

Master Students

Biologically Inspired Computation – Coursework 1

Multilayer Perceptron, Genetic Algorithms and Black-Box Optimization

Brice Cagnol and Halvin Dufour

November 2016

Heriot-Watt University

Introduction

For our first assessment of the semester, we had to make a multilayer perceptron. Its goal was to approximate a function. In order to obtain it, we did not use back propagation but a genetic algorithm, so we could learn both how to implement a MLP and a GA.

Then, we also had to work with COCO, a framework working as a black-box to calculate the successfulness and the performances of our algorithms.

You can find our project here on Github: <https://github.com/Solahtar/BICCW1/>

Chapter 1

Evolve a Multilayer Perceptron

1.1 Creating the perceptron model

For realising the multilayer perceptron, we decided to use Matlab as it offers many mathematical functions (like tangent sigmoid we have used for this coursework) and tools for displaying our results.

Our perceptrons are cell arrays with one line and as many columns as layers. Each layer is represented by its weights matrix. Our weights are located between -1 and 1 . The generation of one MLP is made by the function `emphcreateMLP` which takes two parameters: the number of neurons in each layer (*LayLen*) and the size of an input. Then it creates each layer depending on its size and the previous one (the size of the inputs for the first layer).

To calculate the output of an MLP given an input, we use the function *realOutput*. It applies the activation functions and calculates the product giving the result.

1.2 Approximate a function with a genetic algorithm

1.2.1 Starting point

We started by approximating the sphere equation: $x^2 + y^2$. Our population is made of a hundred perceptrons with random weights. We used perceptrons with a first layer of ten neurons, five layers of five neurons and a final layer of just one neuron. We generate a hundred couples of inputs and outputs.

Once our first generation is made, we calculate the fitness of every individual. This is given by the absolute value of the actual output minus the desired output. We will make tests with two hundred generations.

Here are our first results:

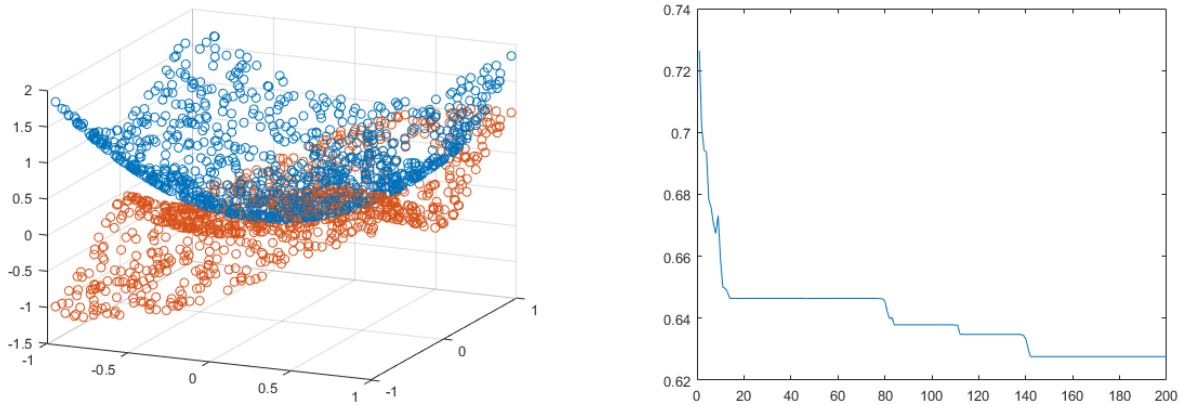


Figure 1.1: Left: Our result (in orange) and the original function (in blue) — Right: Average of the fitness depending on the number of generations

1.2.2 Adding a bias

We noticed that adding a bias to the calculation of the output actually improves our results. Without any bias, our best fitness was around 0.63. We tried to run our program with some different bias and it appeared that we could have this fitness dropping to 0.21 with a bias of 0.55, giving a far better approximated function:

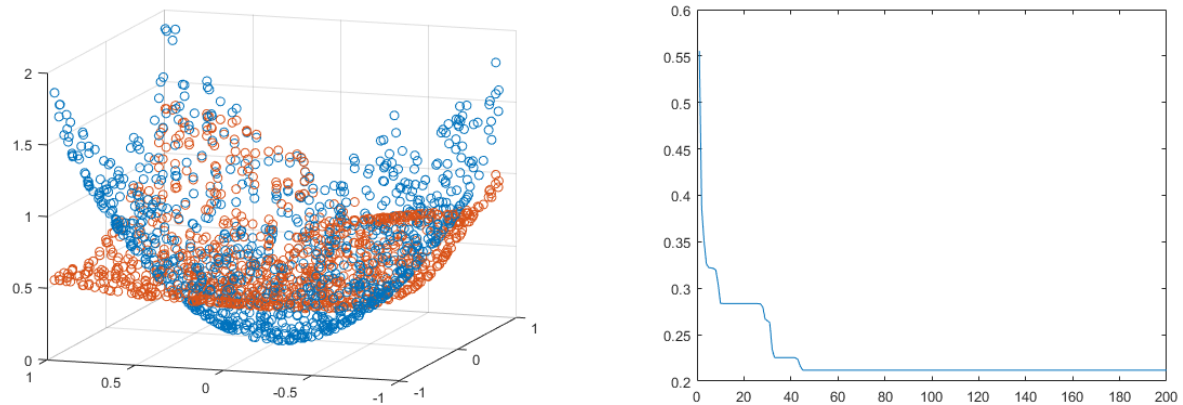


Figure 1.2: Left: Our result (in orange) and the original function (in blue) — Right: Average of the fitness depending on the number of generations

We will keep this bias for our next tests.

1.2.3 Create a new generation

To obtain the results above, we created our generations as follows:

- We selected twenty parents with a tournament method with ten draws for each parent.
- Those parents make many children so we can only keep the parents and their offspring for the next generation (*generateChildren*).

- We mutate all the new generation (*mutatePopulation*).
- We do another tournament to only keep a hundred individuals.
- We calculate the fitness of this generation before creating another one.

We made some tries in order to improve our method, but those did not give really good results. For instance, here is our result when we tried to reuse the current generation to mutate its individuals and have less children in the next generation:

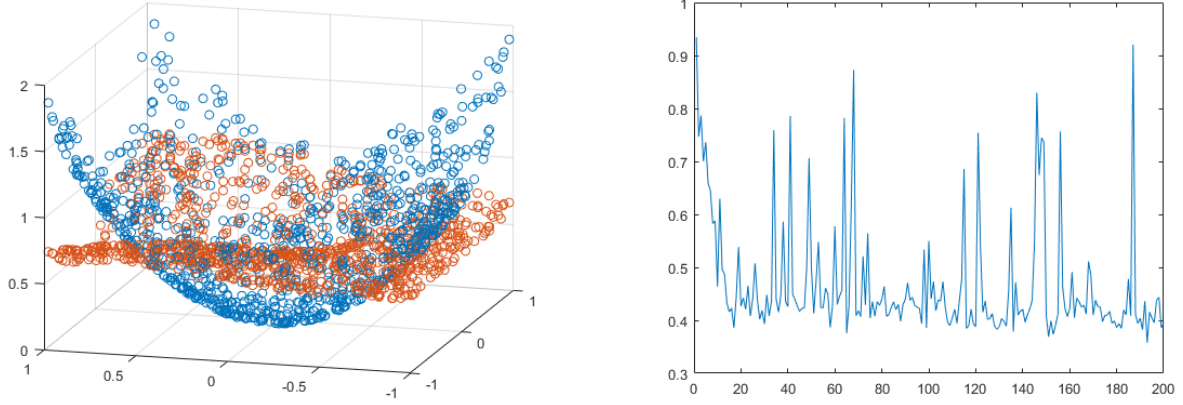


Figure 1.3: Left: Our result (in orange) and the original function (in blue) — Right: Average of the fitness depending on the number of generations

Actually, we use more than one mutation. Every chromosome has one chance over five of being mutated. Then it can have two lines or two columns from a layer swapped. There is also one chance over five that a layer has new random values. If we try to keep only one of these three mutations, our results are different. We finally noticed that the mutation that works the best is when we swap two columns from a layer of the neuron with a final fitness under 0.15 on our best try. The two others never give these good results but when we use them, we have generally better results, that is why we kept them.

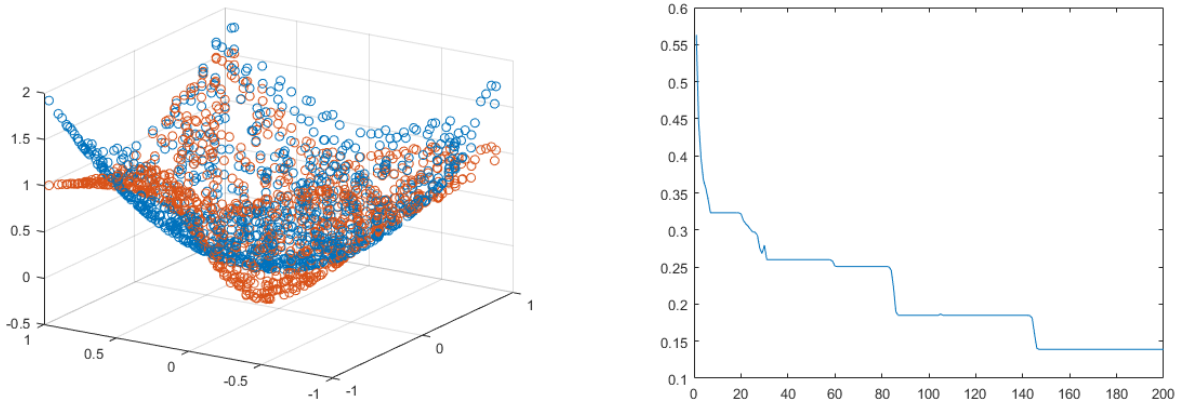


Figure 1.4: Left: Our result (in orange) and the original function (in blue) — Right: Average of the fitness depending on the number of generations

1.2.4 Changing the topology of the perceptrons

Adding layers to our perceptrons did not gave us better results. When we reduced the number of layers to just two hidden layers, the fitness almost was the same than for ten hidden layers. It finally seems that we have our best results for five hidden layers.

We tried then to remove or add neurons to each layer. We saw that reducing it gave us bad results, but our results are very close to our best tests when we add one neuron to each layer. We finally obtain good results with some layers with six neurons and others with five. With the hidden layers with six, five, four, five and six neurons, we obtained a best fitness of 0.16 resulting in this approximation very close to the original:

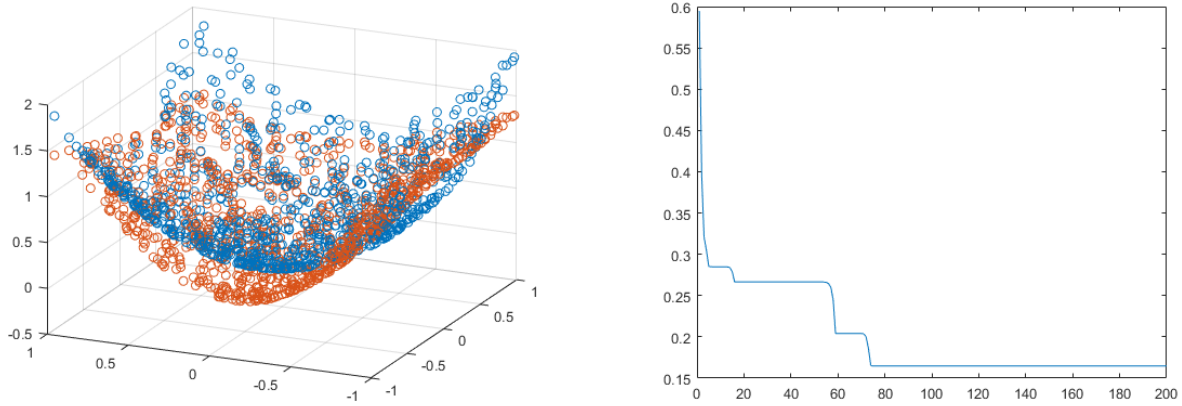


Figure 1.5: Left: Our result (in orange) and the original function (in blue) — Right: Average of the fitness depending on the number of generations

1.2.5 A first conclusion

It is finally hard to find the best way of applying a genetic algorithm to a MLP as we do not really know the factors resulting in a good approximation of the function. But by trying many methods, we finally can find something kind of good, even if we always end on a local minimum in less than a hundred generations. Our genetic algorithm even works on other functions than the sphere equation like the mean of the two inputs:

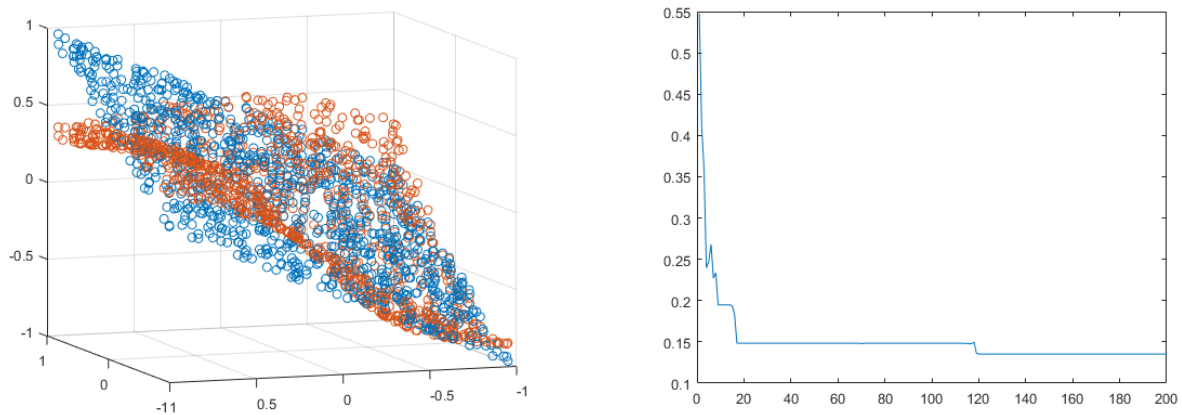


Figure 1.6: Left: Our result (in orange) and the original function (in blue) — Right: Average of the fitness depending on the number of generations

For sure, we could just work with more pairs of inputs and outputs and more neurons (because just adding inputs does not give better results) our compute more operations, but the algorithm would take to much time to be applied.

1.3 Minimising the approximate function

This time we do not apply a genetic algorithm to perceptrons but inputs. Their fitness is the output given by our best MLP for each individual as we would like to find the inputs giving the minimum of our approximate function. We chose to generate a hundred inputs and make a thousand generations for our GA. The process is simple: for each chromosome, we mutate it. If the mutant is better than the original, then it replace its "father". The mutation consists in adding a random value between -0.1 and 0.1 to the x and y of the input. We can observe that in our case this converge quite quickly (in less than two hundred generations) and we obtain a very good approximation as our approximate function does not have multiple local minima. In this case, we would probably have fallen in these local minima as our mutations probably would not have made possible to leave a local minimum.

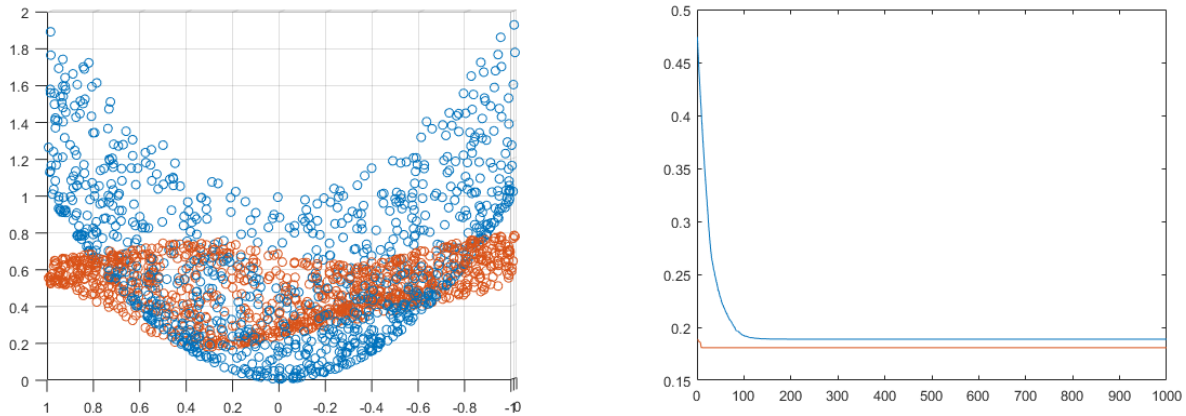


Figure 1.7: Left: Our result (in orange) and the original function (in blue) — Right: Average of the fitness depending on the number of generations

Chapter 2

Black-Box Optimization

2.1 Choosing an Evolutionary Algorithm

The chosen EA for the Black-Box Optimization is the hill-climbing algorithm. We started with a basic implementation of the hill-climbing algorithm, in which the new generations are created via mutation of the parent only.

We wanted to first analyse the results of our algorithm to be sure it was correctly implemented before improving it, but as explained later it has not been possible to analyse the results so the EA algorithm was not modified.

This part has been realised using Matlab as well, since the example in the subject is for Matlab this gave a slightly better starting/fallback point for this part.

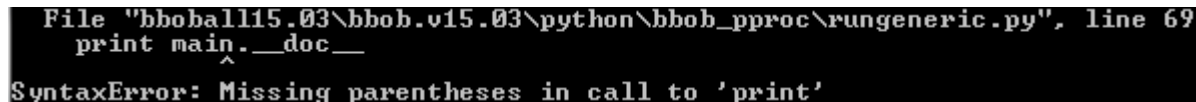
2.2 Running the Evolutionary Algorithm and analysing the results

The EA has been implemented by modifying the existing starter code for COCO.

However, if results were generated by COCO's Black-Box algorithm, they couldn't be exploited in any way.

2.2.1 Using COCO's post-processing Python script

We were supposed to use COCO's post-processing Python script to generate exploitable tables and figures but, after following the instructions given to run the script via a command line, an unexpected error happened.

A screenshot of a command prompt window with a black background and white text. The text shows the file path "bboball15.03\bbob.v15.03\python\bbob_pproc\rungeneric.py", line 69, and the code snippet "print main.__doc__". Below this, a red error message is displayed: "SyntaxError: Missing parentheses in call to 'print'".

```
File "bboball15.03\bbob.v15.03\python\bbob_pproc\rungeneric.py", line 69
    print main.__doc__
SyntaxError: Missing parentheses in call to 'print'
```

Figure 2.1: Screenshot of the command prompt when trying to run the Python script

As we can see, the error does not come from giving the wrong DATAPATH folder (folder where the Black-Box result data is stored), or any other user error which would be easy to correct, but from the script itself even though it has not been modified.

2.2.2 Using Matlab's plot() function

Even though we couldn't run the Python script to generate exploitable data, we still have generated the data that the script was supposed to use, therefore we tried using Matlab's plot() function as an alternate way to get figures.

We weren't expecting it to be nearly as good or exploitable as COCO's post-processing script, but there weren't really any other option left to try to analyse the results.

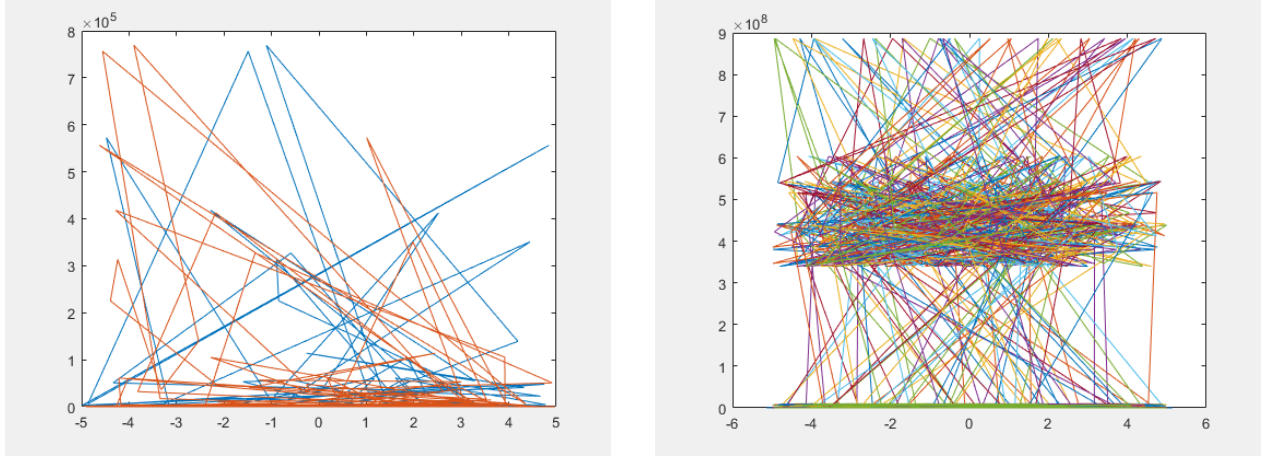


Figure 2.2: Left: Resulting figure when restricting to dimension 2 functions — Right: Resulting figure for all default dimensions (2,3,5,10,20,40)

These plots were supposed to show the evolution of the error (ordinate) relatively to the best solution of each generation (abscissa) but, as we can see on the two figures above, nothing seems really exploitable.

All 24 of COCO's Black-Box functions are shown on the figures, but we can clearly see (especially on the left figure) that a single point on the abscissa can give multiple values on the ordinate for the same function, probably because the input values are in two or more dimensions instead of one.

Without COCO's post-processing script, and with our unsuccessful attempt at using Matlab's plot() function as alternative, we couldn't analyse how effective our implementation of the hill-climbing algorithm is. And without that data, we wouldn't be able to see any improvement that could be made by modifying the algorithm.

Conclusion

Finally, we gained a better understanding of the MLP and GA. We do not keep in mind only theoretical notions but the knowledge of how to implement them and how they work, what they allow us to do, their benefits and their limits. We also acquired more experience over Matlab that we did not know very well before. Finally, even if we did not understand everything about COCO, we have an idea of what we can do with this complicated but also powerful tool that we could try to learn later for future projects.