Heriot-Watt University

Biologically Inspired Computation

F20BC

Coursework 1

Tyler McAllister - H00153533
tsm33@hw.ac.uk

Raw test results and the code used can be found at the following github:

https://github.com/mcallistertyler95/Bio

1 Introduction

The introduction section of this report will focus on the implementation of my multi-layered perceptron(MLP) and its evolutionary algorithm. The aim of the MLP is to approximate test functions, commonly used for optimization, by using the evolutionary algorithm to continuously improve its results.

Due to my knowledge of the language and it's pragmatic nature, I chose to do my implementation in Python 2.7. Python is also a popular language for various machine learning techniques and bio-inspired computation.

1.1 Multi-Layered Perceptron Structure

To begin, I created a class for my, multi layered perceptron neural network. The structure of the MLP is an input layer consisting of an input layer, one hidden layer and a final output layer. When initialized, the parameters input into the class determine the number of neurons that will be created at each layer. Neurons are represented by activation functions and are all connected by weights. Both weights and activation functions are created as Python lists containing sublists, simulating matrices.

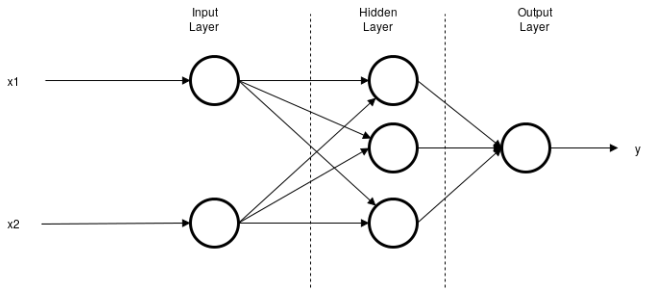

Figure.1 A basic example of the implemented MLP

Matrix multiplication is done to transfer training inputs from the input neuron to the hidden layer neurons. At the hidden layer the inputs are put through an activation function. Most commonly this function is either:

- Tanh - The hyperbolic tan function

$$\phi(v_i) = tanh(v_i)$$

- Sigmoid - A logistic function

$$\phi(v_i) = (1 + exp(-v_i))^{-1}$$

- Rectifier

$$f(x) = max(0, x),$$

Throughout the development of my MLP all three of these activation functions were used and tested on various inputs.

1.2 Evolutionary Algorithm Structure

The structure of the evolutionary algorithm is fairly basic. To begin, the population is set as the number of multi-layered perceptrons that will be run within one generation. Initially the weights for each MLP are generated randomly - between -0.1 and 0.1 for the input layer to the hidden layer and between -1 and 1 for the weights connecting the hidden layer to the output layer. For each member of the population a *cost* is recorded along with all of their weights. This *cost* represents how well the MLP performed on approximating the correct values for the function. Ideally the aim of the algorithm is to minimise this *cost* to as low to zero as possible. The cost function within the evolutionary algorithm is defined as:

$cost = d^2(1/2)$ where d = difference between expected inputs and the outputs of the MLP

When selecting a member to evolve from, the lowest *cost* of the entire population of MLPs is selected, this is treated as the fitness function within the MLP. This MLP is treated as the current population parent. From this network its weights are taken as the initial weights for the next generation of generated networks. Each member of the population then has its weights mutated by selecting a random weight at each layer and multiplying it by 0.01 multiplied by the current best *cost*.

This process is repeated for the set number of generations. In *Section2* the ideal number of generations and population members will be discussed.

2 Experiments

The MLP, with its accompanying evolutionary algorithm, were run on three noiseless test functions, each progressing in complexity:

- Sphere
- Rastrigin
- Different Powers

Due to complications with the python implementation of the COCO system I was unable to use my neural network within the black box tester. However results were produced for the functions tested on their own graphs. These results can be seen on the github page linked at the beginning of this section. As a result, testing was done manually using hard coded functions within python and my own graphing tools. Evidence of these graphs as well as the code for the three functions chosen can be found on the linked github page.

2.1 Parameters

For every function the parameters chosen to achieve acceptable approximation results were:

- *tanh* as the activation function with in the MLP
- A population size of 500 neural networks for each generation
- A generation size of 200

These parameters resulted in the best outputs from the MLP and generally managed adequate optimisations on the Sphere function. If the population is set to low then the diversity of the weights is overall lower and results can end up being skewed. Likewise, if the generations in the algorithm are set too low then there will be less improvement in the weights of the population members meaning our results will be further away from the ideal outputs of the function. Upon setting the generations and population to high values, above 700, the computation running time increases and the results generated tend to plateau, meaning that it is redundant to set these parameters so high.
The activation function used, *tanh* produced the most optimal results for multiple inputs into the MLP when compared to the previously mentioned activation functions(Section 1.1). However With one or two sets of inputs into the MLP, the *Rectifier* function gave very exceptional results but became erratic among high amounts of input data.

2.2 Results and Discussion

The results of the MLP compared against the function it is approximating are detailed in this section. Each figure depicts the MLPs evolutionary algorithm with the parameters defined in (Section 2.3) and another figure with the MLP approximating the function with a population of 100 and generation of 50, which can be defined as "bad" inputs for approximation.

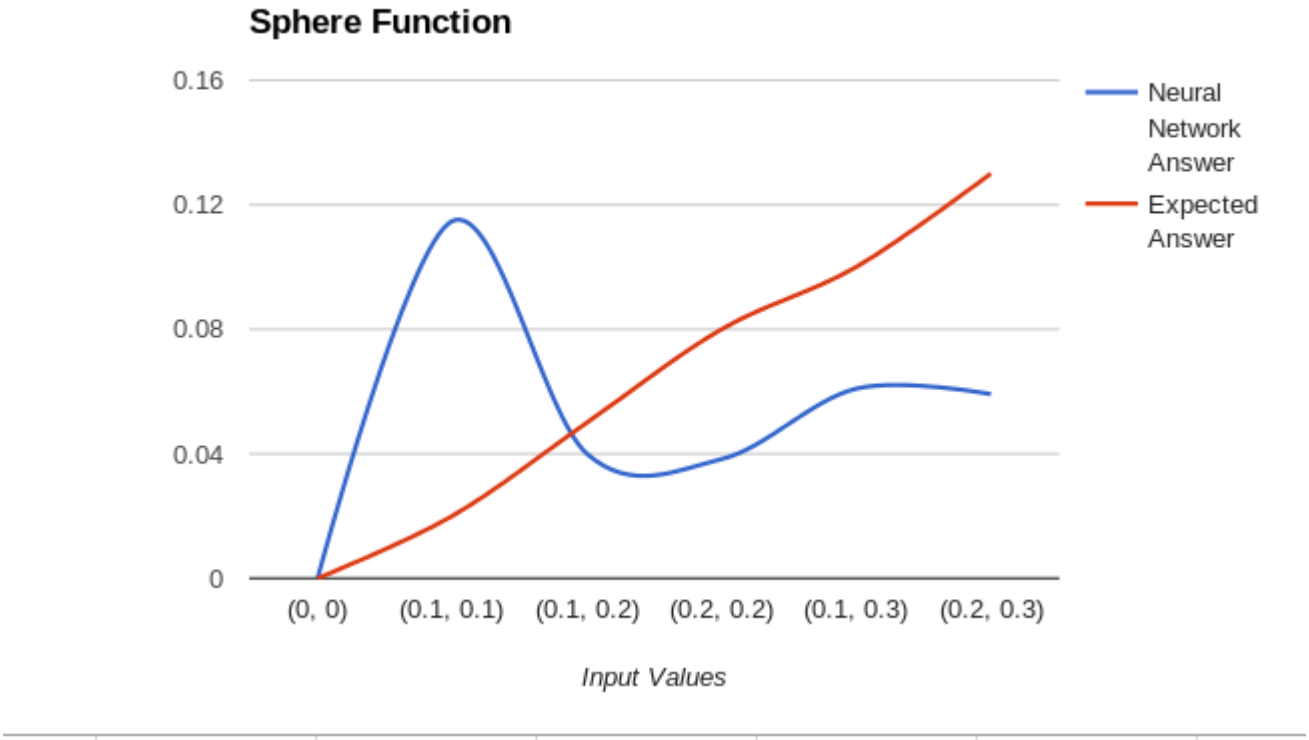

Figure2. Approximation of Sphere with bad parameters

Here, when train using a small population and generation size the results are extremely poor and wild in nature.

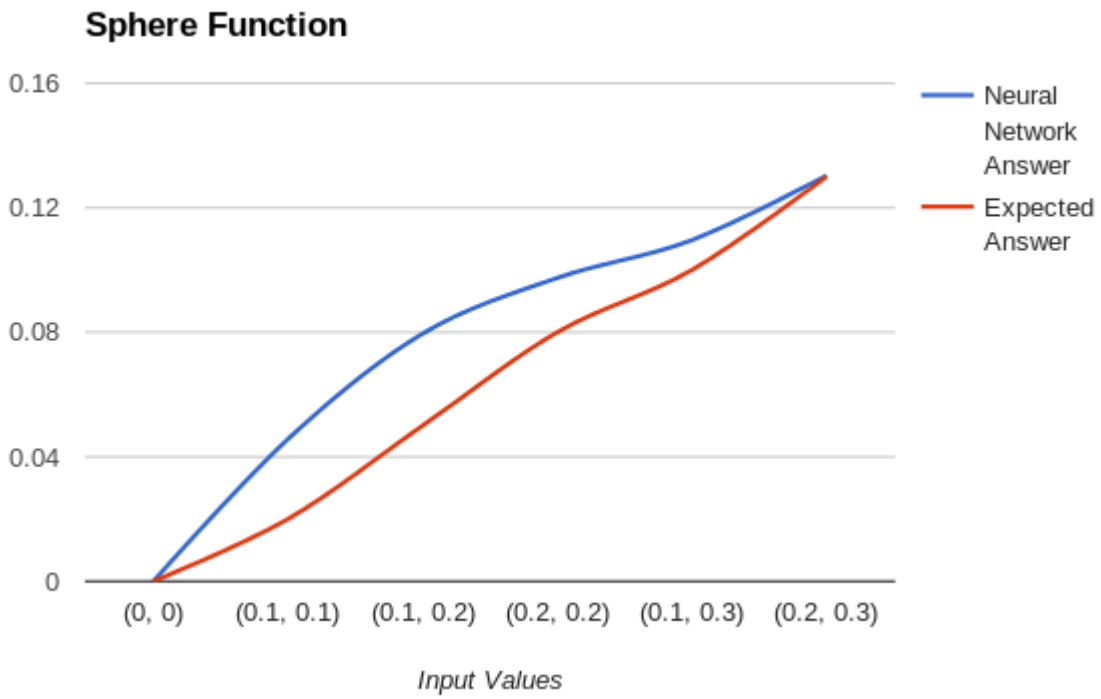

Figure3. Approximation of Sphere with good parameters

In Figure3 we can see that the MLP is much closer to approximating the function as it gradually curves into the shape of the function, particularly for the later input values in which we can see the MLP and Sphere function meeting.

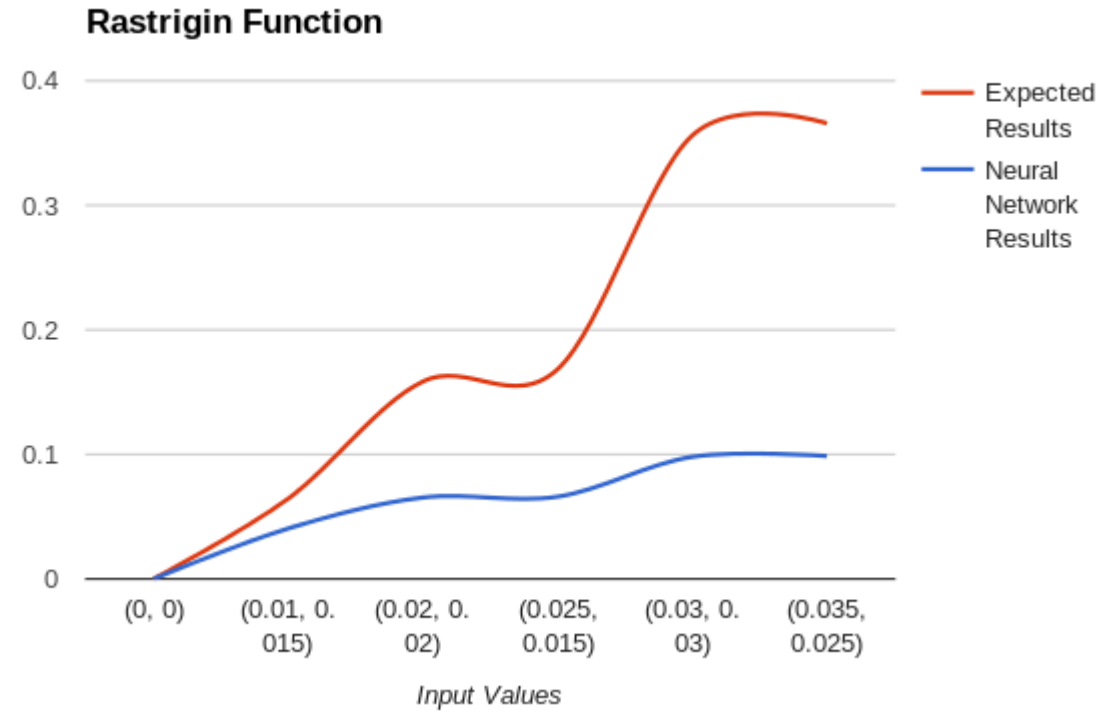

Figure4. Approximation of Rastrigin with bad parameters

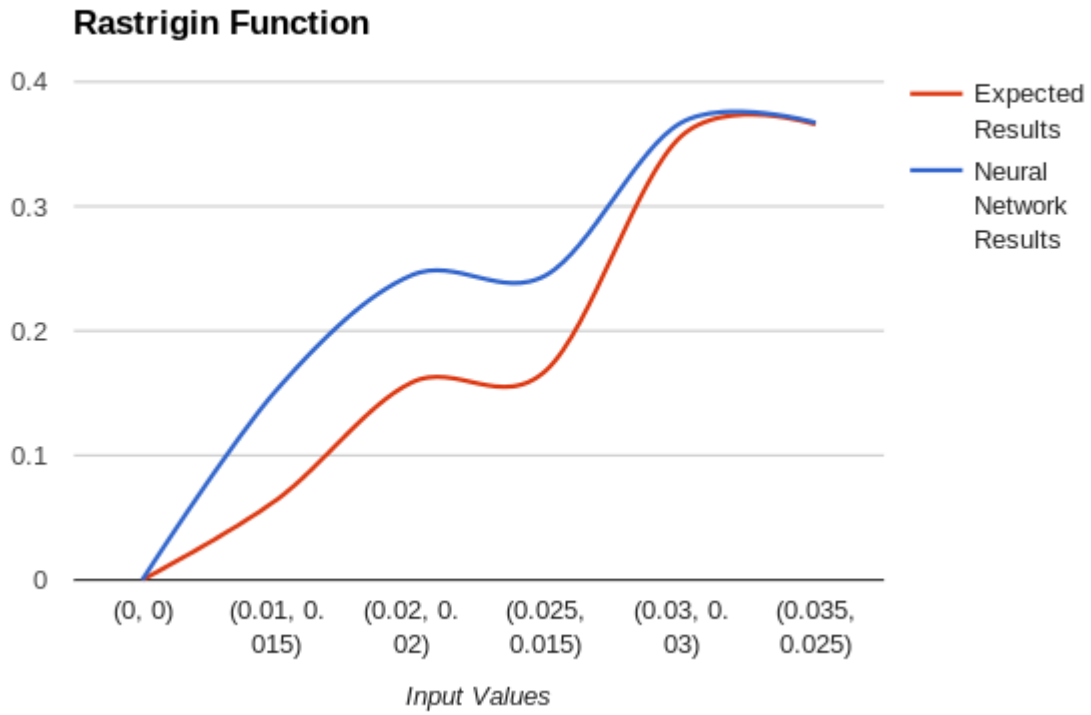

Figure5. Approximation of Rastrigin with good parameters

The complexity of the rastrigin function is much greater than the sphere function. With the low population and generation size the result of Figure4 is to be expected, as the network is nowhere near close to approximating the function. Function 5 approximates the function at a more reasonable level with the increased population and generation sizes. From these results it can be seen that the current fitness function was having difficulty when being used in functions containing greater complexity.

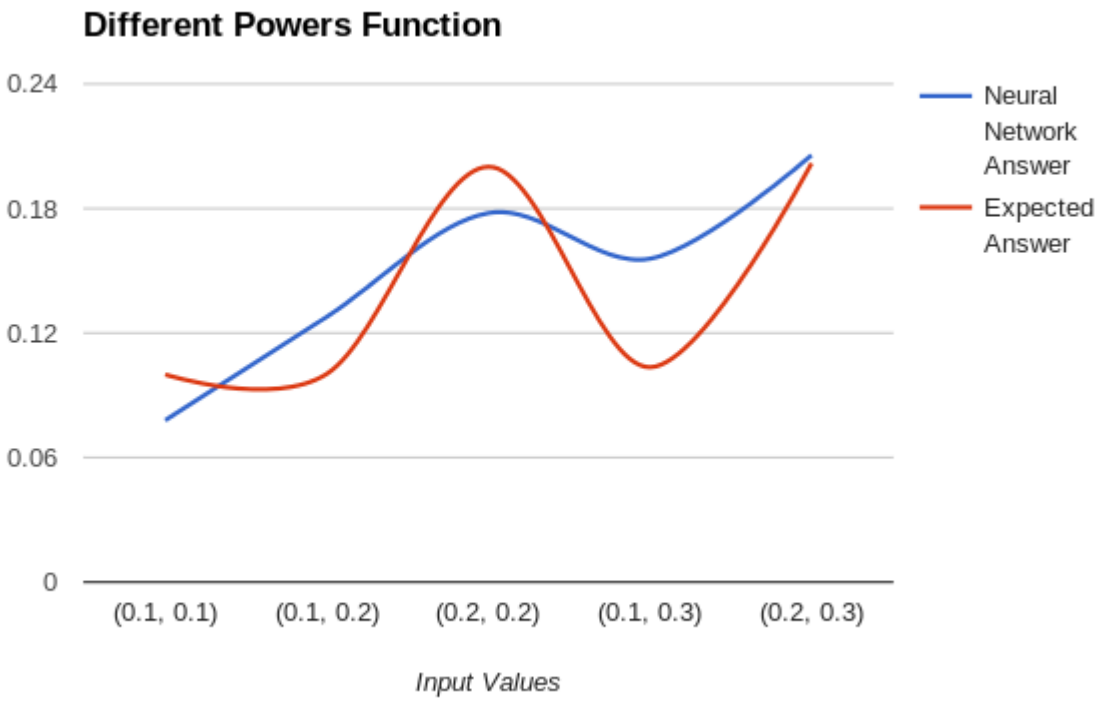

Figure6. Approximation of Different Powers with good parameters

Figure6 portrays a relatively bad approximation of the different powers function. This graph was generated with the population size set to 500 and generations set to 200 - our "good" parameter settings. The approximation created by the MLP does keep the overall shape of the function, its values are quick to shoot up rather than curve like in the different powers function. It also never reaches close to any maximum or minimum the function provides. The reason for this poor approximation, despite the use of what we have defined as good parameters, the complexity of the function makes it difficult for our fitness function to help the MLP reach any conclusive values. The implementation of a stochastsic fitness function(Montana and Davis 762-767). would have been a greater challenge but would have developed greater results when dealing with a wider variety of inputs and more complicated functions.

3 Conclusion

From Section 2 it was clear that as the complexity of the functions increased the MLP and EA were unable to accurately approximate. Noted in Section 2, the fitness function used was an issue. Another main issue within the network was the trouble it had with taking inputs greater than 1. This could often lead to erroneous results with the activation functions. This is due to the lack of normalization given to inputs before they are put into the network. With normalization methods(Haykin 176-178). The lack of implementation into COCO also signals that the MLP and EA lack full black box testing, meaning that a large variety of inputs, combinations and patterns have not been successfully tested on the network.

References

Montana, David and Lawrence Davis. "Training Feedforward Neural Networks Using Genetic Algorithms". *Proceedings of the 11th International Joint Conference on Artifical Intelligence* 1(1989):762-767.Print

Haykin, Simon S. *Neural Networks And Learning Machines.* 3rd ed. New York: Prentice Hall/-Pearson, 2009