The Implementation of Two-Layer Binary Perceptrons

Jacob L. Cybulski, Adam Kowalczyk, Andrew Jennings and Frank Morselt,
Telecom Australia Research Laboratories

1.0 Introduction

Perceptrons, which are a form of more general Neural Networks (Minsky and Papert 1969, Lippman 1987), are a model of information processing and representation rapidly gaining popular acceptance amongst the Artificial Intelligence community for their ability for pattern classification, association, mapping, compression and vectorisation (Reddy, Carrol and Coleman 1989). The applications of Neural Networks range from Robotics (Nicewarner 1989), to Vision (Cheok, Smith and Fernando 1989), Speech Recognition and Synthesis (Bourlard and Wellekens 1987, McCulloch, Ainsworth and Linggard 1988), Optical Character Recognition (Gouhara, Imai and Uchikawa 1989), Natural Language Understanding and Generation (Cybulski and Jennings 1989, Kukich 1987), Expert Systems (Li, Teng and Wang 1989), and other areas.

This paper, however, focuses on a special case of multilayer perceptrons only. The perceptrons to be investigated have the following characteristics:

> *Two layer binary perceptrons* (TBP) are collections of interconnected processors of limited capacity and storage. The processors are repositories of binary values also known as activations and are arranged in two layers of active units, hidden and output, and an additional layer of passive input units. The hidden units calculate logical conjunctions of their connected inputs, the output units calculate the exclusive disjunction of all the connected unit activations. TBPs may be trained to associate appropriate output patterns with given sets of binary input values by massively parallel computation and communication of activations (Cybulski et al. 1989b).

For instance, Figure 1.a depicts a simple decision table of seven rules, each rule based on four conditions and two actions. Two different binary perceptrons equivalent to the decision table are given in 1.b and 1.c. Both of the TBPs have four input units (corresponding to the rule conditions), seven hidden units performing a logical AND on the connected inputs, and two outputs XORing the hidden units (equivalent to the rule actions). It may be shown that many other different perceptron architecture would fulfil our decision table functionality (at least 36 in this example). A number of different learning algorithm to generate such perceptrons from their training sets (decision rules) may be found elsewhere (cf. Lippman 1987, Rumelhart, Hinton and Williams 1986).

(a)

	R1	*R2*	*R3*	*R4*	*R5*	*R6*	*R7*
i1	1	1	1	0	1	0	0
i2	1	1	0	1	0	1	0
i3	1	0	1	1	0	0	1
i4	1	0	1	1	0	0	0
o1	1	0	0	1	1	1	1
o2	0	1	1	1	1	1	1

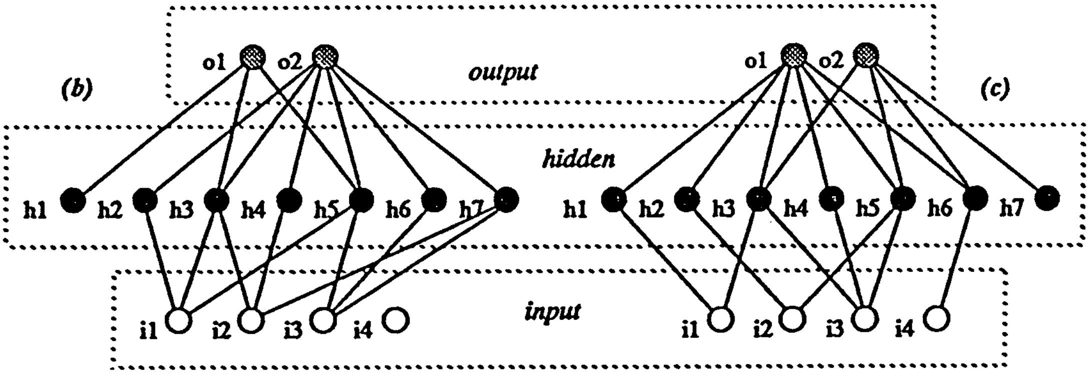


Figure 1 - Two TBP representations of a decision table

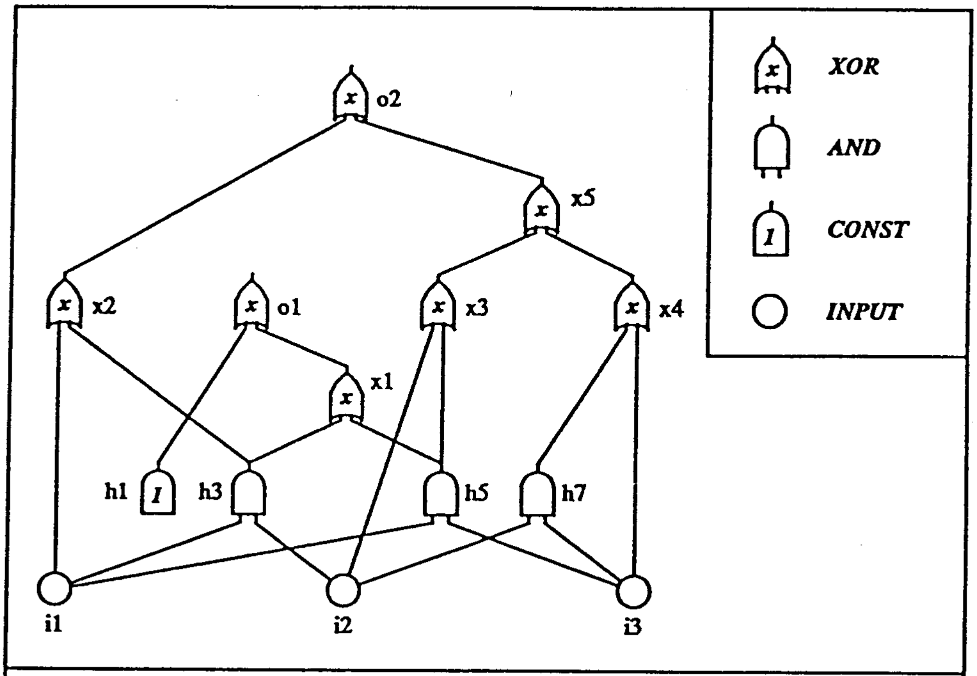
Despite the binary perceptron simple architecture, it can be demonstrated (Cybulski et al. 1989a,b) that in noise-free environments, TBPs become powerful pattern-classifiers and information-retrieval tools. Other types of perceptrons, e.g. *real, integer,* or *modulo,* may prove to be more beneficial in non-deterministic or fuzzy application domains (Kowalczyk and Ferra 1989, 1990).

One of the major problems in efficient use of perceptrons is the representation of their massively interconnected architecture and the subsequent computation of unit activations. We propose two different models of perceptron architecture; the first, based on decision trees, allowing very compact and efficient representation of learned patterns and the second, based on boolean matrices, offering a great flexibility in its hardware implementation. Both proposals are suitable for either sequential or parallel realisation.

2.0 Decision Tree Implementation of TBP

It can be easily seen that the function of any TBP can be expressed with a simple polynomial of boolean expressions, such polynomials are equivalent to decision trees of AND and XOR nodes. Additional elimination of redundant hidden units and the substitution of higher-order nodes with the binary ones results in a very compact and computationally efficient AND/XOR tree.

Figure 2 displays a decision tree implementation of the perceptron shown in Figure 1.b. Some of the perceptron hidden units, e.g. *h2*, *h4* and *h6*, were replaced by direct links between inputs and outputs, whereas the higher-order output nodes were expanded into equivalent multi-layer structures of ANDs and XORs.



$$o1 = 1 + i1 * i2 + i1 * i3$$
$$o2 = i1 + i2 + i3 + i1 * i2 + i1 * i3 + i2 * i3$$

Figure 2 - Binary decision tree and its boolean polynomial

Any classification task performed by so implemented TBP will require the number of logical operations equal to the number of the binary tree nodes, this can be estimated to be:-

$$N \approx \sum_{i_k} \log_2 \overline{H}_{i_k} + \sum_{i_o} \log_2 \overline{O}_{i_o}$$

where $\overline{H}_{i_k}$ is the order of $i_k{}^{th}$ hidden unit and $\overline{O}_{i_o}$ is the order of $i_o{}^{th}$ output unit.

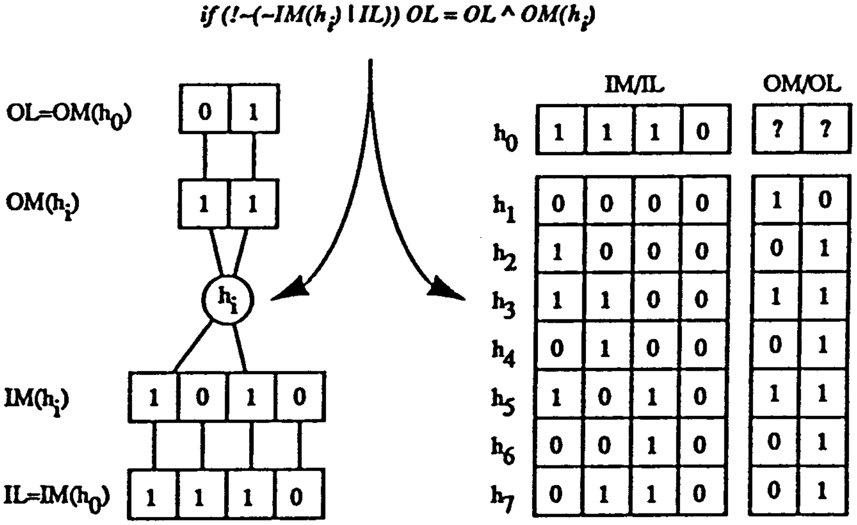
A parallel realisation of a decision tree can certainly improve the execution time of perceptron classification as its independent subtrees may act in full concurrence. Thus, the process complexity is proportional to the tree height, which will not exceed the order of $O\,(sup(\log_2 \overline{H}_{i_k}) + sup(\log_2 \overline{O}_{i_o}))$ (same notation as above).

Hence, a large perceptron (as in word lexical classification - Cybulski et al. 1989a) of at least 1000 hidden units of order up to 3, and 260 output units of order up to 700, may be implemented in hardware with 2 layers of AND gates and 10 layers of XOR gates, thus performing any classification task in 12 machine cycles.

However the hardware implementation iof the above architecture can become very irregular. It is also quite inflexible to changes in the perceptron structure. A slight retraining of the network may require a complete revision of the hardware design. Finally, a possibility of non-planar topologies of perceptron trees may have a detrimental effect on the hardware design process. Some of these problems may be alleviated with the use of redundancy-adding tree-building algorithm or with the use of a bus architecture allowing indirect linking of tree nodes (cf. Martinez and Vidal 1988). In any case the simplicity and the efficiency of the solution would be lost. Thus, it is clearly worth exploring some other, more uniform, approaches to the implementation of two layer binary perceptrons.

3.0 Boolean Matrix Implementation of TBP

Hidden units of two layer binary perceptrons may be viewed not only as intermediate decision nodes in an AND/XOR tree (as described in the previous section), but could also be seen as active processors mapping inputs nodes into outputs according to a predetermined method (Figure 3).



	IM/IL				OM/OL	
h_0	1	1	1	0	?	?
h_1	0	0	0	0	1	0
h_2	1	0	0	0	0	1
h_3	1	1	0	0	1	1
h_4	0	1	0	0	0	1
h_5	1	0	1	0	1	1
h_6	0	0	1	0	0	1
h_7	0	1	1	0	0	1

Figure 3 - Boolean matrix simulator corresponding to Figure 1.b

In our method, the mapping process uses two masks determining the hidden unit connections to input (IM masks) and output units (OM mask). Each mask is represented with a string of bits, where 1 indicates the existence of a connection between the hidden and input or output units, 0 means there is no such connection. All input and output masks are collected in a boolean matrix, one hidden unit per row, fully describing a given perceptron (e.g. a matrix in Figure 3 represents the perceptron 1.b). All input (IL) and output (OL) unit activations are also stored in the same matrix as that of the hidden units (row 0).

Once all of the hidden unit masks are defined, the perceptron may start the classification of patterns arriving on its input (IL) and to set its outputs (OL) appropriately to the pre-trained exemplars. To achieve this, each hidden unit checks

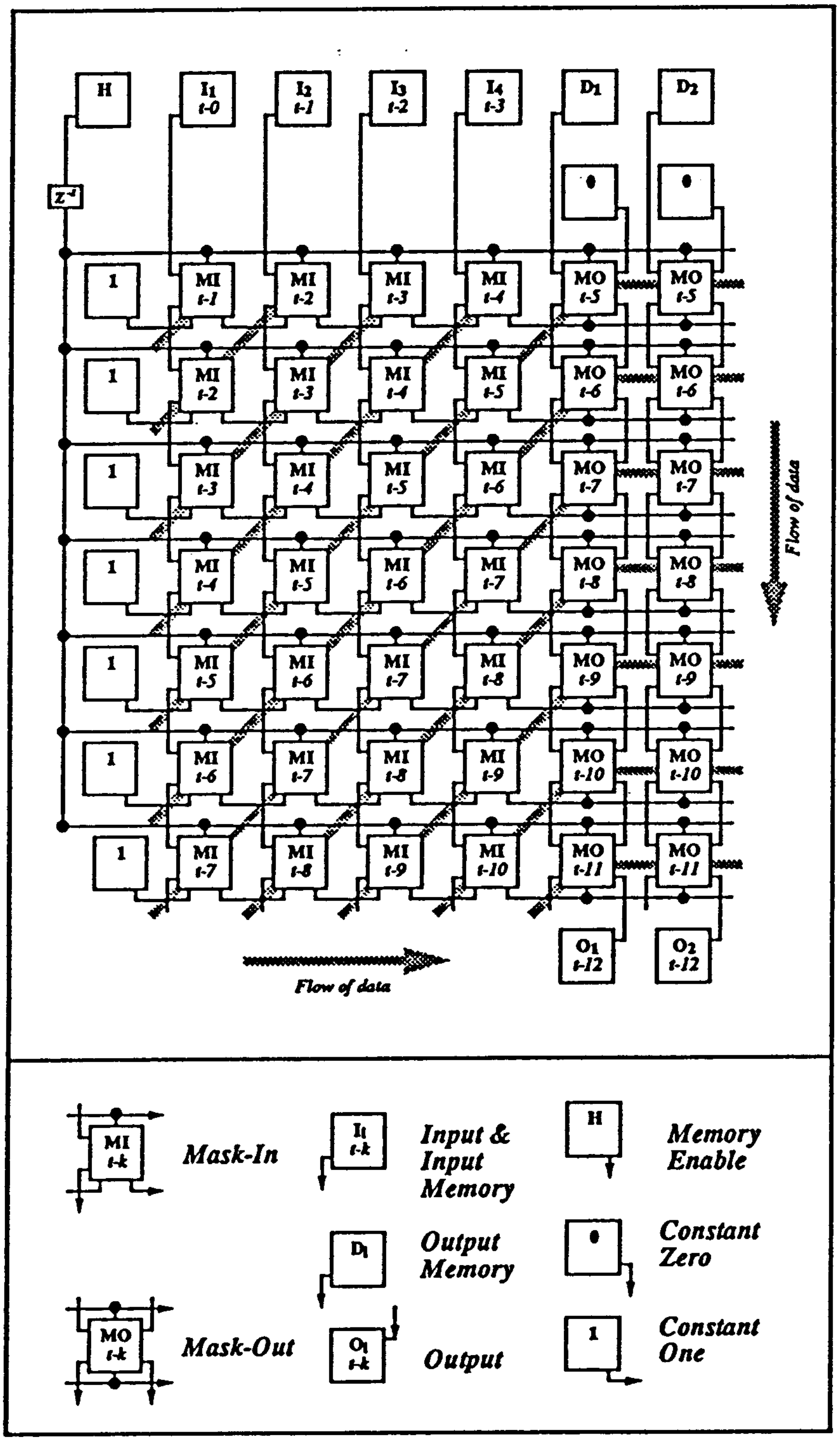


Figure 4 - Pipelined boolean matrix implementation

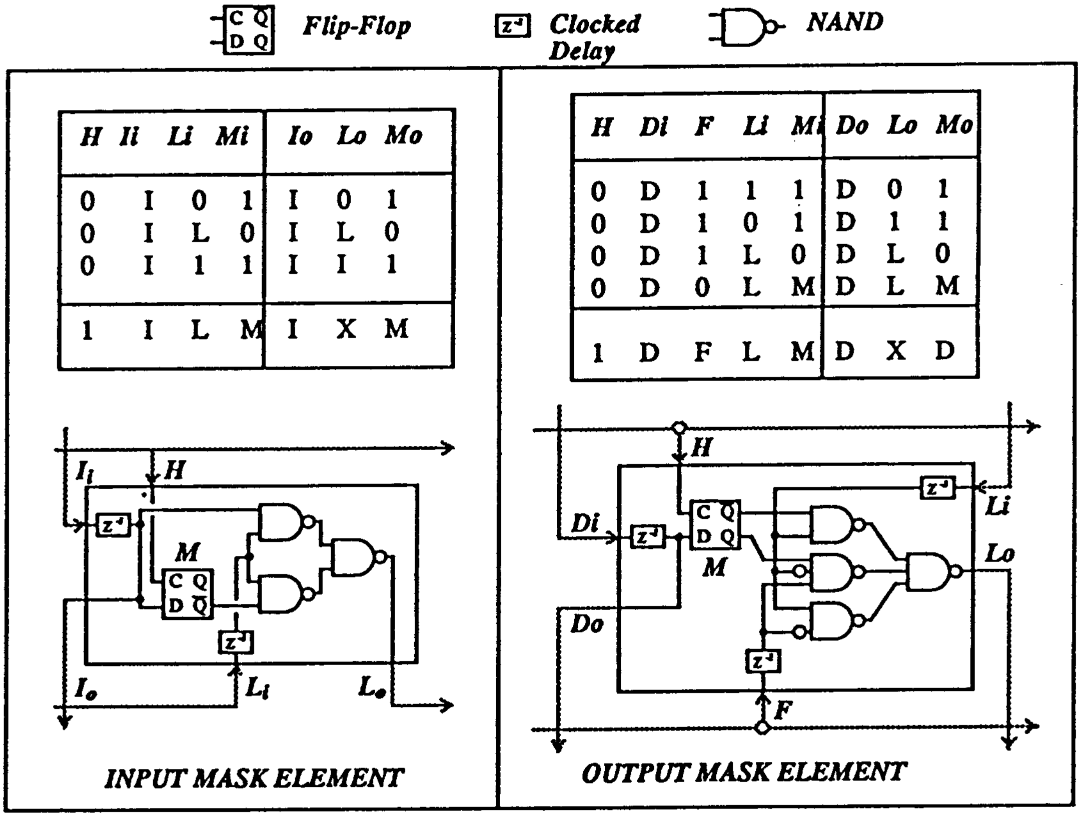
(clock signals not shown)

whether all of the connected input units are set to 1 (in C notation equivalent to *!~(~IM(h_i) | IL))*, and if so it XORs 1 with the value of all connected output units (in C notation equivalent to *OL = OL ^ OM(h_i)*). Assuming that the boolean matrix consists of vectors *IM(h_i), OM(h_i), IL* and *OL*, the hidden unit procedure may be viewed as a simple and very efficient bit vector operation (cf. Figure 3).

In a software simulation of perceptron operation, the boolean matrix approach requires actually much less space than a decision tree implementation of TBPs due to the existence of node-linking pointers adding extraneous space to the TBP representation. The matrix implementation on the other hand is very compact and uniform where each mask could be viewed as a continuous bitmap. The hardware realisation of a boolean matrix, even though more complex that of a decision tree, permits much more flexibility in the perceptron use and lends itself to retraining.

Each matrix mask may be substituted by a number of active elements (Figure 4). Some of these elements (denoted with MI) perform conjunction of input rows $(I_1,..., I_4)$, others (denoted by MO) calculate XOR of output columns (O_1 and O_2). To load the boolean matrix (Figure 3) into memory, the elements of the matrix must be stored at the appropriate mask locations. This can be done by setting $H = 1$ and presenting masks values in the form of vectors $(I_1,..., I_4, D_1,..., D_2)$ sequentially, in the reverse order, through a number of machine cycles equal the number of hidden units. Additional components generating constant signals are required to attain the correct network computation (0 and 1).

The logic of the most complex components (MIs and MOs) and their realisation in hardware is shown in Figure 5.



H	Ii	Li	Mi	Io	Lo	Mo
0	I	0	1	I	0	1
0	I	L	0	I	L	0
0	I	1	1	I	I	1
1	I	L	M	I	X	M

H	Di	F	Li	Mi	Do	Lo	Mo
0	D	1	1	1	D	0	1
0	D	1	0	1	D	1	1
0	D	1	L	0	D	L	0
0	D	0	L	M	D	L	M
1	D	F	L	M	D	X	D

Figure 5 - Hardware implementation of TBP masks (clock signals not shown).

The size of the proposed architecture is greater than that of a boolean decision tree (not considering pointers) and is equal to $N_H * (N_I + N_O)$, where N_H, N_I, and N_O are the number of hidden, input and output units respectively. The sequential complexity is equal to the number of elements in the perceptron representation, in parallel architecture, however, the input to output delay is proportional to the path length necessary for the signal to traverse from inputs to output, thus it is of order $O(N_I + N_H)$.

To increase the efficiency of the architecture, pipelining was implemented. The input values corresponding to an exemplar are fed into the processor matrix at consecutive machine cycles $t\text{-}0$ to $t\text{-}N_I\text{-}1$, and processed over $(N_I + N_H)$ machine cycles before returning output. At each of these cycles, the partial computation for each exemplar is performed by a number of processors placed along one of the matrix diagonals (shown by shadowed lines in Figure 4). Meanwhile, the processing of subsequent $(N_I + N_H)$ patterns can be initiated and continued concurrently. Thus, the effective process complexity for continuing sequence of inputs is of the order of $O(1)$.

The model permits retraining the perceptron to any other task requiring a similar number of inputs, hidden units and outputs (or less). It is also very easy to modularise the perceptron structure into a number of large input and output units consisting of MI and MO elements, thus resulting in a more economical architecture.

The simulation of the above mentioned architecture was successfully implemented in PROLOG and is fully described in the Appendix.

4.0 Conclusions

This paper proposed and analysed two different implementations of two-layer perceptrons with binary weights.

The first based on the binary decision trees is conceptually simple, memory efficient, fast in sequential machine simulation, but at the same time not easily lending itself for hardware realisation and if so virtually limited to a single configuration, due to the lack of retraining abilities.

On the other hand, the second implementation of TBP (boolean matrix) although far more complex and requiring more space, has a uniform planar architecture suitable for VLSI implementation and is very easily retrained. Also the introduction of pipelining gives it an effective processing speed of one machine cycle per pattern in continuous sequence of inputs. These features allow for efficient applications of TBPs in high volume transaction processing, e.g. on-line encoding and decoding, speech and image recognition, information retrieval, and other batch data processing activities.

To further optimise TBPs hardware implementation, additional research in alternative hardware architecture should be conducted. Currently, further approaches to reducing the processing delay and asychronous implementations are being investigated.

5.0 Acknowledgements

We would like to express our thanks to Mr. Greg Aumann and Mr. Chris Rowles for their helpful comments during preparation of this paper. Permission of Executive General Manager of Telecom Australia Research Laboratories to publish this paper is also acknowledged.

6.0 References

Bourland, H. and Wellekens, C.J. (1987): "Multilayer Perceptrons and Automatic Speech Recognition", *Proc. 1st IEEE Conf. on Neural Networks*, IV:407-415.

Cheok, K.C., Smith, J.C. and Fernando, J.P. (1989): "Neurocontrol of Auto-Lock-On Target-Tracking Sight Control System," in Hamza 1989, 129-133.

Cybulski, J.L., Ferra, H.L., Kowalczyk, A., and Szymanski, J. (1989a): "Determining Word Lexical Categories with a Multi-Layer Binary Perceptron," *Proc. Conf. IASTED Expert Systems and Neural Networks*, Honolulu, Hawaii; also Telecom Australia Research Laboratories, CSS Branch Paper 182.

Cybulski, J.L., Ferra, H.L., Kowalczyk, A., and Szymanski, J. (1989b): "Experiments with Multi-Layer Perceptrons," to appear in *Proc. Conf. AI'89*, Melbourne, Australia; also Telecom Australia Research Laboratories, CSS Branch Paper 181.

Cybulski, J.L. and Jennings, A. (1989): "COSIMO: A Massively Parallel Integrated Parser," *Proc. First Australia-Japan Joint Symposium on Natural Language Processing*, University of Melbourne, Melbourne, Vic, Australia, 15-27.

Gouhara, K., Imawa, K. and Uchikawa, Y. (1989): "Position and Size Representation by Neural Networks," in Hamza 1989, 148-153.

Hamza, M.H. (1989): *Expert Systems and Neural Networks: Theory and Applications*, Proc. Fifth IASTED Int. Symp., Honolulu, Hawaii, Anaheim: ACTA Press.

Kowalczyk, A. and Ferra, H. (1989): "Classification with Real and Modulo Perceptron," Telecom Australia Research Laboratories, CSS Branch Paper 191.

Kowalczyk, A. and Ferra, H. (1990): "A Study of Simplification of Multi-Layer Perceptrons Using Weight Rounding and Factorisation," Telecom Australia Research Laboratories Report.

Kukich, K. (1987): "Where do phrases come from: Some preliminary experiments in connectionist phrase generation," in Kempen, G. (edt.) *Natural Language Generation: New Results in Artificial Intelligence, Psychology and Linguistics, Proc. of the NATO Advanced Research Workshop*, 405-421.

Li, P.Y., Teng, T.L. and Wang, S.L. (1989): "A Decision Support System Based on Neural Networks," in Hamza 1989, 143-147.

Lippman, R.P. (1987): "An introduction to computing with neural nets," *IEEE ASSP Magazine*, April, 4-22.

Martinez, T.R. and Vidal, J.J. (1988): "Adaptive Parallel Logic Networks," *J. Parallel and Dist. Comp.*, V5, 26-58.

McCulloch, N., Ainsworth, W.A. and Linggard, R. (April 1988): "Multi-layer Perceptrons Applied to Speech Technology," *British Telecom Technology Journal*, V6, N2, 131-139.

Minsky, M.L. and Pappert, S.A. (1969): Perceptrons, Cambridge, Massachusetts: The MIT Press. (Expanded edition, 1988.)

Nicewarner, K.E. (1989): "An Artificial Neural Network Control System for a Six-Legged Autonomous Robot," in Hamza 1989, 134-138.

Reddy, G.N., Carroll, E.J. and Coleman, N.P. Jr. (1989): "Applications of Neural Networks: A Review," in Hamza 1989, 222-227.

Rumelhart, D.E., Hinton, G.E. and Williams, R.J. (1986): "Learning Internal Representations by Error Propagation," in Rumelhart, D.E. and McClelland, J.L. (eds): *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume1: Foundations*, Cambridge, Massachusetts: The MIT Press, pp 318-362.

A Appendix - Hardware Matrix Simulator (Prolog)

```
/* Elementary logic elements: */

/* gconst(C, V, S) - CONSTant output
/*           C - Clock stream
/*           V - Constant value
/*           S - Constant impulse
*/
gconst([], V, []).
gconst([C | CT], V, [V | VT]) :-
           gconst(CT, V, VT).

/* gconnect(C, I, O) - Connects O to I line
/*           C - Clock stream
/*           I - Input line
/*           O - Output line
*/
gconnect([], X, []).
gconnect([CH | CT], [], S) :-
           gconst([CH | CT], 0, S).
gconnect([CH | CT], [H1 | T1], [H1 | T2]) :-
           gconnect(CT, T1, T2).

/* gneg(C, I, O) - NEGates input
/*           C - Clock stream
/*           I - Input stream
/*           O - Output stream of input complements
*/
gneg([], X, []).
gneg([CH | CT], [], S) :-
           gconst([CH | CT], 0, S).
gneg([CH | CT], [1 | InT], [0 | OutT]) :-
           gneg(CT, InT, OutT).
gneg([CH | CT], [0 | InT], [1 | OutT]) :-
           gneg(CT, InT, OutT).

/* gdelay(C, I, O) - DELAY input by one cycle
/*           C - Clock stream
/*           I - Input stream
/*           O - Output stream
*/
gdelay(C, S, R) :-
           gconnect(C, [0 | S], R).

/* gand(C, I1, I2, O) - AND conjunction of inputs
/*           C - Clock stream
/*           I1, I2 - Two input streams
/*           O - Stream of input logical conjunctions
*/
gand1([], [], []).
gand1([1 | InT1], [1 | InT2], [1 | OutT]) :-
           gand1(InT1, InT2, OutT).
gand1([0 | InT1], [0 | InT2], [0 | OutT]) :-
           gand1(InT1, InT2, OutT).
gand1([0 | InT1], [1 | InT2], [0 | OutT]) :-
           gand1(InT1, InT2, OutT).
gand1([1 | InT1], [0 | InT2], [0 | OutT]) :-
           gand1(InT1, InT2, OutT).
gand(C, I1, I2, O) :-
           gconnect(C, I1, A1),
           gconnect(C, I2, A2),
           gand1(A1, A2, O).

/* gnand(C, I1, I2, O) - NAND gate
/*           C - Clock stream
/*           I1, I2 - Two input streams
/*           O - Stream of negated logical conjunctions
*/
gnand(C, I1, I2, O) :-
           gand(C, I1, I2, O1),
           gneg(C, O1, O).

/* gnand3(C, I1, I2, I3, O) - Multiple NAND
/*           C - Clock stream
/*           I1, I2, I3 - Three input streams
/*          O - Stream of negated logical conjunctions
*/
gnand3(C, I1, I2, I3, O) :-
            gand(C, I1, I2, O1),
            gand(C, O1, I3, O2),
            gneg(C, O2, O).

/* gflip(C, W, D, M, Q, R) - Memory FLIP-FLOP
/*          C - Clock stream
/*          W - Write flags (1 - write, 0 - read)
/*          D - Data stream written into memory (when C = 1)
/*          M - Stream of memory statuses
/*          Q - Stream of memory values
/*          R - Stream of Q complements
*/
gflip1(1, 1, M, 1, 1, 0).
gflip1(1, 0, M, 0, 0, 1).
gflip1(0, D, 0, 0, 0, 1).
gflip1(0, D, 1, 1, 1, 0).

gflip2([], [], M, [], []).
gflip2([W | WT], [D | DT], MP, [Q | QT], [R | RT]) :-
            gflip1(W, D, MP, MN, Q, R),
            gflip2(WT, DT, MN, QT, RT).

gflip(C, W, D, Q, R) :-
            gconnect(C, W, WC),
            gconnect(C, D, DC),
            gflip2(WC, DC, 0, Q, R).

/* gim(C, H, Ii, Li, Io, Lo) - Input Mask (MI)
/*          C - Clock stream
/*          H - Memory activator
/*          Ii - Input/data line
/*          Io - Copy of input/data
/*          Li - Unit combined input
/*          Lo - Unit combined output
*/
gim(C, H, Ii, Li, Io, Lo) :-
            gdelay(C, Ii, Iz),
            gdelay(C, Li, Lz),
            gdelay(C, H, Hz),
            gconnect(C, Iz, Io),
            gflip(C, Hz, Iz, Mq, Mr),
            gnand(C, Iz, Lz, Na),
            gnand(C, Lz, Mr, Nb),
            gnand(C, Na, Nb, Lo).

/* gom(C, H, Di, Li, F, Do, Lo) - Output Mask (MO)
/*          C - Clock stream
/*          H - Memory activator
/*          Di - Data line
/*          Do - Copy of data
/*          Li - Unit combined input
/*          Lo - Unit combined output
/*          F - Row activator for output firing
*/
gom(C, H, Di, Li, F, Do, Lo) :-
            gdelay(C, Di, Dz),
            gdelay(C, F, Fz),
            gdelay(C, Li, Lz),
            gdelay(C, H, Hz),
            gneg(C, Lz, Ln),
            gneg(C, Fz, Fn),
            gconnect(C, Dz, Do),
            gflip(C, Hz, Dz, Mq, Mr),
            gnand(C, Mr, Lz, Na),
            gnand3(C, Mq, Ln, Fz, Nb),
            gnand(C, Lz, Fn, Nc),
            gnand3(C, Na, Nb, Nc, Lo).

/* Perceptron building blocks:
/*          Any perceptron may be built of a constant signal components,
/*          input and output units. Their number and arrangement depends
/*          on the type and size of a perceptron (see example).
*/
```

```
c(C, V, L) :-
        gconst(C, V, L).
i(H, Ii, Io, Li, Lo) :-
        gim(H, H, Ii, Li, Io, Lo).
o(H, Di, Do, F, Li, Lo) :-
        gom(H, H, Di, Li, F, Do, Lo).

/* Sample perceptron:
/*
/* perc_4x2x7(Hs, Is, Ds, Os) - 4 input, 2 output, 7 hidden units.
/*      H (H) - Memory activator + clock
/*      Is (I1-I4) - Input streams
/*      Ds (D1-D2) - Data streams
/*      Os (O1-O2) - Output streams
*/

perc_4x2x7([H], [I1, I2, I3, I4], [D1, D2], [O1, O2]) :-

        c(H, 1, A11),                   /* MI constants */
        c(H, 1, A21),
        c(H, 1, A31),
        c(H, 1, A41),
        c(H, 1, A51),
        c(H, 1, A61),
        c(H, 1, A71),

        c(H, 0, L11),                   /* MO constants */
        c(H, 0, L12),

        i(H, I1, I21, A11, A12),        /* 1st hidden unit */
        i(H, I2, I22, A12, A13),
        i(H, I3, I23, A13, A14),
        i(H, I4, I24, A14, F1),
        o(H, D1, D21, F1, L11, L21),
        o(H, D2, D22, F1, L12, L22),

        i(H, I21, I31, A21, A22),       /* 2nd hidden unit */
        i(H, I22, I32, A22, A23),
        i(H, I23, I33, A23, A24),
        i(H, I24, I34, A24, F2),
        o(H, D21, D31, F2, L21, L31),
        o(H, D22, D32, F2, L22, L32),

        i(H, I31, I41, A31, A32),       /* 3rd hidden unit */
        i(H, I32, I42, A32, A33),
        i(H, I33, I43, A33, A34),
        i(H, I34, I44, A34, F3),
        o(H, D31, D41, F3, L31, L41),
        o(H, D32, D42, F3, L32, L42),

        i(H, I41, I51, A41, A42),       /* 4th hidden unit */
        i(H, I42, I52, A42, A43),
        i(H, I43, I53, A43, A44),
        i(H, I44, I54, A44, F4),
        o(H, D41, D51, F4, L41, L51),
        o(H, D42, D52, F4, L42, L52),

        i(H, I51, I61, A51, A52),       /* 5th hidden unit */
        i(H, I52, I62, A52, A53),
        i(H, I53, I63, A53, A54),
        i(H, I54, I64, A54, F5),
        o(H, D51, D61, F5, L51, L61),
        o(H, D52, D62, F5, L52, L62),

        i(H, I61, I71, A61, A62),       /* 6th hidden unit */
        i(H, I62, I72, A62, A63),
        i(H, I63, I73, A63, A64),
        i(H, I64, I74, A64, F6),
        o(H, D61, D71, F6, L61, L71),
        o(H, D62, D72, F6, L62, L72),

        i(H, I71, I81, A71, A72),       /* 7th hidden unit */
        i(H, I72, I82, A72, A73),
        i(H, I73, I83, A73, A74),
        i(H, I74, I84, A74, F7),
        o(H, D71, D81, F7, L71, O1),
        o(H, D72, D82, F7, L72, O2).
```

```
/* Test:
*/

?- perc_4x2x7(

  /* Clock and memory control */
  [     [0, 0, 0, 0, 0, 0, 1,
         0, 0, 0, 0, 0, 0, 0,   0, 0, 0, 0,   0, 0, 0, 0, 0, 0, 0,   0, 0]],

  /* Input masks and data */
  [     [0, 0, 1, 0, 1, 1, 0,                 1, 1, 1, 0, 1, 0, 0],
        [1, 0, 0, 1, 1, 0, 0,   0,            1, 1, 0, 1, 0, 1, 0],
        [1, 1, 1, 0, 0, 0, 0,   0, 0,         1, 0, 1, 1, 0, 0, 1],
        [0, 0, 0, 0, 0, 0, 0,   0, 0, 0,      1, 0, 1, 1, 0, 0, 0]],

  /* Output masks and data */
  [     [0, 0, 1, 0, 1, 0, 1,   0, 0, 0, 0,   0, 0, 0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 0,   0, 0, 0, 0,   0, 0, 0, 0, 0, 0, 0]],

  /* Results */
  [     O1, O2]).

  /* Output results
  O1 = [0, 0, 0, 0, 0, 0, 0,
         0, 0, 1, 1, 0, 1, 0,   0, 1, 1, 1,   1, 0, 0, 1, 1, 1, 1,   1, 1]
  O2 = [0, 0, 0, 0, 0, 0, 0,
         1, 0, 1, 0, 0, 1, 1,   1, 1, 1, 0,   0, 1, 1, 1, 1, 1, 1,   0, 0]

         Programming + H-Delay  I-Delay       Pattern-Matching        After
  */
```