Using Promoters and Functional Introns in Genetic Algorithms for Neuroevolutionary Learning in Non-Stationary Problems

F. Bellas, J.A. Becerra, R. J. Duro

Grupo Integrado de Ingeniería, Universidade da Coruña, Spain
fran@udc.es, ronin@udc.es, richard@udc.es

Corresponding autor:

F. Bellas
Escuela Politécnica Superior
Universidade da Coruña,
Mendizábal s/n
15403 Ferrol (A Coruña)
Spain

Tel: 34-981-337400 ext 3886
Fax: 34-981-337410
e-mail: fran@udc.es

Francisco Bellas is a *Postdoctoral researcher* at the University of A Coruña, Spain. He received the B.S. and M.S. degree in Physics from the University of Santiago de Compostela, Spain, in 2001, and a PhD in Computer Science from the University of A Coruña in 2003. He is a member of the Integrated Group for Engineering Research at the University of A Coruña. Current research activities are related to evolutionary algorithms applied to artificial neural networks, multiagent systems and robotics.

José Antonio Becerra received the B.S. and M.S. degree in Computer Science from the University of A Coruña, Spain, in 1999, and a PhD in Computer Science from the same university in 2003. He is currently a *Profesor Ayudante Doctor* in the Department of Computer Science and member of the Integrated Group for Engineering Research at the University of A Coruña. His research activities are mainly related to autonomous robotics, evolutionary algorithms and parallel computing.

Richard J. Duro received a M.S. degree in Physics from the University of Santiago de Compostela, Spain, in 1989, and a PhD in Physics from the same University in 1992. He is currently a *Profesor Titular* in the Department of Computer Science and head of the Integrated Group for Engineering Research at the University of A Coruña. His research interests include higher order neural network structures, signal processing and autonomous and evolutionary robotics. He is a Senior member of the IEEE.

Using Promoters and Functional Introns in Genetic Algorithms for Neuroevolutionary Learning in Non-Stationary Problems

F. Bellas, J.A. Becerra, R. J. Duro

Integrated Group for Engineering Research, Universidade da Coruña, Spain
fran@udc.es, ronin@udc.es, richard@udc.es

Abstract. This paper addresses the problem of adaptive learning in non-stationary problems through neuroevolution. It is a general problem that is very relevant in many tasks, for example, in the context of robot model learning from interaction with the world. Traditional learning algorithms fail in this task as they have mostly been designed for learning a single model in a static setting. Neuroevolutionary techniques have obtained promising results in this non-stationary context but are still lacking in certain types of problems, especially those dealing with information streams where different portions correspond to different models. An extension through the introduction of the concept of introns and promoter genes enables neuroevolutionary algorithms to improve their performance on this type of problems. Following this approach, an implementation of these concepts on a genetic algorithm for neuroevolution is presented here. This algorithm is called Promoter Based Genetic Algorithm (PBGA) and it uses a genotypic representation with a set of features that allows for an intrinsic memory in the population that is self-regulated, in the sense that functional parts of the individuals are preserved through generations without an explicit knowledge about the number of different tasks or models that have to arise from the data stream. Some illustrative tests of the potential of these techniques based on the continuous switch between completely different objective functions that must be learnt are presented and the results are analyzed and compared to other neuroevolutionary algorithms.

Keywords: genetic algorithms, artificial neural networks, neuroevolution, non-stationary functions, robotic learning

1 Introduction

In real-world robot learning, there are usually no direct targets that permit choosing the correct action for every situation, leading to processes whereby optimal behaviour must be learnt by exploring different actions and observing their results, thus, in a certain sense, by obtaining models of the interaction with the world. The domains in which these processes take place are usually continuous, partially observable, non-stationary and, in general, episodic. These four characteristics are what make it so difficult to provide good enough learning strategies for them.

Continuity implies that well-established approaches like Reinforcement Learning are not scalable in large state spaces because of the infinite number of possible states that must be handled [1]. On the other hand, the fact that the domains are non-stationary means that, the robot state, the environment and the objective may change in time, thus, there is a lack of a fixed mapping between solution encoding and solution fitness, and this problem is usually compounded with partial observability. It is in this context where neuroevolution, that is, to evolve artificial neural networks (ANN) using some type of evolutionary algorithm, becomes a reference tool due to its robustness and adaptability to dynamic environments [2] and in some types of non-stationary tasks [3].

However, it is also necessary to consider the episodic nature of the problem. The robot perceives episodes of sensorial information to be modelled as one model intermingled with episodes of sensorial information corresponding to other models. This implies that whatever perceptual streams the robot receives could contain information corresponding to different learning processes or models that are intermingled (periodically or not), that is, learning samples need not arise in an orderly and appropriate manner. Some of these sequences of samples are related to different sensorial or perceptual modalities and might not overlap in their information content; others correspond to the same modalities but should be assigned to different models.

The problem that arises is how to learn all of these different models, the samples of which are perceived as partial sequences that appear randomly intermingled with those of the others. Most traditional learning algorithms fail in this task as they have mostly been designed for learning a single model from sets of data that correspond to that model and, at most, some noisy samples or outliers within the training set. This problem becomes even more interesting if we consider that it would be nice to be able to reuse some of the models, or at least parts of them, that have been successful in previous tasks in order to produce models for more complex tasks in an easier and more straightforward manner.

In this work, we propose considering the concept of promoters and introns in neuroevolutionary algorithms. The concept of promoter is not new, but until now it has been applied very few times and only in terms of the ability to turn on or off any gene in a genotypic representation, no matter the meaning and the context of that gene. An example of this is the sGA [4][5][6] by Dasgupta or the promoters introduced in NEAT [7][8]. In this paper we go one step further and pair the

concept of promoters with that of functional introns in terms of turning on or off functional units and not just any gene. The concept has been implemented in an algorithm called Promoter Based Genetic Algorithm (PBGA) that uses a genotypic representation with a set of features that allows for an intrinsic memory in the sense that functional parts of the ANNs are preserved in the individuals through the generations. With this representation a self-regulated evolutionary mechanism is obtained that stores previously learned information without explicit knowledge about the number or type of target functions or models.

The rest of the paper is structured as follows: section 2 formally presents the domain of the problem. Section 3 introduces the concepts of intron and promoter and how they are implemented in the PBGA. Section 4 is devoted to the application results and to comparing this approach, as implemented in the PBGA, with other neurovolutionary algorithms in regards to this particular type of problem. Finally, section 5 presents some conclusions.

2 Background

Neuroevolution is the artificial evolution of neural networks through evolutionary algorithms and has shown to be a very powerful technique for learning in non-stationary problems [2][9]. Evolution has been applied to ANNs at three different levels: connection weights, architectures and learning rules. Typical approaches, where the architecture is fixed and evolution searches the space of connection weights, have been successfully applied in the last decade for solving very complex problems [10] but, recently, several articles [7][11] have argued that this approach limits the functionality of the ANN. As a consequence, several researchers have proposed different algorithms that evolve both the connection weights and the architecture of the ANN [2][7][11].

Some of the most relevant neuroevolution methods presented in last few years are SANE [12], a cooperative coevolutionary algorithm that evolves a population of neurons instead of complete networks; ESP [10], similar to SANE but allocating a separate population for each of the units in the network, where each neuron can only be recombined with members of its own subpopulation; and NEAT [7], nowadays probably the most widely used neuroevolutionary algorithm, which can evolve networks of unbounded complexity from a minimal starting point and that is based on three fundamental principles: employing a principled method of crossover of different topologies, protecting structural innovation through speciation, and incrementally growing networks from a minimal structure [8].

Initially, most of the work on testing and benchmarking evolutionary algorithms found in the literature was carried out on stationary problems through either linear or non-linear control benchmarks like the pole balancing problem. The last decade, however, has seen an increase in the application of evolutionary algorithms to non-stationary problems. Most of this work was focused on optimization problems and using typical benchmarks for non-stationary optimization such as Osmera's dynamic problems [13] or the dynamic Knapsack problem [14].

The application field of the work presented in this paper is not directly optimization but learning. As established by Yao in [2], "learning is different from optimization because we want the learned system to have best generalization, which is different from minimizing an error function on a training data set". In particular, the objective is to consider learning in non-stationary problems.

Thus, to formalize and frame the application domain of this work we will resort to the formalism in Trojanowsky´s work [3] for delimiting the scope of non-stationary optimization problems and extend it to learning problems in terms of the types of changes that may take place in the objective function. This way, real-world learning problems can be, in general, modelled by:

$$M(P) = (D,F,C)$$

Meaning that a model M of a problem P can be expressed by defining the variables of the problem and their domains (D), the objective function to be learned (F) and a set of constraints that must be satisfied (C).

In a general non-stationary problem these 3 elements D, F and C, can change over time. The constraints of the problem C may vary in time because solutions that were acceptable in a given instant of time, become unacceptable. For a detailed reference in dynamic constraint optimization problems see [15]. On the other hand, D could change if the domains of the variables are modified or if the number of dimensions of the search space changes. Finally, changes in the objective function to be learnt may occur with time either because the function is intrinsically time dependent or, in the case of robotic systems, because the robot moves around or changes tasks and this implies a different F.

In this paper, which initially considers the problem of robot learning in non-stationary problems, we will leave aside constraints and we will consider that the robot has an unchanging set of sensors and actuators (D does not change). Thus, we are trying to learn in real environments through models of robot-environment interaction and all that is clear is what sensor and what actuators the system has, but it is not clear for each task or learning process which of these are necessary or what is the real target to be learnt in each case. This implies dealing with changes in the objective function F, the most typical case in real-world learning in robotics, referenced in general as non-stationary problems.

In this sense, there are different types of changes of F that may occur in time [3]:

1. *Random changes*: where the next change in F does not depend on the previous one. This usually leads to the learning of different problems. It is the case where two streams of data corresponding to different models are intermingled and occurs when, for example, a robot is exploring an unknown dynamic environment.
2. *Non-random non predictable changes*: the changes in F are not random but they are too complex to predict. This is another typical situation in robotics where the types of different environments and/or tasks are limited but the robot cannot predict the next one it will be faced with.
3. *Predictable changes*: these are changes that may be predicted and they come in two flavours: cyclical and non cyclical. This situation is possible in real-world robotics, and would make life much easier, but it is not typical.

Furthermore, the changes in the objective function can be continuous (adiabatic) or discrete. For the former case, several authors have analysed different high diversity evolutionary algorithms or local search techniques that are able to follow the adiabatic changes in the objective function [14][16]. In the latter, it is obvious that the larger the jump, the more difficult it will be to follow the evolution of the landscape through local search techniques. In general real-world situations in robotics no assumption can be made about the type of change and, as therefore, any algorithm or learning process in this domain must support both.

Consequently, the problem domain that is being dealt with here is: *learning in dynamic environments characterized by changes in the objective function F (non-stationary problems) that correspond to any of the categories in [3] or their combination (random, non random, predictable, unpredictable, continuous or discrete).* The simplest case occurs when changes are predictable and continuous and the most difficult one when changes are random and discrete, but all of them are possible in real-world learning in robotics.

The solutions proposed in the literature to deal with the application of evolutionary algorithms to non-stationary problems can be classified into two broad categories [14]: *memory-based* approaches and *search-based* approaches. In the first group, the algorithm includes some kind of memory structure that stores genotypic or phenotypic information that can be used in the future to improve the optimization process. This memory may be internal, included in the chromosomes and evolved [5][17], or external, storing successful individuals that are usually introduced in the population as seeds [18][19]. In the particular case of neuroevolutionary algorithms, an example of memory based approaches is the work by D'Silva [8], where the authors apply the real-time version of the NEAT algorithm and study how to increase the probability of obtaining populations that can remember old skills as they learn new ones while in a dynamic video game situation by considering an external memory that stores successful ANNs that are inserted as seeds in future generations. However, the main problem of external-memory based approaches to non-stationary problems is that they are limited to periodic or predictable or detectable changes as it is necessary to know when an individual must be stored in the memory. That is, it is necessary to identify the different functions the system is learning or, at least when they change, usually by analysing the evolution of the error [20]. This procedure is very noisy and hard to apply in complex problems. Consequently, memory-based approaches perform better in periodic non-stationary problems [18], with predictable or easily detectable changes, where the individuals can be associated to a given objective function and stored in detected cases. For non-periodic non-stationary problems, both in the case of random changes and non-random non predictable changes, where it is not easy to detect the changes in objective function, search-based algorithms have been proposed. The algorithms developed for these cases usually rely on an extreme ability to continuously search and thus adapt relatively fast to the new objective function. This is generally achieved by trying to maintain a high level of diversity in the population [14][16]. These techniques, notwithstanding the high level of diversity in the populations, unless these populations are extremely large, basically follow the objective function as it changes, but do not have any memory information so as to return fast to previous situations. As a consequence, these techniques function at their best in adiabatic problems where transitions are smooth and thus can be easily followed.

Here we are interested in the ability to deal with both periodic and non-periodic non-stationary problems where changes of the objective function are not predictable or easily detectable. In addition, we are interested in environments where robots will operate, which imposes the need to remember to an extent previously encountered situations (models) and even reuse parts of them. Therefore, an intermediate approach based on internal memories and on maintaining diversity using a genotype-phenotype encoding that prevents the loss of relevant information through generations, unlike more destructive approaches such as [15] or [16], is necessary. To this end, there are several biologically inspired solutions in the literature that consist on tweaking with the representation of the individuals and their genotype-phenotype transformation through the selective expression of genes [17][21][22]. In this paper we will concentrate on the introduction of the concepts of functional introns and promoter genes within the genotypic representation of the chromosomes and we will show the potential of this type of approach by the integration of these mechanisms in a Genetic Algorithm called the Promoter Based Genetic Algorithm (PBGA).

An initial approach to the introduction of promoter genes was implemented on the Structured Genetic Algorithm (sGA), developed by Dasgupta and McGregor [4] as a general hierarchical genetic algorithm. They applied a two level interdependent genetic algorithm for solving the knapsack problem and developing application specific neural networks [6]. A two layer sGA was used to represent the connectivity and weights of a feed-forward neural network. Higher level

genes (connectivity) acted as a switch for sections of the lower level weight representation. Sections of the weight level, whose corresponding connectivity bits were set to one, were expressed in the phenotype. Those whose corresponding bits had the value of zero were retained, but were not expressed. The main difference between the sGA applied to neuroevolution and the PBGA is that the activation genes in sGA act at connection level while PBGA works with neuron units (neurons and their input connections), that is, functional units. This is a very relevant difference even though to enable/disable a neuron unit is much more disruptive, it permits preserving complex functional units and their relationships.

3 Promoter Based Genetic Algorithm

3.1 Biological background

There exist two basic biologically based approaches to gene expression: diploid representations and promoter based mechanisms. Diploid genotypes are made up of a double chromosome structure where each strand contains information for the same functions. Whenever a phenotype is constructed from the genotype, one of the two possible alleles for each gene is chosen following a dominance mechanism which may change with time. As not all of the genes making up the chromosomes are expressed, and as the fitness of an individual is determined by its phenotype, the recessive genes are shielded from selective pressure thus providing a memory within the encoding of the genotype. These techniques were introduced in computational evolution by Goldberg [23], who claimed that a diploid representation combined with a dominance map could outperform a standard evolutionary algorithm in dynamic problems. In this work, however, we will concentrate on the other gene shielding/expression mechanism, that is, the use of gene promoters and, consequently, loosely speaking, of functional introns as unexpressed pieces of genetic code because it is a more intuitive representation to work with ANNs.

In prokaryotes (bacteria and other simple cells) the entire DNA coding for a protein is continuous. In more complex, eukaryotic, cells, however, the encoding DNA is generally discontinuous: sequences of encoding DNA (exons) are interspersed with long sequences of non-encoding DNA. This non-encoding DNA sequences, usually about 10-fold longer than the exons, are called introns and even though for a long time they have been considered “junk”, the fact that they are so common and have been preserved during evolution leads many researchers to believe that they serve some function.

To control where a protein is encoded, the chromosome contains protein begin and protein end signals called codons. When the machinery of the cells sees that first begin codon, it knows that the instructions for making a protein begin at this point. A stop codon tells the cell's machinery that it has reached the end of the protein and should stop translating the code.

It must also be considered that almost every cell in an organism has a copy of every single gene the whole organism needs. Different genes are expressed in cells corresponding to different organs. Obviously, one would not want a gene coding for toes to be expressed in the lungs. Gene promoters are in charge of controlling these effects and they are important regulatory structures that control the initiation and level of transcription of a gene. They sit upstream of the gene and dictate whether, or to what extent, that gene is turned on or off.

3.2 Genotype-Phenotype encoding

At this point, we have provided an indication of the elements that should go into genotype encoding in order to allow for gene expression. These elements are exons and introns, gene promoters and codons. Using these elements, the “cell machinery”, that is, the part of the genetic algorithm (GA) in charge of constructing the phenotype, will know how to make the final organism from the genotype. If these elements are an intrinsic part of the encoding, they will provide for an unobtrusive way of evolving what is expressed and how and, thus, make the operation of the algorithm much smoother.

To demonstrate the potential of this type of structures when applied to neuroevolution, we have considered a GA that evolves the weights of feedforward artificial neural networks (any other evolutionary algorithm with a similar type of encoding could have been chosen). These neural networks are encoded into sequences of genes for constructing a basic ANN unit. Each of these blocks is preceded by a gene promoter acting as an on/off switch that determines if that particular unit will be expressed or not. In order to simplify the algorithm, it was decided to make use of these gene promoters also as start and end codons due to the position they occupy in the chromosome. For example, using the simple feedforward ANN representation shown in Fig. 1, the following chromosome could represent an individual in a neuroevolutionary algorithm with no genotypic-phenotypic transformation:

$$[W_{13}\ W_{23}\ W_{14}\ W_{24}\ W_{15}\ W_{25}\ W_{36}\ W_{46}\ W_{56}]$$

The genotypic representation used in the PBGA for the ANN phenotype shown in Fig. 1 is:

$$[1\ 1\ 1\ W_{13}\ W_{23}\ 1\ W_{14}\ W_{24}\ 1\ W_{15}\ W_{25}\ 1\ W_{36}\ W_{46}\ W_{56}]$$

where all the genes of value 1 are promoter genes. Thus, the first two genes represent that the two input neurons are enabled, the fifth gene represents that neuron 3 is enabled (controlling weights W_{13} W_{23}), and so on. Continuing with the same example, Fig. 2 shows the phenotypic representation of the PBGA chromosome:

$$[1\ 1\ 1\ W_{13}\ W_{23}\ 0\ W_{14}\ W_{24}\ 0\ W_{15}\ W_{25}\ 1\ W_{36}\ W_{46}\ W_{56}]$$

where promoter genes of neurons 3 and 4 are disabled and consequently these two neurons and their inbound connections are not shown in the phenotype.

As we can see, the basic unit in the PBGA is a neuron with all of its inbound connections as represented in Fig. 3. Consequently, the genotype of a basic unit is a set of real valued weights followed by the parameters of the neuron (in this case, a traditional sigmoid, but it could be any other) and proceeded by an integer valued field that determines the promoter gene value and, consequently, the expression of the unit. By concatenating units of this type we can construct the whole network. With this encoding we want to impose that *the information that is not expressed is still carried by the genotype in evolution but it is shielded from direct selective pressure*, maintaining this way the diversity in the population, which was a design premise as established in the previous section. Therefore, a clear difference is established between the search space and the solution space, permitting information learned and encoded into the genotypic representation to be preserved by disabling promoter genes.

As a consequence, in order to maintain previously learnt information in the chromosomes when dealing with non-stationary problems and taking into account that the information is in the structure of the ANN, the genetic operators must be tailored towards preserving these topological relationships. Although other approaches are possible, we have chosen to use the same topology for the genotypic representation of all the ANNs in the population to avoid complexities, like a continuous growth in the ANN size (which results in cpu intensive tasks) or the high number of parameters that are needed to control the combination of the different topologies, associated to other approaches where the topology is completely free like in NEAT [7]. This way, all the ANN genotypes have the same number of total neurons, in this case, within a two-layer feedforward representation. The designer simply imposes a maximum number of neurons per hidden layer, and all the ANNs are created with the same chromosome length. This does not mean that the ANNs resulting from the genotype-phenotype transformation have the same topology, as this depends on what functional units are enabled by the promoters. The PBGA usually starts with minimal phenotypical ANNs (just one neuron enabled per hidden layer) and evolution makes different types of ANNs (in terms of enabled neurons) coevolve together.

3.3 Crossover and mutation

The main problem that had to be dealt with in the implementation of the algorithm is how to perform crossover and mutation without being extremely disruptive or generating a bias in the evolution of what genes are expressed. We must bear in mind that we are crossing over not only weight values, but the neurons that conform the topology of the network. Consequently, it is necessary to be careful about how disruptive crossover or mutation will be on the information units found in the genotype. If two parent chromosomes are taken at random, they will probably not have the same expressed neurons, and these are the ones that directly affect the fitness of the individual when implemented in the phenotype. Thus, we find two types of information in the parent genotypes that must be recombined in order to produce an offspring: genes corresponding to expressed neuron units, which are responsible for fitness, and genes for unexpressed neurons, about which we have little information.

The crossover is panmitic in the PBGA, that is, one child chromosome is created from two parent chromosomes. This process implies crossing over whole neuron units. To be statistically neutral regarding the expression of the genes, crossover must be performed carefully taking into account the promoter genes that control the expression of the gene sequences. This crossover follows the following 3 rules:

1. If both parent units are expressed, the offspring unit is expressed and the weights are obtained applying a simple BLX-α crossover to the neuron unit's weights.
2. If both parent units are not expressed, the offspring unit is not expressed and the weights are directly inherited from one of the parents (50% chance)
3. When one unit is expressed and the other is not, there is a 50% chance of the offspring unit being expressed and the weights are inherited from the corresponding parent.

Thus, on average, the number of expressed units are preserved and a bias in this term prevented. In addition, the strategy of preserving the disabled neuron units and performing information crossover only in cases when both neurons are active, that is, where the crossover effect can be tested is followed.

Regarding mutation, things are simpler, and the only consideration that needs to be made is that gene promoters must be mutated at a different rate from that of regular genes. Note that mutating gene promoters may be very disruptive as it affects in a very serious way the composition of the phenotype, whereas mutation of the rest of the genes is, on average, much more gradual on the resulting phenotype. Consequently, we decided to use different mutation rates on the gene promoters (structural mutation) and on the real valued genes (parametric mutation). The structural mutation operator simply inverts the activation value from 0 to 1 or from 1 to 0, and the parametric mutation operator applies a non linear cubed random mutation mechanism ($f(x) = f(x) + rand(0,1)^3$) only to genes belonging to active neurons. As will be shown in the examples presented later, the values for these mutation probabilities are quite critical for the performance of the algorithm.

3.4 Basic operation

The PBGA is a genetic algorithm and follows the basic scheme of this type of algorithms, performing crossover and mutation over a selected pool of individuals that represent ANNs. The working cycle of the PBGA is very standard:

1. Creation of a random population of N individuals using the representation commented above
2. Fitness calculation over the whole population.
3. Selection of 2N individuals using a tournament selection operator.
4. Panmitic crossover with a probability P_c over the 2N population. The crossover operator is applied twice over the same parents and the offspring with highest fitness is selected. After crossover, an N individual offspring population is produced.
5. Mutation with probabilities P_{sm}, P_{pm} over the offspring population.
6. Fitness calculation over the offspring population.
7. Elitism that substitutes the worst individuals of the offspring population with the best individuals of the original population.
8. Return to step 3 for n generations.

The number of parameters that must be established by the user in the PBGA are 6: maximum number of neurons of the ANNs, population size, crossover probability, structural mutation probability, parametric mutation probability and the number of generations of evolution. All of them are problem-dependent but, as we will show in the next section, their values are intuitively easy to set up.

4 Application

To test the PBGA in the conditions used for its design and development, we take inspiration from the type of task that a real robot must perform when learning in a dynamic environment. Thus, we assume that a robot is executing a given task in an environment that changes due to several possible reasons, like a change in the ambient conditions or because the robot moves to another environment, and as a consequence, the model of the environment that the robot is learning, changes too. The objective of the experiments is to study effects of promoters over functional introns on the capability of dealing with changing objective functions (*F* in Trojanowsky´s terms [3]) as compared to other approaches. Consequently, in what follows we will concentrate on this, leaving aside other issues such as precision for which other types of functions and experiments should be chosen.

To simulate this change of objective functions to be learnt, we have used two different 3D functions that the PBGA must learn, and that are cycled periodically and non-periodically:

$$F_1(x,y) = (x+y)/2 \qquad x,y \in [-10,10]$$
$$F_2(x,y) = \sin(4x) + y\sin(y) \qquad x,y \in [-10,10]$$

It can be seen that both functions are very different (the second one is much more complex than the first) in order to test the capability of the PBGA of preserving the learned information in completely new situations.

To show the basic features of the PBGA as a consequence of its architecture, in a first experiment we used a given fitness function (F_1) for 100 generations of evolution and a different one (F_2) for the next 100 generations and kept cycling between them, simulating thus a periodic change of the environment. It is important to note that even though the change occurred periodically, this information is not known to the algorithm and what it is experimenting is an unpredictable change. The parameters used in this first experiment are shown in Table 1. We expect the PBGA to converge faster as the iterations (fitness function switch cycles) progress, because some of the previously learned information has a chance of

remaining in the unexpressed part of the genotype. Fig. 4 displays the root mean squared error (RMSE) for the first 2000 generations (20 cycles) of evolution (top graph) and 25000 generations (250 cycles) later (bottom graph). Function F_1 is learnt with a lower error than function F_2, as expected due to its simpler nature. The RMSE decreases during each 100 generation cycle as expected for a typical error evolution. When a change of objective function occurs, there is an error peak that is larger in the cycle from F_1 to F_2 due to the higher complexity of the second function. This peak rapidly decreases as the networks adapt to the new function. In addition, it can be observed how the error level at the end of each 100 generation cycle decreases in both functions along the cycles until it stabilizes. This result is clearer in Fig. 5 which displays the RMSE at the end of each cycle. The upper points correspond to the error level obtained for function F_2 and the lower ones those for function F_1. What is interesting in this figure is that the final error level decreases and stabilizes as in a typical evolutionary process, as if the changes of fitness or objective functions do not occur.

Fig. 6 displays the same behaviour as Fig. 5 but from a different perspective: the number of generations required by the PBGA in order to achieve a given error value, for example, 1.5 RMSE for the F_2 function when, as in the previous case, this function appeared interspersed with F_1 every 100 generations. The x axis represents the number of cycles when the F_2 function was learnt. It is clear from the graph that, except in the 8 initial cycles where the PBGA is not able to reach the desired error level of 1.5 RMSE in the 100 generations, there is a decreasing tendency in the error level (represented in the figure with a pointed grey logarithmic trend line). This is the main property of the PBGA and clearly indicates that the preservation of information in the unexpressed part of the genotype really leads to improvements in the speed of evolution towards a solution the system has seen totally or partially before. This is even made more evident if these results are compared to those obtained using a standard variable chromosome length genetic algorithm and that will be presented later.

Obviously, if the cycles are very long, there comes a point where probabilistically the information that is being preserved in the unexpressed parts of the chromosome will tend to degrade and be lost, especially if the functions are complex and require a large number of neurons for their modelling. This is equivalent for the case of a non-periodical change in the fitness function: this capability of preserving the information learnt depends on the length of each cycle. It seems obvious that if the cycle is too short, the algorithm may not be capable of learning the function and, if the cycle is too long, the population could converge and the probability of losing the information stored in the inactive neurons would increase. For example, in Fig. 7 we have represented the evolution of the RMSE with the same periodical switch of fitness function between the original F_1 and F_2 functions every 100 generations until cycle 50 (generation 5000). At this point a non-periodical and non-predictable change in the environment is simulated by maintaining function F_1 continuously during 7 cycles (700 generations). At the end of this period (generation 5700), the previous periodic change every 100 generations starts again. As seen in Fig. 7, from generation 5000 to 5700 the error level is very low because the PBGA is learning only the simple function F_1. In generation 5700, the PBGA must learn function F_2 again and is able to reach, in this case, an error level of 1.564 RMSE in just 100 generations. It must be pointed out that this error level was achieved in the initial run for the first time in generation 1186, as shown in Fig. 7, which means that the PBGA required 6 cycles of function F_2 to reach this level starting from an ANN population that had learnt function F_1 from generation 0 to 100. This experiment clearly indicates that the PBGA is able to preserve information of a fitness function in the chromosomes after a period of learning a completely different function and, as a consequence, this allows it to reach low error levels earlier. On the other hand, even though 1.564 RMSE is achieved in just 100 generations when switching back to function F_2 after 700 generations, this is a little bit more error than the algorithm was obtaining for the generation switches immediately before. This is a consequence of the fact that as more time is spent in a given objective function, more information stored in the genotype is lost. The parameters in this experiment are the same as for the previous one (Table 1).

The way the PBGA is switching between learnt representations can be appreciated in Fig. 8, where we display de average number of neurons that make up the networks in the population (black line) and the number of neurons of the best individual (grey line) in each generation throughout the evolution presented in Fig. 4 bottom. There are clearly two average sizes. One of them corresponds to the minimum size achieved by the algorithm for the network to solve the problem indicated by function F_1 which is around 18 neurons average, but with minimum peaks of 14 neurons. We must take into account that the minimum permitted number of neurons is 5 (2 inputs, 2 hidden layers and 1 output). The other size in Fig. 4 corresponds to the minimum possible size for the harder function F_2, which is around 22 neurons with peaks of 26. It is interesting to note how the PBGA switches neurons on and off when a transition between fitness functions occurs. Note that this number is the number of neurons active in the network, but it does not provide any indication of the distribution of neurons among layers.

Some comments must be made about the parameters required to obtain these results which are shown in Table 1. As explained at the end of section 3, the PBGA has 6 parameters to be adjusted: number of generations of evolution, maximum number of neurons of the ANNs, population size, crossover probability, structural mutation probability and parametric mutation probability. The first 3 parameters must be adjusted as in any other evolutionary algorithm with a compromise between computational time and accuracy (higher number of neurons implies a larger population size). The crossover probability has been fixed to 70% in all the trials to obtain the typical genetic algorithm behaviour with a balance in the preservation of the best individuals. Finally, the structural mutation probability and the parametric mutation probability are very relevant and must be adjusted carefully to achieve accurate learning. At this point, we

must say that the objective of the development of the PBGA is not to obtain a highly accurate evolutionary algorithm, because in learning processes this is not the main problem. Consequently, we have used the error level obtained in modelling functions F_1 and F_2 as a quality measure when cross-comparing, and not as an absolute measure of accuracy. Table 2 displays the RMSE obtained after 150 cycles for both functions F_1 and F_2 using different values for the structural and parametric mutation probabilities. For these functions, the ideal value of the parametric mutation parameter is lower than 1% but not zero, and for the structural mutation parameter the ideal value is around 2%. So, as a conclusion, it must be pointed out that structural mutation is very disruptive and it is not easy to adjust. Parametric mutation, on the other hand, must be assigned a low value.

As mentioned before, the main features of the PBGA are clearer when we compare it with a generic variable chromosome length genetic algorithm (VCLGA) we have implemented, with no gene promoters, that uses a direct transformation between genotype and phenotype. The ANNs in this genetic algorithm were created with the same architecture as in the PBGA and the algorithm works as a standard GA where the genes represent the connection weights and neuron parameters, and where the mutation operator creates or deletes hidden neurons and their corresponding connections. In fact, when hidden units are created, random values are assigned in the weights and parameters.

Fig. 9 displays the results obtained for the two algorithms in the first cycle and after a few thousand fitness function switches. In this case, and to make the problem simpler for the VCLGA we switched between two fitness functions corresponding to neural networks that modelled the sum of the inputs and networks that modelled the product of the inputs (the parameters used in the PBGA in this example are shown in Table 3). On the left hand side of the figure we have graphs corresponding to switching between the two fitness functions every 20 generations and on right hand side the switch is performed every 100 generations. The top graphs correspond to the evolution in the first function switch, the bottom ones to the evolution after many function switches (the x axis displays the number of generations that have been run). The dashed line in each graph indicates the PBGA and the solid line the VCLGA. All the data have been normalized to the best solution (whose error we indicate by 1). Several things can be seen in this figure. First, in the first cycle, both GAs perform similarly, especially at the beginning. After many cycles, it can be observed that the PBGA obtains much better results. This is especially noticeable in the case where only 20 generations of evolution are carried out between fitness function switches. As the chromosomes do not have time to converge, the genotype preserves the necessary information and the PBGA obtains the best result after two generations of evolution, whereas the VCLGA needs basically the same number of generations as in the first cycle. In the case where we run 100 generations before switching, the result is very similar, but now, as the phenotype population has more time to converge, it takes the PBGA a little longer to achieve the best results. Despite this longer time, it is still a lot faster than the standard GA and the final error obtained is almost halved.

Obviously, any comparison would not be complete if one of the current successful neuroevolutionary algorithms were not considered and the one that seemed more appropriate was the NEAT algorithm [7] for two reasons: it has been demonstrated to be very successful in certain types of dynamic tasks and it does implement a version of promoters, although they are applied to individual genes as in the case of the sGA and not to functional units. In order to perform the comparisons with respect to changes in objective functions the latest version of the standard NEAT [24] was employed. We have not employed rtNEAT because it uses an external explicit memory, so it represents an alternative method that, in fact, is also compatible with PBGA [20]. The parameters considered for NEAT were those of the p2nv.ne file included in the source code and used for a non-Markovian (no-velocity) pole balancing problem in the authors' work [7], with three exceptions:

- The population was raised to 2000 individuals to make it equal to that of the PBGA experiments.
- Consequently, *compat_thresh* was raised to 8.0.
- *recur_only_prob* was set to 0.0 because recursion is not necessary here.

The ANNs were initially minimal with two inputs, a bias neuron and one output. NEAT's behaviour may change considerably depending on the parameter's values, thus, tuning may be required for particular fitness functions. But, as Fig.10 shows, this fine tuning is not very relevant in our problem, as NEAT is not able to preserve genetic information over fitness function changes. This figure represents the evolution of the RMSE for the first 1200 generations cycling the fitness function between the previously defined F_1 and F_2 functions every 100 generations. The solid black line corresponds to the NEAT algorithm and the dashed grey line to the PBGA (it is the same data shown in Fig. 4 top). The results for NEAT are qualitatively comparable to those of the VCLGA. The RMSE level does not improve between F_1 or F_2 cycles and the fitness function seems to be relearned from scratch each cycle.

It is necessary to point out that NEAT was not designed for non-stationary problems and there are many reasons because it is not suitable for that:

- Elitism is not always applied. It has elitism inside a given species only if there are more than 5 individuals. If the species owning the best global individual has fewer than 6 individuals the individual will probably be lost.
- Species are heavily punished if their fitness doesn't improve in *dropoff_age* generations, making them disappear, but this is precisely what happens when we switch to a more difficult fitness function.

- Most of the species are purged if the global optimum does not improve in *dropoff_age* + 5 generations, which, again, will happen when we switch to a more difficult fitness function.

To give NEAT a chance, in a second test, *dropoff_age* was raised to such a high number (100000) that, in fact, it was deactivated and permanent elitism was established by eliminating the restriction of having more than 5 individuals in a given species for it to take place. The results can be appreciated in Fig.11. This figure depicts the evolution of the RMSE for the PBGA (grey dotted line) and the modified version of NEAT (black solid line) when cycling the fitness function between F_1 and F_2 every 100 generations. Things are a bit better for NEAT with these changes than in the previous case, and, sometimes, genetic information is preserved through fitness function changes, but it is still not the general rule. This is due to the fact that, unlike in the case of the PBGA, that to deactivate / activate functional blocks of an ANN is a very slow process (the algorithm deactivates / activates one connection at a time), and thus, the feature introduced does not generally imply an advantage over evolving again from the scratch when a fitness function change happens. It is obvious from the figure that the behaviour of the PBGA is much more stable and the preservation of genetic information is clear.

5 Conclusions

From the results obtained it is evident that allowing the genotypic representation of the organism to carry more information than is necessarily expressed in the phenotype results in a great advantage in terms of preserving genotypic diversity in the population even when the phenotypes have converged to a solution. When this occurs, the expressed parts of the genotypes tend to become very similar, thus increasing the frequency of the genes that help to achieve the solution within the population. When the fitness function changes, the genetic algorithm will be able to keep on working appropriately by drawing from the diversity present in the unexpressed parts of the genotype in order to modify the phenotypes and follow the new fitness function. This has been demonstrated to be more advantageous when this information is preserved in the form of functional units and not just any gene.

What is very relevant now is that a sort of genetic memory is created within the genotype due to the high probability of those useful functional units that were successful in previous generations, and consequently appeared more frequently in the population, to become a part of the unexpressed parts of the genotype. The immediate result of this memory is that when a fitness function that has been seen before (or which requires combinations of basic units that were used in previous successful runs) is contemplated again, the GA achieves the desired phenotype much faster than before. Not only is the PBGA able to reuse previously learnt components (neurons with their connections) to solve previously observed problems, but it is also able to do that without any explicit indication about how many different problems / situations will happen or how long they will occur, that is, it is not necessary to detect a change in the problem (a change in the fitness function from the evolutionary point of view).

Another advantage of this type of encoding is that phenotypes can grow or decrease in size depending on the processing required for each fitness function. The appropriate number of neurons will be selected in order to achieve the desired goal. This provides a very simple mechanism for obtaining variable size neural networks on one hand and as a consequence, it allows the GA to select the necessary inputs to perform the function it needs to carry out without having to compensate for redundant or unnecessary information. This is a very important point as it is much more difficult for a neural net to compensate for an input than to simply turn off that input neuron, which is basically what this type of evolution allows.

Summarizing, this mechanism provides for a clear separation between the search and solution space, that is, genotypic and phenotypic spaces, and consequently lack of diversity in the latter does not imply this lack in the former. It also permits a preservation of functionality while allowing a continuation of search, something that is very difficult when using traditional GAs evolving variable length ANNs. In the traditional case, any neuron pruning or neuron addition is a very dramatic change which usually leads to undesired results.

Acknowledgements

This work was funded by the MEC of Spain through project DPI2006-15346-C03-01and DEP2006-56158-C03-02

References

1. Gomez, F., Schmidhuber, J. Miikkulainen, R.: Efficient Non-Linear Control through Neuroevolution. Proceedings of the European Conference on Machine Learning (2006).

2. Yao, X.: Evolving artificial neural networks, Proceedings of the IEEE, 87(9), (1999), 1423-1447
3. Trojanowski, K., Michalewicz, Z.: Evolutionary Approach to Non-stationary Optimisation Tasks, Lecture Notes In Computer Science Vol 1609, (1999), 538-546
4. Dasgupta D., McGregor, D.: A structured genetic algorithm: The model and first results. Computer Science Department IKBS-2-91, University of Strathclyde, Glasgow, U.K., (1991).
5. Dasgupta, D., MacGregor, D. R.: Nonstationary function optimization using the structured genetic algorithm. Proceedings Parallel Problem Solving from Nature 2, (1992), 145-154.
6. Dasgupta D., McGregor, D.: Designing application-specific neural networks using the structured genetic algorithm. In COGANN-92 Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks, Baltimore, MD, (1992), 87-96.
7. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies, Evolutionary Computation 10, (2002), 99-127.
8. D'Silva, T., Janik, R., Chrien, M., Stanley, K., Miikkulainen, R.: Retaining Learned Behavior During Real-Time Neuroevolution, Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference (2005).
9. Dreiseitl, S., Jacak, W.: Genetic algorithm based neural networks for dynamical system modelling, Proceedings IEEE 1995 Int. Conf. Evolutionary Computation, Part 2 (1995), 602–607.
10. Gomez, F. and Miikkulainen, R.: Solving non-Markovian control tasks with neuroevolution, Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, (1999) 1356–1361.
11. Gruau, F., Whitley, D., and Pyeatt, L.: A comparison between cellular encoding and direct encoding for genetic neural networks. Proceedings of the First Annual Conference in Genetic Programming, (1996), 81–89.
12. Moriarty, D.E.: Symbiotic Evolution of Neural Networks in Sequential Decision Tasks, PhD thesis, University of Texas at Austin (1997), Tech. Rep. UT-AI97-257
13. P. Osmera, V. Kvasnicka and J. Pospichal,: Genetic algorithms with diploid chromosomes, In Proc. of Mendel '97, (1997), 111-116.
14. Mori, N., Kita, H., Nishikawa, Y.: Adaptation to a Changing Environment by Means of the Feedback Thermodynamical Genetic Algorithm, Lecture Notes In Computer Science; Vol. 1498, (1998), 149-158.
15. Farmani, R. Wright, J. A.: Self-Adaptive Fitness Formulation for Constrained Optimization, IEEE Transactions on Evolutionary Computation, Vol 7, Num 5, (2003), 445-455
16. Cobb, H. G.: An investigation into the use of hypermutation as an adaptive operator in genetic algorithms having continuous, time-dependent nonstationary environments, NRL Memorandum Report 6760, (1990), 523-529.
17. Ryan, C. Collins, J. J. Wallin, D.: Non-stationary Function Optimization Using Polygenic Inheritance, Lecture Notes in Computer Science vol 2724, (2003), 1320-1331.
18. Mori, N., Imanishi, S., Kita, H., Nishikawa, Y.: Adaptation to Changing Environments by Means of the Memory Based Thermodynamical Genetic Algorithm, Proceedings of the 7th ICGA, (1997), 299-306.
19. Eggermont J., Lenaerts, T.: Non-stationary Function Optimization using Evolutionary Algorithms with a Case-based Memory, Technical Report TR 2001-11 (2001)
20. Branke, J.: Memory enhanced evolutionary algorithms for changing optimization problems, Proceedings CEC99, Volume 3, (1999), 1875-1882.
21. Peow. K., Cheong Wong, K.: A new diploid sceme and dominance change mechanism for non-stationary function optimisation, In Proceedings of the Sixth International Conference on Genetic Algorithms, (1995), 159-166.
22. Kargupta, H., Sarkar, K.: Function Induction, Gene Expression, and Evolutionary Representation Construction, Proceedings of the Genetic and Evolutionary Computation Conference. vol 1, (1999), 313-320.
23. Goldberg, D. E., Smith, R. E.: Nonstationary function optimization using genetic algorithms with dominance and diploidy. In Second International Conference on Genetic Algorithms, (1987), 59-68.
24. NEAT source code. http://www.cs.utexas.edu/users/nn/downloads/software/neat.1.1.tar.gz

FIGURE CAPTIONS

Fig. 1. Genotypic representation of an ANN with 2 inputs neurons, one hidden layer with 3 neurons maximum and one output neuron corresponding to a PBGA chromosome: [1 1 1 W_{13} W_{23} 1 W_{14} W_{24} 1 W_{15} W_{25} 1 W_{36} W_{46} W_{56}]

Fig. 2. Phenotypic representation of an ANN with 2 inputs neurons, one hidden layer with 3 neurons maximum and one output neuron corresponding to a PBGA chromosome: [1 1 1 W_{13} W_{23} 0 W_{14} W_{24} 0 W_{15} W_{25} 1 W_{36} W_{46} W_{56}]

Fig. 3. Basic unit in the PBGA genotypic representation: a neuron with all of its inbound connections

Fig. 4. Evolution of the RMSE for the first 2000 generations (top graph) and 25000 generations later (bottom graph), obtained using the PBGA cycling between functions to be learned F_1 and F_2 every 100 generations.

Fig. 5. RMSE at the end of each 100 generation cycle obtained using the PBGA cycling between functions to be learned F_1 (lower points) and F_2 (upper points).

Fig. 6. Number of generations required by the PBGA to achieve an RMSE error value of 1.5 for function F_2 when this function appeared interspersed with F_1 every 100 generations. The grey dashed line represents a logarithmic trend line of the points.

Fig. 7. Evolution of the RMSE with a periodical change of fitness function between F_1 and F_2 every 100 generations until cycle 50 (generation 5000) where function F_1 is maintained continuously for 7 cycles (700 generations). At the end of this period (generation 5700), the previous periodical change every 100 generations starts again.

Fig. 8. Average number of active neurons in the population (black line) and number of active neurons of the best individual (grey line) through the generations obtained using the PBGA cycling between functions to be learned F_1 and F_2 every 100 generations.

Fig. 9: Evolution of the relative error using the PBGA (solid line) and the VLCGA (pointed line). On the left hand side we have graphs corresponding to switching between two different fitness functions every 20 generations and on the right hand side the switch is performed every 100 generations. The top graphs correspond to the initial generations and the bottom ones to the evolution after many cycles.

Fig. 10. Evolution of the RMSE using the original NEAT (black solid line) and the PBGA (grey dotted line) cycling between functions to be learned F_1 and F_2 every 100 generations.

Fig. 11. Evolution of the RMSE using the adapted NEAT (black solid line) and the PBGA (grey dotted line) cycling between functions to be learned F_1 and F_2 every 100 generations.

Table 1. Parameters used in the PBGA for the results shown in all figures except in Fig. 9

Table 2. RMSE obtained after 150 cycles for both functions F_1 and F_2 using different values for the structural and parametric mutation probabilities.

Table 3. Parameters used in the PBGA for the results shown in Fig. 9

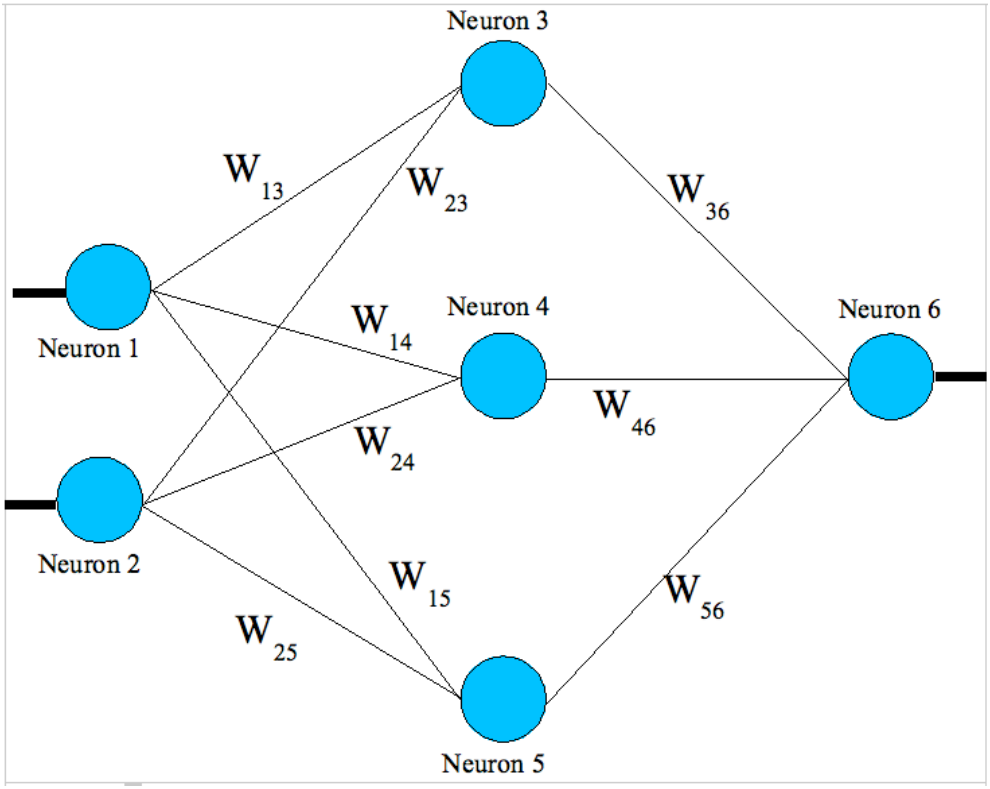
Neuron 3
W13
W23
W36
Neuron 1
W14
Neuron 4
Neuron 6
W46
W24
Neuron 2
W15
W56
W25
Neuron 5

Fig. 1

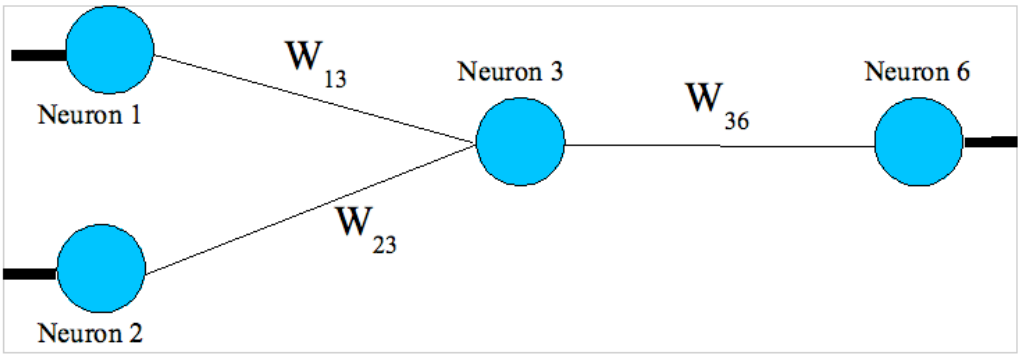

Fig. 2

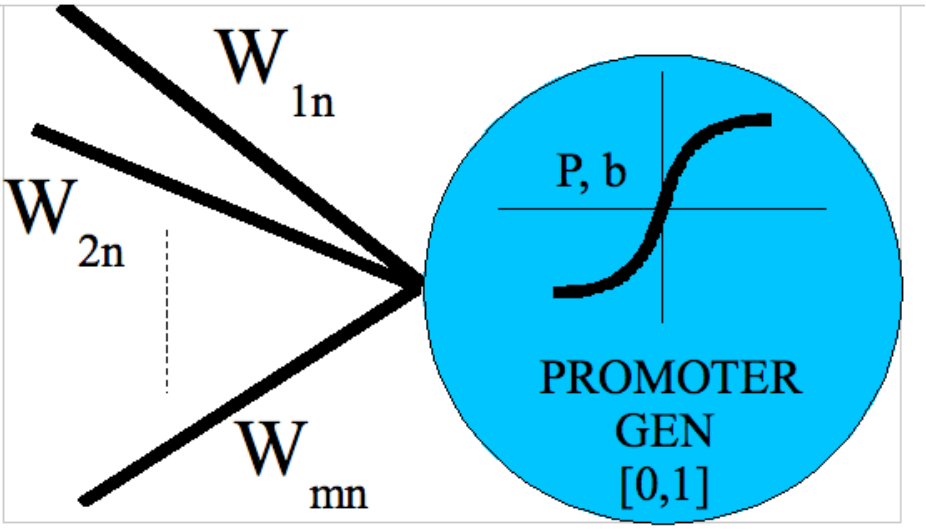

Fig. 3

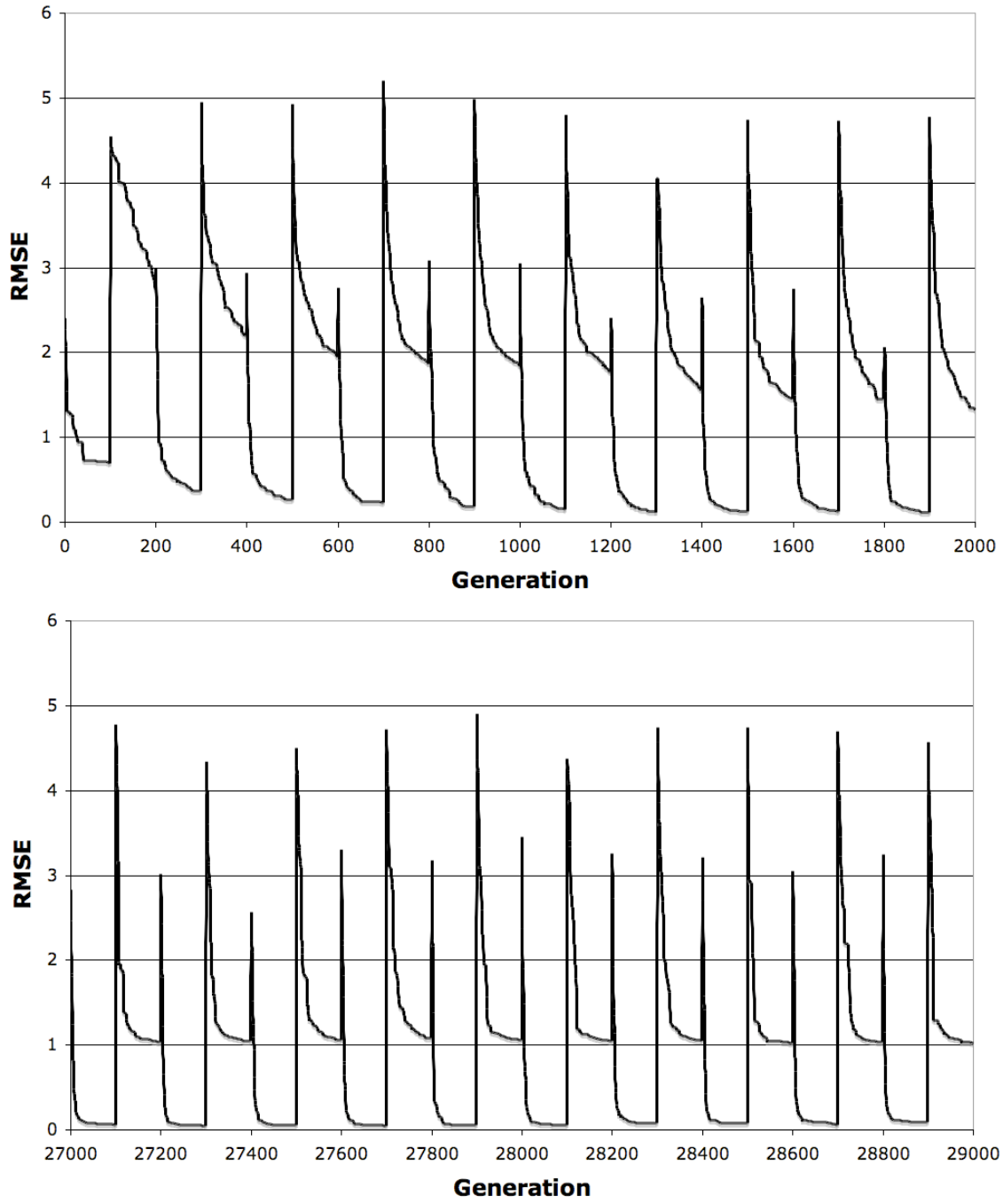
RMSE
Generation
0
1
2
3
4
5
6
0
200
400
600
800
1000
1200
1400
1600
1800
2000
RMSE
Generation
0
1
2
3
4
5
6
27000
27200
27400
27600
27800
28000
28200
28400
28600
28800
29000

Fig. 4

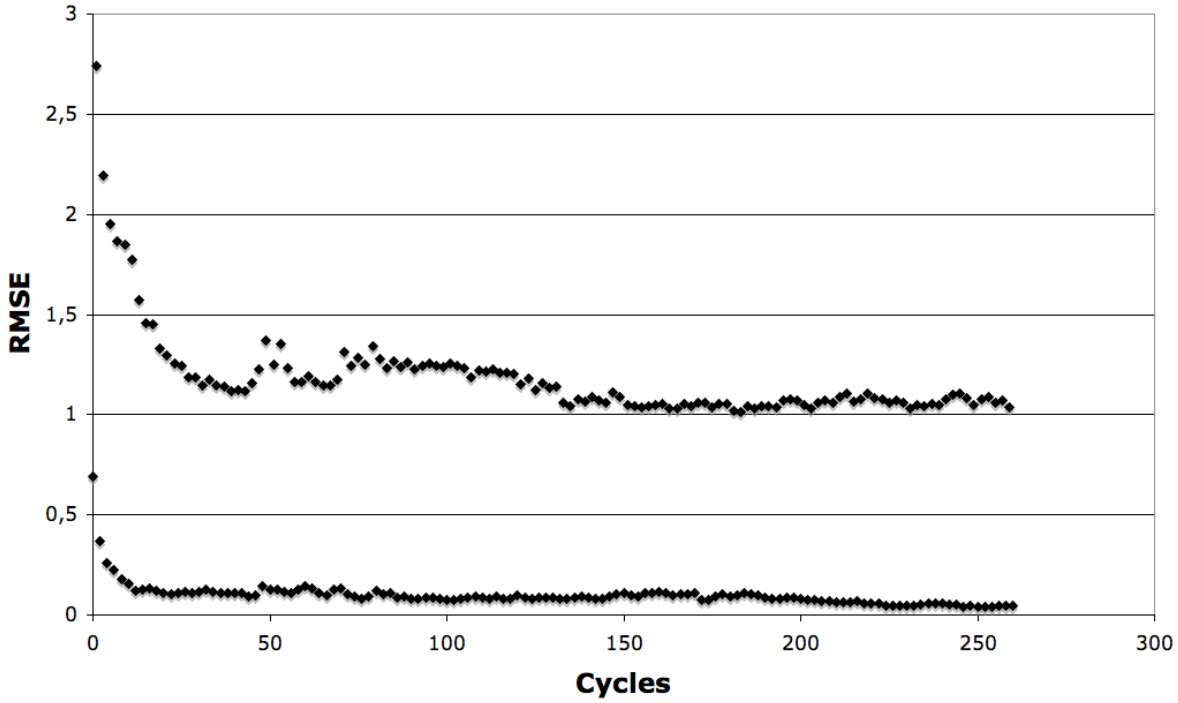

Fig. 5

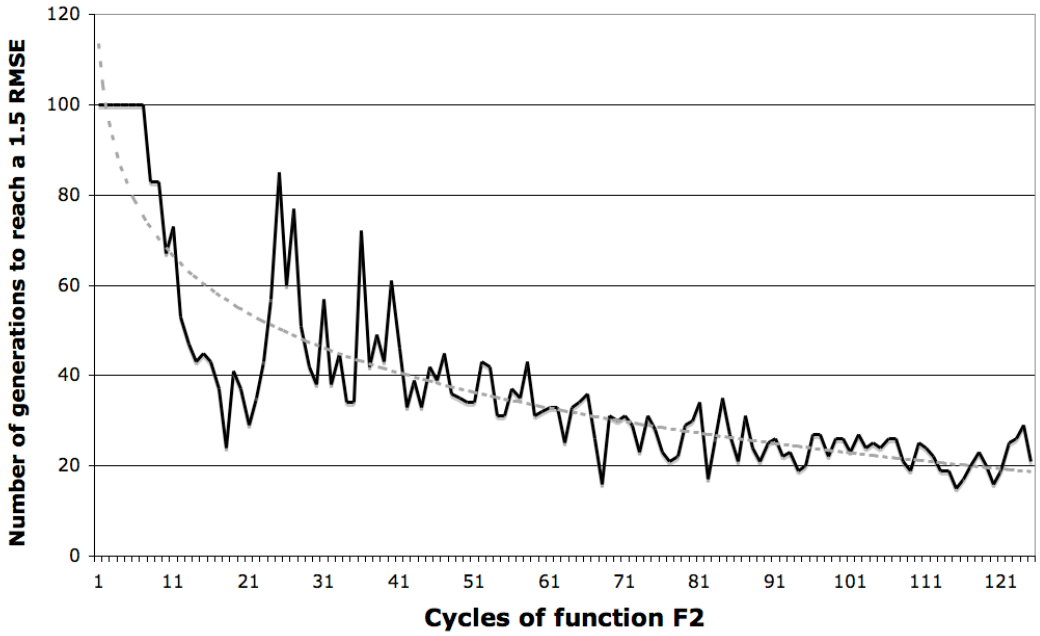
Number of generations to reach a 1.5 RMSE
120
100
80
60
40
20
0
1
11
21
31
41
51
61
71
81
91
101
111
121
Cycles of function F2

Fig. 6

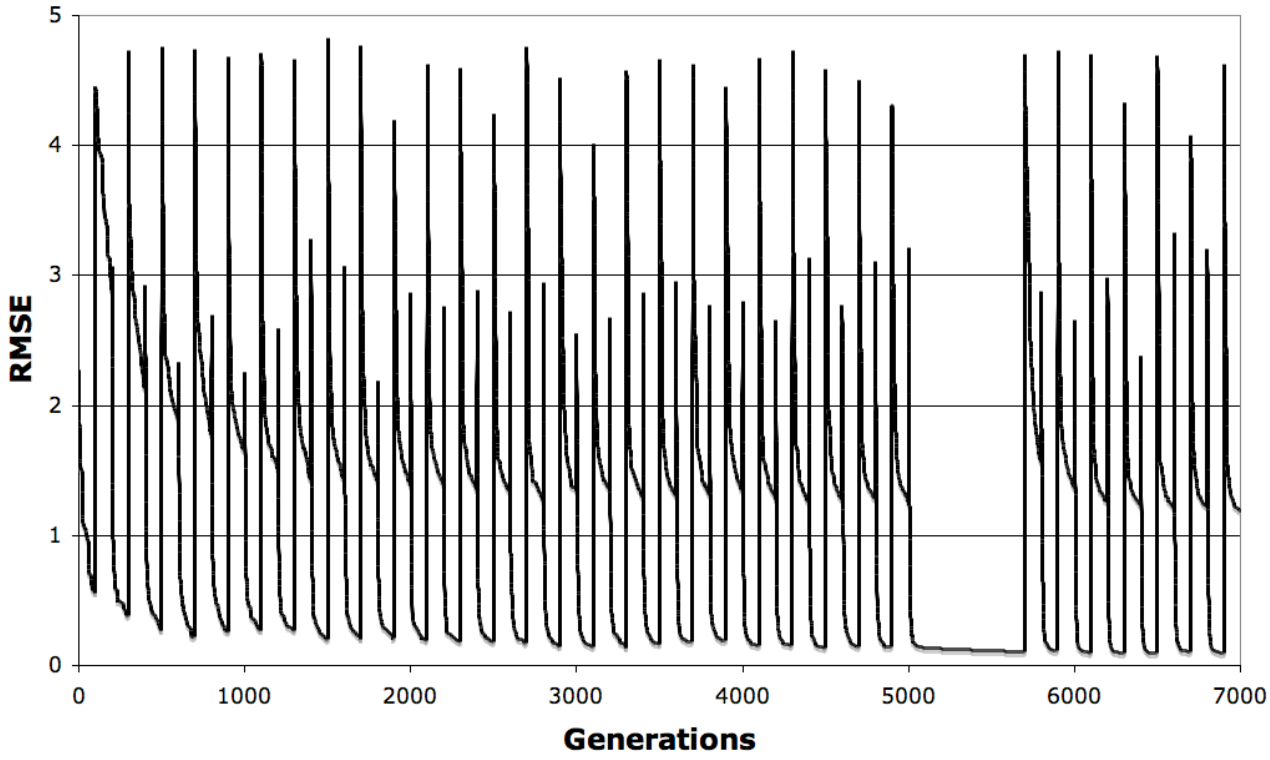
RMSE
Generations
0
1
2
3
4
5
0
1000
2000
3000
4000
5000
6000
7000

Fig. 7

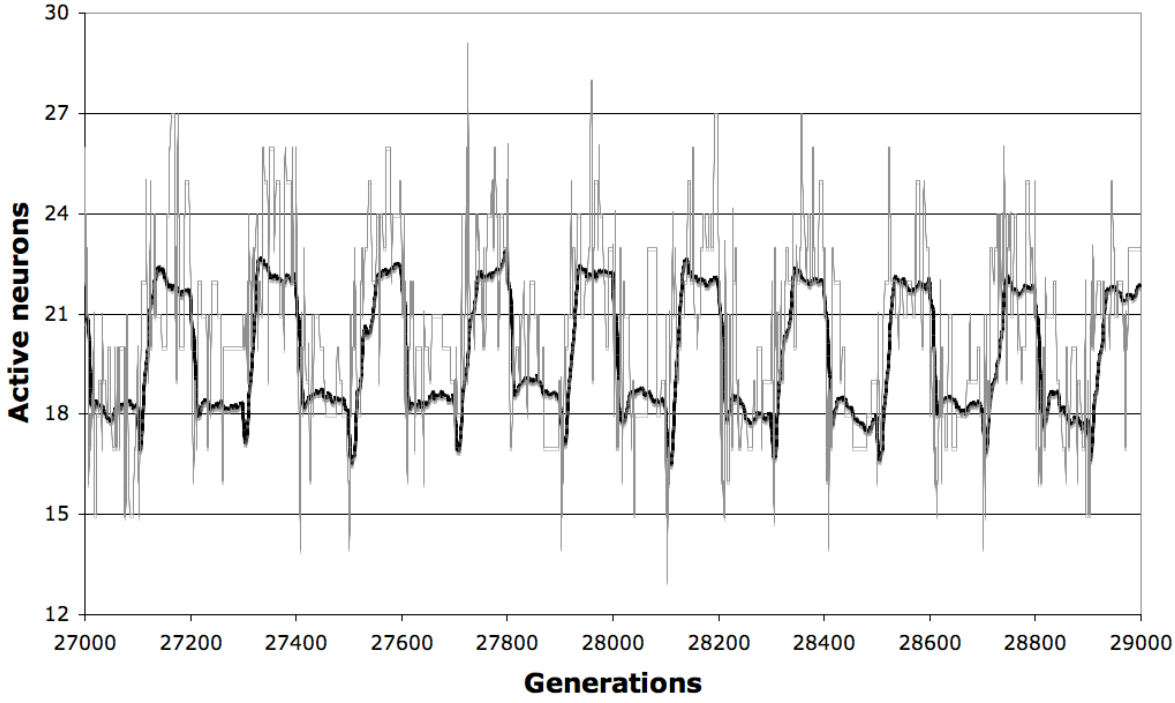
30
27
24
21
18
15
12
Active neurons
27000
27200
27400
27600
27800
28000
28200
28400
28600
28800
29000
Generations

Fig. 8

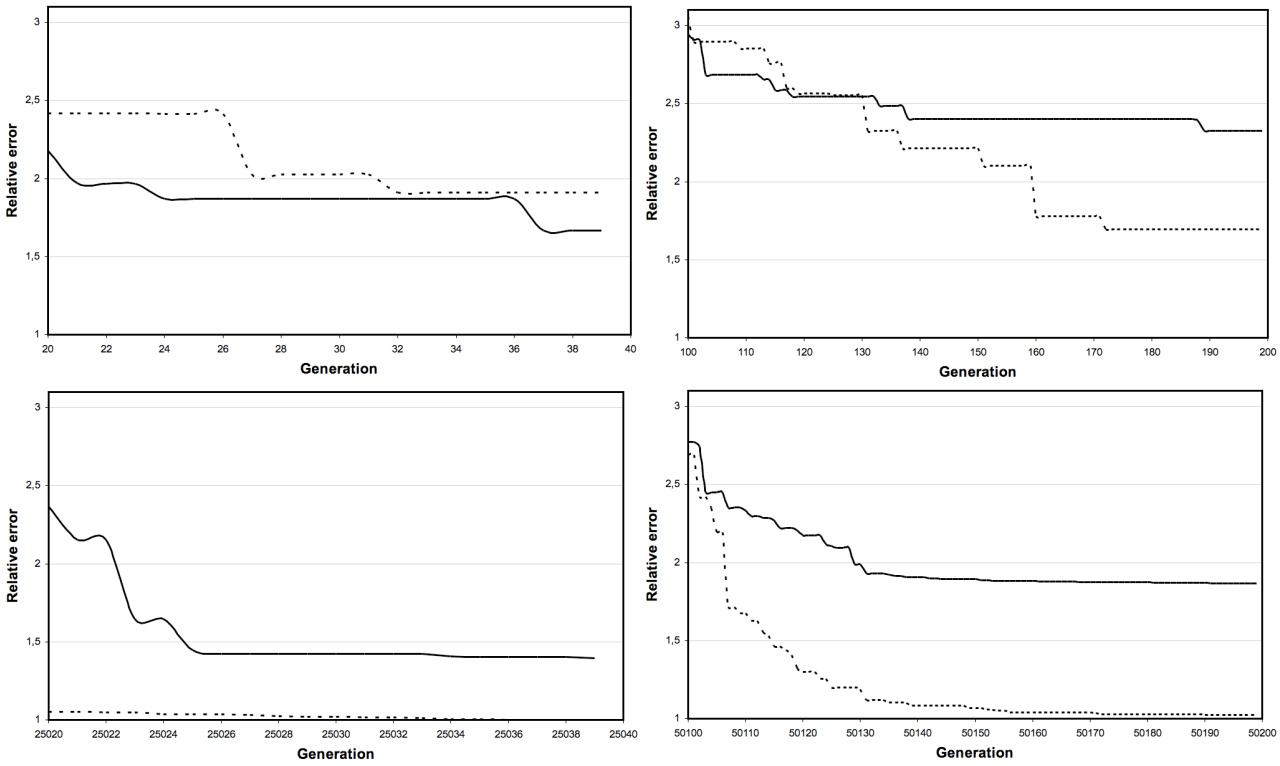
Relative error
Generation
3
2,5
2
1,5
1
20 22 24 26 28 30 32 34 36 38 40
100 110 120 130 140 150 160 170 180 190 200
25020 25022 25024 25026 25028 25030 25032 25034 25036 25038 25040
50100 50110 50120 50130 50140 50150 50160 50170 50180 50190 50200

Fig. 9

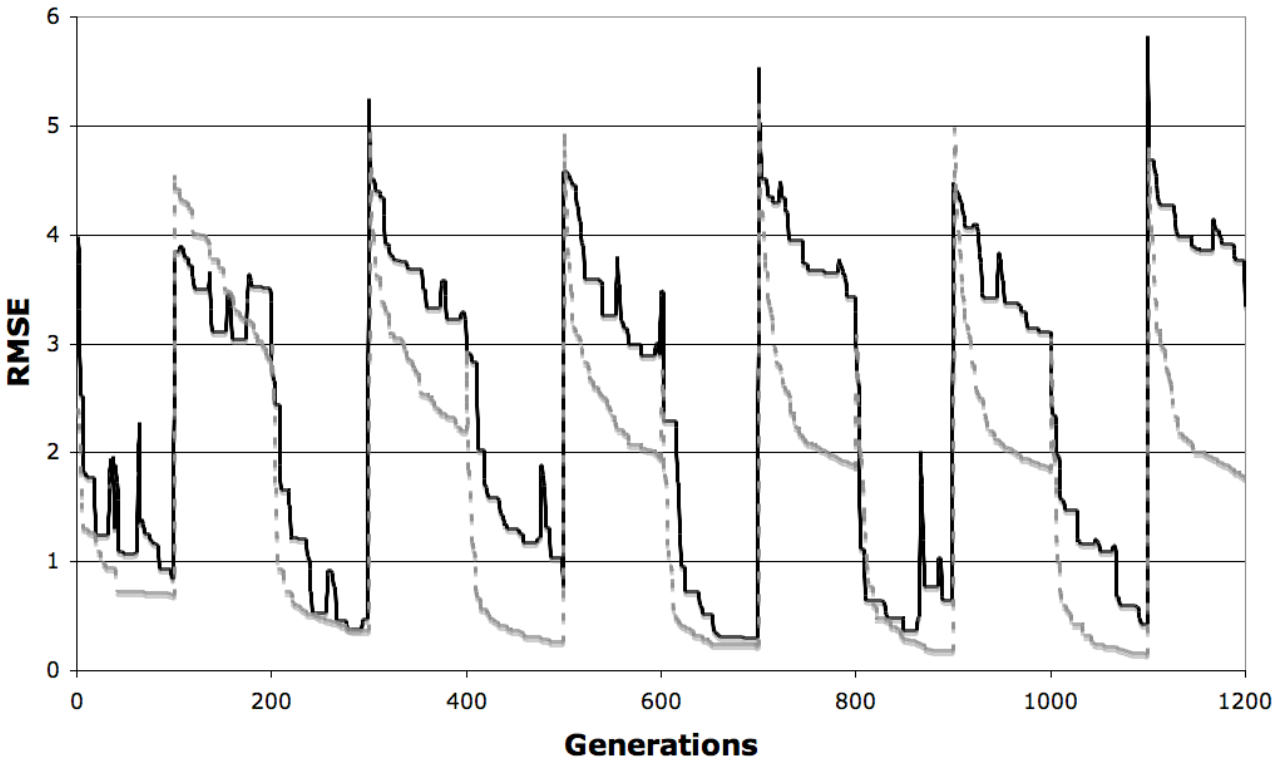

Fig. 10

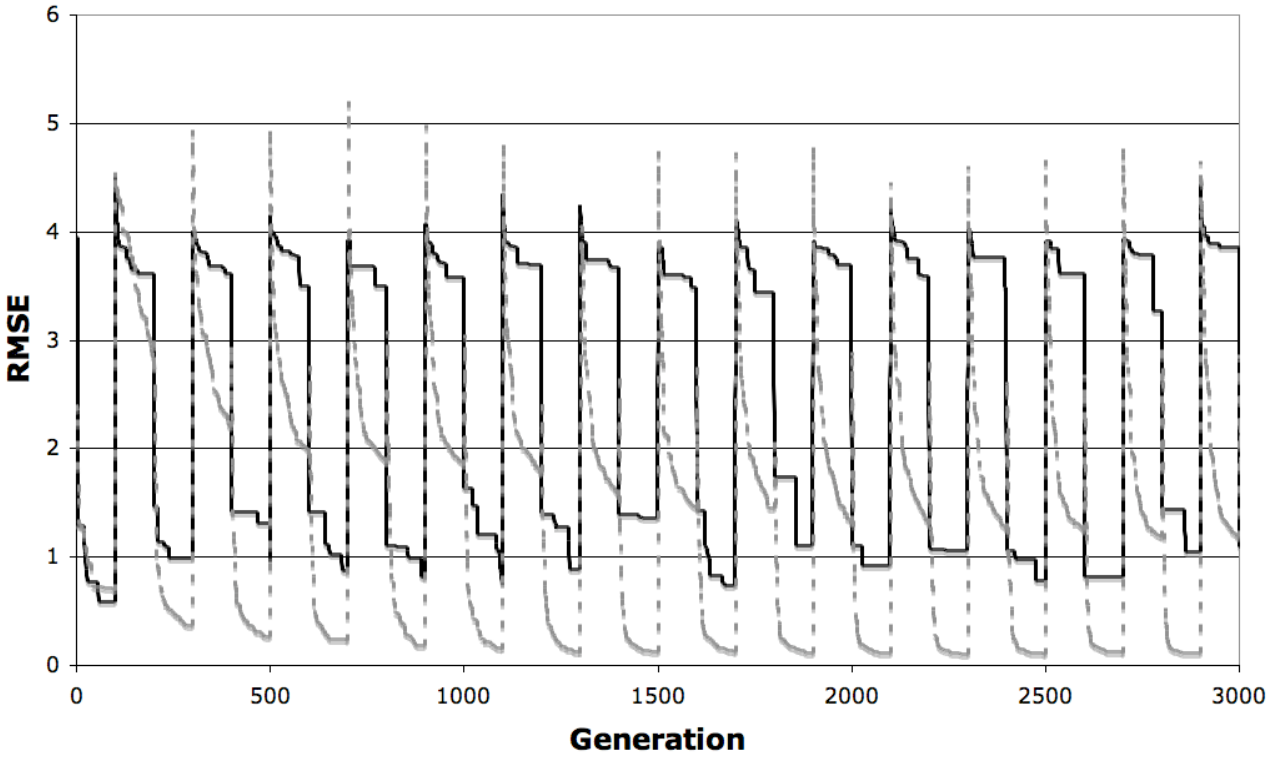

Fig. 11

	Parameter value
Generations	30000
Population size	2000
Maximum number of neurons in hidden layers	15
Crossover probability	70%
Structural mutation probability	2%
Parametric mutation probability	1%

Table 1

Probability values		**RMSE after 150 cycles**	
Structural mutation	*Parametric mutation*	*F1*	*F2*
2%	1%	0.051719	0.005134
0.5%	1%	0.065835	0.004277
10%	1%	0.122638	0.013085
2%	0%	0.133644	0.012396
2%	0.5%	0.048061	0.004425
2%	10%	0.149349	0.028701

Table 2

	Parameter value (left figures)	Parameter value (right figures)
Generations	30000	60000
Population size	600	600
Maximum number of neurons in hidden layers	6	6
Crossover probability	70%	70%
Structural mutation probability	2%	2%
Parametric mutation probability	1%	1%

Table 3