F21BC Biologically Inspired Computation Coursework 1

Auguste Bourgois
Wanjun Feng

November 14, 2016

1 Introduction

This report aims at presenting our understanding of main tools used in Biologically Inspired Computation and the results of our work.

Both parts of the report focus on computing the global optimum of a given mathematical function by using BIC techniques. The first part uses a Multi-Layer Perceptron (MLP) to approximate the function, this Neural Network (NN) being optimized by a Genetic Algorithm (GA); then a second GA is used to approximate the input giving the global optimum of the function. The second part only uses a GA on the real function to find the optimum.

2 Artificial Neural Network and Genetic Algorithms - Part A

2.1 Context

The technique developed in this part is interesting for a wide range of applications. Indeed, one cannot always define a precise mathematical model of a process (eg. the dynamical model of a robot). Using a NN that can learn the behaviour of that process is then an interesting solution to model it. The learning part is performed using a GA. Several methods of selection, mutation and cross-over have been implemented, and they will be explained further. Once a precise enough approximation of the target function has been reached, another GA is used to determine the approximate optimum of the latter.

2.2 Multi-layer perceptron

The MLP architecture has been chosen for the neural network. It can indeed be considered as a universal function approximator, as explained in [Haykin, 2009], and is easy to implement compared to other NNs, given the fact that the MLP is a feed-forward only NN. The following hypothesis have been done to implement it:

- Each layer is fully connected to its following neighbour, ie each neuron from the latter receives the output of the previous layer as an input.

- Each neuron computes an output given its inputs composed of the outputs of all the neurons in the previous layer $\mathbf{x}_{i-1}$ and a bias $b_{i,j}$, a vector of weights $\mathbf{w}_{i,j}$, and an activation function f_i, where i is the layer index and j the neuron index in a given layer. $\mathbf{w}_{i,j}$ contains the weights the current neuron applies to each of the inputs in $\mathbf{x}_{i-1}$.
- The structure of the NN is not supposed to be modified during a simulation.

As seen in class, the output $\mathbf{y}$ of a given layer i can be formally expressed as follows.

$$\mathbf{y}_i = f_i\left(\mathbf{W}_i \cdot \mathbf{x}_{i-1} + \mathbf{b}_i\right) \tag{1}$$

where $\mathbf{W}_i = \begin{bmatrix} \mathbf{w}_{i,1} \\ \vdots \\ \mathbf{w}_{i,j} \\ \vdots \\ \mathbf{w}_{i,n} \end{bmatrix}$ and $\mathbf{b}_i = \begin{bmatrix} b_{i,1} \\ \vdots \\ b_{i,n} \end{bmatrix}$.

The output of the entire NN can't easily be computed recursively given (1). Moreover, after reading [Trenn, 2008], we understood that the structure of the NN was very important. Indeed, if the number of neurons is too low, it cannot approximate complicated functions (multi-modal ones for example). But if it is too high, then having it evolve and converge towards a good solution is very long because there are much more parameters to change. It could be interesting as an improvement for our algorithm to evolve the NN towards a structure that would fit the function complexity.

2.3 Optimizing the MLP

In order to choose the correct parameters for the NN, multiple methods exist. One of the most commonly used for MLPs is the back-propagation algorithm. However, a GA can also be used successfully.

2.3.1 Problem encoding

To evolve the NN to get a good approximation of a target function, one has to choose an encoding, or chromosome, as a work material for the GA. In other word, one has to choose the parameters, or genes, that the GA may evolve.

As the behaviour of the NN entirely depends on its weights and biases, these must be chosen as the genes of a chromosome. So a chromosome contains in a way all the weights and the biases of the NN. In practice, a chromosome is organized as in (2). The weights matrices and bias vectors are concatenated and serialized in a chromosome (row vector).

$$\mathbf{C} = \begin{bmatrix} \mathbf{w}_{1,1} & b_{1,1} & \cdots & \mathbf{w}_{1,n} & b_{1,n} & \cdots & \mathbf{w}_{i,j} & b_{i,j} & \cdots \end{bmatrix} \tag{2}$$

A population of chromosomes is then created, and can be manipulated by the GA. The initialization of the population influences the performance of the GA. We found out that the wider the range of the biases was, the better the NN could approximate a multi-modal function. So while the weights are initialized randomly between $[-1, 1]$, the biases get their initial values from the whole input space of the target function.

2.3.2 Fitness evaluation

To evaluate the fitness of a chromosome, ie of the MLP it encodes, one must compute its output $\tilde{\mathbf{y}}$. Then, one can define the fitness of this chromosome as a function of the errors of the NN. Our program uses the following formula, where N is the dimension of the output.

$$f = \frac{1}{\sum_{i=1}^{N} \left(\tilde{\mathrm{y}}_i - \mathrm{y}_i\right)^2} \tag{3}$$

Using the squared error to compute the fitness ensures that the worst ones will be corrected first. It is important to notice that the higher f, the better the chromosome. After initializing the population (10000 individuals for example), the fitness of each chromosome is evaluated. The best ones (around 100 individuals) are chosen to be part of the real initial population of the algorithm.

2.3.3 Evolving the population

We have implemented and tried a few methods and algorithms to evolve the initial population. Different types of mutations have been tested :

- *Vector mutation method* : addition of normal random noise to each gene of the chromosome. We found out that the higher the range of the noise was, the lesser its efficiency was. However, a higher noise in a chromosome mutation can help to jump out of a local optimum of the search space.
- *Vector mutation method with low noise* works identically, but as the added noise is lower, this kind of mutation helps finding a local optimum (may it be the global one).
- *n-genes mutation method* tends to be more efficient than the two latter when the algorithm is somehow close to the solution. Indeed, as fewer changes are made in the selected chromosomes, their off-springs keep some of their genes, while others are slightly changed.
- *n-genes mutation method with very low noise* almost always improves the selected chromosome, which is useful at the end of the search algorithm, when the other methods are useless. However, the improvement are consequently very low, which means that it will take a very long time before getting significantly better results.

These crossovers have also been tested :

- *Uniform random crossover method* requires two parents. Their child inherits some of the genes of the first parents and others from the second one. It can help a lot at the beginning of the algorithm to eliminate the worst individual of the population. However, it doesn't give excellent results on the long term compared to the following ones, because it very often introduces too big changes in a chromosome.
- The *line crossover method* seems to be the most efficient one. Indeed, it often creates a best chromosome. It requires two parents : one of them is used as a reference, to which the difference between the parents is added.
- We also tested the *box crossover method*, which is an evolution of the latter, but it doesn't seems as efficient as the line method.

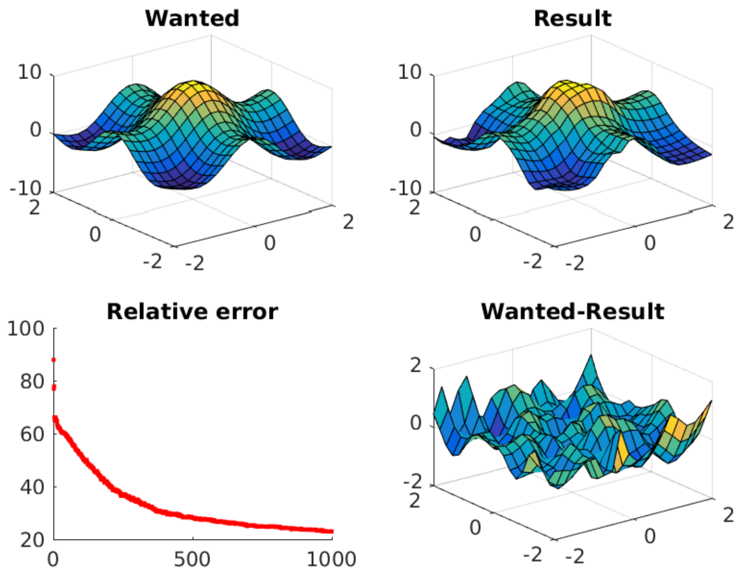

Figure 1: Result the combined genetic operators

We tested each method alone, and we reached the following conclusion :

The crossover methods are useful to make the population converge quickly towards a good individual. However, its main drawback is the reduction of diversity. Very quickly, the population is composed of the same individuals.

This is where mutations intervene. They introduce noise in the population, which can eventually produce a better individual.

The use of both evolution operators is interesting, because their advantages are complementary whereas their drawbacks tend to cancel each others. That why our algorithm uses all these methods together. Some graphs illustrating these results have been included as annexes. The function to approximate was the following one.

$$g(x, y) = 5 \cdot \sin(x + y) \tag{4}$$

Two selection methods have been implemented : *roulette wheel* and *tournament*. The latter performs better than the other, because it limits the loss of diversity in the population.

Concerning the updating choice, a variant of the generational-with-elitism based update algorithm has been implemented, thanks to [Mitchell, 1996]. This idea comes from Once the population of children has been created, it is mixed with the old one, and the new population is sorted with respect to the fitness of each individual. The n best ones are part of the new population, where n is the size of the initial population. The steady state algorithm has also been tested, but it gave worse results, as it is much slower in terms of replacement of individuals.

2.3.4 Finding the global optimum

After having created the best NN for the function approximation, another GA has been used to find the maximum of the approximated function. The studied function is quite simple, which means that with a relatively small number of individual and a simple hill-climbing search method, the goal can be easily reached. However, the implemented search method is based on a *Vector*

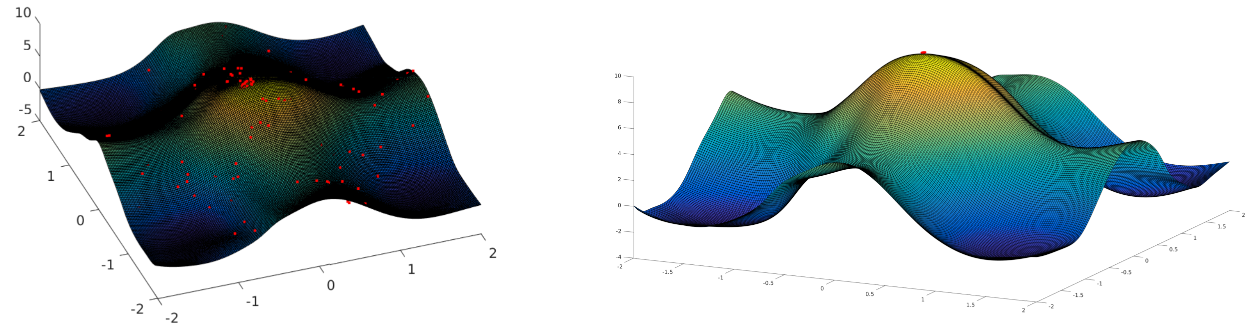

Figure 2: Initialization and results of the search algorithm with one best NN

mutation method with low noise. The population size is important for these two main reasons : if it is too low, the population can converge, despite the relative performance of the mutation algorithm, towards a local maximum; and even if it is a global maximum, the algorithm is slower. If the population size is too large, then the program requires more computational power, which also leads too a slowdown.

3 Genetic algorithms - Part B

In this part, the COCO software has been used to improve our GA. This software tests the optimizer on a bunch of specific functions. Each of them enables to test one aspect of the GA : its robustness against plateaux, how it behaves in presence of multi-modal functions, if it can takes advantages of some of the functions characteristics like symmetry ...

The tested GA is very simple: it is based on two vector mutation methods (low noise and very low noise). The following graph shows its performance against the whole lot of 2D functions. The average error has been calculated with 15 instances of each function.

The GA is extremely efficient on the sphere function. The number of iteration is low (20). This can be interpreted as the maximum convergence rate of the algorithm. The third function shows that our GA is quite robust with respect to multi-modality. The seventh function also shows that it is robust with respect to plateaux.

A higher dimensionality leads to a larger search space. Then, we tried to compute the average error in more dimensions.

The figure 4 shows that our algorithm could still be improved. Indeed, thanks to [Finck et al., 2014], we could figure out that other GAs operators would perform better than ours if they could extract some informations about the explored function.

4 Conclusion

Thanks to this coursework, we have discovered a lot of algorithms and methods used in BIC. We have implemented a working MLP, and different GAs that were able to evolve different kinds of chromosomes. We have been confronted to a lot of issues, like encoding a problem or evolving a chromosome given its encoding. We have been able to test a lot of existent solutions to these

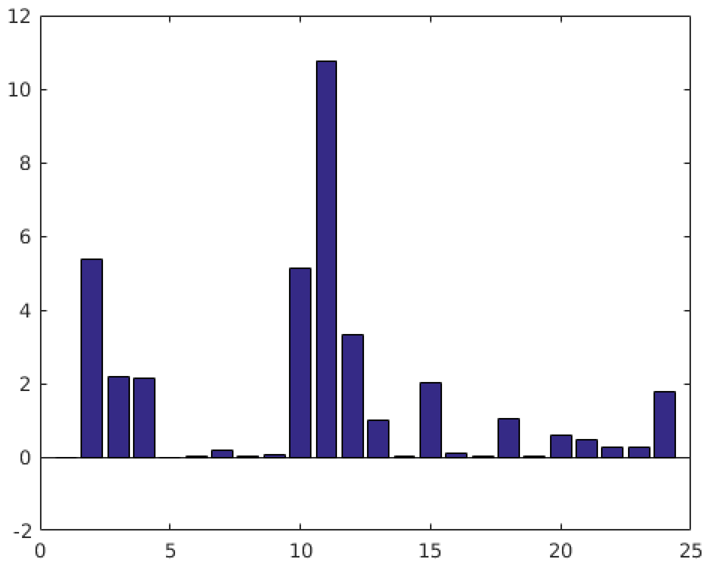

Figure 3: Performance of the GA using COCO

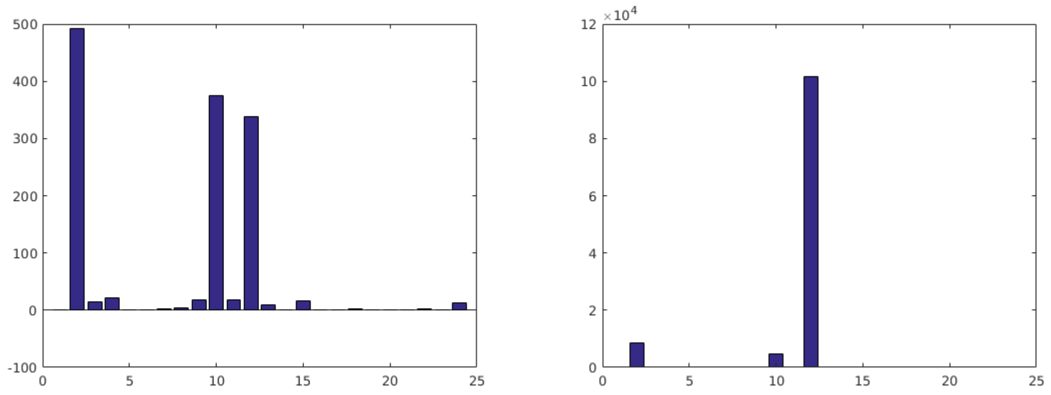

Figure 4: 5D and 10D results

issues. However, we only explored a very little part of the BIC methods, and we could have done much more experimentations with a bit more time.

5 Annexes

Each figure is composed of four graphs. The top left one is the function to approximate, the top right one is the output of the NN after 1000 iterations of the algorithm, the bottom left one is the relative error of the output of the NN, and the bottom right one is the error between the desired output and the NN's one.

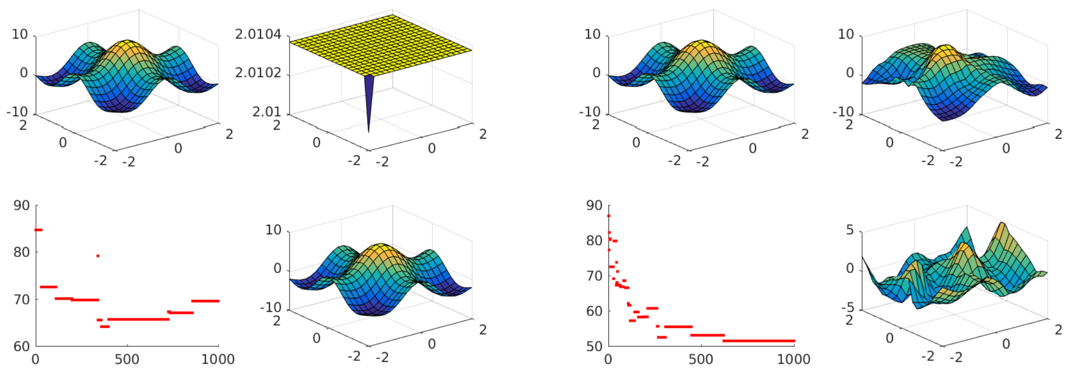

Figure 5: *Vector mutation method and Vector mutation method with low noise*

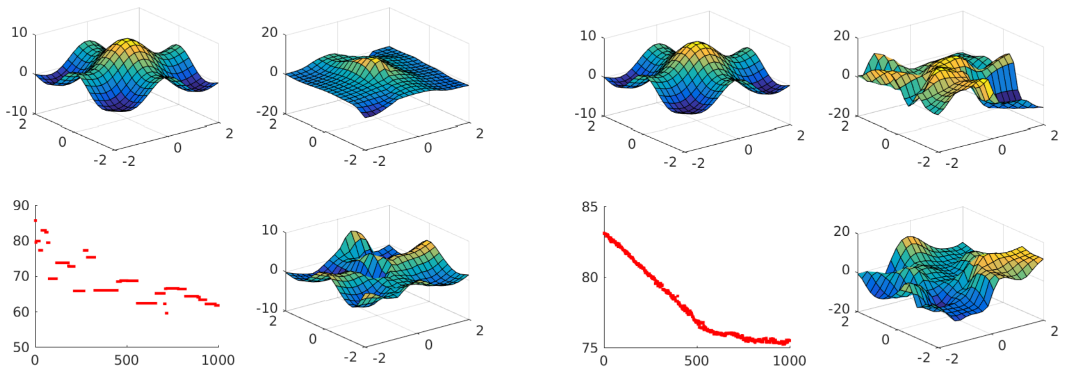

Figure 6: *n-genes mutation method and n-genes mutation method with very low noise*

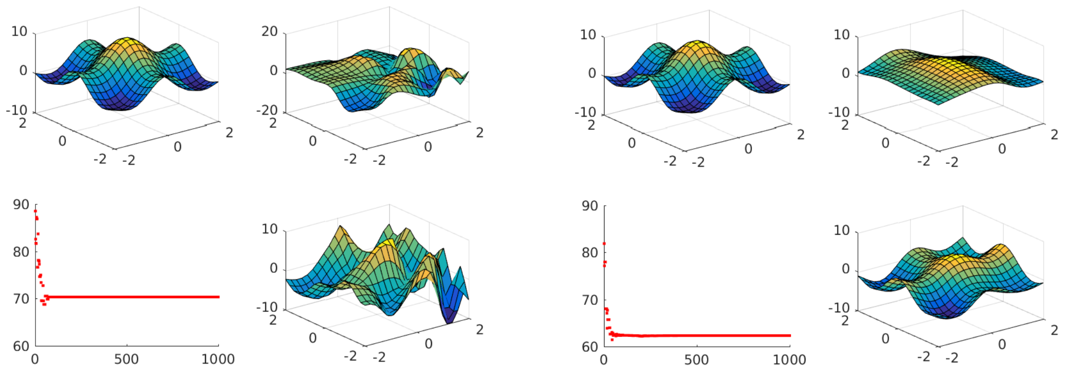

Figure 7: *Uniform random crossover method and Line crossover method*

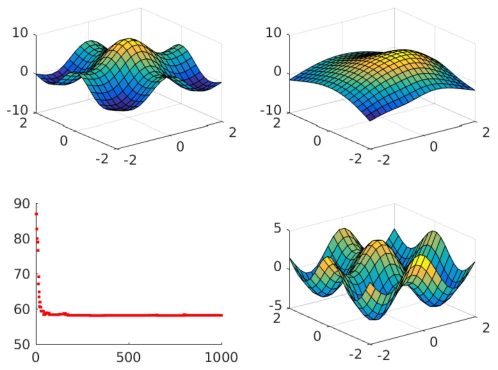

Figure 8: *Box crossover method*

References

[Haykin, 2009] Haykin, S. (2009). *Neural networks and learning machines.* 3rd ed. Upper Saddle River, N.J.: Pearson, pp.196-201.

[Mitchell, 1996] Mitchell, M. (1996). *An introduction to genetic algorithms.* 1st ed. Cambridge, Mass.: MIT Press, pp.124-128.

[Finck et al., 2014] Finck, S., Hansen, N., Ros, R. and Auger, A. (2014). *Real-Parameter Black-Box Optimization Benchmarking 2010: Presentation of the Noiseless Functions.* [online] Coco.lri.fr. Available at: http://coco.lri.fr/downloads/download15.01/bbobdocfunctions.pdf [Accessed 14 Nov. 2016].

[Trenn, 2008] Trenn, S. (2008). *Multilayer Perceptrons: Approximation Order and Necessary Number of Hidden Units.* IEEE Transactions on Neural Networks, 19(5), pp.836-844.