Black-Box Optimization

Helmi Fraser (hmf30) & Job van Dieten (jv71)

(https://github.com/job-1994/Neural_Network)

I. Introduction

Black-Box Optimization software is used extensively within search and optimization research, especially with regard to measuring and comparing the performance of numerical optimization algorithms. The algorithm in question can query the value of the function being approximated at a given point but it is blind to any further descriptions of the function, like gradient or function type (linear, exponential, quadratic etc).

In this case, benchmark functions provided by the COCO (*Comparing Continuous Optimisers*) platform is used to test the performance of a *multilayer perceptron* [1] artificial neural network that is trained via a genetic algorithm.

II. Background

Artificial Neural Networks

An artificial neural network (ANN) is a machine that attempts to mimic the way a biological brain performs a given task or function, such as vision or hearing. An ANN is comprised of many simple computing nodes, termed "*neurons*", and it is through the connection of multiple neurons that an ANN provides its computational capabilities. [7]

Each neuron can receive an undefined number of input values either from the environment or from other neurons. Based on these inputs the neuron calculates, by way of an *activation function*, what value to output. The connections between neurons have associated weights, known as *synaptic weights*, which help determining the importance of a signal to the neuron, and how much it contributes to the value passed through the *activation function*. In effect, it is a hugely distributed and parallel processor.

Through a learning process, an ANN acquires knowledge from its environment. The name given to this process is called a *learning algorithm* and its objective is to modify the network's synaptic weights in order for the network to perform its desired function. The ability for a synapses to modify is known as *synaptic plasticity*. [7]

Genetic Algorithms

A genetic algorithm is an iterative optimisation method based on natural selection that attempts to emulate the process of biological evolution. [2]

Initially, a random set of possible solutions to the problem at hand is generated. Every solution in this set is an *individual*, its properties represented as *chromosomes* containing *genes* and the entire set of possible solutions is the *population*. Each individual in the population is then evaluated based on how well it performs as a solution to the problem and this measure is the *fitness*. Based on this fitness value, a selection operator chooses individuals to create the next generation, often termed *reproduction*. This is achieved through a genetic operator called *crossover*, which is the mixing of a certain number of traits (genes) from each parent to produce an offspring. After this, random mutation of individuals in the population may occur, which could be large or small. The algorithm terminates when a certain number of generations has been produced or when a solution with an acceptable level of fitness has been produced.

III. Implementation

Neural Network

The neural network was structured as shown in Fig. 1, with arrows representing neuronal input/output and weight.

The entire solution domain of synaptic weights is represented linearly in a $N - by - weights$ matrix [3], where each row is a set of weights for the network, a chromosome. These individual set of weights are extracted and encoded into another data structure, which contain weights sorted in matrices in the order that they are represented in the network: input layer, hidden layers and output layer.

The implemented network is capable of changing its topology, with variables in the code that define the number of: inputs, hidden layers and neurons within the hidden layers. The number of inputs scale in equal step with the dimensionality of the problem i.e 2 inputs for 2 dimensions, 5 for 5 dimensions etc. Though the structure is fixed during optimisation, the ability to change its topology in this way would also allow the network to evolve.

Table 1 shows the parameters of the network as used during the experiment. The activation function was scaled in the x axis in order to better the network's approximation abilities.

The function `"neural_net_func"` is the neural network operator. It takes in as arguments: a matrix of inputs, a matrix of encoded weights and an integer value of the number of hidden layers desired. The resulting output of this function is a single value.

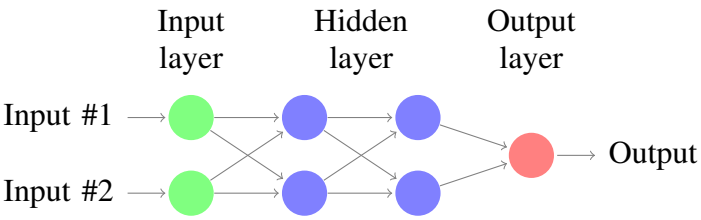

Fig. 1. Network topology for two dimensions

TABLE I
NEURAL NETWORK PARAMETERS

Attribute	Value
Activation function	$a\tanh(bx)$, $a = 1$, $b = 2$
Inputs	$n_i = DIM$
Hidden layers	$n_{hl} = 2$
Neurons per hidden layer	$n_N = 2$
Weight range	$-1 < n_r < 1$
Output weight range	$-10^9 < n_r < 10^9$
Outputs	$n_o = 1$

Genetic Algorithm

The genetic algorithm is utilised when the function `"Genetic_Algorithm"` is called from within the `"MY_OPTIMIZER"` script. The algorithm takes in as arguments: the real input/output values for a given function from COCO, the number of input/output pairs and the dimensionality of the problem. Within the script, there are variables that define some of the parameters of the algorithm, these are listed in Table 2. Apart from the type of operator, such as the type of mutation or crossover, the algorithm parameters can be tweaked as desired.

The overall operation of the algorithm is described by the chart shown in Fig. 3.

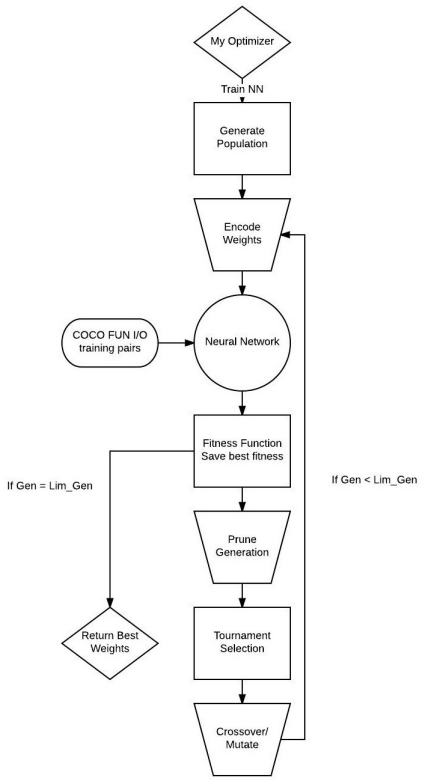

Fig. 2. Operational steps of the algorithm

The population is initialised via a function that fills a matrix with Gaussian distributed random numbers in the range of -1 to 1, these are the initial weights.

This matrix is then read and a cell is produced based on these weights that provides the weights per layer. This is needed in order to effectively map the problem into a chromosomal representation that the genetic algorithm can work with.

The design of a fitness evaluation algorithm depends entirely on the type of data received and the end goal. In this case, the fitness was to be as low as possible for good chromosomes. To achieve this, the output value from the NN was subtracted from the desired output from the COCO function for

the same input, and totalled across all input/output pairs. In order to prevent chromosomes with one large error being overlooked for chromosomes with multiple moderate errors, a second fitness function was implemented, which calculated how many of the comparisons were within 10% of each other, and the result was applied to the original fitness equation [8].

What is slightly different in the genetic algorithm presented here is the implementation of a generational pruning operator. This assesses the current population based on their fitnesses and removes the bottom x-percent. This has the effect of not only improving the quality of the population without adversely effecting diversity, but also to speed up computation by reducing the number of calculations being performed. [5]

The selection strategy used is tournament selection. This involves choosing k individuals from the population and comparing their fitness values. The fittest 2 from this selection are chosen as parents to produce a child.

The crossover operator used is uniform crossover. Each gene in the child has a random chance of being from either parent, here fixed at 0.5, equivalent to 50%. In comparison to other crossover operators such as single or two point, this operator allows chromosomes from the parents to contribute at the gene level of the child, as opposed to the segment level. This leads to a more exhaustive search of the solution space. It is also more applicable in this case as it only allows weights in the same position in the network to crossover, avoiding the problem of combining weights belonging to neurons of different roles. [9]

Mutation was handled by randomly choosing N members of the population and modifying x numbers of genes in the individual via the addition/subtraction of a Gaussian distributed random value. [4] [6]

TABLE II
PARAMETER SET FOR GENETIC ALGORITHM

Function	Parameter
Population size	$p = 200$
Weight initialisation routine	Gaussian, $mean = 0, std = 1$
Fitness function	Percentage of correct classification & Accuracy test
Selection function	Tournament, $k = 10$
Crossover function	Uniform, $mixing\ ratio = 0.5$
Mutation operator	Gaussian, $mean = weight, std = 1$
Generational pruning	$prune\ rate = 0.01$, $limit = 100$

IV. RESULTS

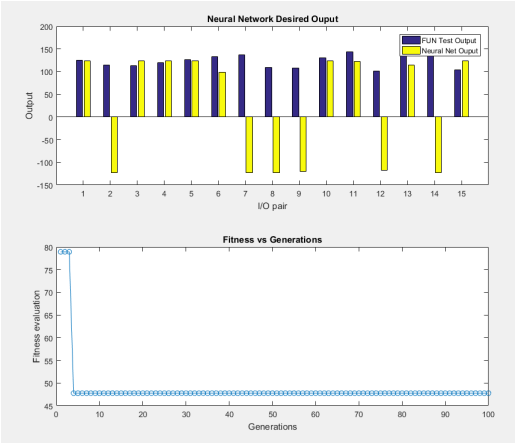

Fig. 3. COCO function 1, instance 45, popsize 200

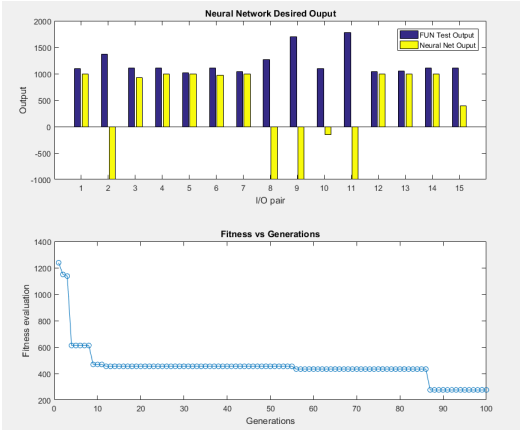

Fig. 4. COCO function 15, instance 1, popsize 100

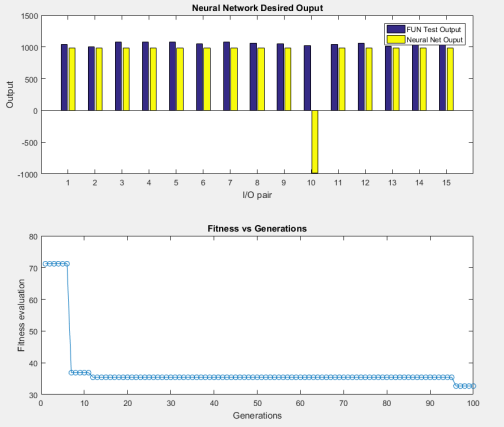

Fig. 5. COCO function 22, instance 45, generation limit 100

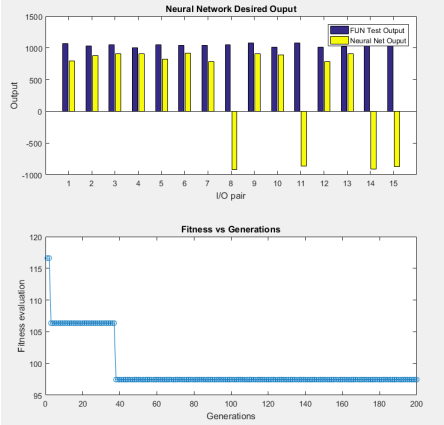

Fig. 6. COCO function 22, instance 45, generation limit 200

The algorithm was tested multiple times, and multiple of the values mentioned in tables I and II were explored. The initial set up for the neural network was taken from U. Seiffert's paper[3], in which a 2-layer 2-neuron topology was chosen. Other topologies, with more neurons and layers, were evaluated, however yielded results of negligible difference. For the genetic algorithm, the main variables that effect the results are population size and limit of generations. Figure 3 and Figure 4 show the difference the population size makes. In figure 3 the best weights is found after 3 generations, making the final 97 generations redundant.

By reducing the population, the computational time per generation decreases, and the likelihood of the best fitness being reached is still secure.

For limit of generations, figure 5 uses a limit of 100 generations, and figure 6 uses double that

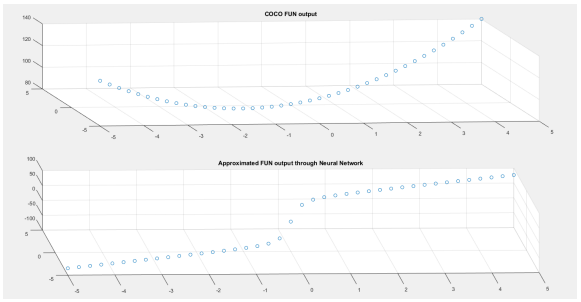

Fig. 7. COCO function vs NN approximation 3d graph

limit. The graphs clearly show that the increase in generation limit does not provide a better best fitness.

Looking at all 4 graphs, there are issues with real values being the negative of the desired value. Looking at figure 7 shows why. The final output of the neural network is merely a graphical representation of the activation function than the COCO function. This is consistent for all 24 functions the GA and NN were trained for.

The data produced by COCO was post-processed, and is included in the files on the github link.

V. Discussion

Initially, the neural network was thought to have been implemented fairly well, due to the accuracies shown in figures 3 to 6, however from figure 7 it is obvious that the results are not reliable. However, though the neural network has failed, the implemented genetic algorithm performed well, with better fitnesses being discovered up until the generational limit. This shows that the evolution of the chromosomes is still having an effect on the overall best fitness.

VI. Conclusion

In summary, the aim of this piece of work was to utilise an artificial neural network - whose synaptic weights have been trained via a genetic algorithm - and test its performance by comparing the network's output to those provided by the COCO platform.

In some cases, backpropagation is used as a learning algorithm. This solves the learning problem like an optimisation problem by using a gradient descent algorithm in order to approximate

the most optimal solution. [10] However, if the error surface is multimodal and the chosen starting point is suboptimal, gradient descent can get stuck in local minima. Hence, backpropagation is a local optimisation algorithm and it is for this reason that an evolutionary approach was used. Since genetic algorithms can include the entire search space when seeking a solution to a problem, it has a better chance of avoiding getting stuck in local minima and producing a solution that is close to the global minima.

Further work that can be done to improve on that shown here includes: the addition of evolvable biases to the neurons in the network, having a generational limit that increases with the number of genes and having a variable mutation rate that decreases as the generation count tends to its limit. The former would allow the network to properly approximate given functions while the latter two would combat population uniformity and help to minimise major deviations from fit individuals as well as act to fine tune the solutions.

REFERENCES

[1] M. Minsky and S. Papert, Perceptrons: An introduction to computational geometry (Cambridge MA.: MIT Press, 1969).
[2] A. Fraser and D. Burnell, Computer Models in Genetics (New York: McGraw-Hill, 1970)
[3] U. Seiffert, Multiple Layer Perceptron Training Using Genetic Algorithms (University of South Australia, 2001)
[4] H.G Beyer and H.P Schwefel, Evolution strategies: A comprehensive introduction (2002)
[5] S. M. Hedjazi, S. S. Marjani, Pruned Genetic Algorithm (Islamic Azad University Tehran, 2010)
[6] A. C. Koenig, A Study of Mutation Methods for Evolutionary Algorithms (Pennsylvania State University, 2002)
[7] S. Haykin, Neural Networks and Learning Machines, Third Edition (2008)
[8] G. F. Luger and W. A. Stubblefield, Artificial Intelligence: Structures and Strategies for Complex Problem Solving, Third Edition (1997)
[9] P.K. Chawdhry, Soft computing in engineering design and manufacturing p. 164, (1998)
[10] P. Werbos, Information Processing conference (New York, 1982)