Coursework 1 of Biologically Inspired Computing

Black-Box Optimization

Genetic Algorithms and Neural Networks

by Francisco Mendonça
H00254547

Available on GitHub: https://github.com/fjpavm/BioInspiredCW1

Table of Contents

Software Developed

In this section we describe the software that was implemented in python. The code is available on Github (https://github.com/fjpavm/BioInspiredCW1).

We start with the general Genetic Algorithm, its parameters and operators. We then describe the minimal neural network library developed. Finally the implemented genetic operators for optimizing over continuous functions and over our neural network implementation are described.

All of the parts of the Genetic Algorithm that are not problem dependant where gathered in the file `GeneticAlgorithm.py`. The the genetic operators and random individual creation, that are problem dependant parts, were put into two files `VectorGAOperators.py` and `NeuralNetGAOperators.py`. The first file implements them for general multi-dimensional real-valued vectors. The second file implements them for our neural networks.

The Neural Network implementation is in the file `NeuralNetwork.py`

General Genetic Algorithm

The structure for this genetic algorithm was take from the class slide [Biologically Inspired Computing 2016 EA].

The generic algorithm is represented by the python class `GeneticAlgorithm`. It's structure was split into two parts: Initialize and Run.

Initialize

Implemented in the method `initializePopulation` of `GeneticAlgorithm` class. It utilizes a function to generate random individuals that must be provided and is problem dependant

Psedo-Code:

Discard all information of previous run

Until initial population number is reached

Create new individual and evaluate its fitness

Update best individual and add individual-fitness pair to population

Run

Implemented through methods `run` and `advanceOneGeneration` of `GeneticAlgorithm` class

The Run part goes through the Selection of parents, the Genetic Operators to create the children and the Population Update for each generation until some termination condition is met.

Depending on configuration the population update can keep the **Best** from last generation to the new generation (forcing elitism) and it can introduce a completely

Alien random individual (that increases variability in the population). The idea of introducing completely random individual comes from localization particle filters where it is done to allow re-localization after kidnapping [Thrun, Burgard and Fox 2006]

It utilizes a Parent Selection Strategy and a Generational Selection Strategy that are both problem independent and can be provided at object creation or a default strategy will be used. It also utilizes Genetic Operators that are problem dependant. These are the Mutation and the Crossover. Only the Mutation needs to be provided for the algorithm to be able to run. If a crossover operator was provided Crossover is used to generate a child with probability equal to the `crossoverRate` parameter of the algorithm

Psedo-Code:

Until a termination condition is met

Select `patentRate*populationSize` Parents with Parent Selection Strategy

Create `childRate*populationSize` children with Genetic Operators

Update population with Best and Alien and the rest with Generational Selection Strategy

Update best individual found and increase generation by 1

Parent Selection Strategies

The implemented parent strategies where the ones described on the class slides [Biologically Inspired Computing 2016 EA]

-**Roulette Wheel** was modified to allow negative value fitness and implemented in a class called `ProbabilisticFitnessSelection` inspired by its description in the book Machine Learning [Mitchel 1997]

-**Tournament Selection** where the tournament size is configurable was implemented in the class called `TournamentSelection`

-**Rank Based Selection** where the bias is configurable (bias 1 is normal Rank Based Selection) is implemented in the class `RankSelection`

Generational Selection Strategy

The only implemented generational selection strategy was one based of elitism described in the slides [Biologically Inspired Computing 2016 EA]. The worst of the previous generation are replaced with the children and the best are kept for the next generation. This was implemented in the class `ChildrenAndBestCurrent`.

Neural Network Library

The Neural Networks where implemented a through matrix representation in the class `NeuralNetwork` by using the NumPy library.

The class constructor takes in the layer sizes (including input and output) and optional function vectors to use for the hidden layers. When no function vectors are present the sigmoid function is used.

The class is callable so a network can be used as a function in the python code. The weight matrices are accessible and settable directly through the `getWeightMatrix` and `setWeightMatrix` methods. Each weight can also be individually manipulated through the `getWeight` and `setWeight` methods. There are also methods to for manipulating the functions for the hidden layers. These are the `getFunctionForNodeOnHiddenLayer` and `setFunctionForNodeOnHiddenLayer` methods. Only two functions where implemented for use with the neural networks the sigmoid and a linear function (just lets the value through unaltered).

A method for performing a step of the stochastic back-propagation algorithm was also implemented in the `backpropagate` method.

The formulas and algorithmic knowledge from the book Machine Learning [Mitchel 1997] where instrumental in providing the knowledge for this implementation.

Vector Genetic Operators

The operators are gathered in a single class called `VectorGAOperators` and are separate methods of this class

New Individuals

The new individuals are created at random within configurable minimum and maximum values for each dimension. If no values are provided -5 and 5 are used for each dimension as it is the range used in COCO [Hansel et al. 2016]. The method that implements this is called `createRandom`

Mutation

Initially mutation was implemented as simply perturbing the current vector by one drawn from a normal distribution. Through the initial experiments it was seen that that was too random and so a `deviationBase` parameter was added so a the normal distribution could be multiplied by the deviation and become a zero mean Gaussian distribution with the given deviation. While trying to find a suitable deviation it became apparent that small deviations would lead to being able to achieve the targets but would slowdown the algorithm immensely. A random approach where the `deviationBase` was determined with equal probability to be raised to one of the values given by the `deviationExponents` parameter was devised. The method that implements this is called `mutate`

Crossover

The only crossover strategy implemented was the box crossover [Biologically Inspired Computing 2016 EA]. The method that implements this is called `boxcrossover`

Neural Network Genetic Operators

The operators are gathered in a single class called `NeuralNetworkGAOperators` and are separate methods of this class

New Individuals

The new individuals are created by `createRandom`. They are created with a random number of hidden layers (up to `maxHiddenLayers` provided on class construction), randomized hidden layer function vectors and matrices taken from a normal distribution with 0.1 standard deviation

Mutation

Mutation was implemented in the `mutate` method and associated sub methods: `mutateBackpropagate`, `mutateWeights`, `mutateFunction` and `mutateNodes`.

The `mutate` method manages the probabilities of each type of mutation and then calls the specific method.

The `mutateBackpropagate` method preforms a randomly sized back-propagation step based on a few of training samples. The `mutateWeights` method preforms a perturbation drawn from a normal distribution with random deviation to one of the weight matrices. The `mutateFunction` method mutates a hidden node so it uses a different function. Finally, the `mutateNodes` method adds or removes a node from one of the hidden layers.

Crossover

No crossover operator was developed due to the complexity of devising a crossover when the network can have different shapes

Genetic Algorithm Experiments

To have a baseline for our experiments the example experiment with the Pure Random Search algorithm was run and the data kept for future comparisons

The data gathered from each set of parameters was put under the Results folder of the repository. Separated into sub folders based on the variance/deviation used for the mutation operator. The folder for the data was given the following name structure for experiments with just the Genetic Algorithm: GA followed by P `PopulationSize` C `ChildRate` P `ParentRate` 'R' or 'T' for Rank or Tournament selections and `SelectionParameter` (Bias for Rank Selection and tournament size for Tournament

Selection). Some optional parameters are appended to the main formate. Crossover Rate is appended as C `CrossoverRate`, Alien inclusion is appended as 'Alien' and Best inclusion is appended as 'Best'

Parent Selection Strategies

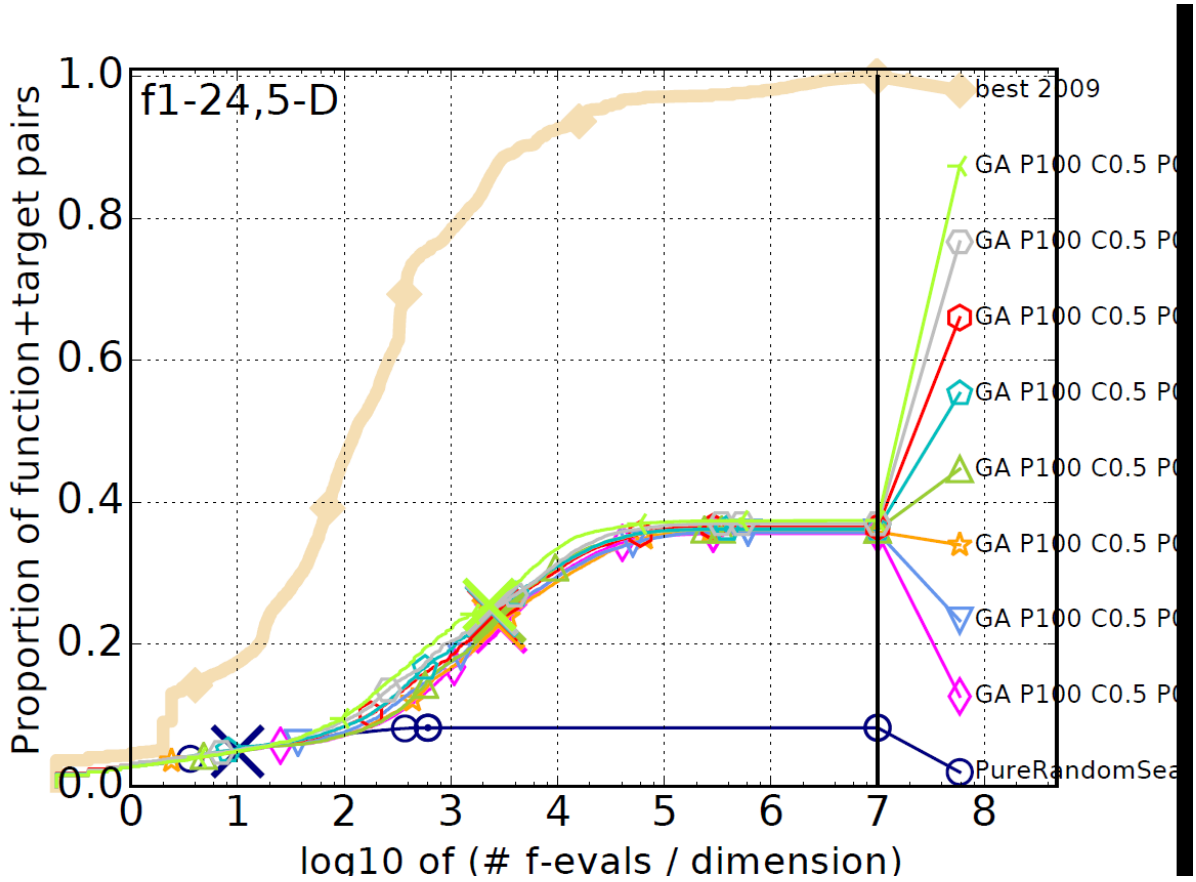
For this experiment the mutation variance/deviation was 0.1 and only the 5D functions where used. The parent rate was set to 5%. Population size 100 and child rate 50% (half way between steady state and generational approach).

For the Parental Selection the following where tested:

-Tournament Selection with sizes 2, 5, 10 and 20

-Rank Selection with bias 0.5, 1, 2, 4

Results and Drawn Conclusions



In this graph of the results over all noiseless functions we can again see very little difference between the strategies but we do see a marked increase in the results overall. Meaning that the reduction of the variance/deviation had a significant impact.

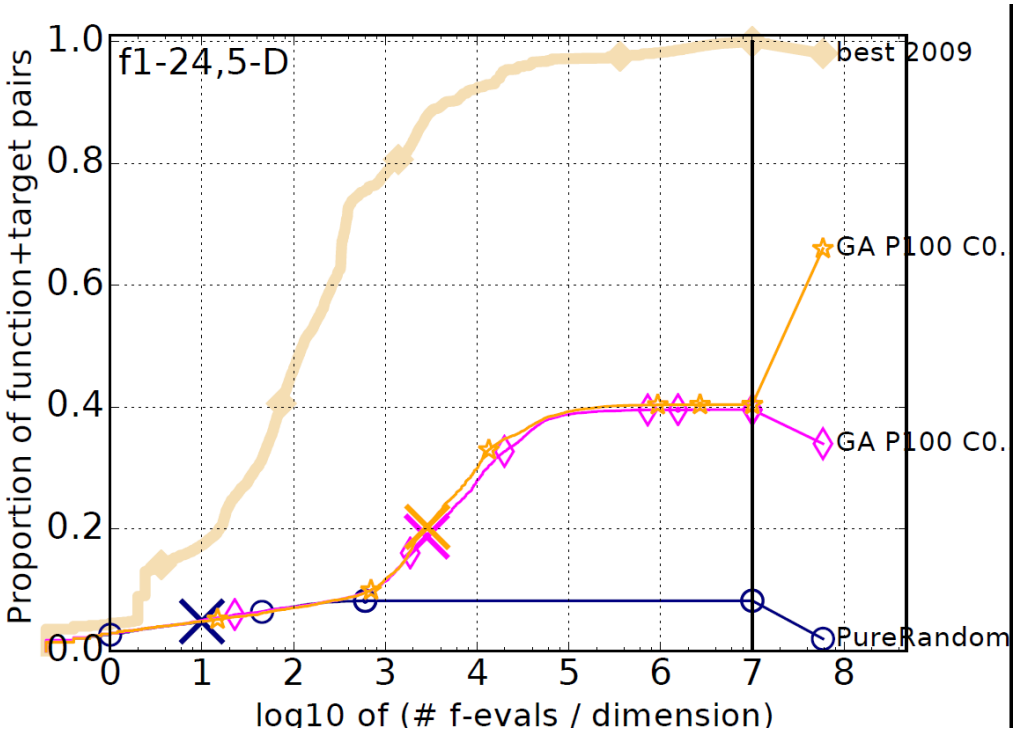
Δf_{opt}	1e1	1e0	1e-1	1e-2	1e-3	1e-5	1e-7	#succ
f1	11	12	12	12	12	12	12	15/15
PureRandomSearch	**7.6** (13)	∞	∞	∞	∞	∞	∞*50*	0/15
GA P100 C0.5 P0.05 R0.5	21 (13)	177 (93)	241 (117)	283 (64)	1854 (2138)	∞	∞*1e4*	0/15
GA P100 C0.5 P0.05 R1	29 (33)	159 (60)	212 (50)	263 (52)	1331 (1879)	∞	∞*1e4*	0/15
GA P100 C0.5 P0.05 R2	15 (16)	107 (26)	153 (52)	182 (54)	797 (245)	∞	∞*1e4*	0/15
GA P100 C0.5 P0.05 R4	**6.1** (18)	83 (26)	117 (37)	140 (22)	2146 (1258)	∞	∞*9250*	0/15
GA P100 C0.5 P0.05 T2	23 (33)	149 (42)	203 (50)	243 (42)	2152 (3252)	∞	∞*1e4*	0/15
GA P100 C0.5 P0.05 T5	14 (19)	90 (43)	123 (50)	149 (55)	860 (955)	∞	∞*9600*	0/15
GA P100 C0.5 P0.05 T10	11 (18)	**68** (22)	**95** (25)	**114** (23)	**526** (884)	∞	∞*1e4*	0/15
GA P100 C0.5 P0.05 T20	10 (10)	**54** (24)	**74** (37)	**93** (47)	**658** (435)	∞	∞*1e4*	0/15

The above table shows the Expected Running Time for function 1. We show it here as a representative of what the best ones are for most of the functions. Overall Tournament Selection with size 20 is the one with the lowest ERTs for most functions with T10 and R4 being contenders in specific instances.

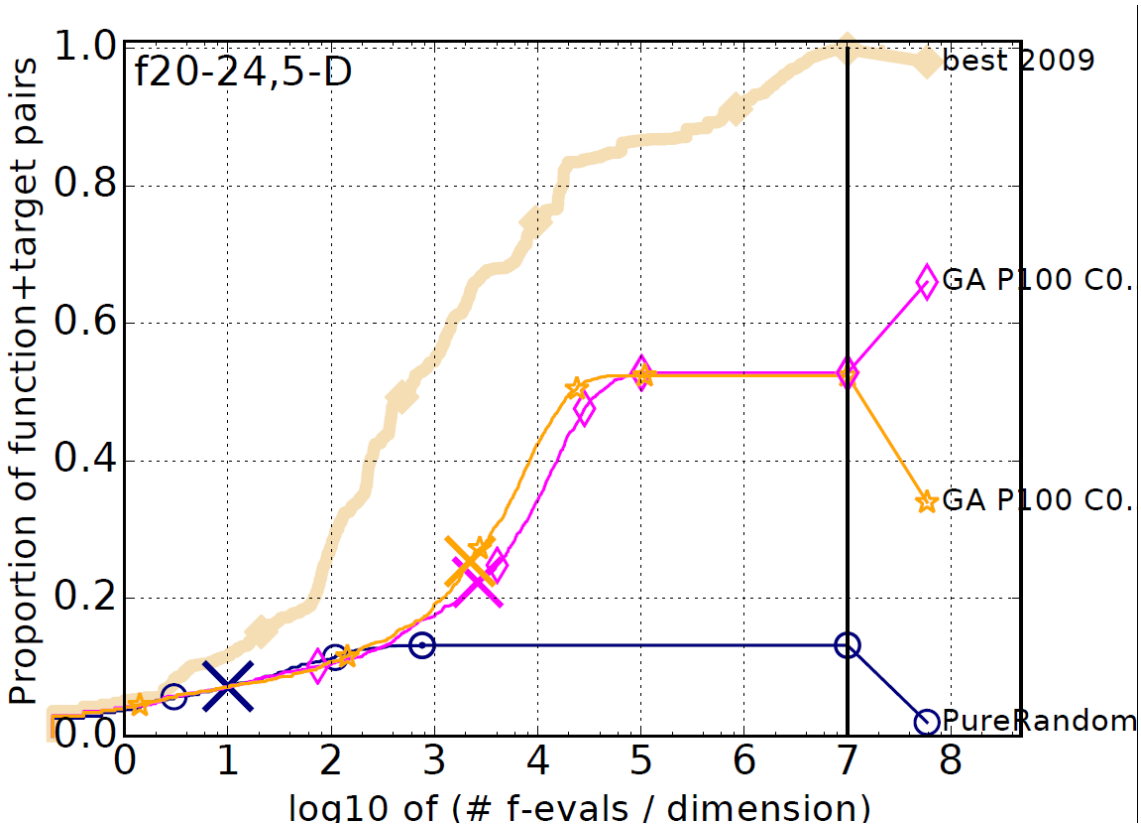
Introducing an Alien

For this experiment the variance was again reduced to 0.01. The remaining parameters where the same as in the previous experiment with T20 chosen for parent selection. Then we tested with and without Alien introduced each generation.

Results and Drawn Conclusions



Overall the Alien introduction seems to be slightly better with the best improvements being in one of the sets of multi modal functions. (The one with Alien is the yellow star)



Crossover Rate

In this experiment we kept the previous parameters and the Alien inclusion and tested different crossover rates 0.05, 0.1, 0.25, 0.5, 0.75

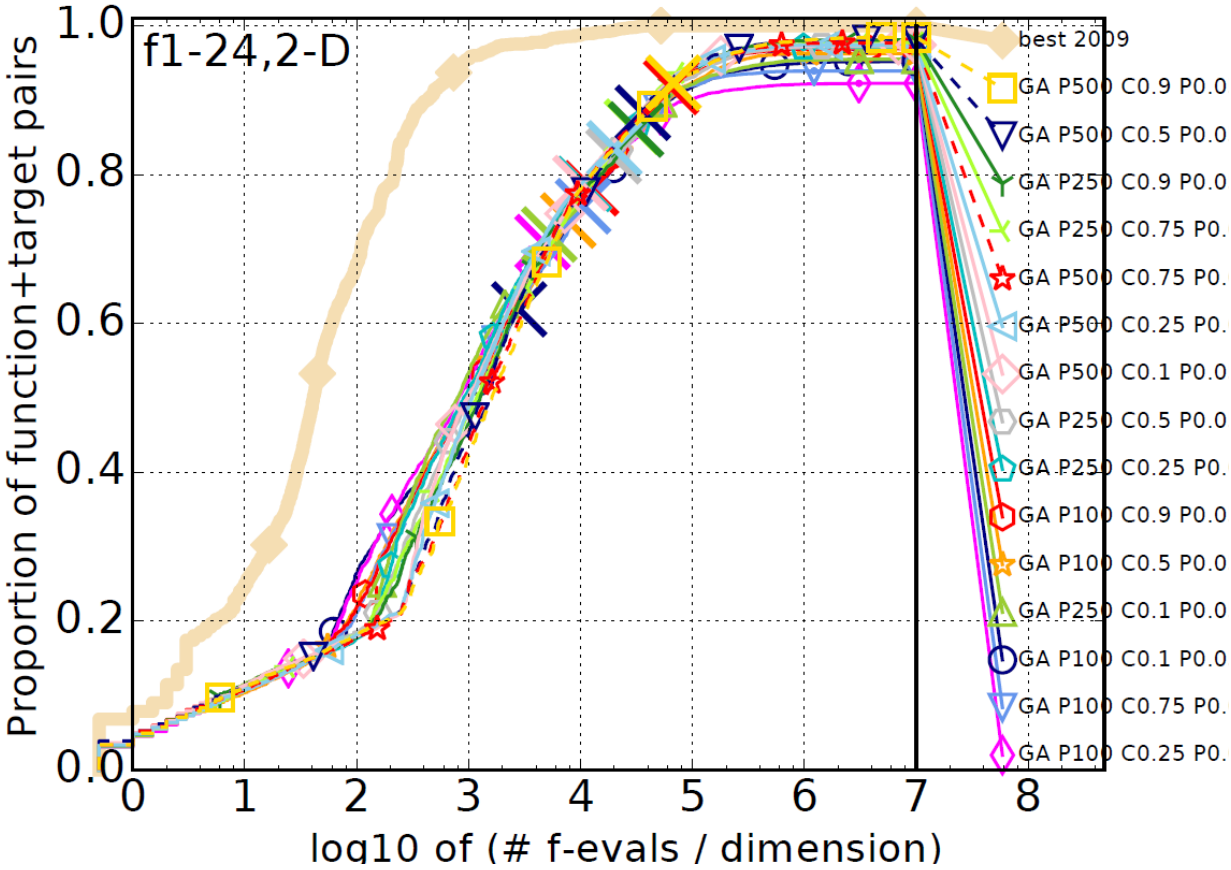
Results and Drawn Conclusions

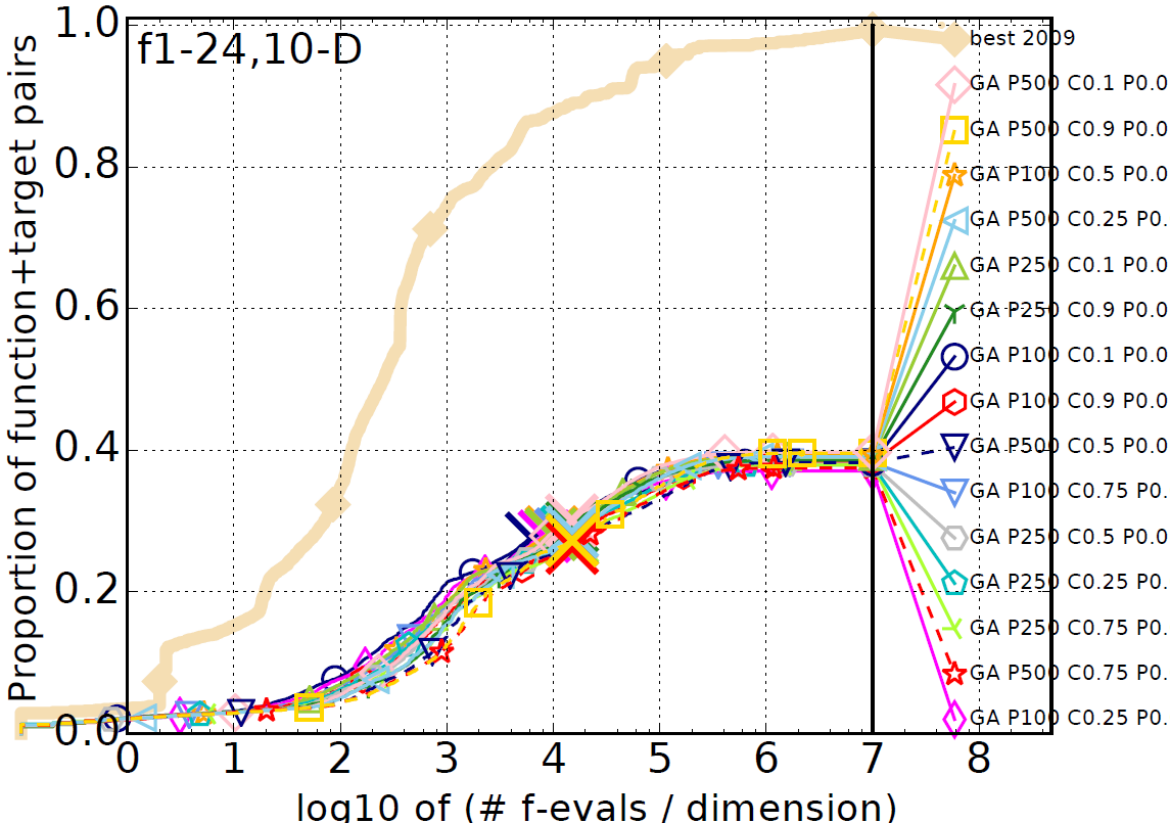
Unfortunately the results where highly variable from function to function. It would be cumbersome to report all the tables and graphs here. However there seemed to be a minor tendency towards 0.1 and 0.25 having good results.

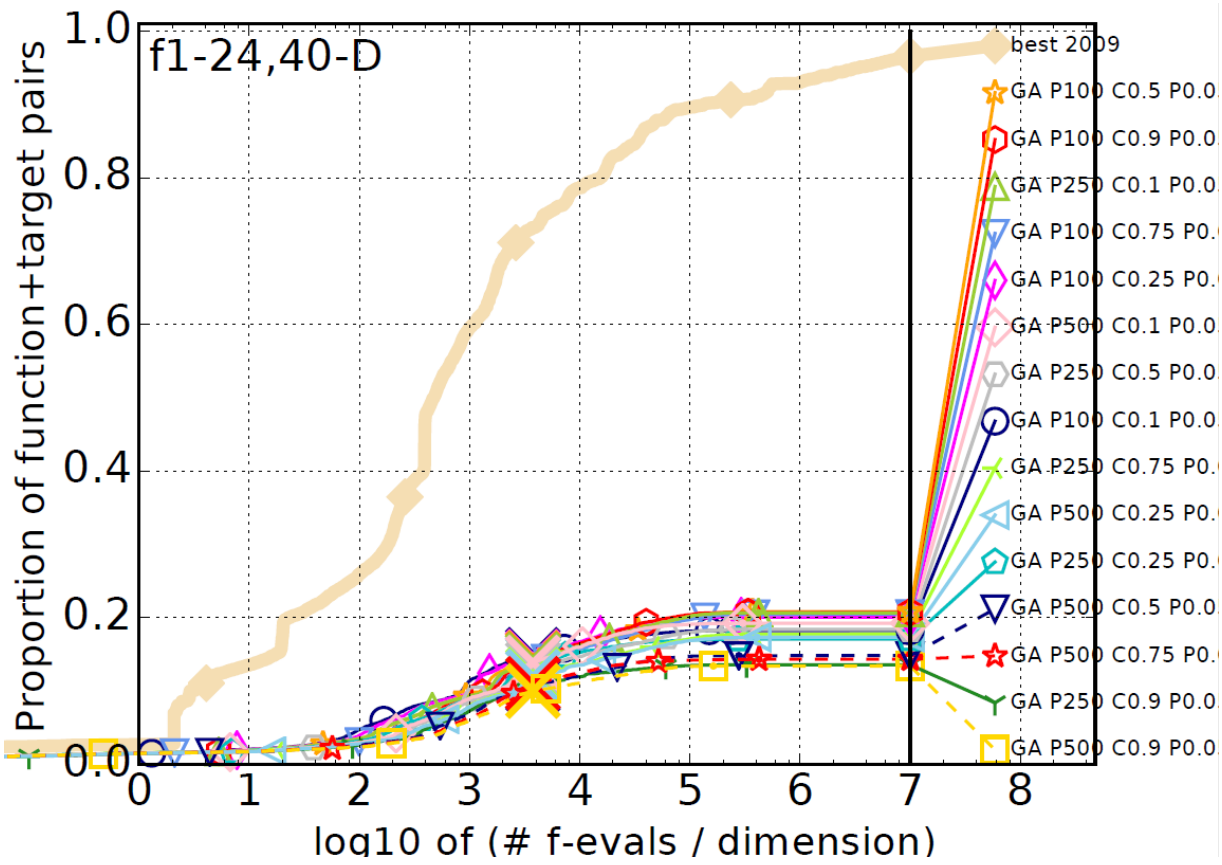
Population Size and Child Rate

In this experiment we used the random variance mutation method along with the parameters that we have been selecting. This time we used child rates of 0.1, 0.25, 0.5, 0.75, 0.9 and population sizes 100, 250, 500 over 2D, 10D and 40D to get a sense of the influence of the dimension on these parameters choice.

Results and Drawn Conclusions







The results are all very close as in previous experiments. For the 2D case the higher populations seem to have a slight advantage as they do in the 10D case. As for the 40D case the tendency seems to reverse. This is probably indicates that in a huge search space with comparatively very limited budget a small population gives an earlier start on the optimization part and has higher pressure using the same tournament size achieving slightly better results early on.

Neural Network Experiment

Given the time intensive nature of genetic algorithm optimization and especially given the implementation was made in python a full training and then using of neural networks to reproduce even one of the parameter sets of the previous experiments would be more than time would allow. Only a single experiment was performed where a neural network was “trained” with a GA algorithm for one single 2D function from each function class: f1 for separ, f6 for lcon, f10 for hcon, f15 for multi and f20 for multi2.

The GA was configured with parameters derived from the earlier experiments: population size 500, parent rate 5%, child rate 15%, tournament size 20. Given the poor results for 40D a much higher budget of 10000 generation maximum was used.

The fitness function was based on the quadratic error from 500 random samples of the functions.

The result files with the full matrices, and functions can be found in the github

repository under `Results\NeuralNets`. Here we will just report the layer sizes and the quadratic error for the training set and an equally sized testing set and draw conclusions from those.

F1 separ

```
test error value: 25.7784799131
training error value: 21.594915274320478
layer sizes:[2, 4, 4, 1]
```

This was the only function where the training network seems to more or less manage to follow the function

F6 Icon

```
test error value: 1017113032.73
training error value: 1083757371.655787
layer sizes:[2, 4, 4, 4, 1]
```

This like all the following results seems to fall far short of the desired function approximation.

F10 hcon

```
test error value: 1.31629462557e+14
training error value: 155148346777851.7
layer sizes:[2, 1]
```

F15 multi

```
test error value: 554826.814839
training error value: 407041.7447616186
layer sizes:[2, 4, 1]
```

F20 multi2

```
test error value: 150794077.614
training error value: 53969100.45436498
layer sizes:[2, 4, 4, 1]
```

Conclusions

The biggest conclusion that can be taken from the Genetic Algorithm experiments is that the mutate function has a huge impact on achievability and speed of convergence to the target. The remaining parameters hover important seem to pale in comparison in he differences that could be observed throughout the experiments.

As for the Neural Network experiment several things where very apparent from sizes of the layers and the appalling results overall. All the networks that had hidden layers converged to have the maximum size that was allowed for them. This can be

explained by the fact that removing a node causes a much bigger change that adding a node with small weights attached. Most of the hidden layer functions ended up with a linear function even though that could never achieve the final desired result simply because changing a sigmoid to a linear function quickly raises the final value range.

So, in conclusion, the designed set-up for the Neural Network part of the assignment was not conducive to good results and the extreme time frames for running these algorithms in a standard CPU made the iterations over the code very time consuming to test out. If designing a Genetic Algorithm for Neural Networks with changing structure it makes very little sense to use exclusively a the genetic algorithm to evolve completely weights when back propagation is a proven way to get good results much quicker. A hybrid GA where after each mutation the back propagations training is run to finalise the mutation would be much more likely to evolve reasonable networks in shorter time.

P.S.: A more complete report is available in the github repository. This is a trimed version to accommodate the word count.

Bibliography

Biologically Inspired Computing. (2016). *Evolutionary Algorithms: Details, Encoding, Operators* available: https://vision.hw.ac.uk/bbcswebdav/pid-2324478-dt-content-rid-2116861_2/courses/F21BC_2016-2017/02_BIC_ealec3%282%29.pdf

Hansen,N., Auger,A., Mersmann,O., Tušar,T., Brockhoff,D.. (2016). 'COCO: A Platform for Comparing Continuous Optimizers in a Black-Box Setting'. *ArXiv e-prints* arXiv:1603.08785

Mitchel,T. . (1997). *Machine Learning*. McGraw-Hill

Thrun,S., Burgard,W., Fox,D.. (2006). *Probabilistic Robotics*. The MIT Press