CMST 14(1) 27-38 (2008)

DOI:10.12921/cmst.2008.14.01.27-38

COMPUTATIONAL METHODS IN SCIENCE AND TECHNOLOGY **14(1)**, 27-38 (2008)

Modular Neural Networks in Assembler Encoding

Tomasz Praczyk

Naval University, ul. Śmidowicza 69, Gdynia, Poland
T.Praczyk@amw.gdynia.pl

(Received: 23 March 208; revised 16 May 2008, accepted: 21 May 2008, published online: 3 June 2008)

Abstract: Assembler Encoding represents Artificial Neural Network in a form of a simple program called Assembler Encoding Program. The task of the program is to create the so-called Network Definition Matrix which maintains all the information necessary to construct a neural network. To generate Assembler Encoding Programs and in consequence neural networks evolutionary techniques are used. The paper addresses the problem of creating modular neural networks in Assembler Encoding. The paper discusses different methods used in Assembler Encoding for this purpose. The methods are described and analyzed in terms of their effectiveness and frequency of use in Assembler Encoding Programs.

Key words: artificial neural networks, evolution

I. INTRODUCTION

Artificial Neural Networks (ANNs) are used to solve a wide range of problems. They play more and more substantial and responsible roles and solve more and more difficult problems. As usual, in order to solve complex problems, appropriate sophisticated tools are required, hence it is also necessary to apply proper methods to create complex ANNs. A technique that is more and more frequently used for this purpose is Evolutionary Algorithms. To use the evolutionary approach to produce ANNs, we have to represent them in a form of chromosomes. However, to obtain effective, complex neural architectures by means of evolutionary techniques, we cannot simply copy the structure and parameters of ANN into chromosome(s). In such a case we would deal with long chromosomes whereas numerous experiments conducted in the field of evolution showed that applying long chromosomes may hamper or even prevent generating optimal solutions. Thus, we need ANN representation that would be relatively simple on the one hand, but on the other hand a representation that would permit creating complex neural architectures is needed. However, the desirable feature of ANNs is not the same complexity but, first of all, effectiveness. In order to make ANNs not only complex but also effective, it seems that we should try to copy the nature which superbly copes with constructing both complex and effective organisms. Note that most, if not all, living formations of the nature, which are more advanced in terms of construction, have a modular structure. Let us look at a human being. We have two very similar hands, legs, eyes, etc. Therefore, the next desirable property of the ANN encoding scheme, apart from the ability to generate complex neural architectures by means of relatively short chromosomes, is also a potential to produce modular architectures. In the present paper, a modular ANN means ANN made of several sub-ANNs [1, 15]. To obtain such ANN, it is enough to repeatedly use the information included in a genotype. If, for example, a single chromosome was a definition of ANN, then multiple use of the information contained in this chromosome would create a number of the same ANNs. To obtain one large ANN, it is sufficient to put all small ANNs together. This way, we obtain large ANN composed of many instances of some sub-ANN, i.e. modular ANN.

One of ANN encoding methods that enable creating modular ANNs is Assembler Encoding (AE). AE originates from the cellular and edge encoding although it also has features common with Linear Genetic Programming presented, among other things, in [5, 8]. AE assumes that ANN is represented in a form of a program (Assembler Encoding Program – AEP) whose structure is similar to that of a simple assembler program. The task of AEP is to create a Network Definition Matrix (NDM) containing all the information necessary to produce ANN. The process of ANN construction consists of two stages. First, AEP creates and fills up NDM. Then, the created matrix is transformed into ANN.

AE offers several methods that can be used to create modular ANNs. The mentioned methods repeatedly use the same operations or/and the same data from AEP in many areas of NDM. Thus arise NDMs including many elements of the same value. Such NDMs, in turn, represent modular ANNs. To test the potential of AE to create modular ANNs, experiments in the capture-prey problem [2-4, 6, 16] were carried out. During the experiments the task of each ANN was to control a team of agents-predators. The common goal of the predators was to capture an escaping agent-prey behaving according to a simple deterministic strategy. Since the speed of each predator was lower than or equal to the speed of the prey, the predators had to cooperate to accomplish the goal.

The main motivation to use the capture-prey problem as a test bed for AE is its similarity to the real problem in which we deal with control of a group of autonomous underwater vehicles (AUV). The task of AUVs is to guard an entrance to a harbour and to capture all dangerous underwater objects that can appear near the guarded area. The problem described above is the target problem that we want to solve within the framework of the future research.

The paper is organized as follows: section II is a short presentation of AE; section III is a detailed description of techniques that can serve to create modular ANNs; section IV illustrates the results of the experiments; and section 5 contains a summary.

II. ASSEMBLER ENCODING – FUNDAMENTALS

In AE ANN is represented in a form of a program called AEP, which is composed of two parts, i.e. a part including operations (the code part of AEP) and a part including data (the memory part of AEP). The task of AEP is to create NDM and to fill it in with values. To this end, AEP uses the operations which are run one after another. When working, the operations use data located at the end of the program (Fig. 1). Once the last operation finishes its work, the process of creating NDM is completed. The matrix created is then transformed into ANN.

II.1. Operations

AEPs can use various operations. The main task of most operations is to modify NDM. The modification can involve a single element of the matrix or a group of elements. Figures 2 and 3 present the implementation of two example operations.

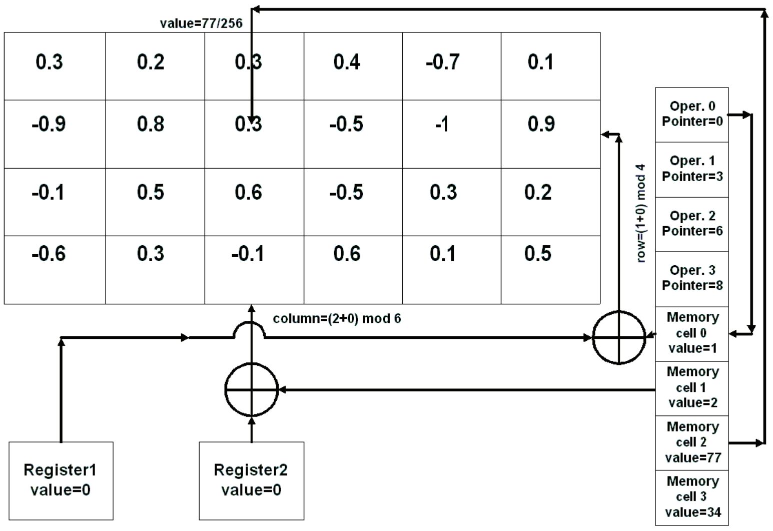


Fig. 1. Diagram of AE (AEP presented on the right includes four operations and four memory cells. Operation 0 changes a single element of NDM. To this end it uses three consecutive memory cells. The first two cells store an address to the element of NDM being updated. To determine the final address of the element mentioned values of registers are also used. The third memory cell used by Operation 0 stores a new value of the element. The value is scaled before NDM is updated. A pointer to the memory part of AEP where three cells used by Operation 0 are located, is included in the Operation itself.)

```
CHG(p0,p1,p2,*)
{
row=(abs(p1)+R1)mod NDM.width;
column=(abs(p2)+R2)mod NDM.height;
NDM[row,column]=p0/Max_value;
}
```

Fig. 2. CHG operation changing a single element of NDM (in AE two classes of operations are used, i.e. four-parameter operations and three-parameter operations [15]. The paper includes examples of only four-parameter operations. Parameters that are unimportant for implementation of the operation are marked by “*”)

```
CHGC0(p0,p1,p2,*)
{
column=(abs(p0)+R2)mod NDM.height;
numberOfIterations=abs(p2)mod NDM.width;
for(i=0;i<=numberOfIterations;i++)
      {
      row=(i+R1)mod NDM.width;
      NDM[row,column]=D[(abs(p1)+i)mod D.length]/Max_value;
      }
}
```

Fig. 3. CHGC0 operation changing a part of NDM column (D[i] denotes ith data in AEP)

The task of CHG operation, presented in Fig. 2, is to change a value of a single element of NDM. The new value of the element, stored in parameter p_0, is scaled to <-1, 1>. The address of the element being changed depends on both parameters p_1, p_2 and registers R_1, R_2. The role of the registers is detailed in the further part of the paper.

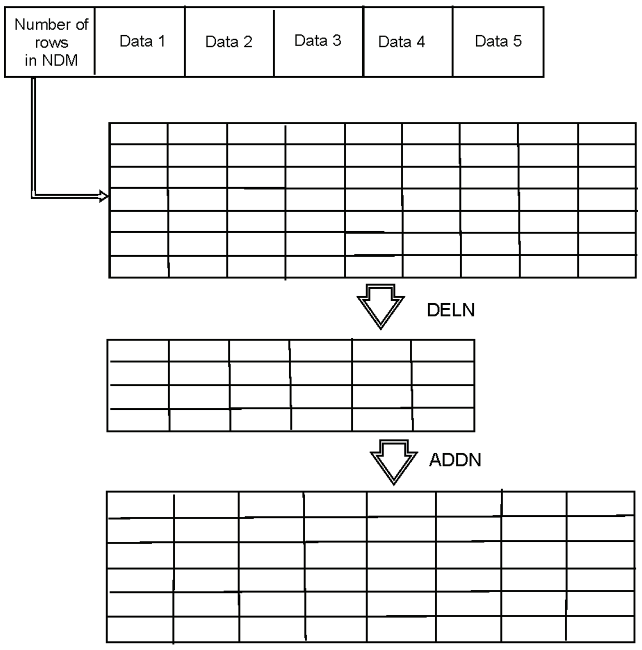


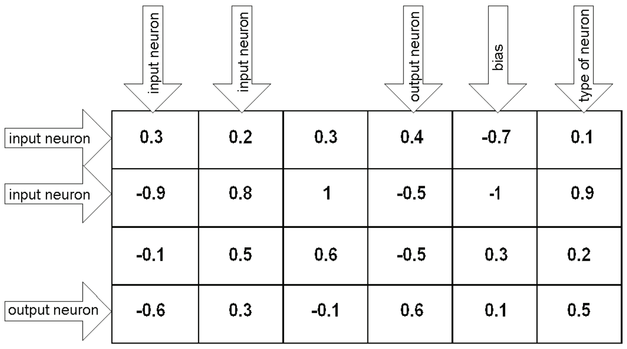
Fig. 4. Using ADDN and DELN by AEP

The CHGC0 operation presented in Fig. 3 modifies values of NDM elements located in the column indicated by parameter p_0 and register R_2. The number of elements being updated is stored in parameter p_2. The index of the first element being updated is located in register R_1. To update elements of NDM, CHGC0 uses data from AEP. The index to a memory cell including the first data used by CHGC0 is stored in p_1.

In addition to the operations whose task is to modify elements of NDM, AE also uses operations changing the size of NDM. AE assumes that the initial size of NDM is encoded in the chromosome with data (Fig. 4). Then, each AEP has some potential to modify the size of NDM by means of operations ADDN and DELN. ADDN adds new rows and columns to NDM. This procedure corresponds to adding new neurons to ANN – neurons unconnected with the rest of ANN. The number of added neurons is a parameter of ADDN. Adding new neurons does not destroy connections already established in ANN. The task of DELN is to remove a single neuron from ANN. The number of the neuron is a parameter of the operation. The elimination of the neuron practically takes place through removing the corresponding row and column from NDM.

II.2. Network Definition Matrix

Once AEP finishes its work, the process of transforming NDM into ANN is started. To construct ANN based on NDM, the latter has to include all the information necessary to create the network. When we wish to create the same skeleton of the network, i.e. the network without determined weights of interneuron connections, NDM can take a form of a classical connectivity matrix (CM) [7], i.e. a square, binary matrix of a number of rows and columns equal to the number of neurons. The value “1” in i-th column and j-th row of such a matrix means the connection between i-th neuron and j-th neuron. In turn, value “0” means a lack of the connection between these neurons. When the purpose is to create a complete ANN with determined values of weights, types and parameters of neurons, then NDM should take a form of a real valued variety of CM with extra columns or rows containing definitions of individual neurons. The example of such a matrix is presented in Fig. 5.



	input neuron	input neuron		output neuron	bias	type of neuron
input neuron	0.3	0.2	0.3	0.4	-0.7	0.1
input neuron	-0.9	0.8	1	-0.5	-1	0.9
	-0.1	0.5	0.6	-0.5	0.3	0.2
output neuron	-0.6	0.3	-0.1	0.6	0.1	0.5

Fig. 5. NDM as Connectivity Matrix

Generally, NDM can have any structure. It is necessary only to adjust the size of the matrix to the number of parameters we want to store in it.

II.3. Evolution of AEPs

In AE the evolution of AEPs proceeds according to a scheme proposed by Potter and De Jong [9-12]. The scheme assumes division of an evolutionarily created solution into parts. Each part evolves in a separate population. The complete solution is formed from selected representatives of each population. Willing to use the above scheme in relation to AEPs, it is necessary to divide them into parts. In the case of AEPs the division is natural. The operations and data make up natural parts of AEPs. Since the chosen evolutionary scheme assumes evolution of each part in a separate population, AEP consisting of *n* operations and a sequence of data evolves in *n* populations with operations and one population with data. During the evolution AEPs expand gradually. Initially, all AEPs include one operation and a sequence of data. The operations and the data come from two different populations. When the evolution stagnates, i.e. the lack of progress in the fitness of generated solutions is observed over some period, a set of the populations containing the operations is enlarged by one population. This procedure extends all AEPs created by one operation. During the evolution each population can also be replaced with the newly created population. Such situation takes place when the influence of all individuals from a given population to the fitness of generated solutions is definitely lower than the influence of individuals from the remaining populations (a population can be replaced when, for instance, the fitness of a population measured as an average fitness of all individuals from the population is definitely lower than that of the remaining populations).

III. MODULAR ANNs IN AE

In principle, AE uses two methods to create modular ANNs. In order to accomplish a modular architecture of ANN, both methods take advantage of the same fragment

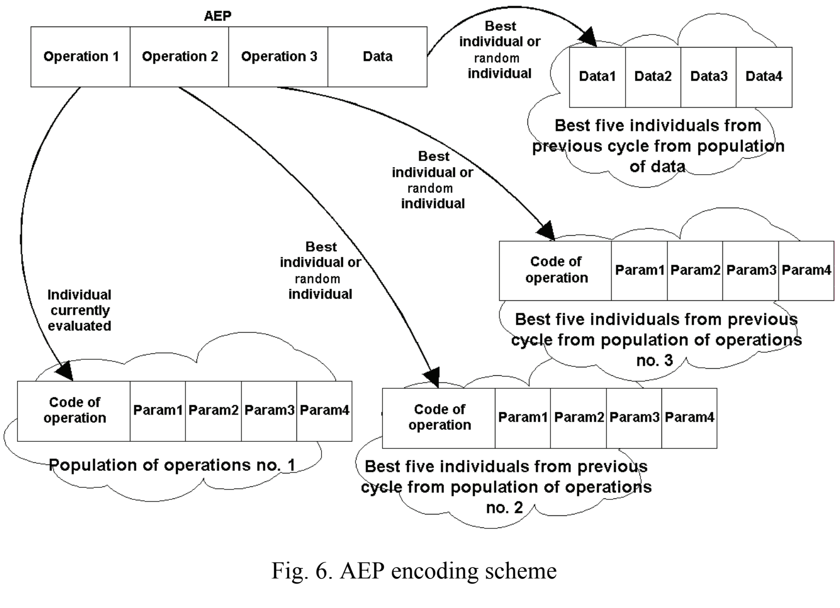

AEP
Operation 1
Operation 2
Operation 3
Data
Best individual or random individual
Data1
Data2
Data3
Data4
Best five individuals from previous cycle from population of data
Best individual or random individual
Code of operation
Param1
Param2
Param3
Param4
Best five individuals from previous cycle from population of operations no. 3
Best individual or random individual
Individual currently evaluated
Code of operation
Param1
Param2
Param3
Param4
Population of operations no. 1
Code of operation
Param1
Param2
Param3
Param4
Best five individuals from previous cycle from population of operations no. 2

Fig. 6. AEP encoding scheme

of AEP many times. Repeated use of the information included in operations and data (e.g. repeated use of the same data by different operations) is applied by the first method. Each use involves a different fragment of NDM. The second method, i.e. jumps, is based on executing the same piece of code in different places of NDM. In addition to the mentioned methods, attempts were also made to use procedures in AE as a method to construct modular ANNs (Fig. 7). However, experiments reported in [13, 14] showed that AEP encoding schemes used to form multi-procedure AEPs are considerably less effective than the AEP encoding scheme used currently in AE (Fig. 6). That is why the procedures are not used in the present version of AE.

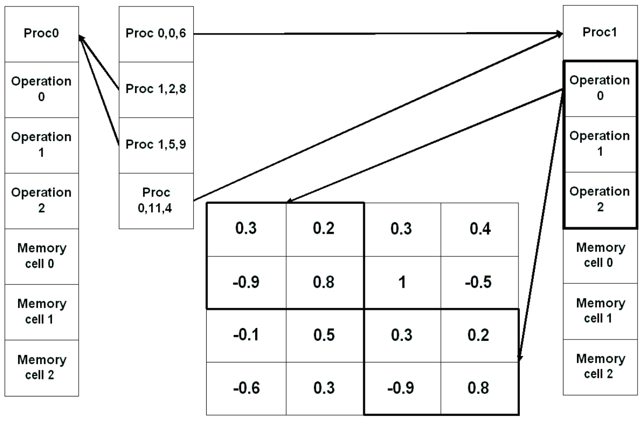


Fig. 7. Using procedures to create modular NDMs and ANNs. Each procedure could be run many times, each time in different place of NDM. The main program executed procedures one after another, changing values of registers before invoking each of them. New values for registers were stored in the main program

III.1. Repeated use of information included in operations and data

The first solution which enables AEPs to form modular neural architectures is repeated use of the same data by operations. To be performed, the operations very often have to refer to information placed in the memory part of each AEP. Because the data are common for all operations included in the same AEP, different operations can use the same data. This means that the information contained in the data part of AEP can be used many times to alter various fragments of NDM. In consequence, NDM can include the same elements in many locations, which is the base for modular neural architectures (Fig. 8) to arise.

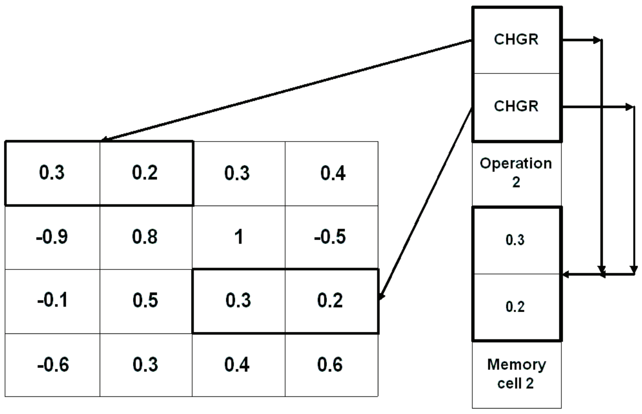


Fig. 8. Repeated use of the same data

Generally, it is also possible to repeatedly use the same data by a single operation. Let us analyze the CHGM0 operation presented in Fig. 9. Its task is to change values of elements located in a fragment of NDM. Elements are updated in columns, one after another, starting from the element pointed by parameters p_0 and p_1 of the operation. The number of changed elements and the place in the memory where new values for the elements are located, are determined by parameters p_2 and p_3. Generally, the operation can modify all elements in NDM (see variable iterations, Fig. 9). To do so, the operation uses data from AEP. However, AEP usually does not include enough

```
CHGM0(p0,p1,p2,p3)
{
rowInit=abs(p0);
columnInit=abs(p1)-1;
iterations=abs(p2)mod(NDM.width*NDM.height);
for(i=0;i<=iterations;i++)
        {
        sumRow=i mod NDM.height;
        if(sumRow == 0)
              columnInit++;
        row=(rowInit+R1+sumRow)mod NDM.height;
        column=(columnInit+R2)mod NDM.width;
        NDM[row,column]=D[(abs(p3)+i)mod D.length]/Max_value;
        }
}
```

Fig. 9. Implementation of CHGM0

```
CHGR1(p0,p1,p2,p3)
{
row=(abs(p1)+ R1) mod NDM.height;
columnInit=abs(p3) mod NDM.width;
iterations=abs(p2)mod NDM.width;
for(i=columnInit;i<=iterations+columnInit;i++)
       {
       column=(i+R2)mod NDM.width;
       NDM[row,column]= p0/Max_value;
       }
}
```

Fig. 10. Implementation of CHGR1

```
CHGR3(p0,p1,p2,p3)
{
columnInit=abs(p3) mod NDM.width;
iterations=abs(p2)mod NDM.width;
row1=(abs(p0)+ R1) mod NDM.height;
row2=abs(p1) mod NDM.height;
for(i=columnInit;i<=iterations+columnInit;i++)
       {
       column=(i+R2)mod NDM.width;
       NDM[row1,column]= NDM[row2,column];
       }
}
```

Fig. 11. Implementation of CHGR3

data so as to assign a different value to each element of NDM. In consequence, to accomplish a task the operation has to use the same data many times. They can be read many times thanks to the use of mod operator (see the last line in CHGM0).

Modularity of ANNs can also be accomplished by means of a single operation which updates a set of elements in NDM and does not use data for that purpose. This is the case, for example, with the CHGR1 operation presented in Fig. 10. It assigns the same value stored in a parameter p_0 to several elements in a column of NDM. Thus, many connections in ANN have the same strength. The information concerning the strength is stored in the operation only once.

The modular architecture of ANNs can also be generated by means of the CHGR3 operation presented in Fig. 11. In this case, new values for the elements of NDM are stored neither in the data part of AEP, nor in parameters of the operation. The operation simply copies values from one row of NDM to another. Thus, weights assigned by one operation to some interneuron connections are also assigned to other connections. Values of the weights are stored in either the data part of AEP or in some operation as parameters only once.

III.2. Jumps

The next method which can be used to create modular ANNs are jumps denoted as JMP. JMP makes it possible to repeatedly use the same code of AEP in different places of NDM. It is possible thanks to changing values of registers once the jump is performed. Figure 12 presents an example of performing AEP including the jump operation. The program mentioned proceeds as follows. First, both registers are initiated to 0. Then, the first two operations are performed, the result of which is visible in the top left corner of NDM. The next operation of AEP is the jump denoted in the figure as JMP(0,2,0,*). It first updates values of the registers and then control goes back to the first operation of AEP. AEP reads new values for the registers from the memory part. R_1 is set to 0 (Memory cell 0) whereas R_2 to 2 (Memory cell 1). Once values of the registers are updated, the two operations preceding the jump are performed once again. This time, however, working of both operations involves a different fragment of NDM. Since the jump is run overall twice, each time with different values of the registers, the two first operations of AEP are executed in three different areas of NDM.

IV. EXPERIMENTS

In order to test the potential of AE to create modular ANNs, forty NDMs and AEPs generating them were examined. Selected NDMs and AEPs represented feed-

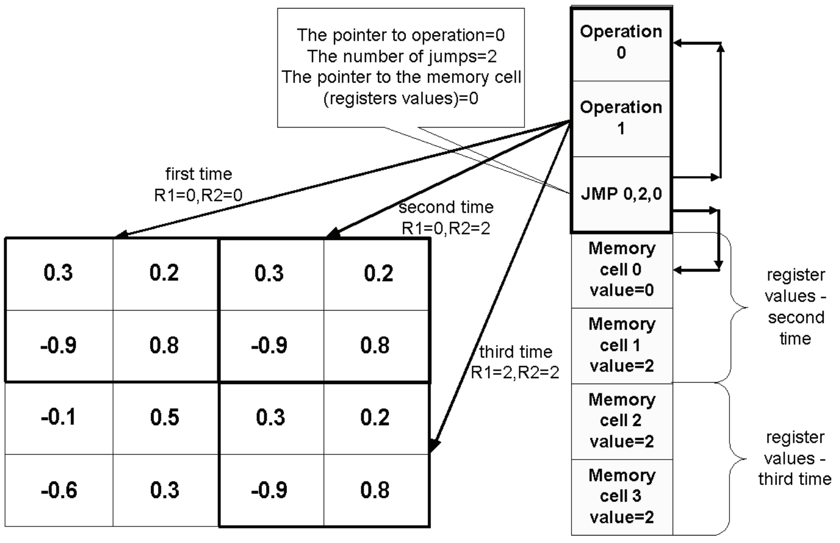


Fig. 12. Illustration of JMP operation

forward ANNs generated during the experiments in which the task of ANNs was to control a set of three cooperating predators (two predators were insufficient to capture the prey behaving according to strategy (2) described further) whose common goal was to capture a fast moving prey (the speed of the predators was either two times lower or the same as the speed of the escaping prey so it would not be enough for the predators to simply chase the prey to grasp it) behaving according to a simple deterministic strategy. Two different strategies of the prey used in the experiments are presented below.

$$\pi_1(s) = \begin{cases} \text{StandStill if } \underset{p\in P}{\forall} d(p,s) > 5 \\ \arg\max\limits_{a\in A} D\left(s, a, \arg\min\limits_{p\in P} d(p,s)\right) \text{ otherwise} \end{cases} \quad (1)$$

$$\pi_2(s) = \begin{cases} \text{StandStill if } \underset{p\in P}{\forall} d(p,s) > 5 \\ \arg\max\limits_{a\in A} \left(\frac{1}{\left|P_5(s)\right|} \sum\limits_{p\in P_5(s)} D(s,a,p) \right) \text{ otherwise} \end{cases} \quad (2)$$

P – set of predators;

A – set of actions of prey and predators $\left(A = \{\text{StandStill, North, South, West, East}\}\right)$;

$d(p,s)$ – distance between prey and predator p in state of environment s;

$D(s,a,p)$ – distance between prey and predator p in state of environment which is direct consequence of action a performed by prey in state s;

$P_5(s) = \{p \in P, d(p,s) \le 5\}$ – set of predators whose distance to prey is lower or equal to 5.

All NDMs and AEPs intended to be used in tests represented successful ANNs, i.e. ANNs which captured the prey in 20 different tested situations. ANNs generated throughout the experiments had 6 inputs (each input informed about the vertical or horizontal distance between the prey and one of the predators) and 3 outputs (3 predators). NDMs representing ANNs had a structure presented in Fig. 5.

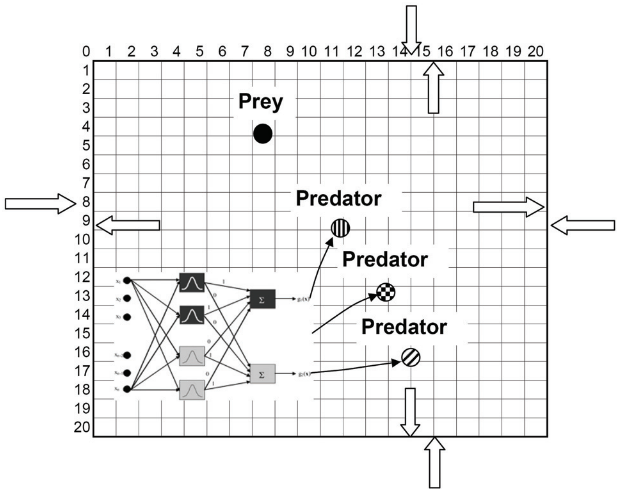


Fig. 13. Artificial world in which the predators' task was to capture prey (periodic boundary conditions are implemented and each attempt to move beyond upper, lower, right or left border of the environment above caused the object making such an attempt to move to the opposite side of the environment)

IV.1. Experimental results

To estimate modularity of generated ANNs, we decided to use the following measure:

$$M = \frac{N}{G} \qquad (3)$$

where N denotes a number of elements different from 0 located above the diagonal of NDM (all ANNs used in the experiments were feed-forward ANNs, and only elements above the diagonal were used to construct them) and G is the number of integer genes in an encoded form of AEP. N informs about the total number of parameters of ANN defined in NDM whereas G informs about the amount of information stored in the genotype defining ANN. The greater value of M, the greater degree of modularity. The greater value of M, the more information from AEP is repeatedly used to create ANN.

a)

```
Operations:
0000001 0110110 1100100 1010011 1110111
0011010 1010011 0010110 1010111 0011101
Data:
0010110 0001101 0100001 0111011 1001110
0101100 1010100 1101001 0001001 1010101
1111000 0000000 0001110 0110110 0001001
0010010
```

b)

```
0 0 0 0 0 0 0 0 0 -0.15873 0.52381 0.428571 -0.666667 0.206349
0 0 0 0 0 0 0 0 0 -0.587302 0.873016 0.571429 -0.111111 0
0 0 0 0 0 0 0 0 0 0.571429 -0.444444 0.285714 0 0
0 0 0 0 0 0 0 0 0 -0.666667 0.206349 0.698413 0.444444 0
0 0 0 0 -0.793651 -0.793651 -0.793651 0 0 -0.111111 -0.15873 0.52381 0.428571 0
0 0 0 0 0 0 0 0 0 0.285714 0 -0.587302 0.873016 0.571429
0 0 0 0 0 0 0 0 0 0.698413 0.444444 0.571429 -0.444444 0.285714
0 0 0 0 0 0 0 0 0 0.52381 0.428571 -0.666667 0.206349 0.698413
0 0 0 0 0 0 0 0 0 0.873016 0.571429 -0.111111 -0.15873 0.52381
0 0 0 0 0 0 0 0 0 -0.444444 0.285714 0 -0.587302 0.873016
0 0 0 0 0 0 0 0 0 0.206349 0.698413 0.444444 0.571429 -0.444444
```

Fig. 14. (a) example of successful AEP encoded, $G = 26$,
(b) NDM generated by AEP, $N = 64$, $M = 2.46$

a)

```
Operations:
0000001 0110110 1100100 1010011 1110111
0011010 1010011 0010110 1010111 0011101
Data:
0010110 0001101 0100001 0111011 1001110
0101100 1010100 1101001 0001001 1010101
1111000 0000000 0001110 0110110 0001001
0010010
```

b)

```
0 0 0 0 0 0 -0.714286 0 -0.603175 -0.349206 0.68254 -0.47619 0.047619 -0.047619
0 0 0 0 0 0 0.619048 0 -0.0952381 -0.253968 0.714286 -0.285714 -0.456349 -0.174603
0 0 0 0 0 0 0 0 -0.68254 0.365079 -0.904762 0.142857 0.436508 -0.126984
0 0 0 0 0 0 -0.603175 0 -0.301587 0.920635 0.619048 -0.714286 -0.0595238 0.18254
0 0 0 0 0 0 -0.0952381 0 -0.222222 -0.031746 0.285714 0.619048 0.00396825 0.384921
0 0 0 0 0 0 -0.68254 0 -0.47619 -0.984127 0.793651 -0.0952381 -0.418651 -0.0714286
0 0 0 0 0 0 -0.301587 0 -0.285714 -0.380952 -0.603175 -0.349206 -0.047619 -0.611111
0 0 0 0 0 0 -0.222222 0 -0.904762 0.142857 -0.412698 -0.0952381 -0.656746 -0.261905
0 0 0 0 0 0 -0.47619 0 0.619048 -0.714286 -0.777778 -0.68254 -0.103175 0.0714286
0 0 0 0 0 0 -0.285714 0 0.285714 0.619048 -0.698413 -0.301587 0.261905 -0.357143
0 0 0 0 0 0 0.142857 0 0.793651 -0.0952381 0.111111 -0.222222 0.246032 0.309524
```

Fig. 15. (a) example successful AEP encoded, $G = 44$,
(b) NDM generated by AEP, $N = 79$, $M = 1.79$

a)

```
CHGM1|59|19|58|34
CHGM2|45|-38|45|2
ADDN|47
CHGM2|30|29|-16|43
Data:47|18|-39|28|58|-57|-52|8|31|-51|30|45|-45|-18|-1|-18|-
54|58|5|-59|16|36|-32|19|-19
```

b)

```
Operations:
1010011 0110111 0110010 0010111 0010001
0100001 0101101 1011001 0101101 0010000
0100010 0111101 1011100 0001100 1010001
0111100 0011110 0101110 1000010 0110101
Data:
0111101 0010010 1111001 0001110 0010111
1100111 1001011 0000100 0111110 1110011
0011110 0101101 1101101 1010010 1100000
1010010 1011011 0010111 0101000 1110111
0000010 0001001 1000001 0110010 1110010
```

c)

```
0.4 0.9 0 0 0 0 0 0 0 0 0 0 -0.2 -0.5 -0.8 -0.2 -0.3
-0.8 0 0 0 0 0 0 0 0 0 0 0 0.03 0.3 0.1 -0.8 0.7
0.4 0 -0.3 -0.4 -0.4 0.1 0 0 0 0 0 -0.4 -0.5 -0.6 0.4 0.9 0.2
0.7 0 0.2 0.2 0.4 -0.1 0 0 0 0 0 -0.4 -0.5 0.5 -0.8 0.07 -0.6
-0.7 0 0.4 0.3 0.03 0.3 0 0 0 0 0 0.06 0.4 0.4 0.4 -0.9 0.4
0.9 -0.2 -0.4 -0.3 -0.4 0.1 0 0 0 0 0 0.2 -0.5 0.07 -0.6 0.7 0.2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-0.9 -0.01 -0.4 -0.1 0.1 -0.3 0 0 0 0 0 -0.4 -0.1 -0.9 0.4 -0.7 0.5
-0.8 -0.2 0.06 -0.007 0.2 0.2 0 0 0 0 0 0.2 0.8 0.2 0.9 -0.2 -0.5
0.1 -0.8 0.2 -0.1 -0.2 0.4 0 0 0 0 0 0.3 -0.3 0.5 -0.9 -0.01 0.3
```

Fig. 16. (a) Example AEP which created successful ANN (b) encoded form of AEP presented in point (a), (c) NDM generated by AEP presented in points (a) and (b)

Having used the set of forty successful AEPs and NDMs to estimate modularity of ANNs generated by means of AE, the following results were obtained:

- average $M = 1.22$
- the best $M = 2.46$
- the worst $M = 1.01$.

The results above show that AE is the method which successfully creates ANNs of modular architecture. Given selected NDMs it is necessary to state that all of them represent modular ANNs, defining repeatedly occurring schemes of interneuron connections, and the information concerned with the connections is included in AEPs generating NDMs only once.

In addition to the modularity itself, another interesting issue is how individual methods presented in section III contribute to creating modular ANNs (Tab. 1).

Table 1.

Method used in AEP	Percentage of use in successful AEPs [%]
Jump	1.34
Data (e.g. CHGM0)	59.06
Constant (e.g. CHGR1)	39.60
Copy (e.g. CHGR3)	0.00

Table 1 shows that repeated use of data by different operations is the most frequent method used by AEPs to create modular ANNs. Another method very often used by successful AEPs is using constants to update NDMs. The remaining methods are used infrequently or are not used at all.

Generally, it is necessary to state that even though NDMs very often contain repeatedly occurring values, the created ANNs do not include repeated sub-ANNs of a larger size. Usually, if some sub-ANNs repeat in ANNs, they do not contain a large number of neurons. It seems that the lack of large scale modularity in ANNs is mainly caused by the fact that there is no need to divide ANNs into large repeated sub-ANNs. In other words, each of the predators controlled by ANNs carried out a different task to capture the prey as the roles assigned to each of them were different. Consequently, ANNs could not use similar sub-ANNs to control the predators.

V. SUMMARY

The aim of the paper was to present AE in general, and to show that AE is capable of creating modular ANNs that satisfy two conditions. First, they have to be composed of sub-ANNs. Second, the information necessary to create sub-ANNs has to be stored in a genotype encoding ANN only once.

AE offers several methods that can be used to create modular ANNs. All the mentioned methods take advantage of the same fragment of AEP many times. AEPs include operations and data. In turn, the operations include parameters. To create modular ANNs, AEPs repeatedly use operations, data or parameters of the operations.

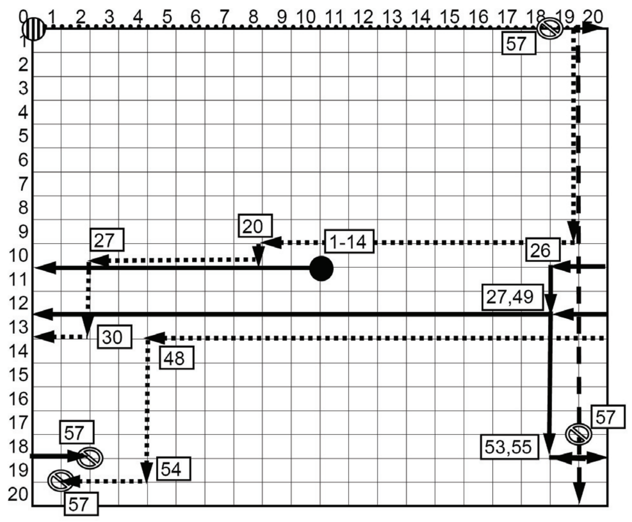


Fig. 17. Example behavior of predators and prey (neuro-controller: ANN whose NDM is presented in Fig. 16 (c)). Circles indicate initial positions of predators and prey (black circle – prey, circle with vertical stripes – predators; all predators started from the same point), round symbols with diagonal lines denote final positions, arrowed lines indicate directions of movement (solid line – prey, dashed or dotted lines – predators), whereas black boxes determine time of occurrence of individuals in given place

All the methods that can generate modular ANNs were described in the paper. Then, the potential of AE to create modular ANNs was examined. The experiments showed that AE is able to create ANNs of modular architecture. All ANNs generated throughout the experiments were modular, i.e. they included sub-ANNs which were encoded in AEPs only once. During all the experiments no case of a large scale modularity was noticed. In most cases the modularity related only to small fragments of ANNs. Lack of modularity on a large scale seems to result not from the inability of the encoding method proposed to produce large scale modular ANNs, but from the problem ANNs had to solve. The task of ANNs created during the experiments was to control a team of autonomous agents called predators. The common goal of the predators was to grasp the fast moving prey behaving according to a simple deterministic strategy. During the experiments it turned out that in order to carry out the task, each of the predators had to do something else. Thus, to control different predators ANNs could not use sub-ANNs of similar construction.

To test whether AE is able to produce large-scale modular ANNs, i.e. ANNs including sub-ANNs of a large size (in relation to the size of the whole ANN), further experiments are necessary. In the future research we plan to modify the predator–prey problem used in the current experiments so as to increase the chance to create large-scale modular ANNs. The task of ANNs in the future experiments will be to control two teams of predators. Each team will live in a separate artificial world, and to survive each of them will have to capture a prey.

Generally, to effectively control two separate teams of predators, ANNs can use different strategies. One of them is to force both teams to behave in the same way. In order for both teams to do so, ANN controlling them has to consist of the same two large sub-ANNs, i.e. it has to be a large-scale modular ANN that we want to obtain.

In the future experiments we also aim to use additional operations. The main goal of the mentioned operations will be to facilitate AEPs to create large-scale modular ANNs. The task of the simplest operation we plan to use is to fill in several columns (rows) of NDM with the same values. Since each column (row) of NDM defines a simple ANN including more neurons than sub-ANNs created so far, it seems that using the operation described above can help create large-scale modular ANNs.

References

[1] F. Gruau, *Neural network Synthesis Using Cellular Encoding And The Genetic Algorithm*, PhD Thesis, Ecole Normale Superieure de Lyon (1994).

[2] T. Haynes and S. Sen, *Co-adaptation in a team*, International Journal of Computational Intelligence and Organizations, **1(4)** (1996).

[3] T. Haynes and S. Sen, *Evolving behavioral strategies in predators and prey*. Lecture Notes in Computer Science 113-126 (1996).

[4] T. Haynes and S. Sen, *Crossover operators for evolving a team*, In: Proceedings of Genetic Programming 1997: The Second Annual Conference 162-167 (1997).

[5] K. Krawiec and B. Bhanu, *Visual Learning by Coevolutionary Feature Synthesis*, IEEE Trans. on Systems, Man, and Cybernetics, Part B: Cybernetics **35**, 409-425 (2005).

[6] G Miller and D. Cliff, *Co-evolution of pursuit and evasion: Biological and game-theoretic foundations*, Technical Report CSRP311, School of Cognitive and Computing Sciences, University of Sussex, UK (1994).

[7] G. F. Miller, P. M. Todd and S. U. Hegde, *Designing Neural Networks Using Genetic Algorithms*, Proceedings of the Third International Conference on Genetic Algorithms. 379-384. of Schaffer J. D. (1989).

[8] P. Nordin, W. Banzhaf and F. Francone, *Efficient Evolution of Machine Code for {CISC} Architectures using Blocks and Homologous Crossover*, Advances in Genetic Programming III, MIT Press, L. Spector, W. Langdon, U. O'Reilly and P. Angeline 275-299 (1999).

[9] M. Potter, *The Design and Analysis of a Computational Model of Cooperative Coevolution*, PhD thesis, George Mason University, Fairfax, Virginia (1997).

[10] M. Potter and K. A. De Jong, *Evolving neural networks with collaborative species*, In: T. I. Oren, L. G. Birta (Eds.), Proceedings of the 1995 Summer Computer Simulation Conference, 340-345. The Society of Computer Simulation (1995).

[11] M. A. Potter and K. A. De Jong, *A Cooperative Coevolutionary Approach to Function Optimization*, The Third Parallel Problem Solving From Nature, Jerusalem, Israel, 249-257, Springer-Verlag (1994).

[12] M. A. Potter and K. A. De Jong, *Cooperative coevolution: An architecture for evolving coadapted subcomponents*, Evolutionary Computation **8(1)**, 1-29 (2000).

[13] T. Praczyk, *Evolving co-adapted subcomponents in Assembler Encoding*, International Journal of Applied Mathematics and Computer Science **17(4)** (2007).

[14] T. Praczyk, *Procedure application in Assembler Encoding*, Archives of Control Science 17(LIII)**, 1**, 71-91 (2007).

[15] T. Praczyk, *Using genetic algorithms and assembler encoding to generate neural networks*, Computing and Informatics (2008) (in press).

[16] C. H. Yong and R. Miikkulainen, *Cooperative Coevolution of Multi-Agent Systems*, Technical Report AI01-287, The University of Texas at Austin (2001).

APPENDIX 1

A LIST OF OPERATIONS USED IN EXPERIMENTS

CHG – Update of element. Both the new value and address of the element are located in parameters of the operation.

CHGC0 – Update of a certain number of elements in column. Index of column, index of the first element in the column that will be changed, the number of changed elements and a pointer to data, where new values of elements are memorized, are located in parameters of the operation.

CHGC1 – Update of a certain number of elements in the column. Index of column, index of the first element in the column that will be changed, the number of changed elements and a new value for the column's elements, the same for all elements, are located in parameters of the operation.

CHGC2 – Update of a certain number of elements in the column. A new value of every element is a sum of the operation's parameter and the current value of this element. The second parameter of the operation is an index of the column. The third and fourth parameter of the operation determine the number of changed elements and index of the first element in the column that will be changed, respectively.

CHGC3 – A number of elements from one column are transformed to another column. Both columns are indicated by parameters of the operation. The number of transferred elements and index of the first element in the column that will be transferred are also included in parameters of the operation.

CHGC4 – Update of a certain number of elements in the column. A new value of every element is a sum of the current value of this element and the respective value from memory of a program. An index of the column, an index of the first element in the column that will be changed, the number of changed elements, and a pointer to data, where ingredients of individual sums are memorized, are located in parameters of the operation.

CHGR0 – like **CHGC0** but an update refers to the row of matrix**.**

CHGR1 – like **CHGC1.**

CHGR2 – like **CHGC2.**

CHGR3 – like **CHGC3.**

CHGR4 – like **CHGC4.**

CHGM0 – Change of a block of elements. Elements are updated in columns, in turn, one after another, starting from an element pointed by parameters of the operation. The number of changed elements and place in the memory where new values for elements are located are determined by parameters of the operation.

CHGM1 – like **CHGM0,** but a new value of every element is a sum of its current value and parameter of the operation.

CHGM2 – like **CHGM0,** but a new value of each element is a sum of its current value and value from the memory part of a program. The number of changed elements and place in the memory where arguments of individual sums are located are determined by parameters of operation.

JMP – Jump operation. The number of jumps, a pointer to the next operation and new values of registers are located in parameters of the jump operation.

Tomasz Praczyk. *Education:* Military University of Technology, Warsaw – MSc (1996); Maritime University, Szczecin – PhD (2001). *Activities:* intelligent navigational systems, neural networks, genetic algorithms, neuroevolution, evolutionary reinforcement learning.