

School of Engineering and Computer Science
AIML426 – Evolutionary Computation and Learning

**Project 2: Evolutionary Algorithms for Optimization,
Computer Vision, and Machine Learning**

25% of Final Mark — Due: 11:59pm, 16 October (Sunday), 2022

Timothy McDermott - 300510457

Submission Guidelines	3
1.0 Evolutionary Programming & Differential Evolution Algorithms	4
1.1 Review	5
1.1.1 Overview	5
Differential Evolution	5
Evolutionary Program	6
1.1.2 Results	10
1.1.3 Comparison	10
1.1.4 Conclusion	13
2.0 Estimation of Distribution Algorithm	14
2.1 Estimation of Distribution Algorithm	15
2.2 Results	16
2.3 Discussion	16
3.0 Cooperative Co-evolution Genetic Programming	17
3.1 Cooperative Co-evolution Genetic Program	18
3.2 Results	19
3.3 Discussion	21
4.0 Genetic Programming for Image Classification	22
4.1 Automatic Feature Extraction through GP	22
4.2 Image Classification Using Features Extracted by GP	22
4.3.1 Feature Extraction	23
4.3.1.2 Feature Extraction Results	23
4.3.2 Image Classification	24
4.3.2.1 Image Classification Results	25
5.0 Evolution Strategies for Training Neural Networks [20 marks]	26
5.1 The Cart Pole Problem	27
5.2 Configuration & Architecture	27
5.3 Results and Discussion	28

Submission Guidelines

Your submission must contain the following files:

- report:

The report document must be in pdf file format. Writing the report will allow you to practise your technical writing skills. It should include the detailed description of the methods, algorithms and criteria used in your project, appropriate presentation, comparison and analysis of the results for each task. In general, readers should be able to understand what you have done purely from the report, without looking at your Programs.

- program.zip:

The program.zip file contains the executable programs you wrote for performing all the tasks. Add necessary comments for each program, and make sure to include ALL the required libraries and dependent files. For example, if you are using Java, then this should be the runnable .jar file. If you are using Python, then this should be the .py files.

- readme.txt:

A readme file to describe the detailed steps and commands on how to run your programs using the command line. If your programs cannot run properly, then you should provide a list of bugs in this file.

- source.zip:

The source code files. If you are using Python, then this contains the .py files (the same as that in the program.zip file). If you are using Java, then this contains ALL the .java files.

1.0 Evolutionary Programming & Differential Evolution Algorithms

In this question, you have the task to implement (1) the Evolutionary Programming (EP) algorithm and (2) either the Differential Evolution (DE) or Evolution Strategy (ES) algorithms. You also need to apply the implemented algorithms to searching for the minimum of the following two functions, where D is the number of variables, i.e., x_1, x_2, \dots, x_D .

Rosenbrock's function:

$$f_1(x) = \sum_{i=1}^{D-1} \left(100(x_i^2 - x_{i+1})^2 + (x_i - 1)^2 \right), x_i \in [-30, 30]$$

Griewanks's function:

$$f_2(x) = \sum_{i=1}^D \frac{x_i^2}{4000} - \prod_{i=1}^D \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1, x_i \in [-30, 30]$$

For $D = 20$, do the following:

- Implement any specific variation of the EP and DE/ES algorithms of your choice.
 - Justify your choice of the EP and DE/ES algorithm variations implemented.
- Choose appropriate algorithm parameters and population size, in line with your algorithm implementations.
- Determine the:
 - Fitness function,
 - Solution encoding, and
 - Stopping criteria in EP and DE/ES.
- Since EP and DE/ES are stochastic algorithms, for each function, $f_1(x)$ or $f_2(x)$, repeat the experiments 30 times.
 - Report the mean and standard deviation of your results, i.e., $f_1(x)$ or $f_2(x)$ values.
- Analyze your results, and draw your conclusions.

Then for $D = 50$, solve the Rosenbrock's function using the same algorithm settings

- Repeat 30 times.
 - Report the mean and standard deviation of your results.
 - Compare the performance of EP and DE/ES on the Rosenbrock's function when $D = 20$ and $D = 50$.
 - Analyze your results, and draw your conclusions
-

1.1 Review

1.1.1 Overview

Differential Evolution

Differential evolution (DE) is a heuristic optimization method that is used to find the global optimum of a function. It works by iteratively improving a set of candidate solutions, called a population, by applying genetic operators such as crossover and mutation. DE is one of the most popular algorithms for global optimization due to its simplicity and robustness. DE has been used in many different fields, including astronomy, chemistry, biology, and many more. The steps of the DE algorithm are:

1. Initialization: The population is initialised with random vectors.
2. Fitness: A fitness is assigned to each member of the population.
3. Mutation: A new vector is generated by mutating (changing) an existing vector.
4. Crossover: A new vector is generated by crossing (combining) existing vectors.
5. Replacement: The new vector replaces an existing vector if it is better.

(Also called survival)

6. Evaluation: The fitness (quality) of each vector is evaluated.

Steps 3 through 6 are repeated until stopping criteria is met.

My first approach to this problem was to utilise a library called *Yabox*, discussed by its developer [here](#). I've included some of the visualisations I was able to generate in a notebook within the part 1 code - however I was not able to fully take advantage of the library due to issues with its dependencies on ffmpeg and trouble running its algorithms.

The implementation of the final algorithm was based on [this approach](#) by [Bruno Scalia](#).

Variations of the algorithm change the methods of accomplishing each step - these varied steps are performed in the following manner. Initialization is done by creating a population of a given size between the bounds of the decision variables of the problem. Each of the individuals within the population corresponds to a vector of optimization variables. The main loop then begins, iterating over generations while stopping criteria is unmet and performing the mutation and crossover operations to create new vectors to test. The individuals I used were an extension of the generic base individual class I created, with extensions to support mutation and crossover. In this class individuals are represented as floats, allowing a continuous domain which fits the sample problem and DE excels at.

The mutation operator chosen is referred to as “*de/rand/l*” - referring to its location in the paper first describing it (Storn & Price, 1997). The pseudocode for this algorithm is available [here](#) - and the equation is given below.

$$\mathbf{v}_i = \mathbf{x}_{r1} + F(\mathbf{x}_{r2} - \mathbf{x}_{r3})$$

Where \mathbf{v} represents the mutant vector at the index i , and $r1, r2, r3$, are additional independent and exclusive indexes, while F is a control parameter that can be modified as needed to alter the mutation amount. F is commonly referred to as the scale parameter.

The crossover strategy used is binomial crossover, where an output vector, \mathbf{u} corresponds to a trial vector which is the product of combining the elements corresponding to “*mutant vectors v and target x.*” For this operation, a parameter CR (crossover rate) performs a similar role to F in the mutation operator, acting a control for the amount of crossover applied - or rather the frequency at which it is by acting as a switch between the two vectors, as shown below.

$$u_{i,j} = \begin{cases} v_{i,j}, & rand(0, 1)_{i,j} < CR \\ x_{i,j}, & rand(0, 1)_{i,j} \geq CR \end{cases}$$

Fitness of an individual is easy to calculate, being only the absolute error between the individual’s value and the true minimum - survival/selection/replacement simply checks whether the new or old value is better.

Selecting the parameters for these operators is relatively straight-forward though domain dependent. Depending on the type of search, exploration or exploitation may be the goal (breadth or depth). For problems which have discontinuous decision spaces which require greater exploration, using a larger value of F would be beneficial, while for exploitation a smaller value limits the expansion and exploration focusing the algorithm more finely. Similar properties apply to crossover. For both, I set the values low to encourage exploration and allow for faster finding of solutions. According to the graphs I created in the Yabox notebook trial the sample equations do not appear to have any local minima to get stuck in, their slopes are quite conspicuous and the optimal direction obvious so it is more important to fully exploit solutions. Likewise I set the parameter of F low and used its complement as the parameter for the crossover rate that ranged from a 70-30 to a 90-10 split.

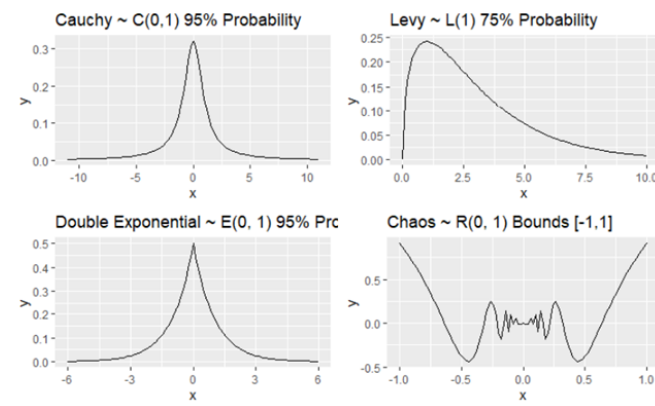
Evolutionary Program

Evolutionary programming (EP) is a similar optimization method that also uses a population of candidate solutions. However, EP uses different genetic operators than DE, such as

reproduction, crossover, and mutation. EP is also typically used to find local, rather than global, optimums.

The EP Algorithm is based on the approach outlined by [Brandon Morgan in Unit 5](#) of the Evolutionary Computation Course. Its implementation is a blend of the Fast-EP and Improved EP modules described in lectures, which take advantage of the Cauchy distribution for choosing random numbers. As described in lecture 13 - a combination of the Cauchy and normal distributions were used to take advantage of the weighting of a normal distribution and the tail of the Cauchy.

The Cauchy distribution is interesting because it has very long tails. This means that values can be far from the mean and still be sampled. This is useful for our purposes because it means that the algorithm can explore a greater range of values, and is less likely to get stuck in a local optimum. These tails and a comparison to other sampling methods are shown in the image from Unit 5 below.



The algorithm works as follows:

1. Initialize a population of candidate solutions
2. Evaluate the fitness of each candidate
3. Select a parent from the population using tournament selection
4. Generate a child solution by mutating the parent
5. Evaluate the fitness of the child
6. If the child is better than the parent, replace the parent with the child
7. Repeat from Step 3 until a termination condition is met

Further modifications were made to the base approach at a few of these stages. For instance, a new mutation technique from IEP was implemented; self-adaptive mutation. This approach scales the standard deviation of the mutation by a small random value. This allows the algorithm to keep track of its progress and gradually decrease the randomness of the mutations as it gets closer to the optimum. Additionally the tournament selection method was modified - previously it exhaustively and explicitly iterated over a population subset to run comparisons - great

performance increases were yielded by replacing this with the python built in sort function which runs in C. To do this sorting, individuals were encoded as floats to allow for a continuous representation of the individuals.

The fitness was evaluated using two methods, first by finding the absolute error between the true minima, and the output from passing the individual through the function, secondly by calculating the relative fitness in order to compare with other alternatives. The second method is integrated into the tournament selection method which tracks the success of an individual and its performance relative to the others. This approach is not technically required by the problem, however relevant game-theory problems such as the prisoner's dilemma which were covered in the lectures could benefit from a system such as this.

This combination of modifications allowed for wide scale exploration and domain coverage with high speeds and good variance.

The parameters can be modified for iterations, generations, and population. Iterations are the number of times the algorithm is run, while generations is the number of cycles for a given iteration of the algorithm. The population controls how many individuals exist for each generation. Increasing any of the variables will increase the coverage - however finding a good balance is important to weight the nature of the coverage and the run time - for instance increasing iterations without sufficient generations prohibits the algorithm from building or exploring deeper solutions - likewise low population does not allow sufficient variance from the genetic operators to take effect for widespread coverage. These parameters can greatly affect the breadth and depth of coverage you see i.e. exploration or exploitation. I tuned them visually after observing strong convergence - however my selection was made more towards execution time rather than accuracy as it is not the target of this assessment. Internally, the parameters for mutation were set as described in Unit 5. First, individual mutation is calculated using the formula below to create offspring by adding a value, delta x, to the parent x.

$$\hat{x}_{ij}(t) = x_{ij}(t) + \Delta x_{ij}(t)$$

where i denotes parent index, j represents variable index, \hat{x}_{ij} represents offspring, x_{ij} represents parent, and Δx_{ij} represents value to mutate by

The delta x is calculated by stepping by nu distance towards a randomly generated value taken from the probability distribution, capital eta with strategy parameters sigma.

$$\Delta x_{ij}(t) = \Phi(\sigma_{ij}(t))\eta_{ij}(t)$$

where Φ denotes a probability distribution, σ_{ij} represents the strategy parameter of that distribution, and η_{ij} denotes the step size in that direction

For self adapting strategy, the parameters were set following the same guide, using the lognormal operator. With this method offspring are created using the below function - in this case capital eta, is the probability distribution. In the example this is the gaussian normal distribution centred at zero with standard deviation of 1 - in the final submission this was changed as described above.

$$\hat{x}_{ij}(t) = x_{ij}(t) + \sigma_{ij}(t)N_{ij}(0, 1)$$

Finally, the step size with a step size of sigma was set as outlined in Lecture 13 under factor calculation - in the lecture it is set as tau while Unit 5 uses mu.

- Factor calculation:

$$\tau = (\sqrt{2\sqrt{n}})^{-1}, \tau' = (\sqrt{2n})^{-1} \quad \text{where } \mu_1 = \frac{1}{\sqrt{2\sqrt{n_x}}}, \mu_2 = \frac{1}{\sqrt{2n_x}}, \text{ and } n_x \text{ denotes number of variables}$$

With each iteration the strategy parameter sigma is updated with the following equation (mu and tau substitutable).

$$\sigma_{ij}(t+1) = \sigma_{ij}(t)e^{(\mu_1 N(0,1) + \mu_2 N(0,1))}$$

1.1.2 Results

<i>D 20</i>		
Rosenbrock	Evolutionary Programming	Differential Evolution
mean	1.08E+07	263487.436
min	1.97E+06	61240.892
max	4.04E+07	522977.211
std	8.18E+06	123604.695
⋮		
Griewank	Evolutionary Programming	Differential Evolution
mean	1.227	0.994
min	1.135	0.931
max	1.398	1.023
std	0.071	0.028

<i>D 50</i>		
Rosenbrock	Evolutionary Programming	Differential Evolution
mean	3.84E+08	1.55E+07
min	2.94E+08	9.78E+06
max	5.40E+08	2.37E+07
std	5.88E+07	4.28E+06
⋮		
Griewank	Evolutionary Programming	Differential Evolution
mean	3.306	1.258
min	2.886	1.207
max	3.702	1.339
std	0.215	0.036

1.1.3 Comparison

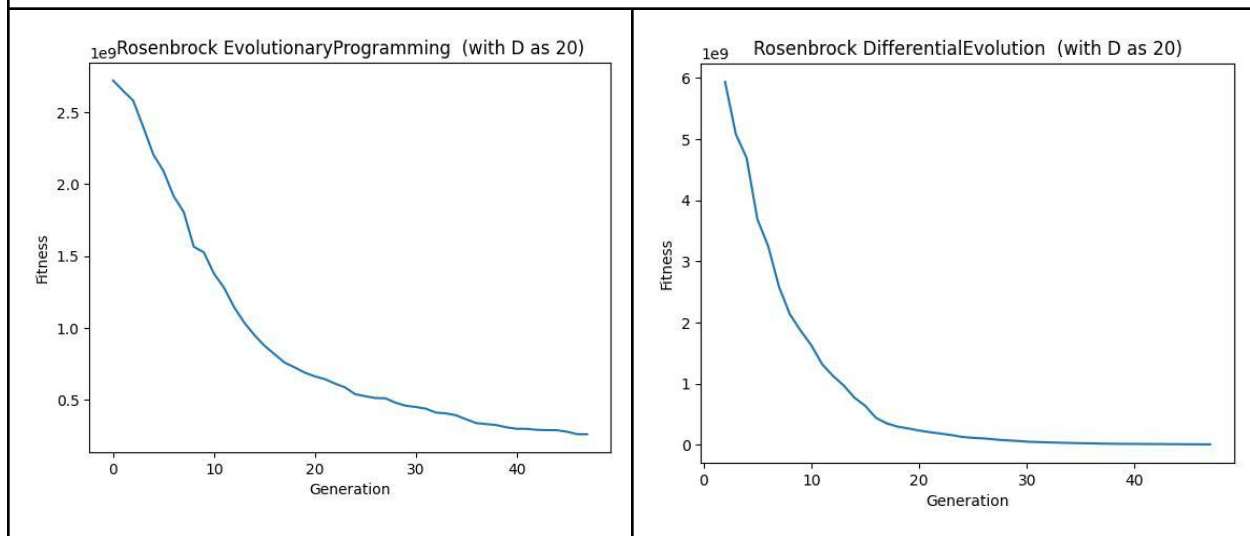
As differential evolution is natively continuous, it is very well suited to the given problem. I expected it to perform better than evolutionary programming based on evolutionary programming existing on a higher level than differential evolution - though it may be more powerful it requires significant adaptation and configuration to be ideally suited to this problem. This seems to be reflected in the results quite clearly as differential evolution out performed

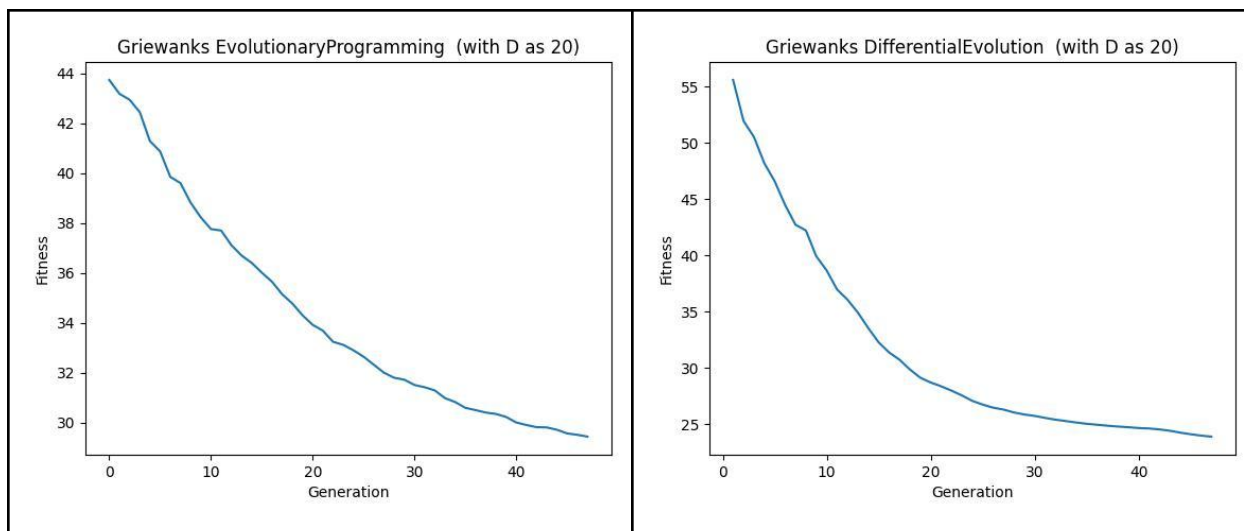
evolutionary programming across both problems in nearly every metric. It is important to note that EP performed better on the lower dimension problem, and the parameter selection can have a much greater effect on the performance of EP - especially because I selected shorter run-times over accuracy.

The convergence graphs below provide useful insight into the performance of the algorithm. As stated above, the Rosenbrock function is particularly obvious with no subtlety to the global optimum, the convergence here is smooth for D20 as both algorithms take steps improving the fitness almost continuously. Comparatively, the Griewank function is quite rough as there are many local optima for the algorithm to stick to - with the smaller value for D both convergence curves were smooth, however even on the Rosenbrock function with higher dimensionality the evolutionary programming approach had varied convergence. This is not necessarily bad, as it means it may be escaping local optima, but it could also be exploring too widely and not fully exploiting existing solutions. This could be tuned by including more elites i.e. the best results from previous generations, to steer the algorithm more heavily towards already found optima.

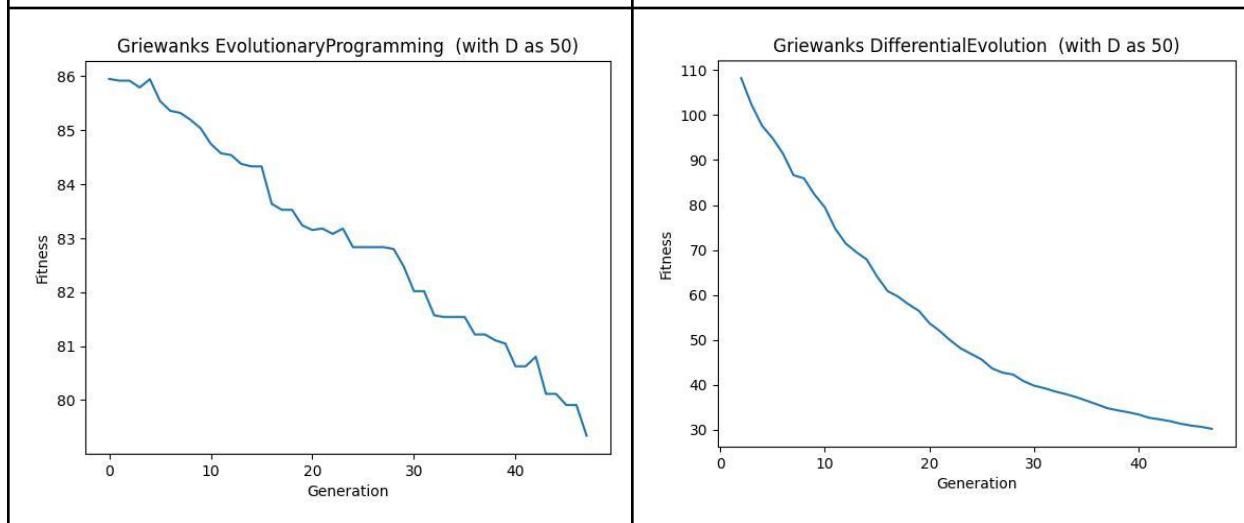
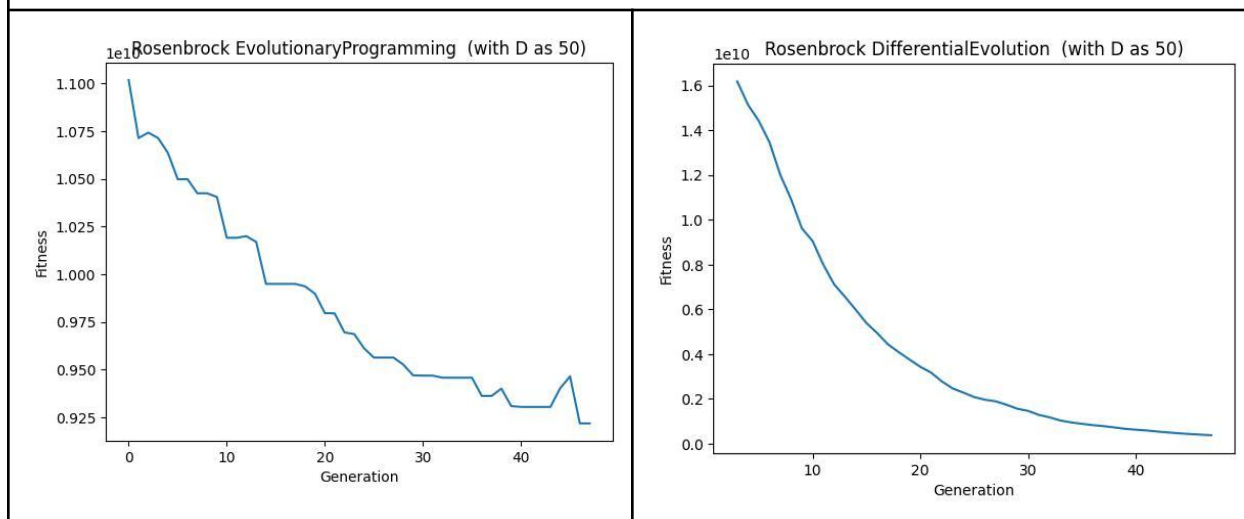
Convergence Graphs

D 20





D 50



1.1.4 Conclusion

There are some trade-offs to consider when using DE. It is typically faster than EP and can find better solutions, but is also more likely to get stuck in a local optimum. EP is slower but can sometimes find more novel solutions. The benefits of using DE include its simplicity and robustness. Additionally, DE can be used to optimize functions without derivatives, which is often the case in real world problems. However, one drawback of DE is that it can get trapped in local minima. Next time, I might consider implementing dither and/or jitter with DE. Dither and Jitter are similar parameters to the self adapting strategy for EP but for DE. Dither and jitter can tune the coverage of the algorithm as needed by modifying the value of F and adding randomness to the process, respectively. Jitter also has the special attribute of adding rotation to the vector by multiplying its components by a random value.

Ultimately the decision is up to the developer based on the requirements of their problem, with both being reasonable approaches with valid use-cases.

2.0 Estimation of Distribution Algorithm

The 0-1 knapsack problem has been studied in project 1. Given a set of M items and a bag, each item i has a weight w_i and a value v_i , and the bag has a capacity Q . The knapsack problem is to select a subset of items to be placed into the bag so that the total weight of the selected items does not exceed the capacity of the bag, and the total value of the selected items is maximised. The formal problem description can be written as follows.

$$\max v_1x_1 + v_2x_2 + \cdots + v_Mx_M, \quad (1)$$

$$s.t. w_1x_1 + w_2x_2 + \cdots + w_Mx_M \leq Q, \quad (2)$$

$$x_i \in \{0, 1\}, i = 1, \dots, M, \quad (3)$$

Where $x_i = 1$ means that item i is selected, and $x_i = 0$ means that item i is not selected. In this question, you are given the following three knapsack problem instances:

- 10 269: with 10 items and bag capacity of 269. The optimal value is 295.
- 23 10000: with 23 items and bag capacity of 10000. The optimal value is 9767.
- 100 1000: with 100 items and bag capacity of 995. The optimal value is 1514.

The format of each file is as follows:

The first number is the number of items, and the second number is the bag capacity.

From the second line, each line has two numbers, where the former is the value, and the latter is the weight of the item.

M	Q
v_1	w_1
v_2	w_2
\dots	\dots
v_M	w_M

Develop a simple Estimation of Distribution Algorithm (EDA), for example based on either the univariate marginal distribution algorithm (UMDA) or the population based incremental learning (PBIL) algorithm, to solve the above knapsack problem instances.

You should

- Determine the proper individual representation and explain the reasons.
- Design the proper fitness function and justify its use.
- Design and implement your EDA algorithm.
- Set proper algorithm parameters, such as population size, individual selection criteria, and learning rate (subject to your choice of EDA variations).
- For each knapsack problem instance, run your EDA implementation for 5 times with different random seeds. Present the mean and standard deviation of EDA in the 5 runs.
- Draw the convergence curve of your EDA implementation for each knapsack problem instance. The x-axis of your convergence curve represents the number of generations.
- The y-axis stands for the average fitness of the best solutions in the population of the x-th generation from the 5 runs. Discuss your convergence curve and draw your conclusions.

In the report, you should describe the details of your EDA algorithm design and implementation (including the overall algorithm outline, solution representation, statistics calculation, new solution generation, and algorithm parameters). You should also show the results (mean and standard deviation, and convergence curves), and make discussions and conclusions in your report.

2.1 Estimation of Distribution Algorithm

Details

Design

Implementation

Overall Outline

Solution Representation

Statistics Calculation

New Solution Generation

Algorithm Parameters

The Estimation of Distribution Algorithm (EDA) works by tracking the statistics of a population of candidate solutions, and generating a new population at each generation from the previous population's statistics. EDA's are typically used to solve discrete optimization problems.

Individuals are represented as a list of booleans to toggle the presence of an item in the knapsack.

The steps of the algorithm are as follows:

1. Generate a population from the probability vector,
2. Evaluate the fitness of each member and rank them,
3. Update and Mutate the population based on the fittest individual,

Repeat steps 1 through 3 until stopping criteria is met.

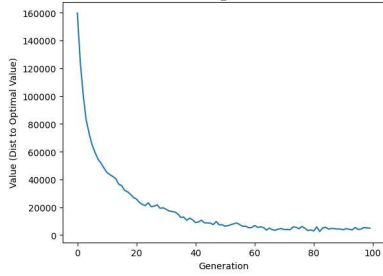
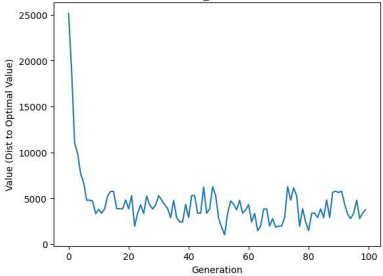
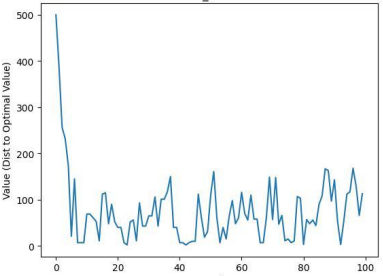
The steps for this algorithm are quite straightforward and main deviations come from the parameter selection. When initialising the population in step one, the population size is important and affects the “momentum” or inertia within the algorithm, similarly to particle swarm optimization - that is to say a smaller population will vary wildly while greater populations have a larger mass and are more difficult to divert from their course - this can have a few effects on the outcome - either by slowing learning and convergence as the large population is slow to adapt (while exhaustive), or by under evaluating areas by merit of a limited population. Here learning rate, mutation probability, and mutation shift act similarly to step size, the mutation rate discussed previously and the “F” parameter described in part 1. The fitness function in step two is a modification of that from the first assignment, this time in an effort to allow unfit solutions to mutate into suitable ones I enabled the use of solutions outside of the fitness criteria - this had some drawbacks which are described in the results.

The update method in the class applies all the modifications, manipulating the indexes in a vector with learning and mutation. Using the learning rate, a multiplication coefficient is created by first subtracting the learning rate from 1 then setting the vector to the product of the coefficient and

itself; the best solution is then multiplied by the learning rate and added to the vector. Mutation is then applied in the same way, with it only occurring a set percentage of the time according to the mutation probability - instead of learning rate, the mutation shift is used, and instead of the best candidate, a random value of either 0 or 1 is used.

Run statistics are kept using a logbook along with values for each generations fitness to allow for creation of the convergence curves.

2.2 Results

<i>Convergence Graphs</i>		
		
Optimal: 1514	Optimal: 9767	Optimal: 295
Fitnesses: [2141.0, 1234.0, 1018.0, 1741.0, 751.0] Mean: 1377.0 Std: 501.477417238304	Fitnesses: [9760.0, 9755.0, 9754.0, 8781.0, 9756.0] Mean: 9561.2 Std: 390.1053191126724	Fitnesses: [288.0, 295.0, 295.0, 295.0, 295.0] Mean: 293.6 Std: 2.8000000000000003

2.3 Discussion

From the resultant convergence graphs it appears at first only one instance was able to converge, though this is likely just due to the scale. Examining the standard deviation shows similar performance across the instances. As mentioned in 2.1, the changes to the fitness function meant to increase the search space and allow for solutions to evolve from unfit solutions ended up polluting the final results with unfit solutions, while now fixed I did not notice this at first as the result seemed close to the expected values, after adding a guard against this the results performed better. To decrease the variance it may have been beneficial to properly utilise elitism to weight the population towards existing optimums, however this was against my initial changes to the fitness function so I did not implement it. The performance overall seems quite reasonable, though the larger instances were less optimal, with 23_10000 getting quite close still, and 100_995 being the furthest.

3.0 Cooperative Co-evolution Genetic Programming

In project 1 we have tried to use GP to evolve a single genetic program to solve the following symbolic regression problem:

$$f(x) = \begin{cases} \frac{1}{x} + \sin x & , \quad x > 0 \\ 2x + x^2 + 3.0 & , \quad x \leq 0 \end{cases} \quad (4)$$

In project 1, we assume that there is no prior knowledge about the target model. In this project, the assumption is changed. Instead of knowing nothing, we know that the target model is a piecewise function, with two sub-functions $f_1(x)$ for $x > 0$ and $f_2(x)$ for $x \leq 0$. In other words, we know that the target function is:

$$f(x) = \begin{cases} f_1(x), & x > 0 \\ f_2(x), & x \leq 0 \end{cases} \quad (5)$$

This question is to develop a Cooperative Co-evolution GP (CCGP) to solve this symbolic regression problem. The CCGP should contain two sub-populations, one for $f_1(x)$ and the other for $f_2(x)$.

You can use a GP library.

You should:

- Determine and describe the terminal set and the function set of each sub-population.
 - Design the fitness function and the fitness evaluation method for each sub-population.
 - Set the necessary parameters, such as sub-population size, maximum tree depth, termination criteria, crossover and mutation rates.
 - Run the implemented CCGP for 5 times with different random seeds. Report the best genetic programs (their structure and performance) of each of the 5 runs. Present your observations and discussions and draw your conclusions.
-

3.1 Cooperative Co-evolution Genetic Program

As a piecewise function is our target function, we can use cooperative coevolution to split the problem into two subproblems with distinct requirements but the same overall optimization goal. This allows each of the programs to work independently of one another but cooperatively aim towards the same goal. In assignment one we completed this task without domain knowledge, optimising the problem as a whole - it is the goal of cooperative co-evolution to use our added domain knowledge to improve the performance of the GP system.

The design of the system was based on the process I used with the previous with variations included from mistakes I made and learnt from the first time. The changes include switching to DEAP from gplearn, as gplearn is a less comprehensive extension of DEAP and this solution used modifications to the base DEAP eaMuPlusLambda algorithm. Some other benefits from this change included not needing to explicitly provide a dictionary to map the terminals to the functions as they just utilise the argument name instead.

The function set I defined was composed of relevant functions to the domain of the problem. As an equation, I gave it the following functions: addition, subtraction, multiplication, division, sine, negative, and inverse. These functions are the operations the genetic program is allowed to use and can be thought of as the phenotype of each node - they also map to the direct function or genotype/implementation of each method.

The number of epochs was set at 50 , meaning the algorithm will run for 50 cycles. This count was chosen to give the algorithm enough time to converge on a solution, in assignment 1 I used 40 generations instead and for a direct comparison I would use the same parameters in the same framework, but as I have switched to deap I have also reconfigured the parameters.

As we have split the function into its components, the fitness function is different depending on the subtree you are within, for either case the fitness is calculated as the residual sum of squares or sum of squared residuals i.e. error. In code this is done using an evaluate function with an internal method that changes the target function depending on which component of the piecewise function is being evaluated.

$$RSS = \sum_{i=1}^n (y_i - f(x_i))^2$$

RSS = residual sum of squares

y_i = i 'th value of the variable to be predicted

$f(x_i)$ = predicted value of y_i

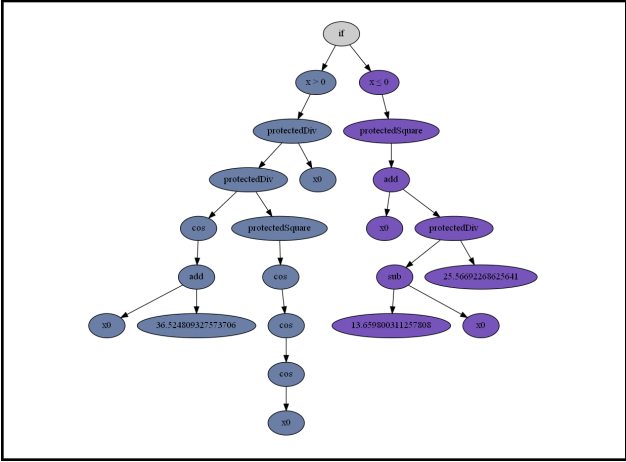
n = upper limit of summation

```
def evaluate(individual, points):
    actual = toolbox.compile(expr=individual)

    def target(x):
        """
        The target function we are trying to approximate
        """
        if x > 0:
            return 1 / x + math.sin(x)
        return 2 * x + x ** 2 + 3.0

    sse = math.fsum([(actual(x) - target(x)) ** 2 for x in points])
    # Return as a tuple for DEAP
    return sse / len(points),
```

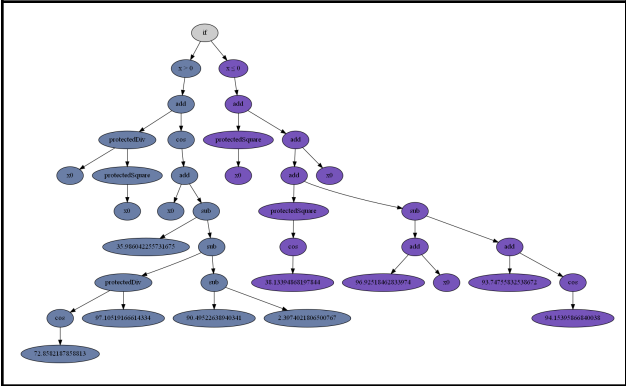

Seed 512



Population 1 best fitness: (0.2957size 12.
--

Population 2 best fitness: (0.9071size 8.

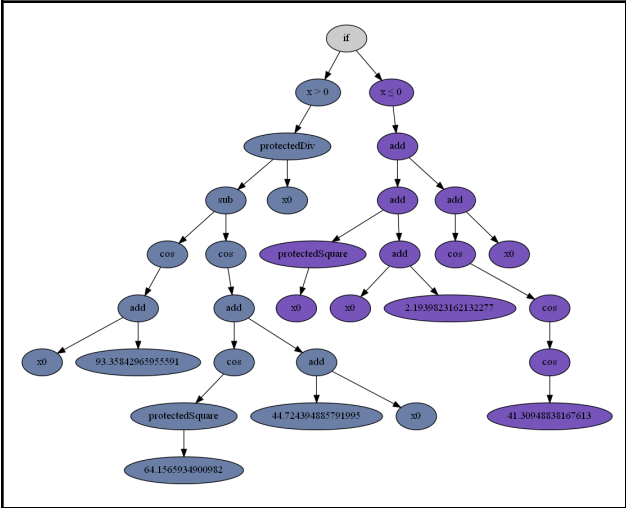
Seed 1337



Population 1 best fitness: 1.4678 size 18.

Population 2 best fitness: 2.0813 size 17.

Seed 2048



Population 1 best fitness: 0.3167 with size 15.

Population 2 best fitness: 1.1054 with size 13.

3.3 Discussion

Graphically, the equation can be broken down into three parts, the downward sloping curve when $X < 0$, the sharp spike at 0 from the switch, and the sine wave when $x > 0$. With seed 6, the algorithm appeared to capture the deviation from the spikes at 0, however the sine wave appeared to be misinterpreted as an upward curve. For seed 42 the initial slope was less accurate but the sine wave was better followed, though the periodicity was off pace. Seed 512 also had a less accurate initial slope but generated the best result for tracking the sine wave. Of the runs, seed 42 seemed to have the best result - this is backed up by the shorter equation which can be clearly seen in the graph representation of it. Seed 512 seemed to over-complicate the equation to try and fit the sine wave to the detriment of the other half of the equation, and seed 6 missed the wave portion of the equation. Compared to the decision tree and random forest models of assignment 1, the GP system was not able to make significant improvements to the predictions. Further tuning of the parameters may improve the results here, but exploring which permutations yield the best result is complex and computationally expensive. GP is better relegated to np-hard problems as a hyper-heuristic system where the extra computational cost is warranted - for this problem, existing solutions outperform it.

It is interesting to see how the fitness of the overall program for each may be similar while the performance of each population within one of the solutions can be quite varied, with one greatly bringing down the overall error rate. The good performance of seed 42 carries over from the first assignment, and it was able to find good solutions for both populations while the others typically only found one. It may be interesting to implement a method similar to hoist mutation here where trees can be dropped between solutions, similar to a hall of fame but for each full subtree between seeds. Comparison of the program structures is also interesting as the simplest tree is also the most accurate. While each population can be evaluated as an individual tree, I noticed that the best program, 42, started each tree with opposite actions, adding for population one and subtracting for population two. This extended to the second layer which also had inverse operations with population one having division or cosine, and population two having square or cosine. This was also somewhat present in tree 512, which started with square and div, for population 1 and 2 respectively.

I feel the large difference in performance between the runs came from being able to evaluate the trees at around the same progression level. While it would require further testing, it seems that anecdotally the runs which progressed one tree faster left the other behind, as seen in tree 6. This could be explainable as running the evaluation of a solution in a balanced way, vs the evaluation becoming dominated by the performance of its counterpart improving the score, causing uneven development.

4.0 Genetic Programming for Image Classification

Image classification is an important and fundamental task of assigning images to one of the pre-defined groups. Image classification has a wide range of applications in many domains, including bioinformatics, facial recognition, remote sensing, and healthcare.

To achieve highly accurate image classification, a range of global and local features must be extracted from any image to be classified. The extracted features are further utilised to build a classifier that is expected to assign the correct class labels to every image. In this question, you are provided with two image datasets, namely FEI 1 and FEI 2 1. The two datasets contain many benchmark images for facial expression classification. Example images from the two datasets are given in Figure 1. All images contain human faces with varied facial expressions. They are organised into two separate sets of images, one for training and one for testing.

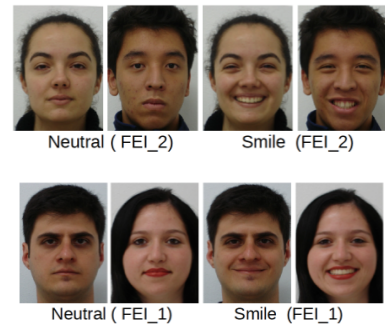


Figure 1.

Your task is to build an image classifier for each dataset that can accurately classify any image into two different classes, i.e., “Smile” and “Neutral ”. This is to be done in two parts as below.

4.1 Automatic Feature Extraction through GP

In this subsection, we use the GP algorithm (i.e., FLGP) introduced in the lectures to design image feature extractors automatically. You will use the provided strongly-typed GP code in Python to automatically learn suitable images features respectively for the FEI 1 and FEI 2 datasets, identify the best feature extractors evolved by GP for both datasets and interpret why the evolved feature extractors can extract useful features for facial expression classification. Based on the evolved feature extractors, create two pattern files: one containstraining examples and one contains test (unseen) examples, for both the FEI 1 and FEI 2 datasets.

Every image example is associated with one instance vector in the pattern files. Each instance vector has two parts: the input part which contains the value of the extracted features for the image; and the output part which is the class label (“Smile” or “Neutral ”). The class label can be simply a number (e.g., 0 or 1) in the pattern files. Choose an appropriate format (ARFF, Data/Name, CSV, etc) for your pattern files. Include a compressed version of your generated data sets in your submission.

4.2 Image Classification Using Features Extracted by GP

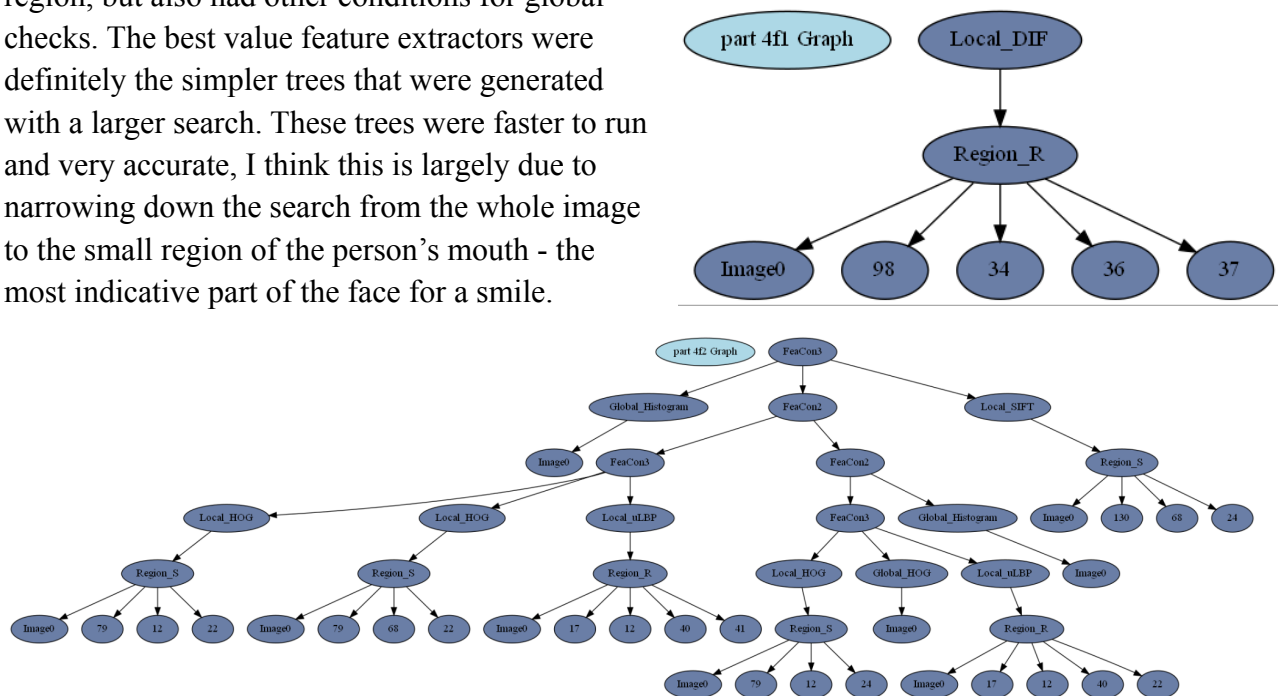
Train an image classifier of your choice (e.g., Linear SVM or Naive Bayes classifier) using the training data and test its performance on the unseen test data that are obtained from the previous step (Subsection 4.1). Choose appropriate evaluation criteria (such as classification accuracy) to measure the performance of the trained classifier on both training and test data. Present and discuss the evaluation results. Study the best GP tre obtained by you from the previous step (Subsection 4.1), with respect to both the FEI 1 and FEI 2 datasets. Identify and briefly describe all the global and local image features that can be extract by the GP trees from the images to be classified. Explain why the extracted global and local image features can enable the image classifier to achieve good classification accuracy.

4.3.1 Feature Extraction

The implementation of this algorithm is largely handled for us by the IDGP code. I made some modifications to the IDGP_main.py file to configure the toolbox, primitive set, and test various parameters. The final approach iterates over the datasets, runs the GPMain method to initialise and runs a basic eaSimple algorithm. Behind the scenes, feature selection and the supporting functionality is all provided for us by the template code. Once the GPMain method has finished we can test the created classifier, first transform the test data, then pass it through the model. Finally we can export the pattern sample, then print and render our results and graph.

4.3.1.2 Feature Extraction Results

The first iteration of the program generated quite complicated trees like the above. This tree contained a very similar subtree to the final tree within it. The first iteration examined a specific region, but also had other conditions for global checks. The best value feature extractors were definitely the simpler trees that were generated with a larger search. These trees were faster to run and very accurate, I think this is largely due to narrowing down the search from the whole image to the small region of the person's mouth - the most indicative part of the face for a smile.



While both trees were generated with the same config, the F2 dataset further exploited the solution space, leading to a deeper tree with much more complex logic, and far greater training time. This increase did result in greater accuracy, and the best approach would depend on the use case. Global features were not used in F1, while they were for F2. Global features can be thought of as statistics for the whole image, rather than looking at a specific part of the image, while the first program just looked for a smile, the second measured attributes from the image as a whole, which allows for greater data throughput in the model, capturing other parts of the image which may have been missed by just looking for a smile. For example, program one might be looking for a smile by actually looking for teeth via local search, while approach two also accounts for a

closed mouth smile or the eyebrows through the combination of global features, and additional local features. Some global features which could be considered are statistics like color, trajectory, or edge types, with frequency being assessed for each of them (easily interpretable for eyes, eyebrows, and mouth). I found that the deeper a run got, the more comprehensive it became and likely it was for global parameters - though it may have become more overfitted.

The extraction of these features allows for the program to have good accuracy as it effectively performs feature selection on its own extracted features to weigh and discard data with low importance to the outcome. The result is a dataset with reduced dimensionality allowing for faster processing, and extraction of complex attributes down to simplified fields, enabling easier modelling of higher-dimension qualities. By no means do I expect either program to be optimal. The algorithm varies between runs and a proper survey should be done with random seeding to assess the model performance and derive optimal parameters and gp-programs for feature selection.

Two pattern files have been created based on these feature extractors and are included in the part 4 directory.

4.3.2 Image Classification

The chosen classifier was Linear Support Vector Machine or LSVM. Support vector machines work by mapping data to a high-dimensional feature space so that data points can be categorized, even when the data are not otherwise linearly separable. A support vector machine classifies data by finding the hyperplane that maximizes the margin between the two classes - LSVM's use only linear support vectors because they scale better, and provide greater freedom for the loss functions, while traditional SVMs allow for easy testing of a number of kernels for a grid search. The feature space is transformed into a high dimensional space in order to fit a hyperplane between the two classes. The extracted features are then used to train the classifier. The classifier is then tested on the unseen data to see how accurately it can classify the images. LSVM was chosen as the classifier because it is known to be effective in high dimensional spaces. It is also effective in cases where there is a large amount of data, as is the case for computer vision problems.

The evaluation metric is controlled by the eval method, if it is called with the cross validation flag then the score is calculated using the cross_val_score method which returns an array of scores of the estimator for each run of the cross validation. When called without this flag, i.e. for testing, the passed data is scaled and fitted to the model, predictions are made, and metrics for the F1 score, and the base LSVM score are calculated. F1 score was chosen to evaluate the performance of the classifier because it is a more comprehensive metric than accuracy and takes both false positives and false negatives into account.

4.3.2.1 Image Classification Results

<i>Dataset: F1 Results</i>	<i>Dataset: F2 Results</i>
Best individual <i>Local_DIF(Region_R(Image0, 98, 34, 36, 37))</i>	Best individual <i>FeaCon3(Global_Histogram(Image0), FeaCon2(FeaCon3(Local_HOG(Region_S (Image0, 79, 12, 22)), Local_HOG(Region_S(Image0, 79, 68, 22)), Local_uLBP(Region_R(Image0, 17, 12, 40, 41))), FeaCon2(FeaCon3(Local_HOG(Region_S (Image0, 79, 12, 24)), Global_HOG(Image0), Local_uLBP(Region_R(Image0, 17, 12, 40, 22))), Global_Histogram(Image0))), Local_SIFT(Region_S(Image0, 130, 68, 24))))</i>
Test results <i>F1 Score: 0.9130434782608696,</i> <i>Base Score: 0.92</i> Training took 438.27 seconds. Testing took 0.14 seconds. Population 1 best fitness: (0.967053,) size 7.	Test results <i>F1 Score 0.92,</i> <i>Base Score: 0.92</i> Training took 9281.08 seconds. Testing took 19.53 seconds. Population 1 best fitness: (0.966488,) size 49.

The classification results from both feature selection methods show just how overfitted the second program became. It had a very similar F1 and Baseline score, with its best fitness actually being lower than the other dataset. I am working on the assumption the datasets are taken from the same sample and all variance is attributable to our process and the small differences between the sets. Because of this I would say that the accuracy was not improved by including global variables, though I expected it could be. The resultant tree was 7 times larger, and took 21 times longer to train, and 140 times longer to test. At some point in the generation process, I did observe a very similar tree for F2 to F1 with a size of around 7, and similar fitness, with a better termination criteria, or a stricter limit on node depth it would be possible to effectively capture such solutions that are simpler, faster and as accurate as the complex selection program.

5.0 Evolution Strategies for Training Neural Networks [20 marks]

Many practically valuable problems can be modeled as reinforcement learning problems. Control problems stand for an important and widely studied family of reinforcement learning problems, including, for example, the cart pole problem, the mountain car problem, the bipedal walker problem and the lunar lander problem. All these problems can be solved by using policy neural networks trained with zeroth-order Evolution Strategies (ES) algorithms.

The ZOO-RL Python library supplemented with this project provides high-quality implementations of several zeroth-order ES algorithms, including the famous OpenAI-ES algorithm. This task requires you to study the performance of OpenAI-ES using the ZOO-RL Python library and report your findings.

- Study the benchmark cart pole problem named “CartPole-v1 ”
 - Provide a short and clear description of this problem in your report. Particularly:
 - Define the state representation,
 - the discrete action space,
 - the reward function,
 - the maximum length of any episode,
 - and the criteria for successful learning.

A short description of the CartPole-v1 problem can be found in (<https://gym.openai.com/envs/CartPole-v1/>).

More information about the cart pole problem can be found in the reference below:

G. Barto, R. S. Sutton and C. W. Anderson, “Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problem”, IEEE Transactions on Systems, Man, and Cybernetics, 1983.

- Experimentally evaluate the performance of OpenAI-ES at solving the CartPole-v1 problem.
 - OpenAI-ES has already been implemented in ZOO-RL. No re-implementation of this algorithm is required for this task.
 - All you need is to configure your YAML file requested by ZOO-RL to use this algorithm.
 - Refer to README of the ZOO-RL Python library for more technical details.
 - Report the architecture of the policy neural network used in your experiments.
 - Provide the detailed algorithm parameter settings of OpenAI-ES.
 - Draw the learning performance curves of OpenAI-ES with the horizontal axis representing the generation number and the vertical axis representing the performance of the trained policy neural network.
 - You must perform 5 independent algorithm runs (each run must use a different random seed) on the CartPole-v1 problem and report the average learning performance in the respective performance curves.
-

5.1 The Cart Pole Problem

The cart pole problem is a simulation problem that takes advantage of Open-AI's gym simulation environment to model a pole balancing on top of a cart which moves without friction across a single dimensional plane.

- The movement of the pole affects the cart and the movement of the cart affects the pole.
- The cart cannot move too far in either direction as there is a “cliff”
- The state representation is a 4-dimensional vector consisting of the cart's position, velocity, angle, and angular velocity.
- The discrete action space is two dimensional, consisting of left and right.
- The reward function is 1 for every timestep that the agent manages to keep the pole balanced. The maximum length of an episode is 200 timesteps.
- The criteria for successful learning is an average reward of 195 over 100 consecutive episodes.

5.2 Configuration & Architecture

The benchmark cart pole problem is a classic example of a reinforcement learning problem. It is a simple problem where an agent must learn to balance a pole on a cart. The state representation is a 4-dimensional vector consisting of the cart's position, velocity, angle, and angular velocity. The discrete action space is two dimensional, consisting of left and right. The reward function is 1 for every timestep that the agent manages to keep the pole balanced. The maximum length of an episode is 200 timesteps. The criteria for successful learning is an average reward of 195 over 100 consecutive episodes.

Report the architecture of the policy neural network used in your experiments.
Provide the detailed algorithm parameter settings of OpenAI-ES.

The architecture of the policy neural network used in the experiments has four dimension with a hidden layer with sixteen dimensions and two outputs. The activation function for the hidden layer uses relu, and linear for the output layer.

The algorithm parameter settings of OpenAI-ES are:

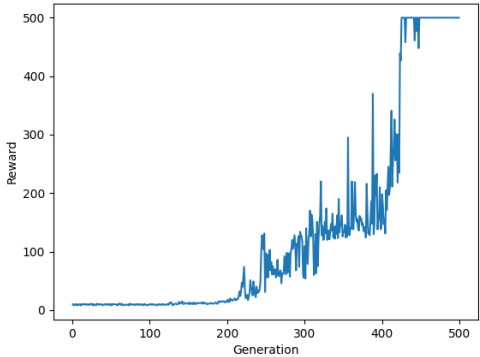
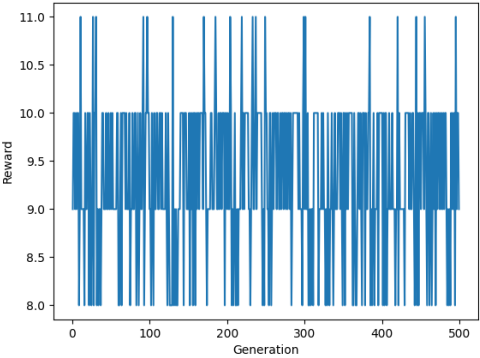
- `sigma_init`: 0.01
- `sigma_decay`: 0.999
- `learning_rate`: 0.001
- `learning_rate_decay`: 0.9999

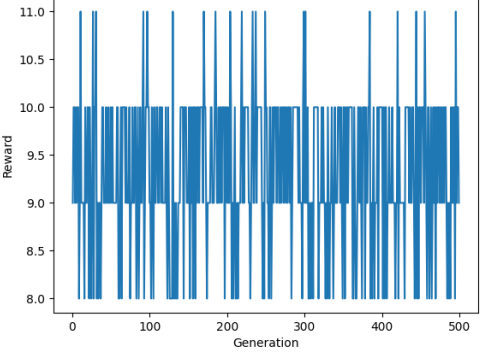
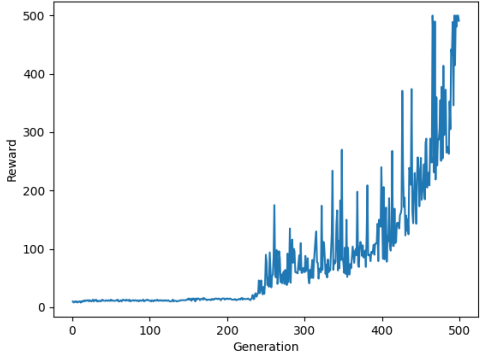
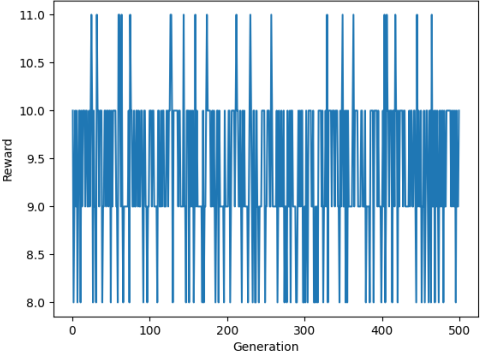
Sigma init is responsible for the initial noise standard deviation. Sigma decay is the parameter that controls how quickly the noise standard deviation decays over time. Learning rate is the

parameter that controls how quickly the network learns. Learning rate decay is the parameter that controls how quickly the learning rate decays over time.

This combination of parameters is designed to allow the network to learn quickly at first, and then slow down the learning as it gets closer to the optimum solution.

5.3 Results and Discussion

<i>Zoo Reinforcement Learning Results</i>	
<i>Seed/Run:</i>	<i>Reward Convergence (500 Generations):</i>
1	
2	

3	
4	
5	

In only two of the instances was the algorithm able to converge with the parameters I set. Earlier runs with less comprehensive exploration could not converge at all, and the same trend was observable with oscillation between two extremes and a relatively common middle ground where the algorithm was trapped. Across the other three instances this was observable, with the majority of the generations staying in the range of 9.0 to 10.0 on the reward function. Of the two runs which did seem to learn through the training process, only run 1 was able to plateau - However the success criteria of finding an average reward of 195 over 100 episodes was exceeded for each of these two, with the reward approaching 500. The exponentiality of the reward growth is clearly observable in the graph, with a sharp increase in the average reward around generation 250.

The lack of success in the other runs is likely down to the exploration noise not being high enough in the early generations. This can be seen in run 4, where the average reward quickly stagnates at 10.0 after an initial increase. It is likely that with more noise the algorithm would have been able to explore the state space more and found the optimum solution, rather than becoming trapped in strategies that yield short term benefits with overall penalties to the performance (Balancing the cart in the immediate instant at the cost of it failing in the next).