Biologically Inspired Computation: Coursework 1

Heriot Watt University

Fantine Chabernaud
Giammarco Quaglia

14/11/2016

Introduction to Course Work 1

The coursework is divided into two parts. The first chapter *MLP ANN with GA* deals with a Multi-Layer Perceptron Artifical Neural Network trained in a supervised way with the use of a genetic algorithm to simulate a mapping between inputs and outputs. Much effort has been put in the analysis of the case in which there is only one input node because this scenario allows some useful visualizations. The code has been implemented in python 2.7, using jupyter notebook many comments are provided in it.

The second chapter of the coursework deals with evolutionary algorithm. The code, written in MATLAB, is a heuristic algorithm that manage a population following the pattern of a biological evolution. In this part, we will apply this method to an algorithm that looks for the global minimum of functions in different dimensions.

Both the codes can be found at `https://github.com/GQuaglia/CW1_FCGQ`

Chapter 1

MLP ANN with GA

Papers by Hornik [1], Funahashi [2] and Castro et al. [3] established that multilayer feedforward networks are universal approximators, not limited to approximate only some special classes of functions. The architecture chosen for the ANN is the simple 3 layers structure: one for the inputs, one for the hidden nodes and one for the output that has only one node, infact all the functions the ANN try to simulate go from an input vector $\mathbf{x}$ to one value of the output y (f($\mathbf{x}$): $\mathbf{x} \mapsto y$). The reason for the choice of such a simple architecture is based on the fact that "multilayer feedforward networks with as few as one hidden layer are capable of approximating any function to any desired degree of accuracy, provided that sufficiently many units are available", as stated in [1]. A scheme of the ANN is shown in Figure 1.1.
introduce the figure (maybe by speaking about the model we used to create this method which is the brain: nodes are neurons and links are axones...)

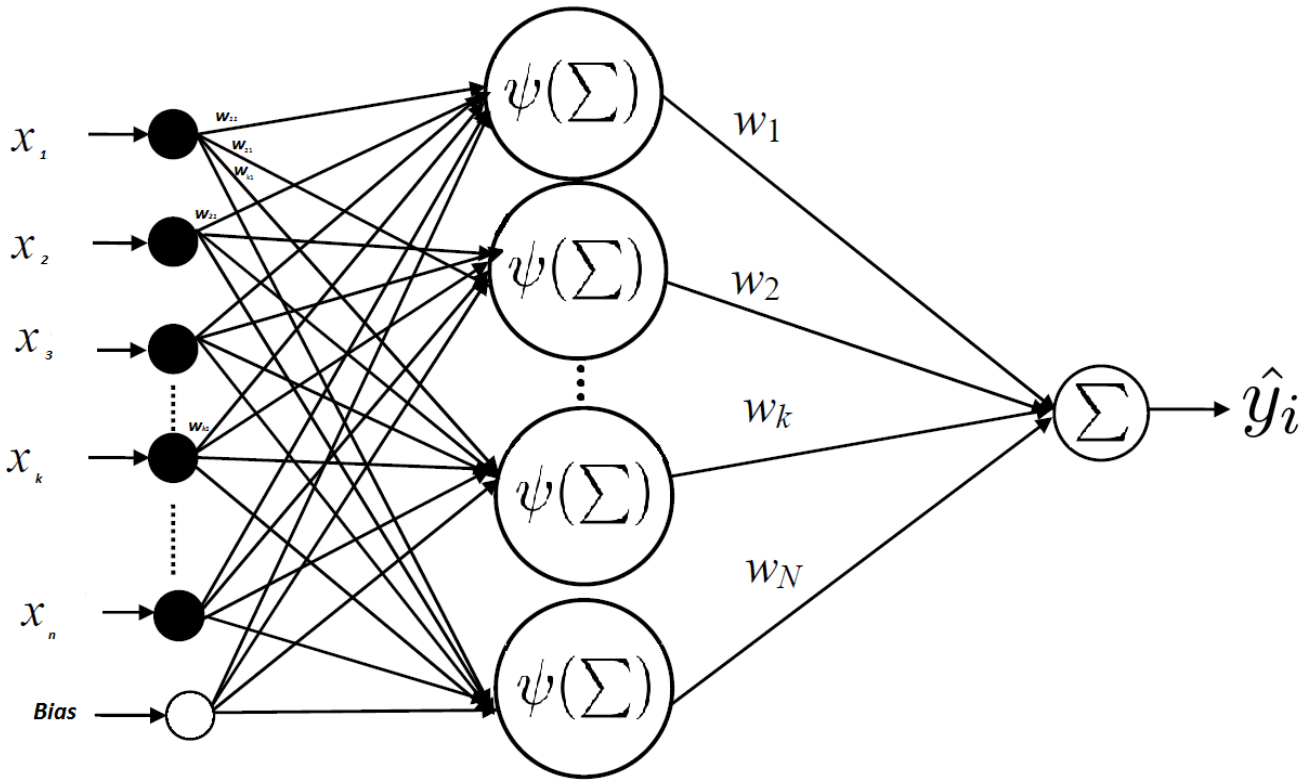


Figure 1.1: Scheme of the ANN with three layers

The strength of the connection between two nodes is w_{ij}, which stands for the weight form the node j

of a layer to node i of the next one. An activation function ψ is applied to the sum of weighted inputs and so an output $\hat{y}$ is generated: $\hat{y_i} = \psi(\sum w_{ij}x_j)$, then this last one will play the role of the input for the next layer. The ANN is initialised defining the dimension of the problem, that means setting the number of inputs, and inserting the desired number of nodes in the hidden layer. At first all the nodes start having a value equal to 1 and all the weights are drawn from a normal distribution *N(0,3)*, then these values are modified. The fitness of each solution has been evaluated with the following error function as defined in [4]:

$$ERR = \frac{1}{2}\sum_{i=1}^{N}(\hat{y_i} - y_i)^2 \tag{1.1}$$

with N being the lenght of the training dataset, $\hat{y_i}$ representing the output, that is estimated value of the function f(x) and y_i standing for the desired output.

The choice of the domain $[-1, 1]$ for the inputs has been driven by Enăchescu [4], in which the author uses the interval $[0, 1]$, but in the following experiments also [-1,0] was used to investigate the results for negative inputs.

1.1 First genetic algorithm

Simple GA without bias

In this subsection one input has been considered without bias.

The target function is the following one:

$$y = f(x) = \frac{1}{x^2 + 0.1} + sin(x) + 10x \tag{1.2}$$

It is complex enough to have non trivial behaviour in the interval $[-1, 1]$. Infact to perform a supervised learning a training dataset was created taking 1000 numbers between 1 and -1 and the corresponding values of y computed through $f(x)$ (in the code $my\_function()$).

In the first GA (in the code $SearchBestSet()$) the chromosome is basically the set of weights: at each generation the set of weights is completely renewed drawing each one from a normal distribution, so $w_{ij} \sim N(0, \sigma^2)$, with zero mean and variable standard deviation drawn itself from a uniform distribution $U[1, 10]$. The use of all these probability distributions aims at reaching a certain diversity in the population. In order to make meaningful comparisons, a seed has been set to create the first ensemble of weights of the first ANN, drawn from a normal distribution $w_{ij} \sim N(0, 3)$ (in the code $first\_initialise()$), then it has been evolved with the simple GA. The parameters of interest in this first analysis are: number of iterations, number of hidden nodes, set of weights ($\{\boldsymbol{w}\}$) and σ of the normal from which they are drawn (σ).

It's like having the following chromosome

N. iterations	N. hidden nodes	Value of σ	set of weights

in which the number of iterations, of hidden nodes and the value of the standard deviation (σ) are fixed, infact the following configuration were studied.

Configurations - logistic				
Subfigure	N. Iterations	N. Hidden Nodes	Value of σ	running time (")
A	1000	3	3	23
B	1000	3	$\sim \boldsymbol{U}[1, 10]$	24
C	1000	20	$\sim \boldsymbol{U}[1, 10]$	109
D	10000	20	$\sim \boldsymbol{U}[1, 10]$	803

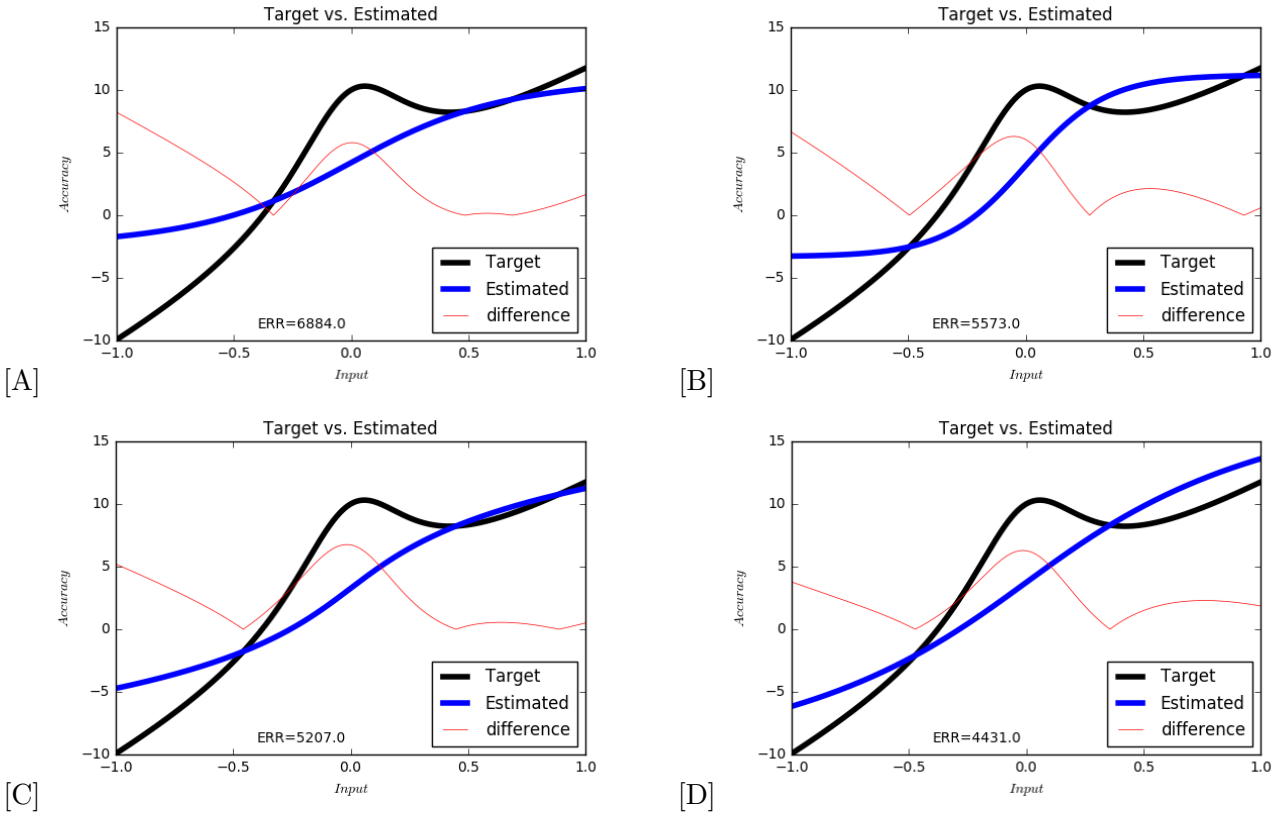


Figure 1.2: Plots for the 1-D experiments - logistic activation function

After all the iterations the ANN with the best fitness was used to produce a plot in which both the target function and the one learnt from the ANN are drawn in order to appreciate the difference betweeen them.

As expected the estimated function is a poor approximation in any case, but reasonably the error decreases as the number of hidden nodes and iterations increases in agreement with the theorem regarding the link between number of hidden nodes and accuracy (see introduction) and also with logic: to a bigger number of iterations (then also of weights sets) and to a wider range of possible values for σ corresponds a higher probability to have a better configuration.

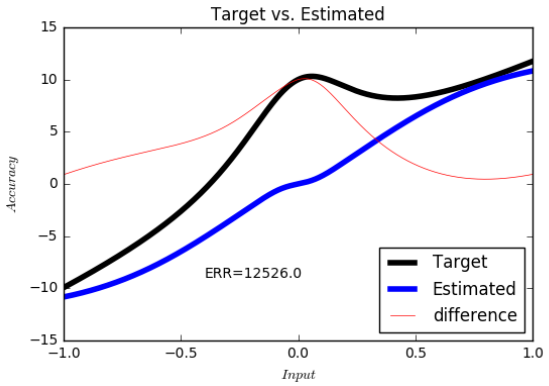
The last two configurations were used again to produce the following plots (Fig.1.3), with the activation function being now the hyperbolic tangent.

Configurations - tanh				
Subfigure	N. Iterations	N. Hidden Nodes	Value of σ	running time (")
E	1000	20	$\sim \boldsymbol{U}[1, 10]$	860
F	10000	20	$\sim \boldsymbol{U}[1, 10]$	1166

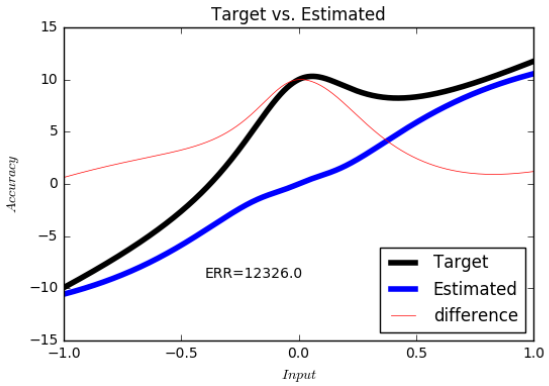
Now the errors are bigger so it seems that the logistic function outperforms the hyperbolic tangent, though the second one catches better the behaviour with the central bump.

Simple GA with bias

Now let's examine the casewith one extra node in the input layer for the bias, an increase of accuracy of the estimation is expected. The choie of putting the bias node in the input layer has been driven by the work of Enăchescu [4]. The configurations exploited are again the two best ones. Tables and plots are shown.



[E]



[F]

Figure 1.3: Plots for the 1-D experiments - hyperbolic tangent activation function

Configurations - logistic - with bias				
Subfigure	N. Iterations	N. Hidden Nodes	Value of σ	running time (")
G	1000	20	$\sim \boldsymbol{U}[1, 10]$	150
H	10000	20	$\sim \boldsymbol{U}[1, 10]$	1694
Configurations - tanh - with bias				
Subfigure	N. Iterations	N. Hidden Nodes	Value of σ	running time (")
I	1000	20	$\sim \boldsymbol{U}[1, 10]$	127
J	10000	20	$\sim \boldsymbol{U}[1, 10]$	1469

The results both for tanh and logistic are quite unexpected, there are no significant improvings or worsenings with the use of the bias, moreover there is the confirmation, given also by the results without bias, that increasing the number of iterations by a factor of 10 doesn't lead to much better performances, though it is much more computationally demanding as can be seen from the running times. These results suggest to focus not on the size of the population but rather on the variation of the other parameters such as the number of nodes, that's the starting point of the next genetic algorithm.

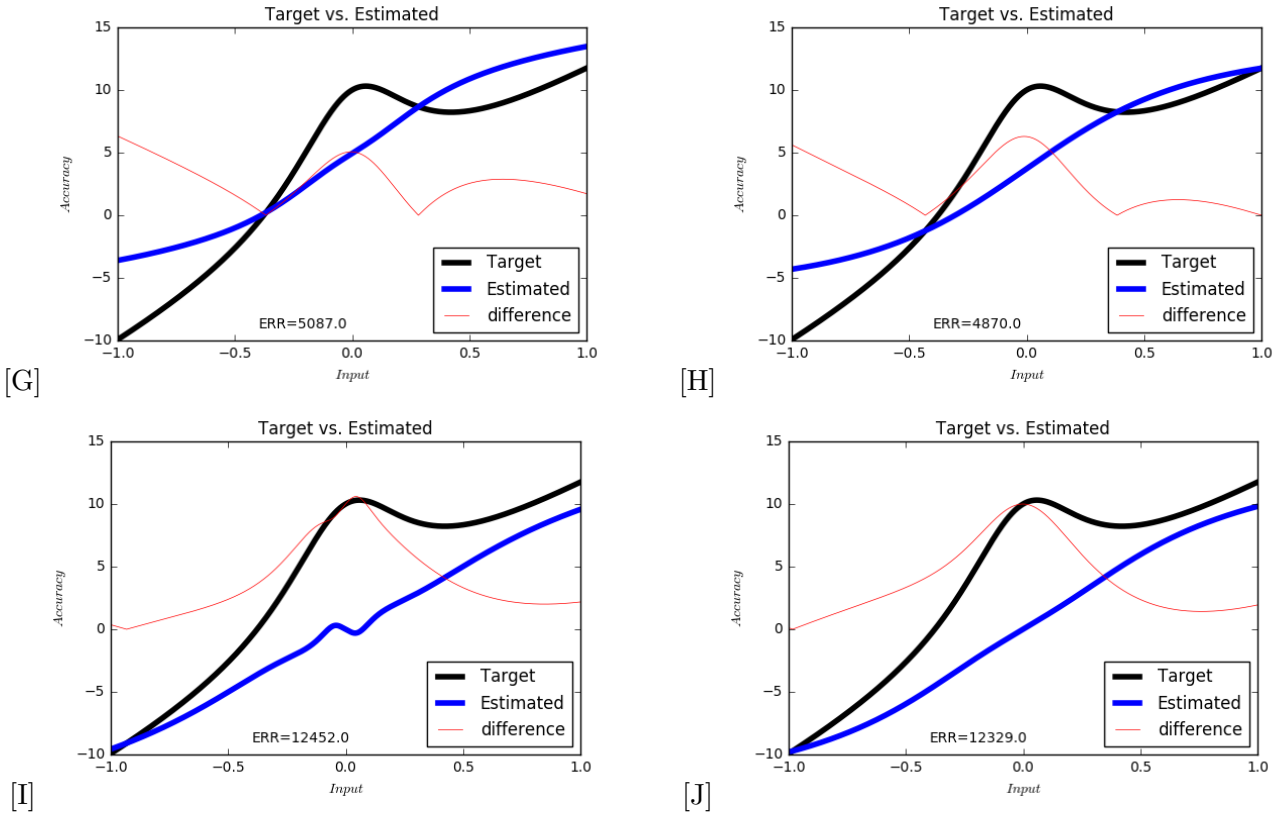


Figure 1.4: Plots for the 1-D experiments - with bias

1.2 Complex GA

In this section, continuing to use the bias node, a more complex Genetic Algorithm is implemented.

The code of this GA is divided in two parts: creation of the initial population and evolution of it. In the first part (*create_initial_population*()) an initial population is created, for each individual the number of hidden nodes is drawn from a $U[3, 10]$, each weight from a $\sim N[0, \sigma^2]$ with $\sigma \sim U[1, 10]$ and the activation function is chosen between logistic and tanh (equal probability). It's like having the following chromosome.

Activ. function	N. hidden nodes	Set of weights

Each individual is evaluated and the fitness is stored to have a dictionary containing the initial population which can be sorted by the fitness (computed as previously as the total error). In the second part the sorted list of individuals is divided into its four quarters to evolve the population. The first quarter (the 25% fittest individuals) is saved and will be part of the new population not to lose the best ones. Always working with the first quarter a part of the new population is created by means of a crossover between 2 parents taken among the best quarter, the parents will be: first and second, second and third and so on, assigning to the new individual the activation function from the i-th parent and the set of weights of i-th + 1 parent. Now the chromosome is the following.

Activ. function	N. hidden nodes + Set of weights

Another quarter of the new population is created through mutation of the second best 25% of the old population: a new set of weights is generated leaving the same activation function and number of hidden nodes.

Activ. function + N. hidden nodes	Set of weights

The last quarter of the new population is completely fresh with a number of hidden nodes from $U[3, 20]$ instead of U[3,10] and weights from a $N[0, \sigma^2]$ with $\sigma \sim U[1, 30]$ instead of $U[1, 10]$ allowing the presence of configurations with more hidden nodes and greater weights (absolute value) in order to get higher diversity.

This method applied with a population of size equal to 100 and 1000, though having a costly running time, leads to much better approximations, as can be seen in the following tables and plots.

Population size = 100			
Subfigure	N. generations	Error	running time (")
A	10	5375	47
B	50	5822	221
C	100	3108	468

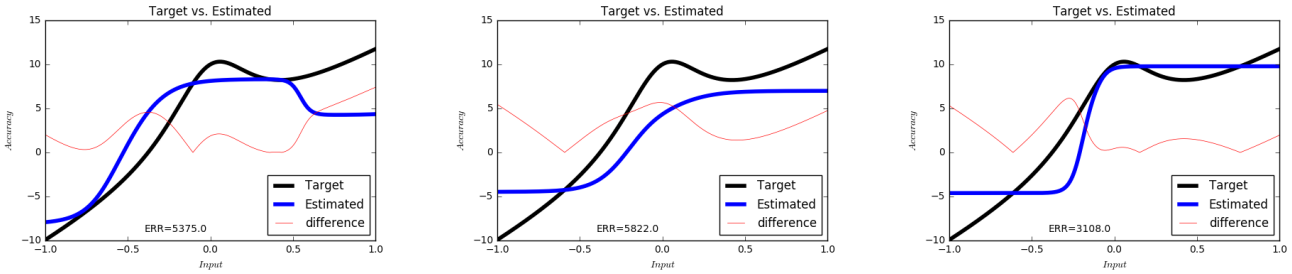


Figure 1.5: Plots for complex GA with population size = 100 and increasing number of generations 10, 50, 100

Population size = 1000			
Subfigure	N. generations	Error	running time (")
A	10	3221	598
B	50	3650	2879
C	100	1088	6277

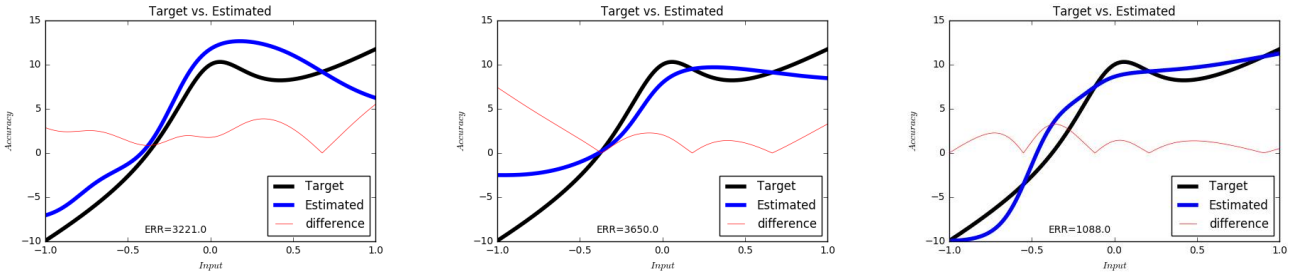


Figure 1.6: Plots for complex GA with population size = 1000 and increasing number of generations 10, 50, 100

Higher dimensions

The code works also with higher dimensions, but the focus was on 1-dimensional case because it's difficult to visualize the functions with greater dimensions. From dimension 2 on, the code creates the training set using the sphere function. For the general case and for the 2 dimensional case the formulas are:

$$y = \sum_{i=1}^{Dim} x_i^2 \qquad \text{and} \qquad z = x^2 + y^2.$$

With the population size equal to 100 three different dimensions were considered: 2D, 4D, 10D.

Population size = 100 - Dimension = 2			
Subfigure	N. generations	Error	running time (")
A	10	125.36254767	54
B	30	113.64850707	180
C	50	93.18026862	279

Population size = 100 - Dimension = 4			
Subfigure	N. generations	Error	running time (")
A	10	363.79168651	64
B	30	256.71140347	190
C	50	231.39673593	286

Population size = 100 - Dimension = 10			
Subfigure	N. generations	Error	running time (")
A	10	1558.4235767	113
B	30	1579.35682098	240
C	50	1210.377275	402

As expected for most of the cases the bigger the dimension the larger the error and this decrease as long as the number generations increases.

Chapter 2

Evolutionary algorithm using COCO

2.1 Evolutionary algorithm with hillclimbing

Principles

Evolutionary algorithms are inspired from biological evolution mechanisms. It follows a general pattern. Firstly we consider a population of individuals. This population may contains our solutions. In order to classify all individuals from the best to the worst solution, we call the fitness function. Knowing what our goal is, the fitness calculate the error between the expected result and the one calculated. According to the fitness of each individual, we can select the best ones. Two parents will be selected to create a child by recombination of their genome. To one parent, we can apply a mutation in order to create a child. Those two methods create more diversity among the population. Step by step childs replace the old population. This processus is iterated untill the solution is found. A limit of iterations can be set in order to fasten the results.
Hillclimbing is an optimization technique used for local search. Iteratively, the algorithm will looks for the best solution by improving itself. It finds a solution and looks for a best one just next to it. It runs untill it finds the best solution around. This method is easy to use, that's why it's implemented in the evolutionary algorithm.

Applied to programmation

From the COCO plateform, we get a basic program "MY_OPTIMIZER" which take a population and test its individuals to keep the best one. The programm "exampleexperiment" run "MY_OPTIMIZER" with 24 functions and 6 different dimensions [2, 3, 5, 10, 20, 40]. In order to realise an evolutionary algorithm, we add two functions to "MY_OPTIMIZER": the mutation and the crossover. The mutation takes the best parent and adds a small noise. Then the newly created child is evaluated and his fitness is compared to its parent's one. The best individual is kept. Meanwhile, the crossover takes the two best individuals of the population according to their fitness. These parents compose a child by combining their genes. So the child has half of each parents instances. Knowing that each function can be improve by complexifying them.

2.2 Discussion

The whole program is implemented on Matlab environment. A population is initialized and fitnessed using the Matlab function "feval()". The results are sorted from the best to the worst. The two best parents are called by the crossover function in order to create a child. The fitness of the child is compared with its best parent. If the child is better, it is kept as the best individual, otherwise it is discarded. The best individual is then mutated. Again its fitness is compared with the best one . The best individual is

kept and become the best solution. There are many possible configurations by changing the population size and using either a mutation or a crossover function or both. When the population size is increased by a factor

$$10^4$$

, the error is small. As we can see at the figure 2.1 where the size was set at

$$10^4$$

chromosomes. The error is represented by the number "fbest-ftarget" whith "fbest" the best solution from the current population and "ftarget" the global minimum of the

$$19^t h$$

function.

```
f19 in 2-D, instance 1: FEs=20004 with 0 restarts, fbest-ftarget=5.4839e-04, elapsed time [h]: 0.00
f19 in 2-D, instance 2: FEs=20002 with 0 restarts, fbest-ftarget=3.4567e-03, elapsed time [h]: 0.00
f19 in 2-D, instance 3: FEs=20002 with 0 restarts, fbest-ftarget=1.4623e-03, elapsed time [h]: 0.00
f19 in 2-D, instance 4: FEs=20002 with 0 restarts, fbest-ftarget=1.1807e-03, elapsed time [h]: 0.00
f19 in 2-D, instance 5: FEs=20004 with 0 restarts, fbest-ftarget=6.2050e-03, elapsed time [h]: 0.00
f19 in 2-D, instance 41: FEs=20004 with 0 restarts, fbest-ftarget=8.0020e-03, elapsed time [h]: 0.00
f19 in 2-D, instance 42: FEs=20004 with 0 restarts, fbest-ftarget=3.4509e-03, elapsed time [h]: 0.00
f19 in 2-D, instance 43: FEs=20004 with 0 restarts, fbest-ftarget=2.1058e-03, elapsed time [h]: 0.00
f19 in 2-D, instance 44: FEs=20004 with 0 restarts, fbest-ftarget=1.7648e-03, elapsed time [h]: 0.00
f19 in 2-D, instance 45: FEs=20004 with 0 restarts, fbest-ftarget=1.8888e-04, elapsed time [h]: 0.00
f19 in 2-D, instance 46: FEs=20004 with 0 restarts, fbest-ftarget=4.5836e-05, elapsed time [h]: 0.00
f19 in 2-D, instance 47: FEs=20004 with 0 restarts, fbest-ftarget=9.3799e-04, elapsed time [h]: 0.00
f19 in 2-D, instance 48: FEs=20002 with 0 restarts, fbest-ftarget=3.0865e-05, elapsed time [h]: 0.00
f19 in 2-D, instance 49: FEs=20002 with 0 restarts, fbest-ftarget=4.2230e-04, elapsed time [h]: 0.00
f19 in 2-D, instance 50: FEs=20004 with 0 restarts, fbest-ftarget=4.7055e-04, elapsed time [h]: 0.00
```

Figure 2.1: Result for the function number 19 with a big population

However, the smallest population with a size of 1 time the dimension obtains greater errors as shown on the figure 2.2:

```
f19 in 2-D, instance 1: FEs=4 with 0 restarts, fbest-ftarget=9.1702e+00, elapsed time [h]: 0.00
f19 in 2-D, instance 2: FEs=4 with 0 restarts, fbest-ftarget=8.6330e+01, elapsed time [h]: 0.00
f19 in 2-D, instance 3: FEs=4 with 0 restarts, fbest-ftarget=8.8282e+00, elapsed time [h]: 0.00
f19 in 2-D, instance 4: FEs=4 with 0 restarts, fbest-ftarget=2.5629e+01, elapsed time [h]: 0.00
f19 in 2-D, instance 5: FEs=4 with 0 restarts, fbest-ftarget=2.1737e+01, elapsed time [h]: 0.00
f19 in 2-D, instance 41: FEs=4 with 0 restarts, fbest-ftarget=1.4384e+02, elapsed time [h]: 0.00
f19 in 2-D, instance 42: FEs=4 with 0 restarts, fbest-ftarget=5.4887e+01, elapsed time [h]: 0.00
f19 in 2-D, instance 43: FEs=4 with 0 restarts, fbest-ftarget=1.1087e+01, elapsed time [h]: 0.00
f19 in 2-D, instance 44: FEs=4 with 0 restarts, fbest-ftarget=9.6195e+00, elapsed time [h]: 0.00
f19 in 2-D, instance 45: FEs=4 with 0 restarts, fbest-ftarget=8.6850e-01, elapsed time [h]: 0.00
f19 in 2-D, instance 46: FEs=4 with 0 restarts, fbest-ftarget=6.0522e+00, elapsed time [h]: 0.00
f19 in 2-D, instance 47: FEs=4 with 0 restarts, fbest-ftarget=9.5790e+01, elapsed time [h]: 0.00
f19 in 2-D, instance 48: FEs=4 with 0 restarts, fbest-ftarget=2.0568e+01, elapsed time [h]: 0.00
f19 in 2-D, instance 49: FEs=4 with 0 restarts, fbest-ftarget=1.7207e+01, elapsed time [h]: 0.00
f19 in 2-D, instance 50: FEs=4 with 0 restarts, fbest-ftarget=2.1508e+01, elapsed time [h]: 0.00
```

Figure 2.2: Result for the function number 19 with a big population

Other configurations have been tested while removing the mutation or the crossover functions or both. Only a slight improvement has been found whith the mutation function. However, the crossover slowes

the processus and make the general algorithm less efficient. Furthermore, the hillclimbing technique is not efficient when it matters finding the global minimum. It can be improved by adding restarts or more complex iterations to avoid a search stucked in a local minimum.

Conclusions

The GA for ANN is a mighty mechanism for the learning process of the ANN, further investigations could involve a bigger variations in the parameters and bigger (compuatationally expensive) populations and number of generations.

The evolutionary algorithm is a powerful tool to find best solution for problem. However the crossover is not a suitable method for the problem using mathematical functions.

Bibliography

[1] Hornik, K., Stinchcombe, M., White, H. 1989. *Multilayer feedforward networks are universal approximators*. Neural Networks, 2, 359–366.

[2] Funahashi, K. 1989. *On the approximate realization of continuous mappings by neural networks*. Neural Networks, 2, pp. 183–192.

[3] Castro, J.L., Mantas, C.J., Benitez, J.M., 2000. *Neural networks with a continuous squashing function in the output are universal approximators*. Neural Networks 13, 561–563.

[4] Enăchescu, C. 2008. *Approximation Capabilities of Neural Networks*. Journal of Numerical Analysis, Industrial and Applied Mathematics (JNAIAM) vol. 3, pp. 221-230.