CONTINUAL EVOLUTION ALGORITHM FOR BUILDING OF ANN-BASED MODELS

Zdeněk Buk, Miroslav Šnorek

Czech Technical University in Prague, Faculty of Electrical Engineering,
Department of Computer Science and Engineering
Karlovo náměstí 13, 121 35 Prague 2, Czech Republic

bukz1@fel.cvut.cz (Zdeněk Buk)

Abstract

The Continual Evolution Algorithm (CEA) for building of models is presented in this paper. We chose an artificial neural networks (ANN) based models in our applications to show properties of CEA algorithm. During CEA evolution process a continual (in time) gradient learning algorithm is combined with a classical genetic (evolutionary) approach. Thus in this application a structure of models is constructed separately from particular parameters optimization in such models (e.g. weights in neural networks). These two optimizations are running at the same time but using different methods. As a platform for our experiments the universal neural network topology implementation based on the fully recurrent neural network has been chosen. This implementation allows the evolution algorithm to create any network structure with no limitations for a usage of gradient real time recurrent learning algorithm. An advantage of using evolutionary algorithms for neural network construction is in finding its optimal structure (number of neurons and connections among them). Splitting the construction process into structure finding part and the particular weight values setting (finding) has an advantage in reduction of the problem dimension. Number of reproduction operation calls is reduced and a part of optimization process is done separately. Results of these two parts are then combined before the next reproduction operation is needed. Individuals in our algorithm contain an age parameter, so the CEA allows for the number of gradient based algorithm steps for the individual quality assignment. The CEA is a universal optimization algorithm with no limitations for neural network construction and evolution. Neural networks created using this algorithm can be used for example in classification, prediction, etc. In this paper we will focus mainly on benchmark tasks showing a function and principles of the novel evolution algorithm in relations to other methods, pure gradient learning algorithm and differential evolution method.

Keywords: Continual Evolution Algorithm, Genetic Algorithm, Neural Networks.

Presenting Author's Biography

Zdeněk Buk works as a postgraduate student and researcher at the Department of Computer Science and Engineering, FEE-CTU. He got his master degree in Electronics and Computer Science & Engineering from FEE-CTU in Prague in 2005. He joined the CIG – Computational Intelligence Group (former known as NCG – Neural Computing Group) in 2002. His research focuses on methods of computational and artificial intelligence, mainly on artificial neural networks and evolutionary techniques.

1 Introduction

Continual evolution algorithm is a novel method based on standard genetic algorithm. It has been partially presented in our previous work, e.g. [1], or [2]. In this paper we have focused mainly on application of the algorithm. The goal was to prove some of its theoretical expected properties. As an application for our algorithm we have chosen the problem of building the models for real systems simulation. We use models based on artificial neural networks (ANN) that are constructed and adapted using our CEA algorithm.

Section 2 is dedicated to the main contribution of this work, which is the continual evolution algorithm (CEA) detailed description.

In section 3 we shortly describe other methods and algorithms used in this work. Brief description of simple genetic algorithm and differential evolution algorithm is given. Fully recurrent neural network that our implementation is based on, and gradient-based real time recurrent neural learning algorithm will be also described.

Main application of the CEA algorithm to neural network construction and adaptation problem is described in section 4. We show particular encoding of individuals (representing the neural networks) in CEA, whole process of evolution, and interesting part of implementation details.

Section 5 is dedicated to description of experiments we have performed to test the implementation, theoretical properties of our CEA algorithm, and selected algorithms comparison. Selected results of experiments are presented in section 7.

2 Continual Evolution Algorithm (CEA)

As well as standard genetic algorithm (SGA) also the CEA is fundamentally inspired by nature and it is a part of group of evolutionary algorithms. It combines genetic operators (representing the evolutionary part of the algorithm) with a gradient optimization method. Evolution in the CEA runs in two relatively independent processes - genetic based process and time-dependent gradient-based process. This two-dimensional evolution is illustrated in figure 2. The main idea of this approach is to separate the evolution of a structure and behavior (parameters) of individuals. When applied to a neural network construction we can imagine the structure as a topology of network and behavior as a particular weight values setting in such network.

The main core of CEA is the SGA extended by new parameters and techniques. Description of these parameters – data structures used for individuals' encoding – is given in next subsections as well as detailed description of evolution control mechanism based on probability functions.

Here are some basic properties, principles, and parameters used in CEA:

- variable size of population,

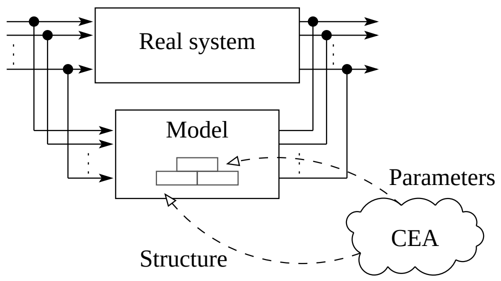


Fig. 1 Schematic diagram of model creation.

- separated encoding of structure and parameters of models,
- age of each individual,
- sequential replacement of individuals,
- original individuals (parents) are kept in population and optimized together with new individuals (offspring),
- evolution of individuals in two dimensions – inter-generation evolution and continual adaptation in time dimension (using gradient algorithm),
- probability based control of whole evolutionary process (depending on size of population, quality of individual, and its age).

Note: term "generation" here is used only for discrimination of parents and offspring. It is not the generation as it is defined in SGA. An example of population evolution and sequential replacement of individuals is shown in figure 3.

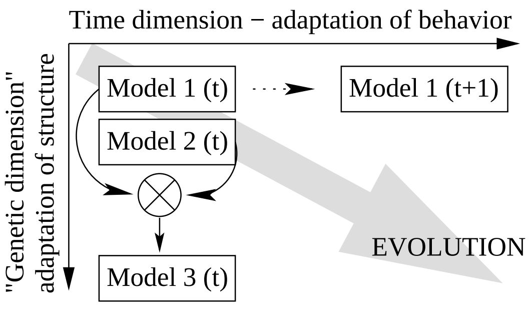

Time dimension − adaptation of behavior
"Genetic dimension" adaptation of structure
Model 1 (t)
Model 1 (t+1)
Model 2 (t)
Model 3 (t)
EVOLUTION

Fig. 2 Two-dimensional evolution in CEA.

The basic principles of CEA are the same as in SGA or genetic algorithms in general. Algorithm works with sets of individuals and there is also a reproduction process with genetic operators as crossover and mutation operators. The main differences are in methods how we are working with the population - how the new generation of individuals is being created, how the individuals are encoded and how they are being modified within the single generation.

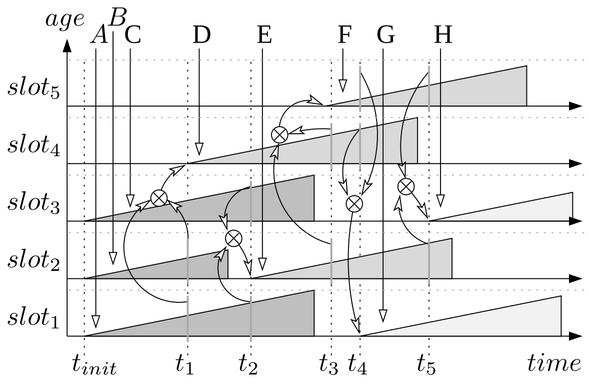


Fig. 3 Visualization of a CEA population example. This example shows the evolution of population with maximal size of five individuals, represented by five slots in the figure. The population is initialized with three individuals at time t_0 (two slots are empty in that moment). The vertical axis represents the age of each individual, maximal age is represented by horizontal dashed gray lines. Here can be seen that only some individuals reach this maximal age - it depends on some other factors such as probability, population size, and fitness value. At time t_1 the new individual (marked as D) is created as the offspring of the (parents) individuals A and B. The next reproduction processes come at t_2, t_3, t_4, and t_5. It is clear that the size of population (number of individuals) varies through time.

2.1 General Structures

This subsection describes the basic data structures used in CEA. Genetic algorithms in general work with some encoding of individuals – each individual represents one solution. In CEA the floating point vector is used for encoding of individuals. This encoding vector is additionally divided into logical parts, representing the structure and behavior of individual – topology and weights setting of neural network represented by the individual. An individual in CEA the individual is represented by the following vector:

$$\bar{x_i} = (a_i, \bar{p_i}, \bar{s_i}, \bar{b_i}), \tag{1}$$

where a_i is the *age* of i-th individual, $\bar{p_i}$ is the initialization *parametric vector* (called instinct), $\bar{s_i}$ is the *structural* parameter and $\bar{b_i}$ is *behavioral vector* of i-th individual, which contains actual set of working parameters of the individual (at the beginning of evolution it is created as a copy of the $\bar{p}$ vector).

The parameters of i-th individual $\bar{x_i}$ are described as follows:

$$\begin{aligned} \bar{p_i} &= (p_{i,1}, p_{i,2}, \ldots, p_{i,u}), \\ \bar{s_i} &= (s_{i,1}, s_{i,2}, \ldots, s_{i,v}), \\ \bar{b_i} &= (b_{i,1}, b_{i,2}, \ldots, b_{i,u}), \end{aligned} \tag{2}$$

where u is the dimension of time (age) dependent parameters vector ($\bar{p}$ and $\bar{b}$) and v is the dimension of the structural parameters vector.

2.2 Probability Functions

The CEA is controlled by several auxiliary parameters that are computed for each individual in population. These parameters are used in the reproduction cycle. The first parameter is the *reproduction probability* which describes the probability, that the i-th individual of age a_i and quality F given by the fitness function, will be used for reproduction operation and that they will produce some new individual (offspring) to the next generation. The reproduction probability is defined as:

$$RP^*(\bar{x_i}) = RP^*(a_i, F(\bar{x_i})), \tag{3}$$

where x_i is the i-th individual that we are computing the probability for, a_i is the age of this individual and function F represents the fitness function – so $F(\bar{x_i})$ represents the fitness value (quality) of the i-th individual. Typical behavior of the reproduction probability can be seen in figure 4(a).

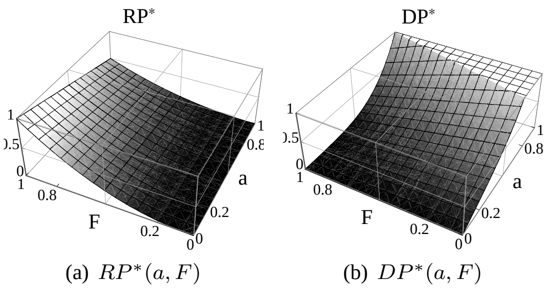


Fig. 4 Example of the raw probability functions.

Parameter *death probability* represents the property that each individual has some maximal age that they can live for. The probability of survival of each individual depends on the quality and actual age of this individual. Here is an example how the death probability is defined:

$$DP^*(\bar{x_i}) = DP^*(a_i, F(\bar{x_i})), \tag{4}$$

where x_i is the i-th individual that we are computing the probability for, a_i is the age of this individual and function F represents the fitness function – so $F(\bar{x_i})$ represents the fitness value (quality) of the i-th individual. Typical behavior of the death probability can be seen in figure 4(b).

All values signed by $*$ are so called *raw* values. So DP^* is the *raw death probability* and RP^* is the *raw reproduction probability*. The final values DP and RP, which the CEA works with, are computed from the raw values using the *balancing functions*. These functions represent the influence of the size of the population to this size itself – the bigger population will grow slowly (to some limit value, where no new individual will be born) and the smaller population will grow faster (for smaller populations the death probability is reduced and goes to zero – see examples below).

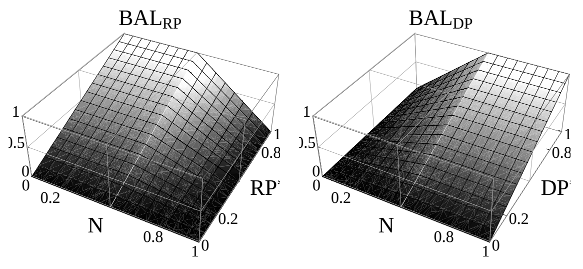
Final probabilities computation:

$$DP(\bar{x_i}) = BAL_{DP}(N, DP^*(\bar{x_i})), \tag{5}$$

(a) $RP^*(a, F)$ (b) $DP^*(a, F)$

$$RP(\bar{x_i}) = BAL_{RP}(N, RP^*(\bar{x_i})), \quad (6)$$

where N is the size of the actual population and the BAL_{DP} and BAL_{RP} are the general balancing functions. Examples of the balancing function is shown in figure 5.



(a) $BAL_{RP}(N, RP^*)$ (b) $BAL_{DP}(N, DP^*)$

Fig. 5 Example of the balancing functions.

2.3 Evolution Control

- $\mathbb{X} = (\bar{x_1}, \bar{x_2}, \ldots, \bar{x_N})$ – Population of individuals
- N – Size of the population
- Δt – Time step for updating the age parameter
- α – Training constant
- $CROSS_p/CROSS_s$ – cross operators
- $RP(\bar{x_i})$ – reproduction probability function
- $DP(\bar{x_i})$ – death probability function
- $F(\bar{x_i})$ – fitness function/fitness value of individual

Algorithm 1 (The General CE Algorithm)

```
1. Initialization
2. Repeat until stop condition
   2.1. Evaluate all individuals
   2.2. Reproduction of the individuals with respect
        to the $RP(\bar{x_i})$ value
   2.3. Elimination of the individuals with respect
        to the $DP(\bar{x_i})$ value
   2.4. Adaptation of the parametrical vector of
        each individual
   2.5. Update all working parameters and age
        parameter of individuals
3. The result processing
```

Algorithm 2 (CEA – 1. Initialization)

```
let $N$ = initial population size
for $i = 1$ to $N$
do
    let $a_i = 0$
    generate random $\bar{p_i} = (p_{i,1}, p_{i,2}, \ldots, p_{i,u})$
    generate random $\bar{s_i} = (s_{i,1}, s_{i,2}, \ldots, s_{i,v})$
    let $\bar{b_i} = \bar{p_i}$
    insert individual $\bar{x_i} = (a_i, \bar{p_i}, \bar{s_i}, \bar{p_i})$ to $\mathbb{X}$
done;
```

Algorithm 3 (CEA – 2.1. Evaluate Individuals)

```
for $i = 1$ to $N$
do   (* calculate the probability parameters *)
    let $DP_i = DP(\bar{x_i}) = BAL_{DP}(N, DP^*(\bar{x_i}))$
    let $RP_i = RP(\bar{x_i}) = BAL_{RP}(N, RP^*(\bar{x_i}))$
done;
```

Algorithm 4 (CEA – 2.2. Reproduction Process)

```
let $M = 0$
for $i = 1$ to $N$
do
    generate random $p$ from interval $< 0; 1 >$
    if $p < RP_i$ then
      let $M = M + 1$
      (* select other individual *)
      define $p_{sel}(n) = \begin{cases} \frac{\sum_{j=1}^{n} RP_i}{\sum_{j=1}^{N} RP_i}, & 1 \leq n \leq N \\ 0, & n = 0 \\ 1, & n = N + 1 \end{cases}$
      generate random $q$ from interval $< 0; 1 >$
      find $j \neq i; j \in < 1, N >$
            $\Rightarrow \; p_{sel}(j-1) < q < p_{sel}(j+1)$
      (* produce new individual *)
      $\bar{s_M} = CROSS_s(\bar{s_i}, \bar{s_j})$
      $\bar{p_M} = CROSS_p(\bar{p_i}, \bar{b_i}, \bar{p_j}, \bar{b_j})$
      $\bar{b_M} = \bar{p_M}$
    fi
done;
let $N = N + M$
```

Algorithm 5 (CEA – 2.3. Elimination Process)

```
for $i = 1$ to $N$
do
    generate random $p$ from interval $< 0; 1 >$
    if $p < DP_i$ then
      mark individual $\bar{x_i}$ as "removed"
    fi
done;
```

Algorithm 6 (CEA – 2.4. Time Dependent Adaptation)

```
for $i = 1$ to $N$
do
    if $\bar{x_i}$ is not marked as "removed" then
      use gradient or other optimization technique
      $b_{i,j} = b_{i,j} - \alpha \frac{\partial F}{\partial b_{i,j}}$
    fi
done;
```

Algorithm 7 (CEA – 2.5. Update Parameters)

```
delete all individuals marked as "removed" from $\mathbb{X}$
let $N = |\mathbb{X}|$
for $i = 1$ to $N$
do
    let $a_i = a_i + \Delta t$
done;
```

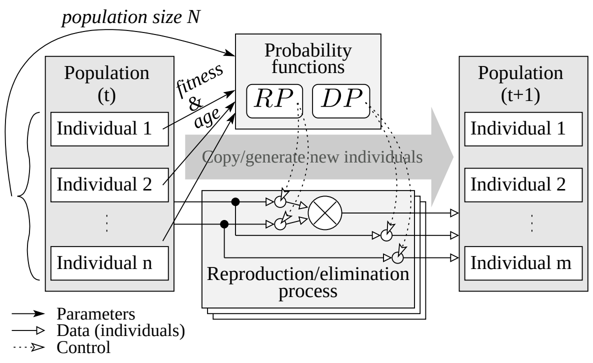


Fig. 6 Diagram of population evolution in the CEA. New generation of individuals is created in process controlled by probability functions. Number (N) of individuals in original population (t) is in general not equal to the number of individual in new population ($t + 1$).

3 Other Methods and Algorithms Used

In this section some other algorithms and methods used in our work are shortly described.

3.1 Standard Genetic Algorithm (SGA)

The standard genetic algorithm (SGA) works with a population of individuals, where each individual represent one solution of problem being solved. It is an iterative process using crossover and mutation operators to modify/generate individuals.

The individuals that will take a part in the reproduction process are selected with respect of their fitness value, which represents quality of the individual (solution). Schematic diagram describing the algorithm is shown in figure 7. All this approach is based on theory that by combination of two good individuals we get one even better individual/solution. J. Holand in 1975 created the schema theory that prove, that this works and that the above-average schemas (each schema represents one ore more individuals) are multiplied exponentially in the population.

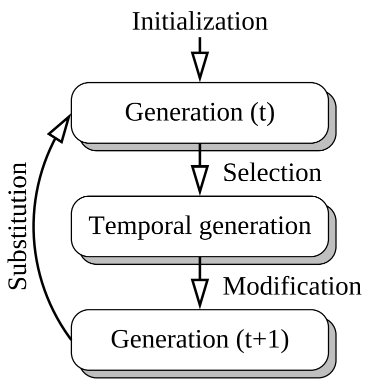


Fig. 7 The SGA flow diagram.

The *cross operator* combines the parts of two individuals and creates new offsprings that replaces the "parent" individuals in new population. For binary encoded individuals – the encoded vector describing the individual is called *chromosome* – the example of cross operator is shown in figure 8.

Cross point

Parent 1	1	0	0	1	0	1	1	1
Parent 2	0	1	0	0	1	1	0	0
Offspring 1	1	0	0	1	1	1	0	0
Offspring 2	0	1	0	0	0	1	1	1

Fig. 8 Example of single point cross operation.

Mutation point

Original individual	1	0	0	1	0	1	1	1
After mutation	1	0	0	1	0	1	0	1

Fig. 9 Example of mutation operation.

The *mutation operator* is based on random bit change in the chromosomes. An example of mutation is shown in figure 9. The reason of using the random mutation operator is to avoid the optimization to stuck in some local optimum. When so high probability of mutation is used the genetic algorithm become the random search. So it is important to set the value appropriate to the current problem/data.

When the genetic algorithm is used in combination with neural networks, each individual can represents the whole network – the topology, weight matrix, or combination of them [3].

More information and links to other literature can be found e.g. in [4, 5] or in technical report [2], where a survey of these nature inspired methods is given.

3.2 Differential Evolution (DE) Algorithm

The differential evolution algorithm is very similar to the genetic algorithm described above. The main difference is the sophisticated method of reproduction operation. This method has been created by Price and Storn in 1997.

In the reproduction process four individuals are used (in standard genetic algorithm only two individuals were used). As the first individual is selected step-by-step each individual in population, the remaining three individuals (r_1, r_2, r_3) are selected randomly (as three different individuals).

The mutation vector is calculated from the randomly selected individuals:

$$v_j = x^G_{r3,j} + (x^G_{r1,j} - x^G_{r2,j})F, \qquad (7)$$

where $x^G_{r,j}$ represents the j-th parameter of the individual r in current generation and F is the mutation constant.

Using the random number and cross threshold value (CR) the new "testing" individual is created:

$$x^{testing}_{i,j} = \begin{cases} x^G_{r3,j} + (x^G_{r1,j} - x^G_{r2,j})F, & \\ \qquad \text{if } rnd(0,1) \le CR & \\ x^G_{i,j} \qquad \text{else,} & \end{cases} \qquad (8)$$

where $x^{testing}_{i,j}$ represent the j-th parameter if i-th testing individual, $x^G_{i,j}$ represents the j-th parameter of i-th individual in current generation, CR is the cross threshold value, and $rnd(0,1)$ represents the random number from interval $< 0; 1 >$.

The quality/fitness value of the new testing individual is then compared with the fitness value of the original individual (the first one – "step-by-step" selected from the population). If the new individual is better than the original one, the new individual is inserted into new population:

$$X^{G+1}_i = \begin{cases} x^{testing} & f(X^G_i) \le f(x^{testing}), \\ x^G_i & \text{else,} \end{cases} \qquad (9)$$

where the X^{G+1}_i represent the i-th individual in new generation and $f(x)$ calculates the fitness value of individual x.

It has been shown that the differential evolution algorithm is good for optimizations in very complex state space with very isolated solutions [6, 7]. This algorithm is also useful for real value encoded individuals. For example whole weight matrix describing the neural network can be used.

3.3 Fully Recurrent Neural Network (FRNN)

Fully recurrent neural network (FRNN) is the most general structure of recurrent network, it is represented by a complete balanced graph, see fig.10.

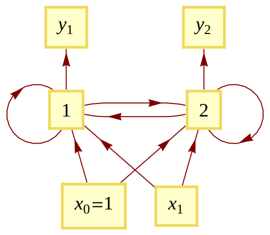


Fig. 10 The fully recurrent neural network with two neurons and two inputs. One of the inputs (x_0) is usually used for threshold value realization connecting to constant value equal to one, so this network has one external input (x_1).

3.4 Real time Recurrent Learning (RTRL)

Real time recurrent learning is a gradient based algorithm for neural networks training. It has been described in [8, 9].

We have network with n neurons and m external inputs. Potential of k-th neuron in network in time t is defined as:

$$s_k(t) = \sum_{l \in U \cup I} w_{kl} z_l(t), \qquad (10)$$

where U is the set of indices l for which $z_l(t)$ represents output value of neurons and I is the set of indices of external input of network. So $z_l(t)$ represents l-th input to neuron and w_{kl} is the corresponding weight.

$$z_k(t) = \begin{cases} x_k(t) & \text{for } k \in I \\ y_k(t) & \text{for } k \in U. \end{cases} \qquad (11)$$

Output of k-th neuron in next time step is defined as:

$$y_k(t+1) = f_k(s_k(t)), \qquad (12)$$

where f_k is activation function of neuron, and $s_k(t)$ is the inner potential of neuron in time t.

Now we can define the difference between expected j-th output and actual output value as:

$$e_k(t) = \begin{cases} d_k(t) - y_k(t) & \text{is } k \in T(t) \\ 0 & \text{else,} \end{cases} \qquad (13)$$

where $d_k(t)$ is expected and $y_k(t)$ actual output value of k-th neuron. $T(t)$ represents set of indices for which the expected value is defined.

Error of whole network in time t can be calculated as:

$$J(t) = \frac{1}{2} \sum_{k \in U} [e_k(t)]^2, \qquad (14)$$

where $e_k(t)$ is error of k-th neuron in time t.

If the calculation runs in time interval $< t_0, t_1 >$, the final error will be:

$$J_{total} = \sum_{t=t_0+1}^{t_1} J(t), \qquad (15)$$

where $J(t)$ is error in time t.

This error we are trying to minimize during the training process using modification of weight matrix **W**:

$$\Delta w_{ij} = \sum_{t=t_0+1}^{t_1} \Delta w_{ij}(t), \qquad (16)$$

where

$$\Delta w_{ij}(t) = -\alpha \frac{\partial J(t)}{\partial w_{ij}} \qquad (17)$$

and $\alpha > 0$ is the training parameter.

Using equation above we can finally compute for all $k \in U$, $i \in U$ and $j \in U \cup I$:

$$p_{ij}^{k}(t+1) = f_k'(s_k(t)) \left[\sum_{l \in U} w_{kl} p_{ij}^{l}(t) + \delta_{ij} z_j(t) \right] \tag{18}$$

where

$$f_k'(s_k(t)) = y_k(t+1)[1 - y_k(t+1)] \tag{19}$$

and

$$\delta_{ij} = \begin{cases} 0 & \text{for } i \neq j, \\ 1 & \text{for } i = j. \end{cases} \tag{20}$$

Because the initial state of network (in time t_0) is independent on weight values

$$\frac{\partial y_k(t_0)}{\partial w_{ij}} = 0, \tag{21}$$

holds the following initial condition:

$$p_{ij}^{k}(t_0) = 0. \tag{22}$$

Finally the differences in weight matrix can be calculated:

$$\Delta w_{ij}(t) = \alpha \sum_{k \in U} e_k(t) p_{ij}^{k}(t). \tag{23}$$

The RTRL algorithm is of course not the only algorithm for training the recurrent neural networks, but we chose it because of its universality. The algorithm described above represents the basic version of RTRL algorithm. More detailed description of RTRL and other gradient methods for training recurrent networks can be found in [8, 9].

4 Application

We chose a neural network based models construction problem as an application for our CEA evolution algorithm. Using this algorithm the structure of the model is being evolved with the goal of finding some optimal neural network topology for given problem. At the same time there is running a process of optimization of parameters of the model.

As a platform for our application we chose the implementation of fully recurrent neural network (FRNN) as a most universal neural network structure. For gradient learning part of CEA we chose a real time recurrent learning algorithm.

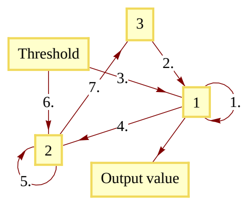
All experiments has been performed in *Mathematica* programming language and environment.

4.1 Individuals' Encoding

Individuals' encoding vector is divided into logical parts as it was shown in (1). Because of the topology of the FRNN (the fully connected graph), its structure can be simply encoded into matrix, where $1's$ represent connections between nodes (neurons) and $0's$ represent missing connection. Encoding of behavior of the

$$\begin{pmatrix} 1. & 0 & 2. & 3. \\ 4. & 5. & 0 & 6. \\ 0 & 7. & 0 & 0 \end{pmatrix}$$

(a) Weight matrix



(b) Corresponding network

Fig. 11 An example of representation of the network by weight matrix.

network – particular setting of weights – is encoded in the similar matrix (same dimensions) but with floating point values.

In CEA two matrices are used - 0/1 matrix representing the structure of network and real value matrix with weights values.

4.2 Evolution Process

An example of general evolution process in CEA is shown in figures 2 and 3. In this chapter some detailed information about evolution in our application of CEA to neural network construction problem will be shown. Reproduction operators and fitness function will be described here.

4.2.1 Fitness function

Fitness value (quality) of each individual is calculated as the negation of normalized error of the network (represented by the individual) on testing data. First the vectors describing the neural network need to be extracted from individual encoding, then it is necessary to convert the linear encoding to matrix form and finally this matrix can be imported into network model and simulated.

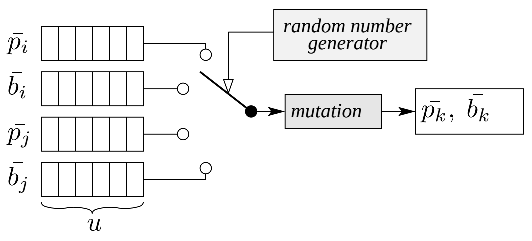


Fig. 12 Schematic diagram of crossover operation on the behavioral vectors. There are randomly selected parts from initial parameter vector and behavioral vector of both individuals (i and j). In combination with the mutation operator the parametrical and behavioral vectors of new individual are being constructed

4.2.2 Reproduction operator

Reproduction process works with two individuals. Each individual is represented by 3 vectors (and other parameters, which are not important in the process now) – initial parametric vector, structural vector, and behavioral vector. Reproduction operator works with 6 sets of parameters then. Figure 12 shows the process of parametric vector of new individual generation. Reproduction process for structural parameter works similarly – there are two "source" structural vectors ($\bar{s_i}$ and $\bar{s_j}$) and two auxiliary vectors – vector of zeros and vector of ones, these vectors represent mutation operation (these vectors are selected with lower probability then vectors $\bar{s_i}$ and $\bar{s_j}$.

4.3 Implementation

All algorithms and experiments have been implemented in Wolfram *Mathematica* environment. We have created the CEA.m library with universal CEA implementation. Example of population (which involves all the CEA parameters settings) initialization:

```
pop = CEAInitialize[
        initial_pop_size,
        min_pop_size,
        max_pop_size,
        age_increment,
        u, v, (* vectors dimensions *)
        fitness_ref,
        crossfunction_ref,
        gradf_ref,
        updatef_ref,
        {additional_parameters}];
```

First three parameters describe the size of population (number of individuals), `age_increment` represents the elementary time increment of age parameter of each individual - age parameter is normalized (0-1), so using this parameter the maximal age of individuals can be specified (0.1 represents maximal age of 10 steps, 0.01 represents maximal age of 100 steps, etc.). Using `u` and `v` parameters the dimensions of structural and behavioral vectors are specified. Parameters `*_ref` are references to following functions: fitness function, crossover operator, gradient learning algorithm, auxiliary function for individual parameters update (it is no necessary to be used). The last parameter represents the set of auxiliary parameters that could be used in some special cases (in our experiments we are using this parameter for passing the number of neurons in network to the fitness function – it is only an implementation detail).

When the initial population `pop` is created and initialized, we can run the evolution process. One step of the process is done using the following code:

```
newpop = CEAReproduction[pop];
```

In the CEA.m library there are also some other functions used mainly for debugging purposes (printing the individuals in population, etc.), but with `CEAInitialize[]` and `CEAReproduction` the whole evolution process can be performed.

5 Experiments

We have chosen the simple experiment "learn to oscillate" [8] to check our implementation and for comparison with other optimization algorithm. The neural network is trained to generate some defined periodical sequence. We have tried pure gradient method (real time recurrent learning), differential evolution algorithm and CEA algorithm.

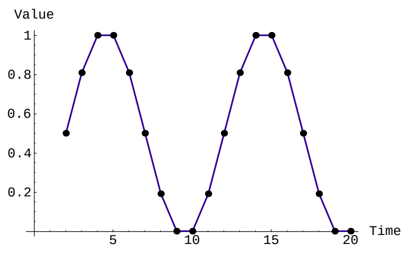


Fig. 13 An example of training set for learn to oscillate experiment. Two period of $sin(x)$ function sampled by 20 points has been chosen in one experiment.

Neural network in this experiment has no external inputs and it has one output value – trained to generate the given sequence.

We have tried many parameters setting of used algorithms to get an appropriate results. Performed experiments confirmed the theoretical properties of our CEA algorithm. For details see section 6.

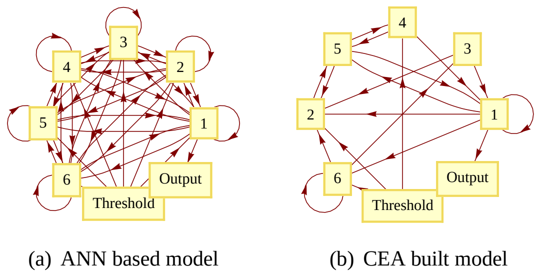


Fig. 14 An example of CEA result in ANN based model evolution experiment. Left figure shows the fully recurrent neural network with all connections – this is the most general structure that we are building our solution on. Right figure shows the result of evolution process – trained recurrent neural network. About 57% connection has been removed by evolution process. Output value of the network is taken from the neuron 1.

6 Discussion

We have chosen benchmark experiments based on "learn to oscillate" experiment described in [8]. Table 1 shows a result from experiment where FRNN with 6 neurons has been trained to $sin(x)$ function. We chose this number of neurons based on our previous experiences – it is enough and it can be shown the possible inter-neurons connections reduction in this network.

Because of variable population size and combination of genetic and gradient-based methods in CEA there was highly reduced the number fitness evaluation needed for

Tab. 1 Selected experiments results. Comparison of number of iterations needed in particular algorithms to train the network with comparable result (errors). DE-differential evolution, CEA-continual evolution algorithm, RTRL-real time recurrent learning. The table shows numbers for whole population and for the best individual for comparison. Shown values are approximate – they come from several runs of algorithms.

	DE	CEA	RTRL
Total for whole algorithm run.			
No. of fitness evaluations	25 000	480	×
No. of gradient steps	×	9 000	500
For best solution (one individual).			
No. of fitness evaluations	500	17	×
No. of gradient steps	×	340	500

Tab. 2 The table shows number of fitness function evaluations (FC), number of gradient algorithm steps performed (GC), average population size (PS), and quality of best solution found (Best). All these parameters were tested for different maximal population size and number of evolution steps. Number of gradient steps in each evolution iteration was set to 20.

	Maximal populaton size setting					
Evolution	**10**			**20**		
steps	**FC/GC**	**PS**	**Best**	**FC/GC**	**PS**	**Best**
5	57/940	10	0.62	91/1440	18	0.61
10	106/1900	10	0.69	183/3220	18	0.67
20	201/3740	10	0.94	365/6780	18	0.87
30	301/5700	10	0.95	526/9880	17	0.95
40	397/7500	10	0.95	674/12580	17	0.95
50	503/9580	10	0.96	865/16340	17	0.96
	Maximal populaton size setting					
Evolution	**50**			**100**		
steps	**FC/GC**	**PS**	**Best**	**FC/GC**	**PS**	**Best**
5	245/3900	49	0.64	477/7580	95	0.65
10	482/8600	48	0.73	958/17120	96	0.68
20	843/15640	42	0.93	1819/33980	91	0.93
30	1175/21960	39	0.96	2645/49700	88	0.96
40	1551/29120	39	0.96	3509/66360	88	0.96
50	1947/36620	39	0.96	4406/83780	88	0.97

evolving good-quality individual (network). Differential evolution algorithm uses constant population size, so number of fitness evaluations is also constant for particular number of individuals and number of generation setting. Gradient-based algorithm (RTRL) works with only one network and it needed less steps than the CEA. In CEA the average size of population was 24 although the maximal size was the same as in DE and it was set to 100.

The RTRL algorithm is not able to create optimal topology of network. It always uses the fully connected network. DE also optimized the whole weight matrix representing the fully recurrent neural network. CEA reduced some connection - as an example see following numbers: for 6 neurons there are 42 weight values ($6 \times (6 + 1)$, one additional input for threshold value), best network from one algorithm runs had 25 active connections, it means 17 connections was removed, which represents about 40% reduction.

Tab. 3 Results of testing CEA for different number of evolution iterations (10, 50, 100) and for different settings of number of gradient steps in each evolution iteration. We measured number of fitness function evaluations (FC - Fitness Count), population size - PS (average, minimal, maximal) and quality of best solution found (Best).

Grad	**FC**	**PS**	**Best**
10 evolution steps			
1	249	(25, 10, 42)	0.607035
5	220	(22, 10, 37)	0.611232
10	282	(28, 10, 45)	0.617294
50 evolution steps			
1	2063	(41, 10, 50)	0.608738
5	2038	(41, 10, 50)	0.673340
10	2150	(43, 10, 50)	0.903866
100 evolution steps			
1	4327	(43, 10, 51)	0.612297
5	4276	(43, 10, 50)	0.884965
10	4478	(45, 10, 51)	0.960214

Based on experiments we can say that we got better results with CEA for bigger state spaces - larger networks (larger maximal number of neurons). For smaller networks the DE algorithm served better results. RTRL algorithm is faster (mainly for very small networks) than DE or CEA, but RTRL is not able to optimize the topology of network.

It is clear that all presented algorithms especially the evolutionary algorithms – DE and CEA – have many parameters that can extensively affect their performance and results. The goal of this work was to prove some of the CEA's theoretical expected properties. We have experimentally tested that CEA is able to work with separate encoding of structure and behavior of individuals and evolve them in two different dimensions (using different algorithms) separately and continuously.

Table 2 shows the results of testing the CEA algorithm for different settings of maximal size of population. In table 3 the results of experiments with different number of gradient steps in each evolution iteration are presented. It can be seen that for low number of gradient steps the algorithm gives relatively bad results. It is because in this case the evolution process is used mainly for structure of network building/adaptation. The evolution process is used also for weight setting but it is a minority function. The weights are mainly adapted by gradient algorithm (realtime recurrent learning was used). In table 3 can be seen that increasing number of gradient steps gives better results.

7 Conclusion and future work

We have described the CEA algorithm in detail in this paper. On the experiments we have successfully tested some theoretically expected properties of CEA algorithm and we have shown that it can be used for ANN-based models creation/evolution. Comparison to other methods (real time recurrent learning and differential evolution algorithm) has been shown. We have presented the ability of CEA to produce smaller (simpler) models (networks) than RTRL and DE (used for weight matrix adaptation).

The first (debugging, not public yet) version of library for Wolfram *Mathematica* that implements the CEA algorithm was created. The experiments with ANN-based models in combination with CEA brings some new ideas how to improve the CEA algorithm, for example specific probability and balancing functions setting.

In future a lot of work on the theory of CEA algorithm and its applications are needed. The public version of CEA.m library is planed to be released.

8 Acknowledgments

This work is supported by the internal grant of Czech Technical University in Prague, IG: CTU0706813 – “Development of the ”continual evolution” optimization algorithm and its application to the artificial neural networks adaptation”, the grant Automated Knowledge Extraction (KJB201210701) of the Grant Agency of the Academy of Science of the Czech Republic and the research program “Transdisciplinary Research in the Area of Biomedical Engineering II” (MSM6840770012) sponsored by the Ministry of Education, Youth and Sports of the Czech Republic.

9 References

[1] Z. Buk and M. Šnorek. A brief introduction to the continual evolution algorithm. In *Proceedings of 6th International Conference APLIMAT 2007, Part III*, pages 71–76, Bratislava, February 2007. ISBN 978-80-969562-6-5.

[2] Z. Buk. Survey of the computational intelligence methods for data mining. Technical Report DCSE-DTP-2007-01, Czech Technical University in Prague, FEE, CTU Prague, Czech Republic, 2007.

[3] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. Evolving neural network agents in the nero video game. In *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games (CIG'05)*, Piscataway, NJ: IEEE, 2005.

[4] X. Yao. Evolving artificial neural networks, 1999.

[5] Kenneth O. Stanley, Bobby D. Bryant, Igor Karpov, and Risto Miikkulainen. Real-time evolution of neural networks in the nero video game. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI-2006, Boston, MA)*, pages 1671–1674, Meno Park, CA: AAAI Press, 2006.

[6] Lampinen Jouni A. Price Kenneth V., Storn Rainer M. *Differential Evolution A Practical Approach to Global Optimization*. 2005. ISBN 978-3-540-20950-8.

[7] Vladimír Mařík, Olga Štěpánková, and Jiří Lažanský et al. *Umělá inteligence 4 (in Czech)*. Academia, Praha, 2003. ISBN 80-200-1044-0.

[8] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989.

[9] R. J. Williams. Gradient-based learning algorithm for recurrent networks and their computational complexity. *Y. Chauvin and D.E. Rumelhart (Eds.) Back-propagation: Theory, Architectures and Applications.*, 1995.