Biologically Inspired Computation

F20BC

Coursework 1

Nicholas Robinson
Sam McNaughton

https://github.com/kyleNR/Genetic-Algorithm-Neural-Network-Evaluation

November 14, 2016

1 Introduction

This report attempts to explain and evaluate the decisions behind our implementations of the Artificial Neural Network(ANN) and Evolutionary Algorithm(EA) that we have developed for this project.

1.1 Evolutionary Algorithm

Evolutionary Algorithms attempt to simulate the real world evolution process on a population of possible solutions to the problem that is being worked on.

For our implementation we use the Genetic Algorithm(GA) type of evolutionary algorithm as it seeks the solution by performing a variety of evolutionary methods to the population such as

- Selection Method
- Crossover
- Mutation

1.1.1 General Description of Algorithm

Our genetic algorithm follows the traditional flow in terms of the order in which its evolutionary processes are applied to the genotypes.

1. Generate Initial random population.
2. Assign fitness values to each member of the population accordingly.
3. Use selection method to determine which chromosomes will reproduce.
4. Use crossover to produce an offspring from the winners of the tournaments.
5. If chosen apply mutation to the child.

Steps 2 - 5 are repeated until we reach the intended number of generations. We then take the best result that we have found and use those weights as the weights within the neural network.

1.2 Implementation Decisions

Our code contains a number of features than make the implementation unique:

The algorithm attempts to minimise the difference between the results returned by the neural network compared to the actual intended output.

Within our algorithm each genotype in the population represents the set of weights that will be used in the neural network.

We encode the chromosomes to represent the weights within the neural network. We use value encoding as opposed to a traditional binary approach as we needed the solution to be usable by our neural network.

If our implementation was essentially a brute force optimisation we would be guaranteed to find very good results given enough time. The issue with this vein however is that this solution is generally far too slow to be of any use. Whereas if we only select the best results from each iteration we will conversely get poor results quite quickly.

Therefore our algorithm attempts to balance between always selecting the best results from the population, through tournament selection, and just performing a random search over our population, through mutation. This should ensure that our implementation produces good results within a relatively fast time period, Ideally we shall discover the global minima within our search space however this is unlikely due to the approximate function nature of a genetic algorithm and the limited time given to run.

2 Experiment

2.1 Neural Network

2.1.1 Topology

The Neural Network implementation was designed to be able to be changed easily. This ultimately meant an input parameter being used to define the size of the input layer dynamically to allow it to calculate a function with any possible dimension size.

The final topology of the network was an input layer of size dimension, a hidden layer of size 4, another hidden layer of size 3 and an output layer of size 1.

This topology was chosen as it could replicate a function well without creating a higher time cost and no notable error reduction.

2.1.2 Activation Function

The activation functions used in the network were switched around a lot until ultimately we decided to use the Bent Identity function to allow an output size from ∞ to -∞ without scaling the output.

2.2 Genetic Algorithm

2.2.1 Data Representation

The GA (Genetic Algorithm) uses a list of real numbers to hold the weights for the NN (Neural Network). Although this is different from the matrix structure used within the network, it allows the GA to be used on any topology of NN.

2.2.2 Selection

Tournament selection was the selection method chosen to be implemented. It takes 4 random chromosomes and finds the fittest out of the first two, and the fittest of the second two. With the two winning chromosomes it uses crossover to create a child to go into the population of the next generation. It was used as the selection method as it has been proven to be effective[1][2]. Although a larger Tournament size could be used to reduce the number of generations needed to reach a local minimum, it was decided to trade this to achieve a better time to run the program in.

We use Elitism to keep the best chromosome of every generation and adding it to the next generation. It allows it to reach a local minimum faster and also to stop the GA regressing if the population does not find a better fitness. While it is possible to use elitism of more than one chromosome between generations a size of one was chosen as it would keep the fitness from decreasing without harming genetic diversity.

2.2.3 Modification

The GA uses a Two-point crossover for its crossover function. This is a simple method, although slightly more complex than One-point crossover. It takes two points as indexes, takes the first parent up to that point and after the second point, then takes the second parent between the two points to create a full child. While this could have been used to create two children, it was decided to use this to only create one to create a larger degree of diversity in the population.

The mutation function is basic but effective at slight variations in the output. It takes the chromosome and loops randomly between 1 and the number of weights in the chromosome. In the loop it selects a random weight and multiplies it by either 0.75 or 1.25.

The mutation chance is the probability that the child created with the crossover function will be mutated before being added to the new population.

2.2.4 Generations

The generations is the number of iterations for the GA to run through. It is set as a default value of 200.

2.2.5 Population Size

The population size is the number of chromosomes to use in each generation in the GA. It is set as a default value of 50.

2.2.6 Training Data Set

The size of the dataset used to train the GA is set with the *DataGenerator* file. The higher the number of sets of data the GA will train with, the more accurate it can be theoretically. In reality the accuracy is based much more on the Generation, Population, and Mutation Chance size. The larger the dataset, the longer the GA will take to train as each fitness test will take longer.

2.3 Neural Network Compare

The Experiment to compare the Neural Network against the COCO functions was set using the *exampleexperiment* file from COCO. This file will train the Neural Network based of the dimensions and functions given before running and comparing each pair of functions.

2.3.1 Functions

The functions chosen to compare against COCO were the noiseless functions. These are represented in COCO with the function IDs from 1 to 24.

2.3.2 Dimensions

The Dimensions chosen to compare the COCO functions with the NN are 2,3,5. The dimensions were chosen to keep the time to run the experiment down while also being compatible with

COCOs post-processing. The Neural Network can handle any size of dimension, but the time to evolve the weights will increase in relation.

3 Results

3.1 Finding Best Weights

3.1.1 Settings

Elitism provided the best values as it never regressed in fitness. When disabled, the population's fitness could and in most cases regress backwards in some generations.
The Mutation Chance only affected the accuracy slightly. It stayed around the same value as long as it was greater than 0.1.
Population Size and Generations both caused the error to decrease but at a cost of time running the program. Generations caused only a minor increase in accuracy about 30.

3.2 Compare Functions

3.2.1 Parameters

Genetic Algorithm Parameters

- Generations = 200
- Population = 30
- Mutation Chance = 25
- Elitism = True
- Dataset Size = 50

Function Comparison Parameters

- Dimensions = 2,3,5
- Functions = 1-24
- Instances = 1-5, 41-50

3.2.2 Time

Running the *NNCompare* program took 1.34 hours to fully complete. This included the time spent Training the network for each dimension and function and running the Monte Carlo selection optimisation. More accurate results may have been attained by using higher parameters for generations, population size and dataset size but this would have resulted in the program running for longer.

3.2.3 Error

The error between functions varies quite a lot between the function being used to test. Originally the goal was to have an accuracy of 1 x 10^{-8}. This was quickly proven to be not possible for us.

The mean optimised error over the 3 dimensions and 24 function is 71673030.893774.

Unoptimised results

Dimension Error Breakdown

- Dim: 2 Mean Error: 66528724.344014
- Dim: 3 Mean Error: 34583987.854396
- Dim: 5 Mean Error: 37929540.188898

Function Error Breakdown

- Function: 1 Mean Error: 179.916752
- Function: 2 Mean Error: 13327474.628473
- Function: 3 Mean Error: 1304.319599
- Function: 4 Mean Error: 1641.492063
- Function: 5 Mean Error: 405.481184
- Function: 6 Mean Error: 347381.234899
- Function: 7 Mean Error: 784.458142
- Function: 8 Mean Error: 133708.078350
- Function: 9 Mean Error: 68374.590549
- Function: 10 Mean Error: 19299998.622307
- Function: 11 Mean Error: 15837652.352132
- Function: 12 Mean Error: 861228409.180600
- Function: 13 Mean Error: 1639.564259
- Function: 14 Mean Error: 199.024445
- Function: 15 Mean Error: 631.727322
- Function: 16 Mean Error: 185.872345
- Function: 17 Mean Error: 212.611501

- Function: 18 Mean Error: 321.394856
- Function: 19 Mean Error: 296.492377
- Function: 20 Mean Error: 57159.200084
- Function: 21 Mean Error: 262.516522
- Function: 22 Mean Error: 375.160063
- Function: 23 Mean Error: 123.937418
- Function: 24 Mean Error: 242.677308

Optimised results

Dimension Error Breakdown

- Dim: 2 Mean Error: 180162184.663644
- Dim: 3 Mean Error: 64892443.572706
- Dim: 5 Mean Error: 32345721.778468

Optimised Function Error Breakdown

- Function: 1 Mean Error: 164.804346
- Function: 2 Mean Error: 10896935.209143
- Function: 3 Mean Error: 1294.011928
- Function: 4 Mean Error: 1276.370219
- Function: 5 Mean Error: 375.322413
- Function: 6 Mean Error: 272930.882969
- Function: 7 Mean Error: 702.017186
- Function: 8 Mean Error: 114860.286929
- Function: 9 Mean Error: 56482.608131
- Function: 10 Mean Error: 15736200.667834
- Function: 11 Mean Error: 15228843.708646
- Function: 12 Mean Error: 733934812.030683
- Function: 13 Mean Error: 986.166639
- Function: 14 Mean Error: 190.375155
- Function: 15 Mean Error: 789.724851

- Function: 16 Mean Error: 206.320422
- Function: 17 Mean Error: 199.310447
- Function: 18 Mean Error: 267.512007
- Function: 19 Mean Error: 307.240868
- Function: 20 Mean Error: 47965.536369
- Function: 21 Mean Error: 251.182707
- Function: 22 Mean Error: 926.811691
- Function: 23 Mean Error: 121.188542
- Function: 24 Mean Error: 233.393122

4 Discussion

Our results show that our implementation of the network and algorithm does improve the error between our actual and intended results.

However while our results show that optimisation has occurred in the large majority of cases they also show that our solution does not approximate the intended function as well as we would have hoped.

Depending on the function being examined we return wildly different mean error results. This generally means that the more complex the function that is being approximated the less effective our genetic algorithm is. This is shown by our algorithms very poor performance for the functions 11,12 and 22, those being the Discus, Bent Cigar and Gallagher's Gaussian 21-hi Peaks Function functions accordingly.

Our neural network could possibly have fallen victim to the problem of over fitting to our input data. This would mean that our network responds very well to its training data but fails to generalise well. This could be caused by the topology of our network, due to the fact it includes several hidden layers between the input and output. It may be too deep for what we want it to do. This is problem is tricky to solve as we could reduce the complexity of our network however this may lead to it not being complex enough to find all of the information that it needs to find a solution.

5 Conclusion

We have established in this report that evolutionary algorithms, in our case a genetic algorithm, prove to be an effective solution to the problem of neural network weight optimisation.

Our results show that our genetic algorithm tends to miss the global minima and as such cannot be described as an exact algorithm, that is one that is guaranteed to reach the best fitness solution within the search space. The term used to describe our algorithm is an approximate function, one that attempt to find near optimal solutions within an acceptable time-frame.

This coincides with the results found within [3] that neural networks are good at finding solutions to complex non-linear functions. This journal article also states that genetic algorithms are good at finding global minima due to the fact that their solutions search in various different directions at once.

However back propagation continues to be extensively used in various real world practices. We can see how this has occurred by looking at our results. Our experimental data showed that the performance of our algorithm generally decreased as the dimension of the problem increased. This means that problems that input a vast quantity of data are one of the various cases in which back propagation generally outperforms genetic algorithms as the time it would take for the algorithm to optimize the large number of weights in this problem would be far to long to be of any particular use.

With more time we could attempt to improve the performance of the genetic algorithm. A quick improvement that could be applied would be to vastly increase the dataset that helps train the neural network.

From this project we have learned how to properly assess the performance of a genetic algorithm. We performed this through analysis of the results produced by the COCO black-box platform. By comparing our results to the results obtained from COCO we have been able to produce a suitable benchmark of our genetic algorithm.

We have also learned that the topology of a neural network can play a part in its performance. A topology that is too complex for the problem may lead to over-fitting while a network that is too simple may lead to the network failing to capture the information required to solve to problem.

Additionally we learned that our relative lack of experience with the benchmarking software, the COCO package, has lead to various set-backs and restarts in the project. We could have mitigated this factor by sufficiently reading the COCO documentation and examining the limited on-line examples before starting the project. This would have given us a wealth of knowledge on the COCO black box system and would have prevented us falling into the preventable pitfalls that hindered our progress.

References

[1] G. Rawlins, Foundations of genetic algorithms, 1st ed. San Mateo, Calif.: M. Kaufmann Publishers, 1991, pp. 78-82.

[2] F. Vavak and T. Fogarty, ”Comparison of steady state and generational genetic algorithms for use in nonstationary environments”, Proceedings of IEEE International Conference on Evolutionary Computation.

[3] Randall Sexton,Robert E. Dorsey,John D. Johnson, Toward global optimization of neural networks: A comparison of the genetic algorithm and backpropagation, Decision Support Systems, Iss. 22, vol. 2, 1998, pp. 171-185.