Simultaneous Optimization of Weights and Structure of an RBF Neural Network

Virginie Lefort, Carole Knibbe, Guillaume Beslon, and Joël Favrel

INSA-IF/PRISMa, 69621 Villeurbanne CEDEX, France
virginie.lefort@insa-lyon.fr

Abstract. We propose here a new evolutionary algorithm, the RBF-Gene algorithm, to optimize Radial Basis Function Neural Networks. Unlike other works on this subject, our algorithm can evolve both the structure and the numerical parameters of the network: it is able to evolve the number of neurons *and* their weights.

The RBF-Gene algorithm's behavior is shown on a simple toy problem, the 2D sine wave. Results on a classical benchmark are then presented. They show that our algorithm is able to fit the data very well while keeping the structure simple – the solution can be applied generally.

1 Introduction

Radial Basis Function Neural Networks (RBF NN) [1] are widely used in regression and classification tasks. They are often coupled with evolutionary algorithms, especially Genetic Algorithms (GAs) [2, 3, 4]. The GA is used to find the optimal parameters of the network.

However, in a neural network, all parameters cannot be considered to be equivalent. In particular, one can distinguish structural parameters from scalar parameters. The former define the general structure of the network: number of layers, number of neurons by layer and the topology of the network; they directly influence the capabilities of the network. The later, namely the neural weights and bias parameters, define the precise input-output mapping of the network. The scalar parameters clearly strongly depend on the structural parameters.

A classical GA can easily encode the scalar parameters, as their number is known once the structure is chosen, and so we would expect it to perform well in this regard. However, a more sophisticated evolutionary algorithm will be required to optimize the structural parameters in conjunction with the scalar parameters.

In this context, we present here a new Genetic Algorithm that can optimize simultaneously both the structural and scalar parameters of a feed-forward RBF NN. We have named it the RBF-Gene algorithm.

In the next section, we will briefly present some work on how to optimize an RBF-NN using Genetic Algorithms. We will then present our algorithm. In the last section, we will present the results obtained with it and compare with other published results on some benchmark tests.

E. Talbi et al. (Eds.): EA 2005, LNCS 3871, pp. 49–60, 2006.

2 Evolving Neural Networks Using Genetic Algorithms

Radial Basis Function Neural Networks are classical neural models with one layer of hidden neurons. They are used in classification or regression tasks as they are universal approximators.

In an RBF NN, the output of the network o is a weighted sum of the output of H hidden neurons. The transfer function of each neuron is a Gaussian function $g(\boldsymbol{X})$:

$$o(\boldsymbol{X}) = \sum_{j=0}^{H} w_j.g_j(\boldsymbol{X}) = \sum_{j=0}^{H} w_j.e^{\frac{-\|\boldsymbol{X}-\boldsymbol{\mu_j}\|^2}{\sigma_j^2}}$$

with $\boldsymbol{X}$ the input vector of the network, $\boldsymbol{\mu_n}$ the vector representing the center of the Gaussian for the n^{th} hidden neuron and σ_n its standard deviation.

Implementing an RBF NN is a two-stages process. First, we have to choose the structure of the network; second, we must find its free parameters. Once the structure is fixed, the free parameters can be chosen manually, often the case for the Gaussian parameters; analytically, with high sensitivity to the quality of the dataset; or by a learning algorithm, generally applied only on the output weights w_{ij}. However the free parameters, and therefore the behavior of the network, strongly depend on the network's structure.

In "difficult" cases, with few examples or significant noise, an evolutionary algorithm (like GAs) is a good solution. GAs can be used to optimize the parameters of the neurons directly: The structure is fixed at the beginning of the run and the algorithm finds strong values for the fixed number of parameters.

Goldberg [5] used GAs in this manner as early as 1989. It is easy to use and is therefore rather popular (e.g. [2].)

In another approach, GAs are used to optimize the structure of the network [3, 6, 7]. Then another algorithm such as a learning algorithm optimizes the scalar parameters for each structure. This method is time consuming since for each step in the structure algorithm we have to perform a complete optimization on the scalar parameters. Moreover the fitness associated to a specific structure strongly depends on the optimization algorithm used to compute the weights.

To overcome the limits of these approaches, "integrative" GAs have been proposed. The idea is to optimize simultaneously the structure and the weights of the network. The main problem is then the encoding of the chromosome. Two different solutions exist: the "direct encoding" scheme, where the structure and the weights are directly encoded in the same string; and the "indirect encoding" scheme, where the chromosome contains generative instructions used to build the network.

The direct encoding method was used successfully in several works (e.g. [4, 8]) with variable-length chromosomes. But the encoding introduces difficulties in creating offspring from parents[1]. One solution is to use species, where each

[1] Chromosomes are typically made of a structural part and a parameter part. Since the size of the parameter sequence depends on the values encoded in the structural sequence, the crossover operator often doesn't make sense.

species represent networks with the same structure. Since genetic exchange may not be possible between species, such an algorithm is closely related to and has many of the problems of two-phases optimization.

The direct encoding scheme also suffers from the "permutation problem"[2] that leads to the failure of the crossover. Some work has been done on this problem [9] but extra computations on the individuals, such as reordering the genes on the chromosome or ignoring duplicated genes, are required.

On the other hand, the indirect encoding scheme (see [3] for a review) is based on generative strategies: It doesn't directly encode the network but rather how to construct it from a seed. Generative strategies can be based on a grammar with a seed and rules or cellular growth, with instructions to add neurons and edges from one progenitor cell.

At first glance, this scheme seems to be better than direct encoding. However, much of the work has been done on simplified networks, such as binary weighted networks. Moreover mutations have counter-intuitive effects in these systems and the fitness landscape can be much more complicated than in a simple GA.

3 The RBF-Gene Model

In this context, we have developed an evolutionary algorithm based on GAs whose main goal is to optimize both the structure and the weights of a feed-forward RBF neural network. Our central idea is to build a chromosome in which each gene is an atomic entity that can be combined to the others to produce the phenotype (i.e. the RBF NN.) In order to avoid the difficulties encountered by generative approach, the product of one particular gene must not depend on the values of the other ones, nor on their relative positions along the chromosome.

Moreover, in order to avoid the "direct encoding" problem, we need to forbid any hierarchy between the genes. In other words, the genes encoded onto the chromosome must be homogeneous, i.e. they all code for the same kind of "basic block" of the answer.

3.1 Principles

This idea of homogeneous units is partly inspired by molecular biology and particularly by the encoding of the proteins on the chromosome. In biology, the translation process is only based on local rules: promoters, terminators, Shine-Dalgarno sequences, start/stop codons and so on. Consequently, adding, deleting or changing a gene (i.e. a protein) will have no consequences on the other proteins: The effect will only be visible at the global, phenotypic level. We wanted to have the same property for our algorithm as we wanted to make structural changes easy by adding, deleting or removing neurons or connections without changing the entire chromosome.

[2] The same genes are encoded in a different order in two individuals and so the one-point crossover leads to individuals without a special gene or with two copies of it.

In the RBF-Gene model, our basic block will be a "kernel": that is a complete hidden neuron together with all its numeric parameters, namely the mean and the standard deviation of the Gaussian, and the output weight.

Each kernel will be encoded on the chromosome as a "gene"; so the number of genes will indicate the number of kernels. In order to allow all the possible structures, we have to allow a variable number of kernels, i.e. a variable number of genes encoded onto the chromosome. The simplest way to do it is to have a variable-length chromosome.

As in biology, our genes will be located on the chromosome using purely local rules. In particular, we include two special sub-sequences indicating the beginning and the end of each gene. The chromosome is then a succession of coding and non-coding sequences of different size and purely local rules are required to decode it: if a coding sequence appears or disappears elsewhere in the genome, there will be no influence at all on the present genes. Some work has been done showing the interest of variable length chromosome and the use of coding and non-coding sequences, for instance [10, 11, 12].

Since all the kernels are equivalent, the order or the position of the corresponding gene on the chromosome doesn't matter and the permutation problem vanishes. Moreover, this property enables us to introduce rearrangements (i.e large scale mutation operators) that will help avoiding local optima by significantly changing the genetic code. Specifically, we will show that the sequence of copy-and-edit used by natural evolution of genes can be used by the RBF-Gene algorithm as well.

To create the next generation, we need to compute the fitness of each individual. This is done in three steps : First, using special sequences, we find the genes on the chromosome; second, using a genetic code, we extract all the parameters of the corresponding kernel; third, as we now have all the hidden neurons and their links, we construct the NN and test it on the dataset.

Once all the individuals are evaluated, we use a standard evolutionary process to compute the next generation. For the recombination/mutation step, we introduce rearrangement operators that change the structure of the chromosome. These operators modify the chromosome on a large scale independently of the nature of the region (coding or non-coding sequence).

3.2 Encoding

In the RBF-Gene algorithm, each gene encodes for the parameters of a hidden neuron: its mean vector $\boldsymbol{\mu}$ and its standard deviation σ. Moreover, it also encodes for the output links w_i. So, if we have n input values and m output weights, we have $n + m + 1$ real values to define for each neuron (n for the mean vector, 1 for the standard deviation and m for the output weights).

The simplest way to encode a value onto a chromosome is to use a binary encoding. So we need a "0" base and a "1" base for each parameter. This gives us an artificial genetic code: Our chromosome will be a string of characters (A, B, C...) and each parametric value has two characters associated with: one for the "0" and one for the "1". As we want a homogeneous chromosome, using purely

local signals to detect genes, we add two special characters for the start and the stop[3]. So our chromosome is a variable length string built with a $2(n+m+1)+2$ character alphabet.

Since we have a homogeneous chromosome, there are no special regions on the chromosome or on the gene. The different characters are mixed together at random (i.e. the parameters are not encoded sequentially). In order to compute a parameter, we only have to extract the corresponding characters: thanks to the genetic code, the mixed character string (the gene) is converted into three ordered binary strings (one per parameter). Each of them is then transformed into a numerical value using a Gray code [13].

One of the major advantages of our model is that gene's length is no longer fixed: it only depends on the relative position of a start character and the next stop character. Thus the number of bits per parameter is not fixed at all. This overcomes a classical drawback of binary encoding in GAs since the precision of each parameter is able to evolve by simply modifying the length of the gene. Consequently, the algorithm is able to generate rough solutions at first, with a small number of kernels or with numeric values of low precision. It can then progressively refine the input/output mapping by either adding new kernels or enhancing the precision of existing ones.

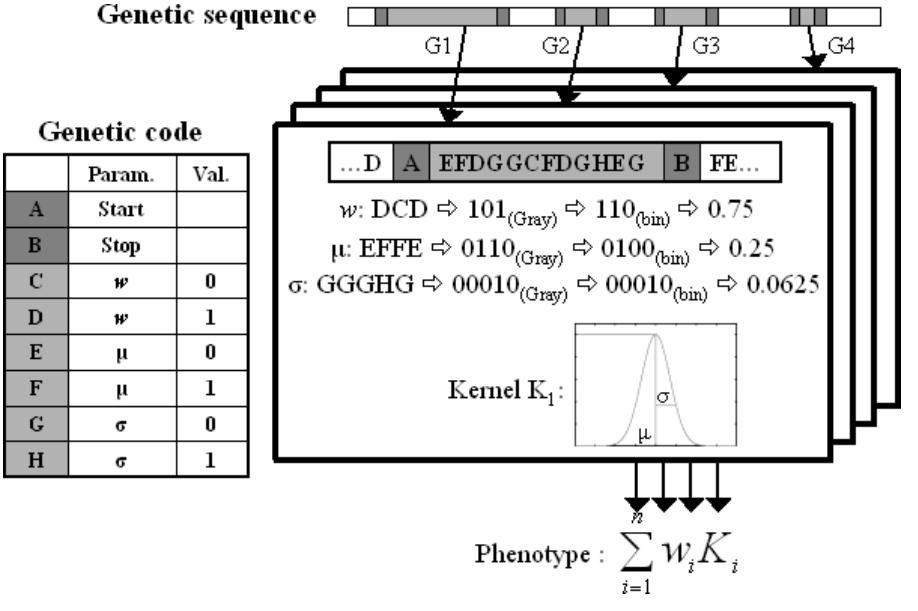

Fig. 1. A simple example of the mapping from the chromosome to the neurons. Here we only have one input value ($n = 1$) and one output value ($m = 1$). Thus the genetic code is composed of $2 + 2(1 + 1 + 1) = 8$ letters.

[3] Of course, we can find start signals inside a gene or stop signals between genes. In such a case they are ignored.

The overall translation algorithm can be summarized as:

1. Find the genes using the alphabetic start and stop signals
2. For each gene
   (a) Extract each parameter sequence using the "genetic" code translation
   (b) Decode each parameter using a variable-length Gray code
   (c) Build the associated Gaussian kernel K
3. Construct the neural network
4. Evaluate the individual on the data

A simple example of the first two steps is shown in figure 1.

3.3 Evolution

Our chromosomes are homogeneous: they are a simple string of characters. Moreover the structure of the chromosome is free to evolve without any influence on the phenotype: a gene can move from one locus to an other or two genes can be swapped. So the chromosome length can vary and so does the number of genes and we have more biologically inspired operators available to us than the two classic ones: point mutation and crossover.

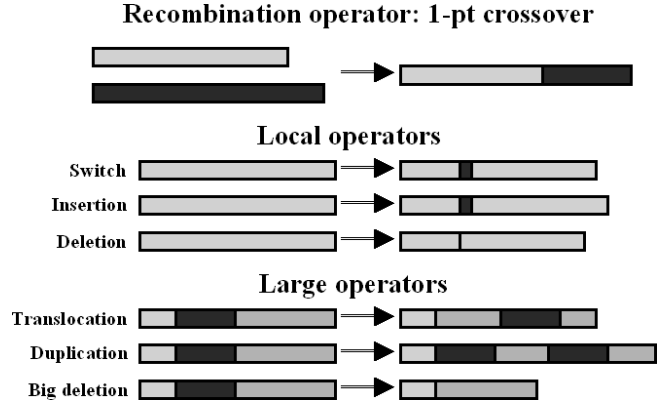

Fig. 2. A schematic view of the different operators

To create the next generation, we use two types of operators: first, the recombination operator if needed, and then the mutation operators. The selection of the fittest is done by a roulette-wheel based on the rank[4].

The recombination operator used here is a classic one-point crossover and the crossing point is chosen randomly. The probability of using it can vary between 0% for clonal reproduction to 100% for sexual recombination.

[4] However, the RBF-Gene algorithm can be used with any selection operator without loss of its properties.

We have two families of mutation operators (see figure 2):

- Local operators: They modify only one base. As the structure can change, we can use the traditional switch operator but also the local insertion or deletion.
- Large operators (rearrangements): They modify the global structure either by translocation, which move one part elsewhere on the chromosome; duplication, which copy and paste a sequence elsewhere; or large deletion, which erase a part of the chromosome.

Notice that the local mutation rates are given per base while the rearrangements rates are given per chromosome. Thus, the mutation effect remains constant whatever the genome size: since the "cut and paste" points are chosen randomly, the average number of bases affected by one rearrangement directly depends on the genome size [14].

The structure of the chromosome can then change and genes can be added, deleted or moved along the genome. So, the algorithm can adapt:

- **the complexity of the solution** by changing the number of genes
- **the local precision of each parameter** by changing the number of bases in each individual gene
- **the structure of the genetic sequence** by changing the order of the genes or the length of the non-coding sequences. It can be seen as the evolution of the robustness and the evolvability of the solution: the algorithm can change the structure in order to resist best to deleterious mutations or to help evolution to find optima more quickly.

4 Simulations and Results

In order to illustrate the behavior of the algorithm, we will first present results on a simple toy problem (the 2D sine wave, a $\mathbb{R}^2 \rightarrow \mathbb{R}$ problem). Then, we will present a real regression problem, the Boston dataset (a $\mathbb{R}^{13} \rightarrow \mathbb{R}$ problem), and compare our results with other results on the same benchmark[5].

4.1 2D Sine Wave: Graphical Results

The 2D sine wave is a straightforward problem which is interesting because the results can easily be visualized. The goal is to approximate the curve:

$$y(\boldsymbol{x}) = 0.8sin(\frac{x_1}{4})sin(\frac{x_2}{2}), x_1 \in [0;10], x_2 \in [-5;5]$$

This test has been proposed by Orr [17] and we use the same protocol for our experiment. We generate a training set of 200 patterns sampled at random. We add a normally-distributed noise with $\sigma = 0.1$ and zero mean. The test set contains 400 noiseless samples arranged as a 20*20 grid covering the input ranges. The fitness used by the algorithm is the total squared error (SE) on the

[5] The algorithm has been tested on other datasets as well [15, 16]: the $sin(12x)$ problem, a $\mathbb{R} \rightarrow \mathbb{R}$ problem or the abalone dataset, a $\mathbb{R}^9 \rightarrow \mathbb{R}$ problem.

learning set (the fitness does not depend on the genetic parameters like the size or the number of genes). The test set, independent from the training set, is used to evaluate the generality of the solution as we want to fit the 2D sine curve at all points, not just at the training points.

We have done 5 runs of 5000 generations with the following parameters:

- Population size: 100 individuals of initially 200 random bases
- Local mutation rates: $5e-4$ per base
- Rearrangements rates: 0.05 per genome
- Crossover rate: 0.6 to create an offspring from two parents (and so 0.4 from one parent)

Figure 3 summarizes some indicators of evolution: the fitness (total squared error) on the training set and the test set, the size and the number of neurons at the 5000^{th} generation for the best individual of each run. The training fitness ranges from 1.7762 to 2.1736, the test fitness from 1.0499 to 1.9613, the size from 614 characters to 1342 and the number of neurons from 8 to 15. However, beyond the given results, this problem is of low enough dimension to see the structure of the proposed solution.

	Mean	Std. dev.
Learning fitness	1.902	0.156
Test fitness	1.432	0.339
Size	1066.0	276.6
Nb. of neurons	11.2	2.6

Fig. 3. Statistical results and indicators on the 2D Sine Wave problem for the best individual of each simulation after 5000 generations

The figure 4 shows the original points, the final surface and the different Gaussian functions. Each function is equivalent to a neuron. The individual shown is the best individual after 5000 generations.

4.2 The Boston Dataset

In order to be able to compare the performance of the RBF-Gene algorithm with other models, we have performed experiments on the Boston dataset. This dataset is a well-known benchmark that can be downloaded on the UCI Machine Learning Repository [18]. It is a real dataset made by the U.S Census Service concerning housing in the area of Boston.

There are 13 inputs and 1 output. Since the inputs have different dimensionality, we have normalized them before applying the algorithm (mean of 0 and standard deviation of 1). The output is between 0 and 50 and we have kept it unchanged in order to compare our results. There are 506 points in the dataset, and no data is missing.

We have done 25 simulations by set of parameters using different seeds and/or different partitions of the data (keeping a ratio of 5 learning points for 1 validation point). The fitness function used by the algorithm is the mean square error

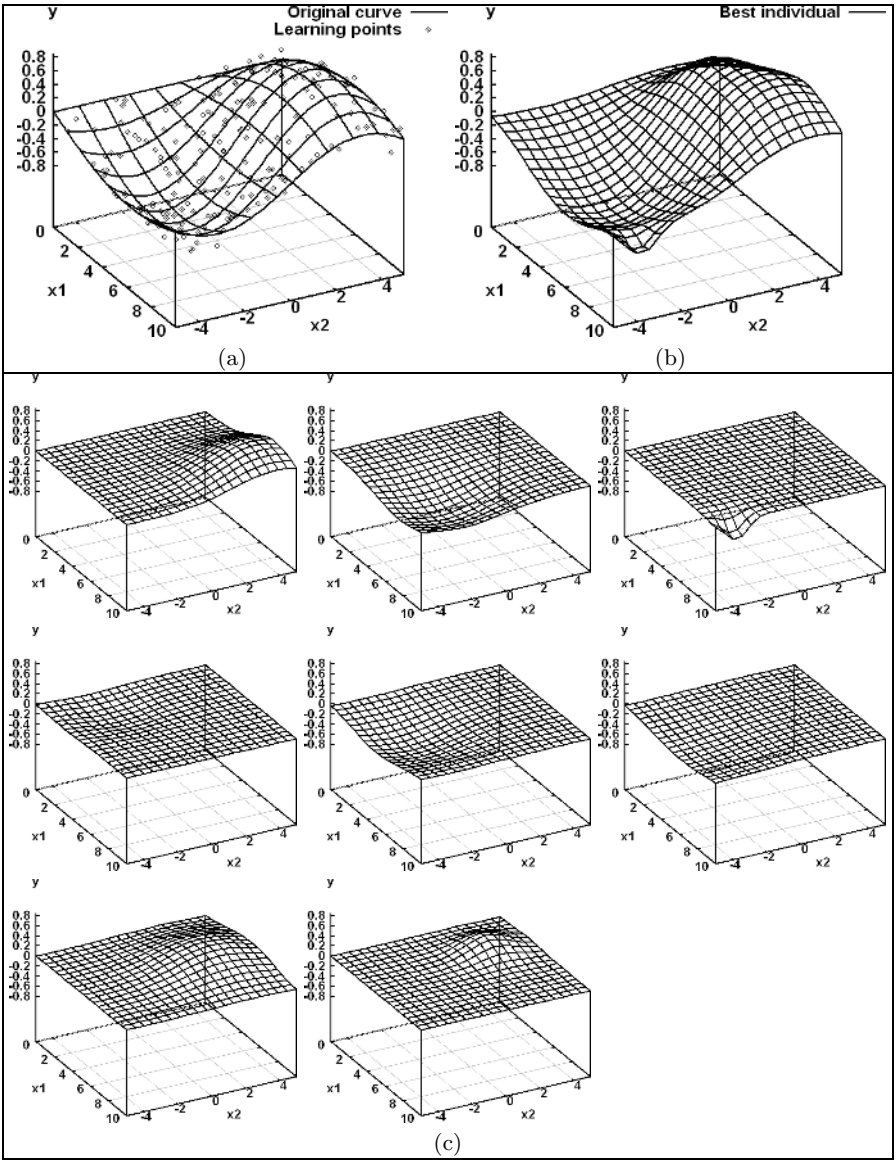

Fig. 4. The target surface and one generated individual after 5000 generations: (a) the desired curve and the training points; (b) one individual from the last generation; (c) the different Gaussian functions of the individual. Note that the noise in the data is partly extracted by the algorithm (kernel nb. 3).

(MSE) on the training set and we compare our results using the MSE on the validation set. The parameters are the same than in the 2D sine wave, save that the local mutation rates is $1e-4$ per base here.

	Mean	Std. dev.	Median
Learning fitness	13.41	3.92	13.06
Validation fitness	16.70	4.59	16.60
Nb. of neurons	29.2	39.2	18.0

Fig. 5. Statistical results and indicators on the Boston dataset for the best individual of each simulation after 5000 generations

	Learning fitness	Validation fitness	Nb. of neurons
Minimum	7.35	10.19	4
Maximum	22.21	27.48	178

Fig. 6. Minimum and maximum values obtained by the best individual of each simulation after 5000 generations

Figure 5 summarizes our results on the dataset. We can compare them with [19] in which different methods are tested. We see that we have similar results to the best of the tested methods. However, since Madigan et al. don't provide enough statistical information to run a statistical test, we cannot make comparative claims with any certainty. Indeed, we have an average MSE of 16.70 ± 4.59 (best individual: 10.19) while the results in [19] range from 14.1 for the GBM two-way method to 25.8 for the Stagewise method.

Figure 6 shows the minimum and maximum value of each of learning fitness, validation fitness and number of neurons. When the standard deviation on the number of kernels is more than the mean, it is because we have a highly skewed distribution. 2 simulations have more than 100 kernels (126 and 178 respectively) while the other ones all have less than 60 kernels.

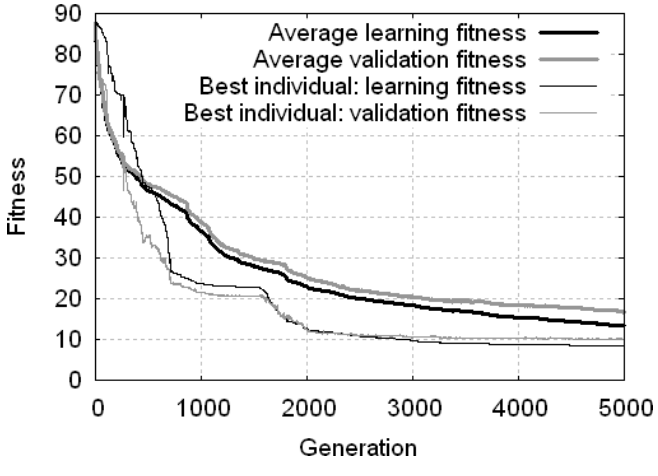

Fig. 7. Average fitness and best individual: the learning fitness is in black and the validation fitness in gray

Figure 7 shows the evolution of the fitness over the course of the runs (average fitness and the evolution of the fitnesses for the individual with the best validation fitness). As we can see, there is no over-learning as the validation fitness is still decreasing and the gap between the learning fitness and the validation fitness remains slight.

Moreover, during evolution, the structure of the genome changes and the number of genes increases progressively from 1 at the first generation to about 30 in the last generation. Of course, while the number of genes increases, so does the genome size (from 200 bases initially to a median size of about 6000 bases) but we also observe that the coding proportion (proportion of coding sequences on the genome) grows from about 20% to 60% without any hard-coded limit to the genome size.

5 Discussion and Future Works

The preliminary results we have obtained with the RBF-Gene algorithm are very encouraging. Quantitatively the algorithm performance is quite good. Qualitatively, the chromosome structure obviously evolves during the evolutionary process thus showing a genuine *simultaneous* evolution of the neural network structure and the scalar parameters.

However, more work has to be done. We would like to study the influence of different parameters such as the mutations rates. A better understanding of the evolution of the genome structure would be quite instructive, especially on the chromosome size, the individual gene size, and the genes' order. This work is in progress; early results suggest that the algorithm is very robust – that is it gives similar results over a wide range of parameters. We are also studying the possibility of using a real code instead of the binary Gray code for the parameter encoding.

References

1. Haykin, S.: Neural Networks - A Comprehensive Foundation. 2nd edn. Prentice Hall (1999)
2. Blanco, A., Delgado, M., Pegalajar, M.: A real-coded genetic algorithm for training recurrent neural networks. Neural Networks **14** (2001) 93–105
3. Kuşçu, I., Thornton, C.: Design of artificial neural networks using genetic algorithms: review and prospect. Technical Report 319, Cognitive and Computing Sciences, University of Sussex, Falmer, Brighton, Sussex, UK (1994)
4. Arotaritei, D., Negoita, M.G.: Optimization of Recurrent NN by GA with Variable Length Genotype. LNAI (Springer-Verlag) **AI 2002** (2002) 681–692
5. Goldberg, D.E.: Genetic Algorithms in Search Optimization and Machine Learning. Addison-Wesley (1989)
6. MacLeod, C., Maxwell, G.M.: Incremental Evolution in ANNs: Neural Nets which Grow. Artificial Intelligence Review **16** (2001) 201–224
7. Barrios, D., Manrique, D., Plaza, R.M., Juan, R.: An Algebraic Model for Generating and Adapting Neural Networks by Means of Optimization Methods. Annals of Mathematics and Artificial Intelligence **33** (2001) 93–111

8. Cliff, D., Harvey, I., Husbands, P.: Incremental evolution of neural network architectures for adaptive behaviour. Technical Report Cognitive Science Research Paper CSRP256, University of Sussex - School of Cognitive and Computing Science, Brighton BN1 9QH, England, UK (1992)
9. Thierens, D.: Non-Redundant Genetic Coding of Neural Networks. In: International Conference on Evolutionary Computation. (1996) 571–575
10. Levenick, J.R.: Inserting introns improves genetic algorithm success rate: Taking a cue from biology. In Belew, R., Booker, L., eds.: Proceedings of the Fourth International Conference on Genetic Algorithms, San Mateo, CA, Morgan Kaufman (1991) 123–127
11. Wu, A.S., Lindsay, R.K.: A survey of intron research in genetics. In Voigt, H.M., Ebeling, W., Ingo, R., Hans-Paul, S., eds.: Parallel Problem Solving From Nature IV. Proceedings of the International Conference on Evolutionary Computation. Volume 1141., Berlin, Germany, Springer-Verlag (1996) 101–110
12. Burke, D.S., De Jong, K.A., Grefenstette, J.J., Ramsey, C.L., Wu, A.S.: Putting more genetics into genetic algorithms. Evolutionary Computation **6** (1998) 387–410
13. Whitley, D.L., Barbulescu, L., Watson, J.P.: Local Search and High Precision Gray codes: convergence Results and Neighborhoods. (2000)
14. Knibbe, C., Beslon, G., Lefort, V., Chaudier, F., Fayard, J.: Self-adaptation of Genome Size in Artificial Organisms. In Capcarrere, M.S., al., eds.: Advances in Artificial Life, Proceedings of the 8th European Conference ECAL 2005. Volume 3630 of Lecture Note in Artificial Life (LNAI)., Springer-Verlag (2005) 423–432
15. Lefort, V., Knibbe, C., Beslon, G., Favrel, J.: The RBF-Gene Model. In: Proceedings of GECCO 04, Late Breaking Papers. (2004)
16. Lefort, V., Knibbe, C., Beslon, G., Favrel, J.: A bio-inspired genetic algorithm with a self-organizing genome: The RBF-Gene Model. In: Proceedings of GECCO 04. Lecture Notes in Computer Science, Springer-Verlag (2004)
17. Orr, M.J.L., Hallam, J., Takezawa, K., Murray, A.F., Ninomiya, S., Oide, M., Leonard, T.: Combining Regression Trees and Radial Basis Function Networks. International Journal of Neural Systems **10** (2000) 453–465
18. Blake, C., Merz, C.: UCI repository of machine learning databases (1998)
19. Madigan, D., Ridgeway, G.: Discussion of "Least Angle Regression" by Efron et al. The Annals of Statistics **32** (2004) 465–469