Biuletyn WAT
Vol. LVII, Nr 3, 2008

Assembler Encoding — a new Artificial Neural Network encoding method

TOMASZ PRACZYK

Naval University, 81-103 Gdynia, ul. Śmidowicza 69

Abstract. The main goal of the paper is to outline a new Artificial Neural Network (ANN) encoding method called Assembler Encoding (AE). In AE, ANN is encoded in the form of a program (Assembler Encoding Program — AEP) of linear organization and of a structure similar to the structure of a simple assembler program. The task of AEP is to create the so-called Network Definition Matrix (NDM) including the whole information necessary to produce ANN. To create AEPs, and in consequence ANNs, genetic algorithms are used.
Keywords: evolutionary neural networks, encoding
Universal Decimal Classification: 004.032.26

1. Introduction

In recent years, an increasing interest in two domains of artificial intelligence, i.e. in evolutionary computation and in artificial neural networks (ANNs), has been noticed. Evolutionary techniques usually are used as global optimization methods. In turn, ANNs are applied in such problems as for example: approximation, identification, feature extraction, reinforcement learning. Successes in both domains resulted in arising a new combined domain called neuroevolution. It assumes using evolutionary approach to search for effective ANNs. Evolution of ANNs proceeds like the evolution of humans. That is, each ANN is represented in the form a genotype, i.e. a chromosome or a set of chromosomes. The chromosomes include entire information necessary to create ANN. The chromosomes encoding different ANNs are concentrated in one or more populations. During the evolution, the chromosomes are replaced by their genetically modified offspring arisen as a result of executing various genetic operators on parental chromosomes. Using a rule, according to

which a genetic material of better chromosomes, i.e. the chromosomes encoding better ANNs, has a greater chance to survive than the genetic material of worse chromosomes, results in better and better ANNs generated during the evolution.

Using Evolutionary Algorithms (EAs) to search for effective ANNs is inseparably associated with encoding of the latter. Even thought the evolutionary techniques can operate immediately on ANNs, in most cases the evolution is carried out on a genotypic level, i.e. on the level of ANNs encoded (i.e. chromosomes). In the paper, a new ANN encoding scheme, called Assembler Encoding (AE), is outlined. It assumes that each ANN is represented in the form of the program (AEP — Assembler Encoding Program) of a structure similar to the structure of a simple assembler program, i.e. it contains an executive part with operations and a memory part with data. The task of AEP is to create the matrix called Network Definition Matrix (NDM) that includes the entire information necessary to construct ANN.

2. Neuroevolution

Successes of the evolutionary approach in solving multifarious problems inspired scientists to make an attempt to apply artificial evolution to search for effective ANNs. The fact that a shape of the human brain, modeled by ANN, is determined in a chromosome and undergoes the evolution was no fewer important inspiration for researchers to undertake this problem. In fact, the evolution of ANNs does not differ from evolutionary searching for solutions in other problems. The only issue, that should be solved before EAs can be used to produce ANNs, is presenting ANN in the form of a genotype appropriate for evolutionary technique chosen, e.g. binary string, real valued vector or a tree. Figure 1 depicts the simple diagram of evolutionary design of ANNs [1]. Initially, a random population of ANNs encoded (chromosomes representing ANNs) is generated. Then, the process of selection, mutation, and recombination (optionally) takes place which results in arising offspring. In the next step, the offspring is decoded, i.e. all newly created individuals are converted into ANNs. Next, each newborn ANN can be trained by means of some learning algorithm (BackPropagation, Q-learning etc.). Once the learning process is finished, all ANNs are tested and then evaluated, i.e. a fitness is assigned to each of them (e.g. each ANN is used to control Autonomous Underwater Vehicle (AUV)). The evaluation of each ANN depends on its effectiveness in performing a task (e.g. each ANN which effectively controlled AUV obtains good fitness whereas ANNs which were ineffective obtain appropriately worse evaluations). In the last step of the evolutionary cycle, the fitness assigned to each ANN is also assigned to a corresponding genotype. The fitness of each genotype is used by EA to produce a next generation of ANNs encoded. In the consecutive generations, genetic material included in genotypes, which represented effective ANNs, has greater chance to

survive than a material contained in genotypes representing ineffective ANNs. Thus, better and better ANNs are created. The evolutionary cycle described above is repeated many times until an assumed stopping criterion is satisfied.

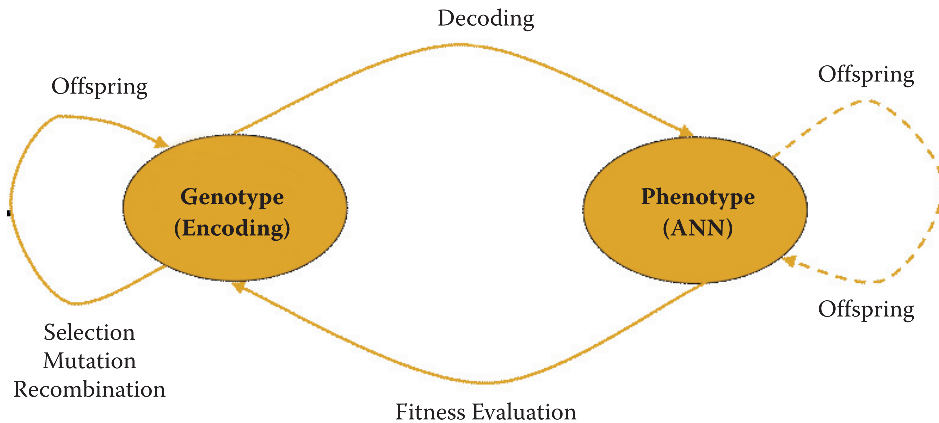


Fig. 1. Evolutionary design of ANN

2.1. Levels of evolution of ANNs

In general, the evolution of ANNs can be performed on various levels and can involve different elements of ANN [33]. In order to completely define ANN, it is necessary to determine such elements as: the topology of ANN, transfer functions of each neuron, and weights of interneuron connections. Equally important as fixing the architecture of ANN is to define a learning process of ANN. That is, it is necessary to select a learning method and to determine its parameters. Both the architecture and the learning procedure can be determined by means of the evolution. Although the evolution can be a tool whereby complete ANN can be defined, this is not always necessary. Sometimes, we only need to determine some elements of ANN, e.g. only the topology. Below, layers of the evolution in ANNs are presented.

2.1.1. Transfer functions

In this case, we deal with a population of chromosomes encoding assignments of functions to neurons. ANNs created based on the assignments are trained and put to the test. The result of the test is used to evaluate the assignments and to bias the evolution towards better assignments. Usually, the evolution of the assignments takes place together with the evolution of topologies of ANNs [11, 31].

2.1.2. Evolution of connection weights

As in the previous case, the evolution of connection weights also assumes a fixed structure of ANN (e.g. [6, 9]). Both the topology of ANN and the transfer functions of individual neurons have to be known before the evolution is started. Learning algorithms commonly used to adjust the connection weights to the task solved can be trapped in locally optimal weights what in turn can lead to difficulties in creating effective ANNs. What is more, the algorithms mentioned frequently require a differentiability of the function being optimized during the learning. Unfortunately, in real problems this condition cannot be always satisfied. In order to overcome the problems above, EAs can be used. The connection weights can be fixed solely by means of the evolution or evolutionary determined values of weights can only constitute starting values to the process of ANN learning. In the latter case, the role of EA is to generate roughly optimal weights whereas the task of the learning algorithm is to find an optimum near the point fixed be EA.

2.1.3. Evolution of topology

The topology is the next element of the architecture of ANN that can be determined by means of the evolution (e.g. [3, 12, 16, 19]). Usually, a designer of ANN does not have sufficient knowledge to determine the structure of ANN by hand. For this reason, other methods have to be used. Constructive and destructive methods were specially designed for that purpose. The constructive methods incrementally develop ANN starting from a small architecture. Initially, ANN has a small number of components to which next components are gradually added until a resultant ANN fully meets the requirements imposed. In turn, the destructive methods prepare a large fully connected ANN and then try to remove individual elements of ANN, such as synaptic connections and neurons. However, since both methods mentioned above are a form of hill-climbing and can be trapped in locally optimal topologies their use to construct ANNs is often limited. EAs are an alternative method to form neural topologies. A typical procedure to construct ANN looks in this case in the following way. First, a population of neural topologies is produced by EA. To this end, each topology has to be encoded and presented in the form of a genotype. Then, connection weights in each blank ANN are initialized (usually at random). In the next step, each ANN is trained. The learning algorithm chosen by the designer adjusts weights of interneuron connections to the problem solved. ANNs prepared in this manner undergo the evaluation which is then used by EA to guide the evolution towards better and better topologies (genotypes encoding better topologies have greater chance to survive in consecutive generations than genotypes representing worse topologies).

2.1.4. Learning rules and parameters of the learning algorithm

The next element deciding about quality of ANN is the learning algorithm used to train the network (e.g. [5, 30]). Usually, both the algorithm and its parameters

are chosen by the designer. It seems that a better approach is to fix the parameters of the learning algorithm by means of the evolution. Sometimes, a key issue is also to assign learning rules to individual interneuron connections. We deal with such a situation when the learning of ANN is carried out, for example, based on Hebb rules [5, 10]. There are several such rules and an assignment of appropriate rules to connections is crucial to the quality of ANN constructed.

2.1.5. Simultaneous evolution

One of the most interesting issues in neuro-evolution is simultaneous evolution of different elements of ANN [8, 15, 17, 18]. Commonly, the evolution involves such elements as topology and connection weights. Other possibility is to evolve the topology and assignments of learning rules to interneuron connections [27]. Connection weights are, in this case, initialized at random.

In the paper, the solution is proposed, called Assembler Encoding (AE), which makes it possible to simultaneously evolve: the topology, connection weights, assignments of transfer functions to neurons and learning rules (Hebb rules) to connections. In fact, AE can be used to produce any ANN, which can be defined in the form of a matrix of parameters. AE originates from cellular [8] and edge encoding [15] although it also has features common with Linear Genetic Programming presented among other things in [13, 20]. AE like cellular and edge encoding, is a set of operations grouped in a chromosome or as we can see in a set of chromosomes. However, there are two significant differences between the schemes mentioned above. First, chromosomes in AE are sequences of linearly ordered data or operations with arguments while in cellular and edge encoding, chromosomes take the form of trees. Second, an execution of individual operations in AE does not create ANN in a direct way, as it is in cellular and edge encoding. In AE, the whole information necessary to create ANN is stored in a special matrix called Network Definition Matrix (NDM) produced by Assembler Encoding Program (AEP). Initially, NDM is designed and ones AEP stops working ANN is constructed based on the information included in NDM.

3. Assembler Encoding — fundamentals

There are three key elements of AE — a program (AEP), the matrix representing ANN (NDM), and two auxiliary registers. AEP is an ordered set of procedures with operations and data. The operations included in the procedures possess parameters. In majority of cases, the operations determine two things, namely: addresses of elements changed (in NDM) and new values of these elements. AEP runs all procedures in turn. The operations included in each procedure are executed one after another, changing elements of NDM. They alter one or more elements of

NDM. The kind of change depends on the type of operation; in turn, the address of change is located in the registers and arguments of the operation (detailed analysis of registers' role is presented in the section where construction of modular ANNs is described). Once AEP ends working, ANN is created based on the information stored in NDM. Figure 2 depicts a diagram of AE.

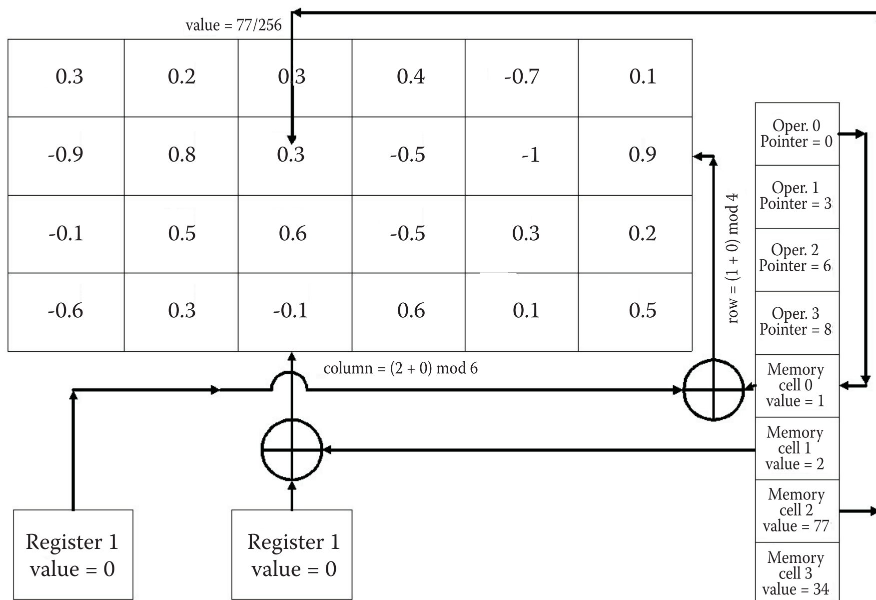


Fig. 2. Diagram of AE (Operation1 from a single-procedure AEP uses three consecutive data to change NDM)

3.1. Assembler Encoding Program (AEP)

AEP is an ordered set of procedures composed of a sequence of operations (a code part of the procedure) and data (a data part of the procedure). In principle, we deal with two classes of AEPs, i.e. single-procedure and multi-procedure AEPs. The single-procedure AEPs, as the name implies, contain a single procedure whereas the multi-procedure AEPs contain many procedures. The task of AEP is to create NDM defining ANN. To create NDM, the procedures are executed in turn, one after another. The execution of a procedure is connected with executing all its operations. Each procedure changes some fragment of NDM dependant on values of parameters of a given procedure (initially all elements in NDM are set to 0 what means that there are not any connections between neurons). AEP can contain many different procedures or it can also contain many instances of the same procedure differing in values of parameters.

The process of creating ANN consists of two sub-processes. First of them is responsible for generating NDM and is performed by AEP. The second process uses NDM to construct ANN. Both sub-processes can be run in sequential manner, i.e. one after another, or they can perform concurrently. In the first case, the process of forming NDM, by means of AEP, has to be completed to start the next process, i.e. the process of creating ANN based on NDM. In the second case, both sub-processes work simultaneously. AEP creates NDM which time to time is transformed into ANN (only new elements are propagated). This can be viewed as a process of growth of ANN from the childhood to the maturity [4, 14]. Between consecutive updates of NDM, ANN can undergo training.

Since, the process of creating ANN from NDM is completely independent of the process of creating NDM by means of AEP, a change of the former (for example, due to a change of interpretation of individual elements of NDM, i.e. values of elements of NDM can be interpreted as values of weights but at the same time they can also determine assignment of Hebb rules to individual connections within ANN) does not force us to change a construction of AEPs. This means that we can alter the way of forming ANN from NDM leaving at the same time the construction of AEPs without any change.

3.2. Network Definition Matrix (NDM)

Network Definition Matrix, as the name implies, is the matrix defining ANN. It stores the whole information necessary to create and to functioning ANN. This information is included both in the size and in individual elements of NDM (it is assumed that all elements of NDM are always scaled to the range <-1, 1>).

In principle, NDM can have any structure, i.e. it can define ANN in any way. The size of NDM determines the number of neurons in ANN created. Individual columns of NDM inform about weights of output connections of neurons but any other interpretation is also possible. The way of representing ANN by means of NDM always depends on the type of ANN we want to obtain. For example, in the experiments reported in [25, 27], two types of ANNs were used, i.e. ANNs whose architecture was permanently fixed as a result of evolutionary process as well as ANNs with Hebb learning [10, 30] whose weights undergo changes during "life" of ANN (successful use of ANNs with Hebb learning to control a real robot is presented among other things in [5, 32]). To define complete architecture of ANN, i.e. weights, topology, transfer functions, NDM can take the form of a connectivity matrix (CM) [16] whose structure is presented in Fig. 4. In turn, to represent ANN with Hebb self-organization somewhat different construction of NDM is necessary (it is important to remember that ANNs described above, i.e. ANN with Hebb learning and ANN with the whole architecture determined permanently, are only examples of ANNs which can be created by means of AE.

To create other types of ANNs, it is only enough to encode ANN in the form of a matrix of parameters).

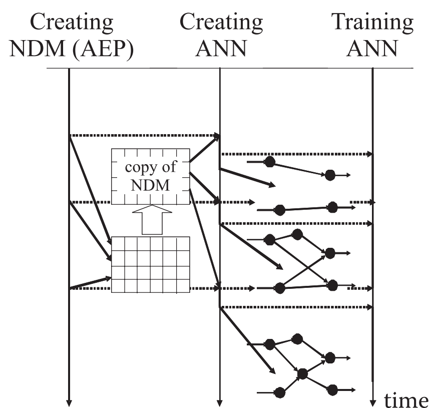


Fig. 3. The possible method of creating ANN by means of AE

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

Fig. 4. NDM used as CM

NDM used as CM is organized as follows. Each element of NDM determines synaptic weight between corresponding neurons. For example, component *i*,*j* defines a link from the neuron *i* to the neuron *j*. Elements of NDM unimportant from the point of view of the process of ANN construction, for example because of assumed feed-forward structure of ANN, are neglected during building ANN. Apart from the basic part, NDM can also possess additional columns that can

describe parameters of neurons, i.e. type of neuron (e.g. sigmoid, radial, linear), parameter of neuron and bias.

NDMs used to represent dynamical ANNs, i.e. ANNs with Hebb self-organization, have somewhat different structure from that of CMs described above. Each NDM (Fig. 5), in addition to the invariable topology of ANN, also defines the process of changes occurring in ANN during its whole "life". As a whole, NDM includes N rows and $Z = 2M + 2$ columns where N denotes the number of hidden and output neurons, whereas M is the number of all neurons in ANN. Extra two columns, as in the previous case, include additional information concerning neurons, i.e. bias and value of single parameter of a neuron (in the experiments with dynamical ANNs only sigmoid neurons were used thus, information about the type of neuron were unnecessary in NDMs). The main part of NDM used to define dynamical ANN consists of two sub-matrices of equal size ($N \times M$). The first sub-matrix determines the constant topology of ANN designed, i.e. it indicates which connections exist in ANN and which do not. Each element of this sub-matrix unequal to zero informs about connection between neurons. The sign of this element determines the sign of the connection, while the value of the element determines the type of Hebb rule assigned to the connection. For example, value –0.2 (it is assumed that all elements of NDM range <–1, 1>) of element NDM $[n, m]$ ($n = 1...N$, $m = 1...M$, neurons are indexed from 0 to M) informs about both the negative connection between m^{th} and $[n + (M - N)]^{th}$ neuron (there are not connections between input neurons) and the plain Hebb rule that is assigned to this connection. In the experiments described in [27], five types of Hebb rules were used [30]:

1. Plain Hebb rule: can only strengthen the synapse proportionally to the correlated activity of the pre- and post-synaptic neurons,

$$\Delta w = (1 - w)xy, \tag{1}$$

   where w is the synaptic weight, Δw corresponds to change in the weight w, and x, y are respectively presynaptic and postsynaptic activity of neuron.

2. Postsynaptic rule: behaves as the plain Hebb rule, but in addition it weakens the synapse when the postsynaptic node is active, and the presynaptic is not.

$$\Delta w = w(-1 + x)y + (1 - w)xy. \tag{2}$$

3. Presynaptic rule: weakens the synapse when the presynaptic unit is active but the postsynaptic is not.

$$\Delta w = wx(-1 + y)y + (1 - w)xy. \tag{3}$$

4. Covariance rule: strengthens the synapse whenever the difference between the activations of the two neurons is less than half their maximum activity, otherwise the synapse is weakened. In other words, this rule makes the synapse stronger when the two neurons have similar activity and makes it weaker when they do not

$$\Delta w = \begin{cases} (1-w)\Psi(x,y) \text{ if } \Psi(x,y) > 0 \\ w\Psi(x,y) \text{ otherwise}, \end{cases} \tag{4}$$

where $\Psi(x,y) = \tanh(4(1-|x-y|)-2)$ is the measure of a difference between presynaptic and postsynaptic activity. $\Psi(x, y) > 0$ if the difference is higher than or equal to 0.5 and $\Psi(x, y) < 0$ if the difference is smaller than 0.5.

5. "Zero" rule:

$$\Delta w = 0. \tag{5}$$

Each Hebb rule presented above corresponds to a different value of NDM.

The second sub-matrix of NDM incorporates learning rates necessary to update the strength of each synaptic weight. For example, NDM $[n, m] = -0.2$ where $n = 1...N$ and $m = M...2M$, informs that the learning rate used to update the connection between $[m - M]^{th}$ and $[n + (M - N)]^{th}$ neuron amounts $|-0.2|$. If there exists a connection between neurons but the learning rate corresponding to this connection amounts to zero, to update the strength of the connection, a nonzero value of the learning rate is used.

Hebb rules from the first part of NDM and learning rates from its second part are necessary to determine changes that take place in each interneuron connection. Each synaptic weight in ANN alters according to the following formula:

$$w_{ij}^{t} = w_{ij}^{t-1} + \eta_{ij}\Delta w_{ij} \tag{6}$$

where w_{ij}^{t}, w_{ij}^{t-1} are the synaptic weights between the jth and i^{th} neuron, respectively, after and before update and $0 \le \eta_{ij} \le 1$ is the learning rate.

First, once ANN is created, all weights of all nonzero connections are fixed in some assumed manner, for example at random. Then, synaptic weights change according to Eq. (6). All synapses can change the strength but they cannot change the sign, which is determined permanently in NDM. The synaptic strength cannot grow indefinitely. All weights are from the range <0, 1>. This is possible thanks to use of the self-limiting mechanism in all of Hebb rules mentioned above (it is

assumed that signals of neurons are also from the range <0, 1>). An update of each synaptic weight occurs once an input signal is propagated to output neurons, i.e. each time a decision has been taken by ANN.

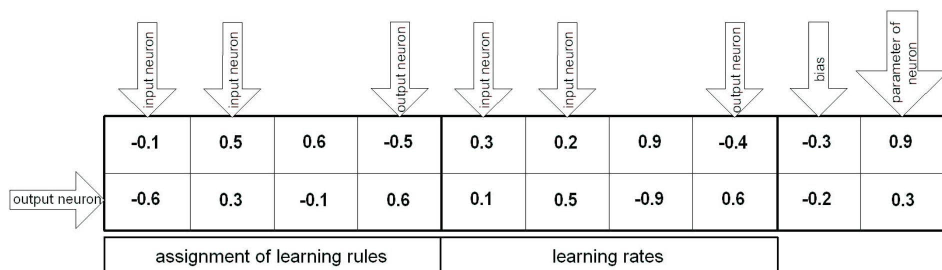


Fig. 5. NDM representing dynamical ANN with Hebb self-organization

3.3. Operations

AE uses two types of the operations, i.e. four-parameter operations and three-parameter operations. The four-parameter operations, as the name implies, have maximally four parameters. In turn, the three-parameter operations always have exactly three parameters. Below, both types of the operations are described in detail.

3.3.1. Four-parameter operations

The basic task of the operations is to change elements of NDM. The change can involve a single element or a larger set of elements of NDM. The simplest operation changes a single element in NDM. The location of change is determined in one of the parameters of the operation and in registers while the value of change is located in another parameter of the operation.

The exact implementation of the operation changing a single element of NDM can be presented as follows:

```
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. 6. CHG operation changing single element of NDM

In AE, it is assumed that each four-parameter operation can have maximum four parameters. Parameters unimportant for implementation of the operation can

be omitted and are marked with "*". In the example above used was the following notation: NDM $[i, j]$ is the element of NDM $i = 1...NDM.width$, $j = 1...NDM.height$, R_i $i = 1,2$ determines value of the ith register, *Max_value* is a scaling value which scales all elements of NDM to the range <–1, 1>. Additionally, in the further part of the paper, the following symbols are also used: $D[i]$ is the ith data in the memory of a procedure and D_{Length} is the number of memory cells.

With regard to operations that alter a larger group of elements of NDM the following operations can be imagined: the change of the whole row or column, determination of elements of a given row (column) as sum (difference) of two other rows (columns), addition (subtraction) of some constant to all elements of a row (column) etc. In the case of the operations used to change a group of elements, information involving both the address of change and the value of change is usually placed in the memory. Each operation determines only a pointer indicating an address in the memory where this information is accessible. In order to illustrate the way the operations are constructed, two examples are presented below.

```
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. 7. CHGC0 operation changing a part of column of NDM

```
CHGC6(p0,p1,*,*)
{
column1=(abs(p0)+R2)mod NDM.height;
column2=abs(p1)mod NDM.height;
for(i=0;i<NDM.width;i++)
      {
      row=(i+R1)mod NDM.width;
      NDM[row,column1]=NDM[row,column2];
      }
}
```

Fig. 8. CHGC6 operation changing the whole column of NDM

Both examples present a column of NDM change operation. CHGC6 fills the whole column indicated by p_0 and R_2 with a value from another column (pointed by p_1), whereas CHGC0 uses, for this purpose, data from memory. p_1 indicates a place in the data part of a procedure where new values for column elements are located.

To create effective AEP consisting of the operations presented above it is necessary not only to find appropriate operations and data but also to put them in a right sequence. Another approach is to exclusively use operations whose working effect does not depend on their sequence, e.g. operations whose outcome is a sum of a parameter of the operation and the value from NDM (in this case, values from NDM are not scaled to the acceptable range until the whole AEP stops working). In this solution, any sequence of operations in AEP yields the same result (in fact, some additional assumptions have to be satisfied to obtain such result, see further). The example modifications of sequence dependent operations are presented below.

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

Fig. 9. Modification of CHG

```
CHGC6(p0,p1,*,*)
{
column1=(abs(p0)+R2)mod NDM.height;
column2=abs(p1)mod NDM.height;
for(i=0;i<NDM.width;i++)
      {
      row=(i+R1)mod NDM.width;
      NDM[row,column1]= NDM[row,column1]+NDM[row,column2];
      }
}
```

Fig. 10. Modification of CHGC6

3.3.2. Three-parameter operations

The operations whose representatives are presented above have three features common. First, they always have maximum four parameters. Second, an address solely of the first updated element of NDM is stored in the parameters of the operation (in the case of the change of the whole column or the whole row one parameter of the operation is sufficient to indicate the first element; otherwise two parameters are necessary). Addresses of the remaining elements altered by the operation are fixed with reference to the first element. Third, the operations whose

examples are illustrated above always change a block of neighboring elements. There is not possible a situation in which altered elements are located in different remote fragments of NDM.

Unlike four-parameter operations, three-parameter operations use only three parameters, i.e. one integer and two lists. The integer determines either a new value for all elements altered or it indicates a place in the data part of a procedure where new values for elements updated can be found. The lists mentioned include numbers of columns and rows of NDM (The first list — *list1*, incorporates numbers of rows whereas the second list — *list2*, contains numbers of columns) which, in turn, indicate elements of NDM updated as a result of executing the operation. All possible combinations of columns and rows considered in both lists determine a set of elements altered by the operation. An example of implementation of the operation using memory to update NDM is illustrated below. A simpler version of the operation is shown in Fig. 12.

```
CHG_MEMORY(p0,list1,list2)
{
for(i=0;i<list1.length;i++)
     for(j=0;j<list2.length;j++)
          {
          row=(list1[i]+R1)mod NDM.width;
          column=(list2[j]+R2)mod NDM.height;
          NDM[row,column]=
          D[(abs(p0)+i*list2.length+j)mod D.length]/Max_value;
          }
     }
```

Fig. 11. Implementation of CHG_MEMORY operation

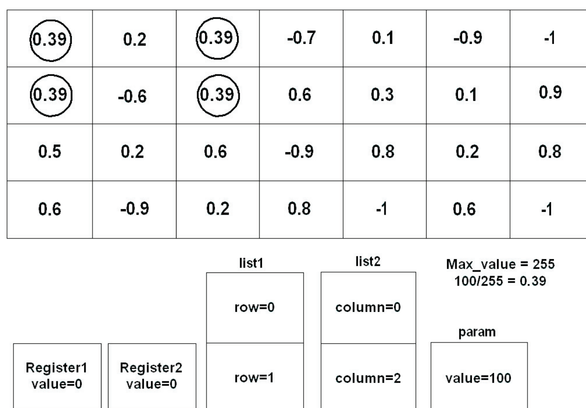


Fig. 12. Illustration of working of CHG_VALUE (changed elements are marked in circles)

3.4. Modular networks

In principle, AE uses three methods to create modular neural networks. To accomplish modular ANNs, all methods take advantage of the same fragment of AEP many times. Repeated use of the same data is applied by the first method. Each use involves other fragment of NDM. The remaining two methods are based on executing the same piece of code in different places of NDM.

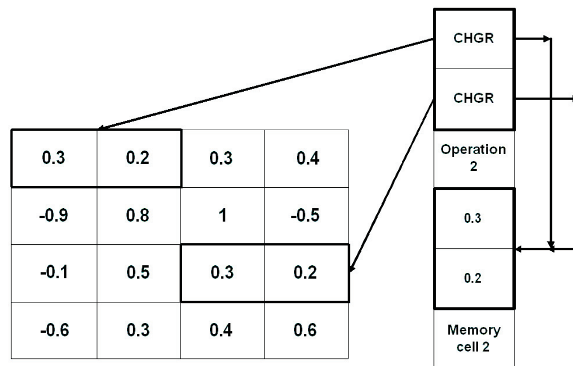


Fig. 13. Illustration of repeated use of the same data

3.4.1. Using the same data to create modular ANNs

The first method which enables AEPs to form modular ANNs is the repeated use of the same data by different operations. To operate, the operations very often have to refer to the information placed in the memory part of each procedure. Since data are common for all operations included in the same procedure, different operations can use the same data. This means that the information contained in the data part of a procedure can be used many times to alter various fragments of NDM. In consequence, NDM can include the same elements in many locations what is the basis to arise modular neural architectures.

3.4.2. Jumps

The next method which allows creating modular ANNs are jumps. Each jump indicates a place in the code part of a procedure where processing should continue (jumps are restricted to the part of the code that precedes the jump; only backward jumps are acceptable). It also determines the number of jumps and a place in the memory where new values for the registers (two values for every jump) are placed. The construction of the jump causes the same part of a code to be run in different locations, i.e. the locations indicated by the registers that are changed at the very start of the jump operation.

Figure 14 shows the situation in which the jump denoted as JMP is run two times. The sequence of two operations (Operation0 and Operation1) is executed three times, but each time in a different place of NDM. The first time, the operations are executed for initial values of the registers. The second time, after the first activation of the jump, the registers are changed to $R_1 = 0$, $R_2 = 2$. The last execution of the operations is connected with the following values of the registers: $R_1 = 2$, $R_2 = 2$.

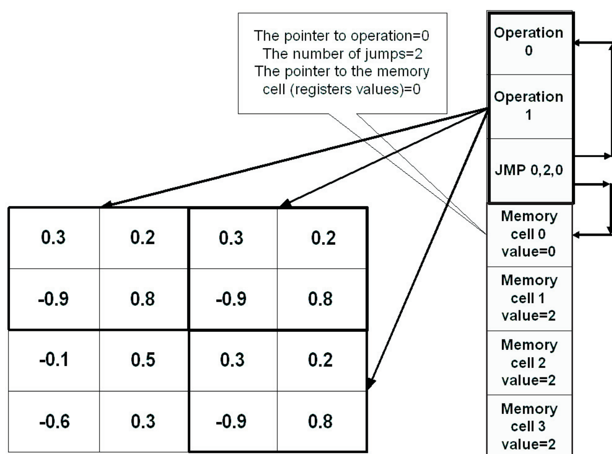


Fig. 14. Using jump

3.4.3. Procedures

The third method which enables AEPs to create modular ANNs are procedures. Each procedure can be run many times, each time in a different place of NDM.

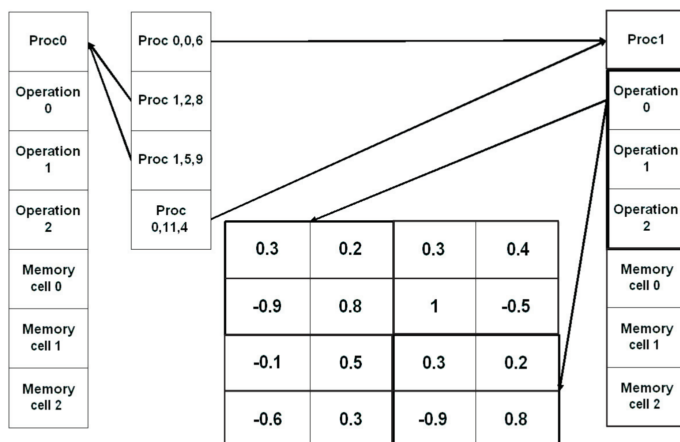


Fig. 15. Using procedures

Executing procedure in different regions of NDM is possible thanks to using the registers. Every change of NDM is conducted with respect to them. To execute a procedure in different places of NDM, it suffices to change values of the registers beforehand. New values for the registers are stored in the main program. The program executes procedures in turn changing values of the registers before invoking each of them.

3.5. Encoding operation into chromosome

Encoding the four-parameter operations is quite simple. Each chromosome-operation includes five blocks of genes. The first block determines a code of the operation (e.g. 00000 denotes CHG) while the remaining blocks contain a binary representation of four parameters of the operation.

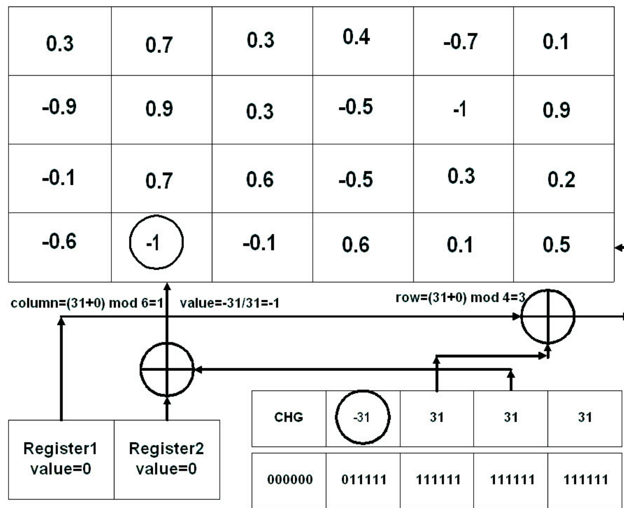


Fig. 16. Encoding CHG

The three-parameter operations are represented in a somewhat different way. Their encoded form resembles classifiers from Learning Classifier Systems [2, 7]. Similarity between the classifiers and the operations encoded results from use of the so-called *don't care* symbol "#" in both cases. Each chromosome-operation consists of four blocks of genes. The first single-bit block determines one of two possible variants of the operation. The second and the third block indicate location of changes performed by the operation (*don't care* symbol is used for this purpose). The last block specifies the value of the integer parameter of the operation.

The operations proposed in the paper have two variants. Selection of the variant depends on the value of the first bit in the chromosome-operation. The

first variant (*CHG_VALUE*) assumes that all altered elements of NDM have the same value. This value is stored in the integer parameter of the operation and is located at the end of the chromosome-operation. The second option (*CHG_MEMORY*) assigns other value to each changed element. New values for the elements are located in the data part of a procedure. The address of the first cell in the memory, where the new values for the elements are memorized, is situated in the integer parameter of the operation. Selecting elements to change consists in determining indexes of rows and columns indicating these elements. To fix indexes of rows (*list1*), the second block of genes in the chromosome-operation is used. In turn, the third block serves to determine indexes of columns (*list2*). To enable a single string of genes to represent a set of numbers we used *don't care* symbol which is treated as either 0 or 1. The more *don't care* symbols are in a string the more columns or rows it indicates. The string which does not have any *don't care* indicates a single column or row. Below, an example of use of *don't care* to locate changes in NDM is illustrated.

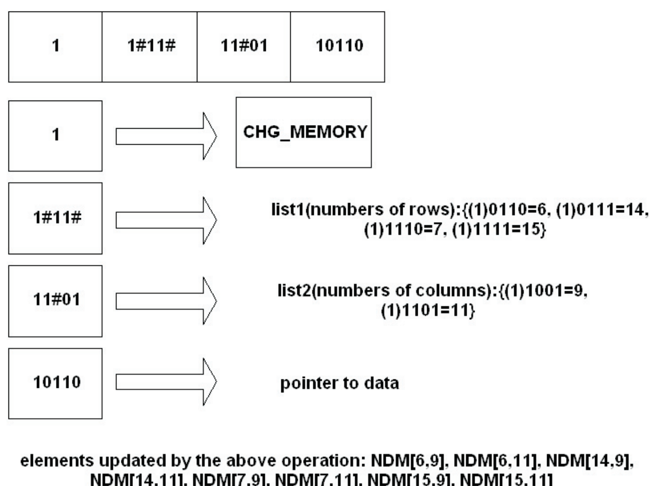


Fig. 17. Encoding three-parameter operation

3.6. Encoding AEP into chromosome(s)

There are many various methods to encode AEP into chromosome or a set of chromosomes. In this section, a few of them are presented. First, we illustrate several AEP encoding schemes that assume AEPs consisting of only one procedure. Afterwards, the schemes are showed which encode AEPs composed of many procedures.

3.6.1. Scheme 1

The simplest single-procedure AEP encoding method is to place AEP within a single chromosome. In this solution, the chromosome has to contain the whole information necessary to create AEP and NDM, i.e. the size of NDM (the size of NDM determines directly the number of neurons in ANN created. Thus, the evolution decides about the size of ANN) and a sequence of operations and data. In order to know where a borderline between operations and data is, the chromosome has to include an additional field storing this information. If this field includes a wrong value, i.e. the value which indicates, for example, a place in the chromosome which does not exist, AEP containing one operation is created. The fragment of the chromosome behind the operation is interpreted as a list of data in this case. The structure of chromosome-AEP can be presented as follows (Fig. 18).

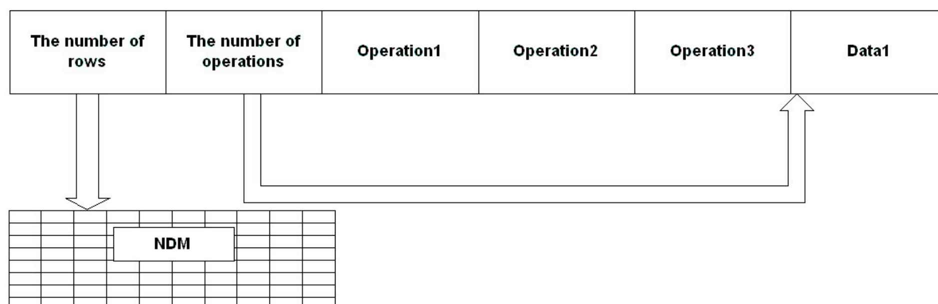


Fig. 18. AEP encoded into single chromosome (Scheme 1)

In this solution, to find effective AEPs a single population of chromosomes-AEPs is processed. Initially random chromosomes-AEPs are gradually replaced with more fit individuals representing better and better AEPs, NDMs, and in consequence ANNs.

To enable AEPs to have different number of operations and data chromosomes-AEPs have to have potential to change length. To prevent uncontrolled growth of chromosomes-AEPs, two solutions can be applied. Both solutions assume that the chromosomes can grow only to the certain limit. In the first solution, the limit to which the chromosomes can extend is constant throughout the evolution. Each crossing of a permissible length is punished, i.e. fitness of a chromosome-AEP which is too long is drastically decreased (penalizing also involves the chromosomes which are not complete, i.e. they do not include enough information to create AEP). In the second solution, the chromosomes grow gradually. First, only chromosomes consisting of not large number of operations and data are allowed. As before, too large chromosomes are also penalized. If the evolution cannot produce any effective AEP within some assumed period, the limit to which the chromosomes can extend

is increased. In consequence, larger chromosomes encoded more complex AEPs are allowed in the next step of the evolution. The process of increasing permissible length of chromosomes-AEPs continues to a threshold which cannot be exceeded.

To process the population of chromosomes-AEPs, Scheme 1 uses either Canonical GA or Steady State GA.

3.6.2. Scheme 2

A next possibility of AEP encoding is to locate its components in different chromosomes. For example, one population can store chromosomes-operations, the next one chromosomes-data and the last population can contain chromosomes-programs with pointers to individuals from the remaining two populations and with the information about the size of NDM (Fig. 19).

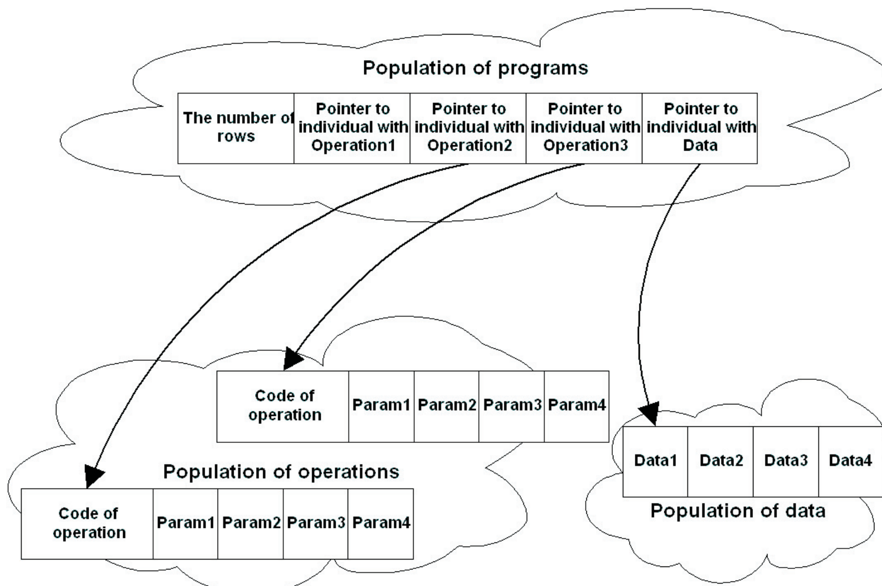


Fig. 19. Illustration of Scheme 2

Scheme 2 is similar to Moriarty and Miikkulainen SANE (Symbiotic Adaptive NeuroEvolution) approach [17, 18] in which we have a population of blueprints and a population of neurons. In our solution, the chromosomes-programs are equivalents of the blueprints which determine which operations and data cooperate well together while the chromosomes-operations and the chromosomes-data are counterparts of neurons from SANE which determine the partial architecture of ANN. Since the chromosomes-programs are equivalents of AEPs, they receive fitness of corresponding AEPs. Chromosomes-data and chromosomes-operations

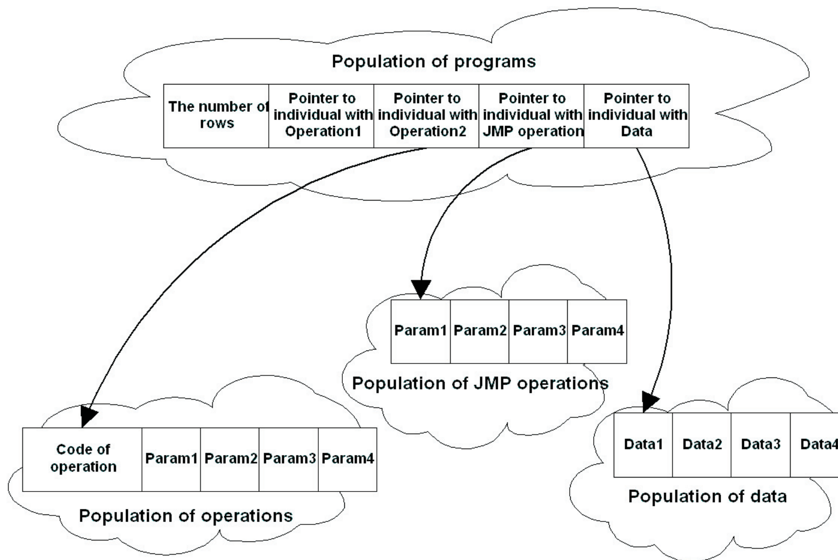


Fig. 20. Illustration of Scheme 3

which can contribute to many AEPs are evaluated in the same way as neurons in SANE. That is, they receive fitness averaged over five the best contributions of each of them.

To make it possible to create varied length AEPs, Scheme 2 uses analogical mechanism as Scheme 1, i.e. it allows the chromosomes-programs and the chromosomes-data to have different number of genes. The chromosomes-programs can contain various number of pointers to operations whereas the chromosomes-data can include different number of binary encoded integers, i.e. data. The chromosomes-operations unlike the remaining types of chromosomes used in Scheme 2 have invariable length. Since, as it turned out during preliminary experiments, the chromosomes-data do not grow so fast as the chromosomes-programs and usually do not include too many data, a procedure of limiting the length of chromosomes in Scheme 2 solely involves the chromosomes-programs. To prevent their uncontrolled growth, a similar solution is applied as in Scheme 1, i.e. each chromosome-program containing too many pointers to operations is punished. The limit for the number of pointers can be fixed permanently, at the same beginning of the evolution, or it can change in the way described in the previous section (gradual growth of the chromosomes-programs). In addition to the situation in which the chromosomes-programs are too long, they are also penalized when they are too short and they do not represent complete AEPs.

The next issue important for the scheme under consideration is selecting genetic technique to apply to each of three populations. As before, Canonical GA or Steady

State GA is used to process the population of the chromosomes-programs. To evolve individuals from the remaining two populations Scheme 2 applies algorithms with steady state replacement strategy. The population of chromosomes-operations is managed by Eugenic Algorithm [29] or Steady State GA while the population of chromosomes-data is processed by Steady State GA. Using algorithms with other than the steady state replacement strategy to process the populations including operations and data may cause a situation in which a set of pointers to operations and data effective in one generation may became completely useless in the next generation due to the change of entire population of chromosomes-operations and chromosomes-data. Creating good AEPs would be very difficult in such a case.

3.6.3. Scheme 3

This scheme is a slight modification of Scheme 2. Whereas Scheme 2 uses sequence dependant operations, Scheme 3 is the only scheme presented in the paper which uses operations whose sequence does not affect working effect of AEP. To make AEPs completely independent of the sequence of operations, no change in values of the registers can take place in the middle of the run of AEP. If such a change happened, different NDMs could be produced by means of different sequences of operations. To prevent this one copy of the jump, i.e. the only operation that can serve to change values of the registers in the scheme considered (we deal with single-procedure AEPs), is always located at the end of each AEP generated. Moreover, the jump mentioned always indicates the first operation in AEP. This way, a single execution of the whole sequence of operations preceding the jump is always performed in the same area of NDM.

In Scheme 3, to generate AEP the following set of chromosomes is required: chromosome-program, chromosomes-operations, chromosome-jump-operation, and chromosome-data (Fig. 20). All the chromosomes mentioned come from separate populations. To reward individuals from each population, the same procedure is applied as in Scheme 2, i.e. the chromosomes-programs receive fitness of corresponding AEPs while fitness of individuals from the remaining populations is averaged over the best five contributions of each of them.

To create varied length AEPs as well as to prevent uncontrolled growth of chromosomes, Scheme 3 uses the same solution as Scheme 2. The varied length AEPs are created thanks to using variable length chromosomes-programs and chromosomes-data. Limitations concerning the length of chromosomes involve only the chromosomes-programs. They can reach their maximum acceptable length immediately, i.e. at the very start of the evolutionary process, or they can increase their size gradually, step by step. The process of growth of the chromosomes-data is remained without any control. As before, the chromosomes-data do not grow so fast as the chromosomes-programs and the problem of their uncontrolled growth rather does not exist.

Scheme 3, analogically as Scheme 2, uses the following genetic algorithms: Canonical GA or Steady State GA to process a population of chromosomes-programs, Eugenic Algorithm or Steady State GA to process populations of chromosomes-operations and chromosomes-jump-operations and Steady State GA to process a population of chromosomes-data.

3.6.4. Scheme 4

Scheme 4 (single-procedure AEP encoding scheme) is an adaptation of CCGA (Cooperative Coevolutionary GA) [21, 22, 23, 24]. To create AEP, scheme 4 combines operations and data from various populations. Each population of chromosomes-operations has a number assigned determining a position of the operation from the population in AEP. In this approach, the number of operations corresponds to the number of populations including chromosomes-operations. Each population delegates exactly one representative to each AEP. At the beginning, AEPs have only one operation and a sequence of data. Both operation and data come from two different populations. Further populations including chromosomes-operations are successively added if generated AEPs cannot accomplish progress in performance over a number of co-evolutionary cycles. The populations with chromosomes-operations and chromosomes-data can be also replaced by newly created populations. Such situation takes place when contribution of a population to AEPs is considerably less than contribution of the remaining populations. In the experiments reported in [25], the contribution of a population was measured as an average fitness of operations or data contained in that population.

The approach described above makes it possible to generate many different AEPs — there are many various combinations of operations and data from different populations. In order to restrict the number of possible AEPs generated in each co-evolutionary cycle, the following solution is used [21]. In each cycle, the best five individuals from each population are selected. These individuals are used in the next cycle to create AEPs. Each AEP is created based on the individual being currently evaluated and based on the individuals belonging to the selected set of the best individuals from the previous cycle. Five AEPs are generated for each individual evaluated. One AEP is produced based on the best individuals from the previous cycle. The remaining four AEPs are constructed based on random individuals from the set of the best individuals from the previous cycle. Because each individual participates in five different AEPs, each of them receives either fitness of the best AEP in which has taken part or average fitness of its all five contributions.

The approach presented above determines the process of creating AEP. It does not say anything about initial size of NDM. Thus, we do not know whether AEP should operate on NDM consisting of nine rows and twelve columns or whether it should modify NDM containing, for example, four rows and seven columns. AE solves this problem in the following way. Initially, each AEP operates on NDM

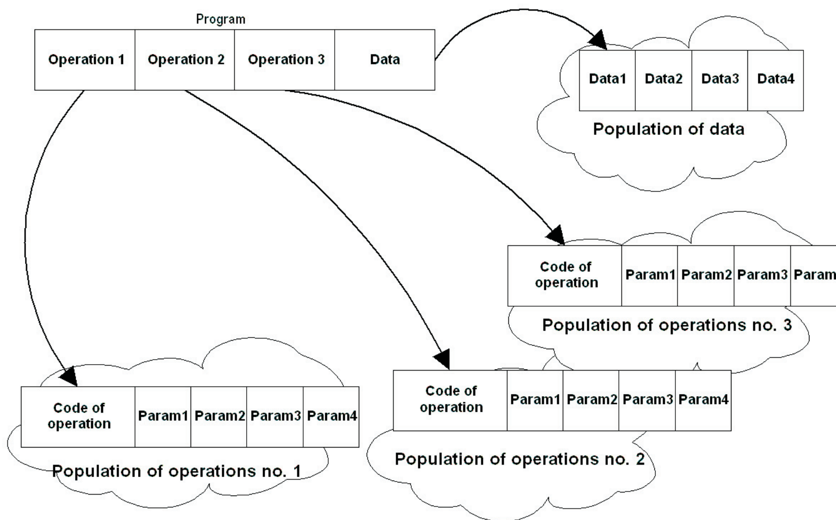


Fig. 21. Illustration of Scheme 4

containing minimal acceptable number of rows and columns (e.g. the number of rows equals the number of input and output neurons while the number of columns equals the number of rows plus extra three columns including additional information about neurons). Then, if AEPs created cannot generate any satisfying solution over some fixed number of co-evolutionary cycles, NDM is expanded by a single row and column corresponding to the next neuron.

Scheme 4 uses the following GAs: Canonical GA or Eugenic Algorithm to process populations with chromosomes-operations and Canonical GA or Steady State GA to process population with chromosomes-data.

3.6.5. Scheme 5

Scheme 5 [26] is the first scheme out of all the schemes presented so far which makes it possible to create multi-procedure AEPs. It uses two types of chromosomes, i.e. chromosomes-programs and chromosomes-procedures to create AEP. The first type of chromosomes includes the information necessary to build AEP and NDM whereas the second type contains the information needed to construct a procedure. Hence, every chromosome-program incorporates the following information: the size of NDM (and consequently the size of ANN), pointers to chromosomes-procedures and new values for the registers. The registers are updated at the very start of each procedure. In turn, each chromosome-procedure is a sequence of operations with their parameters and a sequence of data (Fig. 22). Additionally, the chromosome-procedure has to include a field indicating where is a borderline between data and operations.

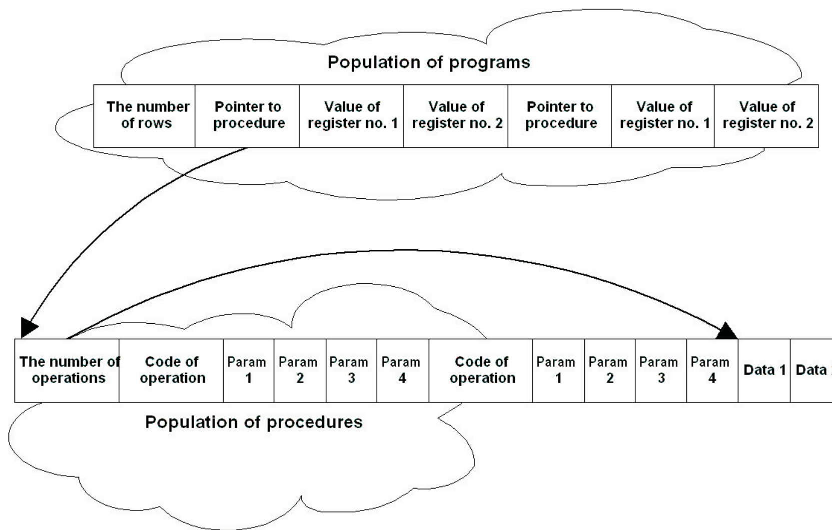


Fig. 22. Illustration of Scheme 5

The way of rewarding individuals from populations of chromosome-programs and chromosomes-procedures is analogical to the method used in Scheme 2 and Scheme 3. That is, chromosomes-programs receive fitness of corresponding AEPs whereas fitness of chromosomes-procedures is averaged over the best five contributions of each of them.

To create varied length AEPs, chromosomes-programs and chromosomes-procedures used in Scheme 5 are allowed to change their length. To prevent infinite growth of both types of chromosomes, the following solution is applied. Chromosomes-programs are penalized if AEPs created based on them include too many operations and data whereas chromosomes-procedures are punished if as a whole they include too many genes. The limit for the number of operations and data is fixed permanently at the very start of the evolutionary process while the limit for the number of genes in chromosomes-procedures increases gradually. Thus, chromosomes-programs can reach their maximum size immediately whereas chromosomes-procedures only grow if the evolution cannot produce any effective AEP within some number of co-evolutionary cycles. Both chromosomes-programs and chromosomes-procedures are also punished if they are incomplete, i.e. if they do not represent complete AEPs or procedures.

Scheme 5 uses the following GAs: Canonical GA or Steady State GA to process the population with chromosomes-programs and Steady State GA to process the population including chromosomes-procedures.

3.6.6. Scheme 6

Another AEP encoding scheme, which can be used to create multi-procedure AEPs, is Scheme 6 (Fig. 23) [26]. It assumes that the whole information necessary to construct AEP is included in four sorts of chromosomes, i.e. in chromosomes-programs, chromosomes-procedures, chromosomes-operations and in chromosomes-data. Chromosomes-programs have the same structure as previously. In turn, chromosomes-procedures are cut into three separated parts. This time, they solely include pointers to chromosomes-operations and chromosome-data. Chromosomes-data are a sequence of data while chromosomes-operations are encoded operations. To evaluate individuals from all populations specified in Fig. 23, the same procedure is applied as in the previous case. Chromosomes-programs receive fitness of corresponding AEPs while individuals from the remaining populations are evaluated based on fitnesses of the best five AEPs to which contributed each of them.

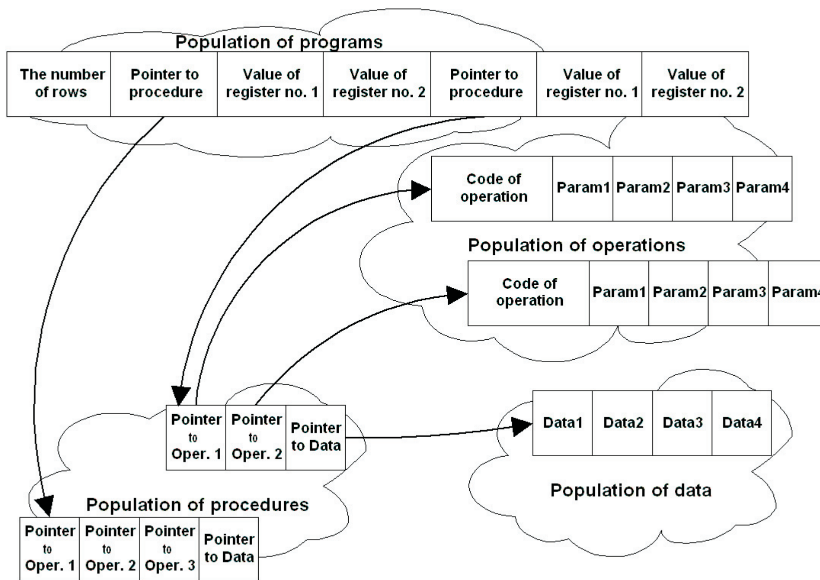


Fig. 23. Illustration of Scheme 6

In Scheme 6, we deal with three types of variable length chromosomes, i.e. chromosomes-programs, chromosomes-procedures and chromosomes-data and with one type of chromosomes which do not alter length over time, i.e. chromosomes-operations. To avoid unlimited growth of chromosomes, Scheme 6 uses the same solution as Scheme 5, i.e. chromosomes-programs are penalized if AEPs created based on them include too many operations and data whereas chromosomes-procedures are punished if they include too many pointers to operations. Chromosomes-data are unlimited in their growth. They do not grow so fast as the remaining chromosomes

and for that reason the problem of their fast growth practically does not exist. Scheme 6 like Scheme 5 assumes that chromosomes-programs can reach their maximum length immediately after staring the evolution. Chromosomes-procedures unlike chromosomes-programs grow gradually, starting from individuals including not large number of pointers to operations and data.

Scheme 6 uses the following GAs: Canonical GA or Steady State GA to process the population with chromosomes-programs, Steady State GA to process populations including chromosomes-procedures and chromosomes-data and Eugenic Algorithm or Steady State GA to process the population containing chromosomes-operations.

4. Examples of use of AE

This section briefly presents two examples of use of AE. The first example involves the situation in which AE and GAs are used to find a matrix being a solution of some optimization problem. The second case concerns the situation in which ANNs created by means of AE are used to control artificial agents-predators whose common goal is to capture a fast moving agent-prey.

4.1. Using AE in optimization problem

In [25], preliminary experiments are reported in which AE is used to solve several optimization problems. In the experiments mentioned, AE is not used to create ANNs but to create matrices being the solution to different optimization problems. During the experiments, the potential of AE to produce optimal matrices was tested. Further, two example objective functions used in the experiments and optimal matrices for these functions are presented. In both cases, the task of AEPs was to find the matrix that would maximize the function optimized. The global maximum for both test functions presented below is zero.

$$f_k(\mathbf{C}) = -\sum_{i}^{10}\sum_{j}^{10}\left|A_k[i,j]\right|, \quad k = 1,2$$

$$A_1[i,j] = \begin{cases} 0.4 - C[i,j], & ((i+j)\bmod 2) = 0 \\ 0.4 + C[i,j], & \text{otherwise} \end{cases}$$

$$A_2\left[i,j\right]=\begin{cases}0.2-C\left[i,j\right], i=j\\0.2+C\left[i,j\right], i=10-j+1\\0.4-C\left[i,j\right], i=6\wedge i\neq j\wedge i\neq 10-j+1\\0.4+C\left[i,j\right], j=6\wedge i\neq j\wedge i\neq 10-j+1\\C\left[i,j\right], \text{otherwise}\end{cases}$$

$$\mathbf{C}_{opt}^{1}=\begin{bmatrix}0.4&-0.4&0.4&-0.4&0.4&-0.4&0.4&-0.4&0.4&-0.4\\-0.4&0.4&-0.4&0.4&-0.4&0.4&-0.4&0.4&-0.4&0.4\\0.4&-0.4&0.4&-0.4&0.4&-0.4&0.4&-0.4&0.4&-0.4\\-0.4&0.4&-0.4&0.4&-0.4&0.4&-0.4&0.4&-0.4&0.4\\0.4&-0.4&0.4&-0.4&0.4&-0.4&0.4&-0.4&0.4&-0.4\\-0.4&0.4&-0.4&0.4&-0.4&0.4&-0.4&0.4&-0.4&0.4\\0.4&-0.4&0.4&-0.4&0.4&-0.4&0.4&-0.4&0.4&-0.4\\-0.4&0.4&-0.4&0.4&-0.4&0.4&-0.4&0.4&-0.4&0.4\\0.4&-0.4&0.4&-0.4&0.4&-0.4&0.4&-0.4&0.4&-0.4\\-0.4&0.4&-0.4&0.4&-0.4&0.4&-0.4&0.4&-0.4&0.4\end{bmatrix}$$

$$\mathbf{C}_{opt}^{2}=\begin{bmatrix}0.2&0&0&0&0&-0.4&0&0&0&-0.2\\0&0.2&0&0&0&-0.4&0&0&-0.2&0\\0&0&0.2&0&0&-0.4&0&-0.2&0&0\\0&0&0&0.2&0&-0.4&-0.2&0&0&0\\0&0&0&0&0.2&-0.2&0&0&0&0\\0.4&0.4&0.4&0.4&-0.2&0.2&0.4&0.4&0.4&0.4\\0&0&0&-0.2&0&-0.4&0.2&0&0&0\\0&0&-0.2&0&0&-0.4&0&0.2&0&0\\0&-0.2&0&0&0&-0.4&0&0&0.2&0\\-0.2&0&0&0&0&-0.4&0&0&0&0.2\end{bmatrix}$$

The experiments showed that AE can be successfully used to solve optimization problems in which solution can be presented in the form of a matrix. Regardless of the problem being optimized, most AEPs generated during the evolution created optimal matrices or matrices very close to the optimal. Below, example AEPs created during the experiments and matrices produced by them are presented.

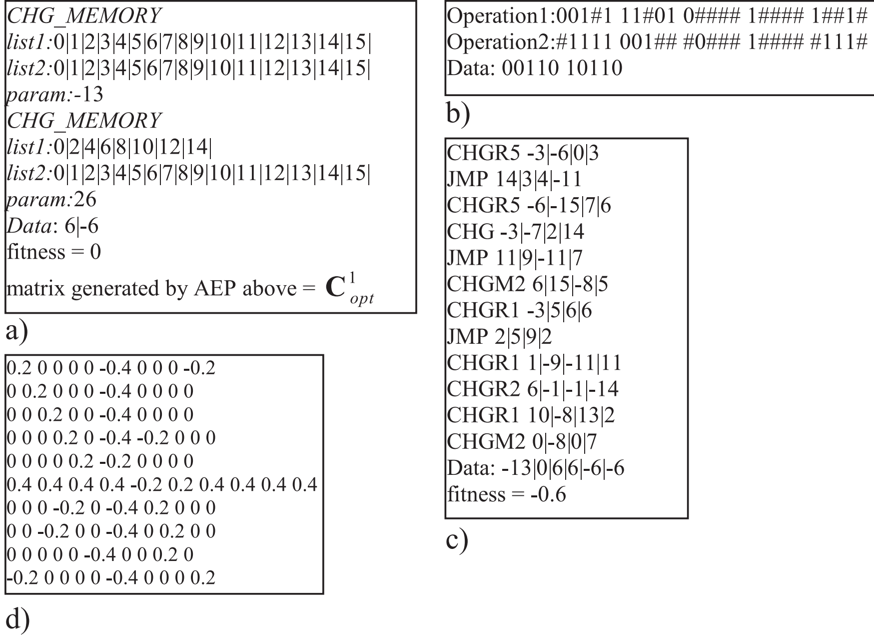


Fig. 24. (a) AEP being the solution to *f*1; (b) encoded AEP presented in point (a); (c) AEP being the solution to *f*2; (d) the matrix generated by AEP presented in point (c) (operations used in AEPs presented above are described at the end of the paper)

4.2. Using AE in predator-prey problem

The next problem in which AE was tested is a simple version of the predator-prey problem [25, 27, 28]. This time, the task of AEPs was to generate ANNs controlling a set of cooperating predators whose common goal was to capture a fast moving prey behaving according to a simple deterministic strategy. The predators and the prey used in the experiments lived in a common environment. We used 20 × 20 square without obstacles but with two barriers located on the left and on the right side of the square to represent the environment. Both barriers caused the predators as well as the prey to move right or left only to the point at which they reached one of the barriers. Moving further in the barrier direction did not cause any effect. In order to ensure infinite space for the predators and the prey and for their struggles, we made the environment open at the bottom and at the top. This means that every attempt of movement beyond upper or lower border of the square caused the object making such an attempt to move to the opposite side of the environment. As a result, the simple strategy of predators consisting in chasing the prey did not work. In such a situation, the prey in order to evade predators, could simply escape up or down.

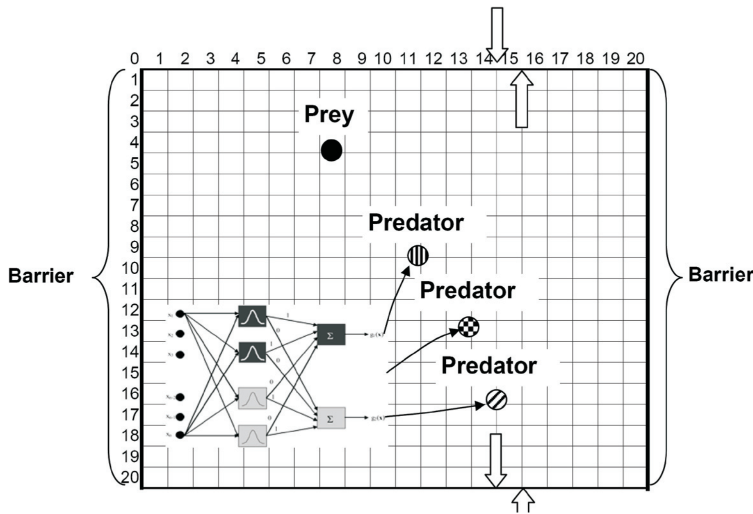
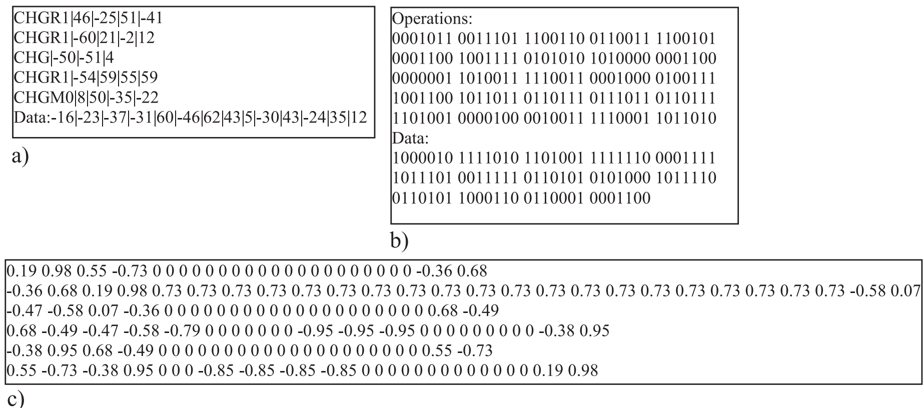


Fig. 25. Artificial world in which task of predators was to capture prey

The tests showed that AE is able to create simple ANNs. Most of ANNs created in the experiments successfully controlled the predators in all tested scenarios (tested scenarios differed in initial positions of the predators and the prey, in the speed of the prey and in the strategy used by the prey).



CHGR1|46|-25|51|-41
CHGR1|-60|21|-2|12
CHG|-50|-51|4
CHGR1|-54|59|55|59
CHGM0|8|50|-35|-22
Data:-16|-23|-37|-31|60|-46|62|43|5|-30|43|-24|35|12

a)

Operations:
0001011 0011101 1100110 0110011 1100101
0001100 1001111 0101010 1010000 0001100
0000001 1010011 1110011 0001000 0100111
1001100 1011011 0110111 0111011 0110111
1101001 0000100 0010011 1110001 1011010
Data:
1000010 1111010 1101001 1111110 0001111
1011101 0011111 0110101 0101000 1011110
0110101 1000110 0110001 0001100

b)

0.19 0.98 0.55 -0.73 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -0.36 0.68
-0.36 0.68 0.19 0.98 0.73 0.73 0.73 0.73 0.73 0.73 0.73 0.73 0.73 0.73 0.73 0.73 0.73 0.73 0.73 0.73 0.73 0.73 0.73 0.73 -0.58 0.07
-0.47 -0.58 0.07 -0.36 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.68 -0.49
0.68 -0.49 -0.47 -0.58 -0.79 0 0 0 0 0 0 0 -0.95 -0.95 -0.95 0 0 0 0 0 0 0 0 0 -0.38 0.95
-0.38 0.95 0.68 -0.49 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.55 -0.73
0.55 -0.73 -0.38 0.95 0 0 0 -0.85 -0.85 -0.85 -0.85 0 0 0 0 0 0 0 0 0 0 0 0 0 0.19 0.98

c)

Fig. 26. (a) Example of AEP which created successful recurrent dynamical ANN, (b) encoded form of AEP presented in point (a), (c) NDM (see Fig. 5) generated by AEP presented in point (a) and (b)

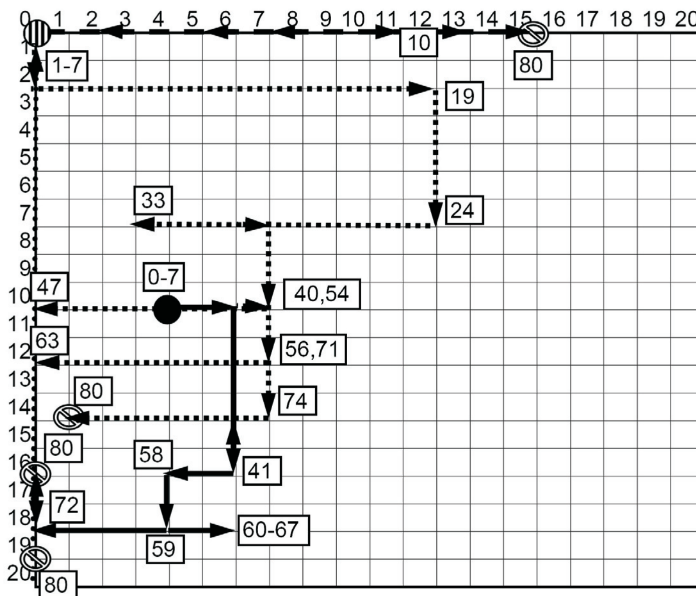


Fig. 27. Example of behavior of predators and prey in some example scenario used in the experiments (neuro-controller: recurrent dynamical ANN whose NDM is presented in Fig. 25(c)). Circles indicate initial positions of predators and prey (black circle-prey, circle with vertical stripes-predators), round 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 a given place

Summary

The paper presents a new ANN encoding scheme called Assembler Encoding. The encoding scheme proposed represents ANN in a very compact form, what allows applying GAs to create effective ANNs. Like cellular encoding and edge encoding, AE encodes ANN in the form of a program. This permits building very complex and large neural architectures by means of relatively small chromosomes. Unlike cellular and edge encoding program, which is represented as a tree, AEP is a linearly ordered set of operations and data. Another difference between cellular, edge and assembler encodings is an object, which is altered by the program. Cellular and edge encodings operate directly on a prototype of ANN. AE creates a network indirectly, changing NDM that represents ANN.

In the paper, different variants of AE are presented. The variants mentioned differ in the operations used in AEPs, in the method used to create modular ANNs and in the scheme used to encode AEP into genotype. AEPs can use two types of the operations, i.e. four-parameter operations and three-parameter operations. The four-parameter operations are encoded in the form of binary strings and they

usually change a block of neighboring elements of NDM. The three-parameter operations are encoded as strings including zeros, ones and the so-called *don't care* symbol denoted as "#". In contrast to the four-parameter operations, a single three-parameter operation can be used to change remote fragments of NDM.

To create modular ANNs, AE uses three methods. Using the same data by various operations is the first method. The remaining two methods, i.e. jumps and procedures, repeatedly use the same operations in different fragments of NDM.

In order to use GAs and AE to generate ANNs, it is necessary to encode AEP in the form of a genotype. The paper presents six schemes that can be used for that purpose. Four schemes are single procedure schemes. The remaining two schemes make it possible to create multi-procedure AEPs. The simples scheme copies all information necessary to create AEP into a single chromosome. The remaining schemes are co-evolutionary schemes in which AEPs evolve in more than a single population.

Received April 7 2008; revised May 2008.

REFERENCES

[1] K. Balakrishnan, V. Honavar, *Properties of Genetic Representations of Neural Architectures*, Proc. of the World Congress on Neural Networks (WCNN'95), 1995, 807-813.

[2] M. V. Butz, *Rule-based Evolutionary Online Learning Systems: Learning Bounds, Classification, and Prediction*, IlliGAL Report, no. 2004034, 2004.

[3] A. Cangelosi, D. Parisi, S. Nolfi, *Cell division and migration in a genotype' for neural networks*, Network: computation in neural systems, 5(4), 1994, 497-515.

[4] J. L. Elman, *Learning and development in neural networks: The importance of starting small*, Cognition, 48, 1993, 71-99.

[5] D. Floreano, J. Urzelai, *Evolutionary robots with online self-organization and behavioral fitness*, Neural Networks, 13, 2000, 431-443.

[6] D. B. Fogel, *Evolving neural networks*, Biological Cybernetics, 63, 1990, 487-493.

[7] D. E. Goldberg, *Genetic algorithms in search, optimization and machine learning*, Addison Wesley, Reading, Massachusetts, 1989.

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

[9] K. A. Gruber, J. Baurick, S. J. Louis, *Evolution of Complex Behavior Controllers using Genetic Algorithms*, http://citeseer.ist.psu.edu

[10] D. O. Hebb, *The organization of behavior*, Wiley, New York, 1949.

[11] M. W. Hwang, J. Y. Choi, J. Park, *Evolutionary projection neural networks*, In Proceedings of the 1997 IEEE International Conference on Evolutionary Computation, ICEC 97, IEEE Press, 1997, 667-671.

[12] H. Kitano, *Designing neural networks using genetic algorithms with graph generation system*, Complex Systems, 4, 1990, 461-476.

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

[14] R. I. W. Lang, *A Future for Dynamic Neural Networks*, Technical Report no. CYB/1/PG/RIWL/V1.0, University of Reading, UK, 2000.

[15] S. Luke, L. Spector, *Evolving Graphs and Networks with Edge Encoding: Preliminary Report*, in: John R. Koza, editor, Late Breaking Papers at the Genetic Programming, 1996, Conference Stanford University July 28-31, pages 117-124, Stanford University, CA, USA, Stanford Bookstore, 1996.

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

[17] D. E. Moriarty, R. Miikkulainen, *Forming Neural Networks Through Efficient and Adaptive Coevolution*, Evolutionary Computation, 5(4), 1998, 373-399.

[18] D. E. Moriarty, *Symbiotic Evolution of Neural Networks in Sequential Decision Tasks*, PhD thesis, The University of Texas at Austin, TR UT-AI97-257, 1997.

[19] S. Nolfi, D. Parisi, *Growing neural networks*, in: C. G. Langton (ed.) Artificial Life III, Reading, MA: Addison-Wesley, 1992.

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

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

[22] M. Potter, 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.

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

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

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

[26] T. Praczyk, *Procedure application in Assembler Encoding*, Archives of Control Science, vol. 17(LIII), no. 1, 2007, 71-91.

[27] T. Praczyk, *Forming dynamical, self-organizing neural networks by means of assembler encoding* (in review).

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

[29] J. W. Prior, *Eugenic Evolution for Combinatorial Optimization*, Master's thesis, The University of Texas at Austin, TR AI98-268, 1998.

[30] J. Urzelai, D. Floreano, *Evolution of Adaptive Synapses: Robots with Fast Adaptive Behavior in New Environments*, Evolutionary Computation, 9(4), 2001, 495-524.

[31] D. White, P. Ligomenides, *GANNet: a genetic algorithm for optimizing topology and weights in neural network design*, In Proceedings of International Workshop on Artificial Neural Networks (IWANN 93), Springer Verlag, 1993, 322-327.

[32] D. Willshaw, P. Dayan, *Optimal plasticity from matrix memories: What goes up must come down*, Neural Computation, 2, 1990, 85-93.

[33] X. Yao, *Evolving Artificial Neural Networks*, in: Proceedings of the IEEE, 87(9), 1999, 1423-1447.

APPENDIX 1
— List of operations presented in figures no. 23 and 25

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

CHGR1 Update of certain number of elements in row. Index of row, index of first element in row that will be changed, number of changed elements and new value for row's elements, the same for all elements, are located in parameters of operation.

CHGR2 Update of certain number of elements in row. New value of every element is sum of operation's parameter and current value of this element. The second parameter of operation is index of row. Third and fourth parameter of operation determine respectively number of changed elements and index of the first element in row that will be changed.

CHGR4 An update of certain number of elements in row. New value of every element is a sum of current value of this element and respective value from memory of AEP. An index of the row, an index of the first element in the row 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 operation.

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

CHGM2 like CHGM0 but new value of each element is a sum of its current value and value from 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 next operation and new values of registers are located in parameters of jump operation.

CHG_MEMORY Change of set of elements of NDM indicated by parameters of operation. Values of updated elements are located in the memory.

T. PRACZYK

Kodowanie Asemblerowe — nowa metoda kodowania sieci neuronowych

Streszczenie. Głównym celem artykułu jest przedstawienie Kodowania Asemblerowego czyli nowej metody kodowania sztucznych sieci neuronowych. W Kodowaniu Asemblerowym sieć neuronowa jest zakodowana w postaci programu (AEP — Assembler Encoding Program) o liniowej organizacji

i o strukturze podobnej do struktury prostego programu asemblerowego. Zadaniem AEP jest stworzenie tzw. Macierzy Definicji Sieci (NDM — Network Definition Matrix) zawierającej całą informację potrzebną do stworzenia sieci. Tworzenie AEP i w konsekwencji sieci neuronowych odbywa się z wykorzystaniem technik ewolucyjnych.
Słowa kluczowe: ewolucyjne sieci neuronowe, kodowanie
Symbole UKD: 004.032.26