Systolic-Based Hardware Realisation of Hopfield Neural Network

Jacek Mazurkiewicz*, Wojciech Zamojski*

Institute of Engineering Cybernetics, Wroclaw University of Technology

***Resumo* - O artigo propõe uma nova metodologia para a simulação de redes neuronais de Hopfield apoiada numa arquitectura sistólica em anel. A metodologia relaciona-se com os métodos de aprendizagem: regra de Hebb e regra Delta, mas também com a fase de acesso aos dados. A discussão fixa-se nas operações realizadas durante os seguintes passos de cada algoritmo e nos dados transferidos entre as unidades de cálculo. A eficiência é discutida na base dos parâmetros modificados definidos por S.Y. Kung.**

***Abstract* - The paper proposed a new methodology for Hopfield neural network simulation based on ring systolic array structure. The methodology is related to training methods: Hebbian and Delta rules as well as to retrieving phase. The discussion is focused on operations which are realized during the following steps of each algorithm and the data transferred among the calculation units. The efficiency is discussed based on the modified set of parameters defined by S.Y. Kung.**

I. INTRODUCTION

The paper is focused on the method of implementation of Hopfield neural network algorithms using ring systolic arrays – an example of SIMD architecture. The main assumption is about partial parallel realisation of learning algorithms as well as retrieving phase using the same processing structure. The proposed methodology creates the theoretical basis for hardware realisation of Hopfield neural network and could easily adopted for other recurrent nets. The methodology is discussed based on the following assumptions:

- the outcome of algorithms realised true to proposed methodology is exactly the same like the outcome of classical Hopfield neural network algorithms,
- the proposed methodology allows to create a universal structure both for learning algorithms and retrieving phase of Hopfield neural network,
- the systolic structure is realized using only digital elements, input and output data are represented in binary code,
- the number of neurons of Hopfield net is unrestricted, a number of processors can be limited.

II. HOPFIELD NEURAL NETWORK ALGORITHMS

The binary Hopfield net has a single layer of processing elements, which are fully interconnected - each neuron is connected to every other unit. Each interconnection has an associated weight. We let $\boldsymbol{w_{ji}}$ denote the weight to unit $\boldsymbol{j}$ from unit $\boldsymbol{i}$. In Hopfield network, the weight $\boldsymbol{w_{ij}}$ and $\boldsymbol{w_{ji}}$ has the same value. Mathematical analysis has shown that when this equality is true, the network is able to converge [1, 3]. The inputs are assumed to take only two values: 1 and 0. The network has $\boldsymbol{N}$ nodes containing hard limiting nonlinearities. The output of node $\boldsymbol{i}$ is fed back to node $\boldsymbol{j}$ via connection weight $\boldsymbol{w_{ij}}$.

A. Retrieving phase

During the retrieving algorithm each neuron performs the following two steps [2]:

- computes the coproduct:

$$\varphi_p(k+1) = \sum_{j=1}^{N} w_{pj} v_j(k) - \theta_p \qquad (1)$$

$\boldsymbol{w_{pj}}$	- weight related to feedback signal,
$\boldsymbol{v_i(k)}$	- feedback signal,
$\boldsymbol{\Theta_p}$	- bias

- updates the state:

$$v_p(k+1) = \begin{cases} 1 & \text{for} \quad \varphi_p(k+1) > 0 \\ v_p(k) & \text{for} \quad \varphi_p(k+1) = 0 \\ 0 & \text{for} \quad \varphi_p(k+1) < 0 \end{cases} \qquad (2)$$

The process is repeated for the next iteration until convergence, which occurs when none of the elements changes state during any iteration:

$$\forall_p \quad v_p(k+1) = v_p(k) = y_p \qquad (3)$$

The initial conditions for the iteration procedure require the following equation:

$$\forall_p \quad v_p(0) = x_p \qquad (4)$$

The converged state of Hopfield net means the net has already reached one of attractors [3]. An attractor is a point of local minimum of energy (Liapunov) function:

$$E(x) = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ij} x_i x_j + \sum_{i=1}^{N} \theta_i x_i \qquad (5)$$

B. Hebbian learning algorithm

The training patterns are presented one by one in a fixed time interval. During this interval, each input data is communicated to its neighbour ***N*** times:

$$w_{ij} = \begin{cases} \frac{1}{N}\sum_{m=1}^{M} x_i^{(m)} x_j^{(m)} & \text{for } i \neq j \\ 0 & \text{for } i = j \end{cases} \quad (6)$$

The realisation of Hebbian learning algorithm is very easy, but the algorithm secures rather low capacity of net:

$$\boldsymbol{M_{max} = 0{,}138\ N} \quad (7)$$

$\boldsymbol{M_{max}}$	- maximum number of training vectors,
$\boldsymbol{M}$	- number of training vectors
$\boldsymbol{w_{pj}}$	- weight related to feedback signal,
$\boldsymbol{v_i(k)}$	- feedback signal,
$\boldsymbol{\Theta_p}$	- bias

C. Delta-rule learning algorithm

The weights are calculated in recurrent way including all training patterns, according to the following matrix equation:

$$W = W + \frac{\eta}{N}\left[x^{(i)} - Wx^{(i)}\right]\left[x^{(i)}\right]^T \quad (8)$$

$\boldsymbol{\eta \in [0{,}7,\ 0{,}9]}$	- learning rate,
$\boldsymbol{N}$	- number of neurons,
$\boldsymbol{W}$	- matrix of weights,
$\boldsymbol{x}$	- input vector

The learning rate has the same influence on the training process as a learning rate appeared with the multilayer networks. The learning process stops when the next training step generates the changes of weights which are less then the established tolerance ε [1, 11].

III. DATA DEPENDENCE GRAPHS FOR HOPFIELD NEURAL NETWORK ALGORITHMS

A Data Dependence Graph is a directed graph that specifies the data dependencies of an algorithm. In a Data Dependence Graph nodes represent computations and arcs specify the data dependencies between computations. For regular and recursive algorithms, the Data Dependencies Graphs are also regular and can be represented by a grid model. Design of a locally linked Data Dependence Graph is a critical step in the design of systolic array [3, 9].

A. Data Dependence Graph for Hebbian learning algorithm

Each node in Data Dependence Graph for Hebbian training algorithm (Fig. 1, Fig. 2) multiplies two of corresponding input signals $\boldsymbol{x_i}$ and obtains this way (6) the weight $\boldsymbol{w_{ij}}$ which is stored in local memory unit. The input signals $\boldsymbol{x_i}$ are passed to the nearest bottom neighbours and the neighbours on the right hand.

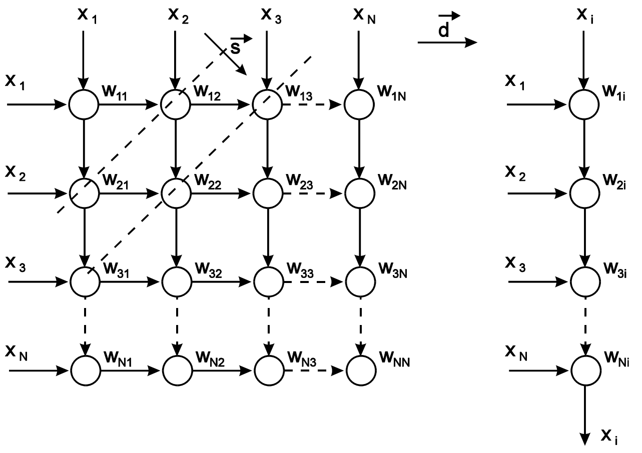


Fig. 1 - Data Dependence Graph and array structure for Hebbian learning algorithm

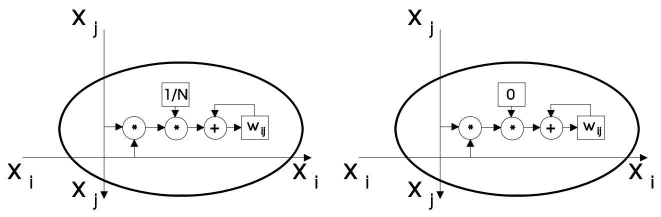


Fig. 2 - Single node of Data Dependence Graph for Hebbian learning algorithm: for ***i ≠ j*** – left side, for ***i = j*** – right side

B. Data Dependence Graph for Delta-rule learning algorithm

The Data Dependence Graph for Delta-rule training algorithm (Fig. 3, Fig. 5) we can divide into two parts: *Relation Graph* $\boldsymbol{G_r}$ and *Value Graph* $\boldsymbol{G_w}$. Each node which belongs to $\boldsymbol{G_r}$ multiplies a corresponding input signal $\boldsymbol{x_i}$ and weight value $\boldsymbol{w_{ij}}$, then it subtracts the multiplication result from the input signal $\boldsymbol{x_i}$. Each node in the $\boldsymbol{G_w}$ part of the Data Dependence Graph is responsible for three operations (Fig. 4). During the first operation the node multiplies the corresponding result obtained at the end of the calculations related to the $\boldsymbol{G_r}$ part of the Data Dependence Graph and the input signal $\boldsymbol{x_i}$. During the second operation each node multiplies the obtained values and the fraction: learning rate/number of neurons. At the end the values of weights are upgraded. This way (8) the weights are obtained and next they are stored in local memory unit. The product of multiplication is passed to the nearest neighbour on the right hand. The input signals $\boldsymbol{x_i}$ are passed to the nearest bottom neighbours [4, 6].

C. Data Dependence Graph for retrieving phase

Each node in Data Dependence Graph for retrieving algorithm (Fig. 6, Fig. 7) multiplies the input signal $\boldsymbol{x_i}$ or feedback signals v_i and corresponding weight $\boldsymbol{w_{ij}}$ which is stored in local memory unit. The product of multiplication is passed to the nearest neighbour on the right hand. The

$\boldsymbol{\varphi_i}$ nodes collect the partial products and calculate the global value of coproduct (1). The last nodes on the right are the comparators to check if the next iteration is necessary (2), (3). The input signals $\boldsymbol{x_i}$ or feedback signals $\boldsymbol{v_i}$ are passed to the nearest bottom neighbours [5, 7].

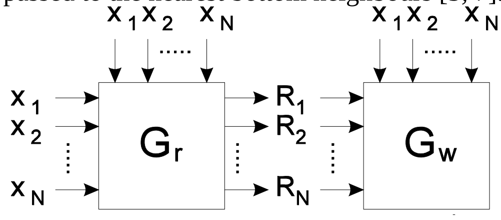


Fig. 3 - Data Dependence Graph for Delta-rule learning algorithm composed of *Relation Graph* $\boldsymbol{G_r}$ and *Value Graph* $\boldsymbol{G_w}$

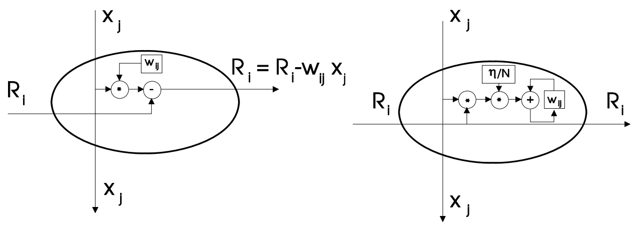


Fig. 4 - Single node of Data Dependence Graph for Delta-rule learning algorithm: node of *Relation Graph* $\boldsymbol{G_r}$ – left side, node of *Value Graph* $\boldsymbol{G_w}$ – right side

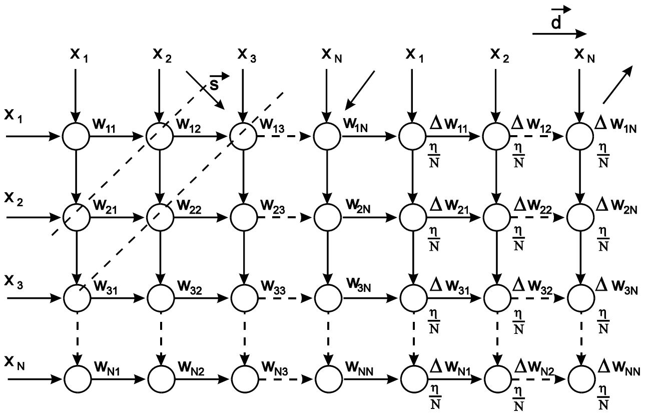

$\vec{d}$
$\vec{s}$
x_1 x_2 x_3 x_N x_1 x_2 x_N
w_{11} w_{12} w_{13} w_{1N} Δw_{11} Δw_{12} Δw_{1N}
w_{21} w_{22} w_{23} w_{2N} Δw_{21} Δw_{22} Δw_{2N}
w_{31} w_{32} w_{33} w_{3N} Δw_{31} Δw_{32} Δw_{3N}
w_{N1} w_{N2} w_{N3} w_{NN} Δw_{N1} Δw_{N2} Δw_{NN}
$\frac{\eta}{N}$

Fig. 5 - Data Dependence Graph for Delta-rule learning algorithm

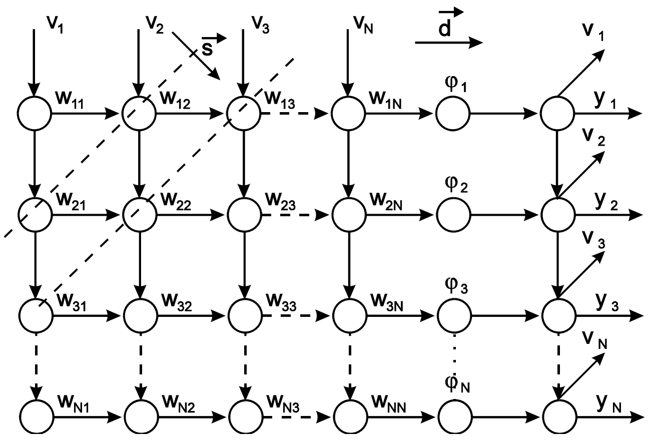
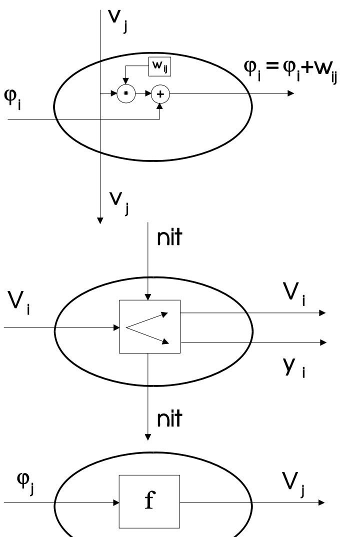

v_1 v_2 v_3 v_N
$\vec{d}$
$\vec{s}$
w_{11} w_{12} w_{13} w_{1N} φ_1 y_1 v_1
w_{21} w_{22} w_{23} w_{2N} φ_2 y_2 v_2
w_{31} w_{32} w_{33} w_{3N} φ_3 y_3 v_3
w_{N1} w_{N2} w_{N3} w_{NN} φ_N y_N v_N

Fig. 6 - Data Dependence Graph for retrieving phase



```
if ($v_i$ <> $v^{old}_i$)
  then  nit = 1
  else  nit = 0
if (nit == 0)
  then  $y_i$ = $v_i$

f: if ($\varphi_i$ > 0)
    then $v_i$  = 1
   else if ($\varphi_i$ == 0)
    then $v_i$  = $v^{old}_i$
   else $v_i$  = 0
```

Fig. 7 - Single node of Data Dependence Graph for retrieving phase and description of function realized step by step during the algorithm

IV. Mapping Data Dependence Graphs onto Array Structure

A. Processor assignment via linear projection

Mathematically, a linear projection is often represented by a *projection vector* $\vec{d}$. Because the Data Dependence Graph of a locally recursive algorithm is very regular, the linear projection maps an *n*-dimensional Data Dependence Graph onto an *(n-1)* dimensional lattice of points, known as processor space [3, 11]. It is common to use a linear projection for processor assignment, in which nodes of Data Dependence Graph along a straight line are projected to an Elementary Processor in the processor array (Fig. 1, Fig. 5, Fig. 6).

B. Schedule assignment via linear scheduling

A scheduling scheme specifies the sequence of the operations in all Elementary Processors. More precisely, a schedule function represents a mapping from the *n*-dimensional index space of the Data Dependence Graph onto a 1-D schedule (time) space. Linear scheduling is very common for schedule assignment (Fig. 1, Fig. 5, Fig. 6). A linear schedule is based on a set of parallel and uniformly spaced hyperplanes in the Data Dependence Graph [3, 9]. These hyperplanes are called *equitemporal hyperplanes* - all the nodes on the same hyperplane are scheduled to be proceed at the same time. A linear schedule can also be represented by a *schedule vector* $\vec{s}$, which points in the direction normal to the hyperplanes. For any computation node indexed by a vector *n* in the Data Dependence Graph, its scheduled processing time is $\vec{s}n$.

C. Mapping policies

Given a Data Dependence Graph and the projection direction $\vec{d}$ not all schedule vectors $\vec{s}$ are valid for the Data Dependence Graph. Some may violate the precedence relations specified by the dependence arcs. For systolic design, the schedule vector $\vec{s}$ in the projection procedure must satisfy the following two conditions [11]:

- causality condition: $\vec{s}^{\,T}\vec{e} > 0$ (9)

$\vec{e}$ - represents any of the dependence arcs in the Data Dependence Graph

- positive pipeline period: $\vec{s}^{\,T}\vec{d} \neq 0$ (10)

This way the rectangular Data Dependence Graphs are converted into linear pipelined systolic arrays. The number of elementary processors which are used for array construction equals the number of neurons in simulated Hopfield neural network. Each elementary processor combines all functions described by nodes of Data Dependence Graph placed at the same horizontal line. If these functions are the same – like in Hebbian learning algorithm there is no need to switch the function realised by single processor (Fig. 8). If the function changes – we can observe such situation within Delta-rule algorithm and within retrieving phase the function executed by elementary processor is switched in proper moments. Switching procedure is driving by clock shape signal, which is responsible for all operations realised in processors and local communication among the neighbouring processors (Fig. 9, Fig. 10) All Data Dependence Graphs have exactly the same number of horizontal lines – so number of elementary processors of systolic structures is also the same for all algorithms related to the Hopfield neural network. It means that is rather easy to create the universal structure which is able to implement all algorithms [4, 7]. This statement is true, because the values of weights calculated and stored in local memories – shifting registers - of the same processors which ought to use them during retrieving phase. A problem of necessary precision we ought to guarantee for weights storing ought to be discussed. If we want to reduce the number of elementary processors we can change the classical linear structure into ring structure – where each elementary processor is responsible for modelling of greater number of neurons. Of course the reduction of number of elementary processors ought to be done in the way which preserve the same number of neurons for single processor. On the hand the capacity of local memory which collaborates with single processor ought to be able to store weights of set of neurons – these neurons which are implemented by single calculation unit. The limited number of processors is also the reason of reduced efficiency parameters specified for systolic array realisation [5, 7].

V. EFFICIENCY OF SYSTOLIC IMPLEMENTATION OF HOPFIELD NEURAL NETWORK

A. Computation time

This is time interval between starting the first computation and finishing the last computation of problem. Given a coprime schedule vector $\vec{s}$, the computation time of a systolic array can be computed as [6]:

$$T = \max_{\vec{p},\vec{q}\in L}\left\{\vec{s}^{\,T}\left(\vec{p}-\vec{q}\right)\right\}+1 \qquad (11)$$

L - is the index set of the nodes in the Data Dependence Graph

In the presented architectures for all algorithms the schedule vector is defined as: $\vec{s} = [1\,,1]$. The indexes of nodes are spread on ***N*** elements within vertical and horizontal axes of space for Hebbian training implementation - so taking number of basic operations into account we can calculate the computation time as:

$$T_{systol} = 3N\left(\frac{N-1}{k}+1\right)\tau M \qquad (12)$$

$\boldsymbol{\tau}$	- processing time for elementary processor,
M	- number of training patterns,
k	- number of elementary processors

For Delta-rule training algorithm we have ***N*** elements within vertical axis and ***2N*** within horizontal axis of space. In fact the Data Dependence Graph for this algorithm is combined by two independent structures of operations. We can notice this observation at (Fig. 3.) - the set of input values is presented two times [5, 6]. The computation time for both parts of algorithm isn’t the same because the number of basic operations is different, number of nodes and the topology of them is the same. Computation time for each part we can estimate as:

$$T_{systol} = 5N\left(\frac{N-1}{k}+1\right)\tau M \qquad (13)$$

In addition the Delta-rule learning is gradient-based algorithm, so it is necessary to present each training pattern many times to obtain the correct value of weights. That is the reason why the estimated computation time ought to be modified by number of iterations related to single training pattern. Based on these remarks we can calculate the computation time for learning algorithms:

$$T_{systol} = 5N\left(\frac{N-1}{k}+1\right)\tau M\beta \qquad (14)$$

$\boldsymbol{\beta}$ - number of iterations for single training pattern

The indexes of nodes are spread on ***N*** elements within vertical axis and ***N+2*** elements for horizontal axis of space for retrieving algorithm. Two extra columns of nodes are responsible for accumulation of the products of elementary multiplication realised by previous nodes and for decision if the next step of calculation is necessary.

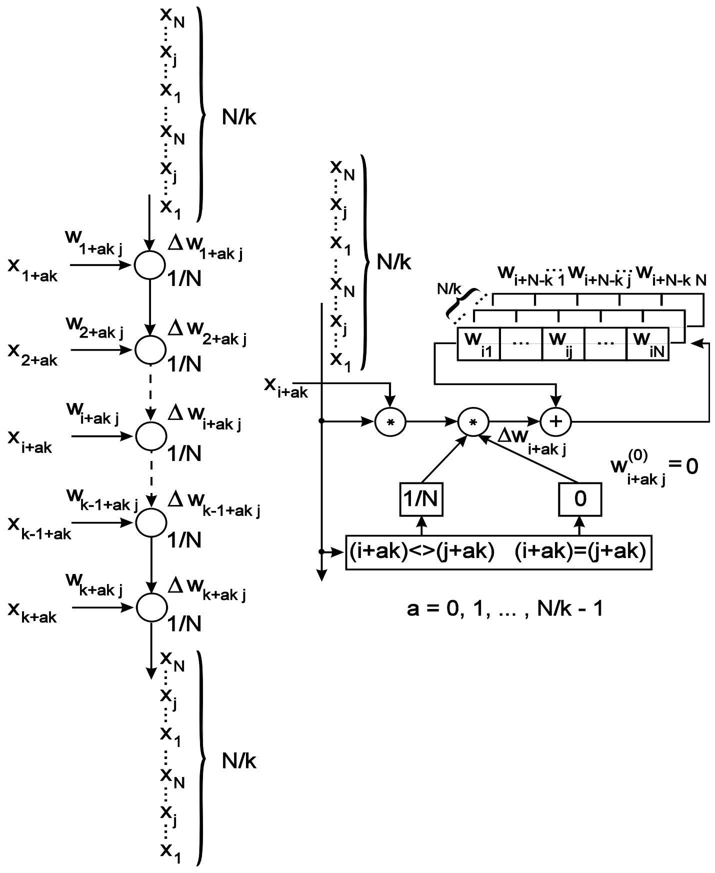


Fig. 8 - Systolic array constructed based on Data Dependence Graph which realises Hebbian learning algorithm

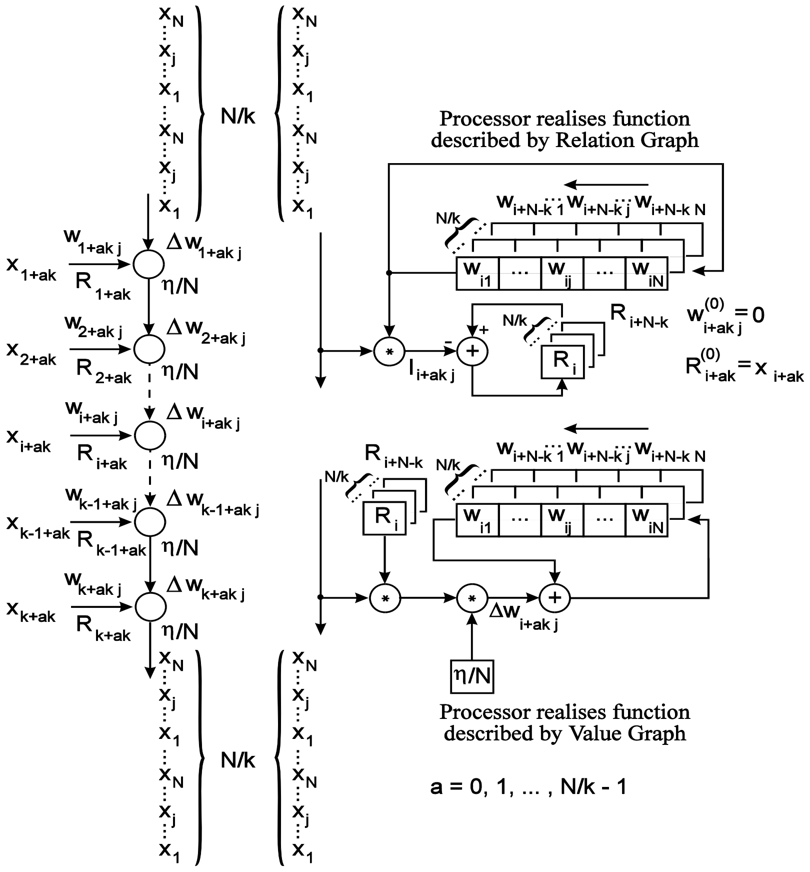

N/k
Processor realises function described by Relation Graph
$w_{1+ak\,j}$
$\Delta w_{1+ak\,j}$
x_{1+ak}
R_{1+ak}
η/N
$w_{i+N-k\,1}$ $w_{i+N-k\,j}$ $w_{i+N-k\,N}$
w_{i1} ... w_{ij} ... w_{iN}
R_{i+N-k}
R_i
$I_{i+ak\,j}$
$w^{(0)}_{i+ak\,j} = 0$
$R^{(0)}_{i+ak} = x_{i+ak}$
$\Delta w_{i+ak\,j}$
Processor realises function described by Value Graph
a = 0, 1, ... , N/k - 1

Fig. 9 - Systolic array constructed based on Data Dependence Graph which realises Delta-rule learning algorithm

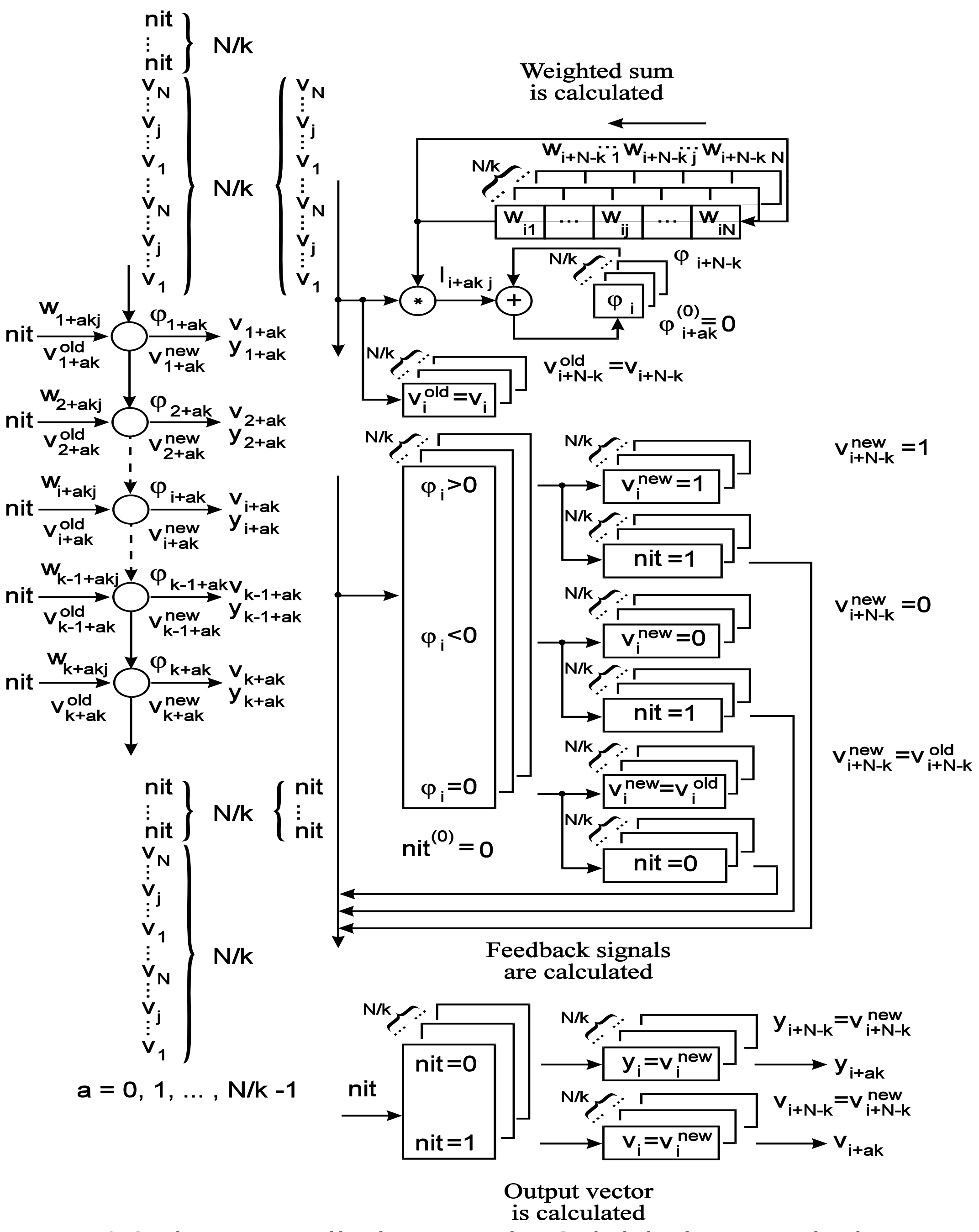


Fig. 10 - Systolic array constructed based on Data Dependence Graph which realises retrieving algorithm

The computation time - modified by number of iterations related to single training pattern - we can describe using the following equation [6]:

$$T_{systol} = 2N\left(\frac{N+1}{k}+1\right)\tau\beta \tag{15}$$

B. Pipelining period

This is the time interval between two successive computations in a processor. As previously discussed, if both $\vec{d}$ and $\vec{s}$ are irreducible, then the pipelining period equals:

$$\alpha = \vec{s}^{\,T}\vec{d} \tag{16}$$

The pipelining period is the same for all Hopfield neural network algorithms and we can calculate it as: $\boldsymbol{\alpha = 1}$. It means the time interval between two successive computations in an elementary processor is as short as possible [4, 8].

C. Block period

This is time interval between the initiation of two successive blocks of operations [3]. If we assume the necessary calculations for single training pattern as a single successive block, the block period equals computation time for single pattern and pipelining period. This remark is true for learning algorithms as well as for retrieving procedure. Based on these observations and values previously calculated the block period equals [5]:

- for Hebbian learning algorithm:

$$T_{block} = 3N\left(\frac{N-1}{k}+1\right)\tau \tag{17}$$

- for Delta-rule learning algorithm:

$$T_{block} = 5N\left(\frac{N-1}{k}+1\right)\tau\beta \tag{18}$$

- for retrieving algorithm:

$$T_{block} = 2N\left(\frac{N+1}{k}+1\right)\tau \tag{19}$$

D. Speed-up an Utilization rate

Lets define the speed-up factor as the ratio between the sequential computation time $\boldsymbol{T_{seq}}$ and the array computation time $\boldsymbol{T_{systol}}$ and the utilization rate as the ratio between the speed-up factor and the number of processors [3].

$$speed-up = \frac{T_{seq}}{T_{systol}} \tag{20}$$

$$utilization\ rate = \frac{speed-up}{k} \tag{21}$$

Sequential computation time for Hopfield neural network algorithms – taking number of neurons, number of weights and number of basic operations into account – equals [6]:

- for Hebbian learning:

$$T_{seq} = 3N^2\tau M \tag{22}$$

- for Delta-rule learning:

$$T_{seq} = 5N^2\tau M\beta \tag{23}$$

- for retrieving algorithm:

$$T_{seq} = 2N(N+2)\tau\beta \tag{24}$$

Based on values of array computation time calculated before we can evaluate the speed-up and utilization rate parameters for all algorithms related to the Hopfield network:

- for Hebbian learning:

$$speed-up = \frac{Nk}{N-1+k} \tag{25}$$

$$utilization\ rate = \frac{N}{N-1+k} \tag{26}$$

- for Delta-rule learning:

$$speed-up = \frac{Nk}{N-1+k} \tag{27}$$

$$utilization\ rate = \frac{N}{N-1+k} \tag{28}$$

- for retrieving algorithm:

$$speed-up = \frac{(N+2)k}{N+1+k} \tag{29}$$

$$utilization\ rate = \frac{N+2}{N+1+k} \tag{30}$$

k	- number of elementary processors,
N	- number of neurons

VI. CONCLUSIONS

Summarizing, the paper proposed a new methodology for Hopfield neural network simulation based on systolic array structure. The methodology is related to training methods: Hebbian learning and Delta-rule learning as well as to retrieving phase. The discussion is focused on operations which are realized during the following steps of each algorithm and the data which are transferred among the calculation units. It is clear which operations can be done in parallel way and when the sequence is necessary. The results of discussion show that it is possible to create the universal structure to implement all algorithms related to Hopfield neural network. This way there are no barriers to tune the Hopfield net to completely new tasks. The proposed methodology can be used as a basis for VLSI structures which implement Hopfield net or as a basis for set of general purpose processors – as transputers or DSP processors - which can be used for Hopfield neural network implementation.

If we resumed the efficiency parameters as a function of a number of elementary processors we must say the comparison of the same criteria for two methods of learning is the most interesting part, because the conclusion related to this discussion can be used as the main aspect for decision which learning procedure choose.

Computation Time - if we assume single presentation of each training vector - is less then two times longer for Delta-Rule learning. Of course such assumption is true for Hebbian learning but isn't in general true for Delta-Rule. Each next presentation of training set makes the Computation Time longer and the dependence is directly proportional.

Pipelining period is - as we discussed before - as short as possible for all presented algorithms, so this implementation criterion can't be useful for any decision related to the systolic array prepared for Hopfield network.

The changes of Block Period look very simple to the changes of Computation Time. It isn't strange because Computation Time is a product of a set of Block Periods. The general difference is that Block Period doesn't depend neither on the number of training vectors or on the number of iterations related to the single training vector. Such observation is true because the partial calculation - responsible for Block Period time is taken as a part of single training vector processing.

Table 1. Efficiency parameters for ring systolic structure related to Hopfield neural network algorithms when number of elementary processors equals number of neurons

	Learning		Retrieving phase
	Hebbian rule	Delta-rule	
Computation time T_{systol} (min)	$3(2N-1)\tau\ M$	$5(2N-1)\tau\ M\beta$	$2(2N+1)\tau\ \beta$
Block period T_{block} (min)	$3(2N-1)\tau$	$5(2N-1)\tau\ \beta$	$2(2N+1)\tau$
Speed-up (max)	$\frac{N^2}{2N-1}$	$\frac{N^2}{2N-1}$	$\frac{(N+2)N}{2N+1}$
Utilization rate (min)	$\frac{N}{2N-1}$	$\frac{N}{2N-1}$	$\frac{N+2}{2N+1}$

Table 2. Efficiency parameters for ring systolic structure related to Hopfield neural network algorithms when number of elementary processors equals one

	Learning		Retrieving phase
	Hebbian rule	Delta-rule	
Computation time T_{systol} (max)	$3N^2\tau M$	$5N^2\tau M\beta$	$2N(N+2)\tau\beta$
Block period T_{block} (max)	$3N^2\tau$	$5N^2\tau\beta$	$2N(N+2)\tau$
Speed-up (min)	**1**	**1**	**1**
Utilization rate (min)	**1**	**1**	**1**

τ - processing time for elementary processor,
β - number of iterations for single training pattern,
M - number of training patterns,
N - number of neurons,

So Block Period is much responsible criterion than the Computation Time discussed before. It is very interesting we can observe exactly the same Speed-Up and Processor Utilization Rate both for Hebbian and Delta-Rule learning procedures. The necessary time-period for calculation of Delta-Rule procedure is longer than time-period related to Hebbian learning - but elementary processors' using is the same. This observation seems to be very important for systolic array project.

At the end proposed methodology can be also useful for parallel programme realisation of Hopfield neural network and can be easily adopted for other recurrent neural nets.

This work has been supported by Wroclaw University of Technology grant No. 342926.

References

[1] K. V. Asari, C. Eswaran, "Systolic array implementation of artificial neural networks", Indian Institute of Technology, Madras 1992

[2] A. Ferrari, Y. H. Ng, "A parallel architecture for neural networks", Parallel Computing'91, Elsevier Science Publishers B. V. 1992, pp.: 283 – 290

[3] S. Y. Kung, "Digital neural networks", PTR Prentice Hall 1993

[4] J. Mazurkiewicz, "A processor pipeline architecture for Hopfield neural network", II SCANN'98 Slovak Conference on Artificial Neural Networks, Smolenice, Trnava, Slovakia 1998, pp.: 158 – 163

[5] J. Mazurkiewicz, "Efficiency of systolic array - constructed with limited number of elementary processors - for Hopfield neural network implementation", VI International MENDEL 2000 Conference on Soft Computing, Brno, Czech Republic 2000, pp.: 325 – 330

[6] J. Mazurkiewicz, "Efficiency of systolic implementation of Hopfield neural networks", V Conference Neural Networks and Soft Computing, Zakopane, Poland 2000, pp.: 748 – 753

[7] J. Mazurkiewicz, "SIMD-type Simulator for Recurrent Neural Nets", 35th Spring International MOSIS'01 Conference Modelling and Simulation of Systems, Ostrava, Czech Republic 2001, vol. 1, pp.: 173 - 180

[8] J. Mazurkiewicz, "Systolic-based Simulator for Hopfield Neural Net", XXIIIrd International Autumn Colloquium ASIS 2001 Advanced Simulation of Systems, Ostrava, Czech Republic 2001, pp.: 223 - 228

[9] N. Petkov, "Systolic parallel processing", North-Holland 1993

[10] S. G. Shiva, "Pipelined and parallel computer architectures", Harper Collins Publishers 1996

[11] D. Zhang, "Parallel VLSI neural system design", Springer-Verlag 1999