F21BC Biologically Inspired Computation – Coursework 1

Introduction

The purpose of this assignment is to use Black-Box Optimization software to benchmark the results of two different tasks. The software in question is called COCO and provides a variety of functions for benchmarking. The first task is to use an evolutionary programming approach to minimise one of these functions and the second task is to use a genetic algorithm to evolve a neural network to approximate one of these functions. This assignment was completed individually. The testing procedure and algorithms for each of these algorithms is described, along with improvements to the algorithms and the corresponding results. All the development done for this project was done in Python within Spyder – a development program as part of the Anaconda2 software package.

Task 1

Introduction

The first task involved creating an evolutionary algorithm to minimise a function. Minimising a function essentially involves finding the input variables that will allow for an output from the function of zero. The evolutionary algorithm was developed and then improved over a series of 5 iterations.

The Algorithm

The evolutionary algorithm was developed separately from COCO and then integrated with it afterwards. The algorithm was created from scratch using a modular approach. The basic outline of the final algorithm is as follows, although not all elements of it were included from the beginning and other variations took place:

1. Generate a random population of candidate solutions, dimension 2 (2 variables) store within an array of size (4 x population number).
2. Feed each of the solutions into the fitness function and store the fitness.
3. Rank each solution in order of fitness and select a certain proportion of them for mutation.
4. Per a pre-set probability randomly select the lowest solutions and mutate them.
5. Feed the new population members back into the current population.
6. Go back to step 2 and repeat until the set iterations are complete or a suitable solution has been found with sufficiently low error.

Each of the key parts of the algorithm were developed in functions which were then called during the main body of the program. The code providing this functionality was split into two files – all the functionality for this task came from the genetic algorithm file.

Testing & Iterative Development

The testing for this algorithm was done within Spyder using the iPython console. The code from my genetic algorithm file was integrated within COCO using the example experiment file as inspiration with heavy modifications. The function used here was F1 of instance 0 and the objective was to minimise this function.

Test 1 – Random population of 10 initially between -10 and 10. Evaluate function, mutate 100% of population randomly by plus or minus 5%, run for 100 iterations. Stop condition for error is < 1.

Test	Iterations	Mutations	Best x1	Best x2	Error
1	37	740	-5.751594106	4.929937805	0.2978772622
2	27	540	-2.464839120	6.162097800	-0.572193657
3	21	420	7.544482657	1.676551701	0.28557888
4	0	0	-7.0	4.0	0.160778
5	15	300	4.56147516	-11.4036879	0.989132957

Table 1. Results of Test 1.1

From Table 1 shows that this brute force approach worked surprisingly well. The performance of this algorithm is clearly poor, but provides a good base benchmark. The large number of mutations allows the best x1 and x2 value to be found eventually. However, this is clearly heavily influenced by the range of the starting random values as can be seen in the result of the 4th test where the initial random values produce an acceptably low error straight away. The other candidate solutions still have very high error.

In the algorithm, the mutation alteration is too random. Therefore, the mutation amount was slightly increased and only affected the lowest 50% of the population at each iteration. This produced markedly worse results with the solution rarely being found so instead the lowest 80% of the population were mutated.

Test 2 – Random population of 10 initially between -10 and 10. Evaluate function, mutate lowest 80% of population for +/- 10% run for 100 iterations.

Test	Iterations	Mutations	Best x1	Best x2	Error
1	11	176	0.700274491	6.30247042	0.767996878
2	21	336	8.218432	-7.044370	0.6891708811
3	100	1600	0	-10	1.22816377
4	64	1024	8.05533707	2.41660112	-0.291573774
5	91	1456	-1.825175350	-12.776227	-0.85477887

Table 2. Results of Test 1.3

The results here showed a mixed response to the changes. In general, the algorithm performs worse although for results 1 and 2 which find a solution in a comparable amount of iterations they manage to do it with far fewer mutations.

It can be seen from the results that the closeness of the initial random population to the objective answer has a large bearing on the efficacy of the algorithm. To combat this the range of random initial values were widened and the population number was increased.

Test 3 – Random population of 20 between -20 and 20. Evaluate function, mutate lowest 80% of population for +/- 10% run for 100 iterations.

Test	Iterations	Mutations	Best x1	Best x2	Error
1	21	756	-9.7673264	-6.5115509	0.601802290
2	40	1440	-8.17948953	2.72649651	-0.403978375
3	4	144	-4.7916	-11.979	0.25179508
4	6	216	-10.320453	-3.969405	0.923676614
5	35	1260	4.98622114	4.57070272	0.769623254

Table 3. Result of Test 1.4

This can be seen to be a significantly more reliable setup compared to the previous configuration. Although the iterations are significantly lower than previously the mutations are higher than the

initial testing of a 100% mutation rate. Unfortunately, in general the other members of the population still have a large error.

The next iterative development phase was designed to deal with this problem. In order to do this the mutation, rather than being a fixed value set at the beginning, is set as a 50% of the error.

Test 4 – Random population of 20 between -20 and 20. Evaluate function, mutate lowest 80% of population for +/-50% of the error, run for 100 iterations.

Test	Iterations	Mutations	Best x1	Best x2	Error
1	13	468	2.0	6.0	0.33037888
2	41	1476	-7.5	3.5	-0.27682112
3	5	180	-7.0	4.0	0.16077888
4	13	468	5.0	-11.0	-0.63602112
5	20	720	-3.5	-12.5	0.63197888

Table 4. Results of Test 1.5

This alteration provided an increase in performance, the average iterations and mutations both decreased in comparison to the previous iteration of the algorithm. The proportion of the error for mutation was increased to 70%. This provided a good balance between mutations and iterations.

The algorithm was then tested on a variety of other functions with this alteration to find the effectiveness of the algorithm on the on the other functions.

It quickly became apparent that the algorithm was reliant on the random initial values when minimising a function. For the first function, which this algorithm was designed around, the initial values were quite close to the target values so the performance was very good. The error was small and could be used as part of the mutation function. However, when the initial value creates a large error the algorithm isn't good enough to recover and change the inputs enough to reduce the error. A method to use the error to alter the inputs should work but isn't reliable across a multitude of algorithms as the error for each input changes. However, although the algorithm couldn't reduce the error down to zero, it could reduce it by several orders of magnitude. Changing the input using a percentage of the previous input rather than the previous error allowed progress to be made. Increasing the iterations to about 350 produced the best result. Beyond that the error would typically not decrease significantly. This behaviour is noted below for example with function 20.

Test 5 – Random population of 30 between -20 and 20. Evaluate function f20, mutate lowest 80% of population for +/-10% of the input, run for 1 iterations.

Test	Iterations	Mutations	Error
1	1	1	2687
2	1	1	5088
3	1	1	11612

Table 5. Results of Test 1.6

Test 6 – Random population of 30 between -20 and 20. Evaluate function f20, mutate lowest 80% of population for +/-10% of the input, run for 350 iterations

Test	Iterations	Mutations	Error
1	350	350	184
2	350	350	186
3	350	350	184

Table 6. Results of Test 1.7

Discussion

The results here showed a variety of improvements and their results on the efficacy of the algorithm. One of the biggest weak points of the algorithm was the way it applied the mutation to the inputs. As the location of the minimising inputs was unknown it was difficult to know whether to use the mutation as an addition or subtraction to each of the inputs, by default the probability of doing either was equal. This was a big problem in the network and a few additions or subtractions the wrong way could cause an input to spiral out of relevance. This could also then create a needless drain on the computational resources. The defence for this was to limit the population being evolved to 70-80%, hopefully preserving the most promising solutions. If this task would be repeated or expanded on more focus would need to be put into overcoming the problem of having initial random weights with very high errors. This is a large problem and simple time constraints meant that testing was affected. Ideally more tests would have been carried out and the results averaged to obtain a more accurate measure of the results of each approach.

Task 2

The second task consisted of creating a multi-layered perceptron network and then evolving it to approximate a function within COCO using a genetic algorithm. The neural network was evolved in a variety of ways, which are detailed later in the report. This task used both the neural network and genetic algorithm files. It inherited some key functionality from the previous task but added elements such as crossover.

Algorithm

The algorithm was created separately from COCO and then integrated for testing. To begin with the basic neural network was created. The initial network consisted of 1 input, 3 neurons in the middle layer and 1 output in the output layer. The network was a simple feed forward network as shown in the lecture notes [1]. The activation function used in each of the perceptrons was a simple tan function. Each of the nodes of the neural network were modelled as MCP neurons. The mechanisms of a neural network were explored with other resources [2].

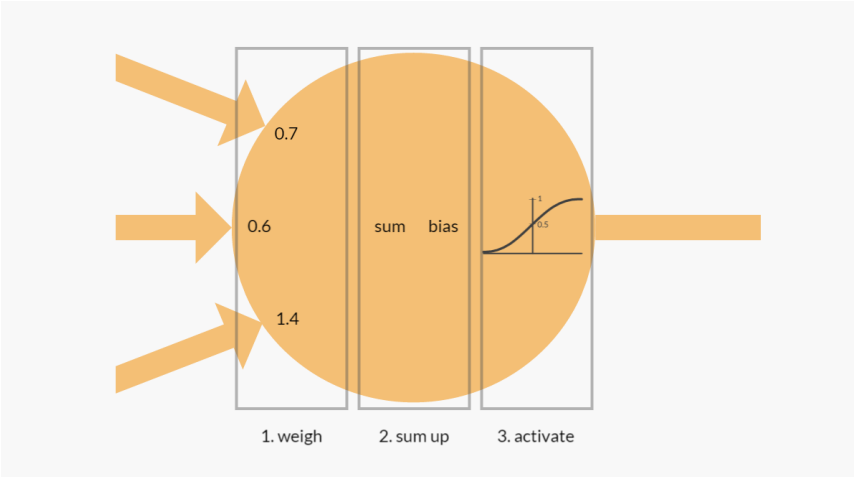


*Figure 1. A Perceptron (**credit: https://appliedgo.net/perceptron/**)*

1. The initial weights for each of the nodes are continuous and initially chosen at random.
2. To begin with an input array was created and populated with a series of random numbers.

3. Each element of the input array is then applied to the function and the target array is created.
4. The input is fed through the network with it being multiplied by each weight, summed up and stored and then passed on to the next layer for each layer.
5. The output of the network is then calculated and compared with the target array to calculate the error.
6. The weights and error are then fed to the genetic algorithm. The next generation of weights is created and then fed back to the network. The algorithm then returns to step 4.

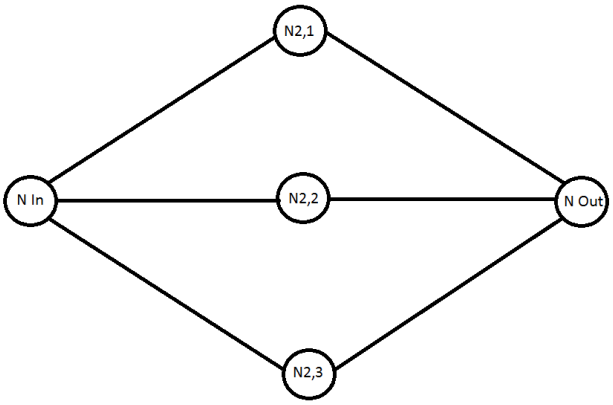
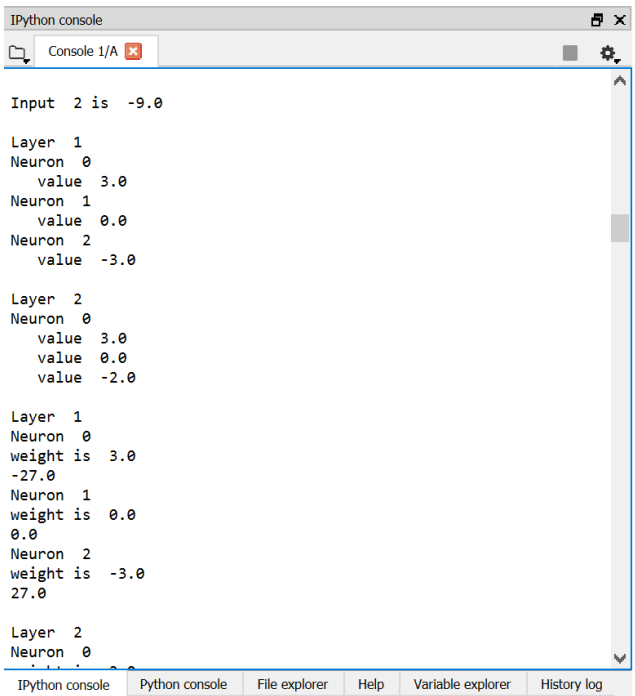


Figure 2. Initial Layout

Testing & Iterative Development

The initial layout provided a network with 3 layers and 6 total weights. The network code was fully reconfigurable with layers and numbers of nodes all available as user input at the beginning of the code. However, this created significant encoding problems so the network was limited to two layers of weights.



The final algorithm process is detailed below; the results of the tests are then presented.

To begin with each input is assigned an initial random value as can be seen in the image. The initial weights for each of the layers are randomly generated for each of the inputs. The inputs are then passed through network, multiplied by the weights and then summed before being passed on.

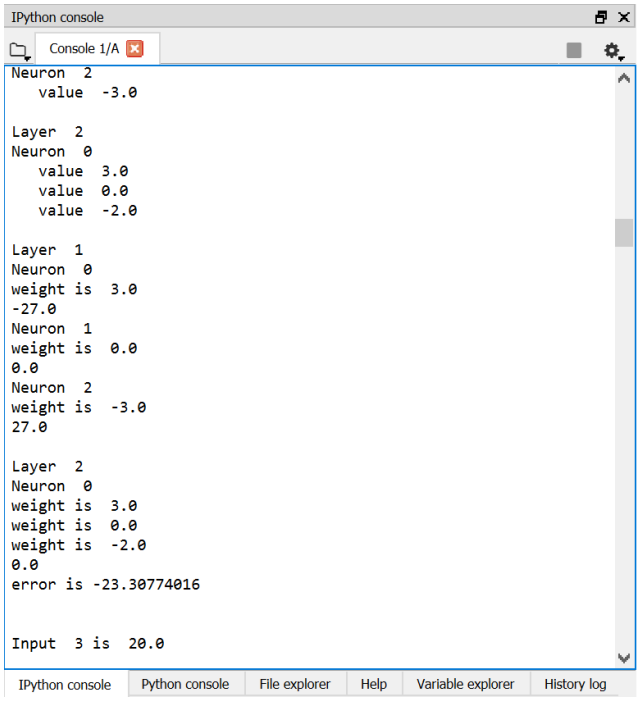


Figure 3. Initial weight selection

Figure 4. Error calculation

Once the weights have been summed the output is then compared with the target output. Then the algorithm moves on to the next input.

The weights are then split into pairs and each are matched with the error they were associated with. The weight pairs are then ranked according to their absolute value.

The set amount of top solutions are then selected – in this case 12 and these are designated as parents. In this example each weight pair is added to the adjacent pair and then divided by 2 to create a child with the average weights of the two parents. An even amount of parents is always generated.

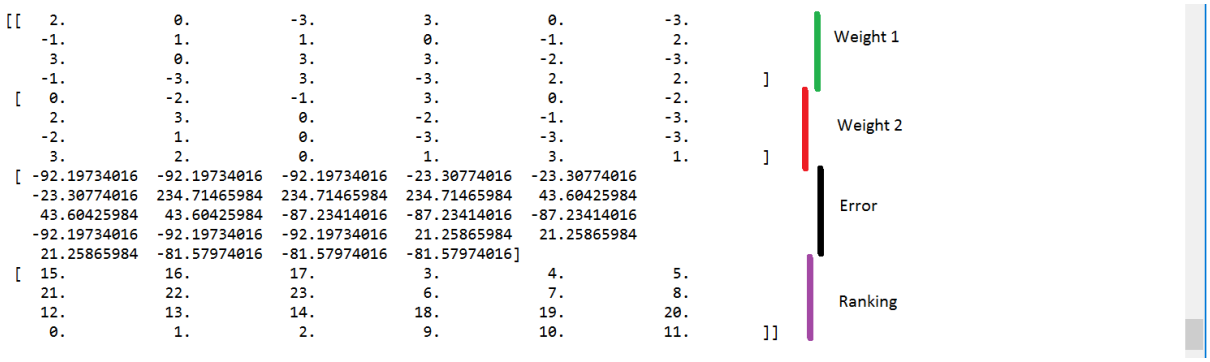
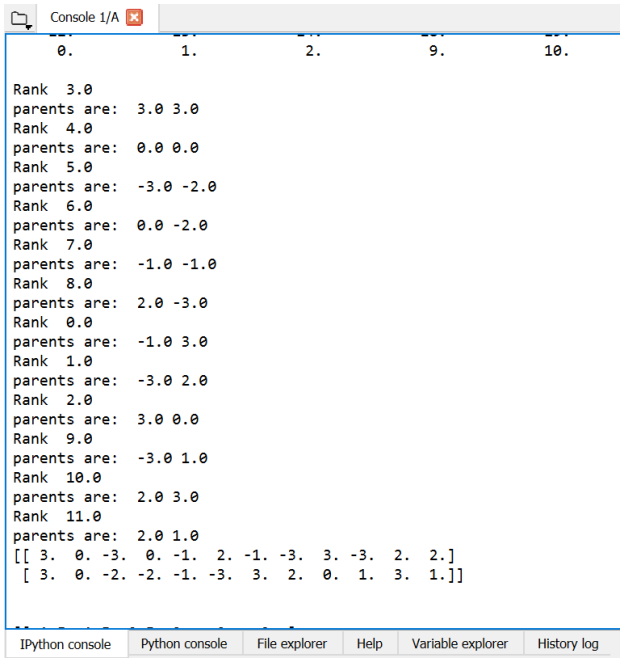
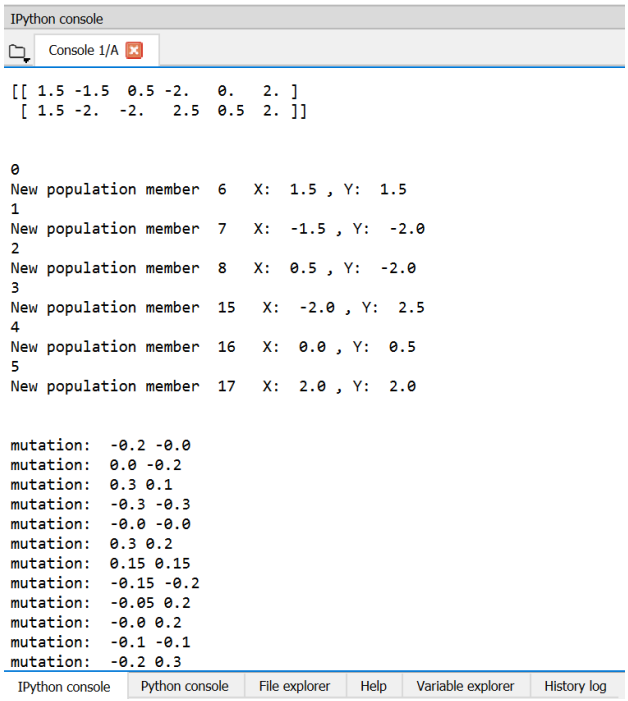


Figure 5. Population array that is handled by the genetic algorithm

Figure 6. Parent generation

Once the parents have been generated they are then stored in a parent array.





The children are printed here and then the new population members are generated. Then a proportion of the population is mutated by a certain proportion of the input. Then the weights are returned to the network and the procedure is repeated until the error starts to decrease.

Figure 7. Children are created and mutation take place

Test 1 – To begin with crossover wasn't used only random mutation to reduce the error of the modelled function. The population consisted of 24 individuals, 75% of which were mutated by 50% for 500 iterations.

Test	Iterations	Mutations	Error
1	500	10800	14.001059
2	500	10800	-33.77814
3	500	10800	-17.43254016
4	500	10800	-38.96214016
5	7	252	0.1225590

Table 7. Results of Test 2.1

Test 2 – Crossover is now used – The population consisted of 24 individuals, 25% of the population will be replaced at each iteration by children that are a result of the most successful 50% of the population. 75% are mutated by 50% for 500 iterations.

Test	Iterations	Mutations	Error
1	500	18000	-4.05014016
2	7	252	-1.53414474984
3	500	18000	-5.65334016
4	500	18000	21.25865984
5	112	4032	0.79158391

Table 7. Results of Test 2.1

Discussion

The results here show that crossover has a significant beneficial effect on the performance of the algorithm in comparison to simple random mutation. The algorithm still doesn't always reduce the error to zero but it performs much better than the previous algorithm. In the first test only one attempt finds the total zero error which can be probably disregarded. In the future changing the crossover operator has the potential to make significant improvements.

Conclusion

This was a very challenging project, the creation of a neural network and genetic algorithm from scratch and then integration of the two in a language never used by the author provided many obstacles to overcome. One weakness of this project is therefore that the actual implementation of the network from scratch coupled with the lack of clear documentation within COCO meant that a lot of time was spent on getting that done rather than the actual testing and adapting required within the course. However, simple mechanisms for random mutation of variables and crossover between different members of the population were produced and did work, succeeding in approximating the specified function. Future work would focus on testing the algorithms for both tasks, for example trying the algorithms on more functions and refining the crossover technique.

References

1. Vargas, Patricia (2015) Artificial Neural Networks, Biologically Inspired Computing. [Lecture notes] MACS. Heriot Watt University, Riccarton, Edinburgh. [Tues 22nd Sep - Thursday 22nd October]
2. Haykin, Simon O. (2008) Neural Networks and Learning Machines. Prentice Hall

GitHub Repository

https://github.com/AntiDoctor/BioComputing