Computing and Informatics, Vol. 28, 2009, 895–912

USING ASSEMBLER ENCODING TO SOLVE INVERTED PENDULUM PROBLEM

Tomasz Praczyk

Naval University
Śmidowicza 69
Gdynia, Poland
e-mail: T.Praczyk@amw.gdynia.pl

Manuscript received 22 October 2008; revised 7 May 2009
Communicated by Vladimír Kvasnička

Abstract. Assembler Encoding is Artificial Neural Network encoding method. To date, Assembler Encoding has been tested in two problems, i.e. in an optimization problem in which a solution is in the form of a matrix and in the so-called predator-prey problem in which the task of ANN is to control agent-predators whose common goal is to capture a fast moving agent-prey. The next problem in which Assembler Encoding was tested is the inverted pendulum problem. In the experiments Assembler Encoding was compared to several evolutionary and reinforcement learning methods. The results of the tests are presented at the end of the paper.

Keywords: Evolutionary neural networks, reinforcement learning

1 INTRODUCTION

Assembler Encoding (AE) is Artificial Neural Network (ANN) encoding method. To test AE several experiments were carried out. The experiments involved two testing problems. The first of them was the optimization problem. In the experiments the task of AE was to generate matrices being a solution of several optimization problems [15, 16, 18]. The main goal of the experiments in this field was to define the initial version of AE. The second test-bed for AE was the predator-prey problem [15, 20]. In the experiments AE was responsible for creating neural controllers for a team of agent-predators. The task of the predators was to capture a single fast-moving

agent-prey behaving by a simple deterministic or stochastic strategy. The tests in the predator-prey problem were necessary to determine the final version of AE.

The next problem in which AE was tested is the inverted pendulum problem. It is a well known testing problem which is very often used to test reinforcement learning (RL) or evolutionary RL methods. Generally, the problem mentioned has many versions. In the experiments the simplest version of the problem, i.e. the version with a single pole and with complete information about the card-and-pole system available to the controller, was tested. To compare AE with other methods the experiments were carried out in the same conditions as the experiments reported in [10].

The paper is organized as follows: Section 2 is a short presentation of AE; Section 3 is a description of the inverted pendulum problem; Section 4 is a short presentation of compared methods; Section 5 is the report on the experiments; and Section 6 is the summary.

2 FUNDAMENTALS OF AE

In AE [15–17] ANN is represented in the form of a program called AEP. AEP is composed of two parts, i.e. a part including operations and a part including data. The task of AEP is to create NDM and to fill it with values. To this end, AEP uses the operations. The operations are run in turn. When working the operations can use data located at the end of AEP (Figure 1). Once the last operation finishes its work the process of creating NDM is completed. NDM is then transformed into ANN.

2.1 Operations

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

The task of CHG (Figure 2) is to change a single element of NDM. The new value of the element, stored in parameter p_0, is scaled to $\langle -1, 1 \rangle$. An address of the element being changed depends on both parameters p_1, p_2 and registers R_1, R_2. A role of the registers is detailed in the following part of the paper. CHGC0 (Figure 3) modifies elements of NDM located in a column indicated by parameter p_0 and the register R_2. The number of elements being updated is stored in a parameter p_2. An index of the first element being updated is located in the register R_1. To update elements of NDM CHGC0 uses data from AEP. An 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 the content of NDM AE also uses the jump operation denoted as JMP. This operation makes it possible to repeatedly use the same code of AEP in different places of NDM. The above is possible, owing to changing values of the registers once jump is performed. Figure 4

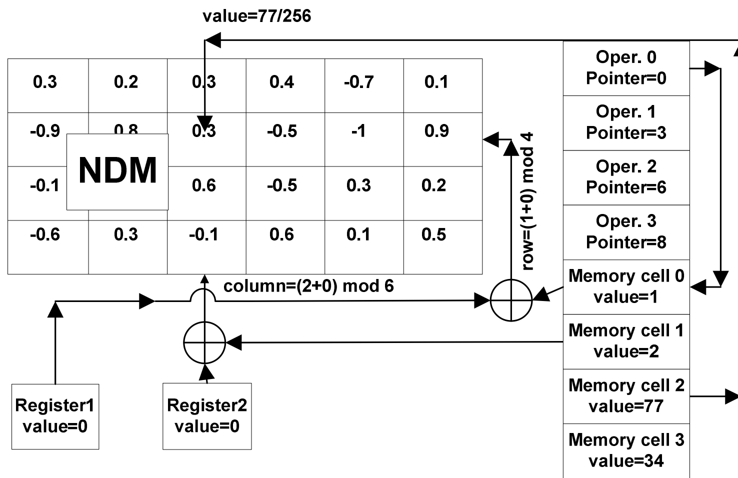

Fig. 1. Diagram of AE (AEP presented on the right includes four operations and four memory cells. Operation 0 changes a single element of NDM. To this end, it uses three consecutive memory cells. The first two cells store an address to the element of NDM being updated. To determine the final address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

$$
\begin{aligned}
&\mathrm{CHG}(p_0, p_1, p_2, *)\\
&\{\\
&\quad \mathrm{row} = (\mathrm{abs}(p_1) + R_1) \bmod \mathrm{NDM.width}\\
&\quad \mathrm{column} = (\mathrm{abs}(p_2) + R_2) \bmod \mathrm{NDM.height}\\
&\quad \mathrm{NDM[row, column]} = p_0/\mathrm{Max_value};\\
&\}
\end{aligned}
$$

Fig. 2. CHG operation

presents an example of performing AEP including the jump operation. The program mentioned proceeds as follows. First, the registers are initiated. Both of them are set to 0. Then, the first of the two operations are performed. The result of performing the operations is shown in the left top corner of NDM. The next operation of AEP is the jump denoted in Figure 4 as $\mathrm{JMP}(0, 2, 0, *)$. It first updates the values of the registers and then the control goes back to the first operation of AEP. AEP reads the 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

$$
\begin{aligned}
&\mathrm{CHGC0}(p_0, p_1, p_2, *)\\
&\{\\
&\quad \mathrm{column} = (\mathrm{abs}(p_0) + R_2) \bmod \mathrm{NDM.height};\\
&\quad \mathrm{numberOfIterations} = \mathrm{abs}(p2)\,\mathrm{modNDM.width};\\
&\quad \mathrm{for}(i = 0; i <= \mathrm{numberOfIterations}; i{+}{+})\\
&\quad \{\\
&\qquad \mathrm{row} = (i + R_1) \bmod \mathrm{NDM.width};\\
&\qquad \mathrm{NDM}[\mathrm{row}, \mathrm{column}] = D[(\mathrm{abs}(p_1) + i) \bmod D.\mathrm{length}]/\mathrm{Max_value};\\
&\quad \}\\
&\}
\end{aligned}
$$

Fig. 3. CHGC0 operation changing a part of column of NDM ($D[i]$ is i^{th} data in AEP, D.length is the number of data in the data part of AEP)

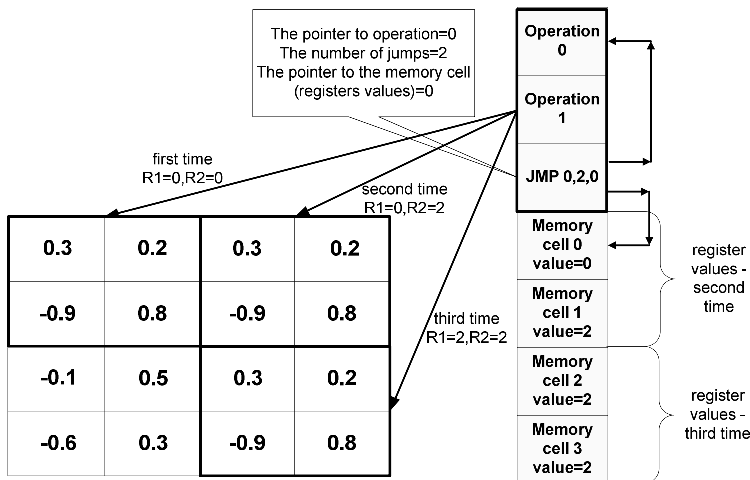

Fig. 4. JMP operation

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 two times, each time with different values of the registers, the two first operations of AEP are executed in three different areas of NDM.

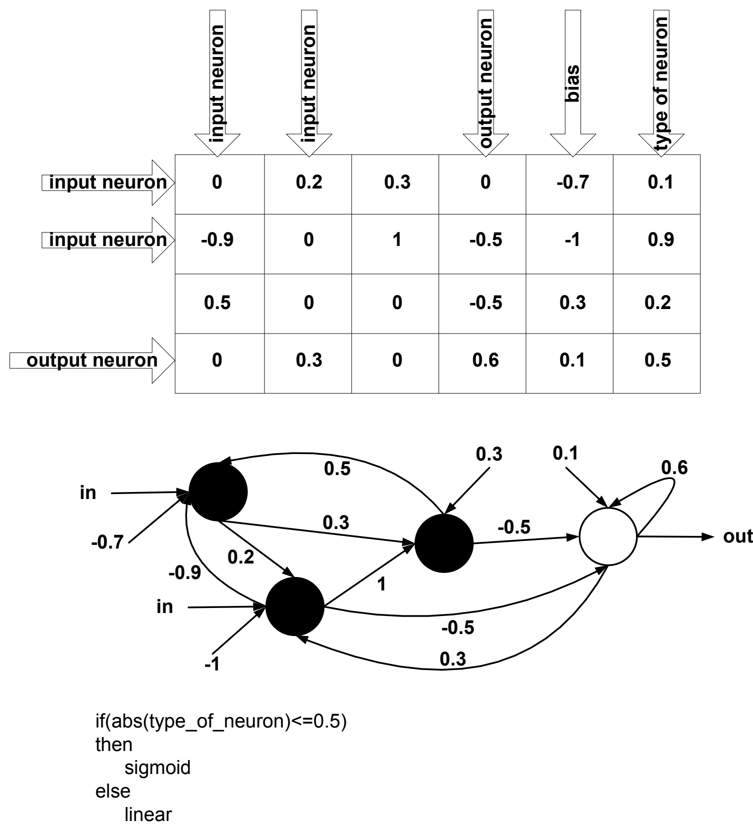

Fig. 5. NDM and ANN created based on the matrix

2.2 Network Definition Matrix

Once AEP finishes its work the process of transforming NDM into ANN is started. To enable construction of ANN based on NDM the latter has to include the whole information necessary to create ANN. When we wish to create the same skeleton of ANN, i.e. ANN without determined weights of interneuron connections, NDM can take the form of the classical connectivity matrix (CM) [9], i.e. a square, binary matrix of a number of rows and columns equal to a number of neurons. The value "1" in the ith column and j^{th} row of such a matrix means a connection between the ith neuron and j^{th} neuron. In turn, the value "0" means lack of the connection between these neurons. When the purpose is to create complete ANN with determined values of weights, types of neurons, parameters of neurons then NDM should take the 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 Figure 5.

2.3 Evolution in AE

In AE, evolution of AEPs and in consequence ANNs proceeds according to the scheme proposed by Potter and De Jong [11–14]. The scheme assumes a division of evolutionarily created solution into parts. Each part evolves in a separate population. The complete solution is formed from selected representatives of each population. To use the scheme above 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 evolutionary scheme chosen 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. lack of progress in 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 newly created population. Such situation takes place when the influence of all individuals from a given population on fitness of generated solutions is definitely lower than the influence of individuals from the remaining populations (a population can be replaced when, for example, fitness of a population measured as the average fitness of all individuals from the population is definitely lower than the fitness of the remaining populations).

To evaluate individuals from different populations the following solution is used. First, AEP including an individual evaluated is created. To this end, the individual is combined with selected individuals from the remaining populations (in the experiments reported in the further part of the paper each individual evaluated was combined with the best individuals from the previous generation (Figure 6)). AEP creates and fills in NDM which is then transformed into ANN. ANN is tested and evaluated. The result of the test determines fitness of the individual evaluated.

In AE, the operations and data are usually encoded in the form of binary strings (in AE, two classes of operations are used, i.e. four-parameter operations encoded as binary strings and three-parameter operations encoded as strings including zeros, ones and the so-called don't cares – "#" [3, 5, 6]. In the experiments reported in the paper only four-parameter operations are used). Each chromosome operation includes five blocks of genes. The first block determines a code of the operation, while the remaining blocks contain a binary representation of four parameters of the operation (e.g. 01000|11000|01000|00000|00100 represents the following operation: CHGC0|−1|1|0|2). Chromosome data are vectors including binary encoded integers. Each integer encodes a single element of data. In AE, all chromosome operations have the same length. Chromosome data can change the length during the evolutionary process.

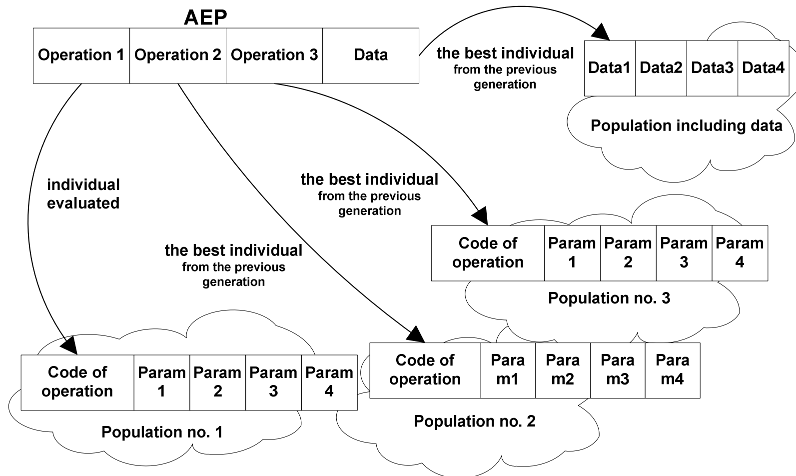

Fig. 6. Evolution of AEPs

3 THE INVERTED PENDULUM PROBLEM

The task of a controller in the inverted pendulum problem is to protect a pole (inverted pendulum), installed on a card, from falling down (it is assumed that the pole is up when an angle of the pole does not exceed an acceptable angle). To do so, the controller has to push the card left or right with some fixed force. An additional requirement for the controller is not to exceed the limit of a track on which the card moves. To accomplish the task the controller uses the following state information: the position of the card (σ), the velocity of the card ($\dot{\sigma}$), the angle of the pole (θ), and the angular velocity of the pole ($\dot{\theta}$). At each time step, the controller has to decide about direction of movement: right or left. The behavior of the card-and-pole system under the influence of the controller (the force F) can be presented by means of the following equations:

$$\ddot{\theta}_t = \frac{mg\sin\theta_t - \cos\theta_t\left(F_t + m_p l\dot{\theta}_t^2\sin\theta_t\right)}{(4/3)m_l - m_p l\cos^2\theta_t} \tag{1}$$

$$\ddot{\sigma}_t = \frac{F_t + m_p l\left(\dot{\theta}_t^2\sin\theta_t - \ddot{\theta}_t\cos\theta_t\right)}{m}. \tag{2}$$

In the experiments reported below the following parameters of the card-and-pole system were used:

- l – the length of the pole = 0.5 m
- m_p – the mass of the pole = 0.1 kg
- m – the mass of the card-and-pole system = 1.1 kg

- F – the magnitude of the force = 10 N
- g – gravity = 9.8 m/s^2
- the limit of the track = 4.8 m
- the acceptable angle of the pole = ±12°.

The parameters above are the same as these used in the experiments reported in [10]. This allows to compare AE with the methods analyzed in [10].

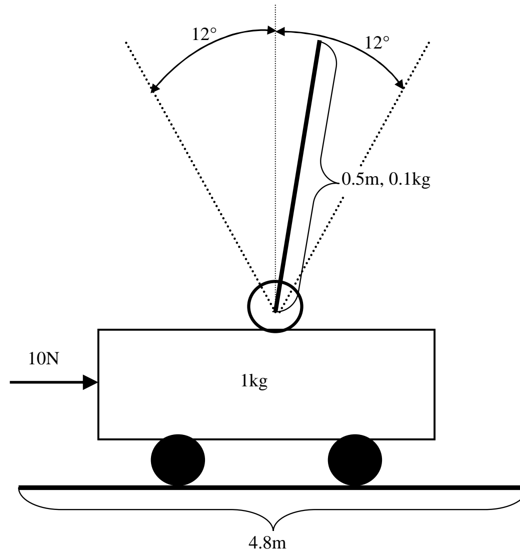

Fig. 7. The card-and-pole system

4 COMPARED SOLUTIONS

Generally, AE was compared to six methods, i.e. Q-learning, 1-layer AHC, 2-layer AHC, GENITOR, SANE and Canonical GA. The first five methods were tested by Moriarty within the confines of the experiments reported in [10]. The last six method was tested together with AE during current experiments. A short description of all the compared methods is presented in the following part of the paper.

4.1 Q-Learning

Many different approaches for solving the RL problem have been proposed so far. One of them is the temporal difference (TD) method [1, 2, 8]. In TD we deal with the so-called value function which is the expected value of a discounted sum of future

rewards that the agent can receive from an environment given that it starts in time t from a state s and uses policy π. The value function is defined as follows [2]:

$$V^{\pi}(s) = E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^{k}r_{t+k+1} \mid s_t = s\right\} \tag{3}$$

where $0 \leq \gamma \leq 1$ is a discount factor which determines whether immediate rewards are more important for the agent (for $\gamma = 0$ the agent behaves so as to maximize only immediate reward) or whether it is more interested in maximizing rewards received over longer period of time (for $\gamma = 1$ the agent behaves so as to maximize infinite sum of rewards) . The main task of TD is to estimate the optimal value function defined as follows [2]:

$$V^{*}(s) = \max_{\pi} V^{\pi}(s) \tag{4}$$

for all $s \in S$. To do this the following update rule is applied [2]:

$$V^{\pi}_{k+1}(s_t) \leftarrow V^{\pi}_{k}(s_t) + \alpha\left\lfloor_{r+1} + \gamma V^{\pi}_{k}(s_{t+1}) - V^{p}_{k}i(s_t)\right\rfloor \tag{5}$$

where α is a learning rate. The most known implementation of TD is the algorithm called Q-learning [1, 2, 8]. Instead of estimating the optimal value function, it strives to estimate optimal Q-function. Both Q-function and optimal Q-function are defined below [2]:

$$Q^{\pi}(s,a) = E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^{k}r_{t+k+1} \mid s_t = s, a_t = a\right\} \tag{6}$$

$$Q^{*}(s,a) = \max_{\pi} Q^{\pi}(s,a), \text{ for all } s \in S \text{ and } a \in A. \tag{7}$$

In Q-learning, initially random Q-function is updated in each step of the agent. Each update refines Q-function and makes it closer and closer to the optimal Q-function. The algorithm stops working once Q-function is appropriately close to the optimal one. When we know the optimal Q-function the following optimal policy is used [2]:

$$\pi^{*}(s) = \arg\max_{a \in A} Q^{*}(s,a) \tag{8}$$

i.e. in each step the action is selected which returns the maximal value of the optimal Q-function.

In experiments reported in [10] Q-function was in the form of a lookup table. However, in order for the lookup table to be able to represent Q-function, S and A have to be discrete. In the inverted pendulum problem described in Section 2 the set of actions is discrete. Two actions are available to the agent in each state of the card-and-pole system, i.e. to move left or right (the magnitude of the force is constant, 10 N). The problem is with the set S. Each parameter describing a state of the card-and-pole system, i.e. position of the card, velocity of the card, angle of the pole, and angular velocity of the pole, are continuous. To make S discrete the whole state space was divided into 162 non-overlapping regions, called "boxes".

Each box represented a single state of the card-and-pole system. This allowed to represent Q-function in the form of the lookup table including 2 rows (2 actions) and 162 columns (162 states).

4.2 AHC (Adaptive Heuristic Critic)

Moriarty used two implementations of AHC, i.e. 1-layer AHC and 2-layer AHC. Both AHC implementations consisted of two ANNs, i.e. value ANN (ANN critic) and action ANN. Regardless of the implementation, the task of the value ANN was to approximate the optimal value function by means of (5), whereas the task of the action ANN was to make decisions. To learn to make optimal decisions the action ANN used the information received from the value ANN.

In the case of 1-layer AHC both ANNs were single layer ANNs, i.e. they consisted of input neurons and a single output neuron. The number of input neurons in both 1-layer AHC ANNs was equal to 162. As before, the whole original input space was divided into 162 non-overlapping "boxes". Each box corresponded to a single input neuron. Activation of the ith neuron meant that the card-and pole system was in the ith box.

ANNs in 2-layer AHC implementation consisted of 5 input neurons (4 input variables describing a state of the card-and-pole system and one bias unit set at 0.5), 5 hidden neurons and an output neuron. Each input neuron was connected to each hidden neuron and to the output neuron. The signal from the output neuron in both AHC implementations was interpreted as the probability of selecting an action.

4.3 GENITOR

GENITOR [21, 22] is a genetic algorithm with a steady state replacement strategy and with a mechanism of adapting a mutation rate to the degree of diversity of individuals in a population. It was used by Moriarty to encode ANNs of the same architecture as ANNs in 2-layer AHC implementation. In the case of GENITOR each ANN was encoded in the form of a single binary chromosome.

4.4 SANE

SANE [10] assumes that information necessary to create ANN is included in two types of individuals, i.e. in blueprints and in encoded neurons. Both types of individuals evolve in separate populations. The task of the blueprints is to record the most effective combinations of neurons. Each blueprint defines a single ANN, i.e. it specifies a set of neurons that cooperate well together. All the encoded neurons evolving in the neuron population represent hidden neurons of two-layered feed-forward ANN. Each individual from this population includes information about neuron's connections with input and output neurons and strength of every connection.

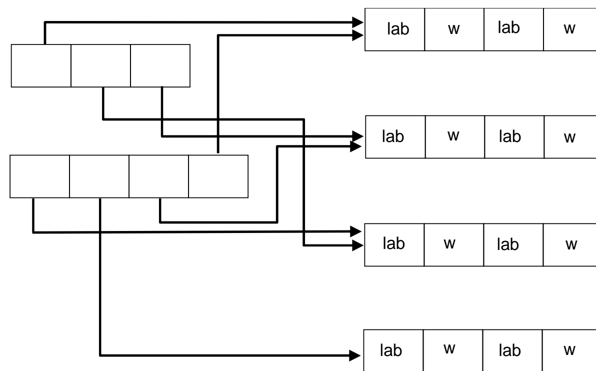

Fig. 8. Example assignment of neurons to blueprints (SANE) [10]

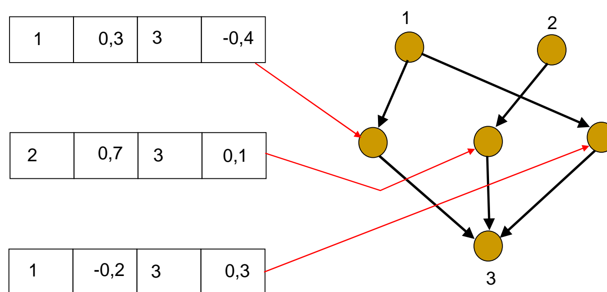

Fig. 9. Creating example ANN based on three encoded neurons (SANE) [10]

Moriarty used SANE to encode 2-layer ANNs consisting of 5 input neurons (4 input variables and bias), 8 hidden neurons and two output neurons. In this case output signal from ANN was not interpreted as the probability of selecting an action. This time, to select an action output signals from output neurons were compared. A neuron whose signal was stronger decided about an action that had to be taken.

4.5 Canonical GA

CGA is a classical GA. It was used to create fully connected feed-forward ANNs of different number of hidden neurons. To obtain ANNs with different number of hidden neurons chromosomes encoding ANNs (each chromosome encoded one ANN) were of varied length. Each chromosome encoded weights of interneuron connections and additionally parameters of neurons. The number of input and output neurons in ANNs created by means of Canonical GA was identical as in SANE, i.e. 5 input neurons and 2 output neurons.

4.6 CoEvolutionary GA

CEGA is a variant of CGA. In CGA ANNs evolve in a single population. Each chromosome from the population represents a single ANN. In CEGA we deal with a different situation. Each ANN created by means of CEGA evolves in three different populations. The populations include chromosomes which define different elements of ANN (Figure 10). The evolution in each population proceeds according to CGA. In the experiments with CEGA, all ANNs had 5 inputs, 2 outputs and 3 hidden neurons. Chromosomes in each population encoded weights of interneuron connections and additionally parameters of neurons.

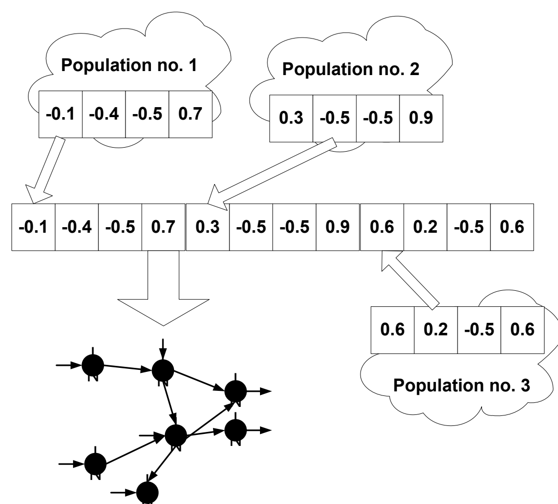

Fig. 10. Evolution of ANNs in CEGA

4.7 The Version of AE Compared to the Methods Above

In AE the only parameter that can be determined in advance is the number of input and output neurons. For the inverted pendulum problem, it was 5 input neurons and 2 output neurons. The interpretation of input and output signals was identical as in SANE. In AE both the number of hidden neurons and connectivity within ANNs created is fixed through the evolution. Thus, ANNs generated during the tests had different architecture.

In the experiments AEPs consisting of 2 operations and a sequence of data were used. The data were of varied length. The minimum length of data amounted to 10. Their maximum length was set to 60. AEPs created in the experiments used two operations, i.e. CHGM and CHGFF. CHGM changes a block of elements in NDM. The elements are updated in columns, in turn, one after another, starting

```
0 0 -0.111111 -0.47619 -0.0793651 -0.206349 -0.111111 0 0 0
0 0 0.920635 -0.269841 -0.460317 0.857143 0.650794 -0.761905 0 0
0 0 -0.47619 -0.492063 0.730159 0.84127 -0.0952381 0 0 0
0 0 -0.746032 0.619048 0.349206 0.52381 -0.825397 0 0 0
0 0 -0.793651 0.380952 0.492063 -0.619048 -0.539683 0 0 0
0 0 0.698413 0.365079 0.68254 -0.015873 -0.238095 0 0 0
0 0 -1 -0.952381 -0.777778 0.015873 -0.936508 0 0 0
```

a)

```
0 0.68254 0.142857 0.730159 0.809524 -0.714286 0.047619 -0.84127 0.269841 0.539683 0.0952381
0 0 0.174603 -0.571429 -0.47619 -0.380952 -0.0793651 -0.777778 -0.603175 0.619048 -0.206349
0 0 0 0.698413 0.111111 -0.142857 0.142857 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 0.714286 -0.222222 -0.126984
0 0 0 0 0 0 0 0.634921 0.555556 0.111111 0.31746
0 0 0 0 0 0 0 -0.619048 0.698413 -0.920635
```

b)

Fig. 11. Example use of a) CHGM and b) CHGFF

from an element pointed by parameters of the operation. New values for elements are located in the data part of AEP. CHGFF updates the part of NDM above the diagonal, i.e. the part defining feed-forward ANNs. As before, new values for the elements of NDM are located in the data part of AEP. Examples of using both operations are presented in Figure 11.

The remaining parameters for AE and for all the methods compared to AE are presented in Tables 1 and 2.

	1-layer AHC	2-layer AHC	Q-learning
Action Learning Rate (α)	1.0	1.0	
Critic Learning Rate (β)	0.5	0.2	0.2
TD Discount Factor (γ)	0.95	0.9	0.95
Decay Rate (λ)	0.9	0	

Table 1. Parameters for RL methods (all the parameters taken from [10])

5 EXPERIMENTAL RESULTS

In the experiments, the learning speed of AE and Canonical GA was tested. Both evolutionary methods were used to create ANNs whose task was to solve the inverted pendulum problem. Fifty ANNs were generated for each method. Conditions of the experiments were the same as in the tests reported in [10]. Each ANN was evaluated one time, i.e. a single balance attempt was performed for each ANN. The fitness for each ANN was determined by the number of steps the pole remained balanced. When the pole remained balanced for 120 000 time steps the test was interrupted. All ANNs created in the experiments had 5 inputs and 2 outputs. Each

	GENITOR	SANE	Canonical GA	AE
Population Size	100	100,50	100	60, 20, 20 (totally 100)
Chromosome Length	35 (floats)	120 (bits)	maximum 75 (floats)	maximum 60 floats (data), 40 bits (oper)
Mutation Probability	Adaptive	0.001	0.01	0.005 (data), 0.05 (oper)
Crossover Probability	not included in [10]		0.7	0.7 (oper and data)
Size of Tournament (tournament selection)			2	1 (data), 2 (oper)

Table 2. Parameters for the evolutionary methods (parameters of GENITOR and SANE taken from [10])

output was connected with a single action. Input values for each tested ANN were selected at random from the following ranges σ–$(-2.4, 2.4)$; $\dot{\sigma}$–$(-1.5, 1.5)$; θ–$(-12, 12)$; $\dot{\theta}$–$(-60, 60)$ and then were normalized to the range $\langle 0, 1\rangle$. The results of the experiments are presented in Table 3. Each cell in the table includes the number of ANNs created up to the point when a successful ANN was formed (in the case of RL methods each cell includes the number of balance attempts necessary to generate successful ANN). A successful ANN is the ANN which could balance the pole for 120 000 time steps. In the table, for the comparison purposes, results of the tests reported in [10] are also included.

Table 3 shows that on the average AE needs more ANN evaluations to generate successful ANN than SANE, Canonical GA and 1-layer AHC. In the case of AHC better performance can result from the fact that both AHC ANNs work in simpler, discrete, prepared in advance input space. With regard to SANE and Canonical GA, it seems that the main reason why they are better than AE is that they encode ANNs in a direct way. In both methods above, chromosomes are collections of parameters of their ANNs. Each gene in a chromosome is a value of a concrete parameter of ANN. In AE we deal with a different situation. Chromosomes in AE are elements of simple programs, i.e. AEPs. Genes in operations are not connected with specific parameters of ANNs. Combining the same operation with different data can cause a different effect in NDM and in consequence in ANN. The same genes in data chromosomes can determine values of different parameters of ANNs. It depends on operations which use data. Generally, it is necessary to state that such problems as the inverted pendulum problem, i.e. the problems that can be solved by means of simple ANNs, rather do not require indirect encodings like AE. Indirect methods can be useful to create larger neural architectures. In the case of larger ANNs direct encodings require long chromosomes to be used (each parameter has to

be encoded separately in a chromosome) whereas numerous experiments conducted in the field of evolution showed that applying long chromosomes may hamper or even prevent generating optimal solutions. An additional advantageous feature of AE in relation to direct methods presented in the paper is its capability to encode not only parameters of ANNs but also other aspects of functioning of ANNs. The example is the possibility to encode a learning process of ANN. AEPs can include not only operations updating NDM but also operations which can organize learning of ANN [19]. Another example is control of the process of growth of ANN. ANNs, like humans, can grow from childhood to maturity [4, 7]. In the meantime they can undergo the procedure of learning. All the process mentioned can be organized by AEPs.

	Mean	Best	Worst	SD
1-layer AHC	430	80	7 373	1 071
2-layer AHC	12 513	3 458	45 922	9 338
Q-learning	2 402	426	10 056	1 903
GENITOR	2 578	415	12 964	2 092
SANE	900	101	2 502	598
Canonical GA	1 150	115	3 356	918
AE	1 421	54	7 130	1 398

Table 3. Comparison of the learning speed (results of the first five methods taken from [10])

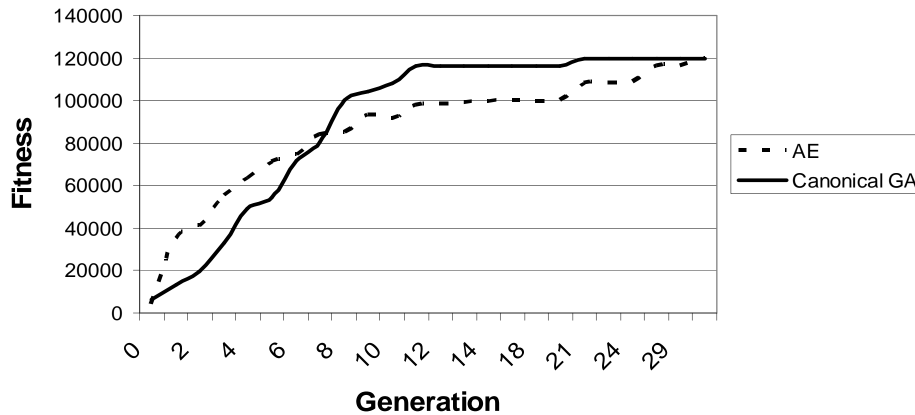

Fig. 12. Learning speed of AE and CGA

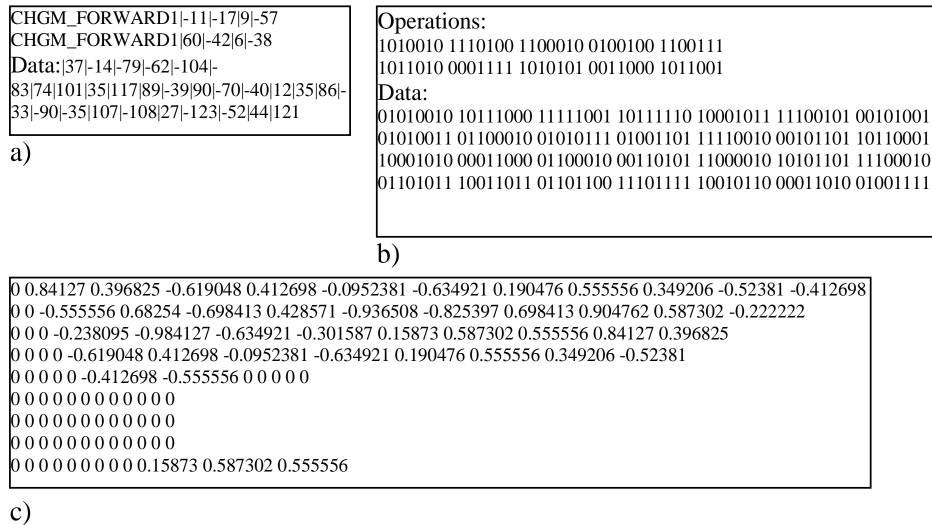

Fig. 13. a) Example AEP which created sucessful ANN b) encoded form of AEP presented in point a), c) NDM generated by AEP presented in points a) and b)

6 SUMMARY

The paper compares AE with RL and evolutionary RL methods on the inverted pendulum problem. To compare the methods the results of the experiments reported by Moriarty in [10] and the results of tests performed by the author were used. The experiments showed that AE is effective tool to solve the inverted pendulum problem. Even though AE is indirect method which seems to be rather suited to solve more complex problems it achieved only somewhat worse results than the best methods presented in the paper. To test AE in more difficult problem the experiments in the variant of the inverted pendulum problem with two poles installed on the card are planned. In the future experiments, to balance the poles only the information about pole angle and cart position will be used.

REFERENCES

[1] Baird III, L.: Reinforcement Learning Through Gradient Descent. Ph. D. thesis, Carnegie Mellon University, Pittsburgh, 1999.

[2] Cichosz, P.: Learning Systems. WNT, Warsaw, 2000.

[3] Butz, M. V.: Rule-Based Evolutionary Online Learning Systems: Learning Bounds, Classification, and Prediction. University of Illinois, IlliGAL Report No. 2004034, 2004.

[4] Elman, J. L.: Learning and Development in Neural Networks: The Importance of Starting Small. Cognition 48, 1993, pp. 71–99.

[5] Goldberg, D. E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison Wesley, Reading, Massachusetts, 1989.

[6] Holland, J. H.: Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, Michigan, 1975.

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

[8] Littman, M. L.—Szepesvari, C.: A Generalized Reinforcement-Learning Model: Convergence and Applications. In Proceedings of the Thirteenth International Conference on Machine Learning, pp. 310–318, 1996.

[9] Miller, G. F.—Todd, P. M.—Hegde, S. U.: Designing Neural Networks Using Genetic Algorithms. Proceedings of the Third International Conference on Genetic Algorithms 1989, pp. 379–384.

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

[11] Potter, M.: 1997. The Design and Analysis of a Computational Model of Cooperative Co-evolution. Ph. D. thesis, George Mason University, Fairfax, Virginia, 1997.

[12] Potter, M.—de Jong, K. A.: Evolving Neural Networks With Collaborative Species. In T. I. Oren, L. G. Birta (Eds.), Proceedings of the 1995 Summer Computer Simulation Conference, pp. 340–345. The Society of Computer Simulation, 1995.

[13] Potter, M. A.—de Jong, K. A.: A Cooperative Co-Evolutionary Approach to Function Optimization. The Third Parallel Problem Solving From Nature, Jerusalem, Israel, Springer Verlag, 1994, pp. 249–257.

[14] Potter, M. A.—de Jong, K. A.: Cooperative Co-Evolution: An Architecture for Evolving Co-Adapted Subcomponents. Evolutionary Computation, Vol. 8, 2000, No. 1, pp. 1–29.

[15] Praczyk, T.: Evolving Co-Adapted Subcomponents in Assembler Encoding. International Journal of Applied Mathematics and Computer Science, Vol. 17, 2007, No. 4.

[16] Praczyk, T.: Procedure Application in Assembler Encoding. Archives of Control Science, Vol. 17 (LIII), 2007, No. 1, pp. 71–91.

[17] Praczyk, T.: Adaptation of Symbiotic Adaptive Neuro-Evolution in Assembler Encoding. Theoretical and Applied Informatics, Vol. 20, 2008, No. 1, pp. 49–68.

[18] Praczyk, T.: Concepts of Learning in Assembler Encoding. Archives of Control Science, Vol. 18 (LIV), 2008, No. 3, pp. 323–337.

[19] Praczyk, T.: Modular Neural Networks in Assembler Encoding. Computational Methods in Science and Technology, Vol. 14, No. 1, pp. 27–38.

[20] Whitley, D.—Kauth, J.: GENITOR: A Different Genetic Algorithm. In Proceedings of The Rocky Mountain Conference on Artificial Intelligence, 1988, pp. 118–130.

[21] Whitley, D.: The GENITOR Algorithm and Selective Pressure. In Proceedings of the Third International Conference on Genetic Algorithms, 1989, pp. 116–121.

Tomasz Praczyk is a senior lecturer at the Institute of Naval Weapons of Polish Naval Academy in Gdynia. He received his M. Sc. degree in computer science in 1996. In 2001, he received his Ph. D. degree; with thesis focused on using artificial neural networks to identify ships. His research interest is in neuro-evolution, artificial immune systems, and reinforcement learning.