Article

Linear data-driven architectures implementing neural network models

M.L. Marchesi,[1] G. Orlandi,[2] F. Piazza [3] and A. Uncini [3]

1. DIBE, University of Genoa, Genoa, Italy
2. Dip. INFOCOM, University of Rome 'La Sapienza', Rome, Italy
3. Dip. di Elettronica e Automatica, University of Ancona, Ancona, Italy

Abstract: *This paper presents a digital implementation of neural network models, based on a linear array of processors provided with a local memory and locally connected through two data buses, one of which is bi-directional. The main advantages of the proposed architecture are the locality of its connections, a high efficiency and the ability to be expanded and reconfigured easily. For these reasons, this architecture is very suitable both for VLSI implementation and for implementation through an array of DSP processors on a board. The implementation of the multilayer perceptron with back propagation learning algorithm and of the counterpropagation neural model is discussed in detail. It is shown that, despite the locality of the connections, the degree of parallelism achieved is very high.*

1. Introduction

Artificial neural networks (NN) are an emerging computational paradigm characterised by massive parallelism, robustness, fault tolerance and ability to learn by examples. Besides other approaches (such as software simulation on serial machines and analog or hybrid implementations), digital architectures are of great interest for the implementation of such systems owing to their flexibility and speed. However, mapping the NN fine-grained parallelism to parallel processors is a difficult task because interconnecting paths are inherently global and there are very many of them, while they have to be necessarily local and limited in number in digital architectures.

Among the many available models of NN, one of the most widely used is the feed-forward multilayer neural network (MLP) (Rumelhart *et al.* 1986). Some proposals have already been made in the literature for digital parallel implementations of such a neural network model (Beynon 1987; Forrest *et al.* 1987; Kamangar *et al.* 1989; Kung & Hwang 1988; Millan & Bofill 1989). However, the proposed architectures present some disadvantages: presence of non-local connections (Kung & Hwang 1988), difficulties of reconfiguration in terms of number and size of layers (Beynon 1987; Kung & Hwang 1988), communication overhead (Millan & Bofill 1989), and low efficiency when implemented with few processors and if the MLP is not regular (Kamangar *et al.* 1989).

This paper presents a digital implementation of the multilayer neural network model, based on a linear array of processors provided with a local memory, which tries to overcome these limitations. The main advantages of the proposed architecture are the locality of its connections, a high efficiency (particularly under certain assumptions on the number of neurons per layer) and the ability to be expanded and reconfigured easily. Preliminary results have shown that it can be used both to implement high granularity parallel processors circuits, with a one-to-one correspondence between neurons and processors (Piazza *et al.* 1989), and to map an MLP composed of many neurons to a limited number of processors (Piazza *et al.* 1990).

The paper is organised as following. Section 2 presents the proposed architecture in detail, assuming a one-to-one mapping between neurons and processing elements (PE). Section 3 shows how it is possible to apply the same architecture when the number of neurons is higher than the number of PEs. Finally, Section 4 shows how the proposed architecture can also be used to implement a non-MLP neural model efficiently:

the counter-propagation network (CPN) (Hecht-Nielsen 1988). This section is interesting because the behaviour of a CPN is inherently global, involving the computation of a maximum, and shows the flexibility of the proposed architecture. Moreover, with only slight modifications, the results can be applied to another widely-used NN model: Kohonen nets.

2. Fine-grained processors

2.1. The proposed architecture

The multilayer perceptron (MLP) is a well known neural network model (Rumelhart *et al.* 1986). A concise description is provided in Appendix 1, which the equations quoted here refer to. In the classical MLP model, the backward move used for network training is considered to be applied to the neurons of the network by an external supervisor. A neuron is therefore characterised only by equations (A.1) and (A.2). To realise an architecture which implements the MLP in both forward and learning operating mode, each PE (which corresponds to a digital neuron) should be able to compute not only equations (A.1) and (A.2) (the classical neuron) but also equations (A.3), (A.4), (A.5) and (A.6) (the BP algorithm) in the correct sequence. In designing such an architecture, some objectives should be kept in mind:

- keep the length of connections as short as possible;
- make a network 'configurable' in terms of number of layers and neurons per layer;
- maximise the computational efficiency in the forward operating mode, through pipelining.

The proposed architecture for a digital implementation of the MLP network (Figure 1) is shown in Figure 1(c). It is composed of a set of PEs and it has the simplest form for such a set: a linear sequence where every processor communicates only with its two nearest neighbours. The folding of the processors on a plane resembles a snake, hence the name of this architecture. Given the total number of PEs, the architecture can be configured to implement a particular MLP topology. Figure 1(b) shows the logical structure that the snake assumes when it is configured to operate as the three-layer network of Figure 1(a).

The snake is in fact just a pipeline processor. Its elements can be viewed as MIMD (Multiple Instruction-stream, Multiple Data-stream) machines, but they actually run the same replicated program. For this reason, it acts more as an SIMD (Single Instruction-stream, Multiple Data-stream) machine, where every PE performs the same operations on different data. Moreover, it is implemented as a data-driven (wavefront) architecture

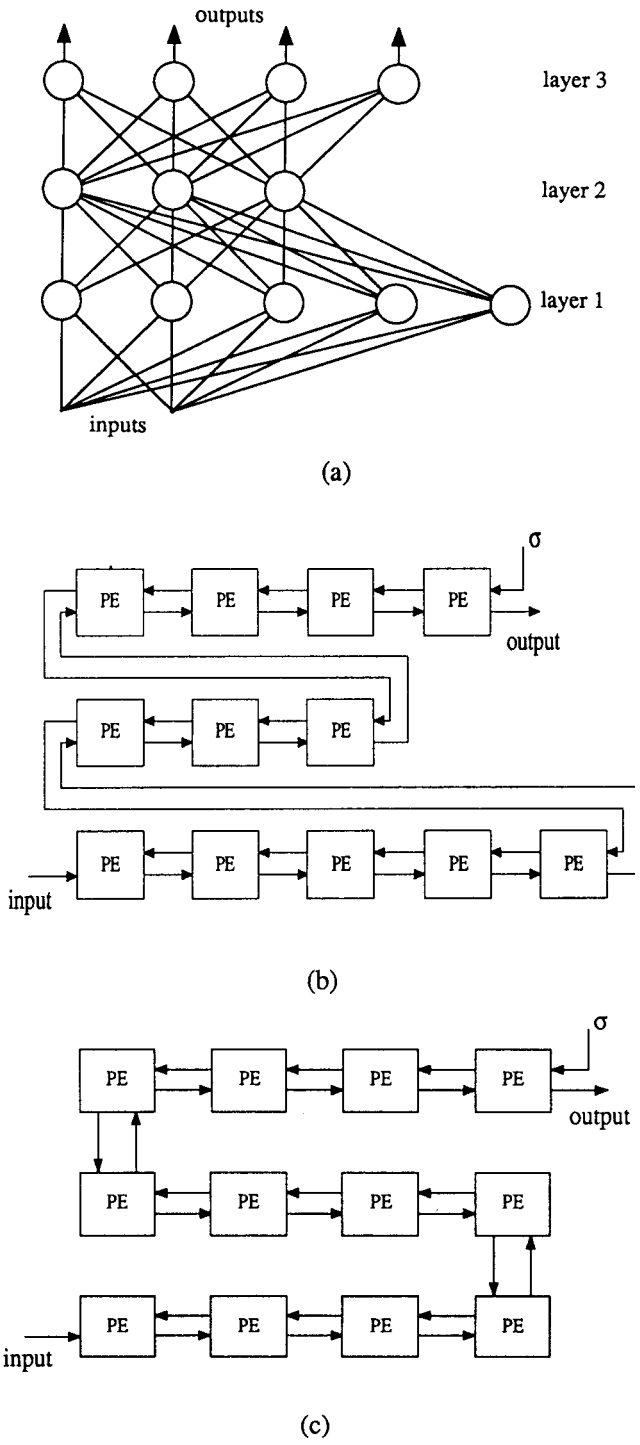


Figure 1: *The Multi Layer Perceptron and the proposed architecture: (a) the MLP; (b) the logical structure; (c) the physical structure.*

to maximise efficiency and to eliminate synchronisation problems for long structures. Finally, it implements the whole MLP model: properly activating the *input* side, it performs the forward mode and the forward move of the learning mode; properly activating the σ side it performs the backward move.

The content of a PE is described in a Pascal-like metalanguage in Figure 2(a), while Figure 2(b) shows its block-level internal structure.

A PE is provided with three input and three output data buses (**Chan1, Chan2, Chan3**); each bus has a validation signal (**Valid1, Valid2, Valid3**) that flags if the data carried on the bus are valid. Every processor also has some activation and control lines, not detailed in the figures, which are used to signal its ready or busy state and which let it be activated on the left side (forward move) or on the right side (backward move). A **Reset** line is also present to reset the PE in a known state.

The RAM memory holds the weights **W[ ]** of the neuron and its inputs **Exc[ ]**, i.e. the outputs of the preceding layer. The value **W[0]** is the biasing value (threshold) $\theta_k^{(s)}$ of equation (A.1). For uniformity reasons, it is taken into account setting the corresponding fictitious input **Exc[0]** to 1. The size of the memory limits the maximum size of neuron layers. The PE is also provided with some registers, a few pre-programmed registers which are used to configure the network, and three programmable counters which make it able to perform the correct sequence of operations. The PE must also be provided with a control unit and with an arithmetic and logic unit able to perform add and multiply operations. Depending on the actual implementation, the control unit is programmable or hardwired. Finally, two look-up tables are used to compute the squashing function $f()$ and its derivative $f'()$.

The snake can operate in three modes: configuration, forward and learning. During the configuration phase, the initial values of the weights **W[ ]**, and of the threshold **W[0]**, together with the value of **Eta**, are stored in each PE. Moreover the **Position**, **RevPosition** and **ExcNumber** registers are pre-programmed with respectively the position of the neuron in the layer, the complement to the width of the layer of this value and the number of inputs. Finally, **Exc[0]** is set to 1. A configuration of the PEs is needed only when the characteristics of the MPL change. A reset signal must be sent to all PEs to initialise them correctly (see Figure 3).

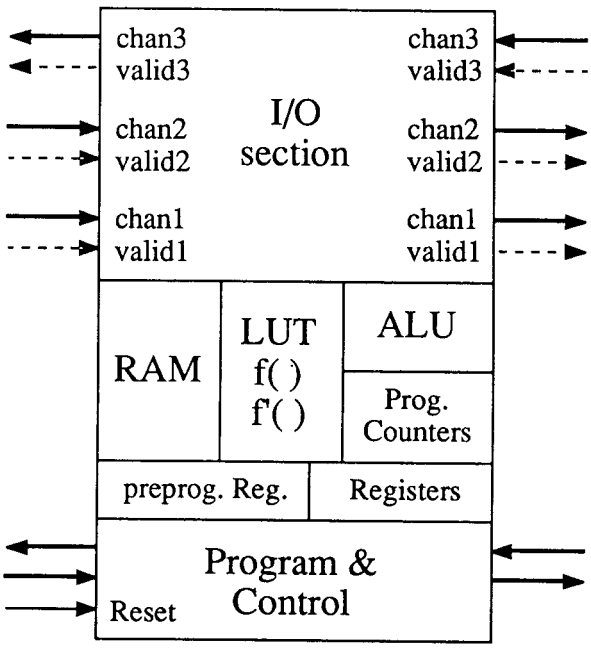
2.2. Forward mode

In forward mode, the computations of equations (A.1) and (A.2) are actually performed. An ideal, fully interconnected MLP would feed all the outputs of a layer into the neurons of the next layer, and compute the weighted sum and the output function in just one step. In the proposed snake, the connections among processing elements

```
TYPE  PreProgrammed = INTEGER;
      Counter       = INTEGER;
      ControlLine   = (on,off);
      Register      = REAL;
      RAM = ARRAY [0..MaxVal] OF REAL;

Neuron = RECORD
          { external lines }
          Reset       :  BOOLEAN;
          Left,Right  :  RECORD
            Chan1,Chan2,Chan3 :  Register;
            Valid1,Valid2 :  ControlLine;
            Valid3        :  ControlLine;
            < activation lines >
            < control lines >
          END;
          { internal entities }
          State          :  Counter;
          Index          :  Counter;
          WaitStates     :  Counter;
          RevPosition    :  PreProgrammed;
          Position       :  PreProgrammed;
          ExcNumber      :  PreProgrammed;
          Net,OutN,OutD  :  Register;
          Delta,EDelta   :  Register;
          Eta            :  Register;
          W,Exc          :  RAM;
         END;
```

(a)



(b)

Figure 2: *(a) Description of the content of a digital neuron in Pascal-like metalanguage; (b) the block-level internal structure of a PE (activation and control lines are not detailed).*

```
PROCEDURE Reset (VAR PE:Neuron);
{ reset actions }
BEGIN
  WITH PE DO
    BEGIN
      IF Reset=TRUE THEN
        BEGIN
          < signal busy state >
          Index := ExcNumber;
          State := 0;
          Net   := W[0];
          < signal ready state >
        END;
    END;
END;
```

Figure 3: *A description of the neuron reset operation, in Pascal-like metalanguage.*

are only local and the global connections among neurons belonging to adjacent layers are obtained by properly computing and passing data through the snake.

In particular, the operations start with feeding the first input sample into the **Chan1** bus of the first PE of the snake, turning **Valid1** line on and left-activating the neuron (see Figure 4). This neuron takes the input, multiplies it with the corresponding weight, accumulates the result in the **Net** register, passes both **Chan1** and **Chan2** (with their corresponding **valid** lines) to its right side and activates the next neuron. Thus a computational wavefront is created, travelling across the snake in the right-side direction. A new wavefront can be pipelined in the snake as soon as the first neuron is ready. After N_0 waves the first PE computes its output **OutN**. This value is immediately output on the **Chan2** bus and propagated concurrently with the last input. When the first output arrives to the end of the first layer, it should be fed into the second layer in the same way as the inputs were fed into the first layer. Therefore, a logical interconnection must be established between the two layers, as shown in Figure 5. The required physical connection can be performed by the last neuron of the layer or can be hardwired during configuration of the snake. When all input samples are fed into the first neuron of the snake, the **Valid1** line is turned off; this neuron, however, continues to be left-activated until all the outputs are read out from the last neuron of the last layer (**Chan2** bus).

The snake allows pipelining of the data to be processed. In fact, a second input vector can be input into the snake after the first and so on. However, there are some limitations to pipelining. In the hypothesis that all the layers have the same length and that the computation of the output function from the weighted sum of the inputs is performed in the same computing step as the last multiply and accumulate, the input data need not be separated and a full pipelining is achieved, with

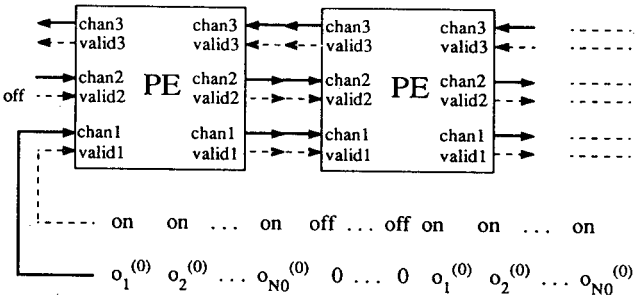


Figure 4: *The first two neurons of the snake during the forward mode (activation lines are not shown).*

all PEs computing at every step. If layer lengths differ, the length of a single input sequence must be at least equal to the longest layer of the net. If the longest layer is not the input one, enough no-ops (input data with **Valid1** off) have to be inserted between input sequences to satisfy this constraint, thus reducing the pipelining efficiency of the computation.

In real networks, unfortunately, the number of outputs is usually small compared to the number of neurons in the hidden layers, and thus a full pipelining will not be achieved. For such cases the coarse-grained architecture presented in Section 3 can provide a better exploitation of the PE's computing power. Figure 6 gives a detailed description of the forward operating mode of a single PE in a Pascal-like metalanguage.

2.3. Learning mode

The learning phase is divided in a forward and a backward move. The forward move performs the same operations as the forward operating mode but on a single input vector (without pipelining any more input patterns).

The backward move starts when the snake has computed all the outputs of the last layer. The $\sigma_k^{(N)}$ quantities of equation (A.3) are then externally computed and fed into the snake by the **Chan3** bus, turning **Valid3** on and right-activating the snake (see Figure 7). Each neuron passes the contents of **Chan3** and **Valid3** to its left side and activates the next neuron of the snake. Thus a computational wavefront is created, travelling across the snake in the left-side direction. When all the $\sigma_k^{(N)}$ samples are fed into the last PE of the snake, the **Valid3** line is turned off; this PE, however, continues to be right-activated until the end of the backward move.

After a number of activations equal to the position of the neuron in the last layer, each PE can compute the value of $\delta_k^{(N)}$ (**Delta**) according to equation (A.4), since **Chan3** holds the correct $\sigma_k^{(N)}$ quantity (**State = -1**). When all the PEs of the last layer have computed their **Deltas**, the $\sigma_k^{(N-1)}$ quantities begin to be computed in pipeline fashion, with each neuron perform-

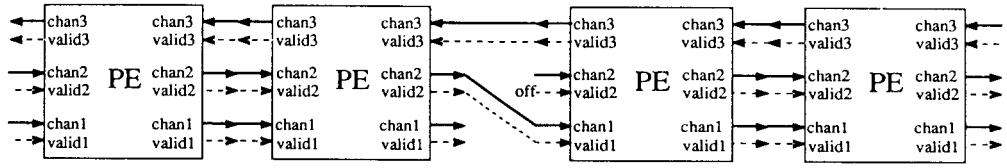


Figure 5: *The logical interconnections between layers (activation lines are not shown).*

```
PROCEDURE ForwardMode (VAR PE:Neuron);
{ forward mode operations }
BEGIN
  WITH PE DO
    BEGIN
      IF LeftActivated THEN
        BEGIN
          < signal busy state >
          Right.Chan1   := Left.Chan1;
          Right.Valid1  := Left.Valid1;
          Right.Chan2   := Left.Chan2;
          Right.Valid2  := Left.Valid2;
          { Valid1 on = process inputs }
          IF Left.Valid1=on THEN
            BEGIN
              Net := Net+W[Index]*Left.Chan1;
              Exc[Index] := Left.Chan1;
              Index := Index - 1;
              IF Index=0 THEN
                BEGIN
                  { compute outputs }
                  OutN     := Sigmoid (Net);
                  OutD     := DervSigmoid (Net);
                  Index    := ExcNumber;
                  Net      := W[0];
                  WaitStates := Position;
                  State    := State + 1;
                END;
            END;
          { if State=1 wait then output }
          IF State=1 THEN
            BEGIN
              WaitStates := WaitStates - 1;
              IF WaitStates=0 THEN
                BEGIN
                  Right.Chan2   := OutN;
                  Right.Valid2  := on;
                  State         := 0;
                END;
            END;
          < wait until next PE is ready >
          < activate next PE >
          < signal ready state >
        END;
    END;
END;
```

Figure 6: *A description of the neuron forward move, in Pascal-like metalanguage.*

ing one step of the sum in equation (A.3). The operations start in the last neuron (with the constraint that if `Valid3 = off` then `Chan3 = 0`) and propagate in the left-side direction (`State = -3`). However, each neuron must wait for the wave relative to the first $\sigma_k^{(N-1)}$; the quantity $\eta\delta_k^{(N)}$ is therefore computed at this time (`State = -2`).

At the left-side end of the last layer, the $\sigma_k^{(N-1)}$ are thus available and can be fed into the $(N-1)$-th layer in the same way as the $\sigma_k^{(N)}$ were fed into the N-th layer. Concurrently with this operation, the neurons of the N-th layer begin to adapt their weights and thresholds according to equations (A.5) and (A.6) (`State = -4`). The same sequence of operations is performed by all the neurons of the snake, layer after layer. When all the neurons of the first layer have computed their new weights and thresholds, the snake is ready to perform a new forward move or to operate in forward mode. Figure 8 shows a detailed description of the backward move of a single neuron in a Pascal-like metalanguage.

3. Coarse-grained architecture

In this section, we show how the proposed architecture can also be applied when the number of PEs is lower than the number of neurons. For this purpose, the forward and backward communication channels have to be connected at the ends of the array. Figure 9 shows how this is accomplished with four PEs, preserving the locality of the connections. It is worth noting that, in this case, the computational capabilities of the PEs must be higher than in the previous case, with more registers

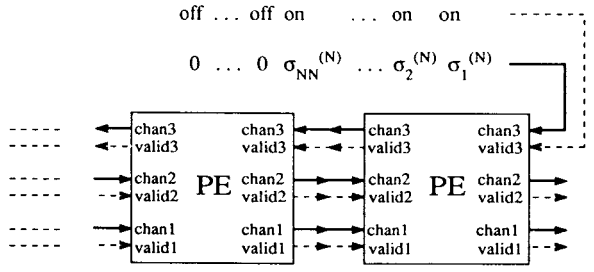


Figure 7: *The last two neurons of the snake during the backward move (activation lines are not shown).*

```
PROCEDURE BackwardMove (VAR PE:Neuron);
{ backward move operations }
BEGIN
  WITH PE DO
    BEGIN
      IF RightActivated THEN
        BEGIN
          < signal busy state >
          Left.Chan3 := Right.Chan3;
          Left.Valid3 := Right.Valid3;
          IF State = 0 THEN
          { wait for a valid input }
            IF Right.Valid3=on THEN
              BEGIN
                State := State - 1;
                WaitStates := Position;
              END;
            IF State = -1 THEN
            { wait for σ then compute δ }
              BEGIN
                WaitStates := WaitStates - 1;
                IF WaitStates = 0 THEN
                  BEGIN
                    Delta := Right.Chan3*OutD;
                    Left.Valid3 := off;
                    State := State - 1;
                    WaitStates := RevPosition;
                    EXIT;
                  END;
              END;
            IF State = -2 THEN
            { wait to properly synchronize }
              BEGIN
                WaitStates := WaitStates - 1;
                IF WaitStates = 0 THEN
                  BEGIN
                    State := State - 1;
                    Index := ExcNumber;
                    EDelta := Eta*Delta;
                    EXIT;
                  END;
              END;
            IF State = -3 THEN
            { build new σs }
              BEGIN
                Left.Chan3 := Right.Chan3
                              + W[Index]*Delta;
                Left.Valid3 := on;
                Index := Index - 1;
                IF Index = 0 THEN
                  BEGIN
                    State := State - 1;
                    Index := ExcNumber;
                    EXIT;
                  END;
              END;
            IF State = -4 THEN
            { adapts its own weights }
              BEGIN
                W[Index] := W[Index]
                            + EDelta*Exc[Index];
                Index := Index - 1;
                IF Index = -1 THEN State := 0;
              END;
            < wait until next PE is ready >
            < activate next PE >
            < signal ready state >
        END;
    END;
END;
```

Figure 8: *A description of the neuron backward move, in Pascal-like metalanguage.*

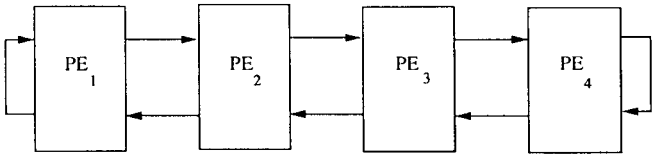


Figure 9: *The proposed architecture with N processors.*

and more memory. This can be accomplished by using microprocessors designed to operate concurrently as transputers or DSP processors.

It is worth noting that there has already been much work in this area, exploiting both bi-directional and 4-directional links among processors for implementing various neural networks models. Some works on this subject have already been cited in Section 1 (Beynon 1987; Millan & Bofill 1989); other works are Kerckhoffs *et al.* (1992), Margaritis & Evans (1992) and Zhang *et al.* (1990).

3.1. The basic architecture

For the sake of simplicity, we suppose that all the layers in the network contain the same number N of neurons and that this number is even. An example net is shown in Figure 10(a). By folding this network on its vertical symmetry axis and projecting the resulting net on the horizontal axis, the structure of Figure 10(b) is obtained. Noting that all the connections among layers are concentrated either inside the projected columns of neurons (the circles in the figure) or between them, it is easy to derive the computing structure of Figure 10(c), where each PE simulates a certain number of columns of neurons (two in the figure) and the two communication channels simulate all the synaptic connections between different PEs. The further connection between the output and the input of the first processing element is useful to simplify the programming task and to gain a higher efficiency. Note that, if P is the total number of PEs, N must be such that $N = 2KP$ where K is an integer greater than 0. The proposed structure may also be derived from the ring architecture in Kung & Hwang (1988), by folding it on the vertical symmetry axis.

Using the symbols reported in the Appendix, if m columns are assigned at every PE, then a generic processing element will simulate the neurons:

$$n_k^{(s)} \begin{cases} k=i,\ldots,i+(m-1) \\ \qquad\qquad\qquad\qquad s=1,\ldots,M \\ k=N-i,\ldots,N-i+(m-1) \end{cases}$$

with $i = 1, 2m-1, \ldots, N-2m-1$ depending from the PE. Figure 10(d) shows the internal logical structure of the processing chain; the k-th circle represents the process which simulates the k-th neuron of every layer in the network. Each PE has three operating modes: Configuration, Forward move and Backward move.

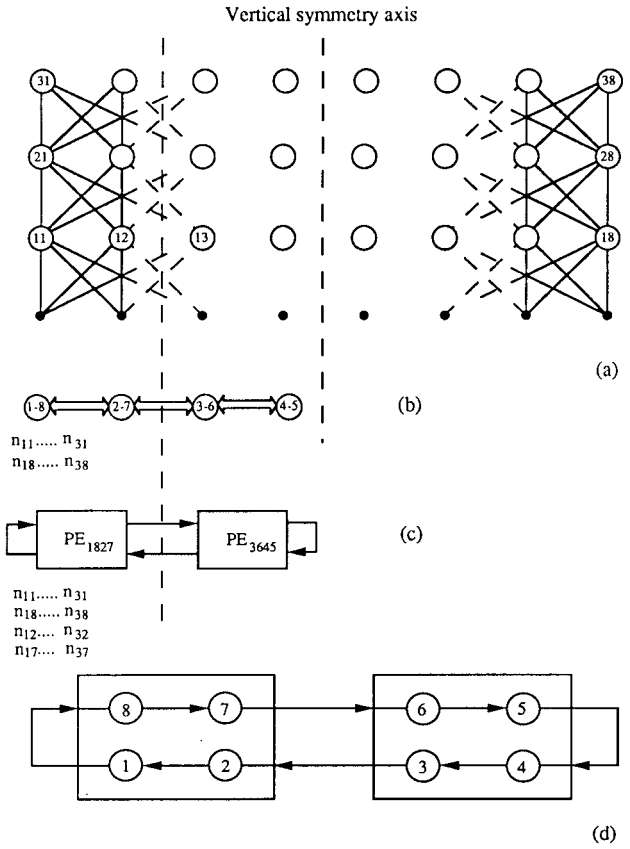


Figure 10: *(a) A generic MLP neural network; (b) the network obtained by folding (a) around the vertical symmetry axis and by projecting the result on the horizontal axis; (c) the computing architecture which implements (b); (d) the logical internal structure of (c).*

The Configuration phase must start after a reset. During this phase, the number of layers, their widths, the number of neurons to be simulated for each layer and the initial values of the corresponding weights and offsets are fed into each PE by a master processor (possibly a host) attached to the first processor of the chain.

To perform a Forward move, the following steps must be completed for each PE:

1. store the inputs relative to the simulated neurons of the first hidden layer;
2. compute, for every simulated neuron, the term of equation (A.1) relative to its present input;
3. move all the inputs one step clockwise, using the two communication paths;
4. if equation (A.1) is not completed go to step (2), else compute the new activation values of each neuron using equation (A.2);
5. repeat steps (2), (3) and (4) for the next layer, considering the newly computed activations as inputs.

When the last layer has also been simulated, the outputs of the network can be read from each PE. A description of the operations performed by a generic neuron $n_k^{(r)}$ is given in Figure 11.

To perform a Backward move, the following steps must be completed for each PE:

1. store the σ's quantities (externally computed by equation (A.3)), relative to the simulated neurons of the last layer;
2. compute, for every simulated neuron, the δs by equation (A.4) and the $\Delta\theta_k^{(r)}$ terms by equation (A.6);
3. compute, for every simulated neuron, the σ's quantities of the corresponding neurons of the lower layer. This can be done by pipelining into the structure all the σs to be computed starting from their own neuron sites. Each neuron adds a term of equation (A.3) to the flowing values, so that at the end of a complete cycle around the structure, each neuron has its own σ correctly computed. Note that this operation is not necessary for the neurons of the layer 1;
4. compute, for every simulated neuron, the weight variations using equation (A.5) by circulating the activations of the lower layer as in the Forward move;
5. repeat steps (2), (3) and (4) for the previous layer, up to the first layer.

The actual updating of the weights and offsets is performed either at each iteration or at the end of a training set. A description of the operations performed by

```
FOR s := 1 TO M DO
  BEGIN
                (s-1)
    input := o        ;
             k

       (s)      (s)
    net     := q     ;
       k        k

    j := k ;

    REPEAT

         (s)         (s)      (s)
      net     := net     + w     *  input ;
         k           k        kj

      j := j + 1 ;

      IF (j > N) THEN j := 1 ;

      { cyclic counter }

      IF (j <> k) THEN

        < move input one step clockwise >

            (s)                 (s)
      ELSE o     :=  f ( net      ) ;
            k                   k

    UNTIL (j = k) ;

  END;
```

Figure 11: *Pseudo-Pascal listing of the operations performed by the neuron $n_k^{(s)}$ during the forward move.*

```
FOR s := M DOWNTO 1 DO
  BEGIN

     (s)      (s)              (s)
    d     =  s     *  f' ( net     ) ;
     k        k                k

      (s)              (s)
    Dq     =  h  *  d     ;
      k                k

    IF ( s > 1 ) THEN

      BEGIN

                      (s)      (s)
        input  :=  d     *  w     ;
                      k        kk

        j  :=  k ;

        REPEAT

          < move input one step clockwise >

          j := j + 1 ;

          IF ( j > N ) THEN j := 1 ;

          IF ( j <> k ) THEN

                                        (s)      (s)
            input  :=  input  +  d     *  w
                                        k        kj

                 (s)
          ELSE  s     :=  input ;
                 k

        UNTIL ( j = k ) ;

      END ;

                (s-1)
    input  :=  o       ;
                k

    j := k ;

    REPEAT

       (s)         (s)
      Dw     :=  Dq     *  input ;
        kj          k

      j := j + 1 ;

      IF ( j > N ) THEN j := 1 ;

      IF ( j <> k ) THEN

        < move input one step clockwise > ;

    UNTIL ( j = k ) ;

  END ;
```

Figure 12: *Pseudo-Pascal listing of the operations performed by the neuron $n_k^{(s)}$ during the backward move.*

a generic neuron $n_k^{(s)}$, except for the weight and offset updating, is given in Figure 12.

It is easy to see that, under the assumed hypotheses, this architecture can obtain a 100% computational efficiency together with a low communication overhead. Moreover, the computational and communication phases are well balanced, since each internal non-trivial operation corresponds to a data movement among neurons.

3.2. Architecture extension

Although the constraints on the number of neurons in the previous section allow many useful MLP structures to be simulated, it is possible to extend the use of this architecture to more general MLP networks. In fact, the constraint that all layers must have the same width ($N_s = N$, $s = 0, 1, \ldots, M$) can be removed. There are three possible cases.

(1) The number N_s of neurons of layer s is: $N_s = 2K_sP$ with $K_s = 1, 2, 3, \ldots$ and $s = 0, 1, \ldots, M$. In this case, either more than one item is moved between two neurons ($N_{s-1} > N_s$) or more than one step clockwise is performed by one item across the communication path ($N_{s-1} < N_s$). The operations relative to the forward move of the neuron $n_k^{(s)}$ are described in Figure 13. The backward move is also modified in a similar way.

(2) The number N_s of neurons of layer s is: $N_s = (2/K_s)P$ with $K_s = 2, 3, \ldots$ and $s = 0, 1, \ldots, M$. In this case each neuron performs the same operations of case (a) but it is duplicated into two or more processors, since the number of neurons is less than the number of PEs. This strategy is not the most efficient one, i.e. the parallel computational capability of the machine is not fully exploited. In order to increase the overall efficiency, it is convenient to split the weights relative to a neuron among all the PEs which implement that neuron. During the forward move each processor computes some terms of summation (A.1) in parallel, while only the accumulation phase is duplicated over some PEs. A similar behaviour is also shown during the backward move. Thus this strategy allows multiplications to be optimised (since they exploit the full machine parallelism) but not additions (since they are duplicated). Moreover, data movements across the structure are minimised.

(3) The number N_s of neurons of layer s is such that it cannot satisfy case (1) or (2). By inserting 'null' neurons (i.e. neurons with their outputs clamped to zero), each number N_s can be transformed to one of the previous cases. Obviously, the price to pay is a reduced efficiency.

```
FOR s := 1 TO M DO
  BEGIN
    FOR i := 1 TO ri DO
                          (s-1)
      input    :=  o            ;
           i         k+i-1

       (s)        (s)
    net     :=  q      ;
       k          k

    j := (k-1) * ri + 1;

    REPEAT
      FOR i := 1 TO ri DO
           (s)           (s)      (s)
        net     :=  net      +  w         * input   ;
           k             k        kj+i-1          i

      j := j + ri;

      IF ( j > N     )  THEN j := 1 ;
                s-1

      IF (j <> ((k - 1) * ri + 1)) THEN

        < move input vector rd steps clockwise >

           (s)                 (s)
      ELSE o     :=  f ( net      ) ;
           k                   k

    UNTIL (j=((k-1) * ri + 1));
  END;
```

$$ri = \begin{cases} 1 & \text{if } N_{s-1} < N_s \\ N_{s-1} \text{ div } N_s & \text{if } N_{s-1} > N_s \end{cases}$$

$$rd = \begin{cases} 1 & \text{if } N_{s-1} > N_s \\ N_s \text{ div } N_{s-1} & \text{if } N_{s-1} < N_s \end{cases}$$

Figure 13: *Pseudo-Pascal listing of the operations performed by the neuron $n_k^{(s)}$ during the forward move in the case (a) of text.*

For the sake of clarity in Figures 11, 12 and 13 we have only shown the internal operations of a single simulated neuron. In an actual implementation, however, all the neurons of a single PE are sequentially simulated by a process inside the processor. Thus it is possible to enable the two neuron sub-array inside each PE (on the left–right and right–left communication path respectively) to have a different length, so that the constraints: N_s even, $N_s = 2K_sP$ with $K_s = 1, 2, 3, \ldots$, can be transformed into the less demanding $N_s = K_sP$ with $K_s = 1, 2, 3, \ldots$. Therefore, providing that either the layer widths are a multiple of the number of processors or vice versa, the overall efficiency of the architecture can be kept very high.

It is worth noting that for real networks, with layers composed of many neurons and with a sufficiently high number of PEs, the average performance increases quite linearly with P, although the constraints cannot be strictly satisfied, as will be shown in Section 5. Other implementations found in the literature (Kerckhoffs *et al.* 1992; Margaritis & Evans 1992; Zhang *et al.* 1990) use a careful scheduling system together with load balancing to increase efficiency. Such systems make use of four or more links among processors and are much more complicated than our approach as regards implementation, programming and testing, while not providing substantial advantages.

4. Implementing the CPN with the proposed architecture

4.1. The counterpropagation network

Although the presented architecture has been designed with the MLP with the BP learning algorithm in mind, its application to other NN models has also been exploited. In this section we present and discuss the implementation of Hecht-Nielsen's CPN (Hecht-Nielsen 1988).

Basically, CPN is a statistically optimum self-programming look-up table. Since CPN is a combination of Kohonen and Grossberg models, this implementation also gives useful hints for implementing these neural models on our architecture.

Let us denote a generic vector in R^n with $\mathbf{x}$ and a generic vector in R^m with $\mathbf{y}$. The purpose of CPN is to learn a mathematical mapping from R^n to R^m by a set of (possibly corrupted) examples $\{\mathbf{x}_i, \mathbf{y}_i\}$ and to reconstruct the best matching samples of corrupted inputs $(\mathbf{x}', \mathbf{y}')$. This task is accomplished by building a look-up table approximating the mapping. Figure 14 shows the CPN architecture, arranged in a different way than in Hecht-Nielsen (1988). Appendix 2 gives a detailed description of CPN learning and the appropriate definition of symbols.

4.2. CPN implementation with the linear array

Implementing CPN using the linear array of PEs is a non-trivial problem, as operations like the computation of the maximum and the notification of this maximum to all middle neurons are inherently sequential and involve passing data forward and backward in the array. Moreover, only $n + m$ weights in the middle layer are updated and only by the winner neuron for every input, which blocks the whole net until it is accomplished.

The proposed implementation, however, takes full advantage of the couple of forward data buses and exploits the parallel execution of operations as much as possible. The first P_0 PEs of the snake are assigned to the middle layer (middle PEs), while the subsequent P_1 PEs are assigned to x- and y-layers (outstar PEs). Data always pass in pairs through buses. To save time, at the expense of a greater memory requirement in PEs, the middle neurons' weights are updated in parallel for the winner neuron belonging to each PE. The updated values, however, actually become the new weights only for the winner middle neuron with maximum weighted sum, and are discarded by the others. In the following we shall assume that $P_0 \leq N$ because usually, in real applications, N is quite high. An extension to the case, $P_0 > N$, is possible, following the approach presented in the previous section.

Figure 15 shows the array and the mapping of middle, x- and y-neurons to PEs. In this implementation, each middle PE is assigned $K_0 = \lceil (n+m)/P_0 \rceil$ neurons, including possible dummy neurons if $N < K_0P_0$ (the operator '$\lceil x \rceil$' means 'the lowest integer greater than or equal to x'). The middle PEs store $K_0(n+m)$ weights but need an extra amount of memory to store the input vectors and the updated weights. So, the total amount of memory needed by middle neurons is $(K_0+1)(n+m)$ words. Outstar PEs hold $K_1 = \lceil (n+m)/P_1 \rceil$ x- and y-neurons. As outstar neurons have N weights, outstar PEs hold K_1N weights, needing the same amount of memory words.

The implementation of CPN on the array starts with the loading of x and y vectors components into all middle PEs. Such loading is performed by pipelining data in pairs through the snake. Each PE, when all input data are fed into its memory, starts the computation of the inner products I_i relative to its neurons. When this computation is finished, it finds the maximum sum among its neurons. Each PE computes the maximum weighted sum among all middle neurons, I_k, by comparing its local maximum sum with the maximum sum related to preceding PEs, which is passed forward through the snake immediately after the completion of the previous computation. The value of the highest weighted

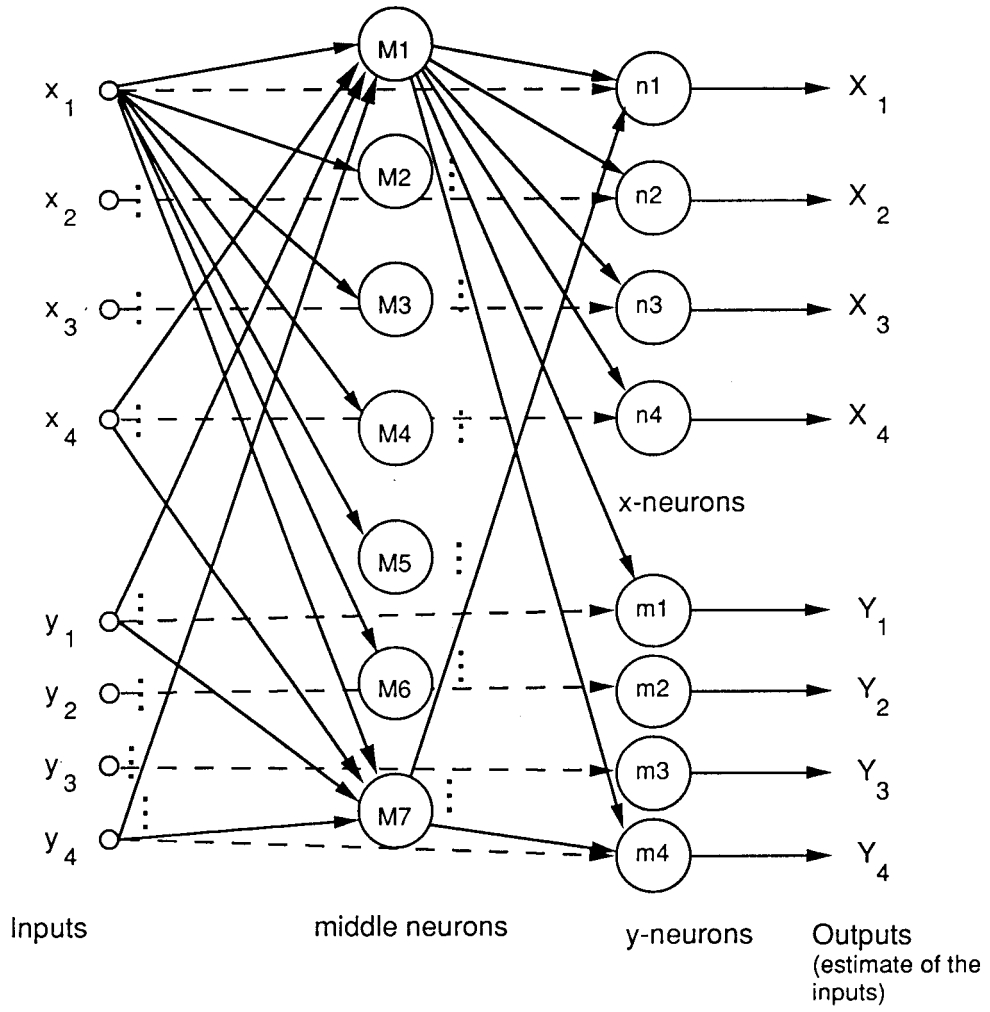


Figure 14: *The CPN architecture.*

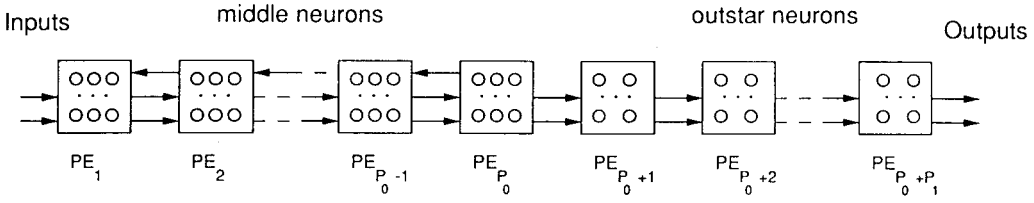


Figure 15: *Mapping CPN architecture to the array of processing elements.*

sum of the preceding neurons is forwarded in bus 1 and the index of the corresponding neuron in bus 2. This pair of data enables every PE to compare its own maximum weighted sum with that of the preceding neurons, and to forward the value and index of the higher. After this comparison, middle PEs compute the updated weights of their neuron with the highest weighted sum using equation (A.8) and store them apart. As already said, only the winner middle neuron will actually use these updated weights.

When the highest weighted sum of inputs arrives at the last PE of the middle layer, this PE is able to compute the highest weighted sum, I_k, definitely. The PE then passes the index of the winner neuron, k, both forward on bus 1 and backward on bus 3. As the index of the winner neuron is passed backward through middle PEs, the PE holding the winner neuron recognises its index and actually updates its weights. The other PEs discard updated weights. When the index of the winner neuron reaches the first PE of the array, this is ready to accept a new input, provided that its weight updating has already ended and that there is no data collision with weight updating of outstar PEs. Otherwise, a new input pair $(\mathbf{x}'', \mathbf{y}'')$ can be presented only after such updating is completed.

The processing made by PEs assigned to outstar layers is much simpler than that of middle layer PEs. Each PE takes in input and passes the presentation of the input pair $(\mathbf{x}, \mathbf{y})$ forward, holding the couple of $\mathbf{x}$ or $\mathbf{y}$ components corresponding to the neurons it is assigned to. If the first outstar PEs are assigned x-neurons and the last ones are assigned y-neurons, when the index of winner middle neuron, k, arrives in bus 1, the first PE passes it forward and, in the next moves, passes forward the weights $u'_{1k}, u'_{2k}, \ldots$ of the x-neurons it is assigned to. In the same move, it starts updating these values by applying equations (A.10) and (A.12) to them. Each other PE of the outstar layer, on receiving the index of the winner middle neuron on bus 1, holds it, then passes forward all the outputs of the preceding outstar PEs, and eventually forwards the proper weights of its neurons. In this way the architecture outputs the inputs to the two forward buses in the order: (x_1, x_2), $(x_2, x_3), \ldots, (x_{n-1}, x_n)$, $(y_1, y_2), \ldots, (y_{m-1}, y_m)$; then, on bus 1, the index of the winner middle neuron, k; and eventually the best estimate of the input pair, according to CPN, in the order: (u'_1, u'_2), $(u'_2, u'_3), \ldots, (u'_{n-1}, u'_n)$, $(u_1, u_2), \ldots, (u_{m-1}, u_m)$.

5. Performance evaluation

5.1. Architecture performance

In this section an estimate is given of the performance of the presented architectures, which results in formulae giving the speed-up with respect to a sequential computer with the same power of a single PE. These formulae are then used to compute the actual performance on some typical NN configurations found in real problems.

The main assumption behind the given performance estimate is that only four main operations are relevant to the computing time: multiplication, sum, data exchange on a bus between two different PEs and look-up table search. All other operations, including retrieving and writing data from and to memory, data exchange between registers, and retrieving and execution of microinstructions, are either negligible or are included in these four main operations. The times elapsed to execute these main operations are t_M, t_S, t_D and t_L, respectively. In a typical 1.5μ CMOS implementation with 10 bit word, these times could roughly be:

$$t_M = 40 \text{ ns}, \; t_S = 20 \text{ ns}, \; t_D = 15 \text{ ns}, \; t_L = 40 \text{ ns} \quad (1)$$

The total numbers of operations in MLP forward and backward moves, and in a CP move, are easily computed, as well as the corresponding times on a sequential computer. Data exchange time inside the processor is neglected, apart from time needed for data input and output. Using the symbols introduced in Appendices 1 and 2, these times are:

(Forward move in MLP)

$$\begin{aligned} \mathrm{TF} &= \sum_{h=1}^{M} N_h (t_M N_{h-1} + t_S N_{h-1} + t_L) \\ &\quad + (N_0 + N_M)\, t_D \end{aligned} \quad (2)$$

(Backward move in MLP)

$$\begin{aligned} \mathrm{TB} &= N_M (t_S + t_D) + \sum_{h=1}^{M-1} N_h N_{h+1} (t_M + t_S) \\ &\quad + \sum_{h=1}^{M} N_h \left[\begin{array}{l} t_M (2 + N_{h-1}) \\ \quad + t_S (1 + N_{h-1}) + t_L \end{array} \right] \end{aligned} \quad (3)$$

(Overall move in CP)

$$\begin{aligned} \mathrm{NC} &= (n+m)\left[N(t_M + t_S) + 2t_M + 4t_S \right] \\ &\quad + 2 \lceil (n+m)/2 \rceil \, t_D \end{aligned} \quad (4)$$

In CP we assume that data are input and output in pairs in the sequential PE, as in the snake. We characterise MLP networks with BP learning algorithm using three elapsed times: the time needed to process a single datum in forward mode, the time needed for a BP

step (one sample presented to the MLP, propagated forward and then backpropagated), and the time between two consecutive inputs in forward mode, when the MLP processes data pipelined one after the other. In the case of the simplest snake architecture, presented in Section 2, a step-by-step analysis of the performed computations leads to the following formulae yielding the total elapsed time to process a sample in forward and in backward mode:

(Forward move)

$$\mathrm{TF}_1 = \sum_{h=1}^{M} \left[\begin{array}{l} (N_{h-1} + N_h - 1) \\ \times (t_M + t_S + t_D) \\ + t_L \end{array} \right] + N_M t_D \quad (5)$$

(Backward move)

$$\begin{aligned} \mathrm{TB}_1 = {} & N_M t_D + t_S \\ & + \sum_{h=2}^{M} \left[\begin{array}{l} (N_{h-1} + N_h - 1) \\ \times (t_M + t_S + t_D) \end{array} \right] \\ & + (N_0 + 1)(t_M + t_S) \\ & + M(t_L + t_M) \end{aligned} \quad (6)$$

The formula yielding TB_1 is valid if the first layer, compared with subsequent ones, has a sufficient number of neurons to end its weight updating after all other layers. If not, equation (6) must be rewritten and becomes much more complicated. As with most MLP used, in practice the layer size does not increase from one layer to another; in such cases we will not go into any further detail.

The time between the corresponding components of two input sequences in pipelined forward mode is the following:

$$\mathrm{TP}_1 = N_{\max}(t_M + t_S + t_D) + t_L \quad (7)$$

where $N_{\max}$ is the number of neurons of the longest layer of the MLP.

In the case of the architecture presented in Section 3, these formulae become more complicated as a new parameter has to be taken into account: the number of neurons of each layer contained in a single PE. If P is the number of PEs and N_s the number of neurons belonging to the s-th layer, let us define the number m_s

$$\begin{array}{ll} m_s = \lceil N_s/2P \rceil & \text{if } N_s > P \\ & s = 0,1,\ldots,M \\ m_s = \dfrac{1}{\lfloor 2P/N_s \rfloor} & \text{if } N_s \leq P \end{array} \quad (8)$$

which is the number of columns of two neurons contained in each PE or, conversely, the reciprocal of the number of PEs containing the weights of a column of two neurons. The operator '$\lfloor x \rfloor$' means 'the biggest integer lower than or equal to x'. To compute the time of the backward move we need also to define the number X_s

$$\begin{array}{ll} X_s = 2m_s(t_M + t_L) & \text{if } N_s > P \\ & s = 0,1,\ldots,M \\ X_s = t_M + t_L + (1/m_s - 1)t_D & \text{if } N_s \leq P \end{array} \quad (9)$$

Lastly, let us define the numbers G_s and t'_D

$$G_s = \max\{N_s, 2P - 1\}, \quad s = 0,1,\ldots,M \quad (10)$$

$$t'_D = \begin{cases} 0 & \text{if } P = 1 \\ t_D & \text{if } P > 1 \end{cases} \quad (11)$$

The total times to process a sample in forward and backward mode are:

(Forward move)

$$\begin{aligned} \mathrm{TF}_2 = {} & \sum_{h=1}^{M} \left[\begin{array}{l} \lceil 2N_{h-1}m_h \rceil (t_M + t_S) \\ + G_{h-1}t'_D + t_L \lceil 2m_h \rceil \end{array} \right] \\ & + t_D(G_0 + G_M) \end{aligned} \quad (12)$$

(Backward move)

$$\begin{aligned} \mathrm{TB}_2 = {} & G_M t_D + t_s \lceil 2m_M \rceil \\ & + \sum_{h=1}^{M-1} \left[\begin{array}{l} \lceil 2N_{h-1}m_{h+1} \rceil (t_M + t_S) \\ + G_h t'_D \end{array} \right] \\ & + \sum_{h=1}^{M-1} \left[\begin{array}{l} \lceil 2(N_{h-1} + 1)m_h \rceil (t_M + t_S) \\ + G_{h-1}t'_D + X_h \end{array} \right] \end{aligned} \quad (13)$$

With this second architecture it is not possible to pipeline a new sample into the snake while the preceding is still being processed, so the time between two subsequent inputs is always TF_2.

Finally, the CP network can be characterised again in terms of the time it takes to process an input completely and of the time delay between two input samples fed one after the other in pipelining. Using the same notation of Appendix 2, the time needed to process an input sample can be easily computed by adding together the time needed to process data in the middle layer, T_1:

$$T_1 = \lceil (n+m)/2 \rceil t_D + [\ (n+m)K_0(t_M + t_S)\ ] + P_0 t'_D \quad (14)$$

(where, similarly to equation (11), $t'_D = 0$ if $P_0 = 1$ and $t'_D = t_D$ otherwise), and the time to pass data in the outstar layer, T_2

$$T_2 = (P_1 + 1 + \lceil (n+m)/2 \rceil) t_D \quad (15)$$

The time delay between two pipelined input samples, T_C, is the maximum among the time T_3 needed to notify the winner to the first PE

$$T_3 = T_1 + (P_0 - 1) t'_D \quad (16)$$

the time T_4 needed by the first PE to upgrade the weights of its winner neuron

$$T_4 = \lceil (n+m)/2 \rceil t_D + [\ (n+m)K_0(t_M + t_S)\ ] + t'_D + (n+m)(t_M + 2t_S) \quad (17)$$

and the time T_5 needed by the first outstar PE to upgrade the weights relative to the winner neuron

$$T_5 = \lceil (n+m)/P_1 \rceil (t_M + 2t_S) \quad (18)$$

Therefore

$$T_C = \max [T_3,\ T_4,\ T_5] \quad (19)$$

5.2. *Architecture evaluation*

To evaluate the proposed architectures, we chose some typical neural networks used in real problems. For MLP, we considered four different networks: two of them are quite small, but MLPs of this size are frequently found in the literature as processing blocks in bigger systems. In terms of number of neurons for each layer, their sizes are: 20, 15, 8 and 24, 10, 10, 1. The third MLP is bigger and is used by the authors to recognise ASCII characters of different fonts, given their pixel matrix 8 x 14. The size of the network is 112, 32, 8. Eventually, the fourth MLP is derived from Sejnowski & Rosenberg (1987) and has layers of size 203, 60, 26.

Table 1 shows a comparison of the times needed to compute a pipelined forward move and a backpropagation step for such MLPs, on a sequential computer and on the snake architecture presented in Section 2. The

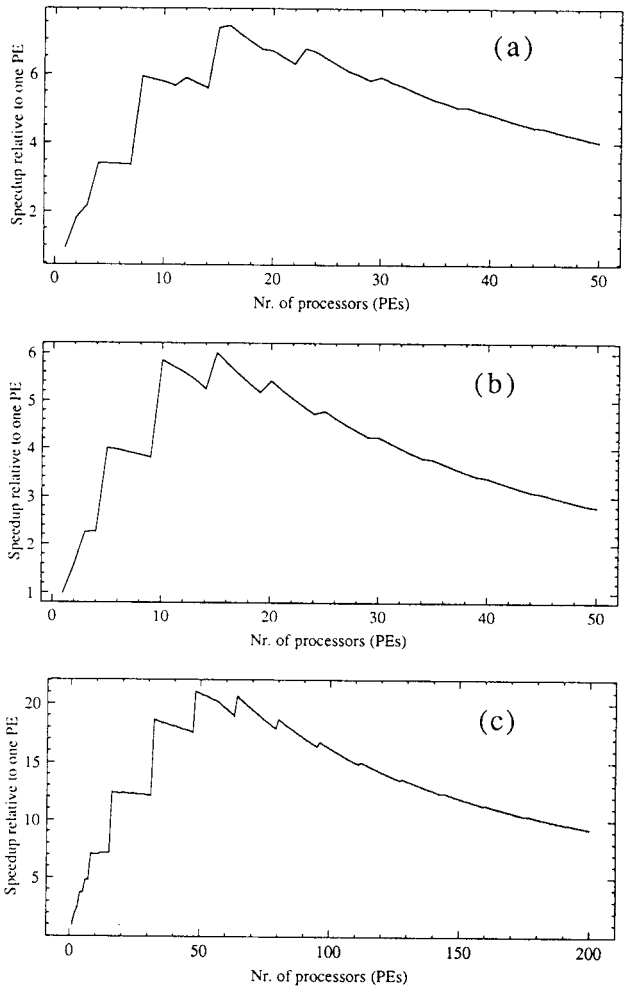


Figure 16: *Performance of three MLPs mapped on the snake, in terms of speed gain with respect to a sequential computer: (a) MLP 20/15/8; (b) MLP 24/10/10/1; (c) MLP 112/32/8.*

gain obtained with the snake when data are pipelined ranges from 55% to 75% of the number of PEs, exploiting the parallelism very well. During the learning, the gain is lower but still sensible, exploiting the parallelism at a rate of about 25% to 50%. The overall performance of the snake is also very good for the biggest MLP. Processing a BP move in 45.7 μs for the fourth MLP means a speed of 300 MCUPS (Mega Connection Updates Per Second) with the CMOS implementation mentioned above. This speed is also very competitive in comparison with data which recently appeared in the literature (Eppler *et al.* 1991).

The snake architecture proposed in Section 3 allows the number of PEs to be varied. Figure 16 shows the result for the first three MLPs of the above example, in terms of speed gain over a sequential computer, while Figure 17 shows the same results in terms of percentage of exploited parallelism. Only results on the forward

Table 1: *Performance of the simplest snake architecture on some MLP networks. Times to perform a forward move pipelining data and to perform a complete BP step on an I/O pair are reported. Times are in microseconds.*

	MLP (layer sizes)			
	20/15/8	24/10/10/1	112/32/8	203/60/26
Number of PEs	23	21	40	86
Forward move				
Time on sequential computer	26.5	22.2	233.8	831.3
Time on the snake	1.5	1.8	8.4	15.3
Number of equivalent PEs	17.2	12.1	27.7	54.5
Exploited parallelism (%)	74.9	57.5	69.3	63.3
Back propagation				
Time on sequential computer	62.4	52.8	485.4	1762.3
Time on the snake	7.6	8.7	23.8	45.7
Number of equivalent PEs	8.2	6.0	20.3	38.6
Exploited parallelism (%)	35.7	28.6	50.8	44.8

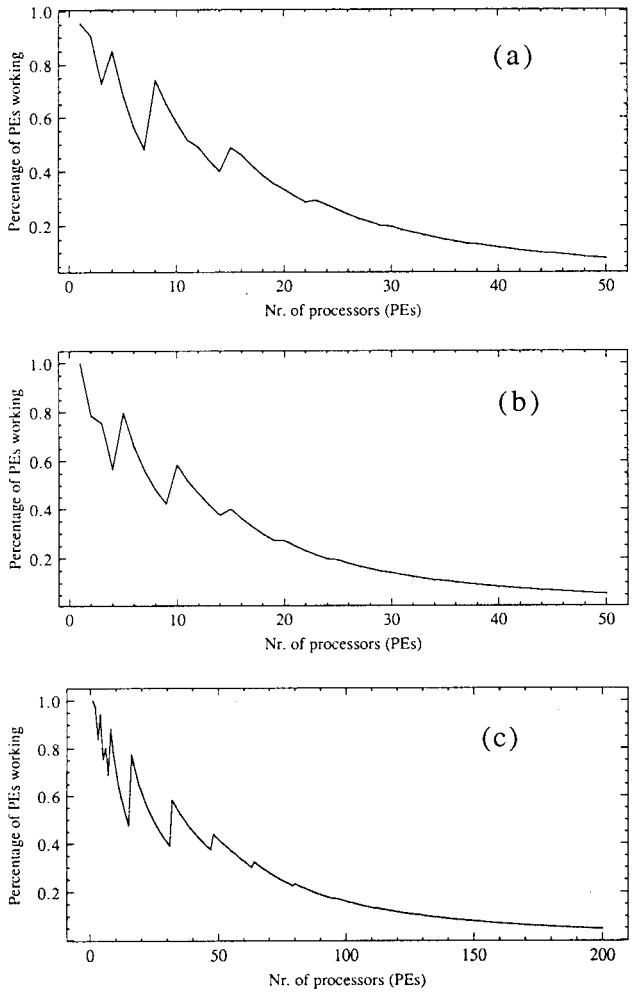


Figure 17: *Performance of three MLPs mapped on the snake, in terms of percentage of exploited parallelism: (a) MLP 20/15/8; (b) MLP 24/10/10/1; (c) MLP 112/32/8.*

move are presented because they are practically identical to those on backpropagation. The curves present a typical 'saw tooth' behaviour, due to the mismatch between the number of PEs and the number of neurons in the various layers. Most interesting is that by increasing the number of PEs, the speed gain increases up to a maximum and then decreases, because if the number of PEs is too high with respect to the number of neurons, communication overhead becomes predominant. For each MLP, there is an optimum compromise between speed and number of PEs.

This falling off of performance due to communications is a general problem in any distributed system. The two classic ways of reducing this problem are either having more links or more powerful PEs, and doing fewer communications. In our case, the first approach would lead to an architectural change from a linear architecture to a grid or a hypercube one — a solution studied by many authors but yielding a much more complicated system. The second approach could be pursued by looking for more powerful computing elements and for faster data buses — a task made easier by the constant technological advances in these fields.

To study the snake performance on CP networks, we considered a CPN able to classify inputs composed of 20 parameters in 200 different classes. For such a network, $M = 200$ while $n + m = 20$. The performance of such a CPN is expressed again both in terms of speed gain over a sequential computer and in terms of percentage of exploited parallelism, varying the total number of PEs from 2 to 100. For a given number of PEs, the number of P_0 and P_1 has been varied and the best combination is reported. Figure 18 shows the performance of CPN implementation on the snake versus the number of PEs. The parallelism is very high, up to almost 95% for 11 PEs. As in the MLP case, the parallelism of the snake tends to decrease with the number of PEs, after the

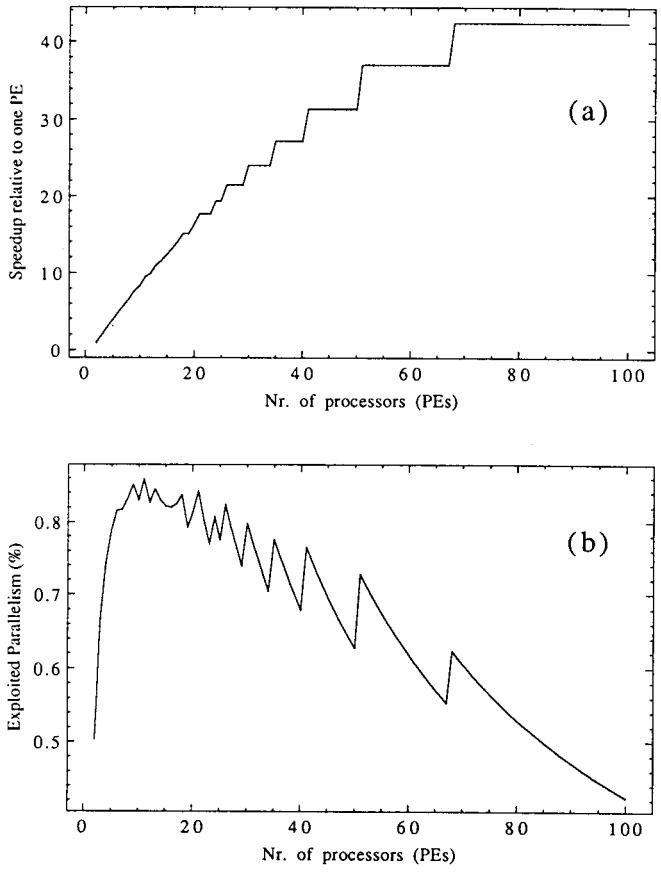


Figure 18: *Performance of the CPN with 200 middle neurons and 20 outstar neurons mapped on the snake, in terms of: (a) speed gain with respect to a sequential computer; (b) percentage of exploited parallelism.*

quoted peak for 11 PEs. By contrast, the speed-up tends to a limiting value of about 42 sequential PEs. With a further increase in the number of PEs, however, it can be shown that the speed-up tends to decrease. Figure 19 shows the performance of an architecture with 100 PEs, varying the number, P_0, of middle PEs. In this case, the efficiency of the snake increases up to a peak for 67 PEs and then tends to decrease. This kind of analysis is useful in determining the optimum percentage of PEs to assign to the middle layer versus the percentage of outstar PEs. The actual choice, however, also depends on the memory requirement for the various PEs.

6. Conclusions

In this paper we have presented an architecture for parallel digital implementation of neural networks. This architecture is a linear array and is very simple, but some characteristics of its data buses and a careful study of the implementation of the neural models on it give very good results and exploit the parallelism to a very high degree. Moreover, the architecture is completely general

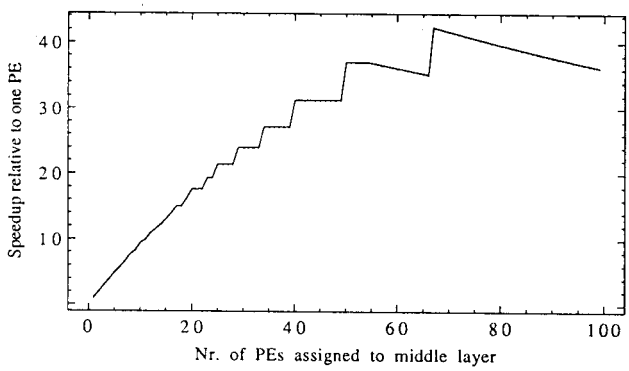


Figure 19: *Performance of the CPN with 200 middle neurons and 20 outstar neurons mapped on a snake of 100 PEs, varying the number of middle PEs, in terms of speed gain with respect to a sequential computer.*

and not bounded to a particular model or to a particular NN size.

We have already made a first implementation of the architecture on a network of transputers, where it can be easily simulated and actually executed in parallel, and our performance estimates were confirmed. The power of the snake, however, can only be fully exploited using an appropriately developed VLSI chip, or a special board of DSP processors, which is the work presently in progress in our laboratory.

Acknowledgement

This work was supported in part by the Consiglio Nazionale delle Ricerche of Italy under the Strategic Project 'Reti Neurali' and in part by the Ministero della Ricerca Scientifica e Tecnologica of Italy.

References

BEYNON, T. (1987) *A parallel implementation of the back propagation algorithm on a network of transputers*, Techn. Rep. 1000/21/88, RIPR, Royal Signals and Radar Establishment, Malvern, UK.

EPPLER, W., M. RINDERSPACHER AND M. RUDOLPH (1991) A digital signal processor for simulating back-propagation networks, in T. Kohonen, K. Makisara, O. Simula and J. Kangas (Eds.), *Artificial Neural Networks*, pp. 1565–1568, North-Holland.

FORREST, B.M., D. ROWETH, N. STROUD, D.J. WALLACE AND G.V. WILSON (1987) Implementing neural network models on parallel computers, *The Computer Journal*, **30**, 413–419.

HECHT-NIELSEN, R. (1988) Application of counter-propagation networks, *Neural Networks*, **1**, 131–139.

KAMANGAR, F.A., R.A. DUDERSTADT AND J.O. SMITH (1989) Efficient implementation of connectionist models on MIMD parallel processors using chordal ring topologies, *Proc. Int. Joint Conf. on Neural Networks*, Washington DC, June.

KERCKHOFFS, E.J.H., F.W. WEDMAN AND E.E.E. FRIETMAN (1992) Speeding up backpropagation training on a hypercube computer, *Neurocomputing*, **4**, 43–63.

KUNG, S.Y. AND J.N. HWANG (1988) Parallel architectures for artificial neural nets, *Proc. IEEE Int. Conf. on Neural Networks*, San Diego, California, pp. II-165–II-172.

MARGARITIS, K.G. AND D.J. EVANS (1992) Systolic implementation of neural networks for searching sets of properties, *Parallel Computing*, **18**, 325–334.

MILLAN, J. DEL R. AND P. BOFILL (1989) Learning by back-propagation: a systolic algorithm and its transputer implementation, *Int. J. Neural Networks*, **1**, 119–137.

PIAZZA, F., M. MARCHESI AND G. ORLANDI (1989) A digital 'snake' implementation of the back-propagation neural network, *Proc. IEEE ISCAS '89*, Portland, OR, May, pp. 2185–2188.

PIAZZA, F., M. MARCHESI, G. ORLANDI AND A. UNCINI (1990) Coarse-grained processor array implementing the multilayer neural network model, *Proc. IEEE ISCAS '90*, New Orleans, LA, May, pp. 2963–2966.

RUMELHART, D.E., J.L. MCCLELLAND AND THE PDP RESEARCH GROUP (1986) *Parallel Distributed Processing (PDP): Exploration in the Microstructure of Cognition (Vol. 1)*, MIT Press, Cambridge, MA.

SEJNOWSKI, T.J. AND C.R. ROSENBERG (1987) Parallel networks that learn to pronounce English text, *Complex Systems*, **1**, 145–168.

ZHANG, X., M. MCKENNA, J.P. MESIROV AND D.L. WALTZ (1990) The backpropagation algorithm on grid and hypercube architectures, *Parallel Computing*, **14**, 317–327.

Appendix 1

In the multilayer perceptron (MLP) the neurons are grouped in sequentially connected layers. Figure 1(a) schematically shows the MLP neural network. The layers are M, sequentially numbered from 1 to M. Each layer is denoted by a layer index; in the formulae, the layer index is reported as superscript, enclosed in parentheses. The number of neurons of the generic layer s is denoted as N_s. The neurons belonging to each layer are numbered in sequence, from 1 to N_s. Each neuron is connected to all the neurons of the two adjacent layers and to no other neuron. The neurons of the first layer have their inputs connected to the inputs of the network.

The neuron $n_k^{(s)}$, which occupies the k-th position of the generic s-th layer, is characterised by one output, $o_k^{(s)}$, and N_{s-1} inputs, which are the outputs of the neurons of the preceding layer. The inputs of the first layer neurons are the inputs of the network, $o_k^{(0)}$, $k = 1, \ldots, N_0$, where N_0 is the number of these inputs. For each input, j, the neuron maintains a weight $w_{kj}^{(s)}$ that is multiplied by the input to compute the quantity $\text{net}_k^{(s)}$, according to the formula

$$\text{net}_k^{(s)} = \sum_{j=1}^{N_{s-1}} w_{kj}^{(s)} o_j^{(s-1)} + \theta_k^{(s)} \qquad \text{(A.1)}$$

where $\theta_k^{(s)}$ is the offset of the neuron. The neuron output is computed by applying a non-linear, bounded function $f()$, the squashing function, to the $\text{net}_k^{(s)}$ value:

$$o_k^{(s)} = f(\text{net}_k^{(s)}) \qquad \text{(A.2)}$$

There are two operating modes of MLP: forward and learning. In forward mode, N_0 input signals are fed into the network and for each layer the outputs are computed and fed into the neurons of the next layer, up to the last layer, according to equations (A.1) and (A.2). In this mode, the weights and the thresholds are kept constant and determine the global behaviour of the network and its ability to process signals.

In learning mode, the weights and the offsets are adjusted in order to obtain the required network behaviour. The backpropagation (BP) is one of the most widely used algorithms in MLP performing such operation (network training). The BP requires a number of forward moves (computation of $o_k^{(M)}, k = 1, \ldots, N_M$, for a given input vector by equations (A.1) and (A.2)), each followed by a backward move, during which weight and offset adjustments occur. A network behaviour consists of a set of input/output pairs, each composed of N_0 input signals $o_k^{(0)} (k = 1, \ldots, N_0)$, and N_M corresponding output signals $d_j (j = 1, \ldots, N_M)$. The backward move consists of the computation of the following equations:

$$\sigma_k^{(s)} = \begin{cases} d_k - o_k^{(s)} & \text{for } s = M \\ \sum_{j=1}^{N_{(s+1)}} w_{jk}^{(s+1)} d_j^{(s+1)} & \text{for } s = 1, \ldots, M-1 \end{cases} \qquad \text{(A.3)}$$

$$\delta_k^{(s)} = \sigma_k^{(s)} f'(\text{net}_k^{(s)}) \qquad k = 1, \ldots, N_s \qquad \text{(A.4)}$$

$$\Delta w_{kj}^{(s)} = \eta \delta_k^{(s)} o_j^{(s-1)} \qquad k = 1, \ldots, N_s \; ; \; j = 1, \ldots, N_{s-1} \qquad \text{(A.5)}$$

$$\Delta\theta_k^{(s)} = \eta\,\delta_k^{(s)} \qquad k = 1, \ldots, N_s \tag{A.6}$$

where $f'()$ is the derivative of the squashing function $f()$, η is a constant (learning rate), and $\Delta w_{kj}^{(s)}$ and $\Delta\theta_k^{(s)}$ are respectively the weight and offset increments. By applying equations (A.3–A.6) iteratively from the last ($s = M$) to the first ($s = 1$) layer, the backward move is completed. During BP, input/output pairs defining the network behaviour are repeatedly applied to the network until a satisfactory learning is obtained. The actual updating of the weights and offsets is performed either at each iteration or at the end of a training set (in which case $\Delta w_{kj}^{(s)}$ and $\Delta\theta_k^{(s)}$ accumulate the variations at each iteration).

Appendix 2

In CPN, the neurons are divided into three layers: a middle layer and two outstar layers (numbered respectively 3, 2 and 4 in Hecht-Nielsen (1988)). The middle layer is composed of N neurons and performs the task of classifying the inputs in N different classes. We will refer to its neurons as middle neurons. The middle layer is fully interconnected with both outstar layers and can exchange data with them; it also receives the external inputs to the network.

The two outstar layers are composed of n and m neurons respectively, and their purpose is to reconstruct an input pair $(\mathbf{x}, \mathbf{y})$ to the network optimally. We will refer to these layers as x-layer and y-layer and to their neurons as x-neurons and y-neurons, respectively. Each outstar layer receives data from the middle layer and from the corresponding inputs ($\mathbf{x}$ for x-layer, $\mathbf{y}$ for y-layer). The outputs of these layers are the outputs of CPN. As a matter of fact, the outstar layers have the same behaviour and can be considered just as a single outstar layer, composed of $n + m$ neurons. In forward-only CPN, only the y-layer is present.

The CPN works as follows:

1. An input pair $(\mathbf{x}, \mathbf{y})$ is fed into the middle layer. The total number of scalar inputs is $n + m$. Each middle neuron computes a weighted sum of all these inputs. Let us call I_i the sum computed by the i-th middle neuron:

$$I_i = \sum_{j=1}^{n} w_{ij} x_j + \sum_{j=1}^{m} V_{ij} y_j \tag{A.7}$$

2. The maximum I_i is computed among middle neurons. Let us assume that the maximum is I_k — the sum of the k-th middle neuron.

3. The output z_i of every middle neuron is set to 0, except for that of the k-th neuron, which is set to 1.

4. The weights $\mathbf{w}$ and $\mathbf{v}$ belonging to the k-th neuron are updated according to the following equations:

$$\begin{aligned} \Delta w_{kj} &= \alpha\,(x_j - w_{kj}) \quad j = 1, \ldots, n \\ \Delta v_{kj} &= \alpha\,(y_j - v_{kj}) \quad j = 1, \ldots, m \end{aligned} \tag{A.8}$$

5. The outputs z_i, together with network inputs x_j, are fed into x-layer neurons. The i-th neuron first outputs the i-th component of the best estimate of input $\mathbf{x}$, corresponding to k-th class, and then updates this estimate:

$$\text{output of } i\text{-th } x\text{-neuron}\ ,\quad u'_{ik} \tag{A.9}$$

$$\text{update of } u'_{ik}\ ,\quad \Delta u'_{ik} = (x_i - u'_{ik}) \tag{A.10}$$

6. The same operation is made on y-neurons, referring to input $\mathbf{y}$:

$$\text{output of } i\text{-th } y\text{-neuron}\ ,\quad u_{ik} \tag{A.11}$$

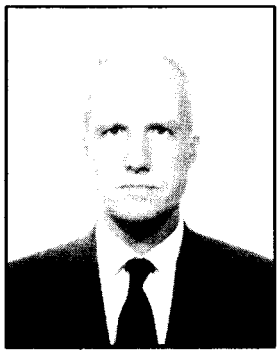
$$\text{update of } u_{ik}\ ,\quad \Delta u_{ik} = (x_i - u_{ik}) \tag{A.12}$$

The CPN outputs are the outputs of outstar layers. They are a pair $(\mathbf{x}', \mathbf{y}')$ that, after learning through a proper number of example pairs, constitutes the best estimate of the input pair $(\mathbf{x}, \mathbf{y})$.

The authors

Michele Marchesi

Michele Marchesi graduated from the University of Genoa, Italy, in Electronic Engineering in 1975 and Applied Mathematics in 1980. From 1976 to 1987 he was a Researcher at the Institute for Electronic Circuits, CNR Genoa. From 1987 to 1990 he was Associate Professor at the University of Ancona, Italy. Since November 1990 he has been Associate Professor of Network Theory at the Department of Biophysical and Electronic Engineering, University of Genoa. His main fields of interest are optimisation techniques applied to circuit design and neural networks.



Gianni Orlandi

Gianni Orlandi graduated from the University of Rome 'La Sapienza' in Electrical Engineering in 1972. From 1972 to 1978 he was with the U. Bordoni Foundation in Rome where he worked on circuit theory and digital signal processing. In 1978 he joined the Department of Information and Communication of the University of Rome 'La Sapienza', first as Assistant Professor and then as Associated Professor of Electrical Engineering. From 1986 to 1989 he was Professor at the Department of Electronics and Automatics of the University of Ancona, Italy, and since 1989 has been Professor of Electrical Engineering at the Department of Information and Communication of the University of Rome 'La Sapienza'. His research interests are in the areas of circuit theory, spectral estimation, array processing, parallel algorithms, VLSI parallel architectures and neural networks.

Francesco Piazza

Francesco Piazza graduated in Electrical Engineering from the University of Ancona, Italy in 1981. From 1981 to 1983 he worked on CCD-image processing at the Physics Department of the University of Ancona. In 1983 he worked at the Olivetti OSAI software development centre. In 1984 he was a consultant in the field of microprocessor-based systems and radar displays. In 1985 he joined the Department of Electronics and Automatics of the University of Ancona as Researcher in Electrical Engineering. He is now Associated Professor of Electrical Engineering at the same institution. His current research interests are in the areas of circuit theory and digital signal processing and include parallel algorithms and VLSI architectures, neural networks and DSP software systems.

Aurelio Uncini

Aurelio Uncini received a 'laurea' degree in Electrical Engineering from the University of Ancona, Italy in 1983. From 1984 to 1986 he was with the U. Bordoni Foundation in Rome, engaged in research associated with the Department of Electronics and Automatics, University of Ancona. His research interests include digital signal processing, digital circuit theory, spectrum analysis and neural networks.