Biologically-Inspired Computation

Coursework One

Heriot-Watt University

Group Members:

Angus Hamilton

Filip Bartoszewski

Part One:

Introduction:

In part one of the assessment, our goal was to use a hybrid genetic algorithm – multi-layer perceptron approach to evolve a neural network to approximate a given benchmark function. Then, using a generic algorithm, we were to minimize that benchmark function and it's MLP approximation.

In order to tackle this, we created two different genetic algorithms to train MLP's. The first was an augmented hill-climbing algorithm which takes a randomly selected point within the weight matrix chromosome and tests 6 different changes to it and goes with the change that gets closer to the desired value. The second was a steady-state, 5-k tournament with mutation at a random point.

All the files for this part, which include code and experimental data can be found here:

https://github.com/A-Hamilton11211/BIC-Project

MLP Structure Description:

Utilizing the ideas presented in class, we implemented the MLP using a series of weight matrices that represent the connections within every layer of our network. The topology of the network grows in relation to the dimensionality of the problem, with the number of neurons in each layer growing in response to the dimensionality of the problem (e.g. if the network gets 5-d input, the network will have 5 neurons in each layer). These neurons take three inputs, due to computational complexity and efficiency issues. Aside from the growing topology, the activation function and the weight network are both evolved during the operation of the genetic algorithm. As such, the chromosomes consist of both the activation function, and the weight network in a 1-D matrix, but it should be noted that the activation function and the weight network are often kept separate.

Algorithm Descriptions:

There are two main algorithms used in evolving the genetic algorithm:

HILLTRAIN – This function takes an input, dimension, chromosome, activation function, and benchfunction ID, and returns the output of the evolved MLP, the desired values given the inputs, the evolved chromosome, the evolved activation function, and sum of errors. This algorithm evolves the MLP using a complicated form of hill-climbing, which mutates 6 potential points the chromosome could shift to and their errors. The point with the lowest error is chosen, and the chromosome and activation function are changed accordingly. This process loops until the error of the algorithm is below one for each input (under 15 in total).

STEADYTOURNAMENT – This function takes an input, dimension, benchfunction ID, and a number of training iteration, and returns the output of the final algorithm, the evolved chromosome, the evolved activation function, and the total error of the evolved MLP over all inputs. This algorithm evolves the MLP using tournament selection, which takes 5 random members of the 50 member population and returns the one closest to the desired value. This 'tournament winner' is then mutated at a random point, either adding 0.5 to that chromosome section, or removing 0.5 from that chromosome section. Depending on which mutation provides the lesser error, that mutated chromosome is tested against the member of the population with the highest error. If the mutated chromosome's error is lower than the member of the population with the highest error, the worst member is replaced and the process is repeated. However, if there is no conceivable progress being made in evolving the network, a generational 'wipe' is done where all elements of the population are removed, save the best member of the population. This is like a generational algorithm with elitism.

The reason why this sort of algorithm was chosen was due to the results of a paper by Vavak & Fogarty (1996).

MY_OPTIMIZER.m was additionally altered to implement a steady state tournament algorithm using crossover, in much the same fashion as the steadytournament algorithm.

Results of Experiments:

The two main experiments were done over all dimensions, on the sphere function(f1). However, these algorithms work on every single of the 24 noiseless functions. All of the coding for part one was done in octave, due to its simplicity in implementing matrices. It should be noted that these experiments took quite a long time to complete, which is why testing over additional functions besides the sphere function was unable to be carried out.

Unsurprisingly, it seems that the steadytournament algorithm performed much more reliably and accurately than hilltrain. Both algorithms seemed to have an easier time at certain dimensions, which was expected due to the fact that certain dimensions suit certain benchmark functions. Error was quite variable over both functions, but markedly less so in steadystate. Hilltrain was also much slower than steadytournament.

Part Two:

Introduction:

Part two of the assessment focused creating an evolutionary algorithm to minimize a set of functions from the COCO benchmark.

At first we wanted to try out some of the basic algorithms described during the lecture to find out how will they work and how long the computation will take. We have created a set of simple and advanced genetic algorithms for this purpose.

All the files including code, data and graphs are available under the following link:

https://github.com/filbart/Uni_stuffThe experiment

All experiments where conducted by changing the code of the run_optimiser function within the exampleexperiment.py python file. First we have written some simple algorithms using the basic principles of EA. After deciding which work best we tried to implement more than one method in the optimizer. We have discovered that hillclimbing and tournament selection give the best results we tried to use them both in one algorithm-PURE_SMART_TOURNAMENT. We have run simulation using it three times, to check how changing the parameters would affect the results. Parameters changed included population size, number of iterations and standard deviation (for normal distribution while mutating our chromosomes).

Description of algorithms:

PURE_FILIP_SEARCH – We create a random population, during each iteration we sample two random chromosomes from it. Better one becomes a parent. Child which is a result of mutation (normal distribution with standard deviation equal 1) takes place of the worst chromosome in the population.

PURE_UNIFORM_CROSSOVER – We create a random population, during each iteration we grab two parents and create a child, grabbing each element randomly from it's parents chromosomes. Child replaces the worst chromosome in the population.

PURE_ONE_POINT_CROSSOVER – Like the previous algorithm, but instead of choosing "genes" randomly each time, we choose a random number k within the size of the chromosome. First k genes are chosen from the first parent, the rest from the second.

PURE_HILLCLIMB – We have only one individual. During each step we check if it's child created using normal distribution (StD = 0.3) would have better fitness. If yes, child becomes the individual.

PURE_TOURNAMENT – We create a random population, during each step two chromosomes undergo a tournament. Looser is replaced by a child of the winner created by mutation (normal distribution, StD = 0.6). Code for this algorithm was lost due to corruption while creating a backup, there are results and data for it though.

PURE_SMART_TOURNAMENT – Like the previous one, but each gene is tested during the mutation, so parts of the chromosome are changed only if mutating them improves the overall fitness of the child (each gene is tested separately for this purpose). Experiment run three times with different parameters. First time +300 timesteps, half of the population, Standard deviation in the normal mutation 0.6; second time +500 timesteps, normal population, StD=0.3; third time +700 timesteps, population full-20 chromosomes and StD=0.2.

GENERATIONAL_FANCY– We create a random population. During each step we choose 2*(popsize-1) pairs from them and create popsize-1 children applying k-point crossover to the pairs. From the original population only the best chromosome stays "alive" rest is replaced by the children.

MUTANT_GENERATION - Similar to FANCY algorithm, but we use uniform crossover for the children and then we mutate them, so we change our genepool slightly with each generation. This method uses population with half of the size then other algorithms.

MUTATING_UNIFORM_CROSSOVER – Like PURE_UNIFORM_CROSSOVER but children are slightly mutated to change the genepool after each step.

All the algorithms using mutation were implementing a check- does the mutation put the chromosome outside of the test area. If yes, the genes would be corrected to put the point back in the space of the experiment. All the mutations used normal distribution.

Results and Comparing Configurations of Algorithms:

We judged our algorithms by comparing the graphs created by the rungeneric.py code with the ones of the provided dummy algorithm PURE_RANDOM_SEARCH which is default one in COCO and samples random points until it finds the solution. PURE_FILIP_SEARCH seemed to work slightly better than RS for the lower dimensions, for the higher ones there was no visible difference. PURE_HILLCLIMB and TOURNAMENT were giving visibly better results for lower dimensions (curve of the postprocessed graphs visibly lower when compared with RS) and slightly better for higher ones. Based on this we decided to create a better steady state algorithm combining those two methods. Sadly, the crossover methods and the generational ones gave next to no results. Rarely they would find the solution, even for the lowest dimensions. We managed to receive data files from COCO for those methods, but because they would not converge on the solution the post processing script would not be able to create any graphs. In addition to it, running the generational algorithms was very time consuming, sometimes taking even up to two hours to finish the computation (to compare, the "slowest" steady state algorithm would finish running in COCO after 5 minutes, even with increased number of time steps). All the graphs and data files are available to access at github.

We have tried to come up with a more advanced algorithm and created PURE_SMART_TOURNAMENT. It is a steady state algorithm with tournament selection, and hillclimbing implemented for each gene in the winning chromosome, using mutation with normal distribution. This algorithm proved to be the best out of all that we have tested. It managed to minimize most of the functions up to 5th or even 10th dimension so we decided to see how changing it's parameters would affect it. Smaller population rapidly decreased runtimes of the computation and lowered the graph curves for the lower dimensions, but decreased the amount of time solution was found for higher dimensions. Increasing the number of time steps did not affect the computation for the lower dimensions, increased runtime for higher ones, but for some functions we started to find solutions in higher dimensions. Most curious was the effect of changing the StD while mutating the chromosomes. Bigger deviation seemed to make the graph curves more steep, resulting in much better results for smaller populations and lower dimensions, but worse results for higher dimensions and more complicated functions. Smaller deviation had opposite results, enabling us to minimize the function in higher dimension, but returning worse results for simpler cases. This shows that mutating the standard deviation of the distribution might prove a good idea in some algorithms. We decided that crossover methods might prove more useful when combined with mutation, simply because if the starting genepool would not include a permutation of genes that would be the solution, the method would fail. Two algorithms, MUTANT_GENERATION and MUTATING_UNIFORM_CROSSOVER proved to be quite successful and in some cases, were able to find solutions when hillclimbing proved not sufficient (mostly functions with large amount of local minima and maxima).

Conclusions:

Genetic algorithms and evolutionary programming are a very flexible tool. Functions that were minimized in the benchmark represent a wide array of problems, and our algorithms were able to evolve and were able to solve given problem most of the times. Some algorithms like hillclimbing and tournament selection proved better and easier to use then the others (crossovers without mutation). Steady state methods gave much better results than generational ones, both in terms of runtime and minimizing the functions. One reason for this, might be that for crossovers, if the solution wasn't a permutation of the genes in our whole population at the start, the method would not be able to converge. That idea might require some additional testing in the future. Results for crossovers with mutation were much more satisfactory.

Bibliography:

Vavak, F. and Fogarty, T. (1996). Comparison of steady state and generational genetic algorithms for use in nonstationary environments. In: *Proceedings of IEEE International Conference on Evolutionary Computation*. [online] IEEE. Available at: https://vision.hw.ac.uk/bbcswebdav/pid-2324479-dt-content-rid-2116870_2/courses/F21BC_2016-2017/Comparison%20of%20steady%20state%20and%20generarional%20GA.pdf [Accessed 13 Nov. 2016].