Developing an Algorithm to Evolve a Multi-Layer Perceptron using a Genetic Algorithm to Minimise a Set of Continuous Functions

F20BC Biologically Inspired Computation
Stuart Marples (sm861) & Robbie Dunn (rjd65)
https://github.com/rjd65/Bio-Inspired-Coursework-1

1 Introduction

This project aims to implement an algorithm of an artificial neural network evolving through the use of an evolutionary algorithm. The evolved artificial network is then used as a surrogate function to attempt to optimise continuous functions in the CoCo (COmparing Continuous Optimisers) platform. We can then compare the output of the neural network to that of the actual function to evaluate the effectiveness of our algorithm. The algorithm required implementing an artificial neural network and the genetic algorithm to evolve it. A second genetic algorithm was also implemented to optimise the functions once the neural network is trained.

2 Experiment Setup

CoCo

CoCo is a platform for optimising algorithms against a set of continuous functions. The CoCo platform provides twenty four different continuous functions to test our algorithm on, with any dimension of inputs. Our algorithm is provided with 4 arguments by CoCo: the evaluation function which returns a fitness based on the inputs provided, the dimensions of the input, the maximum number of function evaluations and the target fitness to reach. The inputs are optimised until either the max evaluations has been reached or the fitness has been minimised below the target.

Artificial Neural Network

The artificial neural network (ANN) was created using a 3-layer feed forward algorithm. It generates a weight between each individual input node to each hidden layer node and will then generate a weight between each hidden node to the output node. We set the number of nodes in the hidden layer to 20 because we found that having more

didnt lead to better results and made the network more complex while also increasing the run time, which is best done through a trial-and-error process [2]. The neural network (NN) will randomly generate a set of inputs between -5 to 5 and use these inputs as the neurons in the input layer of the NN. In order to feed through the neural network, the activation function was called on the weighted sum of the inputs to a node to see whether the node fires or not. The activation function used was the tanh function.

The design for implementation was inspired by another python project [1] which represents the neural network as weight matrices representing synapses. We initially attempted to use the sphere function itself to generate our training outputs, though it seemed inappropriate in the CoCo platform so we used the CoCo fun() function instead. We also looked at using backpropagation to train our network to familiarise ourselves with the workings of a neural network. Outputs from the neural network needed to be scaled by a factor of 1000, since the activation function returns values in the range -1,1.

Genetic Algorithms

The NN utilises two genetic algorithms (GA) to evolve and optimise the functions once the neural network has been trained. The first genetic algorithm utilises the hill climbing method in which the fittest individual of each population is carried forward to the next generation and the rest of the population is filled with mutations of this fittest individual. The hill climbing algorithm was suitable since it is an easy to implement yet also effective method of evolution [3]. Chromosomes for the first GA were encoded as a list of weights for the network where the first n weights represent weights from the input layer to the hidden and so forth. The population for the first generation was initialised as a matrix of random weights from -1 to 1 and mutation was carried out by each weight in the chromosome having a percentage probability of being mutated to a new random weight. Fitness for each chromosome was determined by the difference between the expected function fitness for the provided inputs and the neural network predicted fitness.

Once the NN was trained, the algorithm was used to optimise a dimension of inputs in the CoCo platform. For this section of the project we used another genetic algorithm however used a different approach. Instead of a hill-climbing method we decided to use elitism and mutation to evolve our population. This algorithm involved a smaller range of inputs so utilised reduced generations and mutation rate versus the first algorithm.

Algorithm Settings

In order to create an effective algorithm, careful thought had to be put towards parameter settings for the genetic algorithms. Having an algorithm with too high of a mutation rate or too many generations can result in the algorithm over-mutating certain chromosomes. Both algorithms used a relatively small population size of 50 individuals. We found that anything higher had a large impact on the runtime of the experiment and anything less resulted in reduced accuracy in the results. Both algorithms also used five generations of individuals and a mutation rate of 0.1. We carefully optimised the mutation rate in order to prevent the algorithm becoming a random search. For the second algorithm we also introduced elitism with a value of 10. This means that after each generation, the ten fittest individuals are carried forward to the next. Having an elitism of ten allowed us to create a variety of solutions to our optimisation problem.

3 Results

The algorithm outputted results of training the NN and input optimisation to the console output in a readable manner. An example of said console output is shown below:

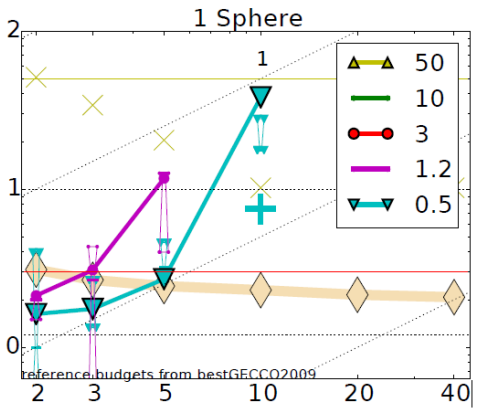
```
Initialising training data...
100 sets of training data initialised.

Beginning training...
   10 patterns learned | Average error: 0.485721932907
   20 patterns learned | Average error: 0.704754196864
   30 patterns learned | Average error: 0.687099875599
   40 patterns learned | Average error: 0.722842699766
   50 patterns learned | Average error: 0.660703432045
   60 patterns learned | Average error: 0.647744016231
   70 patterns learned | Average error: 0.643755072314
   80 patterns learned | Average error: 0.630686289816
   90 patterns learned | Average error: 0.621809951593
Training complete. 100 patterns learned with average error 0.613714253293

Optimising inputs with ANN...

Task achieved:
  Target fitness: -13.07999999
  Optimised inputs: [ 4.15104125 -4.36349704 -4.38398659 -1.54670743]
  ANN predicted output: -997.204490719
  CoCo actual output: 97.2080574772
  Error of ANN: 1094.4125482
```

The algorithm managed to learn each individual set of training data very well and could return an error value of below 1 for each output but could not manage to remember the data once it had moved on to a different input. This means that the algorithm can not predict previously used data well and means the algorithm can take longer than it should due to going through similar data multiple times to find the optimal value. This was shown by error values in the neural network prediction, which reached error values of 1000+. One of the CoCo generated graphs are shown below, the rest of the results are available on GitHub.



4 Discussion

Due to a lack of experience using CoCo the algorithm took longer to get started with than expected. As mentioned before in the Artificial Neural Network section, we had started creating the Sphere function as our activation function but discovered that this function did not work well with CoCo and were forced to change to receive better results. Due to this and the lack of experience we think that CoCo might not have been the best application environment for creating this algorithm and that we may have been able to do better using other applications. This lack of understanding in CoCo also lead to some issues with generating the test input. The test input that we generated was generated randomly at a low value but with more understanding of the CoCo platform we believe we could have generated more appropriate test data to train the algorithm with. After creating the algorithm we found there were possibly other ways that we could have implemented it.

One of the best methods that we found was by using NeuralEvolution of Augmented Technologies (NEAT) which outperforms the best fixed-topology method on a challenging benchmark reinforcement learning task [2]. We also think that if we had some more time we might have been able to perfect some of the variables such as the mutation rate. We have managed to achieve a very good error rate but the inputs could be being over mutated and could be negatively affecting the code.

5 Conclusions

In conclusion, we found this project challenging yet rewarding and has allowed us to familiarise ourselves with the usage of neural networks and genetic algorithms. One of our main issues in implementing this project was using the CoCo platform and more time experimenting with it could have outputted more promising results. If we were to implement a neural network again, we would possibly look into using a different method of evolution e.g. NEAT (neuroevolution of evolving topologies) or more advanced genetic algorithms to improve the evolution of our network.

References

[1] iamtrask. (2015) A Neural Network in 11 lines of Python. http://iamtrask.github.io/2015/07/12/basic-python-network/

[2] MIT Press. (n/a) The MIT Press Journals. http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf

[3] Frank Vavak , Terence C. Fogarty. (n/a) Comparisons of Steady State and Generational Genetic Algorithms for Use in Nonstationary Environments. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=542359