UDC: 004.8
Original scientific paper

SELECTING NEURAL NETWORK ARCHITECTURE FOR INVESTMENT PROFITABILITY PREDICTIONS

Tonimir Kišasondi, Alen Lovrenčić
University of Zagreb, Faculty of Organization and Informatics, Varaždin, Croatia
{tonimir.kisasondi, alen.lovrencic}@foi.hr

Abstract: *In this paper we present a modified neural network architecture and an algorithm that enables neural networks to learn vectors in accordance to user designed sequences or graph structures. This enables us to use the modified network algorithm to identify, generate or complete specified patterns that are learned in the training phase. The algorithm is based on the idea that neural networks in the human neurocortex represent a distributed memory of sequences that are stored in invariant hierarchical form with associative access. The algorithm was tested on our custom built simulator that supports the usage of our ADT neural network with standard backpropagation and our custom built training algorithms, and it proved to be useful and successful in modelling graphs.*

Keywords: *Neural networks, MLP training algorithms, neural network ADT, neural network graph modelling.*

1. INTRODUCTION

In this paper we present a modified neural network architecture and an algorithm that enables neural networks to learn vectors in accordance to user designed sequences. This enables us to use the modified network algorithm to identify, generate or complete specified patterns that are learned in the training phase. The algorithm is based on the idea that neural networks in the human neurocortex represent a distributed memory of sequences that are stored in invariant hierarchical form with associative access. (See [4]) The algorithm was tested on our custom built simulator that supports the usage of our ADT neural network with standard backpropagation and our custom built training algorithms.

2. PREVIOUS RESEARCH IN THIS FIELD

Similar works that can employ some sort of temporal processing are Elman and Jordan networks (see [5] page 528) which employ feedback loops from the hidden to input layer and Time delayed neural network (TDNN) (see [5] page 479). Both present a method that employs modifications to the original algorithm's inner functioning and cannot represent dispersed time patterns, while this modification can be utilized on the original algorithm without changing of the algorithms internal functioning. Also our modifications to the original algorithm can be extended to a number of architectures and training algorithms, because they represent a concept of transfering information in the neural network. Also our algorithm can model graphs with vertices and edges so it has a broad use.

3. MODIFICATIONS TO THE FEEDFORWARD NEURAL NETWORK ARCHITECTURE

Each neural network has the ability to transform an input vector to an output vector by means of the weights and biases that the network contains. Those weights and biases represent the networks information processing memory. We define that transformation ability that can organize output vectors as a set that contains certain vectors (edges) and nodes. The output vector organization can be represented as a mathematical graph. Nodes of the vector only represent a certain state that a neural net can be in, in which only the vectors from that state are available. A vector is defined as a set of data that represents the input or output of the neural net. Input vectors are marked as: $[x_1, x_2, x_3, ..., x_n]$ and output vectors are marked as: $[y_1, y_2, y_3, ..., y_n]$.

A node is defined as a point in the graph in which all vectors that are assigned to that node have a unique representative part that can be used in the feedback loop of the neural net. Some vectors point to the same node in the graph, while some vectors can be modelled so that the can shift the current node the neural net uses. Nodes are marked as “Nn:” where the lowercase “n” represents the numerical designation of the node.

Also, we define a specific architecture and a principle of neural net training which is based on modifications to the feedforward algorithm that enables the usage of feedback loops where the user uses only the relevant part of the neurons that represent input neurons which can also be feedback neurons or stand-alone input neurons like neuron #3 as represented on figure 1. The feedback loop connects output layer neurons with their input counterparts with or without usage of special trigger neurons (designated T) that are used for additional modelling or preparation of the combined input vector. By that, we add information into the input and output vectors of our neural net in the training phase, so we can organize vectors into sequential, parallel or combined patterns. In accordance to that, the network has a temporal effect: current output of the network is affected by previous vectors that were computed.

Every vector has the ability to transpose the network into another node in which for the same inputs the net gives out different outputs that are defined by our network training because the part of the information about the output is transmitted to the input in the next feedforward phase. Our algorithm was tested on a three layered neural network with backpropagation training with usage of linear momentum and jitter. (See [2],[7] and [8])

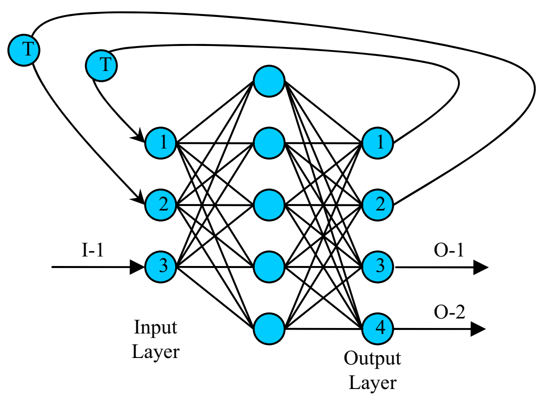

Figure 1. Modified neural network architecture

To employ the organization of neural net memory into sequences, each input and corresponding output vector used in training or running the network must have a specific number of neurons devoted for usage in the feedback loop. Those neurons can be stand-alone neurons that are only used for the feedback loop like the neurons 1 and 2 on figure 1, or if we have a set of unique inputs with unique outputs we can use the input neurons normally and also use them in the feedback loop. As an example of organization of input vectors from figure 1, we can derive expression 1, where we see that if we want a transition from node 1 to node 2 that y_1, y_2 that are the outputs of the neural net in feedforward phase "t" must be transferred to x_1, x_2 which are the input neurons in the beginning of the next feedforward phase "t+1"

$$N_1 : [y_1, y_2]_t \mapsto N_2 : [x_1, x_2]_{t+1} \tag{1}$$

Following that, we can define any number of neurons that will be used for the feedback loop, where each output vector defines which input vectors can be selected in the next feedforward phase. If we use a greater number of neurons for the feedback loop, we can create more nodes. We can represent the memory organization of vectors and nodes with a memory organization graph. For example we can create the graph for network from figure 1, with 2 nodes and 5 vectors.

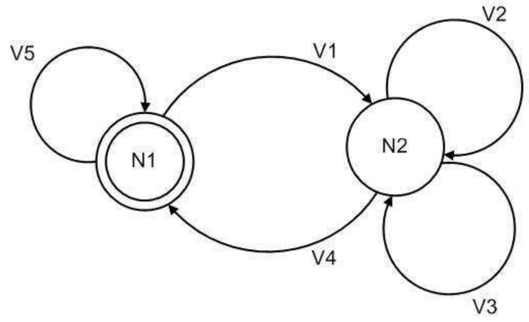

Figure 2. Example memory organization graph

From figure 2 we can see that the user generated graph of memory organization has 2 nodes, N1 and N2. Node 1 has a vector V5 that is only used in that node and a transition vector V1. Also, N1 is the starting node and is marked with double outlined circle. Node 2 has two vectors that are only accessible from that node, V2 and V3 and has a transition vector V4. As previously noted, we can generate as many nodes or vectors as we want as long as there are enough weights and biases to represent them since the information about transformation of the input is contained in the weights and biases of the network. To evaluate the memory capacity of any multilayered MLP network, we created expression 2. This gives us the exact number of weights and biases in the whole network.

$$\sum(w \wedge b) = \sum_{i=1}^{\lambda-1}(NPL_i \cdot NPL_{i+1}) + \sum_{j=2}^{\lambda}(NPL_j) \qquad (2)$$

In expression 2 the Greek sign lambda represents the number of layers of the network and NPL_i stands for “neurons per layer” the number of neurons in a network layer. The first part of the expression before the summation sign is connected to the number of weights while the second part of the expression is connected to the number of biases. Also when modelling the feedback loop data, we need to note that the neurons of the feedback loop will most easily convert the input values that are the asymptotical values of their activation functions. Additional improvements to the feedback loop can be made by usage of a layer of trigger neurons on the feedback loop that utilise certain rules based on value of the input that they receive. Most effective solutions are based on binary or bipolar triggers that convert the input to the desired output based on a certain tolerance ratio like 5% or 10% deviation from the desired value, using these neurons we can avoid certain situations that put our network in an unidentified node or an unstable state. It is advised to use such neurons if we use a larger number of nodes. Also, any discrete value with enough distance from another identifier can be used for the feedback loop. Activation function of such neuron can be easily represented by the source code

```
double binary_trigger(double input)
{
if(input>0.5)
return 1;
else
return 0;
}
```

4. CREATION OF GRAPHED VECTORS FOR BACKPROPAGATION LEARNING

Each vector that wants to be used in this architecture must be composed of two parts, the data part and the feedback part. Each output vector must have a unique identifier in its feedback part about the node that follows from that vector. As an example, we can take the memory organization graph from figure 2. For the neural network from figure 1, Neurons 1 and 2 are used in the feedback loop and create the feedback part, while neuron 3 creates the data part of the vector. Output vectors have 4 neurons. 1 and 2 which create the feedback part and 3 and 4 which create the data part.

Node 1 will use the designation (0,0) so the input vector for node 1 looks like: $(0,0,\Gamma)$

Node 2 will use the designation (0,1) and it's input vector is: $(0,1,\Gamma)$, where Γ represents any input value.

And from that, given for some input or output data, we can create the set of training vectors that look like:

$$\begin{array}{l}
Input\ vector \xrightarrow{Edge} Output\ \ vector \\
N1:(0,\ \ 0,\ \ 0.3)\xrightarrow{V5}(0,\ \ 0,\ \ 0.4,\ \ 0.8) \\
N1:(0,\ \ 0,\ \ 0.6)\xrightarrow{V1}(0,\ \ 1,\ \ 0.1,\ \ 0.5) \\
\\
N2:(0,\ \ 1,\ \ 0.3)\xrightarrow{V2}(0,\ \ 1,\ \ 0.7,\ \ 0.3) \\
N2:(0,\ \ 1,\ \ 0.9)\xrightarrow{V3}(0,\ \ 1,\ \ 0.2,\ \ 0.6) \\
N2:(0,\ \ 1,\ \ 0.6)\xrightarrow{V4}(0,\ \ 0,\ \ 0.9,\ \ 0.9)
\end{array}$$

If we want to use another approach, we can use graph structures that today present a method for visualization and understanding a wide variety of problems. The most common and standard way to mathematically describe graphs is with use of adjacency matrixes. The problem is to transfer the graph matrix into a set of input output vector combinations. For a proof of concept the following example for a graph with 4 nodes and 10 vectors is given:

Table 1. Graph adjacency matrix

	N1	N2	N3	N4
N1	V_1	V_{10}	V_5	
N2		V_2,V_3		V_{11}
N3	V_7			
N4	V_8		V_9	V_4

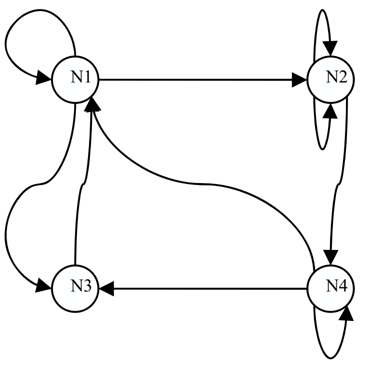

Figure 3. Graph for matrix in table 1

From Table 1 we created the graph in figure 3. Conversion to the input and output pairs will be given with some assumptions: Each of the node designators (Nn) where the "n" numerical designation of the node, will be treated as generated binary set, as we said in chapter 3 of this paper. In this case, for example this would be (N1: 00, N2: 01, N3: 10, N4:11). Each of the vectors is shown as "Vn" where each vector would state an input / output combination pair for the neural network. The conversion is simple, because the adjacency matrix in reality shows that each row specifies the start node, and each column designates and end node connection. An example conversion is done in table 2.

Table 2. Input / Output vector combinations created from Table 1.

#	Input vector	Output Vector
1	N1,V_1	N1,V_1
2	N1,V_{10}	N2,V_{10}
3	N1,V_5	N3,V_5
4	N2,V_2	N2,V_2
5	N2,V_3	N2,V_3
6	N2,V_{11}	N4,V_{11}
7	N3,V_7	N1,V_7
8	N4,V_8	N1,V_8
9	N4,V_9	N3,V_9
10	N4,V_4	N4,V_4

From this table we can see that conversions from matrix to vector set form are simple for manual or automatic conversion and that they can be easily automated with a simple program.

If we want to use the algorithm on our created vector set, we need to transfer the output feedback loop data to the input feedback loop data, transfer the input data part and then initiate the feedforward sequence. This can be described with the source code:

```
void Feedback_Forward(){

input_vector[0] = network_layer[1].neuron[0].output;
input_vector[1] = network_layer[1].neuron[1].output;
// the feedback loops are defined here,here, we see the example for 2 loops, while we can add any number of
loops.

for(a = 0; a < architecture[1]; a++)
network_layer[0].neuron[a].output =0; // Clean up the outputs of hidden layer neurons
for(a = 0; a < architecture[2]; a++)
network_layer[1].neuron[a].output =0;


for(a = 0; a < architecture[1]; a++)
{
network_layer[0].neuron[a].output += network_layer[0].neuron[a].bias;  // Hidden layer FeedForward
for(b = 0; b < architecture[0]; b++)
{
network_layer[0].neuron[a].output += network_layer[0].neuron[a].weight[b] * input_vector[b];
}
network_layer[0].neuron[a].output = sig(network_layer[0].neuron[a].output, prim_slope_type);
}
for(a = 0; a< architecture[2]; a++)
{
for(b = 0; b< architecture[1]; b++)
{
network_layer[1].neuron[a].output += network_layer[1].neuron[a].weight[b] * network_layer[0].neuron[b].output;
}
network_layer[1].neuron[a].output += network_layer[1].neuron[a].bias;
network_layer[1].neuron[a].output  =  sig(network_layer[1].neuron[a].output  ,  sec_slope_type);  //  Output  layer
feedforward
}
} // End of function Feedback_Forward
```

As we can see from the code, the original algorithm is intact, all changes and modifications can be created as an additional phase that is executed prior the feedforward phase. Also, we call the function *sig(x,y)* in our code, which returns the value of the sigmoid function of that neuron in a specific layer.

After training, our network is in the last node that the training sequence initiated with its feedback_forward function. We must set the initial node of the network with the function starting_node that will put the starting node in the input vectors feedback part. We can modify the function to accommodate a larger set by simply adding more elements for the feedback loop and more hidden neurons. Here the starting nodes designation in the feedback part of the vector is (0,0)

```
void starting_node()
{
network_layer[1].neuron[0].output = 0;  // Set the feedback part of the vector to (0,0,.....
network_layer[1].neuron[1].output = 0;
}
```

5. VERIFICATION OF ALGORITHM AND ARCHITECTURE / SIMULATIONS AND RESULTS

We verified our algorithm on several sets in our NN-SIM2 simulator for validation and as a proof of concept. As an example we will take a memory organization graph composed of 10 nodes with 30 vectors. The graph's organization and dataset is shown on figure 4.

Node	FEEDBACK DATA							FEEDBACK DATA					
	1	1	1	1	0.3	0.9	→	0	0	0	1	0.1	0.6
1	1	1	1	1	0.7	0.1	→	1	0	0	0	0.2	0.7
	1	1	1	1	0.4	0.5	→	1	0	0	1	0.6	0.2
	0	0	0	1	0.3	0.9	→	0	0	1	0	0.3	0.6
2	0	0	0	1	0.7	0.1	→	1	1	1	1	0	0
	0	0	0	1	0.4	0.5	→	0	0	0	1	0.2	0.1
	0	0	1	0	0.3	0.9	→	0	0	1	1	0	0
3	0	0	1	0	0.7	0.1	→	0	0	0	1	0.2	0.7
	0	0	1	0	0.4	0.5	→	0	1	1	0	0.2	0.1
	0	0	1	1	0.3	0.9	→	0	1	0	0	0.2	0.7
4	0	0	1	1	0.7	0.1	→	0	0	1	0	0.2	0.5
	0	0	1	1	0.4	0.5	→	0	0	1	1	0.1	0.1
	0	1	0	0	0.3	0.9	→	0	1	0	1	0.7	0.3
5	0	1	0	0	0.7	0.1	→	0	0	1	1	0.9	0.6
	0	1	0	0	0.4	0.5	→	0	1	0	0	0.8	0.2
	0	1	0	1	0.3	0.9	→	0	1	1	0	0	0
6	0	1	0	1	0.7	0.1	→	1	0	0	1	0.2	0.6
	0	1	0	1	0.4	0.5	→	1	0	0	0	0.3	0.5
	0	1	1	0	0.3	0.9	→	0	1	1	1	0.5	0.3
7	0	1	1	0	0.7	0.1	→	1	0	0	1	0.3	0.6
	0	1	1	0	0.4	0.5	→	0	1	1	0	0	0
	0	1	1	1	0.3	0.9	→	1	0	0	0	0.2	0.6
8	0	1	1	1	0.7	0.1	→	0	1	1	0	0.2	0.7
	0	1	1	1	0.4	0.5	→	1	0	0	1	0.4	0.7
	1	0	0	0	0.3	0.9	→	1	0	0	1	0.3	0.5
9	1	0	0	0	0.7	0.1	→	0	1	1	1	0.3	0.6
	1	0	0	0	0.4	0.5	→	1	0	0	1	0.1	0.5
	1	0	0	1	0.3	0.9	→	1	1	1	1	0.3	0.4
10	1	0	0	1	0.7	0.1	→	0	1	0	0	0.6	0.8
	1	0	0	1	0.4	0.5	→	0	0	0	1	0.6	0.9

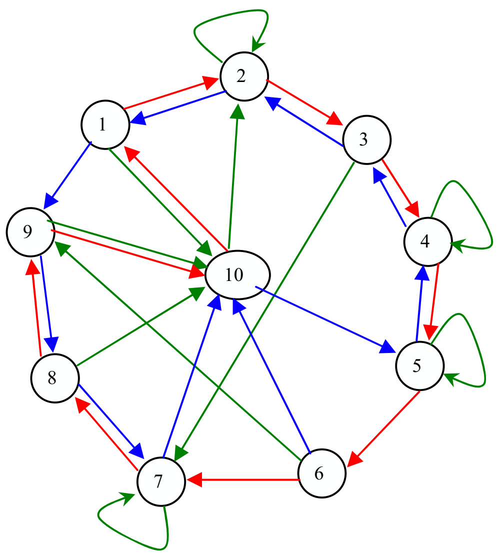

Figure 4. Dataset and graph for proof of concept

The network's training architectures are shown on table in figure 3, also each training architecture was tested with a variety of learning rates, momentum's and jitter factors. Validation neurons were also used, and their usefulness was tested.

Table 3. Architecture test data.

Input Feedback	Input Data	Hidden neurons	Output Feedback	Output Data
4 Neurons	2 Neurons	10	4 Neurons	2 Neurons
4 Neurons	2 Neurons	13	4 Neurons	2 Neurons
4 Neurons	2 Neurons	14	4 Neurons	2 Neurons
4 Neurons	2 Neurons	20	4 Neurons	2 Neurons
4 Neurons	2 Neurons	30	4 Neurons	2 Neurons
4 Neurons	2 Neurons	60	4 Neurons	2 Neurons

5.1 SIMULATION RESULTS

Each simulation was executed in iterations of 1000 to 50000 iterations with MSE measurements after each 1000th iteration. The architecture and algorithm proved to complete and gave correct outputs on all sufficient sets. We will show the MSE measurements for only 2 sets that are the most interesting. The minimal architecture is 14 neurons, which can be seen from the unique output set vectors. The line on the graph symbolizes the MSE.

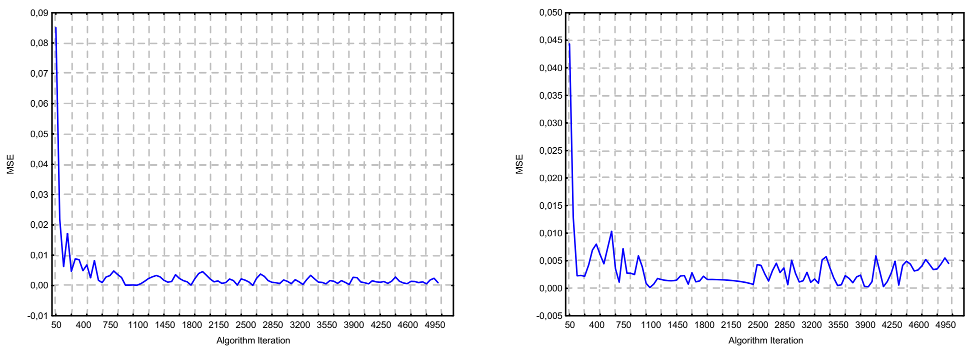

Figure 5. MSE outputs, left: 14 hidden neurons; right: 30 hidden neurons

As we can see, the smallest deviation is achieved with the 30 hidden neurons (right side of Figure 5). This is logical because each hidden neuron is allocated to one vector the network has to learn. The network with 14 hidden neurons is a minimal solution because 13 unique vectors that are represented in the entire network while there are some vectors that are doubled, but are not represented in the same state. Also, the network with 30 hidden neurons converges to a global minimum faster.

Also if we don't use any trigger neurons on the feedback loop, we noticed that by sequential execution of some vectors like vector 3 in node 2 and any other local loop vectors on the graph that by adding a slight error or a small jitter impulse like maximum of 20% noise to the input, the network will slightly deviate from it's learned output and after some iterations, it will return to it's original, learned state. This is shown on figure 6, as a deviation that continues on 3 iterations of the algorithm.

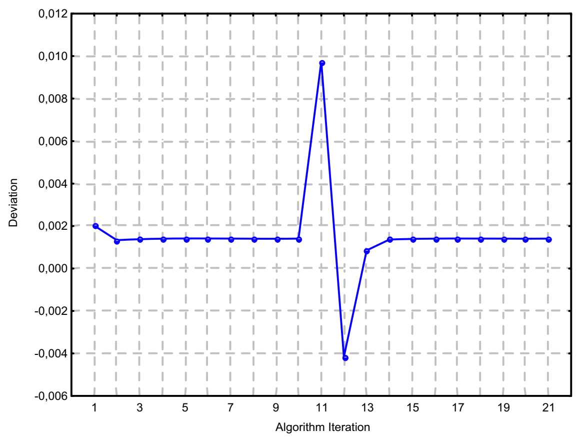

Figure 6. Output deviation and recovery based on single addition of noise

5.2. GENERATION OR CONTINUANCE OF SEQUENCES WITH THE network

Our neural network can be used to generate or continue sequences. In that order, the feedback loop works as a trigger in which the whole network continues to the next node by the automatic change in the input vectors feedback part. For example: data part from figure 3 has only 3 unique inputs: (0.3, 0.9), (0.7, 0.1) and (0.4, 0.5) if we set the data part to any of those vectors, and sequentially execute our algorithm, the network will traverse the graph's nodes based on the same value data inputs.

5.3. ALGORITHM BENCHMARKING AND CONCLUSION OF TEST RESULTS

As we tested and implemented our algorithm with mixed datasets, of one which is showed above, we found out that the algorithm gives us the ability to model ADT Graphs, and that it doesn't slow down or disrupt the classic backpropagation or any of its modified variants. Standard benchmarking datasets cannot be used because the algorithm appends additional neurons for its feedback loop (as we can see from our research), and actually only changes the way that the vectors are presented and transferred to the neural network and since there are no networks that use this principle, this is the reason why we used our datasets. The only difference is that our network needs to use additional hidden neurons to "store" the processing information because of the added data, and some modifications to the architecture and algorithms, we can conclude that this method doesn't disrupt the original backpropagation, and gives us additional possibilities stated in this paper.

6. CONCLUSION AND FUTURE RESEARCH

The feedback loop architecture can be used in areas that require memory organization abilities with all the features and benefits of neural networks. The benefits of organizing the output patterns into graphs are valued in some fields that can employ this architecture in areas that require sequential or mixed pattern recognition, pattern completion and dynamic control. Such systems require some art of output generation based on previous inputs. We will try to implement this network and some other data evaluation and extraction principles in an "adaptive expert system" that will use a neural network as part of our hybrid main data evaluation part and will function as an expert system shell.

The problem with adding the feedback part of the vector in large sets can be easily suppressed by temporally grouping the training input-output vector pairs into sets for pre-processing. First the pairs are grouped in which all output vector point to a certain node. Each of these sets gets a unique identifier which is the output vector feedback part. Next, the vector pairs are grouped in sets in which the same input vectors are representatives of their nodes, and they get their unique identifiers. This can be easily implemented into a script for pre-processing of training data sets.

Future research on this network will be based on automated learning method that will employ an ADT that can represent the memory organization graph in any form that will be a compact and effective way for training representation. Because of the simplicity of this algorithm and modifications to the architecture, we can modify any program or implementation to utilise the feedback loop architecture. Also, future works will include attempted modifications on other algorithms (See: [3] and [2]) that are not strictly backpropagation based.

REFERENCES:

[1] Arbib, M. A.: The handbook of brain theory and neural networks, MIT press 2002.

[2] Fausett, L. : Fundamentals of Neural Networks, architectures, algorithms and applications, Prentice Hall, 1994. Chapters : 5 and 6.

[3] Grossberg, S.; Carpenter, G.: Adaptive Resonance Theory from: Arbib, M. A.: The handbook of brain theory and neural networks, MIT press 2002, pages 79 – 82.

[4] Hawkins, J.: On intelligence, Times Books 2004.

[5] Principe, J.C.; Euliano, N.R.; Lefebvre, W.C.: Neural and adaptive systems, fundamentals through simulations, J.Wiley & Sons, 2000. Chapter 10 – Temporal processing with neural networks and Chapter 11 – Training and using recurrent networks.

[6] Rumelhart, D.E.; Hinton, G.E.; Williams R.J.: Learning Internal Representations by Error propagation

[7] Rumelhart, D.E.; McClelland J.L.: Parallel Distributed Processing, Volume 1, MIT Press, Cambridge, MA, 1986

[8] Webros, P.: Backpropagation: Basics and new developments, from: Arbib, M. A.: The handbook of brain theory and neural networks, MIT press 2002, pages 134 – 139.

Received: 9 December 2005
Accepted: 29 June 2006