.1.

ρ' ⇔ RR ⇔ to move right on the second component, 6.2.3.1.2.

ρ'' ⇔ ρ'ρ' ⇔ to move right on the fourth component, 6.2.3.1.2.

ρ ⇔ RRRRR ⇔ to move right on the fifth vector, 6.2.3.1.2.

ℜ & in the input subvector there is at least one component different from zero, between the fifty first and the seventy fifth ⇔ UU(E{0})ll(E{1})R(W{0})ll(W{1})DD ⇔ to move on the component at the right, 6.4.1.

ℜ & all the input subvector components between the fifty first and the seventy fifth are zero ⇔ UU(E{+})(E{x})DD ⇔ to terminate the rightward movement, 6.4.1.

U ⇔ uuuuu ⇔ to move up on the next component, 6.2.3.1.1.

U'" ⇔ uuu ⇔ to move up three squares, 6.2.3.1.1.

U' ⇔ uuuu ⇔ to move up four squares, 6.2.3.1.1.

υ' ⇔ UU ⇔ to move up on the second component, 6.2.3.1.2.

υ'' ⇔ υ'υ' ⇔ to move up on the fourth component, 6.2.3.1.2.

υ ⇔ UUUUU ⇔ to move up on the next vector, 6.2.3.1.2.

W{x} ⇔ wluwrrwddwllwur ⇔ to write the times sign, 6.2.3.8.

W{+} ⇔ wlwrrwldwuuwd ⇔ to write the plus sign, 6.2.3.6.

W{0} ⇔ uwrwdwdwlwlwuwuwrd ⇔ to write the digit zero, 6.2.3.4.

W{1} ⇔ uwdwdwu ⇔ to write the digit one, 6.2.3.2.

Table 6. 1.. *The identifiers of the general assemblies introduced in the auxiliary training procedures.* For each entry, from the left to the right, first, we give the name of the general assembly the GA BNN input subsubvector components have to assume, for the output subvector to output, after a delay, the sequence of primitive and general assembly names, namely the table each entry second component, controlling the action described as the table each entry third component. The chapter where the relative training procedure has been described is given as the table each entry last element.

6.3. Training the GABNN to compute the addition.

For simplicity sake, see also the chapter 2.2.2., we assume the operand digits to be written one below the other. Actually, the first, second, operand digits are written in the third, fourth, component, from the top, of consecutive environment vectors. The plus sign on top of the rightmost digit, namely in the uppermost component belonging to the rightmost environment vector in the sequence of the environment vectors containing the addends. Respectively, the result and the carry are to be written as the second and the fifth, from the top, component content. In accordance with the notation introduced in the chapter 2.2., we call sign component the environment vector topmost component, carry component the next to the topmost component, first addend component or first operand component the middle component, second addend component or second operand component the fourth component, result component the bottommost component.

We present a training procedure for the GABNN to learn to control the robot to execute the addition of two binary numerals with operands composed by any number of digits. In other words, we want to train the net to learn to control an iterative behavior, with the number of iterations dependent on the number of digits present in the operands, i.e., on the number of input digits. To this end, we must either modify the general assembly wiring diagram so the net can be trained to compute an IF ... THEN ELSE, or we must present a suitable training procedure. Because the first solution has been presented elsewhere, see [6], we discuss the second alternative. Here, as in [21], we present the train-

ing procedure for a binary numeral system. To extend it to the decimal system case, we need simply more assemblies: their number depends on the numerical base we choose.

We give in the Table 6.2., see also the chapter 2.2.2. and the Table 2.1., the GABNN input-output correspondences for the addition. The addition starts on the operands rightmost digit and terminates at the left of the first operand leftmost digit, ie, on the environment vector at the left of the result leftmost digit and on the third from the top component. That is, the GABNN output primitive sequences, meaningful for the addition, corresponding to the environment vector configurations. The procedures to train the GABNN in the given correspondences are the following.

TABLE 6. 2.

(+, b, **0**, v, b) ⇒ DD(W{0})UUUU(E{+})L(W{+})DD
(+, b, v, **0**, b) ⇒ DD(W{0})UUUU(E{+})L(W{+})DD
(+, b, v, **1**, b) ⇒ DD(W{1})UUUU(E{+})L(W{+})DD
(+, b, **1**, v, b) ⇒ DD(W{1})UUUU(E{+})L(W{+})DD
(+, **1**, v, v', b) ⇒ DD(W{1})UUU(E{1})U(E{+})L(W{+})DD
(+, b, **1**, **1**, b) ⇒ DD(W{0})UUUU(E{+})L(W{+})D(W{1})D
(+, **1**, v, **1**, b) ⇒ DD(W{0})UUU(E{1})U(E{+})L(W{+})D(W{1})D
(+, **1**, **1**, v, b) ⇒ DD(W{0})UUU(E{1})U(E{+})L(W{+})D(W{1})D
(+, **1**, **1**, **1**, b) ⇒ DD(W{1})UUU(E{1})U(E{+})L(W{+})D(W{1})D
(+, b, b, b, b) ⇒ UU(E{+})DD

Table 6. 2.. *The GABNN input-output correspondences for the addition.* Instead of the GABNN input subvector sequences, ie, instead of one the hundred and twenty five binary digit sequences, at the left we give the ten environment vectors as seen by the robot sensors, actually for typographical reasons they have been rotate by nineteen degree toward the left, ie, the rightmost component is the uppermost. At the right, we give the corresponding GABNN output subvector sequences, ie, the commands the robot has to execute to perform the addition, see the chapter 6. 2. 3. 17.. The symbol b represents a blank space and the symbols v and v' represent, independently one from the other, either a blank space or the digit zero.

6.3.1. *Training the GABNN to respond to (+, b, 0, v, b) and to (+, b, v, 0, b).*

To the environment vectors (+, b, **0**, v, b) and (+, b, v, **0**, b), the GABNN has to output the sequence DD(W{0})UUUU(E{+})L(W{+})DD. That is, either there are two general assemblies corresponding to the two environment vectors content or there is just one general assembly having both as calling sequence. For simplicity sake we choose the latter solution, ie, let us call AD1 be the general assembly name corresponding to (+, b, **0**, v, b) and to (+, b, v, **0**, b). Accordingly, the GABNN has to be trained to output the said sequence every time its input subvector assumes the value AD1. The six steps training procedure is the following:

I) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $AD1_1$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly DD calling sequence, see the chapter 6.2.3.1.1.. As the primitive d has terminated to control the last instance of the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly W{0} calling sequence. If the input triplet [$AD1_1$, DD, W{0}] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by

the triplet symbols, then, the GΑBNN has been trained to output the sequence DD(W{0}) every time its first modality input is $AD1_1$.

II Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD1_2$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD1_1$ calling sequence. As the primitive d has terminated to control the last instance of the task d, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly υ'' calling sequence, see the chapter 6.2.3.1.2.. If the input triplet [$AD1_2$, $AD1_1$, υ''] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{0})υ'' every time its second modality input is $AD1_2$.

III Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $AD1_3$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $AD1_2$ calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly E{+} calling sequence. If the input triplet [$AD1_3$, $AD1_2$, E{+}] is the value assumed by the GΑBNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GΑBNN has been trained to output the sequence DD(W{0})υ''(E{+}) every time its first modality input is $AD1_3$.

IV) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD1_4$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD1_3$ calling sequence. As the primitive d has terminated to control the last instance of the task d, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly L calling sequence, see the chapter 6.2.3.1.1.. If the input triplet [$AD1_4$, $AD1_3$, L] is the value assumed by the GΑBNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GΑBNN has been trained to output the sequence DD(W{0})υ''(E{+})L every time its second modality input is $AD1_4$.

V) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD1_5$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD1_4$ calling sequence. As the primitive d has terminated to control the last instance of the task d, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly W{+} calling sequence. If the input triplet [$AD1_5$, $AD1_4$, W{+}] is the value assumed by the GΑBNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{0})υ''(E{+})L(W{+}) every time its second modality input is $AD1_5$.

VI) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly AD1 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $AD1_5$ calling sequence. As the primitive d has terminated to control the last instance of the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly DD calling sequence. If the input triplet [AD1, $AD1_5$, DD] is the value assumed by the GABNN input subvector a number of times suffi-

cient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{0})υ''(E{+})L(W{+})DD every time its first modality input is AD1.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

6.3.2. *Training the GABNN to respond to (+, b, v, 1, b) and to (+, b, 1, v, b).*

To the environment vectors (+, b, v, 1, b) and (+, b, 1, v, b), the GABNN has to output the sequence DD(W{1})UUUU(E{+})L(W{+})DD. That is, either there are two general assemblies corresponding to the two environment vectors or there is just one general assembly having both as calling sequence. For simplicity sake we choose the latter solution, ie, let us call AD2 be the general assembly name corresponding to (+, b, 1, v, b) and to (+, b, v, 1, b). Accordingly, the GABNN has to be trained to output the said sequence every time its input subvector assumes the value AD2. The six steps training procedure is the following:

I) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $AD2_1$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly DD calling sequence, see the chapter 6.2.3.1.1.. As the primitive d has terminated to control the last instance of the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly W{1} calling sequence. If the input triplet [$AD2_1$, DD, W{1}] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1}) every time its first modality input is $AD2_1$.

II) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD2_2$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD2_1$ calling sequence. As the primitive u has terminated to control the last instance of the task u, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly υ'' calling sequence, see the chapter 6.2.3.1.2.. If the input triplet [$AD2_2$, $AD2_1$, υ''] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1})υ'' every time its second modality input is $AD2_2$.

III) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $AD2_1$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $AD2_2$ calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly E{+} calling sequence. If the input triplet [$AD2_2$, $AD2_2$, E{+}] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet sym-

bols, then, the GABNN has been trained to output the sequence DD(W{1})υ''(E{+}) every time its first modality input is $AD2_3$.

IV) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD2_4$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD2_3$ calling sequence. As the primitive d has terminated to control the last instance of the task d, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly L calling sequence, see the chapter 6.2.3.1.1.. If the input triplet [$AD2_4$, $AD2_3$, L] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1})υ''(E{+})L every time its second modality input is $AD2_4$.

V) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD2_5$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD2_4$ calling sequence. As the primitive d has terminated to control the last instance of the task d, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly W{+} calling sequence. If the input triplet [$AD2_5$, $AD2_4$, (W{+})] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1})υ''(E{+})L(W{+}) every time its second modality input is $AD2_5$.

VI) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly AD2 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $AD2_5$ calling sequence. As the primitive d has terminated to control the last instance of the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly DD calling sequence. If the input triplet [AD2, $AD2_5$, DD] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1})υ''(E{+})L(W{+})DD every time its first modality input is AD2.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

6.3.3. *Training the GABNN to respond to (+, 1, v, v', b).*

To the environment vector (+, 1, v, v', b), the GABNN has to output the sequence DD(W{1})UUU(E{1})U(E{+})L(W{+})DD. For simplicity sake, let us call AD3 be the general assembly name corresponding to (+, 1, v, v', b). Accordingly, the GABNN has to be trained to output the said sequence every time its input subvector assumes the value AD3. The nine steps training procedure is the following:

I) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $AD3_1$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly DD calling sequence, see the chapter 6.2.3.1.. As the primitive d has terminated to control the last in-

stance of the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly W{1} calling sequence. If the input triplet [$AD3_1$, DD, W{1}] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1}) every time its first modality input is $AD3_1$.

II) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD3_2$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD3_1$ calling sequence. As the primitive u has terminated to control the last instance of the task u, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly υ' calling sequence, see the chapter 6. 2. 3. 1. 2.. If the input triplet [$AD3_2$, $AD3_1$, υ'] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1})υ' every time its second modality input is $AD3_2$.

III) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $AD3_3$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $AD3_2$ calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly U calling sequence. If the input triplet [$AD3_3$, $AD3_2$, U] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1})υ'U every time its first modality input is $AD3_3$.

IV) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD3_4$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD3_3$ calling sequence. As the primitive u has terminated to control the last instance of the task u, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly E{1} calling sequence. If the input triplet [$AD3_4$, $AD3_3$, E{1}] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1})υ'U(E{1}) every time its second modality input is $AD3_4$.

V) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $AD3_5$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $AD3_4$ calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly U calling sequence. If the input triplet [$AD3_5$, $AD3_4$, U] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1})υ'U(E{1})U every time its first modality input is $AD3_5$.

VI) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD3_6$ and, with a delay no greater than the maximum

delay D, the first modality input assumes the value of the general assembly $AD3_5$ calling sequence. As the primitive u has terminated to control the last instance of the task u, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly E{+} calling sequence. If the input triplet [$AD3_6$, $AD3_5$, E{+}] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1})υ'U(E{1})U(E{+}) every time its second modality input is $AD3_6$.

VII) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $AD3_7$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $AD3_6$ calling sequence. As the primitive d has terminated to control the last instance of the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly L calling sequence, see the chapter 6. 2. 3. 1. 1.. If the input triplet [$AD3_7$, $AD3_6$, L] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1})υ'U(E{+})L every time its first modality input is $AD3_7$.

VIII) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD3_8$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD3_7$ calling sequence. As the primitive l has terminated to control the last instance of the task l, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly W{+} calling sequence. If the input triplet [$AD3_8$, $AD3_7$, W{+}] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1})υ'U(E{+})L(W{+}) every time its second modality input is $AD3_8$.

IX) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly AD3 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $AD3_8$ calling sequence. As the primitive d has terminated to control the last instance of the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly DD calling sequence. If the input triplet [AD3, $AD3_8$, DD] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1})υ'U(E{+})L(W{+})DD every time its first modality input is AD3.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

6.3.4. *Training the GABNN to respond to (+, b, 1, 1, b).*

To the environment vector (+, b, 1, 1, b), the GABNN has to output the sequence DD(W{0})UUUU(E{+})L(W{+})D(W{1})D. For simplicity sake, let us call AD4 be the general assembly name corresponding to (+, b, 1, 1, b). Accordingly, the GABNN has

to be trained to output the said sequence every time its input subvector assumes the value AD4. The eight steps training procedure is the following:

I) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $AD4_1$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly DD calling sequence, see the chapter 6.2.3.1.1.. As the primitive d has terminated to control the last instance of the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly W{0} calling sequence. If the input triplet [$AD4_1$, DD, W{0}] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{0}) every time its first modality input is $AD4_1$.

II) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD4_2$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD4_1$ calling sequence. As the primitive d has terminated to control the last instance of the task d, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly υ'' calling sequence, see the chapter 6.2.3.1.1.2.. If the input triplet [$AD4_2$, $AD4_1$, υ''] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{0})υ'' every time its second modality input is $AD4_2$.

III) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $AD4_3$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $AD4_2$ calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly E{+} calling sequence. If the input triplet [$AD4_3$, $AD4_2$, E{+}] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{0})υ''(E{+}) every time its first modality input is $AD4_3$.

IV) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD4_4$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD4_3$ calling sequence. As the primitive d has terminated to control the last instance of the task d, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly L calling sequence, see the chapter 6.2.3.1.1.. If the input triplet [$AD4_4$, $AD4_3$, L] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{0})υ''(E{+})L every time its second modality input is $AD4_4$.

V) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD4_5$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD4_4$ calling sequence. As the primitive l has terminated to control the last instance of the task l, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly W{+} calling sequence. If the input triplet [$AD4_5$,

$AD4_4$, (W{+})] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{0})υ''(E{+})L(W{+}) every time its second modality input is $AD4_5$.

VI) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $AD4_6$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $AD4_5$ calling sequence. As the primitive d has terminated to control the last instance of the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly D calling sequence. If the input triplet [$AD4_6$, $AD4_5$, D] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{0})υ''(E{+})L(W{+})D every time its first modality input is $AD4_6$.

VII) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD4_7$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD4_6$ calling sequence. As the primitive d has terminated to control the last instance of the task d, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly W{1} calling sequence. If the input triplet [$AD4_7$, $AD4_6$, W{1}] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{0})υ''(E{+})L(W{+})D(W{1}) every time its second modality input is $AD4_7$.

VIII) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly AD4 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $AD4_7$ calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly D calling sequence. If the input triplet [AD4, $AD4_7$, D] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{0})υ''(E{+})L(W{+})D(W{1})D every time its first modality input is AD4.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

6.3.5. *Training the GABNN to respond to (+, 1, v, 1, b) and to (+, 1, 1, v, b).*

To the environment vectors (+, 1, 1, v, b) and (+, 1, v, 1, b), the GABNN has to output the sequence DD(W{0})UUU(E{1})U(E{+})L(W{+})D(W{1})D. That is, either there are two general assemblies corresponding to the two environment vectors or there is just one general assembly having both as calling sequence. For simplicity sake we choose the latter solution, ie, let us call AD5 be the general assembly name corresponding

to (+, 1, 1, v, b) and to (+, 1, v, 1, b). Accordingly, the GΑBNN has to be trained to output the said sequence every time its input subvector assumes the value AD5. The eleven steps training procedure is the following:

I) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $AD5_1$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly DD calling sequence, see the chapter 6.2.3.1.1.. As the primitive d has terminated to control the last instance of the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly W{0} calling sequence. If the input triplet [$AD5_1$, DD, W{0}] is the value assumed by the GΑBNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GΑBNN has been trained to output the sequence DD(W{0}) every time its first modality input is $AD5_1$.

II) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD5_2$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD5_1$ calling sequence. As the primitive d has terminated to control the last instance of the task d, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly υ' calling sequence, see the chapter 6.2.3.1.2.. If the input triplet [$AD5_2$, $AD5_1$, υ'] is the value assumed by the GΑBNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{0})υ' every time its second modality input is $AD5_2$.

III) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $AD5_3$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $AD5_2$ calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly U calling sequence. If the input triplet [$AD5_3$, $AD5_2$, U] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{0})υ'U every time its first modality input is $AD5_3$.

IV) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD5_4$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD5_3$ calling sequence. As the primitive u has terminated to control the last instance of the task u, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly E{1} calling sequence. If the input triplet [$AD5_4$, $AD5_3$, E{1}] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GΑBNN has been trained to output the sequence DD(W{0})υ'U(E{1}) every time its second modality input is $AD5_4$.

V) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $AD5_5$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $AD5_4$ calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes, again with a delay no greater than the maximum de-

lay D, the value of the general assembly U calling sequence. If the input triplet [$AD5_3$, $AD5_4$, U] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{0})υ'U(E{1})U every time its first modality input is $AD5_3$.

VI) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD5_6$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD5_5$ calling sequence. As the primitive u has terminated to control the last instance of the task u, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly E{+} calling sequence. If the input triplet [$AD5_6$, $AD5_5$, E{+}] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{0})υ'U(E{1})U(E{+}) every time its second modality input is $AD5_6$.

VII) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $AD5_7$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $AD5_6$ calling sequence. As the primitive d has terminated to control the last instance of the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly L calling sequence, see the chapter 6.2.3.1.1.. If the input triplet [$AD5_7$, $AD5_6$, L] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{0})υ'U(E{1})U(E{+})L every time its first modality input is $AD5_7$.

VIII) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD5_8$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD5_7$ calling sequence. As the primitive 1 has terminated to control the last instance of the task 1, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly W{+} calling sequence. If the input triplet [$AD5_8$, $AD5_7$, W{+}] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{0})υ'U(E{1})U(E{+})L(W{+}) every time its second modality input is $AD5_8$.

IX) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $AD5_9$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $AD5_8$ calling sequence. As the primitive d has terminated to control the last instance of the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly D calling sequence. If the input triplet [$AD5_9$, $AD5_8$, D] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{0})υ'U(E{1})U(E{+})L(W{+})D every time its first modality input is $AD5_9$.

X) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD5_{10}$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD5_9$ calling sequence. As the primitive d has terminated to control the last instance of the task d, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly W{1} calling sequence. If the input triplet [$AD5_{10}$, $AD5_9$, W{1}] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{0})υ'U(E{1})U(E{+})L(W{+})D(W{1}) every time its second modality input is $AD5_{10}$.

XI) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly AD5 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $AD5_{10}$ calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly D calling sequence. If the input triplet [AD5, $AD5_{10}$, D] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{0})υ'U(E{1})U(E{+})L(W{+})D(W{1})D every time its first modality input is AD5.

If in the given training procedure we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

6.3.6. *Training the GABNN to respond to (+, 1, 1, 1, b).*

To the environment vector (+, 1, v, v', b), the GABNN has to output the sequence DD(W{1})UUU(E{1})U(E{+})L(W{+})D(W{1})D. For simplicity sake, let us call AD6 be the general assembly name corresponding to (+, 1, 1, 1, b). Accordingly, the GABNN has to be trained to output the said sequence every time its input subvector assumes the value AD6. The eleven steps training procedure is the following:

I) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $AD6_1$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly DD calling sequence, see the chapter 6.2.3.1.1.. As the primitive d has terminated to control the last instance of the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly W{1} calling sequence. If the input triplet [$AD6_1$, DD, W{1}] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1}) every time its first modality input is $AD6_1$.

II) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD6_2$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD6_1$ calling sequence. As the primitive u has terminated to control the last instance of the task u, the

first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly υ' calling sequence, see the chapter 6.2.3.1.2.. If the input triplet [$AD6_2$, $AD6_1$, υ'] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1})υ' every time its second modality input is $AD6_2$.

III) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $AD6_3$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $AD6_2$ calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly U calling sequence. If the input triplet [$AD6_3$, $AD6_2$, U] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1})υ'U every time its first modality input is $AD6_3$.

IV) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD6_4$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD6_3$ calling sequence. As the primitive u has terminated to control the last instance of the task u, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly E{1} calling sequence. If the input triplet [$AD6_4$, $AD6_3$, E{1}] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1})υ'U(E{1}) every time its second modality input is $AD6_4$.

V) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $AD6_5$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $AD6_4$ calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly U calling sequence. If the input triplet [$AD6_5$, $AD6_4$, U] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1})υ'U(E{1})U every time its first modality input is $AD6_5$.

VI) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD6_6$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD6_5$ calling sequence. As the primitive u has terminated to control the last instance of the task u, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly E{+} calling sequence. If the input triplet [$AD6_6$, $AD6_5$, E{+}] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1})υ'U(E{1})U(E{+}) every time its second modality input is $AD6_6$.

VII) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $AD6_7$ and, with a delay no greater than the maximum

delay D, the second modality input assumes the value of the general assembly $AD6_6$ calling sequence. As the primitive d has terminated to control the last instance of the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly L calling sequence, see the chapter 6.2.3.1.. If the input triplet [$AD6_7$, $AD6_6$, L] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1})υ'U(E{+})L every time its first modality input is $AD6_7$.

VIII) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD6_8$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD6_7$ calling sequence. As the primitive 1 has terminated to control the last instance of the task 1, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly W{+} calling sequence. If the input triplet [$AD6_8$, $AD6_7$, W{+}] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1})υ'U(E{+})L(W{+}) every time its second modality input is $AD6_8$.

IX) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $AD6_9$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $AD6_8$ calling sequence. As the primitive d has terminated to control the last instance of the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly D calling sequence. If the input triplet [$AD6_9$, $AD6_8$, D] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1})υ'U(E{1})U(E{+})L(W{+})D every time its first modality input is $AD6_9$.

X) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD6_{10}$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD6_9$ calling sequence. As the primitive d has terminated to control the last instance of the task d, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly W{1} calling sequence. If the input triplet [$AD6_{10}$, $AD6_9$, W{1}] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DD(W{1})υ'U(E{1})U(E{+})L(W{+})D(W{1}) every time its second modality input is $AD6_{10}$.

XI) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly AD6 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $AD6_{10}$ calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly D calling sequence. If the input triplet [AD6, $AD6_{10}$, D] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the

GABNN has been trained to output the sequence DD(W{1})υ'U(E{1})U(E{+})L(W{+})D(W{1})D every time its first modality input is AD6.

6.3.7. *Training the GABNN to respond to (+, b, b, b, b).*

To the environment vector (+, b, b, b, b), the GABNN has to output the sequence UU(E{+})DD. For simplicity sake, let us call AD7 be the general assembly name corresponding to (+, b, b, b, b). Accordingly, the GABNN has to be trained to output the said sequence every time its input subvector assumes the value AD7. The three steps training procedure is the following:

I) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $AD7_1$ and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly UU calling sequence, see the chapter 6. 2. 3. 1. 1.. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly E{+} calling sequence. If the input triplet [$AD7_1$, UU, E{+}] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence UU(E{+}) every time its first modality input is $AD7_1$.

II) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $AD7_2$ and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $AD7_1$ calling sequence. As the primitive d has terminated to control the last instance of the task d, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly D calling sequence. If the input triplet [$AD7_2$, $AD7_1$, D] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence UU(E{+})D every time its second modality input is $AD7_2$.

III) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly AD7 and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $AD7_2$ calling sequence. As the primitive d has terminated to control the last instance of the task d, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly D calling sequence. If the input triplet [AD7, $AD7_2$, D] is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence UU(E{+})DD every time its first modality input is AD7.

If in the given training procedures we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

If all the input sequences given in the present chapter are the GABNN input subvector component values a sufficient number of times, ie, a number of times sufficient to

freeze the Hebbian connections, belonging to the second Hebbian set and obeying the second Hebbian rule, between the general assemblies, identified by the said sequences, and the previously trained general assemblies and the primitives, then the GABNN has learnt to execute the addition with addends containing any number of digits

6.4. Training the GABNN to control the conditional actions.

Up to now, we have discussed the procedures to train the GABNN to control given movement sequences. However, an autonomous system must have the capacity to execute movements dependent on the environment state. That is, the GABNN has to be trainable to control the robot movements in function of its input subvector component values. In other words, it have to be trainable to output a sequence of its primitive calling sequences, depending on the value assumed by one or by a subset of its input subvector components. Eg, it have to tell the robot to move when the present central component is blank and to stop, or to start a different action, otherwise, ie, when the present central component is written.

6.4.1. *Training the GABNN to shift an expression.*

In the chapter 6.2.3.15. we have presented a training procedure for the GABNN to control the copy of the presently observed central component content in the next component at the left, right, the relative general assembly calling sequence being CCL, CCR. Here we consider an expression, ie, a finite sequence of written and adjacent components. Adjacent to the sequence of written components there are two sequences of blank components, one at the right and the other at the left of the components sequence containing the said expression. That is, an expression written in one sequence of adjacent components is delimited by at least one blank component at its left and by at least one blank component at its right. Now we are going to discuss a procedure to copy the said expression toward the left, right, one digit, ie, one component at the time, every time the GABNN environment vector central component content assumes the value CNLE, CNRE. That is, we discuss a procedure starting from the leftmost, rightmost, written component and terminating the respective journeys on the rightmost, leftmost, written component holding the copied expression.

First case, every time the GABNN input subvector components comprised between the twenty sixth and the fiftieth, the last two components included, assume both the value CNLE, CNRE, and in the input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one component different from zero, the output subvector components assume as value the sequence: U(CCR)(EC)D(CCL)(EC)R, U(CCL)(EC)D(CCR)(EC)L. Otherwise, second case, every time the GABNN input subvector components comprised between the twenty sixth and the fiftieth, the last two components included, assume both the value CNLE, CNRE, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components assume as value the sequence: U(EC)DL, U(EC)DR.

The two training procedures are the followings:

A) First case

I) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly CNLE{1}, CNRE{1}, and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general

assembly U calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly CCR, CCL, calling sequence. If the input triplet [CNLE{1}, D, CCR], [CNRE{1}, D, CCL], is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence U(CCR), U(CCL), every time its first modality input is CNLE{1}, CNRE{1}.

II) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $CNLE_2$, $CNRE_2$, and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly CNLE{1}, CNRE{1} calling sequence. As the primitive r, l, has terminated to control the last instance of the task r, l, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly EC calling sequence. If the input triplet [$CNLE_2$, CNLE{1}, EC], [$CNRE_2$, CNRE{1}, EC], is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence U(CCR)(EC), U(CCL)(EC) every time its second modality input is $CNLE_2$, $CNRE_2$.

III) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $CNLE_3$, $CNRE_3$, and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $CNLE_2$, $CNRE_2$, calling sequence. As the primitive l has terminated to control the last instance of the task l, the second modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the primitive D calling sequence. If the input triplet [$CNLE_3$, $CNLE_2$, D], [$CNRE_2$, $CNRE_2$, D], is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence U(CCR)(EC)D, U(CCL)(EC)D, every time its first modality input is $CNLE_3$, $CNRE_3$.

IV) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $CNLE_4$, $CNRE_4$, and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $CNLE_3$, $CNRE_3$, calling sequence. As the primitive u has terminated to control the last instance of the task u, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly CCL,CCR, calling sequence. If the input triplet [$CNLE_4$, $CNLE_3$, CCL], [$CNRE_4$, $CNRE_3$, CCR], is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence U(CCR)(EC)D(CCL), U(CCL)(EC)D(CCR), every time its second modality input is $CNLE_4$, $CNRE_4$.

V) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $CNLE_5$, $CNRE_5$, and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $CNLE_4$, $CNRE_4$, calling sequence. As the primitive l has terminated to control the last instance of the task l, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the primitive EC calling sequence. If the input triplet [$CNLE_5$, $CNLE_4$, EC], [$CNRE_5$, $CNRE_4$, EC], is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the

IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence:

U(CCR)(EC)D(CCL)(EC),
U(CCL)(EC)D(CCR)(EC),

every time its second modality input is $CNLE_5$, $CNRE_5$.

VI) Both, first, the first modality input subvector components comprised between the twenty sixth and the fiftieth, the last two components included, assume both the value CNLE, CNRE, second, in the input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one component different from zero and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $CNLE_5$, $CNRE_5$, calling sequence. As the primitive l has terminated to control the last instance of the task l, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the primitive R, L, calling sequence. If the input triplet [CNLE, $CNLE_5$, R], [CNRE, $CNRE_4$, L], is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence:

U(CCR)(EC)D(CCL)(EC)R, U(CCL)(EC)D(CCR)(EC)L,

every time its first modality input is CNLE, CNRE, and in the input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one component different from zero.

B) Second case

I) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $CNLE_5$, $CNRE_6$, and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly U calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly EC calling sequence. If the input triplet [$CNLE_5$, U, EC], [$CNRE_5$, U, EC], is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence U(EC) every time its first modality input is $CNLE_5$, $CNRE_5$.

II) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $CNLE_5$, $CNRE_6$, and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $CNLE_5$, $CNRE_5$ calling sequence. As the primitive l has terminated to control the last instance of the task l, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly D calling sequence. If the input triplet [$CNLE_6$, $CNLE_5$, D], [$CNRE_6$, $CNRE_5$, D], is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence U(EC)D every time its second modality input is $CNLE_6$, $CNRE_6$.

III) Both, first, the first modality input subvector components comprised between the twenty sixth and the fiftieth, the last two components included, assume both the value CNLE, CNRE, second, the input subvector components comprised between the fifty first

and the seventy fifth, the last two included, have all the value zero and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly CNLE$_6$, CNRE$_6$, calling sequence. As the primitive u has terminated to control the last instance of the task u, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the primitive L, R, calling sequence. If the input triplet [CNLE, CNLE$_6$, L], [CNRE, CNRE$_6$, R], is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence U(EC)DL, U(EC)DR, every time its first modality input is CNLE, CNRE and in the input subvector components comprised between the fifty first and the seventy fifth, the last two included, all the components have the value zero.

If in the given training procedures we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

Later on we need the GABNN to recall the given output subvector sequences for different input subvector sequences. That is, the GABNN output subvector component contents have to assume either the sequences D(CCR)(EC)U(CCL)(EC)R, D(CCL)(EC)U(CCR)(EC)L, or the sequences D(EC)UL, D(EC)UR, every time both the first modality input subvector components comprised between the seventy sixth and the one hundredth, the last two components included, assume the value CNLE', CNRE', and in the input subvector components comprised between the fifty first and the seventy fifth, the last two included, either at least one of them is different from zero or have all the value zero. The relative training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

The GABNN output subvector component contents have to assume either the sequences:

DD(CCR)(EC)UU(CCL)(EC)R,
DD(CCL)(EC)UU(CCR)(EC)L

or the sequences

DD(EC)UUL,
DD(EC)UUR,

every time both the first modality input subvector components comprised between the hundred and first and hundred and twenty fifth, the last two components included, assume the value CNLBE, CNRBE, and in the input subvector components comprised between the fifty first and the seventy fifth, the last two included, either at least one of them is different from zero or have all the value zero. The relative training procedure is substantially the same as the one given above to train the GABNN to shift at the left, right, an expression.

It can be useful to have a general assembly controlling the leftward, rightward, copy of a sequence of written components when the said sequence is in the midst of a number of closely packed written sequences of components. In this case the only space left for writing the controlling general assembly calling sequence is the just erased, after having been copied, component. Accordingly, if the environment vector observed component contains the sequence CNLE'', CNRE'', and in its right, left, neighbor component there is at least one written square, the GABNN output subvector components assume as value the sequence (EC)R(CCL)(CNLE''), (EC)L(CCR)(CNRE''). Instead, if the environment vector observed component contains the sequence CNLE'', CNRE'', and its right, left, neighbor component is blank, the GABNN output subvector compo-

nents assume as value the sequence ((EC)R)LL, ((EC)L)RR. To understand the sequence of general assemblies called into play, let us discuss the GABNN inputs in more detail. Every time the environment vector central component contains the value CNLE' ', CNRE' ', first, there is the general assembly CNLE' ', CNRE' ', controlling the output sequence (EC)R, (EC)L, second, there are the two general assemblies WS and NS accepting the central component state as input on a modality different from the previous one, WS accepts the input corresponding to the central component written state and NS accepts as input the central component non written state, third, there two other general assemblies to be trained to compute: one the logical conjunction CNLE' '&WS, CNRE-' '&WS, and the other CNLE' '&NS, CNRE' '&NS. In any case the journey starts at the left, right, of the leftmost, rightmost, written component and ends on the rightmost, leftmost, copied component.

To train the GABNN to erase the presently observed expression once it has copied it one component below, up, with respect to the presently observed component, starting from the expression leftmost digit, first case, its output subvector components have to assume the sequence: U(CCR)(EC)D(CC1D)(EC)R, U(CCR)(EC)D(CC1U)(EC)R, every time, first, its first modality input subvector components comprised between the twenty sixth and the fiftieth, the last two components included, assume the value CN1DLE, CN1ULE, second, among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero. Otherwise, second case, every time the GABNN input subvector components comprised between the twenty sixth and the fiftieth, the last two components included, assume both the value CN1DLE, CN1ULE, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components have to assume as value the sequence: U(EC)DL, U(EC)DL.

Later on we need the GABNN to recall the given output subvector sequences for different input subvector sequences. That is, the GABNN output subvector component contents have to assume either the sequences U(CCR)(EC)D(CC1D)(EC)R, U(CCR)(EC)D(CC1U)(EC)R, or the sequences D(EC)UL, D(EC)UL, every time both the first modality input subvector components comprised between the seventy sixth and the one hundredth, the last two components included, assume the value CN1DLE', CN1ULE', and in the input subvector components comprised between the fifty first and the seventy fifth, the last two included, either at least one of them is different from zero or have all the value zero. The relative training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

To train the GABNN to erase the presently observed expression once it has copied it one component below, up, with respect to the presently observed component, starting from the expression rightmost symbol, first case, its output subvector components have to assume the sequence: U(CCL)(EC)D(CC1D)(EC)L, U(CCL)(EC)D(CC1U)(EC)L, every time, first, its first modality input subvector components comprised between the twenty sixth and the fiftieth, the last two components included, assume the value CN1DRE, CN1URE, second, among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero. Otherwise, second case, every time the GABNN input subvector components comprised between the twenty sixth and the fiftieth, the last two components included, assume both the value CN1DRE, CN1URE, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components have to assume as value the sequence: U(EC)DR, U(EC)DR.

Later on we need the GABNN to recall the given output subvector sequences for different input subvector sequences. That is, the GABNN output subvector component contents have to assume either the sequences D(CCL)(EC)U(CC1D)(EC)L, D(CCL)(EC)U(CC1U)(EC)L, or the sequences D(EC)UR, D(EC)UR, every time both the first modality input subvector components comprised between the seventy sixth

and the one hundredth, the last two components included, assume the value CN1DRE', CN1URE', and in the input subvector components comprised between the fifty first and the seventy fifth, the last two included, either at least one of them is different from zero or have all the value zero. The training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

To train the GABNN to erase the presently observed expression once it has copied it three components below, up, with respect to the presently observed component, starting from the expression leftmost symbol, first case, its output subvector components have to assume the sequence: U(CCR)(EC)D(CC3D)(EC)R, U(CCR)(EC)D(CC3U)(EC)R, every time, first, its first modality input subvector components comprised between the twenty sixth and the fiftieth, the last two components included, assume the value CN3DLE, CN3ULE, second, among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero. Otherwise, second case, every time the GABNN input subvector components comprised between the twenty sixth and the fiftieth, the last two components included, assume both the value CN3DLE, CN3ULE, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components have to assume as value the sequence: U(EC)DL, U(EC)DL. The training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

To train the GABNN to erase the presently observed expression once it has copied it three components below, up, with respect to the presently observed component, starting from the expression rightmost symbol, first case, its output subvector components have to assume the sequence: U(CCL)(EC)D(CC3D)(EC)L, U(CCL)(EC)D(CC3U)(EC)L, every time, first, its first modality input subvector components comprised between the twenty sixth and the fiftieth, the last two components included, assume the value CN3DRE, CN3URE, second, among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero. Otherwise, second case, every time the GABNN input subvector components comprised between the twenty sixth and the fiftieth, the last two components included, assume both the value CN3DRE, CN3URE, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components have to assume as value the sequence: U(EC)DR, U(EC)DR. The training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

To train the GABNN to copy, without erasing, the presently observed expression three components below, up, with respect to the presently observed component, starting from the expression leftmost symbol, first case, its output subvector components have to assume the sequence: U(CCR)(EC)D(CC3D)R, U(CCR)(EC)D(CC3U)R, every time, first, its first modality input subvector components comprised between the twenty sixth and the fiftieth, the last two components included, assume the value CN3DL, CN3UL, second, among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero. Otherwise, second case, every time the GABNN input subvector components comprised between the twenty sixth and the fiftieth, the last two components included, assume both the value CN3DL, CN3UL, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components have to assume as value the sequence: U(EC)DL, U(EC)DL. The training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

To train the GABNN to copy, without erasing, the presently observed expression three components below, up, with respect to the presently observed component, starting from the expression rightmost symbol, first case, its output subvector components have to assume the sequence: U(CCL)(EC)D(CC3D)L, U(CCL)(EC)D(CC3U)L, every time,

first, its first modality input subvector components comprised between the twenty sixth and the fiftieth, the last two components included, assume the value CN3DR, CN3UR, second, among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero. Otherwise, second case, every time the GABNN input subvector components comprised between the twenty sixth and the fiftieth, the last two components included, assume both the value CN3DR, CN3UR, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components have to assume as value the sequence: U(EC)DR, U(EC)DR. The training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

To train the GABNN to copy the presently observed expression five components below, up, with respect to the presently observed component, starting from the expression leftmost symbol, first case, its output subvector components have to assume the sequence: DD(CCR)(EC)DD(CC1VD)R, DD(CCR)(EC)DD(CC1VU)R, every time, first, its first modality input subvector last components assume the value CN1VDLB, CN1VULB, second, among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero. Otherwise, second case, every time the GABNN input subvector last twenty five components assume both the value CN1VDLB, CN1VULB, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components have to assume as value the sequence: DD(EC)DDL, DD(EC)DDL. The training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

To train the GABNN to copy the presently observed expression five components below, up, with respect to the presently observed component, starting from the expression rightmost symbol, first case, its output subvector components have to assume the sequence: DD(CCL)(EC)UU(CC1VD)L, DD(CCL)(EC)UU(CC1VU)L, every time, first, its first modality input subvector last twenty five components assume the value CN1VDRB, CN1VURB, second, among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero. Otherwise, second case, every time the GABNN input subvector last twenty five components assume both the value CN1VDRB, CN1VURB, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components have to assume as value the sequence: DD(EC)UUR, DD(EC)UUR. The training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

To train the GABNN to erase the presently observed expression once it has copied it five components below, up, with respect to the presently observed component, starting from the expression leftmost symbol, first case, its output subvector components have to assume the sequence: DD(CCR)(EC)DD(CC1VD)(EC)R, DD(CCR)(EC)DD(CC1VU)(EC)R, every time, first, its first modality input subvector last components assume the value CN1VDLE, CN1VULE, second, among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero. Otherwise, second case, every time the GABNN input subvector last twenty five components assume both the value CN1VDLE, CN1VULE, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components have to assume as value the sequence: DD(EC)DDL, DD(EC)DDL. The training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

To train the GABNN to erase the presently observed expression once it has copied it five components below, up, with respect to the presently observed component, starting from the expression rightmost symbol, first case, its output subvector components have to

assume the sequence: DD(CCL)(EC)UU(CC1VD)(EC)L, DD(CCL)(EC)UU(CC1VU)(EC)L, every time, first, its first modality input subvector last twenty five components assume the value CN1VDRE, CN1VURE, second, among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero. Otherwise, second case, every time the GABNN input subvector last twenty five components assume both the value CN1VDRE, CN1VURE, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components have to assume as value the sequence: DD(EC)UUR, DD(EC)UUR. The training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

To train the GABNN to copy the presently observed expression ten components below, up, with respect to the presently observed component, starting from the expression leftmost symbol, first case, its output subvector components have to assume the sequence: DD(CCR)(EC)UU(CC2VD)R, DD(CCR)(EC)UU(CC2VU)R, every time, first, its first modality input subvector last twenty five components assume the value CN2VDL, CN2VUL, second, among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero. Otherwise, second case, every time the GABNN input subvector last twenty five components assume both the value CN2VDL, CN2VUL, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components have to assume as value the sequence: DD(EC)UUL, DD(EC)UUL. The training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

To train the GABNN to copy the presently observed expression ten components below, up, with respect to the presently observed component, starting from the expression rightmost symbol, first case, its output subvector components have to assume the sequence: DD(CCR)(EC)UU(CC2VD)L, DD(CCR)(EC)UU(CC2VU)L, every time, first, its first modality input subvector last twenty five components assume the value CN2VDR, CN2VUR, second, among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero. Otherwise, second case, every time the GABNN input subvector last twenty five components assume both the value CN2VDR, CN2VUR, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components have to assume as value the sequence: DD(EC)UUR, DD(EC)UUR. The training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

To train the GABNN to erase the presently observed expression once it has copied it ten components below, up, with respect to the presently observed component, starting from the expression leftmost symbol, first case, its output subvector components have to assume the sequence: DD(CCR)(EC)UU(CC2VD)(EC)R, DD(CCR)(EC)UU(CC2VU)(EC)R, every time, first, its first modality input subvector last twenty five components assume the value CN2VDLE, CN2VULE, second, among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero. Otherwise, second case, every time the GABNN input subvector last twenty five components assume both the value CN2VDLE, CN2VULE, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components have to assume as value the sequence: DD(EC)UUL, DD(EC)UUL. The training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

To train the GABNN to erase the presently observed expression once it has copied it ten components below, up, with respect to the presently observed component, starting from the expression rightmost symbol, first case, its output subvector components have to as-

sume the sequence: DD(CCL)(EC)UU(CC2VD)(EC)L, DD(CCL)(EC)UU(CC2VU)(EC)L, every time, first, its first modality input subvector last twenty five components assume the value CN2VDRE, CN2VURE, second, among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero. Otherwise, second case, every time the GABNN input subvector last twenty five components assume both the value CN2VDRE, CN2VURE, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components have to assume as value the sequence: DD(EC)UUR, DD(EC)UUR. The training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

To train the GABNN to erase the presently observed expression once it has copied it twenty five components below, up, with respect to the presently observed component, starting from the expression leftmost symbol, first case, its output subvector components have to assume the sequence: DD(CCR)(EC)UU(CC)(EC)R, DD(CCR)(EC)UU(CCU)(EC)R, every time, first, its first modality input subvector last twenty five components assume the value CN5VDLE, CN5VULE, second, among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero. Otherwise, second case, every time the GABNN input subvector last twenty five components assume both the value CN5VDLE, CN5VULE, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components have to assume as value the sequence: DD(EC)UUL, DD(EC)UUL. The training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

To train the GABNN to erase the presently observed expression once it has copied it twenty five components below, up, with respect to the presently observed component, starting from the expression rightmost symbol, first case, its output subvector components have to assume the sequence: DD(CCL)(EC)UU(CC)(EC)L, DD(CCL)(EC)UU(CCU)(EC)L, every time, first, its first modality input subvector last twenty five components assume the value CN5VDRE, CN5VURE, second, among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero. Otherwise, second case, every time the GABNN input subvector last twenty five components assume both the value CN5VDRE, CN5VURE, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components have to assume as value the sequence: DD(EC)UUR, DD(EC)UUR. The training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

To train the GABNN to copy, without erasing, the presently observed expression twenty five components below, up, with respect to the presently observed component, starting from the expression leftmost symbol, first case, its output subvector components have to assume the sequence: DD(CCR)(EC)UU(CC)R, DD(CCR)(EC)UU(CCU)R, every time, first, its first modality input subvector last twenty five components assume the value CN5VDL, CN5VUL, second, among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero. Otherwise, second case, every time the GABNN input subvector last twenty five components assume both the value CN5VDL, CN5VUL, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components have to assume as value the sequence: DD(EC)UUL, DD(EC)UUL. The training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

To train the GABNN to copy the presently observed expression twenty five components below, up, with respect to the presently observed component, starting from the ex-

pression rightmost symbol, first case, its output subvector components have to assume the sequence: DD(CCL)(EC)UU(CC)L, DD(CCL)(EC)UU(CCU)L, every time, first, its first modality input subvector last twenty five components assume the value CN5VDR, CN5VUR, second, among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero. Otherwise, second case, every time the GABNN input subvector last twenty five components assume both the value CN5VDR, CN5VUR, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components have to assume as value the sequence: DD(EC)UUR, DD(EC)UUR. The training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

To train the GABNN to erase the presently observed expression once it has copied it twenty four components below, up, with respect to the presently observed component, starting from the expression leftmost symbol, first case, its output subvector components have to assume the sequence: U(CCR)(EC)D(CC-1)(EC)R, U(CCR)(EC)D(CCU-1)(EC)R, every time, first, its first modality input subvector components comprised between the twenty sixth and the fiftieth, the last two components included, assume the value CN4V4CDLE, CN4V4CULE, second, among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero. Otherwise, second case, every time the GABNN input subvector components comprised between the twenty sixth and the fiftieth, the last two components included, assume both the value CN4V4CDLE, CN4V4CULE, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components have to assume as value the sequence: U(EC)DL, U(EC)DL. The training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

To train the GABNN to erase the presently observed expression once it has copied it twenty four components below, up, with respect to the presently observed component, starting from the expression rightmost symbol, first case, its output subvector components have to assume the sequence: U(CCL)(EC)D(CC-1)(EC)L, U(CCL)(EC)D(CCU-1)(EC)L, every time, first, its first modality input subvector components comprised between the twenty sixth and the fiftieth, the last two components included, assume the value CN4V4CDRE, CN4V4CURE, second, among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero. Otherwise, second case, every time the GABNN input subvector components comprised between the twenty sixth and the fiftieth, the last two components included, assume both the value CN4V4CDRE, CN4V4CURE, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components have to assume as value the sequence: U(EC)DR, U(EC)DR. The training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

To train the GABNN to erase the presently observed expression once it has copied it twenty three components below, up, with respect to the presently observed component, starting from the expression leftmost symbol, first case, its output subvector components have to assume the sequence: DD(CCR)(EC)UU(CCD-2)(EC)R, DD(CCR)(EC)UU(CCU-2)(EC)R, every time, first, its first modality input subvector last twenty five components assume the value CN4V3CDLE, CN4V3CULE, second, among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero. Otherwise, second case, every time the GABNN input subvector last twenty five components assume both the value CN4V3CDLE, CN4V3CULE, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components have to assume as value the sequence: DD(EC)UUL,

DD(EC)UUL. The training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

To train the GABNN to erase the presently observed expression once it has copied it twenty three components below, up, with respect to the presently observed component, starting from the expression rightmost symbol, first case, its output subvector components have to assume the sequence: DD(CCL)(EC)UU(CCD-2)(EC)L, DD(CCL)(EC)UU(CCU-2)(EC)L, every time, first, its first modality input subvector last twenty five components assume the value CN4V3CDRE, CN4V3CURE, second, among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero. Otherwise, second case, every time the GABNN input subvector last twenty five components assume both the value CN4V3CDRE, CN4V3CURE, and the input subvector components comprised between the fifty first and the seventy fifth, the last two included, have all the value zero, the output subvector components have to assume as value the sequence: DD(EC)UUR, DD(EC)UUR. The training procedure is substantially the same as the one given above to train the GABNN to shift left, or right, an expression.

6.4.2. *Training the GABNN to modify an expression.*

6.4.2.1. *To erase a sequence of symbols.*

A expression can be erased starting from its leftmost, rightmost, symbol and writing the relative general assembly calling sequence on the environment vector bottommost, topmost, component. Respectively, the calling sequences are: ELB, ERB, ELT, ERT. Let us call W{ELB}, W{ERB}, W{ELB}, W{ERB}, the calling sequences of the general assemblies controlling the robot to write, respectively, ELB, ERB, ELT, ERT. The GABNN output subvector component content are:

(EC)DD(EC)R(W{ELB})UU,
(EC)DD(EC)L(W{ERB})UU,
(EC)UU(EC)R(W{ELB})DD,
(EC)UU(EC)L(W{ERB})DD,

every time the environment vector first, first, last, last, component content is ELB, ERB, ELT, ERT, and in the first modality input subvector there is at least one non zero component between the fifty first and the seventy fifth, the last two components included. Instead, the GABNN output subvector component content are the sequences: DD(EC)UU, DD(EC)UU, UU(EC)DD, UU(EC)DD, every time the environment vector first, first, last, last, component content is ELB, ERB, ELT, ERT, and in the first modality input subvector all the components between the fifty first and the seventy fifth, the last two components included, are zero.

If in the given training procedures we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

As the relative training procedures are substantially the same as the one given in the chapter 6.4.1. to train the GABNN to shift left, or right, an expression, we leave it to the interested reader to write them.

6.4.2.2. *To write a sequence of equal symbols.*

A string of zeros, ones, can be written from its leftmost, rightmost, symbol until a marked component is found, starting from the presently observed component and writing

the relative general assembly calling sequence on the environment vector bottommost, topmost, component. Respectively, the calling sequences are: WZLB, WZRB, WZLT, WZRT, WOLB, WORB, WOLT, WORT. Let us call W{WZLB}, W{WZRB}, W{WZLT}, W{WZRT}, W{WOLB}, W{WORB}, W{OLT}, W{ORT}, the calling sequences of the general assemblies controlling the robot to write, respectively, WZLB, WZRB, WZLT, WZRT, WOLB, WORB, WOLT, WORT. The GABNN output subvector component content are the sequences:

```
(W{0})DD(EC)R(W{WZLB})UU,
(W{0})DD(EC)L(W{WZRB})UU,
(W{0})UU(EC)R(W{WZLT})DD,
(W{0})UU(EC)L(W{WZRT})DD,
(W{1})DD(EC)R(W{WOLB})UU,
(W{1})DD(EC)L(W{W{1}RB})UU,
(W{1})UU(EC)R(W{WOLT})DD,
(W{1})UU(EC)L(W{W{1}RT})DD,
```

every time the environment vector first, first, last, last, first, first, last, last, component content is WZLB, WZRB, WZLT, WZRT, WOLB, WORB, WOLT, WORT, and all the components between the fifty first and the seventy fifth, the last two components included, in the first modality input subvector have the value zero. Instead, the GABNN output subvector component content are:

```
(EC)(W{0})DD(EC)UU,
(EC)(W{0})DD(EC)UU,
(EC)(W{0})UU(EC)DD,
(EC)(W{0})UU(EC)DD,
(EC)(W{1})DD(EC)UU,
(EC)(W{1})DD(EC)UU,
(EC)(W{1})UU(EC)DD,
(EC)(W{1})UU(EC)DD,
```

every time the environment vector first, first, last, last, first, first, last, last, component content is WZLB, WZRB, WZLT, WZRT, WOLB, WORB, WOLT, WORT, and in the first modality input subvector there is at least one non zero component between the fifty first and the seventy fifth, the last two components included.

If in the given training procedures we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

As the relative training procedures are substantially the same as the one given in the chapter 6.4.1. to train the GABNN to shift left, or right, an expression, we leave it to the interested reader to write them.

6.4.2.3. *To erase a sequence of symbols and then to write one symbol.*

A expression can be erased starting from its leftmost, rightmost, symbol and writing, in the component initially holding its rightmost, leftmost, symbol, a one if initially all its symbols were ones, a zero otherwise. Initially, the general assembly calling sequence EWL{1v0}, EWR{1v0}, is written in the fourth from the top component of the environment vector.

If the environment vector third component holds a one and the fourth from the top component holds EWL{1v0}, EWR{1v0}, then the GABNN has to be trained to gener-

ate the sequence (EC)D(EC)R(W{EWL{1v0}})U, (EC)D(EC)L(W{EWR{1v0}})U as output subvector component content.

If the environment vector third component holds a zero and the fourth from the top component holds EWL{1v0}, EWR{1v0}, then the GABNN has to be trained to generate the output subvector component content sequence (EC)D(EC)R(W{EWL{0}})U, (EC)D(EC)L(W{EWR{0}})U.

If the environment vector third component holds a blank and the fourth from the top component holds EWL{1v0}, EWR{1v0}, then the GABNN has to be trained to generate the output subvector component content sequence L(W{1})D(EC)U, R(W{1})D(EC)U.

If the environment vector third component holds either a one or a zero and the fourth from the top component holds EWL{0}, EWR{0}, then the GABNN has to be trained to generate the sequence (EC)D(EC)R(W{EWL{0}})U, (EC)D(EC)L(W{EWR{0}})U as output subvector component content.

If the environment vector third component holds a blank and the fourth from the top component holds EWL{0}, EWR{0}, then the GABNN has to be trained to generate the output subvector component content sequence L(W{0})D(EC)U, R(W{0})D(EC)U.

If in the given training procedures we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

As the relative training procedures are substantially the same as the one given in the chapter 6.4.1. to train the GABNN to shift left, or right, an expression, we leave it to the interested reader to write them.

6.4.2.4. *An illustrative example: training the GABNN to erase the addition first and second operand.*

In the chapter 6.3. we have proposed a procedure to train the GABNN to perform the addition. The implemented algorithm leaves the system on the component at the left of the first operand leftmost symbol. Now, we discuss how to modify the addition algorithm exit relation:

$$(+, b, b, b, b) \Rightarrow \texttt{UU(E\{+\})DD},$$

once the plus sign has been erased, first to erase the two operands, second, to copy the result four components up in the sign component, third, to stop on the result rightmost symbol.

To this end, let EO(C{RSLT4U}) be the calling sequence of the general assembly controlling the execution of the procedure the GABNN has to be trained on. In other words, every time the first modality input subvector components comprised between the twenty sixth and the fifties have the value EOCR4U and the environment vector last component contains a symbol, the GABNN output subvector components assume as value the sequence (CCR)(EC)D(EC)D(EC)(W{CC4UR3U})D.

Instead, every time the first modality input subvector components comprised between the twenty sixth and the fifties have the value EOCR4U and the environment vector last component is blank, the GABNN output subvector components assume as value the sequence (EC)LU.

Eventually, the addition algorithm exit relation becomes

$$(+, b, b, b, b) \Rightarrow \texttt{UU(E\{+\})DR(W\{EOCR4U\})D}.$$

If in the given training procedures we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

As the relative training procedures are substantially the same as the one given in the chapter 6.3.7. to train the GABNN to respond to the very same environment vector, we leave it to the interested reader to write them.

6.4.3. *The conditional movements.*

We discuss a GABNN training procedure to control the robot conditional movements toward the left, right. Let us call G a general assembly calling sequence such that the GABNN has been previously trained to respond with one well defined output subvector components sequence every time it is the input subvector value. To fix the ideas, the GABNN output subvector components have to assume the value L, R, at least, every time the environment vector first component assumes the value L{G}, R{G}, and the central component has at least one written square. And the output subvector component assumes the value G every time the environment vector first component assumes the value L{G}, R{G}, and all the central component squares are blank. In other words we are going to present a procedure to find the leftmost, rightmost, symbol in an expression.

To fix the ideas, let the representation of G in the first modality input subvector be the letter G itself and the representation of L{G}, R{G}, be the superposition of the very same two capital letters. That is, with W{L{G}}, W{R{G}}, or E{L{G}}, E{R{G}}, we represents the general assemblies calling sequences controlling the robot to write or to erase the calling sequences enclosed within the curly brackets. To specify the particular value $\wp$ for G we shall write L{G ≡ $\wp$}, R{G ≡ $\wp$}, eg L{G ≡ R} means that the value R has to take the place of G in all the following sequences. When no ambiguity can arise, we write L{$\wp$}, R{$\wp$}, simply.

Then, first, every time the GABNN first modality input subvector first twenty five components assume the value L{G}, R{G}, and there is at least one non zero component between the fifty first and the seventy fifth, the last two components included, the GABNN output subvector components assume the value UU(E{L{G}})L(W{L{G}}DD, UU(E{R{G}})R(W{R{G}})DD.

Second, in the GABNN there is a different general assembly identified both by L{G}, R{G}, in the environment vector first component and by all the components between the fifty first and the seventy fifth, both included, of the first modality input subvector having the value zero. In correspondence of such an input, the GABNN output subvector components assume the value UU(E{L{G}})DDG, UU(E{R{G}})DDG.

In conclusion, to train the GABNN to control the robot movements depending on the environment state it has to learn what to do in response to both inputs. Accordingly, it has to be trained, first, in a four steps procedure and, second, in a two steps one. The two training procedures are the followings:

A) First Procedure

I) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $L\{G\}_1$, $R\{G\}_1$, and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly UU calling sequence, see the chapter 6.2.3.1.1.. As the primitive u has terminated to control the last instance of the task u, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly E{L{G}}, ER{G}, calling sequence. If the input triplet [$L\{G\}_1$, UU, E{L{G}}], [$R\{G\}_1$, UU, E{R{G}}], is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obey-

ing the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence UU(E{L{G}}), UU(E{R{G}}), every time its second modality input is $L\{G\}_1$, $R\{G\}_1$.

II) Both the first modality input assumes the value of the calling sequence corresponding to the general assembly $L\{G\}_2$, $R\{G\}_2$, and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly L, R. calling sequence. As the primitive l, r, has terminated to control the last instance of the task l, r, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly W{L{G}}, W{R{G}}, calling sequence. If the input triplet [$L\{G\}_2$, L, W{L{G}}], [$R\{G\}_2$, R, W{R{G}}], is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence L(W{L{G}}), R(W{R{G}}), every time its first modality input is $L\{G\}_2$, $R\{G\}_2$.

III) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $L\{G\}_3$, $R\{G\}_3$, and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly $L\{G\}_2$, $R\{G\}_2$, calling sequence. As the primitive has terminated to control the execution of the W{L{G}}, W{R{G}}, the first modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly DD calling sequence. If the input triplet [$L\{G\}_2$, $L\{G\}_2$, DD], [$R\{G\}_3$, $R\{G\}_2$, DD], is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence L(W{L{G}})DD, R(W{R{G}})DD every time its second modality input is $L\{G\}_3$, $R\{G\}_3$.

IV) Both, first, the first component of the first modality input assumes the value of the calling sequence corresponding to the general assembly L{G}, R{G}, second, there is at least one non zero component between the fifty first and the seventy fifth, both included, component of the first modality input subvector and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $L\{G\}_1$, $R\{G\}_1$, calling sequence. As the primitive has terminated to erase the component containing the L{G}, R{G}, calling sequence, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly $L\{G\}_3$, $R\{G\}_3$, calling sequence. If the input triplet [L{G}, $L\{G\}_1$, $L\{G\}_3$], [R{G}, $R\{G\}_1$, $R\{G\}_3$], is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence UU(E{L{G}})L(W{L{G}})DD, UU(E{R{G}})R(W{R{G}})DD, every time the first component of its first modality input is L{G}, R{G}, and there is at least one non zero component between the fifty first and the seventy fifth component, the last two components included, of the first modality input subvector.

B) Second Procedure

I) Both the second modality input assumes the value of the calling sequence corresponding to the general assembly $L\{G\}_4$, $R\{G\}_4$, and, with a delay no greater than the maximum delay D, the first modality input assumes the value of the general assembly DD calling sequence. As the primitive d has terminated to control the last instance of the task d, the first modality input assumes again, and again with a delay no greater than the maximum delay D, the value of the general assembly G calling sequence. If the input triplet [$L\{G\}_4$, DD, G], [$R\{G\}_4$, DD, G], is the value assumed by the GABNN input

subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence DDG every time its second modality input is $L\{G\}_4$, $R\{G\}_4$.

II) Both, first, the first component of the first modality input assumes the value of the calling sequence corresponding to the general assembly L{G}, R{G}, second, all the components between the fifty first and the seventy fifth, both included, of the first modality input subvector have the value zero and, with a delay no greater than the maximum delay D, the second modality input assumes the value of the general assembly $L\{G\}_1$, $R\{G\}_1$, calling sequence. As the primitive has terminated to erase the component containing the L{G}, R{G}, calling sequence, the second modality input assumes, again with a delay no greater than the maximum delay D, the value of the general assembly $L\{G\}_4$, $R\{G\}_4$, calling sequence. If the input triplet [L{G}, $L\{G\}_1$, $L\{G\}_4$], [R, $R\{G\}_1$, $R\{G\}_4$], is the value assumed by the GABNN input subvector a number of times sufficient to freeze the Hebbian connections belonging to the second Hebbian set and obeying the second Hebbian rule between the IBNSs identified by the triplet symbols, then, the GABNN has been trained to output the sequence UU(E{L{G}})DDG, UU(E{R{G}})DDG, every time the first component of its first modality input is L{G}, R{G}, and in the first modality input subvector all the components between the fifty first and the seventy fifth, the last two components included, have the value zero.

If in the given training procedures we interchange the words "first modality input" with "second modality input", and vice versa, the GABNN can be trained to output the given sequences with respect to both input modalities.

To control the leftward, rightward, movements while the presently observed is a written component, the GABNN can be trained to assume in the output subvector the sequence DD(E{(LD){G}})L(W{(LD){G}})UU, DD(E{(RD){G}})R(W{(RD){G}})UU, every time the environment vector last component content is (LD){G}, (RD){G}, and there is at least one non zero component between the fifty first and the seventy fifth component, the last two components included, of the first modality input subvector. And to assume in the output subvector the sequence DD(E{(LD){G}})UUG, DD(E{(RD){G}})UUG, every time the environment vector last component content is (LD){G}, (RD){G}, and all the components between the fifty first and the seventy fifth component, the last two components included, of the first modality input subvector have the value zero.

Let us train the GABNN to control the leftward, rightward, movements until the presently observed is a blank component. Again, let us call G a general assembly calling sequence such that the GABNN has been previously trained to respond with one well defined output subvector components sequence every time it is the input subvector value. To fix the ideas, let the representation of G in the first modality input subvector be the letter G itself and the representation of L'{G}, R'{G}, be the superposition of the very same two capital letters. That is, with W{L'{G}}, W{R'{G}}, or E{L'{G}}, E{R'{G}}, we represents the general assemblies calling sequences controlling the robot to write or to erase the calling sequences enclosed within the curly brackets.

Accordingly, every time in the first modality input subvector both the components comprised between the seventy sixth and the one hundredth, the last two components included, have the value L'{G}, R'{G}, and all the components comprised between the fifty first and the seventy fifth, the last two components included, have the value zero, the GABNN output subvector component contents assume the value D(E{L'{G}})L(W{L'{G}})U, D(E{R'{G}})R(W{R'{G}})U. And every time in the first modality input subvector both the components comprised between the seventy sixth and the one hundredth, the last two components included, have the value L'{G}, R'{G}, and at least one component comprised between the fifty first and the seventy fifth, the last two components included, have a value different from zero, the GABNN

output subvector component contents assume the value D(E{L'{G}})UG, D(E{R'{G}})UG.

Instead, every time in the first modality input subvector both the last twenty five components have the value LB'{G}, RB'{G}, and all the components comprised between the fifty first and the seventy fifth, the last two components included, have the value zero, the GABNN output subvector component contents assume the value DD(E{LB'{G}})L(W{LB'{G}})UU, DD(E{RB'{G}})R(W{RB'{G}})UU. And every time in the first modality input subvector both the last twenty five components have the value LB'{G}, RB'{G}, and at least one component comprised between the fifty first and the seventy fifth, the last two components included, have a value different from zero, the GABNN output subvector component contents assume the value DD(E{LB'{G}})UUG, DD(E{RB'{G}})UUG.

To control the leftward, rightward, movements while the presently observed is a blank component, the GABNN can be trained to assume in the output subvector the sequence DD(E{(LNM){G}})L(W{(LNM){G}})UU, DD(E{(RNM){G}})R(W{(RNM){G}})UU, every time the environment vector last component content is (LNM){G}, (RNM){G}, and the central component is blank. And to assume in the output subvector the sequence DD(E{(LNM){G}})UUG, DD(E{(RNM){G}})UUG, every time the environment vector last component content is (LNM){G}, (RNM){G}, and the central component is written, eg marked.

To control the leftward, rightward, movements while the presently observed is a component written except in its lower left square, the GABNN can be trained to assume in the output subvector the sequence DD(E{(LM){G}})L(W{(LM){G}})UU, DD(E{(RM){G}})R(W{(RM){G}})UU, every time the environment vector last component content is (LM){G}, (RM){G}, and the central component is written and not marked, ie, is written except in its lower left square. And to assume in the output subvector the sequence DD(E{(LM){G}})UUG, DD(E{(RM){G}})UUG, every time the environment vector last component content is (LM){G}, (RM){G}, and the central component is marked.

To control the leftward, rightward, movements until the presently observed component content is a parenthesis, either open or closed, the GABNN can be trained to assume in the output subvector the sequence DD(E{(LP){G}})L(W{(LP){G}})UU, DD(E{(RP){G}})R(W{(RP){G}})UU, every time the environment vector last component content is (LP){G}, (RP){G}, and the central component content is not a parenthesis, either open or closed. And to assume in the output subvector the sequence DD(E{(LP){G}})UUG, DD(E{(RP){G}})UUG, every time the environment vector last component content is (LP){G}, (RP){G}, and the central component content is a parenthesis, either open or closed.

To control the upward, downward, movements along written components staked one on top of the other and having at the right-hand side blank components the GABNN can be trained to assume as the output subvector component contents the sequence LURD(CC1U)(EC)U, LDRD(CC1D)(EC)D, every time the environment vector central component assume the value (MUL), (MDL), and the environment vector central and second from the top components are written. Otherwise, ie, if the environment vector observed component assume the value (MUL), (MDL), and the next at the left environment vector central and second from the top components are written, the output subvector component contents assume as value the sequence LURD(EC)L, LDRD(EC)L.

To control the upward, downward, movements along written components staked one on top of the other and having blank components at the left-hand side the GABNN can be trained to assume as the output subvector component contents the sequence RULD(CC1U)(EC)U, RDLD(CC1D)(EC)D, every time the environment vector central component assume the value (MUR), (MDR), and the environment vector central and second from the top components are written. Otherwise, ie, if the environment vector observed component assume the value (MUR), (MDR), and the next at the left of the en-

vironment vector central component and the second from the top component are both written, the output subvector component contents assume as value the sequence RULD(EC)R, RDLD(EC)R.

6.4.4. *Chaining the procedures.*

To combine the GABNN output subvector content sequences so to move, first, an expression down, up, and then left, right, or vice versa, we must *chain the procedures* represented by the subvector content sequences. That is, we must modify an output sequence exit conditions so to start automatically the new output sequence once the present one has terminated its journey.

6.4.4.1. *An illustrative example.*

To fix the ideas, starting from the extreme left, let us see how to chain the output sequences to control the copying of an expression א, three components below the present one, followed by the output sequence to control the shift of א toward the left until א lays below a sequence of blank components followed by the output sequence to move א one component up. In other words, we want to train the GABNN to control the sequences of primitives obtained by chaining the output sequences corresponding to the first modality input sequences CN3DL, CNLE', CN1URE', where the second one, namely CNLE', have to be repeated again and again until the given condition on א is satisfied.

To understand how to chain more output sequences we must modify their exit conditions so that each of them leaves the environment vector present component contents equal to the next output sequence entry condition. To this end, let us recall the said output sequences entry and exit conditions. The output sequence activated by the calling sequence CN3DL starts its journey on the leftmost symbol of the expression to copy and terminates its journey on the rightmost symbol of the expression א it has just copied. The output sequence activated by the calling sequence CNLE' starts its journey on the leftmost symbol of א and terminates its journey on the component containing the rightmost symbol of the copy of the expression א. The output sequence activated by the calling sequence CN1URE' starts its journey on the rightmost symbol of א and terminates its journey on the leftmost symbol of the copy of the expression א, the original having been erased.

That is, the output sequences corresponding to both the input sequences CN3DL and CNLE' end their journeys on the rightmost symbol of א and CNLE' starts its journey on the leftmost symbol of the very same expression. Accordingly, some intermediate output sequences to control the conditional movements are needed to compensate for the said discrepancies. In turn, once the output sequence corresponding to the input sequence CN3DL has terminated its journey, there could be two different outcomes. If the component on top of the one presently observed is written on, it has to be chained to the output sequence corresponding to the first modality input CNLE'. Otherwise, it has to be chained to the output sequence corresponding to the first modality input CN1URE'.

The same has to happen once the output sequence corresponding to the first modality input CNLE' has terminated its journey. That is, the output sequence corresponding to the input sequence CNLE' has to be chained again to the output sequence corresponding to the input sequence CNLE', if the component on top of the one presently observed is written on. Otherwise, it has to be chained to the output sequence corresponding to the first modality input CN1URE'.

Accordingly, the output sequence corresponding to the first modality input CN1URE' has to be chained to the output sequence implementing the exit conditions of the algorithm we are interested in, namely the multiplication algorithm, see the chapter 6. 5.. That is, as CN1URE' terminates its journey on the leftmost symbol of the expression it has just copied, to complete the multiplication algorithm, it has, first, to go three components up, second, to write the calling sequence of the conditional movement going rightward until it finds a non blank component, third, to erase the just found non blank component, fourth, to go backward one component, namely on the next component at the left, fifth, to write the times sign, sixth, to go down two components. The output subvector sequence to control the last four steps, namely the third, the fourth, the fifth and the sixth, is (EC)L(W{x})DD. If we take G ≡ (EC)L(W{x})DD, see the chapter 6.4.3., the output sequence corresponding to the first modality input CN1URE' has to be modified so to control a sequence of primitives terminating with UUU(W{R'{(EC)L(W{x})DD}})U, where W{R'{(EC)L(W{x})DD}} is the calling sequence of the general assembly trained to control the robot to write the value R'{(EC)L(W{x})DD}.

That is, let us call CN1URE'{UUU(W{R'{(EC)L(W{x})DD}})U} the calling sequence of the general assembly trained to control the CN1URE' modified version. Then, every time the GABNN the first modality input subvector components comprised between the seventy sixth and the one hundredth, the last two components included, assume the value CN1URE'{UUU(W{R'{(EC)L(W{x})DD}})U} and in the first modality input subvector components comprised between the fifty first and the seventy fifth, the last two included, either at least one of them is different from zero or all have to be zero, the output subvector components either assume the sequence of values D(CCL)(EC)U(CC1U)(EC)L or the sequence D(EC)URUUU(W{R'{(EC)L(W{x})DD}})U. In turn, every time in the first modality input subvector both the components comprised between the fifty first and the seventy fifth, the last two components included, have the value R'{(EC)L(W{x})DD} and all the components comprised between the twenty sixth and the fifty fifth, the last two included, have the value zero, the sequence D(E{L'{(EC)L(W{x})DD}})R(W{L'{(EC)L(W{x})DD}})U becomes the GABNN output subvector component content value. And every time in the first modality input subvector both the components comprised between the fifty first and the seventy fifth, the last two components included, have the value R'{(EC)L(W{x})DD} and at least one component comprised between the twenty sixth and the fifty fifth, the last two components included, have a value different from zero, the sequence D(E{R'{(EC)L(W{x})DD}})D(EC)L(W{x})DD becomes the GABNN output subvector component content value.

To chain the output sequence activated by the general assembly calling sequence CN3DL both to the output sequence activated by the general assembly calling sequence CNLE' and to the output sequence activated by the general assembly calling sequence CN1URE'{UUU(W{R'{(EC)L(W{x})DD}})U} and to chain the output sequence activated by the general assembly calling sequence CNLE' both to the output sequence activated by the very same general assembly calling sequence CNLE' and to the output sequence activated by the general assembly calling sequence CN1URE'{UUU(W{R'{(EC)L(W{x})DD}})U}, the GABNN has to be trained to compute the following algorithm. We leave to the interested reader to find out the relative training procedures.

To implement the last algorithm we need the general assembly having as calling sequence, first, the value assumed by the GABNN first modality input subvector components comprised between the seventy fifth and the one hundredth, the last two components included, IF{FRST; SCND}, second, the state of the environment vector second from the top component and, third, the state of the environment vector third from the top component. As the last must have a constant value, actually, we are left with just two pos-

sible outcomes. To distinguish between the two possible outcomes, FRST is the name of the exit sequence corresponding to the case of all blank squares in the second from the top component, SCBD is the name of the exit sequence corresponding to at least one non blank square in the very same component. Accordingly, every time the GABNN first modality input subvector component comprised between the seventy fifth and the one hundredth, the last two components included, assume the value IF{FRST; SCND}, and both the environment vector central component and the environment vector second from the top component contain each a digit, either a one or a zero, the GABNN output subvector component contents have to assume the sequence of values FRST ≡ D(E{IF{FRST; SCND}})D(W{(LD){G}})UU, where E{IF{FRST; SCND}} is the calling sequence of the general assembly controlling the robot to erase the value IF{FRST; SCND}, where the present value of G is RD(W{CNLE'{IF{FRST; SCND}}})U and where W{CNLE'{IF{FRST; SCND}}} is the calling sequence of the general assembly trained to control the robot to write the value CNLE'{IF{FRST; SCND}} and where CNLE'{IF{FRST; SCND}} is the calling sequence for the CNLE' modified version, see below. Instead, every time the GABNN first modality input subvector component comprised between the seventy fifth and the one hundredth, the last two components included, assume the value IF{FRST; SCND}, and the environment vector central component contains a digit and the environment vector second from the top component is blank, the GABNN output subvector component contents have to assume the sequence SCND ≡ D(E{IF{FRST; SCND}}){W{CN1URE'{UUU(W{R'{(EC)L(W{x})DD}})U}})U, where the sequence W{CN1URE'{UUU(W{R'{(EC)L(W{x})DD}})U}} is the calling sequence of the general assembly trained to control the robot to write the value CN1URE'{UUU(W{R'{(EC)L(W{x})DD}})U}.

The GABNN output subvector sequence CNLE'{IF{FRST; SCND}} is obtained from the CNLE' standard version by juxtaposing to the latter the sequence D(W{IF{FRST; SCND}})U. That is, the CNLE' standard exit sequence D(EC)UL becomes the CNLE'{IF{FRST; SCND}} exit sequence D(EC)ULD(W{IF{FRST; SCND}})U.

Eventually, to chain the output sequence corresponding to the first modality input subvector component content CN3DL to the output subvector sequence corresponding to the input subvector content IF{FRST; SCND}, we must modify its exit conditions. Let us call CN3DL{IF{FRST; SCND}} the modified version of the calling sequence. Then, the CN3DL standard exit sequence U(EC)DL becomes the exit sequence $U(EC)DL(D)^{4}(W\{IF\{FRST;\ SCND\}\})U$.

6.4.4.2. *A second illustrative example.*

Let us consider a step of the multiplication algorithm: namely the presently observed digit of the second operand is a one, let us call it SOPOD, and the result component below it contains a digit, again. As in the addition algorithm case, to indicate the first operand current digit in the multiplication algorithm SOPOD is contained in the same environment vector as the times sign. To perform the multiplication, the first operand has to be copied, without erasing it, below the result. Let us call ℵ the first operand and its copy ℵ'. The expression ℵ' must be shifted leftward until its rightmost digit is below SOPOD, ie, below the component containing the times sign. Once shifted, ℵ' must be added to the present result PR. The addition result is the new PR. Then, to start anew the carousel, the times sign must be shifted one component to the left and the observed component has to be the one containing the first operand digit below the times sign. To apply the algorithm for the addition given in the chapter 6.3., PR must be copied as it stands, ie, without shifting it, on top or below ℵ'.

The actual algorithm is:

first, the component containing the times sign must be marked;
second, a string of zeros must be written in the components twenty four components below ℵ, starting from the component at the right of the one below SOPOD and going in the rightward direction until the component below ℵ rightmost digit is reached;
third, ℵ has to be copied five environment vectors below, let us call it ℵ ';
fourth, ℵ ' has to be shifted leftward until its rightmost digit is below SOPOD;
fifth, ℵ ' has to be erased after being shifted one component up, ie, next to the string of zeros, see the second step, let us call ℵ " the resulting expression, ie, the one composed by juxtaposing ℵ ' at the left of the string of zeros;
sixth, PR has to be erased after being copied four environment vectors and three components below;
seventh, to add the copied version of ℵ " to PR, the plus sign has to be written in the top component belonging to the environment vector containing the rightmost digit of both ℵ " and PR;

"**Comment**. To distinguish the GABNN output sequence controlling the addition execution implemented here as a routine to be called during the multiplication algorithm execution, without chaining also the GABNN output sequence controlling the addition execution given in the chapter 6. 3. 7., the latter output sequences have to be changed. That is, in the said top component the GABNN has to write the plus sign and the mark in the lover left square."

eighth, to perform the addition, the observed component must be the rightmost digit of ℵ ";

"**Comment**. Under the given condition the addition has to be automatically executed. However, to chain it with the following output sequence, the exit of the GABNN output sequence controlling the addition has to be changed."

ninth, once the addition has been carried out, both ℵ " and PR are erased;
tenth, let us call AR the addition result, AR has to be erased after being copied twenty four components up, in effect becoming the new PR;
eleventh, move so that the observed component must be the sign component marked in the first step;
twelfth, the observed component has to be erased;
thirteenth, a times sign has to be written in the next component at the left;
fourteenth, to start anew the carousel, the observed component has to be the first operand component, ie, the one two components below the top component.

To avoid to keep changing the procedures as the work is going on, we have to start from the algorithm last step, work out the procedures step by step until we arrive at the first step. That is, to chain the GABNN output sequence implementing the eleventh step, the one implementing the research for the times sign, to the GABNN output sequence implementing the last three steps, the former exit conditions have to be modified. In turn, to modify the search for the times sign, it must be known the GABNN output sequence realizing it. However, there is a range of possibilities in choosing the direction to search for the times sign, as the addition exit has to be modified and both ℵ ' and PR have to be erased, and they can be erased together or one after the other.

To fix the ideas, let AR be copied starting from the new PR leftmost and terminating on its rightmost digit. Then, the sequence of output subvector component values has to be modified so it controls, first, an upward movement, second, to write the calling sequence for a leftward search on the empty components, until it is found a written component, namely the component marked in the algorithm first step. The latter calling sequence is L'{G}, see the chapter 6.4.2., where G has to be (EC)L(W{x})DD to implement the multiplication algorithm last four steps. To work properly, the calling sequence W{L'{(EC)L(W{x})DD}} has to be written in the component below the sign component of its environment vector. That is, the output sequence copying AR twenty four components up in the rightward direction and terminating on the presently empty component, originally containing the rightmost digit of the copied expression, has to terminate with the exit sequence (Mυ)UU(W{L'{(EC)L(W{x})DD}}).

To sum it up, the steps eleven, twelve, thirteen and fourteen of the multiplication algorithm terminate with the environment vector central, ie, observed component being one of the first operand components. In other words, the last given exit sequence implies its initial observed component is five environment vectors and two components below the final one.

The output sequence to copy an expression twenty four components up in the rightward direction is the one corresponding to the GABNN input subvector components comprised between the twenty sixth and the fiftieth, the last two components included, assume the value CN4V4CULE, see the chapter 6.4.1.. The modified version name for the multiplication algorithm is

CN4V4CULE{(Mυ)UU(W{L'{(EC)L(W{x})DD}})}.

Whereas the output sequence exit corresponding to the value CN4V4CULE in the GABNN input subvector components comprised between the twenty sixth and the fiftieth, the last two components included, is U(EC)DL, see the chapter 6.3.14.1., the output sequence exit corresponding to

CN4V4CULE{(Mυ)UU(W{L'{(EC)L(W{x})DD}})}

is

U(EC)DL(Mυ)UU(W{L'{(EC)L(W{x})DD}}).

The GABNN output sequence to erase an expression starting from its leftmost, or rightmost, digit and writing the calling sequence in the environment vector topmost component has been given in the chapter 6.4.2.: its calling sequence is ELT. The corresponding output sequence exits have to be modified so to erase, once they have been added, both ℵ" and PR starting from their rightmost digits, ie, from the digit lying on top of AR rightmost digit.

Let

ELT{CN4V4CULE{(Mυ)UU(W{R'{(EC)R(W{x})DD}})}}

be the calling sequence of the general assembly controlling the output sequence erasing PR in the leftward direction. Whereas the output sequence exit corresponding to ELT is UU(EC)DD, see the chapter 6.4.2., the output sequence exit corresponding to the name

ELT{CN4V4CULE{(Mυ)UU(W{R'{(EC)R(W{x})DD}})}},

ie, to the name to be written in the environment vector top component is

```
UU(W{CN4V4CULE{(Mυ)UU(W{L'{(EC)L(W{x})DD}})}})DDD,
```

where the last D has to be inserted because AR has been written one component below PR.

To erase PR in a leftward direction, the starting component has to be the one containing PR rightmost digit. As also ℵ" is erased in the leftward direction, PR rightmost digit has to be found with the help of an output subvector component content sequence corresponding to the GABNN first modality input subvector first twenty five components value

```
R{G≡UU(W{ELT{CN4V4CULE{(Mυ)
        UU(W{R'{(EC)R(W{x})DD}})}}})DDD}.
```

Accordingly, the exit of the said output subvector component content has to be

```
UU(EC)(W{ELT{CN4V4CULE{(Mυ)
        UU(W{R'{(EC)R(W{x})DD}})}}})DDD}.
```

Let

```
ELT{R{UU(W{ELT{CN4V4CULE{(Mυ)
        UU(W{R'{(EC)R(W{x})DD}})}}})DDD}}
```

be the calling sequence of the general assembly controlling the output sequence erasing ℵ" in the leftward direction. Whereas the output sequence exit corresponding to ELT is UU(EC)DD, see the chapter 6.4.2., the output sequence exit corresponding to the name

```
ELT{R{UU(W{ELT{CN4V4CULE{(Mυ)
        UU(W{R'{(EC)R(W{x})DD}})}}})DDD}},
```

ie, to the name to be written in the environment vector top component is

```
UU(W{ELT{R{UU(W{ELT{CN4V4CULE{(Mυ)
        UU(W{R'{(EC)R(W{x})DD}})}}})DDD}}})DDD.
```

where the last D has to be inserted because PR has been written one component below ℵ".

To erase ℵ" in a leftward direction, the starting component has to be the one containing ℵ" rightmost digit. As the addition algorithm terminates on the left of the first operand leftmost digit, ℵ" rightmost digit has to be found with the help of an output subvector component content sequence corresponding to the GABNN first modality input subvector last twenty five component content value (RD){℘}, where

```
℘ ≡ UUUU(W{ELT{R{UU(W{ELT{CN4V4CULE{(Mυ)
        UU(W{R'{(EC)R(W{x})DD}})}}})DDD}}})DD.
```

Accordingly, the GABNN output subvector component content has to terminate with the sequence DD(E{(RD){℘}})UU℘.

To implement the eighth step of the multiplication algorithm, the GABNN output subvector component content sequence implementing the addition algorithm given in the chapter 6.3. has to be chained to the GABNN output subvector component content sequence implementing the erasing of the expression ℵ". To fix the ideas, the environment vector central component content is the first operand, ie, the expression ℵ". The envi-

ronment vector fourth from the top component content is the second operand, ie, the expression PR. The environment vector bottom component content is the addition result, ie, the expression AR. In conclusion, the Table 2.1. has to be modified as given in the Table 6.3..

TABLE 6. 3.

(+., b, **0**, v, b) $\Rightarrow$ DD(W{0})UUUU(EC)L(W{+}) (MPC)DD
(+., b, v, **0**, b) $\Rightarrow$ DD(W{0})UUUU(EC)L(W{+}) (MPC)DD
(+., b, v, **1**, b) $\Rightarrow$ DD(W{1})UUUU(EC)L(W{+}) (MPC)DD
(+., b, **1**, v, b) $\Rightarrow$ DD(W{1})UUUU(EC)L(W{+}) (MPC)DD
(+., **1**, v, v', b) $\Rightarrow$ DD(W{1})UUU(E{1})U(EC)L(W{+}) (MPC)DD
(+., b, **1**, **1**, b) $\Rightarrow$ DD(W{0})UUUU(EC)L(W{+}) (MPC)D(W{1})D
(+., **1**, v, **1**, b) $\Rightarrow$ DD(W{0})UUU(E{1})U(EC)L(W{+}) (MPC)D(W{1})D
(+., **1**, **1**, v, b) $\Rightarrow$ DD(W{0})UUU(E{1})U(EC)L(W{+}) (MPC)D(W{1})D
(+., **1**, **1**, **1**, b) $\Rightarrow$ DD(W{1})UUU(E{1})U(EC)L(W{+}) (MPC)D(W{1})D
(+., b, b, b, b) $\Rightarrow$ UU(EC)DDDDDDR(W{(RD){$\wp$}})UU

Table 6.3.. *The GABNN input-output correspondences for the addition as a subroutine for the multiplication algorithm.* Instead of the GABNN input subvector sequences, ie, instead of the one hundred and twenty five binary digit sequences, at the left we give the ten environment vectors as seen by the robot sensors, actually for typographical reasons they have been rotate by nineteen degree toward the left, ie, the rigth tmost component is the uppermost. The commas separate the content of different components, eg at the extreme left the plus sign and the period belong both to the top component. That is, the top component contains the plus sign and its lower left square has been marked, see the chapter 6.2.3.11.. At the right, we give, in the courier type, the corresponding GABNN output subvector sequences, ie, the commands the robot has to execute. The symbol b represents a blank space and the symbols v and v' represent, independently one from the other, either a blank space or the digit zero. The central component content is the first operand, ie, the expression $\aleph$ ". The fourth from the top component content is the second operand, ie, the expression PR. The bottom component content is the addition result, ie, the expression AR.

The procedures to train the GABNN in the control of the sequences given at the right hand side of the Table 6. 3. have been given, substantially, in the chapter 6.3.. Once the GABNN has been trained in the I/O relations given in the Table 6.3., it executes the addition every time the environment vector assumes one of the values given at the left hand side. To distinguish the addition algorithm presented in the chapter 6.3.7. from the one given here we have changed the GABNN output sequences. That is, instead of the plus sign, to implement the multiplication algorithm eighth step, both the mark and the plus sign have to be written in the top component of the environment vector containing, in its central component, the first operand digit. And to chain it, to the output sequence implementing the multiplication algorithm ninth step, the addition algorithm exit sequence has been modified.

Now the GABNN has to control the writing of both a mark and a plus sign in the top component of the environment vector containing the rightmost digit of both $\aleph$ ' and PR copies. Once PR has been copied, ie, at the end of the multiplication algorithm sixth step, the presently observed component is the one originally containing PR rightmost digit. The seventh step has to end with the observed component containing the addition first operand, ie, $\aleph$ ". Accordingly, the GABNN has to be trained to control the output sequence (Mδ)UUU(W{+}) (MPC)DD. Because it does not contain variables, ie, because its components do not depend on the state of the environment vector squares, it is convenient to juxtapose it to the exit sequence controlling the end of the PR copying and erasing. The latter actions are controlled by the GABNN output subvector sequence corresponding to

to the first modality input last twenty five components assuming the value CN4V3CDLE{W{+℘}}, ie, to the input sequence CN4V3CDLE, see the chapter 6. 4. 1., modified so to terminate with the sequence UUU(W{+}) (MPC)DD. Accordingly, the GABNN has to be trained have the output subvector exit sequence U(EC)DL(Mδ)UUU(W{+}) (MPC)DD every time its first modality input last twenty five components assume the value CN4V3CDLE{W{+℘}}.

The multiplication algorithm fifth, fourth and third steps have been substantially discussed in the chapter 6.4.4.1.. Actually, there is a small variant as in the step three of the present algorithm ℵ has to be copied five environment vectors, instead of three components, below the present one. That is, instead of the command CN3DL, now there is the command CN5VDL.

Accordingly, starting from the extreme left, let us see how to chain the output sequences to control the copying of ℵ, let us call ℵ' its copy, five environment vectors below the present one, followed by the output sequence to control the shift of ℵ' toward the left until it lays below a sequence of blank components, followed by the output sequence to move ℵ' one component up. In other words, we want to train the GABNN to control the sequences of primitives obtained by chaining the output sequences corresponding to the first modality input sequences CN5VDL, CNLE', CN1URE', where the second one, namely CNLE', have to be repeated again and again until the given condition is satisfied.

Let us recall the said output sequences entry and exit conditions. The output sequence activated by the calling sequence CN5VDL, see the chapter 6.4.1., starts its journey on the leftmost digit of the numeral ℵ to copy, let us call ℵ' the copied expression, and terminates its journey on the rightmost digit of ℵ. The output sequence activated by the calling sequence CNLE' starts its journey on the leftmost digit of ℵ' and terminates its journey on the component containing the rightmost digit of ℵ'. The output sequence activated by the calling sequence CN1URE' starts its journey on the rightmost digit of ℵ' and terminates its journey on the leftmost digit of the copy of the expression ℵ', the original having been erased.

That is, the output sequences corresponding to both the input sequences CN5VDL and CNLE' end their journeys on the rightmost digit of ℵ' and CNLE' starts its journey on the leftmost digit of the very same numeral. Accordingly, some intermediate output sequences to control the conditional movements are needed to compensate for the said discrepancies. In turn, once the output sequence corresponding to the input sequence CN5VDL has terminated its journey, there could be two different outcomes. If the component on top of the presently observed one is written on, it has to be chained to the output sequence corresponding to the first modality input CNLE'. Otherwise, namely if the component on top of the presently observed one is blank, it has to be chained to the output sequence corresponding to the first modality input CN1URE'.

The same has to happen once the output sequence corresponding to the first modality input CNLE' has terminated its journey. That is, the output sequence corresponding to the input sequence CNLE' has to be chained again to the output sequence corresponding to the input sequence CNLE', if the component on top of the presently observed one is written on. Otherwise, namely if the component on top of the presently observed one is blank, it has to be chained to the output sequence corresponding to the first modality input CN1URE'.

The output sequence corresponding to the first modality input CN1URE' has to be chained to the output sequence corresponding to the input sequence CN4V3CDLE{W{+℘}}. The former ends and the latter starts its journey on the leftmost digit of the very same numeral, namely ℵ' followed by a string of zeros. Accordingly, CN1URE' modified version, let s call it CN1URE'{℘}, has to terminate with the sequence D(EC)URDD(W{CN4V3CDLE{W{+℘}}})UU, where the calling sequence of

the general assembly trained to control the robot to write the value CN4V3CDLE{W{+℘}} is W{CN4V3CDLE{W{+℘}}}

To chain the output sequence activated by the general assembly calling sequence CN5VDL, see the chapter 6. 4. 1., both to the output sequence activated by the general assembly calling sequence CNLE' and to the output sequence activated by the general assembly calling sequence CN1URE'{CN4V3CDLE{W{+℘}}} and to chain the output sequence activated by the general assembly calling sequence CNLE' both to the output sequence activated by the very same general assembly calling sequence CNLE' and to the output sequence activated by the general assembly calling sequence CN1URE'{CN4V3CDLE{W{+℘}}}, the GABNN has to be trained to compute the following algorithm. We leave to the interested reader to figure out the relative training procedures.

To implement the last algorithm we need a general assembly having as calling sequence, first, the value COND{BLANK; NBLANK} assumed by the GABNN first modality input subvector components comprised between the seventy fifth and the one hundredth, the last two components included, second, the state of the environment vector second from the top component and, third, the state of the environment vector third from the top component.

As in the chapter 6. 4. 4. 1. the last has a constant value, so we are left with just two possible outcomes. Let us call BLANK the exit sequence corresponding to the case of all blank squares in the second from the top component, NBLANK the exit sequence corresponding to at least one non blank square in the very same component. Accordingly, every time the GABNN first modality input subvector component comprised between the seventy fifth and the one hundredth, the last two components included, assume the value COND{BLANK; NBLANK}, and both the environment vector central component and the environment vector second from the top component contain each a digit, either a one or a zero, the GABNN output subvector component contents have to assume the sequence of values D(E{COND{BLANK; NBLANK}})D(W{(LD){G}})UU ≡ BLANK, where W{(LD){G}} as been discussed in 6. 4. 3., where the value of G is RD(W{CNLE'{COND{BLANK; NBLANK}}})U, where W{CNLE'{COND{BLANK; NBLANK}}} is the calling sequence of the general assembly trained to control the robot to write the value CNLE'{COND{BLANK; NBLANK}} and where CNLE-'{COND{BLANK; NBLANK}} is the calling sequence for the CNLE' modified version. Instead, every time the GABNN first modality input subvector component comprised between the seventy fifth and the one hundredth, the last two components included, assume the value COND{BLANK; NBLANK}, and the environment vector central component contains a digit and the environment vector second from the top component is blank, the GABNN output subvector component content is the value D(E{COND{BLANK; NBLANK}})(W{CN1URE'{CN4V3CDLE{W{+℘}}}})U ≡ NBLANK, where the sequence W{CN1URE'{℘}} is the name of the general assembly trained to control the robot to write the general assembly calling sequence CN1URE'{CN4V3CDLE{W{+℘}}}.

The GABNN output subvector sequence CNLE'{COND{BLANK; NBLANK}} is obtained from the CNLE' standard version by juxtaposing to the latter the sequence D(W{COND{BLANK; NBLANK}})U. That is, the CNLE' standard exit sequence D(EC)UL becomes the CNLE'{COND{BLANK; NBLANK}} exit sequence D(EC)ULD(W{COND{BLANK; NBLANK}})U.

Eventually, to chain the output sequence corresponding to the first modality input subvector component content CN5VDL, see the chapter 6. 4. 1., to the output subvector sequence corresponding to the input subvector content COND{BLANK; NBLANK}, we must modify its exit conditions. Actually, the two terminate their journey five environment vectors apart. That is, the former ends its journey five environment vectors on top of the latter. Let us call CN5VDL{COND{BLANK; NBLANK}} the modified version of the

CN5VDL calling sequence. Then, the CN5VDL standard exit sequence DD(EC)UUUL becomes the CN5VDL{COND{BLANK; NBLANK}} exit sequence DD(EC)UUUL(Mδ)D(W{COND{BLANK; NBLANK}})U.

To chain the output sequence corresponding to the input subvector last twenty five component values WZRB to the output sequence corresponding to the input subvector last twenty five component values CN5VDL{COND{BLANK; NBLANK}}, namely to chain the writing of a string of zeros, twenty four components below, to the copy of ℵ five vectors below, the output sequence (EC)(W{0})DD(EC)UU, namely the WZRB exit sequence, has to be modified. Let us call WZRB{CN5VDL{COND{BLANK; NBLANK}}} the modified version of WZRB, both of them being the value assumed by the input subvector last twenty five components. Its exit sequence becomes (EC)(W{0})DD(EC)UUMυD(W{L{G ≡ R(W{CN5VDL{COND{BLANK; NBLANK}}})}}), where W{CN5VDL{COND{BLANK; NBLANK}}} is the calling sequence of the general assembly controlling the writing of the sequence CN5VDL{COND{BLANK; NBLANK}}.

6.4.4.3. *Training the GABNN to find an expression central comma.*

As an help in the algorithm to perform the substitution schema, see the chapter 6. 7., we present a procedure to train the GABNN to find the *central comma* in an algebraic expression. Given an algebraic expression containing open and closed parentheses, we propose the following algorithm to find whether or not they match. Starting from zero on the expression leftmost symbol, we go leftward adding a one every time we find an open parenthesis and subtracting a one every time we find an closed parenthesis. Every time we have a zero, the number of open match the number of closed parentheses. Let us call *match point* the symbol at the right of the closed parenthesis corresponding to the value zero, if any. By construction, the substitution schema leads to an expression with just one match point, the *central comma*. That is, we call *central comma* the match point in the substitution schema, once the expression leftmost parenthesis has been erased.

Given an expression obtained by repeatedly performing the substitution schema, with the sentence: *the GABNN has to find the central comma*, we mean the GABN has to start with the environment vector central component on the expression leftmost symbol and to terminate the journey with the central component on the expression central comma. With the sentence: *the GABNN has to find and mark the central comma*, we mean the GABNN has to start with the environment vector central component on the expression leftmost symbol, then, to find, to mark and to terminate the journey on the central component.

To find the central comma in an expression obtained by repeatedly performing the substitution schema, the environment vector central component must contain the expression leftmost symbol and the input subvector last twenty five component content value has to be CCB{J}, CCBM{J}, CCBE{J}, CCECR{J}, where J is the calling sequence of a previously trained general assembly, if the central component has to be found, found and marked, found and erased, found erased and copied at the right. Accordingly, the GABNN has to be trained in the algorithm:

first, to go two components down, to erase the component, ie, to erase the algorithm calling sequence, to go two components up, to go three components at the left;
second, to mark the observed component;

"**Comment**. The marked component is the starting point to store the expression representing the number of open parentheses."

third, to go three components at the right.
fourth, to go rightward until a component containing a parenthesis is found;
fifth,
if the component content is an open parenthesis,
(fifth, if first) to mark the observed component;
(fifth, if second) to go at the left end of the written component sequence;
(fifth, if third) to go three components at the left;
(fifth, if fourth) to go at the left end of the written component sequence;
(fifth, if fifth) to write a one in the component at the left;

"**Comment**. An open parenthesis as been found and the expression representing the number of open the parentheses has been increased by one."

(fifth, if sixth) to go at the right end of the written component sequence;
(fifth, if seventh) to go three components at the right;
(fifth, if eighth) to go rightward until a marked component is found;

"**Comment**. The present observed component is the one containing the last found open parenthesis."

(fifth, if ninth) to erase the mark from the observed component;
(fifth, if tenth) to go back at the algorithm fourth step;
otherwise, ie, if the component content is a closed parenthesis,
(fifth, otherwise first) to mark the observed component;
(fifth, otherwise second) to go at the left end of the written component sequence;
(fifth, otherwise third) to go three components at the left;
(fifth, otherwise fourth) to go at the left end of the written component sequence;
(fifth, otherwise fifth) to erase the leftmost component;

"**Comment**. As a closed parenthesis has been found, the expression representing the number of open parentheses has been decreased by one."

(fifth, otherwise sixth) to go one component at the right;
(fifth, otherwise seventh),

"**Comment**. As the expression representing the number of open parentheses has been decreased by one, it must be tested whether or not it is zero, ie, whether or not the component at the right is marked. If it is zero, ie, if the component at the right is marked the central comma has been found and the algorithm ends."

if the component is marked,
(fifth, otherwise seventh, if first) to erase the observed component;
(fifth, otherwise seventh, if second) to go two components at the right;
(fifth, otherwise seventh, if third) to go rightward until a marked component is found;

(fifth, otherwise seventh, if fourth) to erase the mark from the observed component;
(fifth, otherwise seventh, if fifth) to go one component at the right and execute the command J; **Algorithm's end.**

"**Comment.** Once the expression representing the number of open parentheses has been decreased by one, it has been tested and it is not zero, ie, the component at the right of the erased component is not marked. Accordingly, the central comma has not yet been found, the mark has to be erased from the component containing the last found closed parenthesis and the algorithm next step is the fourth."

otherwise, ie, if the component is not marked,
(fifth, otherwise seventh, otherwise first) to go at the right end of the written component sequence;
(fifth, otherwise seventh, otherwise second) to go three components at the right;
(fifth, otherwise seventh, otherwise third) to go rightward until a marked component is found;
(fifth, otherwise seventh, otherwise fourth) to erase the mark from the observed component;
(fifth, otherwise seventh, otherwise fifth) to go back at the algorithm fourth step;

To find and mark the central comma in an expression obtained by repeatedly performing the substitution schema, the GABNN has to be trained in the above given algorithm with the following modification:

(fifth, otherwise seventh, if fifth) to go one component at the right, mark the observed component and execute the command J; **Algorithm's end.**

To find and erase the central comma in an expression obtained by repeatedly performing the substitution schema, the GABNN has to be trained in the above given algorithm with the following modification:

(fifth, otherwise seventh, if fifth) to go one component at the right, erase the observed component and execute the command J; **Algorithm's end.**

To find and erase the central comma, to go two components at the right, then, to copy the original expression rightward one component down, the GABNN has to be trained in the above given algorithm with the following modification:

(fifth, otherwise seventh, if fifth) to go one component at the right, erase the observed component, to go two more components at the right, to copy, rightward one component down, the written component sequence and execute the command J; **Algorithm's end.**

To avoid to change the procedures as the work is going on, we have to start from the algorithm last steps. That is, to chain the GABNN output sequence implementing the step (fifth, otherwise seventh, if third), namely the step implementing the control of the rightward search until the marked component, actually the present expression match point, is found, to the GABNN output sequence implementing last three steps, the former exit conditions have to be modified.

If the component containing the central comma has to be found and marked, the GABNN training procedure implementing the algorithm last step, ie, implementing the step (fifth, otherwise seventh, if fifth) has to be such to generate the output subvector sequence corresponding to the input subvector component content value R(MPC)J.

If the component containing the central comma has to be found and erased, the GABNN training procedure implementing the algorithm last step, ie, implementing the step (fifth, otherwise seventh, if fifth) has to be such to generate the output subvector sequence corresponding to the input subvector component content value R(EC)J.

If the component containing the central comma has to be found, erased and, then, the right hand side of the original expression has to be copied rightward one component down, the GABNN training procedure implementing the algorithm last step, ie, implementing the step (fifth, otherwise seventh, if fifth) has to be such to generate the output subvector sequence corresponding to the input subvector component content value R(EC)RRU(W{CN1DLE{J}})D, where W{CN1DLE{J}} is the calling sequence of the general assembly controlling the robot to write the name CN1DLE{J}, where CN1DLE{J} is the calling sequence for a modified version of CN1DLE, see the chapter 6. 4. 3.. In other words, the GABNN has to learn to generate the output sequence U(CCR)(EC)D(CC1D)(EC)R, every time the environment vector second from the top component has the value CN1DLE{J} and the environment vector central component is written. And to generate the output sequence U(EC)DL{J} every time the environment vector second from the top component has the value CN1DLE{J} and the environment vector central component is blank. An illustrative example of the J value is given below, see the chapter 6.7..

If the component containing the central comma has to be only found, the output subvector sequence has to correspond to the input subvector sequence value R. Instead, the output subvector content implementing the step (fifth, otherwise seventh, if fourth) has to correspond to the input subvector component content EPC.

That is, the GABNN has to be trained to perform the algorithm last two steps every time the input subvector components have either the value (EPC)R(MPC), or the value (EPC)R(EC), or the value (EPC)R(EC)U(W{CND1DLE{J}})D.

To be trained to execute the step (fifth, otherwise seventh, if third) and to chain its corresponding output subvector sequence to the output subvector sequence corresponding to the value (EPC)R(MPC), (EPC)R(EC), (EPC)R(EC)U(W{CND1DLE{J}})D, the GABNN has to learn to generate the output subvector sequence corresponding to the input subvector last twenty five component content value (RM){G}, see the chapter 6. 4. 3., with either G ≡ (EPC)R(MPC), or G ≡ (EPC)R(EC), or G ≡ (EPC)R(EC)U(W{CND1DLE{J}})D.

To be trained both to execute the two steps (fifth, otherwise seventh, if first) and (fifth, otherwise seventh, if second) and to chain their corresponding output subvector sequence to the output subvector sequence corresponding to the value (RM){G}, the GABNN has to learn to generate the output subvector sequence corresponding to the input subvector value (EC)RRDD(W{(RM){G}})UU, where W{(RM){G}} is the calling sequence of the general assembly controlling the robot to write the sequence (RM){G}.

To evaluate the (fifth, otherwise seventh) step's conditional and to chain the corresponding output sequence to the output sequence implementing the (fifth, otherwise seventh, if first) step, the GABNN has to be trained to output a suitable sequence every time its first modality input subvector last twenty five components assume the values

COND1{THN1; ELS1}. That is, if the environment vector central component is marked and the last component content value is COND1{THN1; ELS1}, the GABNN output subvector component content has to be THN1 ≡ DD(EC)UU(EC)RRDD(W{(RM){G}})UU. Instead, if the environment vector central component is not marked and the last component content value is COND1{THN1; ELS1}, the GABNN output subvector component content has to be the sequence ELS1 ≡ DD(EC)UU..., we shall present later on. In both the outcomes the sequence EC first occurrence, the one between the DD and UU sequences, is needed to erase the calling sequence COND1{THN1; ELS1}.

To be trained both to execute the six steps from (fifth, otherwise first) to (fifth, otherwise sixth) and to chain their corresponding output subvector sequence to the output subvector sequence corresponding to the step (fifth, otherwise seventh), the GABNN has to learn to respond to the input subvector value (MPC)DD(W{(LD){LLLDD(W{(LD){(EC)DD(W{COND1{THN1; ELS1}})UU}})UU}})UU. That is, the GABNN has to control the robot, first, to mark the observed component, second, to write in the bottom component the calling sequence (LD){LLLDD(W{(LD){(EC)DD(W{COND1{THN1; ELS1}})UU}})UU} to move on the left end of the present written component sequence, third, to move three components at the left, fourth, to write in the bottom component the calling sequence (LD){(EC)DD(W{COND1{THN1; ELS1}})UU} to move on the left end of the present written component sequence, fifth, to erase the observed component, sixth, to move one component at the left, seventh, to chain the resulting output sequence by writing in the bottom component the calling sequence COND1{THN1; ELS1} for the (fifth, otherwise seventh) step conditional.

The output sequence implementing the ten steps from (fifth, if first) to (fifth, if tenth) has to be chained to the output sequence implementing the fourth step conditional. In turn one of the latter two output sequences has to be chained to the output sequence corresponding to the (first, if first) step. Accordingly, the output sequence implementing the fourth step has to be the one corresponding to the environment vector last component content (RP){W{COND2{THN2; ELS2}}}, where THN2 ≡ DD(EC)UU... is the conditional first output sequence to be chained to the output sequence implementing the (first, if first) step, we are going to discuss next, and where the conditional second output sequence, namely ELS2 ≡ DD(EC)UU(MPC)DD(W{(LD){LLLDD(W{(LD){(EC)DD(W{COND1{THN1; ELS1}})UU}})UU}})UU, is the one chained to the previously given output sequence corresponding to the step (fifth, otherwise first).

As we have seen, the fourth step is implemented, repetitively, in the output sequence corresponding to the environment vector last component content value (RP){W{COND2{THN2; ELS2}}}, ie, until the observed component content is a parenthesis, an open or a closed one. The fourth step two outcomes, corresponding to the open and closed parenthesis, respectively, are chained to the (first, if first) and to the (first, otherwise first) steps. Accordingly, the output sequence corresponding to the three step sequence (fifth, if seventh), (first, if eighth), (first, if ninth) and (first if tenth) has to chained to the output sequence corresponding to the fourth step. To this end, the GABNN has to be trained to generate the output subvector component content sequence corresponding to the input subvector last twenty five component content value (RD){G* ≡ (R)^{3}DD(W{(RM){G** ≡ (EPC)DD(W{(RP){G ≡ DD(W{COND2{THN2; ELS2}})UU}})UU}})UU}.

To be trained both to execute the steps (fifth, if fifth) and (fifth, if sixth) and to chain their corresponding output subvector sequence to the output subvector sequence corresponding to the value (RD){G*}, the GABNN has to learn to generate the output subvector sequence corresponding to the input subvector last component content value (RD){G*** ≡ L(W{1})RDD(W{(RD){G*}})UU}.

To be trained both to execute the two steps (fifth, if third) and (fifth, if fourth) and to chain their corresponding output subvector sequence to the output subvector sequence corresponding to the value (RD){G***}, the GABNN has to learn to generate the output subvector sequence corresponding to the input subvector last component content value (LD){G**** ≡ (L)^{3}DD(W{(RD){G***}})UU}.

To be trained both to execute the two steps (fifth, if first) and (fifth, if second) and to chain their corresponding output subvector sequence to the output subvector sequence corresponding to the value (LD){G****}, the GABNN has to learn to generate the output subvector sequence corresponding to the input subvector last component content value (LD){G***** ≡ (MPC)DD(W{(LD){G****}})UU}.

Accordingly, now we can complete the previously introduced second conditional output sequence, ie, the GABNN has to be trained to output the sequence THN2 ≡ DD(EC)(W{(LD){G*****}})UU, every time the environment vector last component content is COND2{THN2; ELS2} and the environment vector central component content is an open parenthesis.

To implement the algorithm first three steps and to chain their corresponding output sequence with the output sequence implementing the algorithm fourth step, every time the input subvector last twenty five component content value is CCB, CCBM, CCBE, CCECR{J}, and the observed component is written, the GABNN has to be trained to generate the output sequence DD(EC)UU(L)^{3}(MPC)(R)^{3}DD(W{(RD){G*}})UU.

To evaluate the (fifth, otherwise seventh) step's conditional and to chain the corresponding output sequence to the output sequence implementing the five steps (fifth, otherwise seventh, otherwise first), (fifth, otherwise seventh, otherwise second), (fifth, otherwise seventh, otherwise third), (fifth, otherwise seventh, otherwise fourth) and (fifth, otherwise seventh, otherwise fifth), the GABNN has to be trained to output a suitable sequence every time its first modality input subvector last twenty five components assume the values COND1{THN1; ELS1}. That is, if the environment vector central component is marked and the last component content value is COND1{THN1; ELS1}, we have seen the GABNN output subvector component content has to be the sequence THN1 ≡ DD(EC)UU(EC)RRDD(W{(RM){G}})UU. Now we can complete ELS1 output sequence, ie, if the environment vector central component is not marked and the last component content value is COND1{THN1; ELS1}, the GABNN output subvector component content has to be the sequence ELS1 ≡ DD(EC)(W{(RD){G*}})UU, where G* ≡ (R)^{3}DD(W{(RM){G**}})UU, where G** ≡ (EPC)DD(W{(RP){G}})UU}})UU}, where G ≡ DD(W{COND2{THN2; ELS2}})UU. As before said, in both THN1 and ELS1 the sequence EC first occurrence controls the robot to erase the calling sequence COND1{THN1; ELS1}.

6.5. Training the GABNN to compute the multiplication.

In accordance with the notation introduced in the chapters 2.2. and 6.3., we assume the multiplication, and the addition, operand digits to be written one below the other. The first, second, operand digits are written in the third, fourth, component, from the top, of consecutive environment vectors, in a leftward direction. That is, with the rightmost as the least significant digit. A times, plus, sign is written on top of the least significant digit, namely in the topmost component of the rightmost vector in the environment vector sequence. The result, carry, is the fifth, second, from the top, component content. Accordingly, we call sign component the environment vector topmost component, carry component the next to the topmost component, first operand component the middle component, second operand component the fourth component, result component the bottommost component. For simplicity sake, we discuss the case of the first operand represented

by a numeral with a number of digits no smaller than the second operand, leaving to the interested reader to give the procedure for the case of the first operand represented by a numeral with a number of digits smaller than the second operand.

The two operand multiplication initial conditions are: the first operand rightmost digit is the present digit, the top component content is the times sign, the carry component is empty, the second operand rightmost digit is the content of the fourth from the top component and the present result is the empty sequence. When the said conditions are meet the GΑBNN can be trained to perform automatically the multiplication. That is, it can be trained, first, to multiply the first operand times the second operand present digit, namely the one lying below the times sign; second, to shift the result so obtained until its rightmost digit is under the times sign; third, to sum it to the present result: let us call the expression so obtained the present result; fourth, to shift the times sign in the next component at its left. Accordingly, the second operand next, toward the left, digit is the second operand present digit. If the fourth component content is a blank, the GΑBNN starts the algorithm exit sequence and ends its journey with the central component on the result rightmost digit. Because the operands are binary numerals, if the second operand present digit is a zero, the first step outcome is a zero. A copy of the first operand, otherwise.

In the chapter 6.3. we have presented a training procedure for the addition as a stand alone algorithm. However, in the multiplication algorithm third step, we need to modify its entry and exit conditions, ie, we need a new version to link its output sequences to the output sequences performing the multiplication algorithm, see the chapter 6.4.4.2. and the Table 6.4..

We give in the Table 6.4. a version of the GΑBNN input-output correspondences for the multiplication. That is, we give the GΑBNN output sequences corresponding to the environment vector input configurations as seen by the robot sensors. We leave to the interested reader to work out the relative training procedures, explicitly.

TΑBLE 6. 4.

(x, b, v, **0**, b) ⇒ DD(W{0})UUUU(E{x})L(W{x})DD

"**Comment**. First, the observed component is the top component; second, the digit zero is written in the result component; third, the times sign is erased; fourth, the times sign is written at the left; fifth, the observed component content is the first operand component below the last written times sign."

(x, b, v, **1**, b) ⇒ UU(MPC)DDDD(W{(LD){G ≡ RU(W{CN3DL{IF{FRST; SCND}}})UU}})

"**Comment**. First, the observed component is shifted from the first operand to the, times, sign component; second, the observed component is marked, see the chapter 6.2.3.11.; third, the observed component is the fifth from the top, namely the, presently blank, result component; fourth, in the observed component is written the calling sequence for the leftward conditional movement; fifth, the conditional movement terminates on the first operand leftmost digit, see the chapters 6.4.2. and 6.4.3.1., as R is the first symbol in the output sequence corresponding to G; sixth, the first operand is copied, and not erased, on the row of the environment vector components below the one containing the first digits, actually a number of zeros, of the result, let us call ℵ such first copy; seventh, if the rightmost digit of the copied numeral is below a

blank component, then, ℵ is copied one component up and, then, ℵ is erased, let us call ℵ ' the new copy lying on the left of the result first digits, else, ℵ is shifted leftward until the condition is fulfilled, now the environment vector observed component content is the present result leftmost digit; eighth, the sign component becomes the observed component: it still belongs to the same environment vector containing the result leftmost digit; ninth, the observed component is shifted rightward until it is the one originally containing the times sign; tenth, the observed component is erased; eleventh, the next component at the left is written with the times sign; twelfth, the observed component is the first operand component."

(x, b, v, **0**, v') ⇒ UU(E{x})L(W{x})DD

"**Comment**. First, the observed component is the top component; second, the times sign is erased; third, the times sign is written at the left; fourth, the observed component is the first operand component below the last written times sign."

(x, b, v, **1**, v') ⇒ UU(MPC)DDMδR(MPC)MυUUW{R{G ≡
MδD(W{WZRB{CN5VDL{COND{BLANK; NBLANK}}}})UU}}DD

"**Comment**. First, the observed component is shifted from the first operand to the, times, sign component; second, the observed component is marked, see the chapters 6.2.3.11. and 6.4.4.2. ; third, the observed component is the third from the top, namely the first operand component; fourth, the observed component is the one belonging to the fifth environment vector below; fifth, the observed component is marked; sixth, the observed component is again the one containing the first operand; seventh, the observed component is the first operand rightmost digit; eighth, the observed component is the one twenty four components below; ninth, a string of zeros is written in a leftward direction, starting from the observed component until it is encountered the component marked in the fifth step; tenth, the marked component is erased; eleventh, the second operand is copied twenty five components below, let us call ℵ ' the copy; twelfth, ℵ ' is shifted leftward until its rightmost digit is below the times sign; thirteenth, ℵ ' is shifted one component up; fourteenth, the present result is erased after being copied twenty three component below; fifteenth, ℵ ' is added to the present result copy; sixteenth, the addition result is copied twenty four components up, so becoming the new present result; seventeenth, the top component containing the times sign marked in the second step is erased; eighteenth, a times sign is written in the next component at the left; nineteenth, the observed component is the one containing the new second operand digit, ie, the one below the component containing the last written times sign."

(x, b, v", b, v'") ⇒ UU(E{x})R(W{R{G ≡ DD}})DD

"**Comment**. First, the observed component is the top component; second, the times sign is erased; third, the calling sequence for the general assembly controlling the search of the first operand rightmost digit is written at the right, see the chapter 6.4.; fourth, the observed component

is the first operand component; fifth, once the rightmost digit of the first operand has been found, the observed component is the one containing the rightmost result digit, two components below."

Table 6.4.. *The GABNN input-output correspondences for the multiplication.* Instead of the GABNN input subvector sequences, ie, instead of the one hundred and twenty five binary digit sequences, at the left we give the environment vectors as seen by the robot sensors. For typographical reasons they have been rotate by nineteen degree toward the left, ie, the leftmost component is the uppermost, actually. At the right we give, in the `courier` type, the corresponding GABNN output subvector sequences, ie, the commands the robot has to execute to perform the multiplication, see the chapters 6.2.3.17. and 6.4.. At the left hand side: the symbol x represents the times sign; the symbol b represents a blank space; the symbols v and v' represent, independently one from the other, either the digit one or the digit zero; the symbols v'' and v''' represent, independently one from the other, either a blank space or the digit one or the digit zero.

6.6. Training the GABNN to compute the delta function.

By definition, the delta function with two arguments is:

$$\Delta[m, n] = 1 \text{ if } m = n$$
$$\Delta[m, n] = 0 \text{ if } m \neq n.$$

In terms of environment vectors and GABNN output sequences, we represent it in the Table 6. 5.. As the GABNN output subvector sequences given in the Table 6. 5. have been discussed in the chapters 6. 2. and 6. 4., we leave it to the interested reader to work out the relative training procedures.

TABLE 6. 5.

(Δ, b, v, v, b) ⇒ `UU(EC)L(W{Δ})RDDDD(W{1})UUL`

"**Comment**. First, the observed component is the top component; second, the Δ sign is erased; third, the Δ sign is written in the component at the left; fourth, the observed component is the result component below the previously erased top component; fifth, a one is written in the observed result component; sixth, the observed component is the first operand component below the last written top component."

(Δ, b, v', v'', b) ⇒ `UU(EC)L(W{Δ})RDDDD(W{0})UUL`

"**Comment**. First, the observed component is the top component; second, the Δ sign is erased; third, the Δ sign is written in the component at the left; fourth, the observed component is the result component below the previously erased top component; fifth, a zero is written in the observed result component; sixth, the observed component is the first operand component below the last written top component."

(Δ, b, v''', v'''', b) ⇒ `UU(EC)L(W{Δ})RDDDD(W{0})UUL`

"**Comment**. First, the observed component is the top component; second, the Δ sign is erased; third, the Δ sign is written in the component at the left; fourth, the observed component is the result component below the previously erased top component; fifth, a zero is written in the

observed result component; sixth, the observed component is the first operand component below the last written top component."

(Δ, b, b, b, b) ⇒ UU(EC)RDDDDD(W{EWL{1v0}})U

"**Comment**. First, the observed component is the top component; second, the Δ sign is erased; third, the observed component is the next component at the right; fourth, the observed component is the fifth component down, namely the top component of the next environment vector below; fourth, in the observed component is written the calling sequence EWL{1v0} of the general assembly controlling the erasing of and, then, the writing in the result component; fifth, the observed component is the result component."

Table 6.5.. *The GABNN input-output correspondences for the delta function.* Instead of the GABNN input subvector sequences, ie, instead of the one hundred and twenty five binary digit sequences, at the left we give the environment vectors as seen by the robot sensors. For typographical reasons they have been rotate by nineteen degree toward the left, ie, the leftmost component is the uppermost, actually. At the right hand side we give, in the courier type, the corresponding GABNN output subvector sequences, ie, the commands the robot has to execute to compute the delta function, see the chapters 6.2.3.17. and 6.4.. The sequence W{Δ} represents the calling sequence of the general assembly controlling the robot to write the Δ symbol, ie, the delta function sign. The sequence W{EWL{1v0}} represents the calling sequence of the general assembly controlling the robot to write the sequence EWL{1v0}, ie, the name of the procedure, first, to erase the present component, second, to write either a zero or a one, depending on whether the erased symbol was a one or a zero, see the chapter 6.4.2.3. At the left hand side: the symbol Δ represents the delta function name; the symbol b represents a blank space; the symbol v represents a digit, either zero or one; the symbols v' and v" represent one the digit zero and the other the digit one, or vice versa; the symbols v"' and v"" represent one a digit, either a one or a zero, and the other a blank space, or vice versa.

6.7. Training the GABNN to perform the substitution.

If the GABNN has been previously trained to compute the two variable functions σ, f_1, f_2, it can be trained to perform the substitution schema

$$\sigma[f_1[x_1, x_2], f_2[x_1, x_2]].$$

That is, the GABNN learns to start with the central component containing the rightmost closed parenthesis and to end its journey with the central component containing the result rightmost digit and all the function names σ, $f_1[x_1, x_2]$, $f_2[x_1, x_2]$, ie, the operation symbol and all the arguments erased. The GABNN learns to perform the algorithm every time the central component content is a closed parenthesis and the bottom component content is the value SS{J*}, where J* is the calling sequence of a previously trained general assembly.

The procedure to train the GABNN to perform the substitution schema is the following:

first, to go two components down, to erase the component, namely to erase the algorithm calling sequence, to go two components up, to go leftward until the leftmost symbol is the content of the central component;
second, to mark the fourth from the top component, ie, the component below the one containing the said leftmost symbol;
third, to go on the next component at the right;
fourth, to erase the observed component, ie, the component holding the leftmost open parenthesis;

fifth, to go on the next component at the right, ie, on the component holding the symbol f_1;
sixth, to copy, rightward and two components down, the right-hand side symbol sequence, ie, the sequence $f_1[x_1, x_2]$, $f_2[x_1, x_2]]$;
seventh, to find and erase the central comma, see in the chapter 6.4.4.3. the algorithm last given exit sequence;
eighth, to go two components at the right;
ninth, to copy, rightward and one component down, the right-hand side written component sequence, ie, the sequence $f_2[x_1, x_2]]$;
tenth, to erase the observed component, ie, the component holding the rightmost closed parenthesis;
eleventh, to go on the sequence leftmost written component;
twelfth, to copy leftward the written component sequence until its rightmost written component is the fourth from the top component belonging to the environment vector having as sign component content the rightmost symbol, ie, the operation symbol;
thirteenth, to go one component at the right and one component up;
fourteenth, to go rightward until the leftmost written component if found, ie, until the first operand leftmost symbol is found;
fifteenth, to copy leftward the just found written component sequence until its rightmost written component is the central component of the environment vector having as sign component content the rightmost symbol, ie, the operation symbol;

"**Comment**. The central component contains the first operand rightmost symbol. If both operands are numerals, the function represented by the sign component content can be performed."

sixteenth,
if both the first and the second operand components contain digits or one contains a digit and the other is blank
(sixteenth, if first)
if the second from the top component is marked
(sixteenth, if first, if first) to erase the mark in the second from the top component below the rightmost written sign component;
(sixteenth, if first, if second) to mark the component at the right of the rightmost written sign component;
(sixteenth, if first, if third) to go one component at the left and two components down;

"**Comment**. Either if the second from the top component is not marked or if it was marked, the function represented by the sign component content can be computed, now."

(sixteenth, if second) to perform the operation ℑ{ℑ**} indicated by the content of the rightmost written sign component;

"**Comment**. The current function has been computed, its name and its arguments have been erased. The observed compo-

nent content is the function value, ie, the result rightmost digit."

(sixteenth, if third) to go one component at the right;
(sixteenth, if fourth),
if the observed component is marked
(sixteenth, if fourth, if first) to erase the component;
(sixteenth, if fourth, if second) to go one component at the left;
(sixteenth, if fourth, if third) to execute J*. "**Algorithm's ends**."
otherwise, ie, if the observed component is not marked
(sixteenth, if fourth, otherwise first) to go one component at the left;
(sixteenth, if fourth, otherwise second) to copy the result rightward until its rightmost digit is below the sign component rightmost symbol;

"**Comment**. The central component contains one of the two operands rightmost digit. If it is the second operand, the function represented by the sign component content has to be computed."

(sixteenth, if fourth, otherwise third),
if the fourth from the top component is non written

"**Comment**. The central component contains the second operand rightmost digit. Both operands are numerals."

(sixteenth, if fourth, otherwise third, if first) to go one component up;
(sixteenth, if fourth, otherwise third, if second) to go back at the (sixteenth, if first) step;
otherwise, ie, if the fourth from the top component is written

"**Comment**. The central component contains the first operand rightmost digit. The second operand is either an expression or is a function to be computed."

(sixteenth, if fourth, otherwise third, if third),
if the fourth from the top component contains either a zero or a one
(sixteenth, if fourth, otherwise third, if third, if first) to go back at the algorithm (sixteenth, if first) step;
otherwise, ie, if the fourth from the top component contains a symbol different from a digit

(sixteenth, if fourth, otherwise third, if third, otherwise first) to go one component down;

"**Comment**. The central component contains the second operand rightmost symbol and the second operand is a function to be computed."

(sixteenth, if fourth, otherwise third, if third, otherwise second) to go at the left end of the sequence of written components;
(sixteenth, if fourth, otherwise third, if third, otherwise third) to copy the sequence of written components until its rightmost written component becomes the central component of the environment vector containing in the second from the top component the first non written component at the left of the one containing the first operand leftmost digit;
(sixteenth, if fourth, otherwise third, if third, otherwise fourth) to go at the left end of the last copied sequence of written components;
(sixteenth, if fourth, otherwise third, if third, otherwise fifth) to go back to the algorithm third step;

otherwise, ie, either the first or the second or both operand components contain symbols different from zero or one;
(sixteenth, otherwise first),
if the central component, ie, the first operand contains a symbol different from a digit

"**Comment**. The central component contains the first operand rightmost symbol and the first operand is a function to be computed."

(sixteenth, otherwise first, if first) to go at the left end of the sequence of written components;
(sixteenth, otherwise first, if second) to copy the sequence of written components one component at the left of the component containing the second operand leftmost digit;
(sixteenth, otherwise first, if third) to go at the left end of the sequence of written components;
(sixteenth, otherwise first, if fourth) to go back to the algorithm third step;
otherwise, ie, if the central component, ie, the first operand contains a digit and the fourth from the top component, ie, the second operand contains a symbol different from a digit
(sixteenth, otherwise first, otherwise first) to go one component down;

"**Comment**. The central component contains the second operand rightmost symbol and the second operand is a function to be computed."

(sixteenth, otherwise first, otherwise second) to go at the left end of the sequence of written components;
(sixteenth, otherwise first, otherwise third) to copy the sequence of written components one component at the left of the component containing the first operand leftmost digit;
(sixteenth, otherwise first, otherwise fourth) to go at the left end of the sequence of written components;
(sixteenth, otherwise first, otherwise fifth) to go back to the algorithm third step;

To avoid to change the procedures as we present them, we have to start from the algorithm last step, ie, from the step (sixteenth, if fourth, if third) and to go backward.

To perform the algorithm last three steps, namely the steps (sixteenth, if fourth, if first), (sixteenth, if fourth, if second) and (sixteenth, if fourth, if third), the GABNN output subvector component content has to be: `(EC)L(J*)`. To implement the step (sixteenth, if fourth) there must be a general assembly having as calling sequence, first, the value `IF1{FRST1 ≡ D(EC)U(EC)L(J*); SCND1}` assumed by the GABNN first modality input subvector components comprised between the seventy fifth and the one hundredth, the last two components included, second, the value corresponding to the environment vector central component state. As the said state can be either written or blank, the corresponding value is either one or zero. Accordingly, whereas `FRST1` is the name of the exit sequence `DD(EC)UU(EC)L(J*)` corresponding to at least one written square in the central component, `SCND1` is the name of the exit sequence, we shall present later on, corresponding to all blank squares in the very same component.

To discuss how to train the GABNN to output a command sequence to perform the steps (sixteenth, if second) and (sixteenth, if third), at the same time chaining the latter to the step (sixteenth, if fourth), let us call $\Im\{J^{**}\}$ the content of the rightmost written sign component. That is, $\Im\{J^{**}\}$ is the name of the operation to be carried out in the algorithm (sixteenth, if second) step. As the number of functions nested within σ is finite and $\Im\{J^{**}\}$ is either the function σ, or one of its arguments, or an argument of one of its arguments, or ..., by hypothesis, the GABNN has been already trained to compute it. The GABNN has been trained to perform the addition, multiplication and delta operation, with the following entry and exit conditions. The journey starts on the first operand rightmost digit and ends on the result rightmost digit having, first, erased the two operands and the operation sign, second, copied the result in the sign component row. Accordingly, in computing $\Im\{J^{**}\}$ the GABNN shall terminate its journey ending on the rightmost digit of the result, copied in the sign component row after having erased the two operands and the $\Im\{J^{**}\}$ sign. To chain it to the output sequence implementing the (sixteenth, if third) and (sixteenth, if fourth) steps, eg see the chapter 6. 4. 4. 2. and the Table 6. 3., $\Im\{J^{**}\}$ exit sequence has to be modified so to control, first, a movement on the next component at the right, second, to write in the fourth from the top component the value `IF1{FRST1; SCND1}`. Let us set `J** ≡ {R(W{IF1{FRST1; SCND1}})}`, where `W{IF1{FRST1; SCND1}}` is the name of the general assembly controlling the robot to write in the fourth from the top component the value `IF1{FRST1; SCND1}`. Eventually, the output sequences corresponding to $\Im\{$`R(W{IF1{FRST1; SCND1}})`$\}$ single instances, see the modifications we have presented for the addition algorithm in the Table 6. 3., are the modified versions of $\Im\{J^{**}\}$ corresponding output sequences.

To train the GABNN to output a command sequence to perform the steps (sixteenth, if first), (sixteenth, if first, if first), (sixteenth, if first, if second) and (sixteenth, if first, if third), at the same time chaining the latter to the step (sixteenth, if second), there must be a general assembly having as calling sequence, first, the value IF2{FRST2 ≡ DD(EC)UU(ESC)UUR(MPC)LDD; SCND ≡ DD(EC)UU} assumed by the GABNN first modality input subvector components comprised between the hundred and first and the hundred and twenty fifth, the last two components included, second, the value corresponding to a mark in the environment vector second from the top component.

"**Comment.** The sequence DD(EC)UU(ESC)UUR(MPC)LDD, implementing the steps (sixteenth, if first, if first), (sixteenth, if first, if second) and (sixteenth, if first, if third), has to be executed only if the environment vector second from the top component is marked. Otherwise, ie, if either the bottom component does not contain the value IF2{FRST2 ≡ DD(EC)UU(ESC)UUR(MPC)LDD; SCND ≡ DD(EC)UU} or the second from the top component has not been marked, or both, the step (sixteenth, if second) has to be executed. The latter calling sequence is the environment vector present content, so no writing command has to be inserted in the FRST2 sequence. However, the sequence DD(EC)UU has to be executed in any case as its task is to erase the very same value IF2{FRST2 ≡ DD(EC)UU(ESC)UUR(MPC)LDD; SCND ≡ DD(EC)UU}''.

To train the GABNN to evaluate the sixteenth step's conditional and to chain the corresponding output sequence to the above given output sequence implementing the step (sixteenth, if first) and to the output sequence implementing the step (sixteenth, otherwise first), to be given later on, the GABNN first modality input subvector last twenty five components assume the value IF3{FRST3 ≡ DD(EC){W{IF2{FRST2}}}UU; SCND3} and both the environment vector central and fourth from the top components contain either a digit each, or one a digit and the other a blank. Accordingly, whereas FRST3 is the name of the exit sequence DD(EC){W{IF2{FRST2}}}UU corresponding to the case of one digit and a blank or two digits in the third and in the fourth from the top component, SCND3 is the name of the exit sequence, we shall present later on, corresponding to the presence of symbols different from digits in both components. The name W{IF2{FRST2}} is the calling sequence of the general assembly trained to control the writing of the value IF2{FRST2} in the environment vector bottom component,

To evaluate the fifteenth step's conditional shift and to chain the corresponding output sequence to the above given output sequence implementing the step sixteenth, ie, both (sixteenth, if first) and (sixteenth, otherwise first), the GABNN has to be trained to output the command sequence corresponding to the first modality input subvector last twenty five component value CNLBE{IF4{FRST4; SCND4}}, ie, a modified version of CNLE, see the chapter 6.4.1.. In other words, the GABNN has to be trained to output the sequence DD(CCR)(EC)UU(CCL)(EC)R every time CNLBE{IF4{FRST4; SCND4}} is the environment vector last component content and the central component is written. And to output the sequence DD(EC)L{W{IF4{FRST4; SCND4}}}UU every time CNLBE{IF4{FRST4; SCND4}} is the environment vector last component content and the central component is blank. Where IF4{FRST4 ≡ DD(EC){W{(LD){G ≡ DD(W{CNLBE{IF4{FRST4; SCND4}}})UU}}}UU; SCND4 ≡ DD(EC){W{IF3{FRST3; SCND3}}}UU} is the calling sequence of two general assemblies: one to control the output sequence we have called FRST4 and the other the output sequence we have called SCND4. The sequence corresponding to FRST4, SCND4, is outputted when the environment vector bottom component content is IF4{FRST4; SCND4} and the sign component is blank, the sign component is written. Moreover, W{CNLBE{IF4{FRST4; SCND4}}} is the calling sequence of the general assembly controlling the robot to write the name CNLBE{IF4{FRST4; SCND4}}.

To train the GABNN to output the sequence implementing the fourteenth step and to chain it to the above given output sequence implementing the fifteenth step, it has to learn to generate the output sequence DD(E{RB'{G}})R(W{RB'{G}})UU, see the chapter 6.4.3., every time in its first modality input subvector both the last twenty five components have the value RB'{G} and all the components comprised between the fifty first and the seventy fifth, the last two components included, have the value zero. And to generate the output sequence DD(E{RB'{G}})UUG every time in the first modality input subvector both the last twenty five components have the value RB'{G} and at least one component comprised between the fifty first and the seventy fifth, the last two components included, have a value different from zero, where G ≡ DD{W{CNLBE{IF4{FRST4; SCND4}}}}UU.

To evaluate the twelfth step's conditional shift and to chain the corresponding output sequence to the thirteenth step and, then, to the above given output sequence implementing fourteenth step, the GABNN has to be trained to output the command sequence corresponding to the first modality input subvector last twenty five component value CNLBE{IF5{FRST5; SCND5}}, ie, a modified version of CNLE, see the chapter 6. 4. 1.. In other words, the GABNN has to be trained to output the sequence DD(CCR)(EC)UU(CCL)(EC)R every time CNLBE{IF5{FRST5; SCND5}} is the environment vector last component content and the central component is written. And to output the sequence DD(EC)L{W{IF5{FRST5; SCND5}}}UU every time CNLBE{IF5{FRST5; SCND5}} is the environment vector last component content and the central component is blank. Where IF5{FRST5; SCND5} is the calling sequence of two general assemblies: one to control the output sequence we have called FRST5 ≡ DD(EC){W{(LD){DD(W{CNLBE{IF5{FRST5; SCND5}}}})UU}}UU and the other the output sequence we have called SCND5 ≡ DD(EC)UR{W{RB'{G ≡ DD{W{CNLBE{IF4{FRST4; SCND4}}}}UU}}}UU. The sequence corresponding to FRST5, SCND5, is outputted when the environment vector bottom component content is IF5{FRST5; SCND5} and the sign component is blank, the sign component is written. That is, to decide whether to output FRST5 or SCND5, the top and the fourth from the top component content has to be examined. Moreover, W{CNLBE{IF5{FRST5; SCND5}}} is the calling sequence of the general assembly controlling the robot to write the name CNLBE{IF5{FRST5; SCND5}}.

To train the GABNN to output the sequence implementing the eleventh step and to chain it to the above given output sequence implementing the twelfth step, it has to learn to generate the output sequence DD(E{(LD){G}})L(W{(LD){G}})UU, see the chapter 6.4.3., every time in its first modality input subvector the last twenty five components have the value (LD){G} and the environment vector central component is written. And to generate the output sequence DD(E{(LD){G}})UUG every time in the first modality input subvector the last twenty five components have the value (LD){G} and the environment vector central component is blank, where G ≡ DD{W{CNLBE{IF5{FRST5; SCND5}}}}UU. The name W{CNLBE{IF5{FRST5; SCND5}}} is the calling sequence of the general assembly controlling the robot to write the calling sequence CNLBE{IF5{FRST5; SCND5}}.

To train the GABNN to output the sequence implementing the seventh step and to chain it to the eighth, ninth, tenth and to the above given output sequence implementing the eleventh step, it must learn to respond to the calling sequence CCECR{J}, see the chapter 6.4.4.3., with J ≡ (EC)LDD{W{(LD){G}}}UU, where the (EC) implements the present algorithm tenth step and the sequence remaining symbols represent the chaining to the present algorithm eleventh step. The name W{(LD){G}} is the calling sequence of the general assembly controlling the robot to write the calling sequence (LD){G}.

To train the GABNN to output the sequence implementing the sixth step and to chain it to the above given output sequence implementing the seventh step, it has to learn

to generate the output sequence U(CCR)(EC)D(CC1D)(EC)R, see the chapter 6.4.1., every time the environment vector second from the top component has the value CN1DLE{G*} and the environment vector central component is written. And to generate the output sequence U(EC)DL(G*) every time in the first modality input subvector the last twenty five components have the value CN1DLE{G*} and the environment vector central component is blank, where G* ≡ DD{W{(LD){G**}}}UU. To complete the sixth step and to chain the given output sequence with the one implementing the seventh step, it is sufficient to have G** ≡ U{W{CN1DLE{DD{W{CCECR{J}}}UU}}}D. The name W{(LD){G**}} is the calling sequence of the general assembly controlling the robot to write the sequence (LD){G**}, the name W{CN1DLE{DD{W{CCECR{J}}}UU}} is the calling sequence of the general assembly controlling the robot to write the calling sequence CN1DLE{DD{W{CCECR{J}}}UU} and the name W{CCECR{J}} is the calling sequence of the general assembly controlling the robot to write the calling sequence CCECR{J}. The command CN1DLE{G*} is a modified version of the command CN1DLE, see the chapter 6.4.1..

To train the GΑBNN to output the sequence implementing the first step and to chain it to the second, third, fourth, fifth steps and, then, to the above given output sequence implementing the sixth step, it has to learn to generate the output sequence DD(EC)UUUU(W{L{G***}})DD, where G*** ≡ D(MPC)UR(EC)RU(W{CN1DLE{G*}})D, every time its central component content is a closed parenthesis and the bottom component content is the value SS{J*}. Where the above introduced name W{CN1DLE{G*}} is the calling sequence of the general assembly controlling the robot to write the calling sequence CN1DLE{G*}.

To train the GABNN to output the sequence implementing the (sixteenth, if fourth, otherwise third, if third, otherwise fourth) and (sixteenth, if fourth, otherwise third, if third, otherwise fifth) steps and to chain them to the above given output sequence implementing the third, fourth, fifth and sixth steps, it has to learn to generate the output sequence DD(E{(LD){G1}})L(W{(LD){G1}})UU, see the chapter 6.4.3., every time in its first modality input subvector the last twenty five components have the value (LD){G1} and the environment vector central component is written. And to generate the output sequence DD(E{(LD){G1}})UU(G1) every time in the first modality input subvector the last twenty five components have the value (LD){G1} and the environment vector central component is blank, where (G1) ≡ R(EC)RU{W{CN1DLE{G*}}}D. Where the above introduced name W{CN1DLE{G*}} is the calling sequence of the general assembly controlling the robot to write the calling sequence CN1DLE{G*}.

To evaluate the (sixteenth, if fourth, otherwise third, if third, otherwise third) step's conditional shift and to chain the corresponding output sequence to the above given output sequences implementing the (sixteenth, if fourth, otherwise third, if third, otherwise fourth) and (sixteenth, if fourth, otherwise third, if third, otherwise fifth) steps, the GΑBNN has to be trained to output the command sequence corresponding to the first modality input subvector last twenty five component value CNLBE{IF6{FRST6; SCND6}}, ie, a modified version of CNLE, see the chapter 6. 4. 1.. In other words, the GΑBNN has to be trained to output the sequence DD(CCR)(EC)UU(CCL)(EC)R every time CNLBE{IF6{FRST6; SCND6}} is the environment vector last component content and the central component is written. And to output the sequence DD(EC)L(W{IF6{FRST6; SCND6}})UU every time CNLBE{IF6{FRST6; SCND6}} is the environment vector last component content and the central component is blank. Where IF6{FRST6 ≡ DD(EC)(W{(LD){DD(W{CNLBE{IF6{FRST6; SCND6}}})UU}})UU; SCND6 ≡ DD(EC)(W{(LD){G1}}UU) is the calling sequence of two general assemblies: one to control the output sequence we have called FRST6 and the other the output sequence we have called SCND6. The sequence corresponding to FRST6, SCND6, is outputted when the environment vector bottom compo-

nent content is IF6{FRST6; SCND6} and the second from the top component is written, the second from the top component is blank. Moreover, W{CNLBE{IF6{FRST6; SCND6}}} is the calling sequence of the general assembly controlling the robot to write the name CNLBE{IF6{FRST6; SCND6}}.

To train the GABNN to output the sequence implementing the (sixteenth, if fourth, otherwise third, if third, otherwise second) step and to chain it to the above given output sequence implementing the (sixteenth, if fourth, otherwise third, if third, otherwise third) step, it has to learn to generate the output sequence DD(E{(LD){G2}})L(W{(LD){G2}})UU, see the chapter 6.4.3., every time in its first modality input subvector the last twenty five components have the value (LD){G2} and the environment vector central component is written. And to generate the output sequence DD(E{(LD){G2}})UU(G2) every time in the first modality input subvector the last twenty five components have the value (LD){G2} and an environment vector blank central component, where (G2) ≡ R(EC)RU(W{CNLBE{IF6{FRST6; SCND6}}})D. The above introduced name W{CNLBE{IF6{FRST6; SCND6}}} is the calling sequence of the general assembly controlling the robot to write the calling sequence CNLBE{IF6{FRST6; SCND6}}.

To evaluate the (sixteenth, if fourth, otherwise third, if third) step conditional and to chain the corresponding output sequence to the output sequences corresponding to the (sixteenth, if fourth, otherwise third, if third, if first) and to the (sixteenth, if fourth, otherwise third, if third, otherwise first) steps and to the above given output sequence implementing the step (sixteenth, if fourth, otherwise third, if third, otherwise second), the GABNN has to be trained to output the command sequence corresponding to the first modality input subvector last twenty five component value IF7{FRST7; SCND7}. With the output sequence FRST7 ≡ DD(EC)(W{IF3{FRST3; SCND3}})UU corresponding to the environment vector last component content IF7{FRST7; SCND7} and to an environment vector fourth from the top component containing either the digit zero or the digit one, where W{IF3{FRST3; SCND3}} is the calling sequence of the general assembly controlling the robot to write the name IF3{FRST3; SCND3}. And with the output sequence SCND7 ≡ DD(EC)D(W{(LD){G2}})UU corresponding to the environment vector last component content IF7{FRST7; SCND7} and to an environment vector fourth from the top written component with neither the digit zero nor the digit one.

To evaluate the (sixteenth, if fourth, otherwise third) step conditional and to chain the corresponding output sequence to the output sequence corresponding to the (sixteenth, if fourth, otherwise third, if first) and (sixteenth, if fourth, otherwise third, if second) steps and to the above given output sequence implementing the step (sixteenth, if fourth, otherwise third, if third), the GABNN has to be trained to output the command sequence corresponding to the first modality input subvector last twenty five component value IF8{FRST8; SCND8}. With the output sequence FRST8 ≡ DD(EC)U(W{IF3{FRST3; SCND3}})UU corresponding to the environment vector last component content IF8{FRST8; SCND8} and to an environment vector fourth from the top blank component, where W{IF3{FRST3; SCND3}} is the calling sequence of the general assembly controlling the robot to write the name IF3{FRST3; SCND3}. And with the output sequence SCND8 ≡ DD(EC)D(W{IF7{FRST7; SCND7}})UU corresponding to the environment vector last component content IF8{FRST8; SCND8} and to an environment vector fourth from the top written component.

As previously seen, to implement the step (sixteenth, if fourth) there must be a general assembly having as calling sequence, first, the value IF1{FRST1 ≡ D(EC)U(EC)LJ; SCND1}, where J* is the argument of the substitution schema algorithm SS{J*}, assumed by the GABNN first modality input subvector components comprised between the seventy fifth and the one hundredth, the last two components in-

cluded, second, the value corresponding to the environment vector central component state. As the said state can be either written or blank, the corresponding value is either one or zero. Accordingly, whereas FRST1 is the name of the exit sequence D(EC)U(EC)LJ corresponding to at least one written square in the central component, SCND1 is the name of the exit sequence D(EC)LD(W{CNRBE{(RD){G3}}})UU corresponding to all blank squares in the very same component. In turn, the name W{CNRBE{(RD){G3}}} is the calling sequence of the general assembly controlling the robot to write the name CNRBE{(RD){G3}} in the environment vector bottom component, where CNRBE{(RD){G3}} is a modified version of the command CNRBE, see the chapter 6. 4. 1.. That is, every time the environment vector last component content is the value CNRBE{(RD){G3}} and the central component is written, the GABNN has to be trained to assume in the output subvector component the sequence DD(CCL)(EC)UU(CCR)(EC)L. By contrast, every time the environment vector last component content is the value CNRBE{(RD){G3}} and the central component is blank, the GABNN has to be trained to assume in the output subvector component the sequence DD(EC)R(W{(RD){G3}})UU. That is, every time the environment vector last component content is the value (RD){G3} and the central component is written, the GABNN has to be trained to assume in the output subvector the sequence DD(E{(RD){G3}})R(W{(RD){G3}})UU. And every time the environment vector last component content is the value (RD){G3} and the central component is blank the GABNN has to be trained to assume in the output subvector the sequence DD(E{(RD){G3}})L(W{IF9{FRST9; SCND9}})UU, ie, where G3 ≡ DDL(W{IF9{FRST9; SCND9}})UU. In turn, the name W{IF9{FRST9; SCND9}} is the calling sequence of the general assembly trained to control the robot to write the sequence IF9{FRST9; SCND9}. Every time the environment vector bottom component content is the name IF9{FRST9; SCND9} and the sign component is blank, the GABNN has to be trained to assume in the output subvector the sequence FRST9 ≡ DD(EC)(W{CNRBE{(RD){G3}}})UU. Every time the environment vector bottom component content is the name IF9{FRST9; SCND9} and the sign component is written, the GABNN has to be trained to assume in the output subvector the sequence SCND9 ≡ DD(EC)(W{IF8{FRST8; SCND8}})UU.

To train the GABNN to output the sequence implementing the (sixteenth, otherwise first, if third) and (sixteenth, otherwise first, if fourth) steps and to chain them to the above given output sequence implementing the third, fourth, fifth and sixth steps, it has to learn to generate the output sequence DD(E{(LD){G1}})L(W{(LD){G1}})UU, see the chapter 6. 4. 3., every time in its first modality input subvector the last twenty five components have the value (LD){G1} and the environment vector central component is written. And to generate the output sequence DD(E{(LD){G1}})UU(G1) every time in the first modality input subvector the last twenty five components have the value (LD){G1} and the environment vector central component is blank, where (G1) ≡ R(EC)RU{W{CN1DLE{G*}}}D. Where the above introduced name W{CN1DLE{G*}} is the calling sequence of the above given general assembly controlling the robot to write the calling sequence CN1DLE{G*}.

To evaluate the (sixteenth, otherwise first, if second) step's conditional shift and to chain the corresponding output sequence to the above given output sequence implementing the (sixteenth, otherwise first, if third) and (sixteenth, otherwise first, if fourth) steps, the GABNN has to be trained to output the command sequence corresponding to the first modality input subvector last twenty five component value CNLBE{IF10{FRST10; SCND10}}, ie, a modified version of CNLBE, see the chapter 6.4.1.. In other words, the GABNN has to be trained to output the sequence DD(CCR)(EC)UU(CCL)(EC)R every time CNLBE{IF10{FRST10; SCND10}} is the environment vector last component content and the central component is written. And to output the sequence DD(EC)L{W{IF10{FRST10; SCND10}}}UU every time

CNLBE{IF10{FRST10; SCND10}} is the environment vector last component content and the central component is blank. Where IF10{FRST10 ≡ DD(EC){W{(LD){DD(W{CNLBE{IF10{FRST10; SCND10}}})UU}}}UU; SCND10 ≡ DD(EC){W{(LD){DD(W{CNLBE{(LD){G1}}})UU}}}UU}, together with the sign component state, is the calling sequence of two general assemblies: one to control the output sequence we have called FRST10 and the other the output sequence we have called SCND10. The sequence corresponding to FRST10, SCND10, is outputted when the environment vector bottom component content is IF10{FRST10; SCND10} and the fourth from the top component is written, the fourth from the top component is blank. Moreover, W{CNLBE{IF10{FRST10; SCND10}}} is the calling sequence of the general assembly controlling the robot to write the name CNLBE{IF10{FRST10; SCND10}}, W{CNLBE{(LD){G1}}} is the calling sequence of the general assembly controlling the robot to write the command CNLBE{(LD){G1}} and CNLBE{(LD){G1}} is a modified version of CNLBE. In other words, the GABNN has to be trained to output the sequence DD(CCR)(EC)UU(CCL)(EC)R every time CNLBE{(LD){G1}} is the environment vector last component content and the central component is written. And to output the sequence DD(EC)L(W{(LD){G1}})UU every time CNLBE{(LD){G1}} is the environment vector last component content and the central component is blank. Where the name W{(LD){G1}} is the calling sequence of the general assembly controlling the robot to write the previously introduced sequence (LD){G1}.

To train the GABNN to output the sequence implementing the (sixteenth, otherwise first, if first) step and to chain them to the above given output sequence implementing the (sixteenth, otherwise first, if second) step, it has to learn to generate the output sequence DD(EC)L(W{(LD){CNLBE{IF10{FRST10; SCND10}}}})UU, see the chapter 6. 4. 3., every time in its first modality input subvector the last twenty five components have the value (LD){CNLBE{IF10{FRST10; SCND10}}} and the environment vector central component is written. And to generate the output sequence DD(E{(LD){CNLBE{IF10{FRST10; SCND10}}}})UU(CNLBE{IF10{FRST10; SCND10}}) every time in the first modality input subvector the last twenty five components have the value (LD){CNLBE{IF10{FRST10; SCND10}}} and the environment vector central component is blank.

To evaluate the (sixteenth, otherwise first) step conditional and to chain the corresponding output sequence to the above given output sequence corresponding to the (sixteenth, otherwise first, if first) step, the GABNN has to be trained to output the command sequence corresponding to the first modality input subvector last twenty five component value IF11{FRST11; SCND11}. With the output sequence FRST11 ≡ DD(EC)(W{(LD){CNLBE{IF10{FRST10; SCND10}}}})UU corresponding to the environment vector last component content IF11{FRST11; SCND11} and to the environment vector central component content being a symbol different from a digit, where W{(LD){CNLBE{IF10{FRST10; SCND10}}}} is the calling sequence of the general assembly controlling the robot to write the name (LD){CNLBE{IF10{FRST10; SCND10}}}. And the output sequence SCND11, we shall give later on, corresponding both to the environment vector last component content IF11{FRST11; SCND11} and to the environment vector central component content being a digit and the fourth from the top component containing a symbol different from a digit.

To train the GABNN to evaluate the sixteenth step's conditional and to chain the corresponding output sequence to the above given output sequences implementing the step (sixteenth, if first), we have previously seen the GABNN first modality input subvector last twenty five components have to assume the value IF3{FRST3 ≡ DD(EC){W{IF2{FRST2}}}UU; SCND3} and both the environment vector central

and fourth from the top components contain either a digit each, or one a digit and the other a blank. Accordingly, whereas FRST3 ≡ DD(EC){W{IF2{FRST2}}}UU corresponding to the case of one digit and a blank or two digits in the third and in the fourth from the top component, SCND3 ≡ DD(EC)(W{IF11{FRST11; SCND11}})UU is the name of the exit sequence corresponding to the presence of symbols different from digits in both components. The name W{IF2{FRST2}} is the calling sequence of the general assembly trained to control the robot to write the value IF2{FRST2} in the environment vector bottom component. The name W{IF11{FRST11; SCND11}} is the calling sequence of the general assembly trained to control the robot to write the value IF11{FRST11; SCND11} in the environment vector bottom component.

To train the GABNN to output the sequence implementing the (sixteenth, otherwise first, otherwise fourth) and (sixteenth, otherwise first, otherwise fifth) steps and to chain them to the above given output sequence implementing the third, fourth, fifth and sixth steps, it has to learn to generate the output sequence DD(E{(LD){G1}})L(W{(LD){G1}})UU, see the chapter 6.4.3., every time in its first modality input subvector the last twenty five components have the value (LD){G1} and the environment vector central component is written. And to generate the output sequence DD(E{(LD){G1}})UU(G1) every time in the first modality input subvector the last twenty five components have the value (LD){G1} and the environment vector central component is blank, where (G1) ≡ R(EC)RU{W{CN1DLE{G*}}}D. Where the above introduced name W{CN1DLE{G*}} is the calling sequence of the above given general assembly controlling the robot to write the calling sequence CN1DLE{G*}.

To evaluate the (sixteenth, otherwise first, otherwise third) step's conditional shift and to chain the corresponding output sequence to the above given output sequence implementing the (sixteenth, otherwise first, otherwise fourth) and (sixteenth, otherwise first, otherwise fifth) steps, the GABNN has to be trained to output the command sequence corresponding to the first modality input subvector last twenty five component value CNLBE{IF12{FRST12; SCND12}}, ie, a modified version of CNLBE, see the chapter 6. 4. 1.. In other words, the GABNN has to be trained to output the sequence DD(CCR)(EC)UU(CCL)(EC)R every time CNLBE{IF12{FRST12; SCND12}} is the environment vector last component content and the central component is written. And to output the sequence DD(EC)L{W{IF12{FRST12; SCND12}}}UU every time CNLBE{IF12{FRST12; SCND12}} is the environment vector last component content and the central component is blank. Where IF12{FRST12; SCND12}, together with the sign component state, is the calling sequence of two general assemblies: one to control the output subvector sequence FRST12 ≡ DD(EC){W{(LD){DD(W{CNLBE{IF12{FRST12; SCND12}}})UU}}}UU and the other the output subvector sequence we have called SCND12 ≡ DD(EC){W{(LD){DD(W{CNLBE{(LD){G1}}})UU}}}UU. The sequence corresponding to FRST12, SCND12, is outputted when the environment vector bottom component content is IF12{FRST12; SCND12} and the second from the top component is written, the second from the top component is blank. Moreover, W{CNLBE{IF12{FRST12; SCND12}}} is the calling sequence of the general assembly controlling the robot to write the name CNLBE{IF12{FRST12; SCND12}}, W{CNLBE{(LD){G1}}} is the calling sequence of the general assembly controlling the robot to write the command CNLBE{(LD){G1}} and CNLBE{(LD){G1}} is a modified version of CNLBE. In other words, the GABNN has to be trained to output the sequence DD(CCR)(EC)UU(CCL)(EC)R every time CNLBE{(LD){G1}} is the environment vector last component content and the central component is written. And to output the sequence DD(EC)L(W{(LD){G1}})UU every time CNLBE{(LD){G1}} is the environment vector last component content and the central component is blank. Where the name W{(LD){G1}} is the calling sequence of the general assembly controlling the robot to write the previously introduced sequence (LD){G1}.

To train the GABNN to output the sequence implementing the (sixteenth, otherwise first, otherwise second) step and to chain them to the above given output sequence implementing the (sixteenth, otherwise first, otherwise third) step, it has to learn to generate the output subvector sequence DD(EC)L(W{(LD){CNLBE{IF12{FRST12; SCND12}}}})UU, see the chapter 6. 4. 3., every time in its first modality input subvector the last twenty five components have the value (LD){CNLBE{IF12{FRST12; SCND12}}} and the environment vector central component is written. And it has to learn to generate the output subvector sequence DD(E{(LD){CNLBE{IF12{FRST12; SCND12}}}})UU(CNLBE{IF12{FRST12; SCND12}}) every time in the first modality input subvector the last twenty five components have the value (LD){CNLBE{IF12{FRST12; SCND12}}} and the environment vector central component is blank.

To evaluate the (sixteenth, otherwise first) step conditional and to chain the corresponding output sequence to the output subvector sequence corresponding to the (sixteenth, otherwise first, otherwise first) and (sixteenth, otherwise first, otherwise second) steps, we have seen the GABNN has to be trained to output the command sequence corresponding to the first modality input subvector last twenty five component value IF11{FRST11; SCND11}. As we have previously seen, the output subvector sequence FRST11 ≡ DD(EC)(W{(LD){CNLBE{IF10{FRST10; SCND10}}}})UU corresponding to the environment vector last component content IF11{FRST11; SCND11} and to the environment vector central component content being a symbol different from digit, where W{(LD){CNLBE{IF10{FRST10; SCND10}}}} is the calling sequence of the general assembly controlling the robot to write the name (LD){CNLBE{IF10{FRST10; SCND10}}}. Similarly, the output sequence SCND11 ≡ DD(EC)(W{(LD){CNLBE{IF12{FRST12; SCND12}}}}) corresponds both to the environment vector last component content IF11{FRST11; SCND11} and to the environment vector central component content being a digit and the fourth from the top component containing a symbol different from a digit.

6.8. Training the GABNN to compute the minimalization schema.

The GABNN can be trained to compute the minimalization schema

$$\mu y[g[f[x_1, x_2], y] = 0]$$

if both it has been trained to compute $g[f[x_1, x_2], y]$ and to every f, x_1, x_2 there exists an y such that

$$g[f[x_1, x_2], y] = 0.$$

Let us consider the three general assembly calling sequences J1, J2, J3, the GABNN has been previously trained on. We call $\Im${J1}, $\Re${J2}, the environment vector sign component content, for the GABNN to evaluate $f[x_1, x_2]$, $g[z_1, z_2]$, whenever the central and fourth from the top component contents are the first and the second operand rightmost digits. That is, once $\Im${J1}, $\Re${J2}, has been computed and the operation symbol together with the two operands have been erased, the sign component content is the result rightmost digit and the GABNN is ready to execute J1, or J2. Accordingly, to find the minimum y such that $g[f[x_1, x_2], y]$ is zero, first, the GABNN evaluates the substitution of f within g, ie, it computes the function $g[f[x_1, x_2], y]$, for y = 0, 1, 2, ..., copying the y value one environment vector on top of the operation symbol, second, once found the desired y, the GABNN erases everything there is at the left of the y value, third, the GABNN ends its journey with the central component on y rightmost digit.

To increment the y value, we need a modified version of the addition algorithm, namely one with the environment vector sign component contents A{J3}. That is, we consider the GABNN previously trained to compute the sum of the operand every time the sign component content is A{J3} and the central and fourth from the top component contents are digits. Once the addition has been executed, the result is copied in the first operand component row without lateral shifts, the A{J3} name and the two operands erased and the central component is positioned so to execute the task J3. For an example of the addition algorithm modified version see the chapter 6. 4. 4. 2. and the Table 6. 3..

It is convenient to train the GABNN to generate the output subvector sequence corresponding both to the environment vector bottom component content μy{J4} and to the central component content being the rightmost parenthesis of the expression g[[x_1, x_2], y]. That is, initially the name y is written as the g function second argument. Accordingly, the GABNN has to be trained:

first, to go two components down;
second, to erase the component content, ie, to erase the name μy{J4};
third, to go two components up;
fourth, to copy, and erase, one vector up, from the right to the left, the written component row;
fifth, to go one vector up and to write a zero;
sixth, to go at the left end of the present written component row, ie, to go on y current value leftmost digit;
seventh, to go two vectors less one component down;
eight, to mark the component;
ninth, to go two vectors less one component up;
tenth, to go at the right end of the present written component row, ie, to go on y current value rightmost digit;
eleventh, to go one vector down;
twelfth, to copy, without erasing, the written component row one vector down, starting from the rightmost written component;
thirteenth, to go one vector down;
fourteenth, to find and mark the central comma, ie, to write in the bottom component CCBM{J};
fifteenth, to erase in the rightward direction the written components comprised between the one containing central comma and the one containing the rightmost parenthesis;

"**Comment**. The central component holds the expression central comma."

sixteenth, to go at the expression left end and to shift leftward the written component row, from the component containing the central comma to the leftmost symbol, initially the g symbol, until the central comma is two components at the left of the second from the top marked component, to erase the mark from the component holding the central comma;
seventeenth, to go two components at the right, to erase the mark from the second from the top component, to go two vectors up;
eighteenth, starting from the leftmost component, to copy, without erasing, y current value two vectors below, in the written component row holding the expression of the function g, in place of the numeral erased in the fifteenth step;

"**Comment**. The central component holds the expression rightmost sign, a closed parenthesis."

nineteenth, to go two components down;

twentieth, to write the name SS{J*};
twenty-first, to go two components up;
twenty-second, to evaluate the substitution schema and to chain the present output sequence to the next one, ie, to execute J*;

"**Comment**. The substitution schema has been computed. The operands and the operation symbol have been erased. The central component contains the result rightmost digit. If the result is zero, ie, y desired value has been found, the present written component row has to be erased, the written component row one vector up has to be erased, the y value one component up has to be copied, and erased, two vectors down and the algorithm terminates with the central component containing y last computed value rightmost digit. Otherwise, ie, if the substitution schema result is not zero, the present written component row has to be erased, a one has to be added to the expression contained in the written component row two vectors up, accordingly, the present central component contains the y current value rightmost digit, the algorithm goes back to the sixth step."

twenty-third, to find out whether or not the present numeral is zero, to go leftward until either a blank or the digit one is found;
twenty-fourth,
if the numeral is zero, ie, because a component written with the digit one has not been found and, instead, a blank component has been found, the y value has been computed
(twenty-fourth, if first) to go at the left end of the written component row;
(twenty-fourth, if second) to erase in a rightward direction the written component row;
(twenty-fourth, if third) to go one vector up;
(twenty-fourth, if fourth) to go at the left end of the written component row;
(twenty-fourth, if fifth) to erase in a rightward direction the written component row;
(twenty-fourth, if sixth) to go one vector up;
(twenty-fourth, if seventh) to go at the left end of the written component row;
(twenty-fourth, if eighth) to copy, and erase, two vectors down, in the left to right direction, the written component row;
(twenty-fourth, if ninth) to execute the command J4. "**Algorithm's ends**."
otherwise, ie, if the numeral is not zero, ie, because a component containing the digit one has been found, the y value has not been computed, yet
(twenty-fourth, otherwise first) to go at the left end of the written component row;
(twenty-fourth, otherwise second) to erase from left to right the written component row;
(twenty-fourth, otherwise third) to go two vectors less one component up;
(twenty-fourth, otherwise fourth) to write the digit one;
(twenty-fourth, otherwise fifth) to go three components up;
(twenty-fourth, otherwise sixth) to write the name A{J3};
(twenty-fourth, otherwise seventh) to go two components down;

(twenty-fourth, otherwise eighth) to execute the addition and to chain the present output sequence to the next output sequence, ie, to execute J3;
(twenty-fourth, otherwise ninth) to go back at the algorithm sixth step;

To avoid to change the procedures as we present them, we have to start from the algorithm last step, ie, from the step (twenty-fourth, if ninth) and to go backward.

To perform the algorithm last two steps, (twenty-fourth, if eighth) and (twenty-fourth, if ninth), the GABNN output subvector component content has to correspond to the name (CN2VDLE){J4}, ie, to CN2VLDE modified version, see the chapter 6. 4. 1.. That is, every time the first modality input subvector last components assume the value CN2VDLE{J4} and among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero, the GABNN has to be trained to generate in its output subvector the sequence DD(CCR)(EC)UU(CC2VD)(EC)R. And every time the first modality input subvector last components assume the value CN2VDLE{J4} and all its input subvector components comprised between the fifty first and the seventy fifth, the last two included, have the value zero, the GABNN has to be trained to generate in its output subvector the sequence DD(EC)UUL(J4).

To train the GABNN to output the sequence implementing the (twenty-fourth, if seventh) step and to chain it to the above given output sequence implementing the (twenty-fourth, if eighth) and (twenty-fourth, if ninth) steps, it has to learn to generate the output sequence DD(E{(LD){NG1}})L(W{(LD){NG1}})UU, see the chapter 6. 4. 3., every time in its first modality input subvector the last twenty five components have the value (LD){NG1} and the environment vector central component is written. And to generate the output sequence DD(E{(LD){NG1}})UU(NG1) every time in the first modality input subvector the last twenty five components have the value (LD){NG1} and the environment vector central component is blank, where (NG1) ≡ RDD(W{(CN2VDLE){J4}})RUU. Where the name W{(CN2VDLE){J4}} is the calling sequence of the general assembly controlling the robot to write the calling sequence (CN2VDLE){J4}.

To train the GABNN to output the sequence implementing the (twenty-fourth, if fifth) and (twenty-fourth, if sixth) steps and to chain them to the above given output sequence implementing the (twenty-fourth, if seventh) step, it has to learn to generate the output sequence (EC)DD(EC)R(W{(ECR){NG2}})UU, every time in its first modality input subvector the last twenty five components have the value (ECR){NG2} and the environment vector central component is written. And to generate the output sequence DD(E)LUU(NG2) every time in the first modality input subvector the last twenty five components have the value (ECR){NG2} and the environment vector central component is blank, where (NG2) ≡ υDD(W{(LD){NG1}}UU. Where the name W{(LD){NG2}} is the calling sequence of the general assembly controlling the robot to write the calling sequence (LD){NG2} and υ is the command to move the central component one vector up, see the chapter 6. 2.3.1.2..

To train the GABNN to output the sequence implementing the (twenty-fourth, if fourth) step and to chain it to the above given output sequence implementing the (twenty-fourth, if fifth) and (twenty-fourth, if sixth) steps, it has to learn to generate the output sequence DD(E{(LD){NG3}})L(W{(LD){NG3}})UU, see the chapter 6. 4. 3., every time in its first modality input subvector the last twenty five components have the value (LD){NG3} and the environment vector central component is written. And to generate the output sequence DD(E{(LD){NG3}})UU(NG3) every time in the first modality input subvector the last twenty five components have the value (LD){NG3} and the environment vector central component is blank, where (NG3) ≡ RDD(W{(ECR){NG2}})UU. Where the name W{(ECR){NG2}} is the calling se-

quence of the general assembly controlling the robot to write the calling sequence (ECR){NG2}.

To train the GABNN to output the sequence implementing the (twenty-fourth, if second) and (twenty-fourth, if third) steps and to chain them to the above given output sequence implementing the (twenty-fourth, if fourth) step, it has to learn to generate the output sequence (EC)DD(EC)R(W{(ECR){NG4}})UU, every time in its first modality input subvector the last twenty five components have the value (ECR){NG4} and the environment vector central component is written. And to generate the output sequence DD(E)LUU(NG4) every time in the first modality input subvector the last twenty five components have the value (ECR){NG4} and the environment vector central component is blank, where (NG4) ≡ υDD(W{(LD){NG3}}UU. Where the name W{(LD){NG4}} is the calling sequence of the general assembly controlling the robot to write the calling sequence (LD){NG4} and υ is the command to move the central component one vector up, see the chapter 6. 2.3.1.2..

To train the GABNN to output the sequence implementing the (twenty-fourth, if first) step and to chain it to the above given output sequence implementing the (twenty-fourth, if second) and (twenty-fourth, if third) steps, it has to learn to generate the output sequence DD(E{(LD){NG5}})L(W{(LD){NG5}})UU, see the chapter 6.4.3., every time in its first modality input subvector the last twenty five components have the value (LD){NG5} and the environment vector central component is written. And to generate the output sequence DD(E{(LD){NG5}})UU(NG5) every time in the first modality input subvector the last twenty five components have the value (LD){NG5} and the environment vector central component is blank, where (NG5) ≡ RDD(W{(ECR){NG4}})UU. Where the name W{(ECR){NG4}} is the calling sequence of the general assembly controlling the robot to write the calling sequence (ECR){NG4}.

To evaluate the twenty-third and the twenty-fourth conditional shifts and to chain the corresponding output sequence to the above given output sequence implementing the (twenty-fourth, if first) step and to the next to be given (twenty-fourth, otherwise first) step, the GABNN has to be trained to output the command sequence corresponding to the first modality input subvector last twenty five component value IF1{FRST1; SCND1}. Actually, the output sequence FRST1 ≡ DD(EC)L(W{IF1{FRST1; SCND1}})UU corresponds to the environment vector last component containing the name IF1{FRST1; SCND1} and to the central component containing the digit zero. Instead, the output sequence SCND1 ≡ DD(EC)R(W{IF2{FRST2; SCND2}})UU corresponds to the environment vector last component containing the name IF1{FRST1; SCND1} and to the central component either being blank or containing the digit one. In turn, every time the environment vector last component contains the name IF2{FRST2; SCND2} and the central component is blank, the GABNN has to be trained to generate the output subvector sequence FRST2 ≡ DD(EC)L(W{(LD){NG5}})UU. And every time the environment vector last component contains the name IF2{FRST2; SCND2} and the central component contains the digit one, the GABNN has to be trained to generate the output subvector sequence SCND2, we shall present later on.

To train the GABNN to output the sequence implementing the eighteenth, nineteenth, twentieth, twenty-first and twenty-second steps and to chain them to the above given output sequences implementing the (twenty-fourth, if first), (twenty-fourth, if second) and (twenty-fourth, if third) steps, it has to learn to generate the output sequence corresponding to the calling sequence CN2VDL{J5}, ie, to CN2VDL modified version, see the chapter 6.4.1.. That is, every time the first modality input subvector last components assume the value CN2VDL{J5} and among its input subvector components comprised between the fifty first and the seventy fifth, the last two included, there is at least one different from zero, the GABNN has to be trained to generate in its output subvector

the sequence DD(CCR)(EC)UU(CC2VD)R. And every time the first modality input subvector last components assume the value CN2VDLE{J5} and all its input subvector components comprised between the fifty first and the seventy fifth, the last two included, have the value zero, the GABNN has to be trained to generate in its output subvector the sequence DD(EC)UUL(J5). Where the name (J5) ≡ DD(W{SS{J* ≡ DD(W{IF1{FRST1; SCND1}})UU}})UU is the calling sequence of the previously trained general assembly controlling the execution of the sequence (J5).

To train the GABNN to output the sequence implementing the sixteenth step and to chain it to the output sequence corresponding to the seventeenth step and to the above given output sequence implementing the eighteenth step, it has to learn to generate the output sequence DD(E{(LD){NG5}})L(W{(LD){NG5}})UU, see the chapter 6. 4. 3., every time in its first modality input subvector the last twenty five components have the value (LD){NG5} and the environment vector central component is written. And to generate the output sequence DD(E{(LD){NG5}})UU(NG5) every time in the first modality input subvector the last twenty five components have the value (LD){NG5} and the environment vector central component is blank, where (NG5) ≡ RDD(W{(CNLBE){J6}})UU. The name W{(CNLBE){J6}} is the calling sequence of the general assembly previously trained to control the robot to write the name (CNLBE){J6}, a modified version of the command CNLBE, see the chapter 6. 4. 1.. That is, every time the environment vector second from the top component content is (CNLBE){J6} and the central component is written, the GABNN has been previously trained to assume in the output subvector the sequence DD(CCR)(EC)UU(CCL)(EC)R. And every time the environment vector second from the top component content is (CNLBE){J6} and the central component is written, the GABNN has been previously trained to assume in the output subvector the sequence DD(EC)UUL(J6). Where J6 ≡ DD(W{IF3{FRST3; SCND3}})UU and W{IF3{FRST3; SCND3}} is the calling sequence of the general assembly previously trained to control the robot to write the sequence IF3{FRST3; SCND3}, where FRST3 ≡ DD(EC)(W{(LD){NG6}})UU if the environment vector last component content is IF3{FRST3; SCND3} and the second from the top component content is not marked. And to write the sequence SCND3 ≡ DD(EC)(W{(LD){NG7}})UU if the environment vector last component content is IF3{FRST3; SCND3} and the second from the top component content is marked. In turn, W{(LD){NG6}} is the calling sequence of the general assembly previously trained to generate the output sequence DD(E{(LD){NG6}})L(W{(LD){NG6}})UU, see the chapter 6.4.3., every time in its first modality input subvector the last twenty five components have the value (LD){NG6} and the environment vector central component is written. And to generate the output sequence DD(E{(LD){NG6}})UU(NG6) every time in the first modality input subvector the last twenty five components have the value (LD){NG6} and the environment vector central component is blank, where (NG6) ≡ RDD(W{(CNLBE){J6}})UU. Instead, W{(LD){NG7}} is the calling sequence of the general assembly previously trained to generate the output sequence DD(E{(LD){NG7}})L(W{(LD){NG7}})UU, see the chapter 6. 4. 3., every time in its first modality input subvector the last twenty five components have the value (LD){NG7} and the environment vector central component is written. And to generate the output sequence DD(E{(LD){NG7}})UU(NG7) every time in the first modality input subvector the last twenty five components have the value (LD){NG7} and the environment vector central component is blank, where (NG7) ≡ RDD(W{(CNLBE){J7}})UU. The name W{(CNLBE){J7}} is the calling sequence of the general assembly previously trained to control the robot to write the name (CNLBE){J7}, a modified version of the command CNLBE, see the chapter 6.4.1.. That is, every time the environment vector second from the top component content is (CNLBE){J7} and the central component is written, the GABNN has been previously

trained to assume in the output subvector the sequence DD(CCR)(EC)UU(CCL)(EC)R. And every time the environment vector second from the top component content is (CNLBE){J7} and the central component is written, the GABNN has been previously trained to assume in the output subvector the sequence DD(EC)UUL(J7). Where J7 ≡ DD(W{(LD){NG8}})UU and W{(LD){NG8}} is the calling sequence of the general assembly previously trained to generate the output sequence DD(E{(LD){NG8}})L(W{(LD){NG8}})UU, see the chapter 6.4.3., every time in its first modality input subvector the last twenty five components have the value (LD){NG8} and the environment vector central component is written. And to generate the output sequence DD(E{(LD){NG8}})UU(NG8) every time in the first modality input subvector the last twenty five components have the value (LD){NG8} and the environment vector central component is blank, where (NG8) ≡ RDD(W{(CNLBE){J8}})UU. The name W{(CNLBE){J8}} is the calling sequence of the general assembly previously trained to control the robot to write the name (CNLBE){J8}, a modified version of the command CNLBE, see the chapter 6.4.1.. That is, every time the environment vector second from the top component content is (CNLBE){J8} and the central component is written, the GABNN has been previously trained to assume in the output subvector the sequence DD(CCR)(EC)UU(CCL)(EC)R. And every time the environment vector second from the top component content is (CNLBE){J8} and the central component is written, the GABNN has been previously trained to assume in the output subvector the sequence DD(EC)UUL(J8). Where J8 ≡ (EPC)RR(ESC)υυDD(W{CN2VDL{J5}})UU. Where the name W{CN2VDL{J5}} is the calling sequence of the general assembly controlling the robot to write the calling sequence CN2VDL{J5}.

To train the GABNN to output the sequence implementing the fifteenth step and to chain it to the above given output sequence implementing the sixteenth step, it has to learn to generate the output sequence (EC)DD(EC)R(W{ELB{J9}})UU every time the environment vector central component is written and the bottom component content is the name ELB{J9}, ie, a modified version of the name ELB, see the chapter 6.4.2.1.. And the output sequence DD(EC)UU(J9) every time the environment vector central component is a closed parenthesis followed by a blank and the bottom component content is the name ELB{J9}. Where W{ELB{J9}} is the calling sequence of the general assembly controlling the robot to write the name ELB{J9}, the output sequence (J9) ≡ DDL(W{(LNM){NG9}})UU, the output sequence (NG9) ≡ DDL(W{(LD){NG10}})UU and the output sequence (NG10) ≡ DDL(W{(LD){NG5}})UU chaining the present to the previous output sequence.

To train the GABNN to output the sequence implementing the fourteenth step and to chain it to the above given output sequence implementing the fifteenth step, it has to learn to search and mark the central comma CCBM{J10}, with (J10) ≡ RRDD(W{ELB{J9}})UU. Where the name W{(ELB){J19}} is the calling sequence of the general assembly controlling the robot to write the calling sequence ELB{J19}.

To train the GABNN to output the sequence implementing the twelfth step and to chain it to the output sequence corresponding to the thirteenth step and to the above given output sequence implementing the fourteenth step, it has to learn to generate the output sequence DD(CCL)(EC)UU(CC1VD)L every time the environment vector central component is written and the bottom component content is the name (CN1VDRB){J11}, ie, a modified version of the name CN1VDRB, see the chapter 6.4.1.. And the output sequence DD(EC)UU(J11) every time the environment vector central component is written and the bottom component content is the name (CN1VDRB){J11}. Where the output sequence (J11) ≡ δDD(W{CCBM{J10}})UU and W{CCBM{J10}} is the calling sequence of the general assembly controlling the robot to write the output sequence CCBM{J10}.

To train the GABNN to output the sequence implementing the tenth step and to chain it to the output sequence implementing the eleventh step and to the above given output sequence implementing the twelfth step, it has to learn to generate the output sequence DD(E{(RD){NG11}})R(W{(RD){NG11}})UU, see the chapter 6.4.3., every time in its first modality input subvector the last twenty five components have the value (RD){NG11} and the environment vector central component is written. And to generate the output sequence DD(E{(RD){NG11}})UU(NG11) every time in the first modality input subvector the last twenty five components have the value (RD){NG11} and the environment vector central component is blank, where (NG11) ≡ LδDD(W{(CN1VDRB){J11}})UU. Where W{(CN1VDRB){J11}} is the calling sequence of the general assembly controlling the robot to write the name CN1VDRB{J11}.

To train the GABNN to output the sequence implementing the sixth step and to chain it to the output sequence implementing the seventh, eight and ninth steps and to the above given output sequence implementing the tenth, eleventh and twelfth steps, it has to learn to generate the output sequence DD(E{(LD){NG12}})L(W{(LD){NG12}})UU, see the chapter 6.4.3., every time in its first modality input subvector the last twenty five components have the value (LD){NG12} and the environment vector central component is written. And to generate the output sequence DD(E{(LD){NG12}})UU(NG12) every time in the first modality input subvector the last twenty five components have the value (LD){NG12} and the environment vector central component is blank, where (NG12) ≡ RδδU(MPC)υυDDD(W{((RD){NG11}})UU. Where the name W{(RD){NG11}} is the calling sequence of the general assembly controlling the robot to write the calling sequence (RD){NG11}.

To train the GABNN to output the sequence implementing the fourth step and to chain it to the output sequence corresponding to the fifth step and to the above given output sequence implementing the sixth step, it has to learn to generate the output sequence DD(CCL)(EC)UU(CC1VD)(EC)L every time the environment vector central component is written and the bottom component content is the name (CN1VURE){J13}, ie, a modified version of the name CN1VURE, see the chapter 6. 4. 1.. And the output sequence DD(EC)UU(J13) every time the environment vector central component is written and the bottom component content is the name (CN1VURE){J13}. Where the output sequence (J13) ≡ υ(W{0})DD(W{(LD){NG12}})UU. Where the name W{(LD){NG12}} is the calling sequence of the general assembly controlling the robot to write the calling sequence (LD){NG12}.

To train the GABNN to output the sequence implementing the first step and to chain it to the output sequence corresponding to the second and third steps and to the above given output sequence implementing the fourth step, it has to learn to generate the output sequence DD(EC)(W{(CN1VURE){J13}})UU every time the environment vector central component content is a closed parenthesis and the bottom component content is the name μy{J4}. Where the name W{(CN1VURE){J13}} is the calling sequence of the general assembly controlling the robot to write the calling sequence CN1VURE{J13}.

Once the y value has been increased by one, ie, once the addition has been executed, the result is copied in the first operand component row without lateral shifts, the A{J3} name and the two operands erased and the central component content is y rightmost digit. Accordingly, to train the GABNN to output the sequence implementing the (twenty-fourth, otherwise eighth) step and to chain it to the output sequence corresponding to the (twenty-fourth, otherwise ninth) step and to the above given output sequence implementing the sixth step, it is sufficient to take J3 ≡ DD(W{(LD){NG12}})UU. Where the name W{(LD){NG12}} is the calling sequence of the general assembly controlling the robot to write the calling sequence (LD){NG12}.

To train the GABNN to output the sequence implementing the (twenty-fourth, otherwise second) step and to chain it to the (twenty-fourth, otherwise third), (twenty-

fourth, otherwise fourth), (twenty-fourth, otherwise fifth), (twenty-fourth, otherwise sixth) and (twenty-fourth, otherwise seventh) steps and to the above given output sequence implementing the (twenty-fourth, otherwise eighth) step, it has to learn to generate the output sequence (EC)DD(EC)R(W{ELB{J14}})UU every time the environment vector central component is written and the bottom component content is the name ELB{J14}, ie, a modified version of the name ELB, see the chapter 6.4.2.1.. And the output sequence DD(EC)UU(J14) every time the environment vector central component is a closed parenthesis followed by a blank and the bottom component content is the name ELB{J14}. Where W{ELB{J14}} is the calling sequence of the general assembly controlling the robot to write the name ELB{J14}, (J14) ≡ υυD(W{1})UUU(W{A{J3}})DD and (J3) ≡ DDL(W{(LD){NG12}})UU.

To train the GABNN to output the sequence implementing the (twenty-fourth, otherwise first) step and to chain it to the above given output sequence implementing the (twenty-fourth, otherwise second) step, it has to learn to generate the output sequence DD(E{(LD){NG13}})L(W{(LD){NG13}})UU, see the chapter 6.4.3., every time in its first modality input subvector the last twenty five components have the value (LD){NG13} and the environment vector central component is written. And to generate the output sequence DD(E{(LD){NG13}})UUR(NG13) every time in the first modality input subvector the last twenty five components have the value (LD){NG13} and the environment vector central component is blank, where (NG13) ≡ DD(W{ELB{J14}})UU. Where W{ELB{J14}} is the calling sequence of the general assembly controlling the robot to write the name ELB{J14}.

As we have seen, to evaluate the twenty-third and the twenty-fourth conditional shifts and to chain the corresponding output sequence to the above given output sequence implementing the (twenty-fourth, if first) step and to the next to be given (twenty-fourth, otherwise first) step, the GABNN has to be trained to output the command sequence corresponding to the first modality input subvector last twenty five component value IF1{FRST1; SCND1}. Actually, the output sequence FRST1 ≡ DD(EC)L(W{IF1{FRST1; SCND1}})UU corresponds to the environment vector last component containing the name IF1{FRST1; SCND1} and to the central component containing the digit zero. Instead, the output sequence SCND1 ≡ DD(EC)R(W{IF2{FRST2; SCND2}})UU corresponds to the environment vector last component containing the name IF1{FRST1; SCND1} and to the central component either being blank or containing the digit one. In turn, every time the environment vector last component contains the name IF2{FRST2; SCND2} and the central component is blank, the GABNN has to be trained to generate the output subvector sequence FRST2 ≡ DD(EC)L(W{(LD){NG5}})UU. And every time the environment vector last component contains the name IF2{FRST2; SCND2} and the central component contains the digit one, the GABNN has to be trained to generate the output subvector sequence SCND2 ≡ DD(EC)L(W{(LD){NG13}})UU. Where W{(LD){NG13}}, W{(LD){NG5}}, is the calling sequence of the general assembly controlling the robot to write the name (LD){NG13}, (LD){NG5}.

Chapter 7

THE JAVA PACKAGE IT.NA.CY.NNET.

7.1. Introduction.

To present the Java package *it.na.cy.nnet* for the development of neural network simulators, we specify the package syntax and semantic following the lexical and syntactic structures of Java, as defined in [23, 24]. Our package is suitable for the simulation of networks with the features:

A1) The network is subdivided in three subvectors or layers, see the Figure 7. 1.: the input subvector or sensor node layer, the hidden subvector or hidden node layer and the output subvector or motor node layer. Accordingly, each node has one among the following three positions: input or sensor, hidden and output or motor.

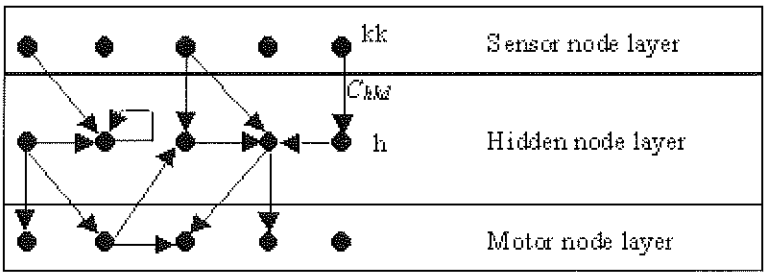


Fig. 7.1. *The network model.* The arrows represent the network connections. The filled circles represent the network nodes. The network is subdivided in three subvectors or layers: the input subvector or sensor node layer, the hidden subvector or hidden node layer and the output subvector or motor node layer. The network has the features defined in A1, A2, ..., A6, see text.

A2) To each network node it is possible to associate the tag *property*, which can assume one among five values: property - 0, 1, 2, 3, 4.

A3) The time t is a discrete variable, ie, t belongs to the positive integer numbers.

A4) To each network node is associated, by means of a one-to-one correspondence, an integer number called the node *index*. The network connections are d-delayed monodirectional channels going from a node with index k to a node with index h. Each d-delayed monodirectional channel is univocally associated with the coupling coefficient C_{hkd}, see the Figure 7.1. Each C_{hkd} can be either time independent or time dependent.

A5) The connections go:

A5.1) from the sensor layer to either the hidden or the motor layer;

A5.2) from the hidden layer to either the hidden or the motor layer;

A5.3) from the motor layer to either the hidden or the motor layer;

A6) Let N be the number of network nodes and let D be the maximum delay of the monodirectional channels, then:

A6.1) At any time both each node accepts an input and each node computes its own state and its own output. The input of the node h at the time t depends on both C_{hkd} and $f_h(t\text{-}d)$, with h = 1, 2,..., N and d = 1, 2,..., D.

A6.2) $f_h(t\text{-}d)$ is the output of the node h at the time t-d, $f_h(t\text{-}d) \in [0,a]$ with $a \in R^+$. The node h is said to be active, or on, at any t if and only if $f_h(t) \neq 0$, otherwise it is said inactive, or off. The node output depends both on its own state and on its own input.

In the chapter 7.3. we evaluate both the average run-time for a simulator implemented by means of the it.na.cy.nnet package and the conditions for a "good performance" of the simulator. In order to describe a neural network having the features A1-A6, we discuss in the chapter 7.4. the Low level Language for Neural Networks, or LLNN. Eventually, in the chapter 7.5. we present the results of some tests on the simulation of a GABNN, see the chapter 5..

The Java classes *NeuralNetSim* and *BoolHebbNet* represent the it.na.cy.nnet package core. The BoolHebbNet is a subclass of NeuralNetSim. An instance of a suitable NeuralNetSim subclass is the simulation of the activity of the desired neural network, ie, of one of the nets having the A1-A6 features.

A Boolean Neural Network with a Hebbian rule, or *HBNN*, as defined in the chapters 1. and 4., verifies both the features A1-A6 and the followings:

A7) The input of the node h at the time t is $I_h(t) = \Sigma_k \Sigma_d C_{hkd} * f_k(t\text{-}d)$.

A8) The state and the output of the node h at the time t is $f_h(t) = \|I_h(t) - T\|$, see the chapter 1.3..

A9) The coupling coefficient C_{hkd} is time dependent if both the tag property of the node k is equal to 2 or 3 and the tag property of the node h is equal to 1 or 3. Otherwise, C_{hkd} is time independent.

A10) The Hebbian rules defined in the chapter 4. regulates the temporal evolution of the time dependent C_{hkd}.

An instance of a suitable BoolHebbNet subclass allows the simulation of the activity of any HBNN instance.

7.2. The syntax and the semantic of the IT.NA.CY.NNET package.

In the Java language, a class is a set of declarations and/or implementations of variables, data structures and functions: the class members. That is, the class declarations define new reference types and describe how they are implemented. The name of a class has as its scope all the type declarations in the package in which the class has been declared. The body of a class declares the *members*, i.e., *fields*, *methods*, *static initializers*, and *constructors*.

1) The field declarations describe class variables.

2) The method declarations describe code that may be invoked by method invocation expressions, i.e., the method declarations are similar to the function declarations. A class method is invoked relative to the class type; an instance method is invoked with respect to some particular object that is an instance of the class type. A method whose declaration does not indicate how it is implemented must be declared abstract.

3) The static initializers are blocks of executable code that may be used to help initialize a class when it is first loaded.

4) The constructors are similar to methods, but cannot be invoked directly by a method call; they are used to initialize new class instances.

The optional *extends* clause in a class declaration specifies the direct superclass of the current class.

A class is said to be a direct subclass of the class it extends. A class inherits from its direct superclass all the methods of the superclass that are accessible to code in the class and are neither overridden nor hidden by suitable declarations in the class. The members of a class include both declared and inherited members from the direct superclass. A class may be declared *abstract*. It must be declared abstract if it is incompletely implemented, that is if it holds abstract methods. It is not possible to create an instance of an abstract class. A direct subclass of an abstract class is *non-abstract* if all abstract methods inherited from the direct superclass are implemented and if itself both is not declared abstract and it does not hold its own abstract methods.

For more information about the Java language see [23, 24].

7.2.1. The BoolHebbNet class.

Now, we discuss a simulation of a HBNN instance by means of the Java package it.na.cy.nnet.

As first step we must write the description of a HBNN instance, by means of the language LLNN, into a file named, e. g., networkFile. As second step we must implement a suitable non-abstract *BoolHebbNet* subclass in the Java language. BoolHebbNet is both an abstract class and a direct subclass of NeuralNetSim. If we want to implement a non-abstract subclass of BoolHebbNet, for example named *MyBoolHebbNet*, we must implement all abstract methods inherited from the direct superclass. The abstract methods in BoolHebbNet are *setActiveSensors*, *handleActiveMotors* and *defaultRun*. The methods setActiveSensors and handleActiveMotors should be implemented in order to define the communications to and from the simulated network, respectively. The method defaultRun should be implemented so to define the standard activity of a MyBoolHebbNet instance.

The class MyBoolHebbNet, in Java, should be implemented in the following way:

1) As first step, we need to implement the MyBoolHebbNet constructor, that is :

```
public class MyBoolHebbNet extends BoolHebbNet
{
//Here, the constructor is implemented. The variable networkName holds the file name
//where the HBNN is described by means of the language LLNN. The variable alfa,
//beta,
//gamma and delta hold the values of the Hebbian parameters, see the chapter 4.,
//present in
//the Hebbian rule.
public MyBoolHebbNet(String networkName, float alfa, float beta, float gamma,
float delta)
        {
          super(networkName, alfa, beta, gamma, delta);
        }
}
```

2) The second step consists in the implementation of the abstract methods inherited from the superclass BoolHebbNet:

```
public class MyBoolHebbNetextends BoolHebbNet
{
//Constructor
```

```
public MyBoolHebbNet(String networkName, float alfa, float beta, float gamma,
Float delta)
        {
          super(networkName, alfa, beta, gamma, delta);
        }
//Here, the abstract methods of the super class are implemented.
//The method setActiveSensors should hold suitable statements to define the com-
//munication
//protocol from the environment to the simulated network. Here, a possible im-
//plementation of the method setActiveSensors is presented.

public final void setActiveSensors()
        {
        Float x;

        ReadStream inputData=new ReadStream(System.in);
        System.out.println("Put an integer number:");
//The object inputData allows to read numbers from standard input (for example, a
//keyboard).

        x=inputData.readNextNumber();

//The method setSensorNodesByTwo, see the chapter 7. 2. 3. 16., is a method in-
//herited from
//the superclass BoolHebbNet. The following statement sets suitably, see the chapter
//7.2.3.,
//the output of the sensor nodes.

        if (x.intValue() >=0)
                setSensorsByTwo(x.intValue());
}

//The method handleActiveMotors should hold suitable statements to define the
//communication protocol from the network to its own environment. Here, a possi-
//ble
//implementation of this method is presented.

public final void handleActiveMotors()
        {

//The method printActiveMotorsAt is a method inherited from the superclass Bo-
//olHebbNet.
//The following statement allows to print the indices of the active motor nodes into
//the file
//named "out.network".

 printActiveMotorsAt(0,"out.network");
}

//The method defaultRun should hold suitable statements to define the default ac-
//tivity of a
//MyBoolHebbNet instance. Here, a possible implementation of this method is pre-
//sented.
```

```
public final void defaultRun()
        {

//The method innerActivity (see 7. 2. 3. 16.) is a method inherited from the super
//class
//BoolHebbNet. The following statement allows to compute the input, the state and
//the
//output of the network  nodes.

        innerActivity();

//The method HebbRule (see 7. 2. 3. 16.) is a method inherited from the superclass
//BoolHebbNet.
//The following statement allows to compute the values of the timedependent
//coupling coefficients following the second Hebbian rule defined in the chapter 4..
//The variable maxHebbDelay set the  maximum Hebbian Delay.

HebbRule(maxHebbDelay);

//The following statement allows to comunicate to the network environment fol-
//lowing the
//instructions implemented in handleActiveMotors.

handleActiveMotors();

nextTime();

        }

}
```

If we name mySimulator an instance of the class MyBoolHebbNet so implemented, that is, if in the Java language we write:

```
mySimulator - new MyBoolHebbNet( "networkFile", alfa, beta, gamma, delta);
```

where networkFile is the file name holding the HBNN description by means of the language LLNN and alfa, beta, gamma, delta are real numbers, then, the mySimulator object is able to simulate the activity of the HBNN described in the file networkFile, with Hebbian parameters equal to alfa, beta, gamma, delta. The following statement represents a possible way to simulate the network activity for the first L times in the Java language:

```
for (int i-0;i<L;++i)
{
mySimulator.setActiveSensors();
mySimulator.defaultRun();
}
```

Obviously, the methods setActiveSensors, handleActiveMotors and defaultRun may be implemented in different ways and the subclass MyBoolHebbNet has other methods inherited from the superclass BoolHebbNet. In the chapter 7. 2. 3. 16. we describe this class and its methods, also.

7.2.2. *The NeuralNetSim class.*

If we want to simulate the activity of an instance of a neural network having the features above described in A1, ..., A6, and having features different from those defined in A7, ..., A9, then we may use the *NeuralNetSim* class.

Also in this case, as first step, we must write the description of the neural network instance, by means of the language LLNN, into a file named, for example, networkFile. As second step we must implement a suitable non-abstract NeuralNetSim subclass in the Java language.

The NeuralNetSim is an abstract class too. If we want to implement a non-abstract subclass of NeuralNetSim, for example named MyNeuralNet, we must implement all abstract methods inherited from the direct superclass. The abstract methods in NeuralNetSim are *setActiveSensors*, *handleActiveMotors*, *computeNodeOutput*, *computeNodeInput* and *defaultRun*. The methods setActiveSensors and handleActiveMotors must be implemented in order to define the communications to and from the simulated network, respectively. The methods computeNodeOutput and computeNodeInput must be implemented to define the node output and the node input, respectively. The method defaultRun must be implemented so to define the default activity of a MyNeuralNet instance.

The class MyNeuralNet must be implemented in the following way:

1) As first step, the constructor has the form:

```
public class MyNeuralNet extends NeuralNetSim
{

//Here, the constructor is implemented. The variable networkName holds the file
//name
//where the HBNN is described by means of the language LLNN. The variable alfa,
//beta,
//gamma and delta hold the values of the Hebbian parameters present in the Heb-
//bian rule,
//see the chapter 4..

public MyNeuralNet(String networkName)
      {
       super(networkName, alfa, beta, gamma, delta);
      }

}
```

2) The second step consists in the implementation of the abstract methods inherited from the superclass NeuralNetSim:

```
public class MyNeuralNet extends NeuralNetSim
{
//Constructor
public MyNeuralNet (String networkName)
      {
       super(networkName, alfa, beta, gamma, delta);
      }
//Here, the abstract methods of the super class are implemented.
//The method setActiveSensors should hold suitable statements to define the
//communication protocol from the environment to the simulated network. Here, a
//possible
```

```
//implementation of the method setActiveSensors is presented.

public final void setActiveSensors()
        {
        Float xIndex,xOutput;
        ReadStream inputData=new ReadStream(System.in);

System.out.println("Put an integer number:");

//The object inputData allows to read numbers from standard input ( for example, a
//keyboard).
        xIndex=inputData.readNextNumber();
        xOutput= inputData.readNextNumber();

//The method setActiveSensor, see the chapter 7.2.3.15., is a method inherited from
//the
//superclass NeuralNetSim. The following statement sets suitably, see the chapter
//7.2.3., the
//output of the sensor node with index equal to xIndex.intValue().

        if ((xIndex.intValue() >=0)&&(xOutput.floatValue()>=0))
                setActiveSensor(xIndex.intValue(),xOutput.floatValue());

        }

//The method handleActiveMotors should hold suitable statements to define the
//communication protocol from the network to its own environment.
//Here, a possible implementation of this method is presented.

public final void handleActiveMotors()
        {

//The method printActiveMotorsAt is a method inherited from the superclass Neu-
//ralNetSim.
//The following statement allows to print the indices of the active motor nodes into
//The file named "out.network".

        printActiveMotorsAt(0,"out.network");

        }

//The method computeNodeInput should implement suitable statements in order to
//define the node
//input at time t. Here, a possible implementation of this method is presented.

public final void computeNodeInput(Node toNode,float Chkd,float fromOutput)
{
        toNode.input += Chkd *fromOutput;
}

//The method computeNodeOutput should implement suitable statements in order to
//define
//the node output at time t. Here, a possible implementation of this method is pre-
//sented.
```

```
public final void computeNodeOutput(Node xNode)
{
      int  t1=10, t2=20;
if ( xNode.input<t1) xNode.output=0;
if (xNode.input >t2) xNode.output=1;
if ( (xNode.input>=t1)&&(xNode.input<=t2)) xNode.output=2;

}

//The method defaultRun should hold suitable statements to define the default ac-
//tivity of a
//MyBoolHebbNetistance. Here, a possible implementation of this method is pre-
//sented.

public final void defaultRun()
      {

//The method innerActivity (see 7.2.3.15.) is a method inherited from the superclass
//NeuralNetSim. The following statement allows to compute the input, the state and
//The output of the network nodes.

      innerActivity();

//The following statement allows to comunicate to the network environment fol-
//lowing the
//instructions implemented in handleActiveMotors.

handleActiveMotors();
nextTime();
}
}
```

If we name mySimulator an instance of the class MyNeuralNet so implemented, that is, if in the Java language we write:

```
mySimulator = new  MyNeuralNet("networkFile");
```

where networkFile is the file name holding the network description by means of the language LLNN, then the mySimulator object is able to simulate the activity of the network described in networkfile. Also in this case, the following statement represents a possible way to simulate the network activity for the first L times:

```
for (int i=0;i<L;++i)
{
mySimulator.setActiveSensors();
mySimulator.defaultRun();
}
```

In the next chapter we discuss also the class NeuralNetSim and its methods.

Both MyNeuralNet and MyBoolHebbNet represent subclasses implemented in order to simulate the activity of a neural network. It is obvious that a subclass of NeuralNetSim or BoolHebbNet may be implemented in different ways, also, making use of the methods inherited from the direct superclasses, in order to simulate the activity of the de-

sired network. For example, it is possible to implement both setActiveSensors and handleActiveMotors in different ways in order to have the suitable communications between the network and its own environment.

7.2.3. *The classes of the java package it.na.cy.nnet.*

The hierarchy of the classes defined in the Java package it.na.cy.nnet are given below. The classes whose names are in **boldface** are in the package it.na.cy.nnet; the others are in the packages java.lang, java.io and java.util.

public interface **Netconstants;**
public class **Node**;
public class **RecordNode**;
public class **Connection**;
public class **InfoConnection**;
public class **NetVector** extends Vector;
public class **ConnectionVector** extends **NetworkVector**;
public class **Connections** extends **NetworkVector**;
public class **NodeVector** extends **NetworkVector**;
public class **PossibleActiveNodes** extends **NodeVector**;
public class **RecordNodeVector** extends **NetworkVector**;
public class **ActiveNodes** extends **NetworkVector**;
public class **Brain** extends **NetworkVector**;
public class **History** extends **NetworkVector**;
public abstract class **NeuralNetSim** implements **NetworkConstants**;
public abstract class **BoolHebbNet** extends **NeuralNetSim**;
public class **ReadStream** extends StreamTokenizer;
public class **ReadPrintRAF** extends RandomAccessFile;

7.2.3.1. *The interface it.na.cy.nnet.NetworkConstants.*

The interface *NetworkConstants* is a set of constants used in package it.na.cy.nnet

```
package it.na.cy.nnet;

public interface NetworkConstants
{
 public static final int UNKNOWN=-999;
 public static final int TYPESTEP=5;
 public static final int SENSOR=0;
 public static final int S_HEBB_IN=1;
 public static final int S_HEBB_OUT=2;
 public static final int S_HEBB_IO=3;
 public static final int HIDDEN=5;
 public static final int H_HEBB_IN=6;
 public static final int H_HEBB_OUT=7;
 public static final int H_HEBB_IO=8;
 public static final int MOTOR=10;
```

```
public static final int M_HEBB_IN=11;
public static final int M_HEBB_OUT=12;
public static final int M_HEBB_IO=13;
public static final int TNUM=MOTOR+TYPESTEP;
public static final int YES_FIRE=1;
public static final int NO_FIRE=0;
public static final int NO_TIME_DEP=0;
public static final int YES_TIME_DEP=1;
}
```

7.2.3.2. *The class it.na.cy.nnet.Node.*

The public class *Node* is a representation of a network node. The field *index* of this class holds the value of one network node index. The field name holds a logical name of the represented network node. The Node constructor initializes a newly created Node so that its fields index, position, property, fireThreshold, timeDepConnTh have the specified values.

```
package it.na.cy.nnet;
public class Node
{
//Public Members
public int input;
public int output;
public int innerState;
public int fireThreshold;
public int timeDepConnTh;
public int property;
public int position;
public int type;
public int index;
public String name;
public Connections timeIndConnections;
public Connections timeDepConnections;
//Constructors
public Node(int xIndex, int xPos, int xProp, int xFireTh, int xTimeDepConnTh);
public Node(String xName, int xIndex, int xPos, int xProp, int xFireTh, int
xTimeDepConnTh);
//Public Methods
public void computeBooleanOutput();
public void update(int xIndex, int xPos, int xProp, int xFireTh,
int xTimeDepConnTh);
public void update(String xName, int xIndex, int xPos, int xProp, int xFireTh, int
xTimeDepConnTh)
}
```

- public int **input**;

The field input holds an integer value. It represents the input of the represented node.

- public int **output**;

The field output holds an integer value. It represents the output of the represented node.

- public int **innerState**;

The field innerState holds an integer value. It represents the state of the represented node.

- public int **fireThreshold**;

The field fireThreshold holds an integer value. It is a threshold in order to compute the node output.

- public int **timeDepConnTh**;

The field timeDepConnTh holds an integer value. It is a threshold in order to regulate the time dependent coupling coefficients.

- public int **property**;

At each node can be associated a value that identifies a property of the node. The field property holds that property assuming one among five integer values: 0, 1, 2, 3 or 4. The default value is 0.

- public int **position**;

Each node can be in one among three position: sensor position, inner position and motor position.
The field position can assume the following values: NetworkConstants.SENSOR (sensor position), NetworkConstants.HIDDEN (inner position) and NetworkConstants.MOTOR (motor position).

- public int **type**;

The field type is the sum of the values of both position and property. The field type assumes the following values: 0, 1, 2, ..., 14. So, type holds both the value of position and the value of property.

- public int **index**;

The field index holds an integer value. It is the value of one network node index. Instances of the class Node which have the same *index* value represent the same network node.

- public String **name**;

The field name holds the logical name of this Node object.

- public Connections **timeIndConnections**;

This field holds the connections with time independent coupling coefficients starting from this Node.

- public Connections **timeDepConnections**;

This field holds the connections with time dependent coupling coefficients starting form this Node.

- public **Node**(int xIndex, int xPos, int xProp, int xFireTh, int xTimeDepConnTh);

This constructor initializes a newly created Node so that its fields index, position, property, fireThreshold, timeDepConnTh have the specified values.

- public **Node**(String xName, int xIndex, int xPos, int xProp, int xFireTh, int xTimeDepConnTh);

This constructor initializes a newly created Node so that its fields name, index, position, property, fireThreshold, timeDepConnTh have the specified values.

- public void **computeBooleanOutput()**;

The output value of this Node is computed in the following way:

```
public void computeBooleanOutput()
    {
```

```
if (this.input >= fireThreshold) output=NetworkConstants.YES_FIRE;
else this.output=NetworkConstants.NO_FIRE;
this.input=0;
}
```

- public void **update**(int xIndex, int xPos, int xProp, int xFireTh, int xTimeDepConnTh);

The fields index, position, property, fireThreshold, timeDepConnTh are updated to the specified values. If a value is equal to NetworkCostants.Unknown then the corresponding field value is unchanged.

- public void **update**(String xName, int xIndex, int xPos, int xProp, int xFireTh, int TimeDepConnTh)

The fields name, index, position, property, fireThreshold, timeDepConnTh are updated to the specified values. If a value is equal to NetworkCostants.Unknown then the corresponding field value is unchanged.

7.2.3.3. *The class it.na.cy.nnet.RecordNode.*

An instance of the class RecordNode holds some data on the activity of a network node at a specific time.

```
package it.na.cy.nnet;

public class RecordNode
{
//Private Members
private Node toNode;
private int output;
private int innerState;
//Constructors
public RecordNode(Node xNode, int xIS, int xOut);
public RecordNode(Node xNode);
//Public Methods
public int getOutput();
public int getInnerState();
public Node getNode();
public void setNode(Node xNode);
public void setOutput(int xOut);
public void setInnerState(int xIS);
}
```

- private Node toNode;

The field toNode holds a Node object representing a network node

- private int output;

The field output holds an integer value. This value should be the output value, at a specific time, of the Node instance held in this toNode.

- private int innerState;

The field innerState holds an integer value. This value should be the innerState value, at a specific time, of the Node instance held in this toNode.

- public RecordNode(Node xNode, int xIS, int xOut);

This constructor initializes a newly created RecordNode so that its own fields assumes the following values: toNode = xNode, innerState = xIS, output = xOut.

- public RecordNode(Node xNode);

This constructor initializes a newly created RecordNode so that its own fields assumes the following values: toNode = xNode, innerState = xNode.innerState, output = xNode.output.

- public int getOutput()

The output value of this RecordNode is returned.

- public int getInnerState()

The innerState value of this RecordNode is returned.

- public Node getNode()

The toNode value of this RecordNode is returned.

- public void setNode(Node xNode)

The toNode value of this RecordNode becomes the xNode value.

- public void setOutput(int xOut)

The output value of this RecordNode becomes the xOut value.

- public void setInnerState(int xIS)

The innerState value of this RecordNode becomes the xIS value.

7.2.3.4. *The class it.na.cy.nnet.Connection.*

The class *Connection* represents a network connection starting from a network node.

```
package it.na.cy.nnet;

public class Connection
{
//Private Members
private Node toNode;
private int coupling coefficient;
private int timeDep;
//Constructors
public Connection(Node xTo, int xCoupling coefficient);
public Connection(Node xTo, int xCoupling coefficient, int xTimeDep);
//Public Methods
public void setTimeDep(int xVal);
public void updateCoupling coefficient(int xSum);
public void setCoupling coefficient(int xCoupling coefficient);
public int getTimeDep();
public int getCoupling coefficient();
public Node getNode();
}
```

- private Node toNode;

This field holds a Node occurrence.

- private int coupling coefficient;

The coupling coefficient value is an integer value. It represents a network coupling coefficient.

- private int timeDep;

The timeDep value is an integer value. If timeDep is equal to NetworkConstants.YES_TIME_DEP then this Connection represents a network connection with time dependent coupling coefficient, otherwise this Connection represents a network connection with time independent coupling coefficient.

- public Connection(Node xTo, int xCoupling coefficient);

A newly created Connection is initialized so that its internal fields toNode and coupling coefficient are set equal to xTo and xCoupling coefficient, respectively. The field timeDep is set equal to NetworkConstants.NO_TIME_DEP.

- public Connection(Node xTo, int xCoupling coefficient, int xTimeDep);

A newly created Connection is initialized so that its internal fields toNode, coupling coefficient and timeDep are set equal to xTo, xCoupling coefficient and xTimeDep, respectively.

- public void setTimeDep(int xVal);

The timeDep value of this Connection becomes equal to xVal.

- public void updateCoupling coefficient(int xSum);

The coupling coefficient value of this Connection is incremented by xSum.

- public void setCoupling coefficient(int xCoupling coefficient);

The coupling coefficient value of this Connection becomes equal to xCoupling coefficient.

- public int getTimeDep();

The timeDep value of this Connection is returned.

- public int getCoupling coefficient();

The coupling coefficient value of this Connection is returned.

- public Node getNode();

The toNode value of this Connection is returned.

7.2.3.5. *The class it.na.cy.nnet.InfoConnection*

The class *InfoConnection* is an object that should hold all data inherent a network connection.

```
package it.na.cy.nnet;

public class InfoConnection
{
//Private Members
private int coupling coefficient;
private Node toNode;
private Node fromNode;
private int delay;
private int timeDep;
//Constructors
public InfoConnection(Node xToNode, Node xFromNode, int xDelay);
public InfoConnection(Node xToNode, Node xFromNode, int xDelay, int xCoupling coefficient);
public InfoConnection(Node xToNode, Node xFromNode, int xDelay, int xCoupling coefficient, int xTimeDep);
//Public Methods
public void setToNode(Node xNode);
public void setFromNode(Node xNode);
public void setCoupling coefficient(int x);
public void setDelay(int x);
public void setTimeDep(int x);
public Node getToNode();
```

public Node **getFromNode**();
public int **getCoupling coefficient**();
public int **getDelay**();
public int **getTimeDep**();
}

- private int **coupling coefficient**;

The field coupling coefficient holds an integer number representing the network connection coupling coefficient.

- private Node **toNode**;

The field toNode holds a Node object representing a network node from which the network connection goes out.

- private Node **fromNode**;

The field toNode holds a Node object representing a network node to which the network connection goes in.

- private int **delay**;

The field delay holds an integer number representing the network connection delay.

- private int **timeDep**;

The field timeDep assumes one between two values: NetworkConstants.NO_TIME_DEP and NetworkConstants.YES_TIME_DEP. If timeDep and NetworkConstants.YES_TIME_DEP are equals the network connection coupling coefficient is time dependent otherwise it is time independent.

- public **InfoConnection**(Node xToNode, Node xFromNode, int xDelay);

This constructor initializes a newly created InfoConnection so that its own fields assumes the following values: toNode = xToNode, fromNode = xFromNode, delay = xDelay, coupling coefficient = NetworkConstants.UNKNOWN, and timeDep = NetworkConstants.NO_TIME_DEP.

- public **InfoConnection**(Node xToNode, Node xFromNode, int xDelay, int xCoupling coefficient);

This constructor initializes a newly created InfoConnection so that its own fields assumes the following values: toNode = xToNode, fromNode = xFromNode, delay = xDelay, coupling coefficient = xCoupling coefficient, and timeDep = NetworkConstants.NO_TIME_DEP.

- public **InfoConnection**(Node xToNode, Node xFromNode, int xDelay, int xCoupling coefficient, int xTimeDep);

This constructor initializes a newly created InfoConnection so that its own fields assumes the following values: toNode = xToNode, fromNode = xFromNode, delay = xDelay, coupling coefficient = xCoupling coefficient, and timeDep = xTimeDep.

- public void **setToNode**(Node xNode);

The toNode value of this InfoConnection becomes xNode.

- public void **setFromNode**(Node xNode);

The fromNode value of this InfoConnection becomes xNode.

- public void **setCoupling coefficient**(int x);

The coupling coefficient value of this InfoConnection becomes x.

- public void **setDelay**(int x);

The delay value of this InfoConnection becomes x.

- public void **setTimeDep**(int x);

The timeDep value of this InfoConnection becomes x.

- public Node **getToNode**();

The toNode value of this InfoConnection is returned.

- public Node **getFromNode**();

The fromNode value of this InfoConnection is returned.

- public int **getCoupling coefficient**();

The coupling coefficient value of this InfoConnection is returned.

- public int **getDelay**();

The delay value of this InfoConnection is returned.

- public int **getTimeDep**();

The timeDep value of this InfoConnection is returned.

7.2.3.6. *The class it.na.cy.nnet.NetworkVector.*

The class *NetworkVector* is a subclass of Vector belonging to the java.util class. Hence, a NetworkVector object is equivalent to a Vector object, i.e., a NetworkVector object is an array containing items that can be accessed using an integer index; however, the size of that object can grow and shrink as needed to accommodate the adding and the removing of items after the object has been created.

```
package it.na.cy.nnet;

public class NetworkVector extends Vector
{
 //Constructors
 public NetworkVector(int initialCapacity);
 public NetworkVector();
 public NetworkVector(int initialCapacity, int capacityIncrement);
 //Public Methods
public Integer getIntegerAt(int i);
public Connection getConnectionAt(int i);
public Float getFloatAt(int i);
public   String getStringAt(int i);
public   Node getNodeAt(int i);
public   NetworkVector getNetworkVectorAt(int i);
public   RecordNode getRecordNodeAt(int i);
public   InfoConnection getInfoConnectionAt(int i);
}
```

- public **NetworkVector**();

This constructor initializes a newly created NetworkVector so that its internal data array has size 10 and its standard capacity increment is zero. Initially, the NetworkVector contains no items.

- public **NetworkVector**(int initialCapacity);

This constructor initializes a newly created NetworkVector so that its internal data array has size initialCapacity and its standard capacity increment is zero. Initially, the NetworkVector contains no items.

- public **NetworkVector**(int initialCapacity, int capacityIncrement);

This constructor initializes a newly created NetworkVector so that its internal data array has size initialCapacity and its standard capacity increment is the value of capacityIncrement. Initially, the NetworkVector contains no items.

- public Integer **getIntegerAt**(int i);

The item of this NetworkVector with the specified index i is, first, cast to Integer class and, next, the item is returned. If run-time class cast exception is thrown then it is caught and the null-pointer value is returned. If the index i is negative or not less than the current size of this NetworkVector, an ArrayIndexOutOfBoundException is thrown, then it is caught and the null-pointer is returned.

- public Float **getFloatAt**(int i);

The item of this NetworkVector with the specified index i is, first, cast to Float class and, next, the item is returned. If run-time class cast exception is thrown then it is caught and the null-pointer value is returned. If the index i is negative or not less than the current size of this NetworkVector, an ArrayIndexOutOfBoundException is thrown, then it is caught and the null-pointer is returned.

- public String **getStringAt**(int i);

The item of this NetworkVector with the specified index i is, first, cast to String class and, next, the item is returned. If run-time class cast exception is thrown then it is caught and the null-pointer value is returned. If the index i is negative or not less than the current size of this NetworkVector, an ArrayIndexOutOfBoundException is thrown, then it is caught and the null-pointer is returned.

- public Connection **getConnectionAt**(int i);

The item of this NetworkVector with the specified index i is, first, cast to Connection class and, next, the item is returned. If run-time class cast exception is thrown then it is caught and the null-pointer value is returned. If the index i is negative or not less than the current size of this NetworkVector, an ArrayIndexOutOfBoundException is thrown, then it is caught and the null-pointer is returned.

- public Node **getNodeAt**(int i);

The item of this NetworkVector with the specified index i is, first, cast to Node class and, next, the item is returned. If run-time class cast exception is thrown then it is caught and the null-pointer value is returned. If the index i is negative or not less than the current size of this NetworkVector, an ArrayIndexOutOfBoundException is thrown, then it is caught and the null-pointer is returned.

- public NetworkVector **getNetworkVectorAt**(int i);

The item of this NetworkVector with the specified index i is, first, cast to NetworkVector class and, next, the item is returned. If run-time class cast exception is thrown then it is caught and the null-pointer value is returned. If the index i is negative or not less than the current size of this NetworkVector, an ArrayIndexOutOfBoundException is thrown, then it is caught and the null-pointer is returned.

- public RecordNode **getRecordNodeAt**(int i);

The item of this NetworkVector with the specified index i is, first, cast to RecordNode class and, next, the item is returned. If run-time class cast exception is thrown then it is caught and the null-pointer value is returned. If the index i is negative or not less than the current size of this NetworkVector, an ArrayIndexOutOfBoundException is thrown, then it is caught and the null-pointer is returned.

- public InfoConnection **getInfoConnectionAt**(int i);

The item of this NetworkVector with the specified index i is, first, cast to InfoConnection class and, next, the item is returned. If run-time class cast exception is thrown then it is caught and the null-pointer value is returned. If the index i is negative or not less than the current size of this NetworkVector, an ArrayIndexOutOfBoundException is thrown, then it is caught and the null-pointer is returned.

7.2.3.7. *The class it.na.cy.nnet.ConnectionVector:*

ConnectionVector is a subclass of NetworkVector. Any item in ConnectionVector is a Connection.

```
package it.na.cy.nnet;

public class ConnectionVector extends NetworkVector
```

{
//Constructor
public **ConnectionVector**();
//Public Methods
public void **addConn**(Node xTo, int xCoupling coefficient);
public void **addConn**(Node xTo, int xCoupling coefficient, int xTimeDep);
public void **addConn**(Connection newConn);
public Connection **findConn**(int xTo);
}

- public **ConnectionVector**();

A newly created ConnectionVector is initialized so that its internal data array has size 0.

- public void **addConn**(Node xTo, int xCoupling coefficient);

The Connection occurrence xConn is created applying the Connection constructor, xConn = new Connection(xTo, xCoupling coefficient). The size of this ConnectionVector is incremented by 1. The last item becomes xConn.

- public void **addConn**(Node xTo, int xCoupling coefficient, int xTimeDep);

The Connection occurrence xConn is created applying the Connection constructor, xConn = new Connection(xTo, xCoupling coefficient, xTimeDep). The size of this ConnectionVector is incremented by 1. The last item becomes xConn.

- public void **addConn**(Connection newConn);

The size of this ConnectionVector is incremented by 1. The last item becomes newConn.

- public Connection **findConn**(int xTo);

If a Connection occurrence with the field toNode equal to xTo exists in this ConnectionVector then this Connection is returned, otherwise the null-pointer is returned.

7.2.3.8. *The class it.na.cy.nnet.Connections.*

The class *Connections* is a subclass of NetworkVector. Any item in Connections is a ConnectionVector. The ConnectionVector that is at index i in this NetworkVector should hold all delayed connections with delay equal to i+1.

```
package it.na.cy.nnet;

public class Connections extends NetworkVector
{
//Constructor
public Connections()
//Methods
public Connection getDelayedConnAt(int xDelay, int xConn)
public ConnectionVector getConnectionVectorAt(int i)
public Connection addConn(Connection xConn, int xDelay)
public Connection addConn(Node xTo, int xCoupling coefficient, int xDelay)
public Connection addConn(Node xTo, int xCoupling coefficient, int xDelay, int
xTimeDep)
public Connection findConn(int xTo, int xDelay)
}
```

- public **Connections**()

A newly created Connections is initialized so that its internal data array has size 0.

- public Connection **getDelayedConnAt**(int xDelay, int xInd)

The Connection that is at index xInd in the ConnectionVector that is at index xDelay in this Connections is returned. If an exception ArrayIndexOutOfBoundsException is thrown then it is caught and the null-pointer is returned.

- public ConnectionVector **getConnectionVectorAt**(int xDelay)

The ConnectionVector that is at index xDelay in this Connections is returned. If an exception ArrayIndexOutOfBoundsException is thrown then it is caught and the null-pointer is returned.

- public Connection **addConn**(Connection xConn, int xDelay)

If the size of this Connections is less than xDelay it becomes equal to xDelay. The size of the ConnectionVector that is at index xDelay is increase by 1 and xConn becomes the new last item of this ConnectionVector.

- public Connection **addConn**(Node xTo, int xCoupling coefficient, int xDelay)

A new Connection occurrence xConn is created applying the Connection constructor, xConn = Connection(xTo, xCoupling coefficient). If the size of this Connections is less than xDelay it becomes equal to xDelay. The size of the ConnectionVector that is at index xDelay is increase by 1 and xConn becomes the new last item of this ConnectionVector.

- public Connection **addConn**(Node xTo, int xCoupling coefficient, int xDelay, int xTimeDep)

A new Connection occurrence xConn is created applying the Connection constructor, xConn = Connection(xTo, xCoupling coefficient, xTimeDep). If the size of this Connections is less than xDelay it becomes equal to xDelay. The size of the ConnectionVector that is at index xDelay is increase by 1 and xConn becomes the new last item of this ConnectionVector.

- public Connection **findConn**(int xTo, int xDelay)

If a Connection occurrence with the field toNode equal to xTo exists in ConnectionVector that is at index xDelay-1 then this Connection is returned, otherwise the null-pointer is returned.

7.2.3.9. *The class it.na.cy.nnet.NodeVector:*

NodeVector is a subclass of NetworkVector. Any item of a NodeVector is a Node.

```
package it.na.cy.nnet;

public class NodeVector extends NetworkVector
{
//Constructors
public NodeVector(int initialCapacity);
public NodeVector();
public NodeVector(int initialCapacity, int capacityIncrement);
//Public Methods
public void addOrUpdateNode(int xTo, int xPos, int xProperty, int xFireTh, int
xTimeDepConnTh);
public void addOrUpdateNode(String xName, int xTo, int xPos, int xProperty,
int xFireTh,
int xTimeDepConnTh);
public void updateNodeAndConn(int xTo, int xFrom, int xDelay, int xCoupling
coefficient);
public Node findNode(int index);
```

}

- public NodeVector();

This constructor initializes a newly created NodeVector so that its internal data array has size 0 and its standard capacity increment is one. Initially, the NodeVector contains no items.

- public NodeVector(int initialCapacity);

This constructor initializes a newly created NodeVector so that its internal data array has size initialCapacity and its standard capacity increment is one. Initially, the NodeVector contains no items.

- public NodeVector(int initialCapacity, int capacityIncrement);

This constructor initializes a newly created NodeVector so that its internal data array has size initialCapacity and its standard capacity increment is capacityIncrement. Initially, the NodeVector contains no items.

- public void addOrUpdateNode(int xTo, int xPos, int xProperty, int xFireTh, int xTimeDepConnTh);

public void addOrUpdateNode(String xName, int xTo, int xPos, int xProperty, int xFireTh, int xTimeDepConnTh);

If the size of this NodeVector is grater than the value of xTo then the Node that is at index xTo is updated by the specified values otherwise both the size of this NodeVector is set to value of xTo+1 and a new Node with the specified values is placed at index xTo in this NodeVector.

- public void updateNodeAndConn(int xTo, int xFrom, int xDelay, int xCouplingcoefficient)

First, a new Connection instance with the specified values is created. Second, the method addOrUpdateNode is invoked for this instance of NodeVector. Last, the new Connection is added at the Node with index xFrom.

- public Node findNode(int xIndex)

If a Node instance with index value equal to xIndex is in this NodeVector then the said instance is returned, otherwise the null pointer is returned.

7.2.3.10. *The class it.na.cy.nnet.PossibleActiveNodes.*

The class *PossibleActiveNodes* is a subclass of NodeVector. So, any item in a PossibleActiveNodes instance is a Node occurrence.

```
package it.na.cy.nnet;

public class PossibleActiveNodes extends NodeVector
{
//Constructor
public PossibleActiveNodes();
//Public Method
public void updateWith(Object xNode);
}
```

- public **PossibleActiveNodes**();

This constructor initializes a newly created PossibleActiveNodes so that its internal data array has size 0 and its standard capacity increment is one. Initially, the PossibleActiveNodes contains no items.

- public void **updateWith**(Object xNode);

If this PossibleActiveNode does not contain xNode then the size of this PossibleActiveNode is incremented by 1 and the last item becomes xNode.

7.2.3.11. *The class it.na.cy.nnet.RecordNodeVector:*

The class *RecordNodeVector* is a subclass of NetworkVector. Any item in a RecordNodeVector instance is a RecordNode occurrence.

```
package it.na.cy.nnet;

public class RecordNodeVector extends NetworkVector
{
 //Constructors
 public RecordNodeVector(int initialCapacity);
 public RecordNodeVector();
 public RecordNodeVector(int initialCapacity, int capacityIncrement);
//Public Methods
 public void addElement(RecordNode xRecordNode);
 public void addElement(Node xNode);
 public void setElementAt(RecordNode xRN, int i) throws
 ArrayIndexOutOfBoundsException
 public void setElementAt(Node xNode, int i) throws ArrayIndexOutOfBound-
sException
 public RecordNode findNode(int index);
}
```

- public **RecordNodeVector**();

This constructor initializes a newly created PossibleActiveNodes so that its internal data array has size 0 and its standard capacity increment is 1. Initially, the PossibleActiveNodes contains no items.

- public **RecordNodeVector**(int initialCapacity);

This constructor initializes a newly created PossibleActiveNodes so that its internal data array has size initialCapacity and its standard capacity increment is 1. Initially, the PossibleActiveNodes contains no items.

- public **RecordNodeVector**(int initialCapacity, int capacityIncrement);

This constructor initializes a newly created PossibleActiveNodes with an internal data array having the size initialCapacity and the standard capacity increment equal to capacityIncrement. Initially, the PossibleActiveNodes contains no items.

- public void **addElement**(RecordNode xRecordNode);

The size of this RecordNodeVector is incremented by 1 and the last item becomes xRecordNode.

- public void **addElement**(Node xNode);

First, a new RecordNode with the specified xNode value is created. Second, the size of the current RecordNodeVector is incremented by 1 and the last item becomes the new RecordNode.

- public void **setElementAt**(RecordNode xRN, int i) throws ArrayIndexOutOfBoundsException

The item of this RecordNodeVector with the specified index i is replaced with xRN, so that xRN is now the item at the specified index i within the current RecordNodeVector.
If the index is negative or not less than the current size of the currrent RecordNodeVector, an ArrayIndexOutOfBoundsException is thrown.

- public void **setElementAt**(Node xNode, int i) throws ArrayIndexOutOfBoundsException

First, a new RecordNode xRN by the specified xNode value is created. Second, the item of this RecordNodeVector with the specified index i is replaced with xRN, so that xRN is now the item at the specified index i within the current RecordNodeVector.
If the index is negative or not less than the current size of this RecordNodeVector, an ArrayIndexOutOfBoundsException is thrown.

- public RecordNode **findNode**(int xIndex);

If this RecordNodeVector contains a RecordNode holding a Node with the index value equal to xIndex, then, the RecordNode is returned; otherwise the null pointer is returned.

7.2.3.12. *The class it.na.cy.nnet.ActiveNodes.*

ActiveNodes is a subclass of NetworkVector and any item of ActiveNodes is a RecordNodeVector.

```
package it.na.cy.nnet;

public class ActiveNodes extends NetworkVector
{
 //Constructor
 public ActiveNodes();
//Methods
 public RecordNodeVector getElementAt(int i);
 public RecordNode getElementAt(int i, int j);
}
```

- public **ActiveNodes**();

A newly created ActiveNodes is initialized with the size NetworkConstants.TNUM: for any index, the item is a RecordNodeVector with size 0 and standard capacity increment equal to one.

- public RecordNodeVector **getElementAt**(int i);

The RecordNodeVector at specified index i is returned.

- public RecordNode **getElementAt**(int i, int j);

The RecordNode at specified index j of the RecordNodeVector at specified index i is returned.

7.2.3.13. *The class it.na.cy.nnet.Brain.*

Brain is a subclass of NetworkVector. Any item of an instance of Brain is a NodeVector object. The size of this Brain is NetworkConstants.TNUM. In this Brain instance the NodeVector at index i should hold all the Node occurrences with field type equal to i.

```
package it.na.cy.nnet;

public class Brain extends NetworkVector
{
//Constructor
public Brain();
//Public Methods
public Node getNodeAt(int i, int xType);
public NodeVector getNodeVectorAt(int xType);
```

public Node **findNode**(int index);
public Node **findNode**(int index, int xType);
}

- **Brain**();

A newly created Brain is initialized: first, the Brain size is set equal to NetworkConstants.TNUM; second, at any index the item is a NodeVector object with size 0 and standard capacity increment equal to one.

- public Node **getNodeAt**(int i, int xType);

The Node occurrence at index i in the NodeVector at index xType in the current Brain is returned. If an exception ArrayIndexOutOfBoundsException is thrown then it is caught and the null-pointer is returned.

- public NodeVector **getNodeVectorAt**(int xType);

The NodeVector at index xType in the current Brain is returned. If an exception ArrayIndexOutOfBoundsException is thrown then it is caught and the null pointer is returned.

- public Node **findNode**(int xIndex);

If an object Node with the field index equal to xIndex exists it is returned; otherwise the null pointer is returned.

- public Node **findNode**(int index, int xType);

If a Node object with both the field index equal to xIndex and the field type equal to xType exists the current Node is returned; otherwise the null pointer is returned.

7.2.3.14. *The class it.na.cy.nnet.History.*

The class *History* is a subclass of NetworkVector. Any item of History is an ActiveNodes occurrence.

```
package it.na.cy.nnet;

public class History extends NetworkVector
{
 //Constructors
 public History(int xMaxDelay);
 // Public Methods
 public void leftShift();
 public RecordNode findNode(int nodeIndex, int xTime);
 public RecordNode findNode(int nodeIndex, int xTimeMin, int xTimeMax);
 public ActiveNodes getActiveNodesAt(int i);
 public void  resetElementAt(int index);
}
```

- public History(int xMaxDelay)

A newly created History is initialized so that its internal data array has size xMaxDelay. At any index the item is a ActiveNodes object.

- public void leftShift()

Any item in the current History with the index k is moved to index k+1if and only if k is different from the last index. The item with the last index is destroyed.

- public RecordNode findNode(int nodeIndex, int xTime)

If the ActiveNodes occurrence with the index xTime of the current History contains an occurrence of RecordNode with the field toNode holding a Node occurrence with index value equal to nodeIndex, then, the current RecordNode is returned; otherwise the null pointer is returned.

- public RecordNode findNode(int nodeIndex, int xTimeMin, int xTimeMax)

If the ActiveNodes occurrences with the index k, for xTimeMin ≤ k ≤ xTimeMax, of the current History contain an occurrence of RecordNode with the field toNode holding a Node occurrence with index value equal to nodeIndex, then the current RecordNode is returned; otherwise the null pointer is returned.

- public ActiveNodes getActiveNodesAt(int i)

The ActiveNodes occurrence with the index i is returned. If an exception ArrayIndexOutOfBoundsException is thrown, then, it is caught and the null pointer is returned.

- public void resetElementAt(int xIndex)

The item of the current History with the specified index xIndex is replaced with a new ActiveNodes.

7.2.3.15. *The class it.na.cy.nnet.NeuralNetSim.*

NeuralNetSim is an abstract class. A not abstract NeuralNetSim subclass must provide an implementation of all abstract methods declared in NeuralNetSim.

If Network is the name of a suitable not abstract NeuralNetSim subclass and myNetwork is an instance of Network then the myNetwork life cycle could be the following:

myNetwork.setActiveSensors() //Abstract method. It must have been previously

//implemented.
myNetwork.innerActivity()
3) myNetwork.handleActiveMotors() //Abstract method. It must have been pre-
//viously
// implemented.
4) myNetwork.nextTime()

One must also suitably define the abstract methods computeNodeOutput, computeNodeInput and defaultRun().

```
package it.na.cy.nnet;
import java.io.*;
public abstract class NeuralNetSim implements NetworkConstants
{
//Private Members
private Brain myBrain;
private NodeVector transientBrain;
private History myHistory;
private PossibleActiveNodes myPosActNodes;
private NetworkVector sensorNodes;
private NetworkVector activeSensorNodes;
private int maxDelay;
private int myTime;
private String networkName;
private int defaultFireTh;
private final static int maxMaxDelay=20;
//Constructor
public NeuralNetSim(String xNetworkName);
//Abstract Methods
protected abstract void computeNodeOutput(Node xNode);
```

protected abstract void **computeNodeInput**(Node toNode, float fromToWeigth, float fromOutput);
public abstract void **setActiveSensors**(Object xObject);
public abstract void **handleActiveMotors**(Object xObject);
public abstract void **defaultRun**(Object xObject);
//Private Methods
private boolean **buildInAndOutNodes**(File ioFile);
private boolean **buildHiddenNodes**(File objFile);
private boolean **udateProperty**(File hbbFile);
private void **setType**();
private void **buildBrain**();
private void **updateNodeInput**(RecordNode rn, int xDelay);
//Public Methods
public void **innerActivity**();
public void **nextTime**();
public NetworkVector **getTimeIndConnOf**(int index);
public NetworkVector **getTimeDepConnOf**(int index);
public int **getCoupling coefficientOf**(int toIndex,int fromIndex,int xDelay);
public History **getHistory**();
public Brain **getBrain**();
public int **getMaxDelay**();
public int **getBrainSize**();
public int **getHistorySize**();
public ActiveNodes **getHistoryElementAt**(int i);
public NetworkVector **getActiveNodesAt**(int xDelay);
public NetworkVector **getActiveMotorsAt**(int xDelay);
public NetworkVector **getActiveSensorsAt**(int xDelay);
public NetworkVector **getActiveHiddenAt**(int xDelay);
public NetworkVector **getSensorNodes**();
public int **getCurrentTime**();
public InfoConnection **findConnection**(int toIndex,int fromIndex, int xDelay);
public NetworkVector **findConnections**(int toIndex,int fromIndex);
public void **setActiveSensor**(int xIndex,float xOutput);
public void **setActiveSensor**(String xName, float xOutput);
public void **setMaxDelay**(int x);
public void **printActiveNodesAt**(int xDelay, File outFile);
public void **printActiveNodesAt**(int xDelay, String nameFile);
public void **printActiveSensorsAt**(int xDelay, File outFile);
public void **printActiveSensorsAt**(int xDelay, String nameFile);
public void **printActiveMotorsAt**(int xDelay, File outFile);
public void **printActiveMotorsAt**(int xDelay, String nameFile);
public void **printActiveHiddenAt**(int xDelay, File outFile);
public void **printActiveHiddenAt**(int xDelay, String nameFile);
public void **printConnections**(int xTo,int xFrom, File outFile);
public void **printConnections**(int xTo,int xFrom, String nameFile);
}

- private int **myTime**;

The field holds an integer number representing the network time at the present time. Initially it is 0.

- private Brain **myBrain**;

The field holds a Brain object. It contains all Node occurrences representing the nodes belonging to the wanted network. In the current myBrain the NodeVector at index i holds all the Node occurrences with field type equal to i.

- private NodeVector **transientBrain**;

The field holds a NodeVector object. The NeuralNetSim constructor uses the current transientBrain object to implement the myBrian object.

- private History **myHistory**;

The field myHistory holds a History object. The size of myHistory is equal to the value of the field maxDelay incremented by 1. Any item in the current myHistory is an ActiveNodes occurrence. The item at the index k holds all the active nodes at the time myTime-k. Eg, the item at the index 0 contains all the active nodes at the present time.

- private PossibleActiveNodes **myPosActNodes**;

It holds a PossibleActiveNodes occurrence. The object myPosActNodes contains the possible active nodes at the present time. Any item of myPosActNodes is also an item of a NodeVector object belonging to myBrain.

- private NetworkVector **sensorNodes**;

It is a NetworkVector and it contains all Node occurrences corresponding to sensor nodes. Any item of sensortNodes is also an item of a NodeVector object belonging to myBrain.

- private NetworkVector **activeSensorNodes**;

It is a NetworkVector and contains the Node occurrences corresponding to the sensor nodes active at the present time. Any item of activeSensortNodes is also an item of a NodeVector object belonging to myBrain.

- private int **maxDelay**;

The field maxDelay holds an integer number representing the maximum delay associated to a delayed connection belonging to the simulated network.

- private String **networkName**;

This field holds the network name.

- private int **defaultFireTh**;

It is a integer number representing the default fire threshold.

- public **NeuralNetSim**(String xNetworkName);

The constructor may be only used by a not abstract NeuralNetSim subclass. If NNetwork is the name of the said subclass then the constructor initializes a newly created NNetwork by opening connections to an actual file, ie, to the file named by the path name xNetworkName.nnt in the file system. A suitable low-level language, see the chapter 7.4, must describe the desired simulated network in the above file.

- public abstract void **computeNodeInput**(Node toNode, float fromToWeigth, float fromOutput);

The method innerActivity() employs the abstract method computeNodeInput to compute the input value of a Node occurrence, however it is a method declared but not yet implemented. This abstract method can be implemented in a subclass of NeuralNetSim. We introduce this abstract method in NeuralNetSim class in order to force the user to define the own network node input as soon as he implements a NeuralNetSim subclass that is not abstract. This method should define as the input of the node toNode is transformed by the output value fromOutput of a node having a connection going from itself to the node toNode with coupling coefficient equal to fromToCoupling coefficient. Here, we give an example of implementation:

```
public abstract void computeNodeInput(Node toNode, float fromToWeigth, float
fromOutput)
    {
```

```
        toNode.input=toNode.input +fromToCoupling coefficient*fromOutput;
    }
```

- public abstract void **computeNodeOutput**(Node xNode);

The method innerActivity() employs the abstract method computeNodeOutput to compute the output value of a Node occurrence, however it is a method declared but not yet implemented. This abstract method can be implemented in a subclass of NeuralNetSim. We introduce this abstract method in NeuralNetSim class in order to force the user to define the own network node output as soon as he implements a NeuralNetSim subclass that is not abstract. Here, we give an example of implementation:

```
public final void computeNodeOutput(Node xNode)
    {
     if (xNode.input >= fireThreshold) xNode.output=NetworkConstants.YES_
     FIRE;
     else xNode.output=NetworkConstants.NO_FIRE;
     xNode.input=0;
    }
```

- public abstract void **setActiveSensors**(Object xObject);

public abstract void **handleActiveMotors**(Object xObject);

The user must define an interface between the wanted network and its own environment. We introduce the abstract methods setActiveSensors and handleActiveMotors in NeuralNetSim class in order to force the user to define the own interface network-environment as soon as he implements a NeuralNetSim subclass that is not abstract.

- public abstract void **defaultRun**(Object xObject);

We suggest to define a default life cycle by implementing the abstract method defaultRun.

- private boolean **buildInAndOutNodes**(File ioFile);

private boolean **buildHiddenNodes**(File objFile);

private boolean **udateProperty**(File hbbFile);

private void **setType**();

private void **buildBrain**();

private void **updateNodeInput**(RecordNode rn, int xDelay);

A private method is accessible only within the class body in which the method is declared and is not inherited by subclasses. Public methods apply these private methods.

- public void **innerActivity**();

The value of both innerState and output field in all Node occurrences contained in myBrain is computed at present time. The ActiveNodes occurrence at index 0 in myHistory is updated, this ActiveNodes occurrence holds all Node occurrences representing the active network nodes at the present time.

- public void **nextTime**();

Any item in myHistory at index k is moved at index k+1 if and only if k is less than maxDelay. The item with the index maxDelay is erased. The value of myTime is increased by 1.

- public Connections **getTimeIndConnOf**(int index);

The Connections object containing the time independent connections starting from the network node with the specified index is returned.

- public Connections **getTimeDepConnOf**(int index);

The Connections object containing the time dependent connections starting from the network node with the specified index is returned.

- public int getCoupling **coefficientOf**(int toIndex, int fromIndex, int xDelay);

The coupling coefficient of the xDelay delayed connection going from the node with index fromIndex to the node with index toIndex is returned. The coupling coefficient is an integer value.

- public History **getHistory**();

The object myHistory is returned.

- public Brain **getBrain**();

The object myBrain is returned.

- public int **getMaxDelay**();

The value of maxDelay is returned.

- public int **getBrainSize**();

The size of myBrain is returned.

- public int **getHistorySize**();

The size of myHistory is returned.

- public ActiveNodes **getHistoryElementAt**(int i);

The item of myHistory at index i is returned.

- public RecordNodeVector **getActiveNodesAt**(int xDelay);

The RecordNodeVector object containing the Node occurrences representing all active network nodes at the time myTime-xDelay is returned.

- public RecordNodeVector **getActiveMotorsAt**(int xDelay);

The RecordNodeVector object containing the Node occurrences representing all active motor nodes at the time myTime-xDelay is returned.

- public RecordNodeVector **getActiveSensorsAt**(int xDelay);

The RecordNodeVector object containing the Node occurrences representing all active sensor nodes at the time myTime-xDelay is returned.

- public RecordNodeVector **getActiveHiddenAt**(int xDelay);

The RecordNodeVector object containing the Node occurrences representing all active hidden nodes at the time myTime-xDelay is returned.

- public NodeVector **getSensorNodes**();

The NodeVector containing the Node occurrences representing all sensor nodes is returned.

- public NodeVector **getMotorNodes**();

The NodeVector containing the Node occurrences representing all motor nodes is returned.

- public int **getCurrentTime**();

The present time is returned.

- public InfoConnection **findConnection**(int toIndex,int fromIndex, int xDelay);

The xDelay delayed connection going from the node with index fromIndex to the node with index toIndex is returned by the object InfoConnection.

- public NetworkVector **findConnections**(int toIndex,int fromIndex);

The connections going from the node with index fromIndex to the node with index toIndex are returned by a NetworkVector object. Any item of this NetworkVector is an InfoConnection object.

- public void **setActiveSensor**(int xIndex, float xOutput);

The Node occurrence with the index value equal to the specified xIndex is added in the active sensor node list, that is activeSensorNodes, and its output value become xOutput.

- public void **setActiveSensor**(String xName float, xOutput);

The sensor node with the specified name xName is added in the active sensor node list, that is activeSensorNodes, and its output value become xOutput.

- public void **setMaxDelay**(int x);

The new value of maxDelay is x if x is not greater than maxMaxDelay, otherwise is maxMaxDelay.

- public void **printActiveNodesAt**(int xDelay, File outFile);
- public void **printActiveNodesAt**(int xDelay, String nameFile);

The indices of all active nodes at the time myTime-xDelay are printed in an actual file, either the file named by outFile or the file named by the path name nameFile in the file system.

- public void **printActiveSensorsAt**(int xDelay, File outFile);
- public void **printActiveSensorsAt**(int xDelay, String nameFile);

The indices of all active sensor nodes at the time myTime-xDelay are printed in an actual file, either the file named by outFile or the file named by the path name nameFile in the file system.

- public void **printActiveMotorsAt**(int xDelay, File outFile);
- public void **printActiveMotorsAt**(int xDelay, String nameFile);

The indices of all active motor nodes at the time myTime-xDelay are printed in an actual file, either the file named by outFile or the file named by the path name nameFile in the file system.

- public void **printActiveHiddenAt**(int xDelay, File outFile);
- public void **printActiveHiddenAt**(int xDelay, String nameFile);

The indices of all active hidden nodes at the time myTime-xDelay are printed in an actual file, either the file named by outFile or the file named by the path name nameFile in the file system.

- public void **printConnections**(int xTo, int xFrom, File outFile);
- public void **printConnections**(int xTo, int xFrom, String nameFile);

The data of all connections going from the node with index xFrom to the node with index xTo are printed in an actual file, either the file named by outFile or the file named by the path name nameFile in the file system.

7.2.3.16. *The class it.na.cy.nnet.BoolHebbNet*

BoolHebbNet is an abstract class. One must define a not abstract subclass of BoolHebbNet. If BNetwork is the name of a suitable not abstract BoolHebbNet subclass and myBoolHebbNet is an instance of BNetwork then the myBoolHebbNet life cycle could be the following:

```
1)  MyBoolHebbNet. setActiveSensors()      //Abstract method. It must be previ-
                                           //ously implemented.
2)  MyBoolHebbNet.innerActivity()
3)  MyBoolHebbNet. handleActiveMotors()    //Abstract method. It must be previ-
                                           //ously implemented.
4)  MyBoolHebbNet.nextTime()
```

One must also define the abstract method defaultRun().

```
abstract public class BoolHebbNetextends NeuralNetSim
{
//Public Members
public  float alfa,beta,gamma,delta;
public final static float alfaDefault=(float)2;
public final static float betaDefault=(float)0.6;
public final static float gammaDefault=(float)0.8;
```

public final static float deltaDefault=(float)0.01;
public NetworkVector changedChkd;
public NetworkVector frozenChkd;
//Constructors
public **BoolHebbNet**(String xNetworkName, float xAlfa, float xBeta, float xGamma, float xDelta);
public **BoolHebbNet**(String xNetworkName);
//Private or Protected Methods
private void **HebbIncrementOn**(RecordNodeVector xHebbInVec, int xDelay);
private void **HebbDecrementAt**(int xDelay);
private void **HebbNaturalDecrAt**(int xType, int xDelay);
protected abstract void **computeNodeOutput**(Node xNode);
protected abstract void **computeNodeInput**(Node toNode, float fromToWeigth, float fromOutput);
//Public Methods
public final void **HebbRule**(int maxHebbDelay);
public final void **HebbRule**(int maxHebbDelay,float xAlfa,float xBeta,float xGamma,float xDelta);
public void **HebbRuleLP**(int maxHebbDelay);
public final void **HebbRuleLP**(int maxHebbDelay,float xAlfa,float xBeta,float xGamma,float xDelta);
public void **wideHebbRule**(int maxHebbDelay)
public final void wideHebbRule(int maxHebbDelay,float xAlfa,float xBeta,float xGamma,float xDelta);
public void **wideHebbRuleLP**(int maxHebbDelay);
public final void **wideHebbRuleLP**(int maxHebbDelay,float xAlfa,float xBeta,float xGamma,float xDelta);
public final boolean **setSensorsByTwo**(int numBit, int code);
public final boolean **setSensorsByTwo**(int code);
public final boolean **setSensorsByOne**(int numBit, int code);
public final boolean **setSensorsByOne**(int code);

- public float alfa, beta, gamma, delta;

The fields hold float numbers used in the final methods HebbRule, HebbRuleLP and so on.

- public static final float alfaDefault=2;
  public static final float betaDefault=0.6;
  public static final float gammaDefault=0.8;
  public static final float deltaDefault=0.01;

These are constants representing default values for the fields alfa, beta, gamma, delta.

- public **BoolHebbNet**(String xNetworkName, float xAlfa, float xBeta, float xGamma, float xDelta);

The BoolHebbNet constructor calls the constructor of its super class. Accordingly, this constructor may be used only by a not abstract BoolHebbNetsubclass. If NNetwork is the name of the latter subclass, then the constructor initializes a newly created NNetwork by opening connections to actual files, ie, to the files named by the path names xNetworkName.io, xNetworkName.hbb and xNetworkName.obj in the file system. The wanted network, namely the one to simulate, must be described by a suitable low level language, see the chapter 7.3., in the above listed files. The fields alfa, beta, gamma and delta assume the values xAlfa, xBeta, xGamma and xDelta, respectively

- public **BoolHebbNet**(String xNetworkName);

The BoolHebbNet constructor calls the constructor of its super class. Accordingly, this constructor may be used only by a not abstract BoolHebbNet subclass. If NNetwork is the name of the said subclass then the constructor initializes a newly created NNetwork by opening connections to actual files, ie, to the files named by the path names xNetworkName.io, xNetworkName.hbb and xNetworkName.obj in the file system. The wanted network, namely the one to simulate, must be described by a suitable low level language, see the chapter 7.3, in the above listed files. The fields alfa, beta, gamma and delta assume the values alfaDefault, betaDefault, gammaDefault and deltaDefault, respectively.

- private void **HebbIncrementOn**(RecordNodeVector xHebbInVec, int xDelay);
- private void **HebbDecrementAt**(int xDelay);
- private void **HebbNaturalDecrAt**(int xType, int xDelay);

A private method is accessible only within the class body in which the method is declared and is not inherited by its subclasses. Public methods apply these private methods.

- protected final void **computeNodeInput**(Node toNode, float fromToWeigth, float fromOutput);

The abstract method computeNodeInput declared in NeuralNetSim is implemented here. Each node input is updated in the following way:

```
protected final void computeNodeInput(Node toNode, float fromToWeigth, float
fromOutput)
    {
     toNode.input=toNode.input+fromToWeigth*fromOutput;
    }
```

- protected final void **computeNodeOutput**(Node xNode);

The abstract method computeNodeOutput declared in NeuralNetSim is implemented here. Each node computes its output in the following way:

```
protected final void computeNodeOutput(Node xNode)
    {
     if (xNode.input >= fireThreshold) xNode.output=NetworkConstants.YES_
     FIRE;
     else xNode.output=NetworkConstants.NO_FIRE;
     xNode.input=0;
    }
```

- public final void **HebbRule**(int maxHebbDelay);
- public final void **HebbRule**(int maxHebbDelay, float xAlfa, float xBeta, float xGamma, float xDelta);
- public final void **wideHebbRule**(int maxHebbDelay);
- public final void **wideHebbRule**(int maxHebbDelay, float xAlfa, float xBeta, float xGamma, float xDelta);

The coupling coefficient C_{hkd} associated to a connection going from the node with index j and property = 2, 3 to the node with index i and property = 1, 3 is a time dependent coupling coefficient. The time dependent coupling coefficients, $C_{hkd}(t)$, are regulated by a Hebbian rule, see the chapter 4. Here, some specific Hebbian learning rules are defined.

7.2.3.17. *The class it.na.cy.nnet.ReadStream.*

The class *ReadStream* is a subclass of StreamTokenizer. Accordingly, it takes an input stream and parses it into "tokens", allowing the tokens to be read one at a time. ReadStream has more input operations than its super class.

```
package it.na.cy.nnet;

import java.io.*;
import java.util.*;
public class ReadStream extends StreamTokenizer
{
//Constructor
public  ReadStream(InputStream in);
//Public methods
public Float readNextNumber();
public String readNextWord();
public NetworkVector readLineNumbers();
public NetworkVector readLineWords();
}
```

- public ReadStream(InputStream in);

The constructor initializes a newly created ReadStream calling the constructor of its super class. Accordingly, the constructor saves its argument, the input stream in, for later use. The ReadStream is initialized to the following default state: all byte values 'A' through 'Z', 'a' through 'z', and 0xA0 through 0xFF are considered to be alphabetic; all byte values 0x00 through 0x20 are considered to be blank; '/' is a comment character; single quote '\'' and float quote '"' are string quote characters; numbers are parsed; end of line is significant; // comments and /* comments are not recognized.

- public Float readNextNumber();

Starting at the current file pointer, any character is skipped, except if a numeric character is encountered. If a number is recognized, it is returned as a Float object. If the EOF is encountered, the null pointer is returned.

- public String readNextWord();

Starting at the current file pointer, any character is skipped, except if the character ' " ' is encountered. In this case an alphanumeric string is parsed, stopping at the last character before the next occurrence of ' " '. The recognized alphanumeric string between the first and second occurrence of ' " ' is returned as a String object. If the EOF is encountered the null pointer is returned.

- public NetworkVector readLineNumbers();

Starting at the current file pointer, any character is parsed, stopping at the last character before the next occurrence of EOL. All the number occurrences are recognized and they are returned as a NetworkVector object. That is, any item of the returned NetworkVector object is a Float object holding a recognized number.

- public NetworkVector readLineWords();

Starting at the current file pointer, any character is parsed, stopping at the last character before the next occurrence of EOL. All the string occurrences are recognized and they are returned as a NetworkVector object. That is, any item of the returned NetworkVector object is a String object holding a recognized number. Here, a string occurrence is an alphanumeric string included between two occurrence of ' " '.

7.2.3.18. *The class it.na.cy.nnet.ReadPrintRAF*

The class *ReadPrintRAF* is a subclass of RandomAccessFile. ReadPrintRAF allows more output operations than its super class. ReadPrintRAF holds methods in order to write strings, spaces and the new line

```
package it.na.cy.nnet;
```

```
import java.io.RandomAccessFile;
import java.io.File;
import java.io.IOException;
public class ReadPrintRAF extends RandomAccessFile
{
//Private members
private static final int LF=10;
private static final int CR=13;
private static final int SPACE=32;
//Constructors
public ReadPrintRAF(String path, String mode) throws SecurityException, IO-
Exception, IllegalArgumentException
public ReadPrintRAF(File file, String mode) throws SecurityException, IOEx-
ception,
IllegalArgumentException
//Public methods
public void newLine() throws IOException
public void space() throws IOException
public void println(String s) throws IOException
public void printString(String s) throws IOException
}
```

- private static final int LF=10;
  private static final int CR=13;
  private static final int SPACE=32;

Constants used in the methods of this class.

- public **ReadPrintRAF**(String path, String mode) throws SecurityException, IOException, IllegalArgumentException

The constructor initializes a newly created ReadPrintRAF calling the constructor of the super class. So, it opens a connection to an actual file, the file named by the path name path in the file system. A new FileDescriptor object is created to represent this file connection. First, if there is a security manager, its checkRead method is called with the path argument as its argument. Next, if the mode is "rw" and there is a security manager, its checkWrite method is called with the path argument as its argument. If the mode is "rw", then the file may be both read and written. If the mode is "r", then the file may be read but may not be written (every write method for this object will simply throw an IOException). If the mode is not "r" or "rw", then this constructor throws an IllegalArgumentException.

- public **ReadPrintRAF**(File file, String mode) throws SecurityException, IOException, IllegalArgumentException

This constructor initializes a newly created ReadPrintRAF calling the constructor of the super class. So, it opens a connection to an actual file, the file named by file in the file system. A new FileDescriptor object is created to represent this file connection. First, if there is a security manager, its checkRead method is called with the path argument as its argument. Next, if the mode is "rw" and there is a security manager, its checkWrite method is called with the path argument as its argument. If the mode is "rw", then the file may be both read and written. If the mode is "r", then the file may be read but may not be written (every write method for this object will simply throw an IOException). If the mode is not "r" or "rw", then this constructor throws an IllegalArgumentException.

- public void **newLine**() throws IOException

The bytes corresponding to the new line are written to the random access file, starting at the current file pointer.

- public void **space**() throws IOException

The bytes corresponding to the space are written to the random access file, starting at the current file pointer.

public void println(String s) throws IOException

First, the string held in the String s is written to the random access file, starting at the current file pointer. Next, the new line is written to the random access file

- public void printString(String s) throws IOException

The string held in String s is written to the random access file, starting at the current file pointer.

7.3. Evaluating the average run-time.

The package it.na.cy.nnet allows us to implement many different network simulators such that the simulated network activity is computed considering only the subset of all active nodes instead of all network nodes. In the following we call *mySimulator* the particular network simulated. We discuss its realization and attempt its evaluation.

The simulator main structures, see the Figure 7. 2., are:

a) N node objects, where N is the number of network nodes. Each node is represented by a six fields record:

a.1) The field index holds an integer value. It is the value of one of the network node indices. For simplicity sake, we will call node i a node with the field index set to i

a.2) The field input holds the input of the network node with the index field set to index.

a.3) The field output holds the output of the node with the index field set to index.

a.4) The field innerState holds the state of the node with the index field set to index.

a.5) The field timeIndConnections holds the data about all the time independent coupling coefficients associated to the connections starting from the network node with the index field set to index.

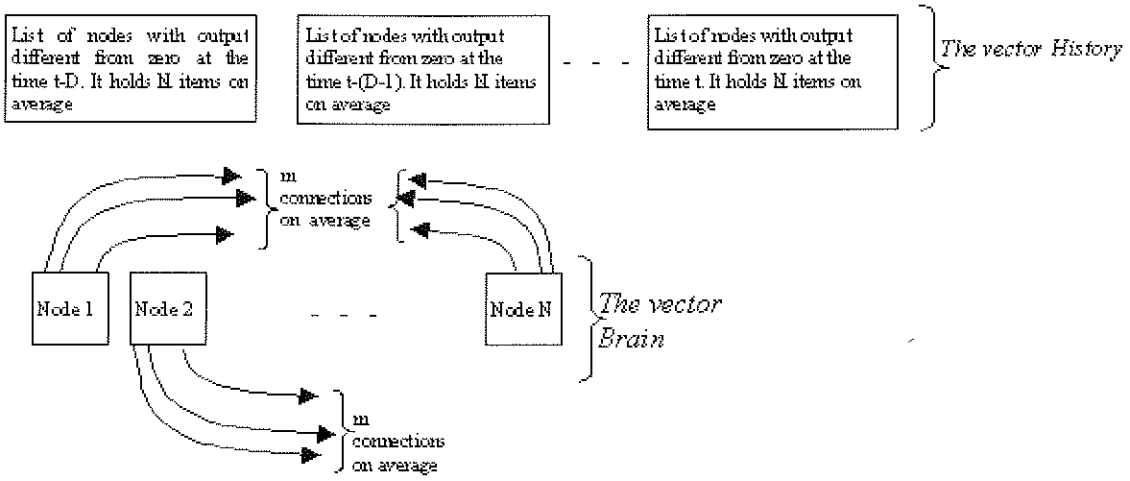


Fig. 7.2. *Main components of a simulator implemented by the package it.na.cy.nnet.* The simulator components are a representation of the desired neural network. The vector history has D + 1 components, where D is the maximum delay. At each time t, historyf k], i.e., the history element with the index k, holds a list of pointers to the node objects with the field output different from zero at the time t - k. The vector brain holds N elements. At any time, brainf k], i.e., the brain element with the index k, holds the node object k. Each node is represented by a six fields record: index, input, output, innerState, timeIndConnections and timeDepConnections, see text. The fields timeIndConnections and timeDepConnections hold the data about all the time independent and time dependent coupling coefficients associated to the connections starting from the network node with the index field set to index. They hold the data about m connections, on average.

a.6) The field timeDepConnections holds the data about all the time dependent coupling coefficients associated to the connections starting from the network node with the index field set to index.

b) The vector history holds D + 1 elements, where D is the maximum delay. At each time t, history[k], i.e., the history of the element with the index k, holds a list of pointers to the node objects having the field output different from zero at the time t - k.

c) The vector brain holds N elements. At any time, brain[k], i.e., the brain element with the index k, holds the node object k.

d) The initially empty list possActNodes holding integer numbers.

e) The variable $t \in \mathcal{N}$ represents the neural network time.

The basic steps of the simulator mySimulator are:

1) In the first step it sets t = 0.

2) For d = D, D-1, ...,1, first, it goes through the list held in history[d]. And, for each node object k pointed to an element of history[d], it considers the coupling coefficients C_{hkd}. Second, it updates the input of the node objects h and it inserts each index h at the top of the list possActNodes, if h is not present in possActNodes.

3) It goes through the list possActNodes. For each index h belonging to possActNodes, it reads both the field input and the field innerState of the node object component held in brain[i], then it computes both the new value of innerState field and the new value of the field output and updates brain[i]. If the output value is different from zero, a pointer to the node object i is inserted in history[0], whenever the node i is not present. Accordingly, history[0] holds all the node object with value of the field output different from zero at the time t.

It sets possActNodes equal to null.

It sets t equal to t+1 and, then, it shifts the vector history to the left, that is:

history[d]=history[d-1] for d=D, D-1, ...1.

history[0] is set equal to null.

It returns to the step 2.

We are going to estimate the average time Dt required to compute the steps from 2. to 6. for the simulator mySimulator.

Let us call:

- dt_1: the average time to compute the value of the field output of a node object once the value of its field input has been computed.
- dt_2: the average time to compute the value of the field innerState of a node object once the value of its field input has been computed.
- dt_3: the average time to update the value of field input of the node object k for each C_{hkd} held in the fields timeIndConnections and timeDepConnections of the node object i with value of its field output different from zero.
- dt_4: the average time to erase or insert an item in a list.
- $\underline{N}$: the average number of node objects with the value of the field output different from zero.
- $\underline{m}(d)$: the average number of time dependent and time independent coupling coefficients, with delay d, holding in the fields timeIndConnections and timeDepConnections.

We set $\underline{M} = \underline{m}(D) + \underline{m}(D-1) + ... + \underline{m}(1)$, then:

$Dt \cong D \cdot \underline{N} \cdot \underline{m} \cdot (dt_1 + dt_2 + dt_3) + D \cdot \underline{N} \cdot \underline{m} \cdot dt_4 + D \cdot dt_4$ $\quad Dt \cong D \cdot \underline{N} \cdot \underline{m} \cdot (dt_1 + dt_2 + dt_3 + dt_4) + D \cdot dt_4$

$$Dt \cong \underbrace{D\cdot \underline{N}\cdot \underline{m}\cdot dt_3}_{\text{Step 2.}} + \underbrace{D\cdot \underline{N}\cdot \underline{m}\cdot (dt_1 + dt_2)}_{\text{Step 3.}} + \underbrace{(D + D\cdot \underline{N}\cdot \underline{m})\cdot dt_4}_{\text{Steps 4.- 6.}}$$

Because $D\cdot dt_4$ is negligible, we have: $Dt \cong D\cdot \underline{N}\cdot \underline{m}\cdot (dt_1+dt_2+ dt_3+ dt_4)$

Let $\underline{Dt}$ be equal to $dt_1+dt_2+ dt_3+ dt_4$, then we have $Dt \cong D\cdot \underline{N}\cdot \underline{m}\cdot \underline{Dt}$

By contrast, to visit all the nodes in order to compute their outputs and their states, the average time Dt^* is required: $Dt^* \cong N\cdot \underline{M}\cdot dt_3 + N\cdot (dt_1 + dt_2) \Rightarrow Dt^* \cong N\cdot (\underline{M}\cdot dt_3 + dt_1 + dt_2)$

If $\underline{M}\cdot dt_3 >> (dt_1+ dt_2)$, then $Dt^* \cong N\cdot \underline{M}\cdot dt_3$

Accordingly, if $\underline{Dt} \cong dt_3$ and if we want $Dt< Dt^*$ it must be $D\cdot \underline{N}\cdot \underline{m} < N\cdot \underline{M} \Rightarrow D\cdot \underline{N} < N$.

Hence, the simulator mySimulator performance depends on the number of active network nodes. If the total number of the network nodes is one order of magnitude greater than the product of the maximum delay times the average number of active nodes, then, mySimulator speed is greater than the speed of a simulator computing the state of all the net nodes every tau.

7.4. The low level language for neural networks LLNN.

We present the *Low level Language for Neural Networks LLNN* in order to describe instances of the neural network paradigms having the features $\Lambda 1$,..., $\Lambda 0$, see the chapter 7.1.. To define the language, first, we define the syntax, second, we define the semantics.

7.4.1. *The Syntax.*

The language LLNN is composed by the following primitive and defined elements.
The primitive elements are:

- -1, 0, 1, 2, 3, 4, 5
- <sp>::= one or more blanks.
- <sp0>::= zero or more blanks.
- <nl>::= one ore more new line.
- <indexFrom>::= integer number.
- <indexTo>::= integer number.
- <coupling coefficient>::= integer number.
- <delay>::= integer number.
- <thr>::= integer number.
- <hThr>::=integer number.

The defined elements are:

- <endConn> ::= <sp0>-1<sp>-1<sp>-1<sp0><nl>
- <endNetwork> ::= <sp0>-1<sp>-1<sp>-1<sp>-1<sp>-1<sp0><nl>
- <position>::={0/5/10}.
- <property>::={0/1/2/3/4}.
- <nodeTo>::=

<sp0><indexTo><sp><thr><sp><position><sp><property><sp><hThr><sp0><nl>

- <connDescription>::=

<sp0><indexFrom><sp><delay><sp><coupling coefficient><sp0><nl>

- <connections>::={< connDescription >/< connDescription ><connections>}
- <connGroup>::={<endConn>/<connections><endConn>}
- <nodeDescription>::=<nodeTo><connGroup>
- <nodeGroup>::= {<nodeDescription>/<nodeDescription><nodeGroup>}
- <networkDescription>::= {<endNetwork>/<nodeGroup><endNetwork>}

7.4.2. *The Semantics.*

The semantic link to the neural network model having the features A1,..., A6, see the chapter 7.1., is:

- <indexFrom> represents the index of a node sending the connections.
- <indexTo> represents the index of a node receiving the connections.
- <coupling coefficient> represents the coupling coefficient associated to a connection.
- <delay> represents the delay associated to a connection.
- <position> represents the position of a node: sensor, inner, motor.
- <property> represents the property associated to a node.
- <thr> represents the fire threshold of a node.
- <hThr> represents a second threshold.
- <nodeTo>::=

<sp0><indexTo><sp><thr><sp><position><sp><property><sp><hThr><sp0><nl>

represents the node, receiving connections, with the following features: index equal to <indexTo>, fire threshold equal to <thr>, position equal to <position>, property equal to <property> and Hebbian threshold equal to <hThr>. E. g., the code in the Table 7.1. is an instance of <nodeTo>

TABLE 7. 1.

-	h	T	p	tp	tdt
row 1	300	10	5	1	0

Table 7. 1. *An instance of <nodeTo>.* h is the index of a node, T the node threshold, p the node position, tp the tag property of the node, tdt the time dependent threshold of the node.

- <connDescription>::=

<sp0><indexFrom><sp><coupling coefficient><sp><delay><sp0><nl>

represents one <delay>-channel going from the node with index equal to <indexFrom> to one network node and having coupling coefficient equal to <coupling coefficient>.

- <connections>::={< connDescription >/< connDescription ><connections>}

represents a set of connections going from some nodes to one network node.

- <connGroup>::={<endConn>/<connections><endConn>}

represents all the connections, described following the syntax of <connections>, going from some nodes to one node.

- <nodeDescription>::=<nodeTo><connGroup>

represents the node <nodeTo> and all the connections going to it. E.g., the code in the Table 7.2. is an instance of <nodeDescription>:

TABLE 7. 2.

-	h	T	p	tp	tdt	k	d	cc	&
row 1	300	10	5	1	0	-	-	-	-
row 2	-	-	-	-	-	200	1	5	-
row 3	-	-	-	-	-	100	2	6	-
row 4	-	-	-	-	-	-	-	-	-1

Table 7.2. *An instance of <nodeDescription>.* We call h the index of a node receiving the connections; T the node threshold; p the node position; tps the tag property of the node; tdt the time dependent threshold of the node; k the index of the node sending the connection to h; cc the coupling coefficient associated to the connection; d the delay associated to the connection; & a special symbol. Accordingly, on the row 1 the number 300 is the index of a node, 10 is its threshold, 5 represents its position, 1 its property and 0 is its second threshold; on the row 2 the numbers 200, 1 and 5 describe a 1-delayed channel, or connection, going from the node of index 200 to the node of index 300 with an associated coupling coefficient equal to 5; on the row 3 the numbers 100, 2 and 6 describe a 2-delayed channel going from the node of index 100 to the node of index 300 and with an associated coupling coefficient equal to 6; on the row 4 the number –1 marks the end of the connections going to the node with index 300.

- <networkDescription> represent the descriptions of an istance of a neural network.

E. g., if we write in a file the code shown in the Table 7.3. we describe the network in the Figure 7.3..

TABLE 7. 3.

-	h	T	p	tp	tdt	K	d	cc	&
row 1	0	10	0	0	0	-	-	-	-
row 2	-	-	-	-	-	-	-	-	-1
row 3	1	10	0	0	0	-	-	-	-
row 4	-	-	-	-	-	-	-	-	-1
row 5	2	10	5	0	0	-	-	-	-
row 6	-	-	-	-	-	0	1	6	-
row 7	-	-	-	-	-	1	1	6	-

row 8	-	-	-	-	-	-	-	-	-1
row 9	3	10	10	0	0	-	-	-	-
row 10	-	-	-	-	-	2	1	12	-
row 11	-	-	-	-	-	-	-	-	-1
row 12	-	-	-	-	-	-	-	-	-1

Table 7.3. *An instance of <networkDescription>.* We call h the index of a node receiving the connections; T the node threshold; p the node position; tp the tag property of the node; tdt the time dependent threshold of the node; k the index of the node sending the connection to h; cc the coupling coefficient associated to the connection; d the delay associated to the connection; & are some special symbols. The Table entries are a description of the neural network depicted in the Figure 7. 3.

If we write in a file an instance of <networkDescription>, we obtain a description of a neural network. This file can be read and interpreted by an instance of a suitable subclass of the NeuralNetSim class or of the BoolHebbNet class belonging to the package it.na.cy.nnet.

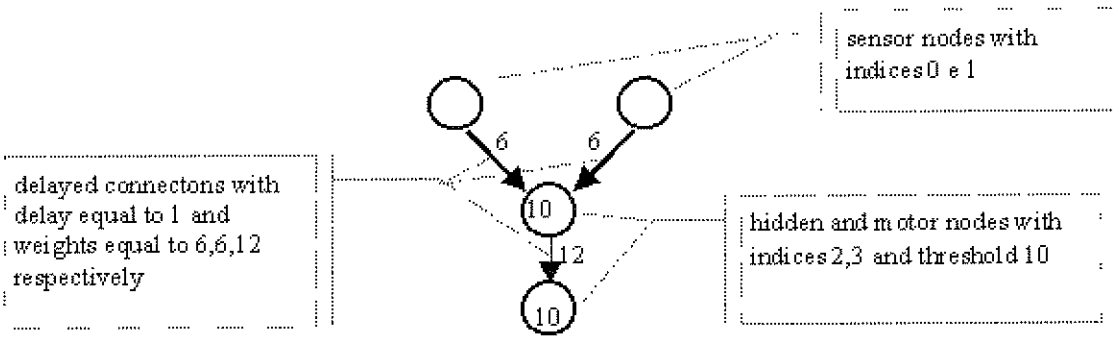


Fig. 7.3. *An instance of a neural network.* The circles represent the network nodes. The arrows represent the network connections.

7.5. Some results.

Using the Java package it.na.cy.nnet we have developed BoolHebbNet: a simulator of Boolean Neural Networks with time dependent coupling coefficients. Actually, in BoolHebbNet the coupling coefficients have a time dependency regulated by a Hebbian rule, as defined in the chapters 2. and 4.. We have simulated a General Adaptable Boolean Neural Network (GABNN), as defined in the chapter 5., using both BoolHebbNet and the high level language CONNET, see [25], in order to study the neural network–environment interactions. We have obtained a GABNN composed by almost $3*10^3$ nodes: actually, by 2^3 primitives and by 2^6-2^3 general assemblies. The GABNN simulation has been tested under some environment constraints. To introduce the latter, let us set some definitions:

1) Let m be the length of the network input subvector: at the time t = n*tau we call *input sequence* **I(k)** the sequence in which the i-th element is equal to the network

input subvector at the time i*tau, with $k \leq n$ and $i \leq k$. That is, **I(k)** represents the sequence of the first k input subvectors going from the past toward the present.

2) We define as subsequence of **I(k),** with base q and length r, the sequence $\mathbf{I_q}$® having the i-th element equal to the (q + i)-th element of **I(k)**, with $q \geq 0$, $r \geq 1$ and $1 \leq q + r \leq k$.

3) Recalling the definitions given in the chapter 5. 6. 14 and said A the identifier of a general assembly and A', A" the identifiers of either general assemblies or primitives, we define the *example-subsequence* E(A, A', A", d', d"), a subsequence $\mathbf{I_q(D_E)}$ of **I(n)** satisfying the conditions:

3.1) The first element of $\mathbf{I_q(r)}$ codes the identifier A, or for short codes A.

3.2) At least one element of $\mathbf{I_{q'}(D')}$ codes A', with q' = q + S, and the first element coding A' is at the place d', where S and D' are constants and $1 \leq d' < D'$.

3.4) At least one element of $\mathbf{I_{q''}(D'')}$ codes A", with q" = q + D' + 2*S and S and D" are constants, and the first element coding A" is at the place d", with $1 \leq d'' < D''$.

4) We call *length of an example-subsequence* the constant $D_E = 2*S + D' + D''$. That is, an example E(A, A', A", d', d") is a subsequence of **I(n)** that includes the identifier A as first element, the identifier A' as (S + d')-th element and the identifier A" as (D' + 2*S + d")-th element.

5) We call *null-example* a subsequence $\mathbf{I_q(D_E)}$ of **I(n)** with no element coding either a general assembly or a primitive identifier.

6) An example-subsequence E = (A, A', A", d1, d2) is learned as soon as both the following two assertion are true:

6.1) the d1-delayed Hebbian connection going from the first Hebb-out node of the general assembly A to the Hebb-in node of A' is frozen.

6.2) the d2-delayed Hebbian connection going from the second Hebb-out node of the general assembly A to the Hebb-in node of A" is frozen.

7) If the example-subsequence E = (A, A', A", d1, d2) has been learned at the time $k*D_E*tau$ and it has not yet been learned at the time $(k-1)*D_E*tau$, then, we will say: *the example-subsequence E has been learned at the k-th input subsequence.*

The GABNN simulation has been tested under the condition c1:

Condition c1:

At the time $t_n = n*D_E*tau$, the subsequence $\mathbf{I_q(D_E)}$ of $\mathbf{I(n*D_E)}$ is either an example-subsequence E(A, A', A", d', d") or a null-example, with $q = i*D_E$ and i = 0, 1, ..., n-1.

In other words, $\mathbf{I(n*D_E)}$ can be decomposed in a sequence of n subsequences $\mathbf{I_q(D_E)}$, each subsequence coinciding with an example-subsequence or a null-example:

$$\mathbf{I(n*D_E) = [I_0(D_E), I_{D_E}(D_E), ..., I_{(n-1)D_E}(D_E)]}$$

Given the condition c1, we call *k-th input subsequence* at the time t, for $t \geq k*D_E*tau$, the k-th subsequence $\mathbf{I_{(k-1)D_E}(D_E)}$ of $\mathbf{I(n*D_E)}$. E.g., the n-th input subsequence at the time $t_n = n*D_E*tau$ is the last subsequence $\mathbf{I_{(n-1)\,D_E}\,(D_E)}$ of $\mathbf{I\,(n*D_E)}$.

We call P1, P2, P3, P4 four primitive identifiers and A1, A2, A3, A4, A5 five general assembly identifiers.

Then, for d1,d2 constants, given the following events:

- H ≡ "the n-th input subsequence at the time $t_n = n*D_E*tau$ is not a null-example"

- $H_1 \equiv$ "the n-th input subsequence at the time $t_n = n^* D_E^* tau$ is the example-subsequence E1 = (A1, P1, P2, d1, d2)"
- $H_2 \equiv$ "the n-th input subsequence at the time $t_n = n^* D_E^* tau$ is the example-subsequence E2=(A2, P3, P4, d1, d2)"
- $H_3 \equiv$ "the n-th input subsequence at the time $t_n = n^* D_E^* tau$ is the example-subsequence E3 = (A3, A1, A2, d1, d2)"
- $H_4 \equiv$ "the n-th input subsequence at the time $t_n = n^* D_E^* tau$ is the example-subsequence E4 = (A4, A2, A1, d1, d2)"
- $H_5 \equiv$ "the n-th input subsequence at the time $t_n = n^* D_E^* tau$ is the example-subsequence E5 = (A5, A3, A4, d1, d2)"
- $H_6 \equiv$ "the n-th input subsequence at the time $t_n = n^* D_E^* tau$ is an example-subsequence different from E1, E2, E3 and the null-example".
- $H_7 \equiv$ "the n-th input subsequence at the time $t_n = n^* D_E^* tau$ is an example-subsequence different from E1, E2, E3, E4, E5 and the null-example".

We have analyzed the behavior of the GABNN in the following cases:

C1. $P(H) = 0.9$, $P(H_1/H) = P(H_2/H) = P(H_3/H) = 0.10$ and $P(H_6/H) = 0.90$
C2. $P(H) = 0.9$, $P(H_1/H) = P(H_2/H) = P(H_3/H) = 0.15$ and $P(H_6/H) = 0.85$
C3. $P(H) = 0.9$, $P(H_1/H) = P(H_2/H) = P(H_3/H) = 0.20$ and $P(H_6/H) = 0.80$
C4. $P(H) = 0.9$, $P(H_1/H) = P(H_2/H) = P(H_3/H) = P(H_4/H) = P(H_5/H) = 0.10$ and $P(H_7/H) = 0.90$
C5. $P(H) = 0.9$, $P(H_1/H) = P(H_2/H) = P(H_3/H) = P(H_4/H) = P(H_5/H) = 0.15$ and $P(H_7/H) = 0.85$
C6. $P(H) = 0.9$, $P(H_1/H) = P(H_2/H) = P(H_3/H) = P(H_4/H) = P(H_5/H) = 0.20$ and $P(H_7/H) = 0.80$

where P(x) is the probability of the event x and P(x/y) is the probability of the event x subordinated on the event y.

For simplicity sake, in the following we shall write (A1, P1, P2), (A2, P3, P4), ..., (A5, A3, A4) instead of (A1, P1, P2, d1, d2), (A2, P3, P4, d1, d2), ...(A5, A3, A4, d1, d2). Initially we suppose each network input already associated to one general assembly, ie, we suppose each network input to be one general assembly identifier, see the chapter 5. 3. 1.. In the chapter 7. 5. 3., we shall test the subnet controlling the association of a general assembly with its own identifier.

7.5.1. *The C1, C2, C3 cases.*

As above said the network receives an input sequence regulated with the condition c1. In the case C1 we have the following constrains $P(H) = 0.9$, $P(H_1/H) = P(H_2/H) = P(H_3/H) = 0.10$ and $P(H_6/H) = 0.90$, in the case C2 we have $P(H) = 0.9$, $P(H_1/H) = P(H_2/H) = P(H_3/H) = 0.15$ and $P(H_6/H) = 0.85$, and in the case C3 we have $P(H) = 0.9$, $P(H_1/H) = P(H_2/H) = P(H_3/H) = 0.20$ and $P(H_6/H) = 0.80$.

In each case let us consider the following scenario: as soon as the example-subsequences E1, E2 and E3 are learned as the i_1-th, j_1-th and k_1-th input subsequence, respectively, we "restart the system", that is we set the time t equal to zero, the nodes outputs equal to zero and the Hebbian coefficients equal to zero. Then, as soon as the example-subsequences E1, E2 and E3 are learned as the i_2-th, j_2-th and k_2-th input subsequence, respectively, we "restart the system" again. This procedure is repeated indefinitely.

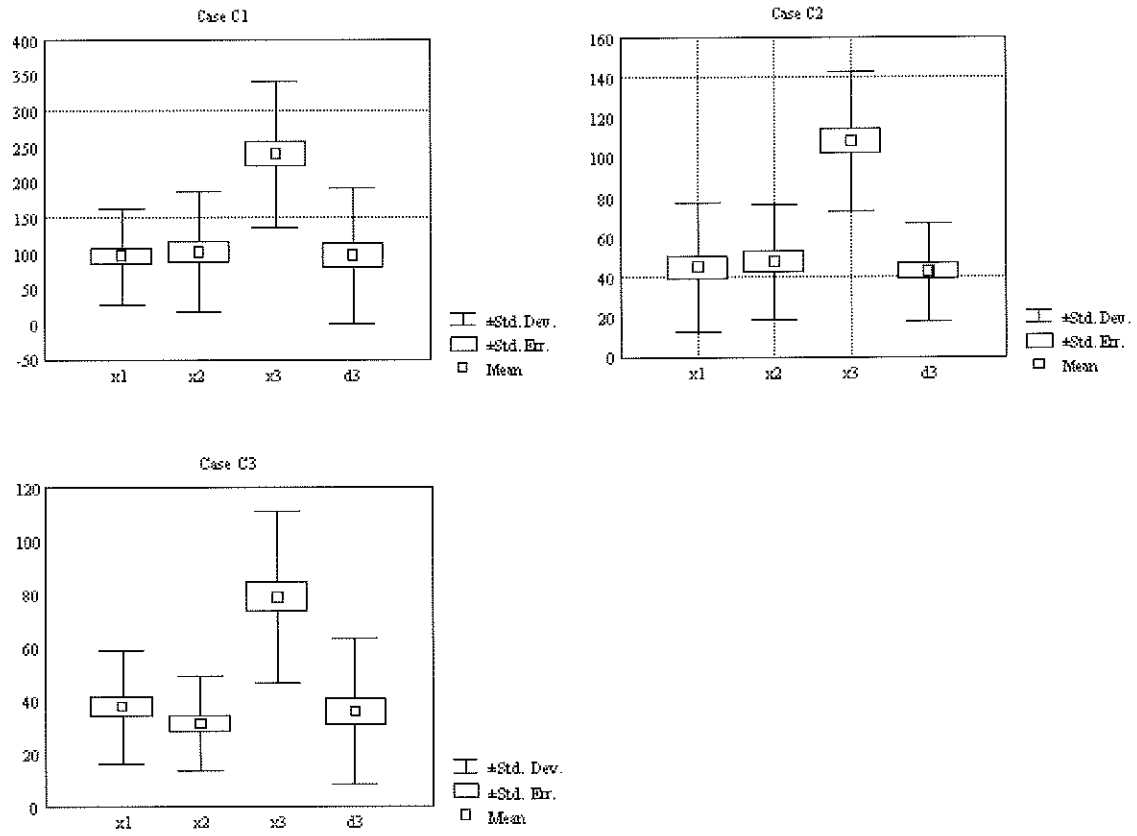


Fig. 7.4. *The sample mean and the standard deviation of the variables x1, x2,x3,d3 in the cases C1, C2 and C3.* The variable d3 represents the shift of the E3 learning with respect to the E1 and the E2 learning.

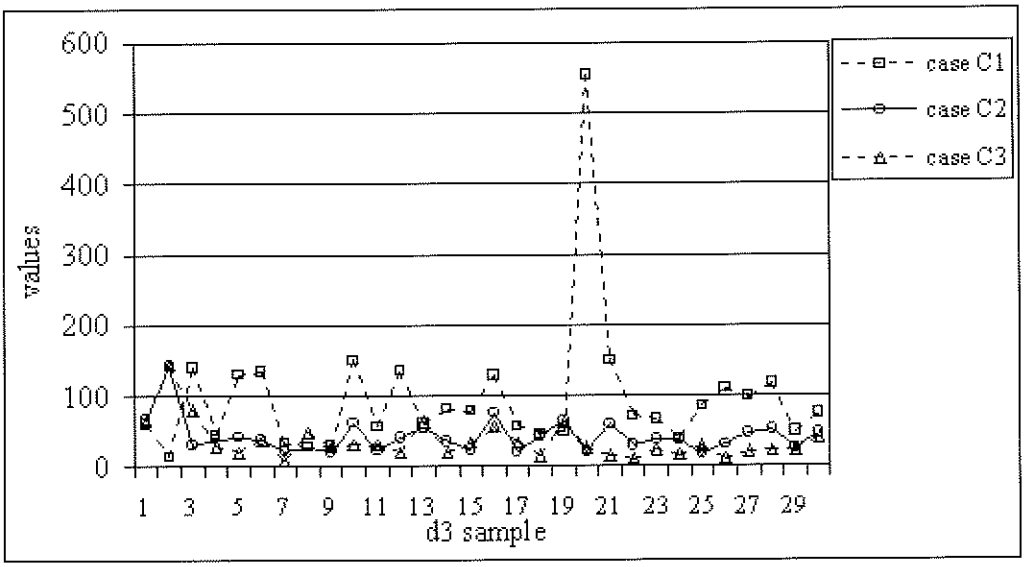


Fig. 7.5. *The d3 values in the cases C1, C2 and C3.* The variable d3 represents the shift of the E3 learning in respect to the E1 and the E2 learning. The d3 values are always greater than zero, that is, the example-subsequence E3 is always learned after both E1 and E2 have been learned.

Let us consider the variable x1 defined on the set $\{i_1, i_2, i_3, ...\}$, the variable x2 defined on the set $\{j_1, j_2, j_3, ...\}$, the variable x3 defined on the set $\{k_1, k_2, k_3, ...\}$, the variable d3 defined on the set $\{ k_1\text{-}\max(i_1, j_1), k_2\text{-}\max(i_2, j_2), ...\}$ and the variable d12 defined on the

set $\{i_1\text{-}j_1, i_2\text{-}j_2, ...\}$. The variable d3 represents the shift of the E3 learning with respect to the E1 and E2 learning, the variable d12 represents the shift of the E1 learning with respect to the E2 learning. In each case C1, C2 and C3, we have taken a sample of 30 elements of the variables x1, x2, x3, d12, d3 and we have computed the sample mean and the sample standard deviation of the variables x1, x2, x3, d12, d3, see the Table 7.4. and the Figure 7.4..

TABLE 7. 4.

-	**CASE C1**		**CASE C2**		**CASE C3**	
	Mean	Std.Dev.	Mean	Std.Dev.	Mean	Std.Dev.
x1	**95.4**	**68.2**	44.9	32.5	37.7	21.2
x2	**101.7**	**85.3**	47.8	28.8	31.5	17.7
x3	**239.0**	**102.6**	108.0	35.2	79.1	32.5
d3	**96.1**	**95.5**	42.7	24.7	35.9	27.2
d12	**-6.2**	**120.3**	-2.9	47.5	6.2	18.5

Table 7.4. *The mean and the standard deviation of the variables x1, x2, x3, d3 and d12 in the cases C1, C2 and C3.* In each case C1, C2 and C3 we have taken a sample of 30 elements of the variables x1, x2, x3, d12, d3, see text, and we have computed the sample mean and the sample standard deviations. The variable d3 represents the shift of the E3 learning in respect to the E1 and the E2 learning, the variable d12 represents the shift of the E1 learning in respect to the E2 learning.

The example E3 - (A3, A1, A2) can be learned only when the examples E1, E2 have been learned already, see the chapter 5.6.5.1.. Accordingly, the variable x3 have to assume, always, a value greater than both the values assumed by x1 and x2 in each repetition, that is, the variable d3 has to assume, always, a value greater than zero. The data in the Table 7.5. and in the Figure 7.5. confirm this behavior.

TABLE 7. 5.

-		**x1**	**x2**	**x3**	**x4**	**x5**	**d3**	**d4**	**d5**	**d12**	**d34**
Case C4	Mean	119.3	111.8	240.8	244.5	427.0	95.0	98.5	146.3	7.5	-3.7
	Std.Dev.	98.6	100.5	121.6	106.7	150.2	70.5	68.4	103.0	82.0	94.8
Case C5	Mean	49.3	44.3	129.2	119.8	205.3	58.3	48.9	62.6	5.0	9.4
	Std.Dev.	54.4	35.6	79.2	67.4	105.0	42.8	32.0	72.2	70.6	42.1
Case C6	Mean	31.3	25.4	75.6	77.0	121.9	33.6	35.0	30.9	5.8	-1.4
	Std.Dev.	19.3	20.4	24.0	31.0	30.9	18.7	27.6	14.1	31.8	38.1

Table 7.5. *The sample mean and the standard deviation of the variables x1, x2, x3, x4, x5, d3, d4, d5, d12 and d34.* The variable d3 represents the shift of the E3 learning with respect to the E1 and E2 learning. The variable d4 represents the shift of the E4 learning with respect to the E1 and E2 learning. The variable d5 represents the shift of the E5 learning with respect to the E3 and E4 learning. The variable d12 represents the shift of the E1 learning with respect to the E2 learning. The variable d34 represents the shift of the E3 learning with respect to the E4 learning.

By contrast, the learning of the example E1 - (A1, P1, P2) is independent from the learning of the example E2 - (A2, P3, P4), that is, the variable d12 has to follow a normal distribution with mean equal to zero in each case. If we compare the d12 sample values belonging to each case with the expected normal values, see the Figure 7.6., it is clear that the d12 sample values follow a normal distribution.

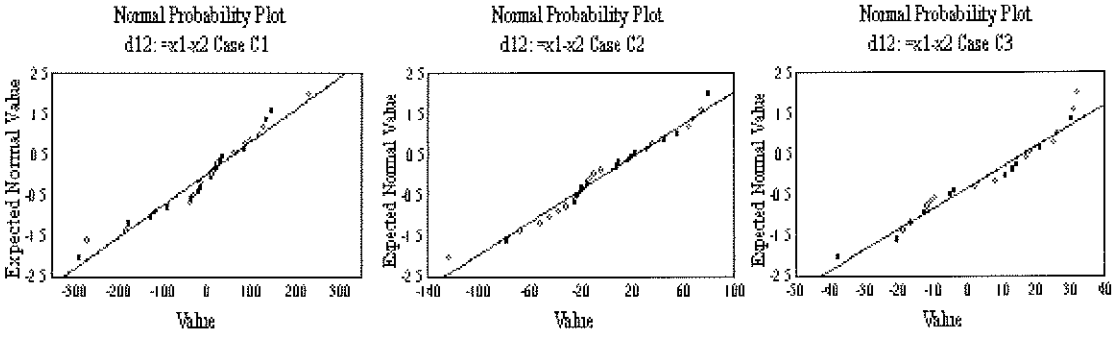


Fig. 7.6. *Normal test.* In the cases C1, C2, C3, the expected normal values of d12 has been compared with the sample values of d12. The circles represent the sample values. The continuous line represents the expected normal values. The variable d12 represents the shift of the E1 learning with respect to the E2 learning.

In each case C1, C2 and C3 we test the two following hypotheses by t-test [261]:

null-hypothesis: the mean of d12 is equal to 0.0
alternative hypotheses: the mean of d12 is different from 0.0
and we obtain the following results:
t-statistic = - 0.28, p-value = 0.78 in the case C1
t-statistic = - 0.33, p-value = 0.74 in the case C2
t-statistic = 1.84, p-value = 0.08 in the case C3
so we do not reject the null-hypothesis for alpha equal 0.05. That is, since the p-values are always greater than, or equal, to 0.05 we cannot reject the null hypothesis. Hence, we can state that the variable d12 follows a normal distribution with mean equal to zero.

7.5.2. *The C4, C5, C6 cases.*

Also in the C4, C5, C6 cases the input sequence is regulated with the condition c1. In the case C4 we have the following constrains $P(H) = 0.90$, $P(H_1/H) = P(H_2/H) = P(H_3/H) = P(H_4/H) = P(H_5/H) = 0.10$ and $P(H_7/H) = 0.90$, in the case C5 we have $P(H) = 0.90$, $P(H_1/H) = P(H_2/H) = P(H_3/H) = P(H_4/H) = P(H_5/H) = 0.15$ and $P(H_7/H) = 0.85$, and in the case C6 we have $P(H) = 0.90$, $P(H_1/H) = P(H_2/H) = P(H_3/H) = 0.20$ and $P(H_6/H) = 0.80$. In each case let us consider the following scenario: as soon as the example-subsequences E1, E2, E3, E4 and E5 are learned as the i_1-th, j_1-th, k_1-th, l_1-th and m_1-th input subsequence, respectively, we "restart the system", that is we set the time t equal to zero, the node outputs equal to zero and the coupling coefficients which are associated to the Hebbian connections equal to zero. Then, as soon as the example-subsequences E1, E2, E3, E4 and E5 are learned as the i_2-th, j_2-th, k_2-th, l_2-th and m_2-th input subsequence, respectively, we "restart the system" again. This procedure is repeated indefinitely.

Like the C1, C2 and C3 cases, let us consider the variable x1 defined on the set $\{i_1, i_2, i_3, ...\}$, the variable x2 defined on the set $\{j_1, j_2, j_3, ...\}$, the variable x3 defined on the set $\{k_1, k_2, k_3, ...\}$, the variable x4 defined on the set $\{l_1, l_2, l_3, ...\}$, the variable x5 defined on the set $\{m_1, m_2, m_3, ...\}$, the variable d3 defined on the set $\{l_1\text{-}\max(i_1, j_1), l_2\text{-}\max(i_2, j_2), ...\}$, the varible d4 defined on the set $\{k_1\text{-}\max(i_1, j_1), k_2\text{-}\max(i_2, j_2), ...\}$, the variable d5 defined on

the set $\{m_1\text{-}\max(k_1, l_1), k_2\text{-}\max(k_2, l_2), \ldots\}$, the variable d12 defined on the set $\{i_1\text{-}j_1, i_2\text{-}j_2, \ldots\}$and the variable d34 defined on the set $\{k_1\text{-}l_1, k_2\text{-}l_2, \ldots\}$. The variable d3 represents the shift of the E3 learning with respect to the E1 and E2 learning. The variable d4 represents the shift of the E4 learning with respect to the E1 and E2 learning. The variable d5 represents the shift of the E5 learning with respect to the E3 and E4 learning. The variable d12 represents the shift of the E1 learning with respect to the E2 learning.

The variable d34 represents the shift of the E3 learning with respect to the E4 learning.

In each case C4, C5 and C6, we take a sample of 30 elements of the variables x1, x2, x3, x4, x5, d3, d4, d5, d12, d34, and we computed the sample mean and the sample standard deviation of those variables, see the Table 7.5. and the Figures 7.7., 7.8. and 7.9..

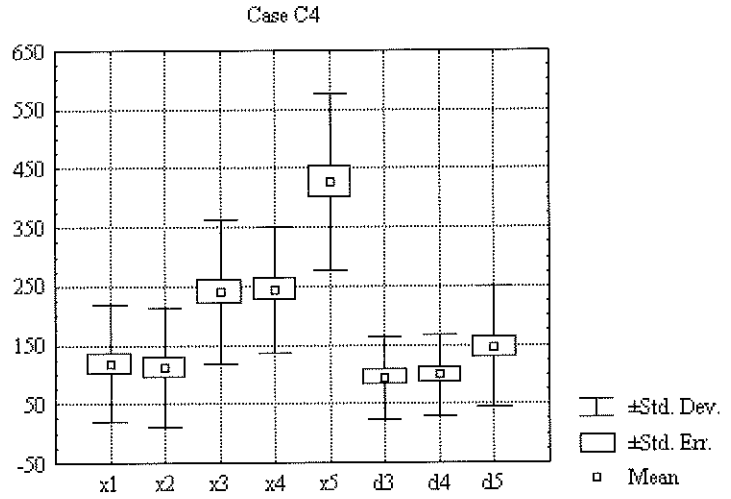


Fig. 7.7. *The sample mean and the standard deviation of the variables x1, x2, x3, x4, x5, d3, d4 and d5 in the case C4.* The variable d3 represents the shift of the E3 learning in respect to the E1 and the E2 learning. The variable d4 represents the shift of the E4 learning in respect to the E1 and the E2 learning. The variable d5 represents the shift of the E5 learning in respect to the E3 and the E4 learning.

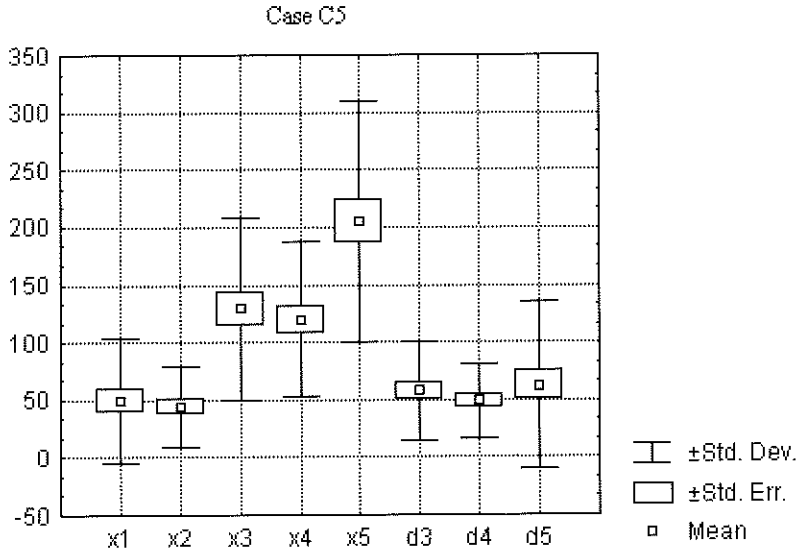


Fig. 7.8. *The sample mean and the standard deviation of the variables x1, x2, x3, x4, x5, d3, d4 and d5 in the case C5.* The variable d3 represents the shift of the E3 learning with respect to the E1 and the E2 learning. The variable d4 represents the shift of the E4 learning with respect to the E1 and the E2 learning. The variable d5 represents "the shift of the E5 learning with respect to the E3 and the E4 learning.

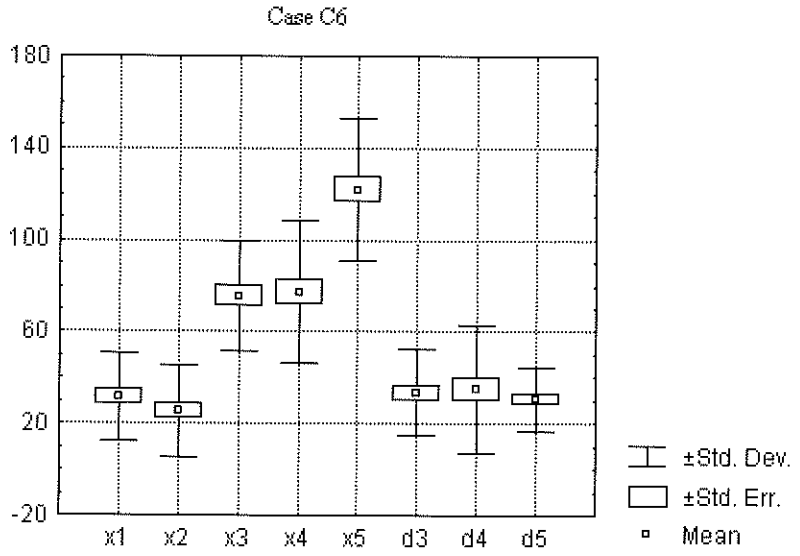


Fig. 7.9. *The sample mean and the standard deviation of the variables x1, x2, x3, x4, x5, d3, d4 and d5 in the case C6.* The variable d3 represents the shift of the E3 learning in respect to the E1 and the E2 learning. The variable d4 represents the shift of the E4 learning in respect to the E1 and the E2 learning. The variable d5 represents the shift of the E5 learning in respect to the E3 and the E4 learning.

The example-subsequences E3 = (A3, A1, A2) and E4 = (A4, A2, A1) can be learned only when the example-subsequences E1, E2 have been learned already. In the same way the example-subsequence E5 = (A5, A3, A4) can be learned only when the example-subsequences E3 an E4 have been learned already. So the variables x3, x4 should assume always a value greater than both x1 and x2 in each repetition, and the variable x5 should assume always a value greater than both x3 and x4 in each repetition. Accordingly, the variables d3, d4 and d5 should assume always a value greater than zero. The data in the Table 7.5 and in the Figures 7.10., 7.11. and 7.12 confirm this behavior.

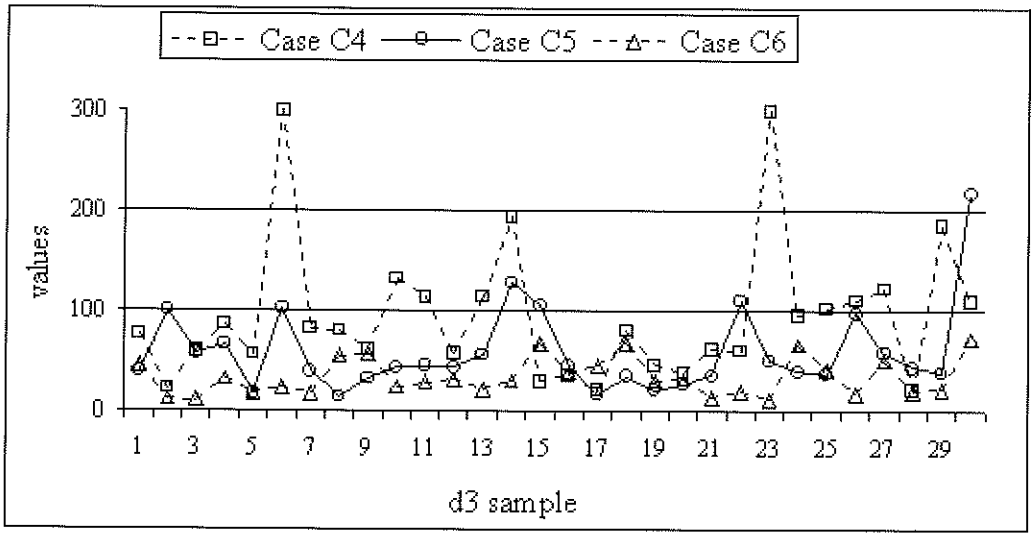


Fig. 7.10. *The d3 sample in the cases C4, C5 and C6.* The variable d3 represents the shift of the E3 learning with respect to the E1 and the E2 learning. The data in the figure confirm that the variable d3 assumes always a value greater than zero.

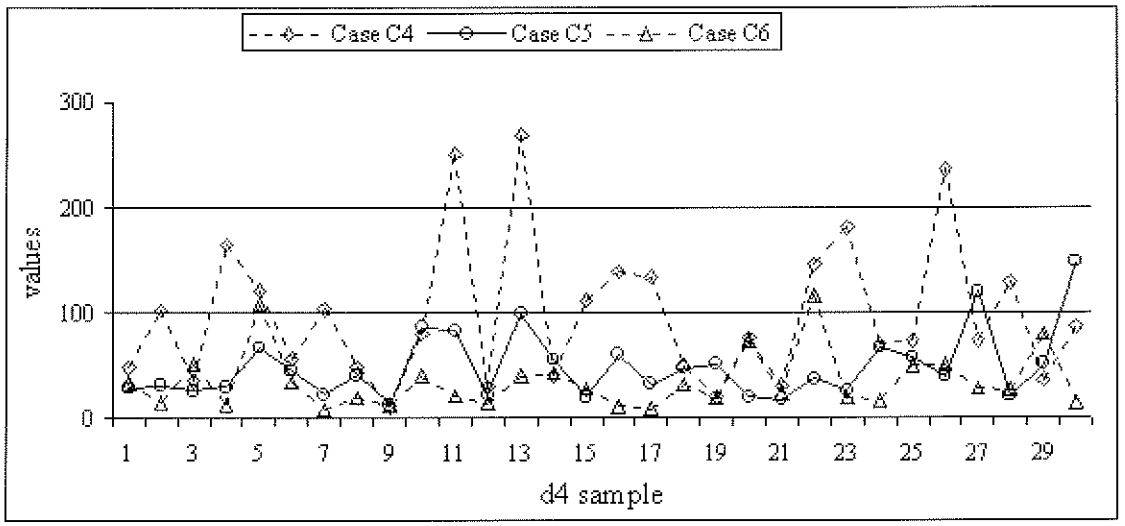


Fig. 7.11. *The d4 sample in the cases C4, C5 and C6.* The variable d4 represents the shift of the E4 learning with respect to the E1 and the E2 learning. The data in the figure confirm that the variable d4 assume always a value greater than zero.

By contrast, the learning of the example-subsequence E1 = (A1, P1, P2) is independent from the learning of the example-subsequence E2 = (A2, P3, P4), the learning of the example-subsequence E3 = (A3, A1, A2) is independent from the learning of the example-subsequence E4 = (A4, A2, A1), that is the variables d12, d34 have to follow a normal distribution with mean equal to zero in each case. If we compare the d12 and d34 sample values belonging to each case with the expected normal values, see the Figure 7.13. for the d34 values, it results the d12 and d34 sample values follow a normal distribution.

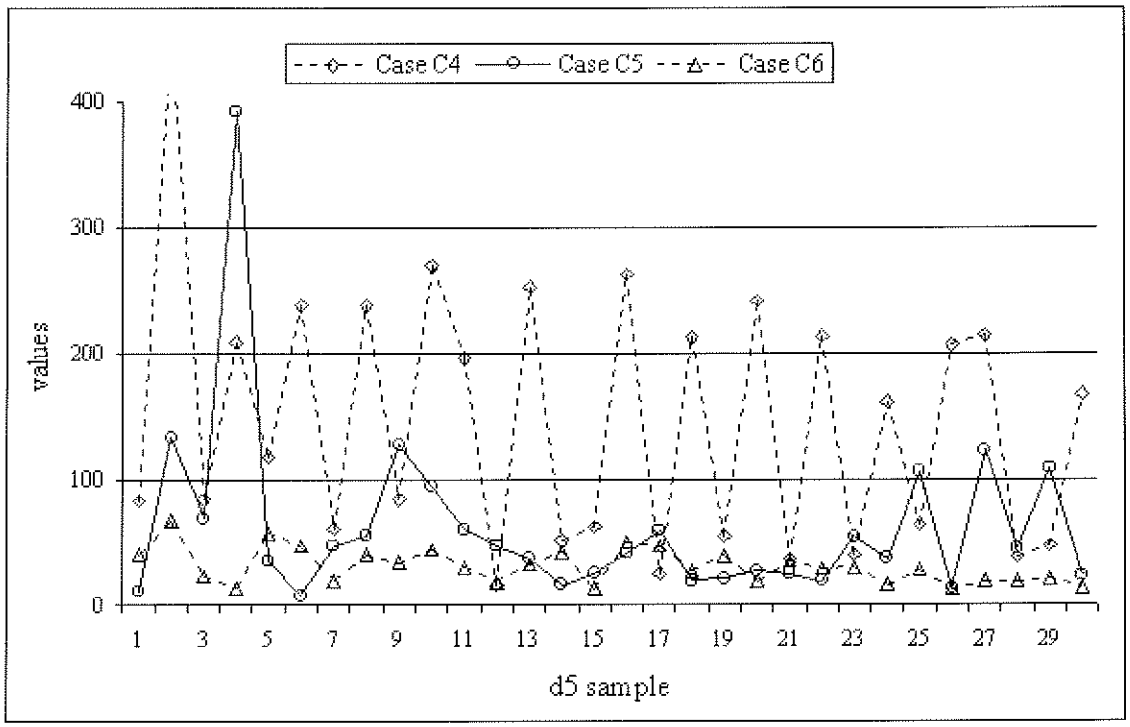


Fig. 7.12. *The d5 sample in the cases C4, C5 and C6.* The variable d5 represents the shift of the E5 learning with respect to the E3 and the E4 learning. The data in the figure confirm that the variable d5 assumes always a value greater than zero.

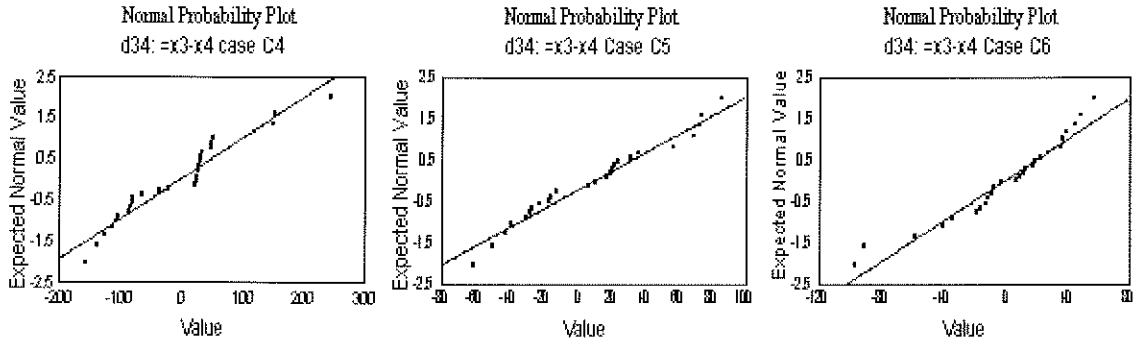


Fig. 7.13. *Normal test.* In the cases C4, C5, C6 the expected normal values of d34 are compared with the d34 sample values. The variable d34 represents the shift of the E3 learning with respect to the E4 learning. The circles represent the d34 sample values. The continuous line represents the expected normal values. The d34 sample values follow a normal distribution

If in each case C4, C5 and C6 we test the four following hypotheses by t-test:

Null-hypothesis: mean of d12 is equal to 0.0
Alternative hypotheses: mean of d12 is different from 0.0
Null-hypothesis: mean of d34 is equal to 0.0
Alternative hypotheses: mean of d34 is different from 0.0

we obtain the following results for d12:

t-statistic = 0.5, p-value = 0.62 in the case C4
t-statistic = 0.39, p-value = 0.70 in the case C5
t-statistic = 1.00, p-value = 0.33 in the case C6

and we obtain the following results for d34:

t-statistic = -0.21, p-value = 0.83 in the case C4
t-statistic = 1.22, p-value = 0.23 in the case C5
t-statistic = -0.20, p-value = 0.84 in the case C6

so we do not reject the null-hypothesis for alpha equal 0.05. That is, since the p-values are always greater than or equal to 0.05 we cannot reject the null hypothesis. Hence, the variables d12 and d34 follows a normal distribution with mean equal to zero, see the Figure 7.13.

7.5.3. *Associating one network input to one general assembly*

In the chapter 5.6. we have presented a GABNN automatically associating one network input to each general assembly. That is, we have implemented in a GABNN an algorithm such that the Hebbian connections going from the network input, see the Figures 5. 9. D) and 5. 9. D'), to one general assembly A are frozen as the example-subsequence E = (A, A1, A2, d1, d2) is learned.

We have tested the algorithm in the following cases:

a) $P(H) = 0.9$, $P(H_1/H) = 0.40$ and $P(H_8/H) = 0.60$
b) $P(H) = 0.9$, $P(H_1/H) = 0.30$ and $P(H_8/H) = 0.70$
c) $P(H) = 0.9$, $P(H_1/H) = 0.20$ and $P(H_8/H) = 0.80$

By definition the event H_8 is:

$H_8 \equiv$ "the n-th input subsequence at the time $t = n^*D_E^*tau$ is an example-subsequence different from both E1 and the null-example"

In each case let us consider the following scenario: as soon as the example-subsequence E1 = (A1, P1, P2) is learned and one network input is associated to the general assembly A1 at the i_1-th input subsequence, we "restart the system", that is we set the time t equal to zero, the node outputs equal to zero and the coupling coefficients associated to the Hebbian connections equal to zero. Then, as soon as the example-subsequence E1 = (A1, P1, P2) is learned and one network input becomes the general assembly A1 identifier at the i_2-th input subsequence, we "restart the system" again. This procedure is repeated indefinitely.

Like the above discussed cases, let z1 be the variable defined on the set $\{i_1, i_2, i_3, ...\}$. So, the variable z1 represents the number of input subsequences as the example-subsequence E1 = (A1, P1, P2) is learned and one particular net input becomes the general assembly A1 identifier.

For each case we have taken a sample of 30 values of the variable z1, and we have computed the sample mean and the sample standard deviation of this variable, see the Table 7.6..

TABLE 7. 6.

	-	z1
Case a)	Mean	10.7
	Std.Dev.	5.0
Case b)	Mean	17.4
	Std.Dev.	4.0
Case c)	Mean	29.6
	Std.Dev.	8.6

Table 7.6. *The mean and the standard deviation of the variable z1 in the cases a), b) and c).* The variable z1 represents the number of input subsequences occurred as the example-subsequence E1=(A1, P1, P2) is learned and one net input becomes the general assembly A1 identifier.

In each case the variable z1 assumes always finite values, that is, just one network input is always associated as identifier to just one general assembly, the A1 general assembly, and vice versa, as the example-subsequence E1 = (A1, P1, P2) is learned, see the Figure 7.14..

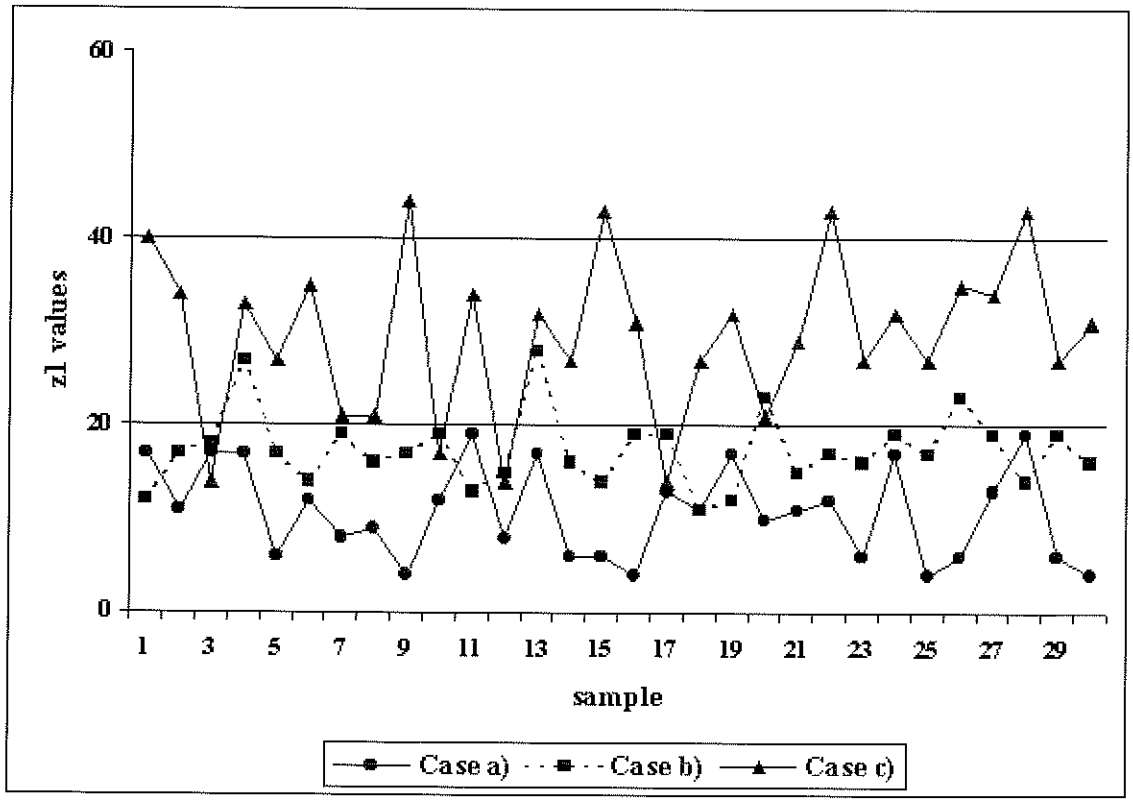


Fig. 7.14. *The distribution of the z1 sample in the cases a), b) and c).* The variable z1 represents the number of input subsequences occurred as the example-subsequence E1 = (A1, P1, P2) is learned and one net input is associated to the assembly A1. In each case the variable z1 assumes finite values, that is, just one network input is associated as identifier to just one assembly, the A1 assembly, and vice versa, as the example-subsequence E1 = (A1, P1, P2) is learned.

7.6. Conclusions.

We have presented the Java package it.na.cy.nnet as a tool to develop neural networks and proved that the performance of a simulator developed by it.na.cy.nnet depends on the number of active network nodes. If the total number of the network nodes is one order of magnitude greater than the product of the maximum delay times the average number of active nodes, then, the simulator speed is greater than the speed of a simulator computing the state of all the net nodes every tau.

Using the it.na.cy.nnet package we have simulated a General Adaptable Boolean Neural Network (GABNN), as defined in Chapter 5., using both the BoolHebbNet class and the high level language CONNET, in order to study the neural network–environment interactions.

Our tests confirm the network GABNN capability to learn autonomously new cause-effect relations between specific inputs and specific output sequences by means of incremental and continual learning.

We have tested both a GABNN having just one network input associated to just one general assembly and a GABNN automatically associating exactly one network input to each general assembly.

Chapter 8

CONCLUSIONS

8.1. Introduction.

Given a Boolean neural net, see the chapter 1. 1., let I be its input subvector, see the chapter 1.3., and the cardinality of the alphabet I containing the symbols corresponding to the I different values assumed by the subvector I: for an instance of the latter correspondence see the chapter 2.. We call P the proper subset of the alphabet I composed by its first P symbols, with $0 < P < I$. Let S be one unbounded sequence of I symbols, generated one symbol after the other. We are going to define an initially empty proper subset M_S of I, relative to the sequence S. As by definition $M_S \cap P = 0$, for any given unbounded sequence S, they are disjoint subsets of I.

To fix the ideas, let S be represented by the sequence $i_1, i_2, ..., i_q, ..., i_m, ..., i_n, ..., i_{q'}, ..., i_{m'}, ..., i_{n'}, ...$, where, first, $i_1, i_2, i_q, i_m, i_n, i_{q'}, i_{m'}, i_{n'}$ represent symbols belonging to I, second, the symbol represented by i_q is different from the symbol represented by either i_m or i_n, third, the symbol represented by $i_{q'}$ is different from the symbol represented by either $i_{m'}$ or $i_{n'}$, fourth, the integers $q < m < n < q' < m' < n'$ represent the absolute positions, in S, of the symbols represented by $i_q, i_m, i_n, i_{q'}, i_{m'}$ and $i_{n'}$, respectively.

If i_q represents the symbol $\Omega \in I - [M_S \cup P]$, i_m represents the symbol $\Xi \in M_S \cup P$, i_n represents the symbol $\Psi \in M_S \cup P$ and if neither Ξ nor Ψ is present in the S subsequence comprised between i_q and i_m, extremes excluded, and if neither Ξ nor Ψ is present in the S subsequence comprised between i_m and i_n, extremes excluded, we call *example*, or *example of* Ω *relative to* S, the quintuple $E_S \equiv [i_q, i_m, i_n, \Delta mq, \Delta nml]$, where Δmq, Δnm, represents the *distance* between i_q and i_m, i_n and i_m, ie, it represents the number of symbols comprised between i_q and i_m, i_m and i_n, in the sequence S. We call *father* the symbol represented by i_q and *first, second, son* the symbol represented by i_m, i_n. We call *example father* the example of the father; *example first son* the example of the first son; and *example second son* the example of the second son.

Because I is finite and S is unbounded, we can consider $i_{q'} \equiv \Omega$, $i_{m'} \equiv \Xi$, $i_{n'} \equiv \Psi$, $\Delta mq - \Delta m'q'$, $\Delta nm - \Delta n'm'$, ie, the example $E'_S \equiv [i_{q'}, i_{m'}, i_{n'}, \Delta m'q', \Delta n'm']$. We represent with $\Delta E'_S E_S$ the relative position in S of the two examples E_S and E'_S, ie, the distance $\Delta q'n$ between i_n and $i_{q'}$. We call *consecutive* the examples E_S and E'_S, if in the sequence S there is no example $E''_S \equiv [i_{q''}, i_{m''}, i_{n''}, \Delta m''q'', \Delta n''m'']$ with $i_{q''} \equiv \Omega$, $i_{m''} \equiv \Xi$, $i_{n''} \equiv \Psi$, $\Delta mq - \Delta m''q''$, $\Delta nm - \Delta n''m''$, $i_n < i_{jq''}$ and $i_{n''} < i_{q'}$, ie, if the sequence comprised between E_S and E'_S contains no example of Ω relative to S. Let us consider the S' symbols subsequence S' belonging to the sequence S, with $\Delta mq + \Delta nm$ smaller than S'. We call *repetion cadence of the example* E_S *relative to the sequence S'*, or simply the E_S *repetion cadence*, for short Γ_S,

the actual number of the E_S instances contained in S'. Let $\Theta = S'/[\Delta mq + \Delta nm]$ be the number of times, as opposed to the actual number of times, the example E_S could be, is, contained in the subsequence S' and let t, u, v, w, θ be positive integers, with $\theta < \Theta$. By definition:

First, if there exists a subsequence S' *of the sequence* S, *such that* S' *contains* S' *symbols and its first component* i_q *represents the symbol* Ω, *second, if* $\Theta^{-1} < \Delta E'_S E_S < \theta^{-1}$, *ie, if the set of the distances between any two consecutive instances of the example* $E_S \equiv [\Omega, \Xi, \Psi, \Delta mq, \Delta nm]$ *in S' has the upper bound* θ^{-1} *and the lower bound* Θ^{-1}, *ie, if the example* E_S *has a repetion cadence* $\Gamma_{S'}$ *such that* $\theta < \Gamma_{S'} \leq \Theta$, *third, if* $t < \Delta mq < u$ *and, fourth, if* $v < \Delta nm < w$, *then, from the* $[i_q + S']$-th *symbol onward* $\Omega \in M_S$.

Obviously, for any given alphabet I, there are sequences S such that their relative sets M_S are empty. Eg, M_S is empty for the sequences obtained by the endless repetion of the very same symbol, for the sequences containing only elements belonging either to P or to I – P and for the sequences containing only examples with $\Gamma_{S'} < \theta$.

8.2. The Caianiello-Hebb transform.

In the chapter 1.3. we have presented the Caianiello, or C, transform applied to the row by column product of a square matrix times a vector having the same number of rows. By definition, to assign a Boolean neural net, we have to assign its coupling coefficients square matrix $\|C_{h,\,k,\,0}\|$ initial value, ie, we have to assign both the matrix constant element initial values and the matrix variable element initial values, see the chapter 4.. Again by definition, to assign the Boolean neural net initial state, we have to assign the vector initial value, ie, the Boolean neural net state vector F_0. The C transform result is the Boolean neural net next state vector F_1. Once we have $\|C_{h,\,k,\,0}\|$, F_0 and F_1 we can apply to them the Hebb, or H, transform, see the chapter 4. 1., to obtain the next coupling coefficient matrix $\|C_{h,\,k,\,1}\|$.

That is, once we have the matrix $\|C_{h,\,k,\,t}\|$ and the state vector F_t, the CH transform projects the state vector F_t in the state vector F_{t+1} and, in the order, the matrix $\|C_{h,\,k,\,t}\|$ in the matrix $\|C_{h,\,k,\,t+1}\|$. Accordingly, as the CH transform maps the state vector and the matrix sets into themselves, it can be applied to its own result to generate both a state vector succession and a matrix succession, see also the chapters 5.4.6. and 5.5.4..

Because of the given Hebbian rules, see the chapter 4.1.1., we say: the vector and matrix successions have *definitively acquired the property relative to the component j at the iteration*, or *time, t** if in the coupling coefficients matrix the j-th column variable elements are time dependent for $t < t^*$ and are constant for $t \geq t^*$.

In [6, 27] and in the chapters 5. and 6., we have proposed some CH transform instances. Actually, we have proposed some initial matrices and a correspondence mapping the alphabet I into the input subvector I, see the chapter 2. 2.. That is, the sequence S symbols are in a one-to-one correspondence with the succession of the I subvectors, one per iteration. Eventually, we have proposed also a correspondence mapping the Boolean neural net output subvector O into the alphabet O, see the chapter 2. 2. 1., having the cardinality P, namely the cardinality of P the proper subset of I introduced in the chapter 8.1.. Obviously, the net universality depends on the alphabet O choice.

8.3. The autonomously self-adaptable boolean neural nets.

Let us consider one of the CH transform instances proposed in [6, 27] or in the chapter 5., and a sequence S containing a finite number of finite subsequences each con-

taining at least one example with a repetion cadence verifying the conditions given in 8. 1. for M_S. That is, we are considering a sequence S containing the subsequence S', in turn, containing at least the example E'_S with a repetion cadence $\Gamma'_{S'}$ and the subsequence S", in turn, containing at least the example $E"_S$ with a repetion cadence $\Gamma"_{S"}$ and ... and the subsequence S'⋯', in turn, containing at least the example $E'^{\cdots'}_S$ with a repetion cadence $\Gamma'^{\cdots'}_{S'^{\cdots'}}$.

Actually, in the proposed CH transforms, we have chosen the Hebbian parameters so that each example $E'^{\cdots'}_S$ with the repetion cadence $\Gamma'^{\cdots'}_{S'^{\cdots'}}$ is sufficient to increase the value of at least some of the square matrix iteration dependent elements until they become constants at the time t*. In other words, if $C_{h,\,k,\,t}$ is a variable coupling coefficient for t smaller than t*, then, the net has definitively acquired the property relative to the component k at the time t* if $C_{h,\,k,\,t}$ is a constant coupling coefficient for t no smaller than t*. Accordingly, for t greater than t*, every time the state vector element k changes its value the same happens to the element h at the next iteration, for short the component k *identifies* the component h. E. g., in the proposed CH transforms, to each symbol belonging to the subset P it corresponds just one I subvector component which, in turn, identifies by construction just one O subvector component, in a one-to-one correspondence. We leave to the interested reader to extend the definition for the component k to identify the component h with a given delay. Or for one particular I subvector state to identify a component h with a given delay.

Once an I symbol becomes an element of the set M_S, every time the state of the I input subvector components represents the said symbol is followed, at the next iteration, by the change of the state of at the least one component belonging to the I complement on the state vector. In other words, whenever a symbol becomes an element of the set M_S it becomes an example father, see the chapter 8.1.. Because an M_S element identifies the two components representing the example first and second son, with their respective delays equal to their distances in the sequence S, an M_S element is an example father. As the last conclusion can be iterated for each of the latter two components and for the components identified by the example sons and for the components identified by the example sons of their example sons and so on, if the I subvector components activity represents a symbol belonging to the set M_S, then, it identifies a well defined output subvector sequence.

We say: *the Boolean neural net autonomously has self-adapted itself to the symbol Ω at the time t* relative to a sequence S, if $\Omega \notin M_S$ for $t < t^*$ and $\Omega \in M_S$ from t* onward.*

In other words, whenever a Boolean neural net has autonomously self-adapted itself to a symbol Ω at the time t*, relative to a sequence S, whereas to each instance of the input subvector components activity representing the symbol $\Omega \notin M_S$ at any time smaller than t* it does not correspond one output subvector sequence, from t* onward to each instance of the very same input subvector components activity representing the very same symbol $\Omega \in M_S$ it does correspond just one well defined output subvector sequence. Actually, the output subvector sequence corresponding to the symbol $\Omega \in M_S$ is the concatenation of the output subvector sequences corresponding to Ω first and second son.

The lengths of the output subvector sequences so identified have an upper limit which depends on the state vector, and hence on the coupling coefficients matrix, dimension. However, whereas it is true that the output sequence lengths have an upper limit, there are no limits on the number of the output subvector sequences which can be chained one after the other, provided we can project the output subvector component values into the input subvector component values. For some illustrative examples on how to project the former into the latter subvector component values and on how to chain different output sequences, see [27, 28] and the chapter 6..

Eventually, if in the sequence S there are triplets of symbols which do not satisfy the example conditions, eg if the father belongs to $M_S \cup P$ or if the same symbol is both the father and one of the two sons, then a loop could be introduced into the system. Accordingly, to have a Boolean neural network autonomously controlling if the conditions given for the examples are always verified, eg whenever the set M_S is empty then both sons have to be P elements, we have proposed for the matrix elements the initial values discussed in the chapter 5.6.. Actually, among the state vector components, belonging to the $I \cup O$ complement, we have inserted some components computing suitable combinations of the NAND and the NOR of their arguments to prevent the net from adapting to the symbols which do not satisfy the example conditions. A test on the latter algorithms and on the correctness of the it.na.cy.nnet package implementation has been discussed in the chapter 7.5..

REFERENCES

[1] McCulloch, W. S. and W. Pitts - "A logical calculus of the ideas immanent in the nervous activity" Bull. Math. Bioph. 5 (1943) 115-143.

[2] Caianiello, E. R. - "Outline of a theory of thought processes and thinking machines" J. of Theor. Biol. 2 (1961) 204-235.

[3] Rumelhart, D. E., J. L. McClelland and the PDP Research Group - "Parallel distributed processing. Vol I Foundations" Bradford, MIT Press, Cambridge (Mass.) USA (1986) 547 + xx.

[4] Hassoun, Mohamad H. - "Fundamentals of artificial neural networks" MIT Press, Cambridge (Mass.) USA (1995) 511 + xxvi.

[5] Hebb, D. O. - "The organization of behavior" J. Wiley & Sons (New York), USA (1966) 335 + xix.

[6] Lauria, F. E., M. Sette, S. Visco - "Adaptable Boolean neural networks" Fridericiana Scientia, Liguori (Napoli) (1997) 212. ISBN 88-207-2676-9. http://www.na.infn.it/Gener/cyber/nnet.htm

[7] Mehra, P., B. W. Wah - "Artificial Neural Networks: concepts and theory" IEEE Computer Soc. Press, Los Alamitos (Cal.) USA (1992) 667 + xi.

[8] Anderson, J. A. - "An introduction to neural networks" Bradford MIT Press, Cambridge (Mass.) USA (1995) 650 + xii.

[9] Kirsh, D. - "Foundations of AI: the big issues" AI 47 (1991) 3-30.

[10] Brooks, R. A. - "Intelligence without representation" AI 47 (1991) 139-159.

[11] Knuth, D. E. - "The art of computer programming" Addison Wesley (Reading, Mass. USA), vol.1 (1969) pg. 634 + xxi.

[12] Knuth, D. E. - "The art of computer programming" Addison Wesley (Reading, Mass. USA), vol.3 (1973) pg. 722 + xi.

[13] Cormen, T. H., C. E. Leiserson, R. L. Rivest - "Introduction to algorithms" MIT Press (Cambridge, Mass. USA), (1990) pg. 1028 + xvii.

[14] D'Arienzo, G. - "Strutture dati intenzionali" (Intentional data structures) Tesi di laurea in fisica Dipartimento di Scienze Fisiche, University of Naples "Federico II" aA 1994-95 (1994).

[15] Romeo, G. - "Rappresentazione intenzionale di dati in liste sequenziali" (Intentional data representation in sequential lists) Tesi di laurea in fisica Dipartimento di Scienze Fisiche, University of Naples "Federico II" aA 1995-96 (1995).

[16] Albanese, M. - "Rappresentazione intenzionale di dati in liste legate" (Intentional data representation in linked lists) Tesi di laurea in fisica Dipartimento di Scienze Fisiche, University of Naples "Federico II" aA 1999-2000 (2001).

[17] Lauria, F. E., R. Prevete, M. Milo, S. Visco - "On the adaptable Boolean neural net paradigm" M. Marinaro ed. "Neural networks WIRN Vietri-97" Springer (London) (1998), 318-323, ISBN 3-540-76157-8.

[18] Violi, T. - "Valutazione in presenza di rumore dei parametri di una regola hebbiana" (Valuation of the Hebbian rule parameters in presence of noise) Tesi di laurea in fisica, Dipartimento di Scienze Fisiche, University of Naples "Federico II" aA 1994-95 (1995).

[19] Caputo, A. - "Interazioni con ambiente rumoroso di reti neurali booleane organizzate in assemblee" (On the interactions of some Boolean neural nets organized in assemblies with a noisy environment) Tesi di laurea in fisica, Dipartimento di Scienze Fisiche, University of Naples "Federico II" aA 1997-98 (1998)

[20] Lauria F. E., M. Sette - "Teaching macros through program execution examples" E. R. Caianiello ed. "Neural nets WIRN Vietri-92" World Sc. (Singapore), (1992) 204-209.

[21] Bini Verona, F., F. E. Lauria - "On an adaptable connectionist robot control system" Neurocomputing 9 (1995) 165-185.

[22] Prevete, R. - "Addestramento di un robot controllato da una rete neurale booleana organizzata in assemblee" (Training a robot controlled by a Boolean neural net organized in assemblies) Tesi di Laurea in fisica. Dipartimento di Scienze Fisiche, University of Naples aA 1994-95 (1994).

[23] Microsoft Corporation, Online Documentation for Visual J++ version 1.1.

[24] Eckel, B. - "Thinking in Java", www.bruceeckel.com.

[25] Lauria, F. E., M.Sette, S. Visco - "The CONNET language and the McCulloch and Pitts network synthesis" R. Moreno-Diaz & J. Mira-Mira eds. "Brain processes, theories and models" MIT Press, (Cambridge, Mass.) (1995) 552-559.

[26] Wagner, S. F. - "Introduction to Statistic" R.A. Weinstein ed., Harper Perennial, (1999), ISBN: 0-06-467134-8.

[27] Lauria, F. E. - "On some sufficient conditions for the Caianiello-Hebb transform convergence" Rend. Acc. Sc. Fis. e Mat., Napoli, Serie IV- Vol. LXII (1995) 7-42.

[28] De Simone S. – "Sull' universalita' di reti booleane adattabili" (On the universality of the adaptable Boolean neural nets) Tesi di Laurea in fisica. Dipartimento di Scienze Fisiche, University of Naples aA 1994-95 (1995).

[29] Lauria, F. E. - "An algebraic approach to an autonomously adaptable system" Rend. Acc. Sc. Fis. e Mat., Napoli, Serie IV - Vol. LXVI (1999) 97-104.

INDEX

ABNN, or Adaptable BNN 1. 3., 5. 1.
  General ABNN, or GABNN 5.6.1.
  six node ABNN 5.4.5.
  sixteen node ABNN 5.5.1.
ABNN accepts the example 5.1.
ABNN has been trained, or has learned, to compute, or converges to, the function g[x] 5.2.2.
absolute inhibitory connection 1.2.
abstract 7.2.
ActiveNodes 7.2.3., 7.2.3.12.
adder 2.2.
assemblies 1.3., 5.3.2.
assembly, father or parent 5.3.3.
  first son or first offspring 5.3.3.
  general 4.3., 5.6.1.
  second son or second offspring 5.3.3.
  six node 5.4.1.
  sixteen node 5.5.1.

birecord 3.2.1.
BNN, or Boolean Neural Nets 1.1.
BNN computes the function g(x) 2.3.
BoolHebbNet 7.1., 7.2.1., 7.2.3., 7.2.3.16.
Brain 7.2.3., 7.2.3.13.

c1, condition 7.5.
cadence 4.1.1.
calling sequence 5.3.1.
  , first 5.6.3.1.
  , second 5.6.3.1.
carry component 2.2.2.
central comma 6.4.4.3.
chain two procedures, to 6.4.4.
compare subnet 5.6.4.
complete 3.2.2.
composite task 4.3.
computeNodeInput 7.2.2.
computeNodeOutput 7.2.2.
computes the function g(x), a BNN 2.3.
Connection 7.2.3., 7.2.3.4.
Connections 7.2.3., 7.2.3.8.
ConnectionVector 7.2.3., 7.2.3.7.
connection 1.1.
  , facilitatory 1.1.
  , inhibitory 1.1.
    , absolute 1.2., 1.3.
    , relative 1.2., 1.3.
constant part 3.2.1.
constructors 7.2.
control subnet 5.5.2.
converges 5.2.2.
coupling coefficient, frozen 4.1.1.
  time dependent $C_{h,k,d}[t]$ 4.1.1.
  time independent $C_{h,k,d}$ 1.3.
coupling coefficients matrix 1.3.

d, delay 1.3.
D, greatest delay 1.3.
defaultRun 7.2.1., 7.2.2.
DLDS 3.1.
duple input modality 5.3.1.

environment vector 2.2.2.
excitatory, connection 1.3.
example 5.1.
example presentation sequence 4.3.
example-subsequence, length of 7.5.
  E has been learned at the k-th input subsequence 7.5.
extends 7.2.

father, or parent assembly 5.3.3.
facilitatory, connection 1.3.
fields 7.2.
first addend component 2.2.2.
first calling sequence 5.6.3.1.
first Hebbian-in connections 5.6.2.
first operand component 2.2.2.

first phase algorithm 5.6.3.
first vector 3.2.1.
frozen, coupling coefficient 4.1.1.

General ABNN, or GABNN 5.6.1.
general assembly 4.3., 5.6.1.
grand-mother neuron 2.3.

handleActiveMotors 7.2.1., 7.2.2.
HBNN 7.1.
Hebbian connection 4.1.1.
Hebbian-in 4.1.1.
Hebbian-in connection 5.4.2., 5.6.2.
Hebbian learning 4.
Hebbian-out 4.1.1.
Hebbian-out connection 5.4.2., 5.6.2.
Hebbian parameters 4.1.1.
Hebbian rule 4., 4.1.1.
, first 4.1.1.
, second 4.1.1.
Hebbian sets 4.1.1., 5.3.2.
Hebbian set, first 5.6.2.
second 5.6.2.
Hebbian threshold 4.1.1.
hidden subvector 1.3.
History 7.2.3., 7.2.3.14.

IBNS, or Iterable Boolean Neural Subnet 5.3.
identifier 5.3.1.
leading 5.4.6., 5.5.4.
trailing 5.4.6., 5.5.4.
subnet 5.6.4.
incremental learning 4.3.
index 7.1.
InfoConnection 7.2.3., 7.2.3.5.
inhibitory, connection 1.3.
inhibitory identifier, or i-identifier 5.3.1.
input control subnet 5.6.4.
input sequence 7.5.
input subvector 1.3.
Iterable Boolean Neural Subnets, or IBNSs 5.3.
intermediate record 3.2.1.
it.na.cy.nnet 7.1.

leading identifier 5.4.6.
learning alternative 5.6.3.2.
learning, assisted and unassisted, or autodidattic 5.1.
linear threshold unit 1.3.
LLNN, or Low Level Language for Neural Networks 7.4.
LTU 1.3.
k-th input subsequence 7.5.

mask 5.4.5., 5.5.2.
match point 6.4.4.3.
members 7.2.
methods 7.2.
multiple input modalities 5.3.1.
mySimulator 7.3.

negative, connection 1.3.
NetworkConstants 7.2.3., 7.2.3.1.
NetworkVector 7.2.3., 7.2.3.6.
NeuralNetSim 7.1., 7.2.2., 7.2.3., 7.2.3.15.
Node 7.2.3., 7.2.3.2.
NodeVector 7.2.3., 7.2.3.9.
non-abstract 7.2.
non-Hebbian connection 4.1.1.
number of the same example presentations 5.1.
number of distinct and different examples needed to train the ABNN 5.1.
null-example 7.5.

offspring 5.3.3.
offspring, first 5.3.3.
second 5.3.3.
or subnet 5.6.4.
output control subnet 5.6.4.
output subvector 1.3.

parent, or father assembly 5.3.3.
pre-input BNN 5.3.1.
PossibleActiveNodes 7.2.3., 7.2.3.10.
primitive 2.2., 3.2., 5.3.2.
primitive tasks 4.3.
property 7.1.

ReadPrintRAF 7.2.3., 7.2.3.18.
ReadStream 7.2.3., 7.2.3.17.
record 3.2.1.
RecordNode 7.2.3., 7.2.3.3.
RecordNodeVector 7.2.3., 7.2.3.11.
redundancy R 1.4.
relative inhibitory connection 1.2.
result component 2.2.2.
reset subnet 5.6.4.

second addend component 2.2.2.
second calling sequence 5.6.3.1.
second Hebbian-out connections 5.6.2.
second operand component 2.2.2.
second phase algorithm 5.6.3.

second vector 3.2.1.
setActiveSensors 7.2.1., 7.2.2.
sign component 2.2.2.
six node assembly 5.4.1.
six node ABNN 5.4.5.
sixteen node assembly 5.5.1.
sixteen node ABNN 5.5.1.
son, first 5.3.3.
    second 5.3.3.
state function $f_H[t]$ 1.3.
state vector 1.3.
static initializers 7.2.
store subnet 5.6.4.
suitable cadence 5.1.
suitable number 5.1.

tan 1.1.
three-record 3.2.1.
threshold 1.1.
trailing identifier 5.4.6.
training procedure 6.2.1.

variable part 3.2.1.