



Evolutionary Algorithms for Optimization, Computer Vision, and Machine Learning

Objectives

The overall goal of this project is to review and practise evolutionary computation technologies for optimization, computer vision and machine learning. Specifically, the following ideas, methods and algorithms should be reviewed, understood and practised in the project:

- Implement evolutionary programming and differential evolution algorithms and study their performance at solving continuous optimization problems;
- Study the effectiveness of using estimation of distribution (EDA) algorithms for solving combinatorial optimization problems with properly designed fitness functions;
- Design appropriate fitness functions required by cooperative co-evolution algorithms for solving symbolic regression problems and study the algorithm performance;
- Explore genetic programming algorithms for computer vision applications (image classification);
- Study the use of evolution strategies algorithms for training neural networks

1 Evolutionary Programming and Differential Evolution Algorithms

Objective

- Implement evolutionary programming and differential evolution algorithms and study their performance at solving continuous optimization problems;

Problem Domain

The problem here is to find 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:

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

Griewanks's function:

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

Dataset Evaluation

For this problem there was no provided dataset. Despite this, we know that Rosenbrock and Griewanks functions are solved problems and we know that both have a global minimum of 0, so we will use this for the evaluation of our algorithm.

Solution

To solve this problem we will first implement a form of the (1)Evolutionary Programming (EP)algorithm and then a (2)Differential Evolution (DE) algorithm.

For both algorithms I had a **stopping criteria** of a) max generations (250) and b) minimum acceptable error threshold reached (defined by the known optimal value for each algorithm). I also had an initial **population size** of 100 individuals.

EP Algorithm

EP was one of the first genetic algorithms ever introduced and differs from standard genetic algorithms in that the emphasis in this paradigm is on behavioral evolution rather than genotype, thus no crossover is used in favor of mutation. Behavior is modeled by strategy parameters from distributions and children and parents are pooled together and fight for survival based upon their relative fitness within the entire generation, instead of using Boltzmann selection to choose whether or not the children replace the parents, or the children always replacing the parents. This is calculated by how many opponents they are better than in a randomly selected tournament.

EP is most often used in constrained environments such as scheduling and routing, power systems, and designing systems. Its application of relative fitness functions have also made it powerful at solving game theory problems such as the prisoners dilemma - However given that we will be solving for a continuous problem domain, this will not be applied here.

The Form of evolutionary programming I implemented was a variation of both Fast EP (FEP) and Improved EP (IEP). That is to say, I implemented Cauchy distribution of noise sampling as it has a higher probability of sampling large noises which speeds up convergence. On top of this, I also implemented the self-adaptive mutation technique from IEP and a variation on tournament selection.

Where traditionally tournament selection involves iterating through a subset of the population to compare each individual in order to place each individual in the community, I found this to be incredibly inefficient and slow when implemented in python. Instead, I used the inbuilt sort function to sort the tournament pool and return the N best values. This is because this sort function is implemented in C which is far faster than python.

To achieve this as simply as possible I created a class to Encode my Individuals which primarily consisted of a list of floats to represent the individual itself as this is a continuous EP problem, as well as a float to represent the fitness of each individual for the sorting function. Also included in this class are the values for factor calculation in IEP (ie strategy parameter), the Individual size (D), and a collection of randomly generated variances to update in IEP.

I choose to implement this hybrid as it facilitated both the higher level of explorability with the Cauchy distribution, and was able to vary this as needed with increased levels of exploitation with the self adaptive mutation and variance provided by IEP.

The **Fitness Function** used to determine the fitness of each individual was in two stages. The first is fairly standard across Evolutionary Computing, being to find the absolute error between the true minimums of the functions (both 0 in this case), and the value calculate when passing the individual through each function. In reality, for this continuous EP problem, this simply equates to the value of the function itself as we know $f(x) \geq 0$ for all values of x and we are simply minimizing this value.

However, given that this is an EP problem I implemented relative fitness on top of this for comparison with co-evolving systems. This is implemented in my tournament selection which is concerned with how many “wins” each individual has compared to the others and maximizes for this. While this is a continuous problem, and not attempting to solve game theory and it is technically not required here, I felt it would be a good learning opportunity for myself to implement. To simplify, my algorithm first minimizes for the error and then finds how many times each individual has a lower fitness than another in the population. I could have extended this to use proportionate relative fitness which compares creating a proportion between fitness values and comparing it with the normal distribution, however this felt overly complicated when the whole premise is largely academic.

Given that no genotype is evolved in EP, the only Parameters we need to be concerned with are that for the mutation and the initialization themselves. While historically we have used static (within a fixed bound) and dynamic mutation (sliding scale), my EP allows for self-adaptive mutation which essentially means it has the ability to increase, decrease, or stay the same at each instance of mutation and makes parameter setting far more complex than for basic EP. The mutation formula for is as follows:

$$x_{i+1}(t) = x_i(t) + \Delta x_i(t)$$

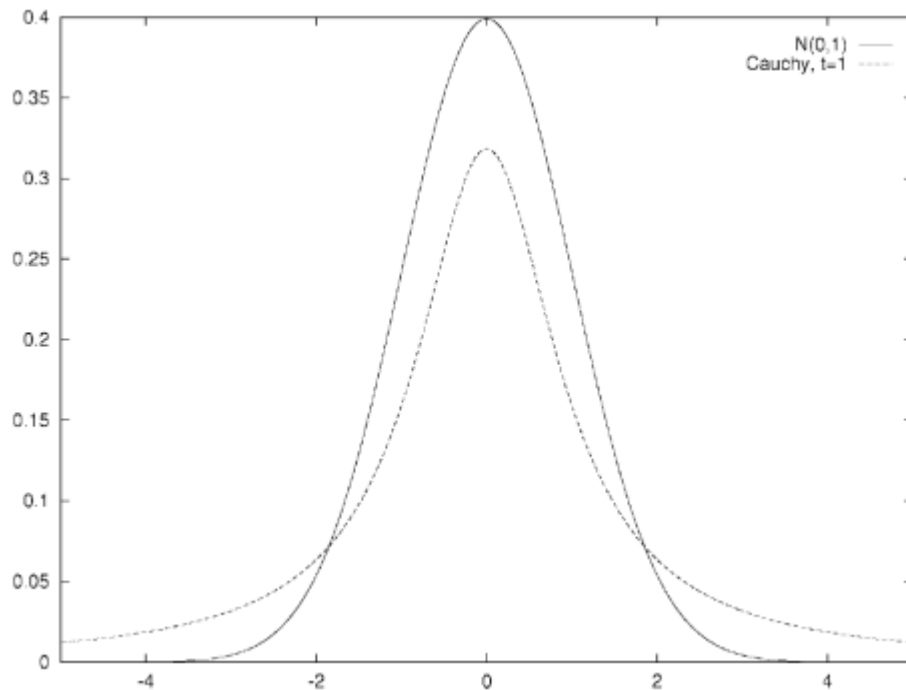
where $i \in [1, N]$ and $\Delta x_i(t)$ is calculated as

$$\Delta x_i t = \Phi(\sigma_i(t))\eta_i(t)$$

where Φ represents the probability distribution, σ represents the strategy parameter (updated using the lognormal method), and η_i represents the step size. Because each individual can mutate in different directions, these values are stored within Individual.

For the **probability distribution parameter**, I used a combined distribution of the Normal Gaussian Distribution with the Cauchy Distribution ($\eta_i(t) \sim N(0, 1) +$

$C(0, v)$), which is similar to normal (gaussian) distribution, except with a fair wider tail:



For the **Strategy Parameter** I used the factor calculation as discussed in lectures with a base sigma (σ) value of $(\sqrt{2\sqrt{n}})^{-1}$ and updated sigma value of $(\sqrt{2n})^{-1}$ and then updated this strategy parameter using the lognormal operator (or **Galton distribution**), which follows the equation:

$$X = e^{\mu + \sigma Z}$$

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


$$\text{where } \mu_1 = \frac{1}{\sqrt{2\sqrt{n_x}}}, \mu_2 = \frac{1}{\sqrt{2n_x}}, \text{ and } n_x \text{ denotes number of variables}$$

It is worth noting that the lognormal operator has a tendency to converge aggressively with small parameter values. For this reason it is typically accompanied by a minimum value that can be used in the event the function dips below this value.

Additionally, on researching for this report, I came across the **exponential evolutionary programming** variant of this algorithm, which uses the double exponential distribution as the mutation operator where the strategy parameter and selection process are variable. This could be worth further investigation however seem beyond the scope of this assignment.

DE Algorithm

Medium

 <https://towardsdatascience.com/differential-evolution-an-alternative-to-nonlinear-convex-optimization-690a123f3413>

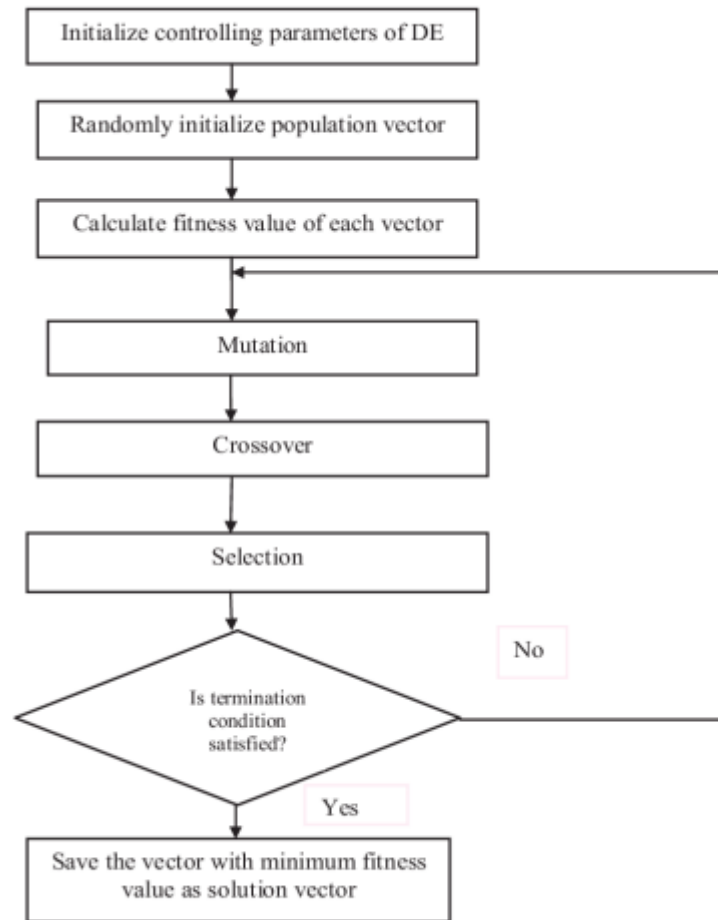


the Differential Evolutionary Algorithm is *population-based* Evolutionary algorithm that was originally designed to solve, and can be very effective in solving, optimization problems over continuous domains. It is commonly used as an alternative to both Particle Swarm Optimization (PSO) and more traditional Genetic Algorithms (GA) as it is a hybrid global optimization algorithm which inherits the crossover and mutation operators from traditional EA, along with movement and velocity from Swarm Intelligence

I choose it for this problem as it has been designed to fulfill some specific requirements I found to be quite useful:

1. Ability to handle non-differentiable, nonlinear, and multimodal cost functions.
2. Parallelizability to cope with computationally intensive cost functions.
3. Ease of use: few control variables to steer the minimization. These variables should also be robust and easy to choose.
4. Good convergence properties: consistent convergence to the global minimum in consecutive independent trials.

The basic workflow of this algorithm is similar to EA as represented below:



The DE algorithm initializes a population of N individuals (or agents) within the boundaries of each decision variable of the problem randomly distributed throughout the search space. Each individual corresponds to a vector of the optimization variables. Again I created a class to hold these values which extended the EP Individual class where the individual itself was represented by a list of floats

Two parameters critical for optimization performance are the values of **Population Size** ($N_{initial}$), **Crossover Rate** (CR), and **Differential Weight** ($0 < F < 1$) - this parameter is also referred to as the mutation parameter or scale factor. As such, the selection of these has been heavily researched to which it is generally accepted that $N_{initial}$ should equate roughly to between 5 and 10 times the number of decision variables (D here), $CR = 0.9$, and $F = 0.8$. This is because most real world problems tend to be discontinuous and nonseparable. However for our problem we have neither of these caviats. This allows us to use a lower value for F due to the low nature of the limits as we seek to prioritize exploitation over exploration. I found values for CR between 0.2-0.5 and the complementary value of 0.2 for F to improve

results. I also used the value of 100 for $N_{initial}$ as this vastly sped up the algorithm run time.

Until the Termination Criterion is met (as described earlier), for each agent x , 3 distinct agents are selected at random (a , b , c , and they must also be distinct from x) which then undergo mutation and crossover.

The mutation scheme DE/rand/1 was chosen as it provides a balance between exploitation and exploration. The formula for this is represented below:

$$v = a + F(b - c)$$

where v_i represents the mutant vector and a , b , and c represent the distinct agents selected prior. normally this should be done per index i , however, the NumPy library in python allows us to perform this operation over the entire list of vectors simultaneously.

The Crossover scheme chosen was the binomial crossover which can be represented as such:

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

where u represents the new vector by the crossover operation on the mutant vectors v and x , CR is the Crossover parameter. An additional rule states that at least one attribute of u **must** be inherited from v to prevent duplication.

Finally the fitnesses of the new vectors are calculated using the **fitness function**. This is again simply the absolute error between the calculated value for the individual in the equation and the true minimum (0). If the fitness is improved the individual x is replaced by the mutated x , otherwise it is ignored. This is then repeated for all individuals in the population.

Results

For $D = 20$, do the following I applied both EP and DE to task of optimising for the Rosenbrock and Griewanks Functions as these are both stochastic algorithms. I was initially expecting DE to outperform EP as it has been designed to find global minimums - however I implemented a more advanced variation of EP than DE.

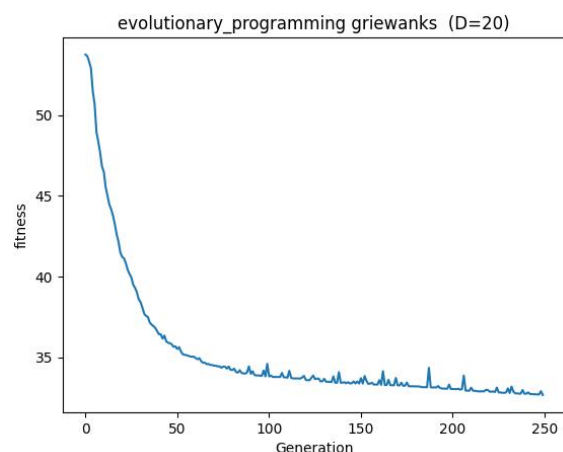
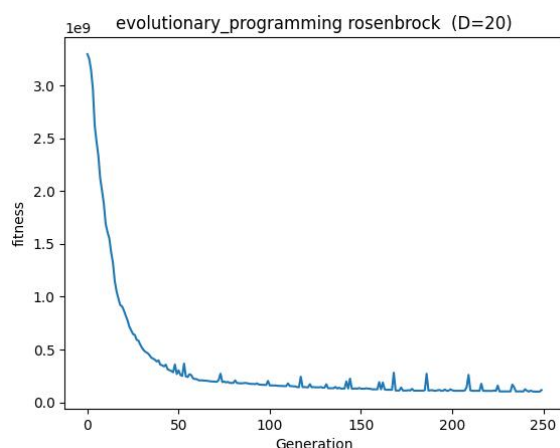
For each of these I ran the algorithms 30 times and recorded the mean, standard deviation, and minimum and maximum fitness values for the best individuals from

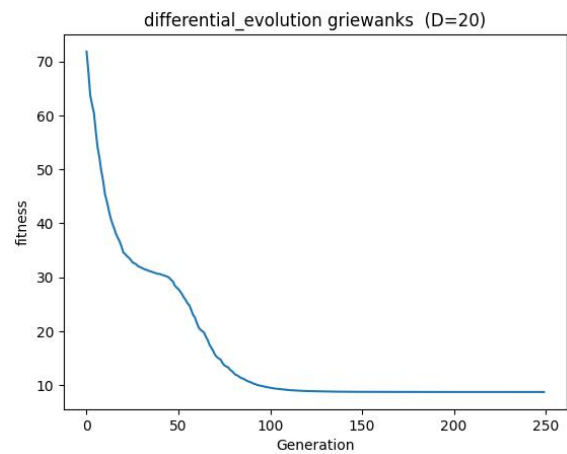
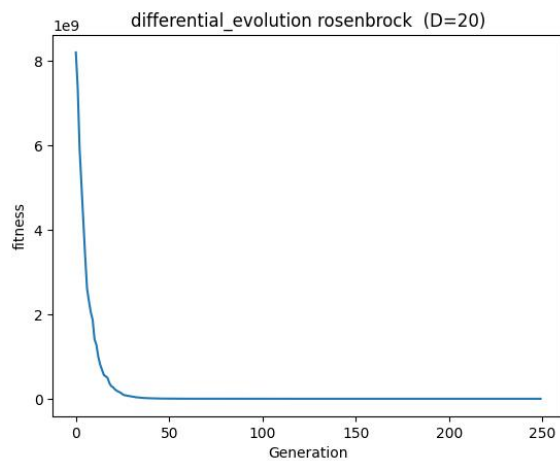
each iteration.

```
===Results for D=20===
    rosenbrock_evolutionary_programming  rosenbrock_differential_evolution  \
mean                                3.904336e+06                        102149.588
std                                3.662128e+06                        320981.233
min                                1.164242e+05                         251.362
max                                1.624838e+07                        1763198.832
    griewanks_evolutionary_programming  griewanks_differential_evolution
mean                                1.089                             0.292
std                                0.064                             0.185
min                                0.892                             0.049
max                                1.216                             0.677
```

As expected DE significantly outperformed the EP algorithms as it was far closer to the target values (0) for both griewanks and rosenbrock. While it does seem that the std for the EP algorithm is better than that for the DE one in Griewanks, you also have to consider that the worst value for the DE over both functions is better than the best value for the EP algorithm.

The convergence curves for these algorithms can also be seen below. It is interesting that the EP approach has noticeable deviations from convergence towards the later generations, despite employing elites. This is likely due to the nature of tournament selection which, while speeding up the execution time significantly, comes at the cost of accuracy. Comparatively the DE algorithm converges much more smoothly and especially quickly on the Rosenbrock function, with a notable slump for the griewanks function.





I then repeated this experiment for $D = 50$, for purely the Rosenbrock function and using the same algorithm settings. The same metrics were recorded and can be seen below:

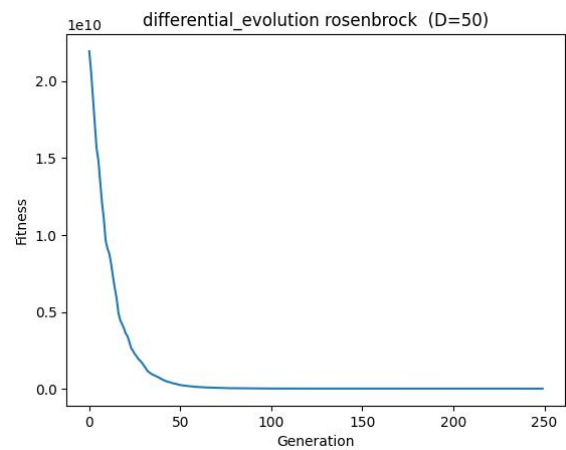
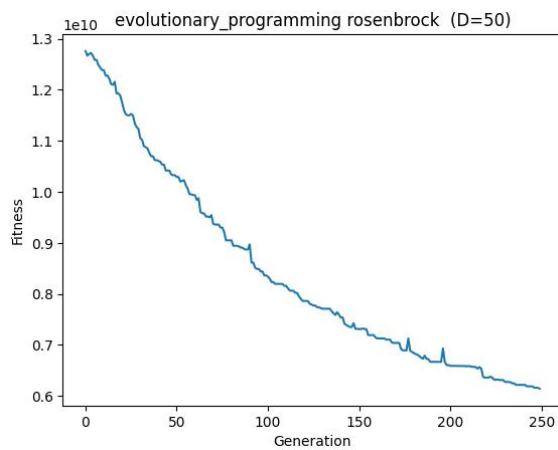
```

===Results for D=50===
      evolutionary_programming  differential_evolution
mean          2.047966e+08          1046777.136
std           5.806939e+07          660885.985
min           1.109811e+08          238546.803
max           3.254362e+08          3033217.584

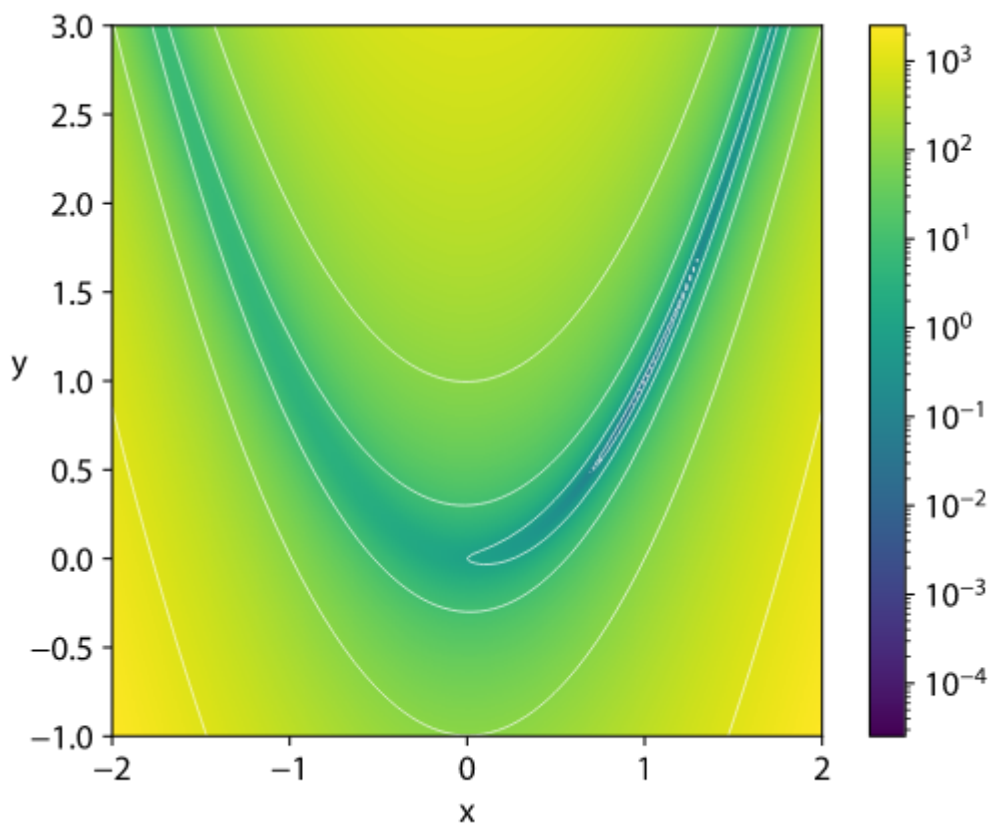
```

As could have probably been predicted both the variability and quality of results has increased dramatically. This is likely due to the increased difficulty in solving the Rosenbrock equation as Dimensionality increases. As mentioned prior this is a solved problem, however it has been solved to have a minimum of 0 at (1, 1), or $D=2$, which is many orders of magnitude in difference from $D=50$, or even $D=20$.

This can be visualised by the distinction between the convergence curves at $D=50$ where the EP algorithm converges almost linearly and likely needs additional generations to converge. We can even see a slight slow down in the DE algorithm compared with $D=20$.

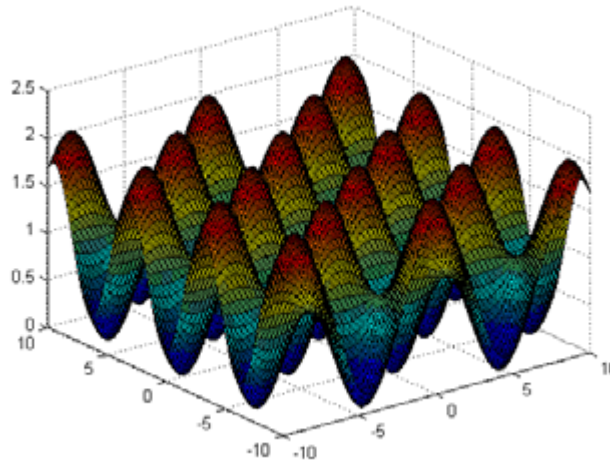


While it is not possible to determine here, a possible reason for the significant distinction between the rosenbrock and griewanks functions could be due to the relative shapes of these algorithms. The rosenbrock function as shown below, which has 1 global minimum with a sharp and narrow descent to it.



<https://en.wikipedia.org/wiki/File:Rosenbrock-contour.svg>

Comparatively the Griewanks function has for more local minimums and a less consistent solution domain as depicted below:



https://www.researchgate.net/figure/The-shape-of-Griewank-function_fig6_241177369

it is therefore possible that over a significantly large number of generations the Rosenbrock function could converge to the true minimum at a similar number of evaluations as the Griewank's function - which appears, to converge to a local minimum very quickly but have trouble finding the global minimum.

2 Estimation of Distribution Algorit

Objectives

- Study the effectiveness of using estimation of distribution (EDA) algorithms for solving combinatorial optimization problems with properly designed fitness functions;

Problem Domain

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.

$$\max(v_1x_1 + v_2x_2 + \dots + v_Mx_M)(1)$$

$$s.t.(w_1x_1 + w_2x_2 + \dots + w_Mx_M \leq Q)(2)$$

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

where $x_i = 1$ means that item i is selected, and $x_i = 0$ means that item i is not selected.

The 0-1 knapsack problem is known to be NP-hard, and exhaustive search is not applicable for practical cases. Genetic algorithm (GA) is a promising approach for this problem, to find a reasonably good solution within a limited time budget.

Dataset Evaluation:

- 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**.
- 100 995: with 100 items and bag capacity of 995. The optimal value is **1514**.

The format of each file is as follows:

```
M Q
v1 w1
v2 w2
... ..
vM wM
```

In other words, in the first line, 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.

Solution

To Solve this problem I developed a simple Estimation of Distribution Algorithm (EDA). This will solve the optimization problem by keeping track of the statistics of the population of candidate solutions. Contrary to most EA's, EDA only tracks the statistics and generates a new population at each generation from the previous populations statistics (probability vector). EDA's also typically solve for discrete optimization problems. For this reason I choose to use a list of booleans as my individual representation with a 1 representing that the item was present in the knapsack and a 0 indicating it was absent.

For this implementation I utilised the Population Based Incremental Learning (PBIL) Variation of EDA which uses first-order distribution approximators to generalise the

univariate marginal distribution algorithm (UMDA)

The steps of this algorithm are as follows:

1. A population is generated from the probability vector. The population size is very important for PBIL where small populations can lead to an “agressive learning process” and large populations can lead to very long convergence times. for this reason I used $N=1000$. I also set my learning rate (lr) to 0.7 as I found this compensated for the increased population size, and mutation probability vector ($mupb$) to 0.1 as this is often omitted in practice so a small value is valid.
2. The fitness of each member is evaluated and ranked. The fitness function I used here was a slight deviation from my fitness function from assignment 1. While they both sum the values and the weights of the items in the knapsack in an attempt to maximize value while minimizing weight - where previously I had excluded bags that had exceeded the capacity from the population, this time I adopted a softer approach to bags that were heavier than the max weight. This was to add greater variety to the knapsack itself.
3. Update population genotype (probability vector) based on fittest individual. This used the added variety from my new fitness function in this selection process.
4. Mutate.
5. Repeat steps 1–4

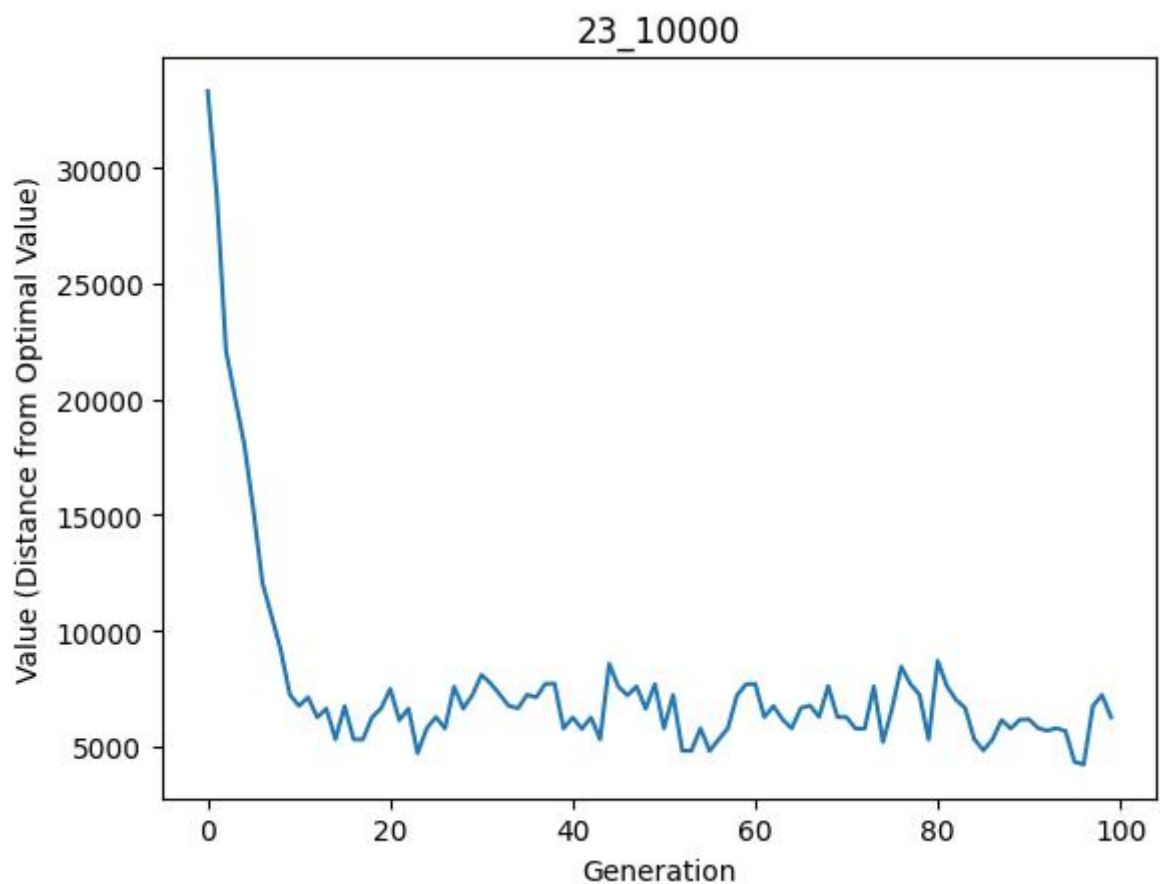
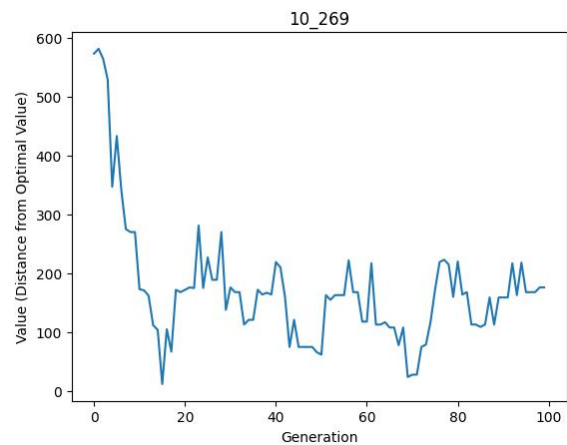
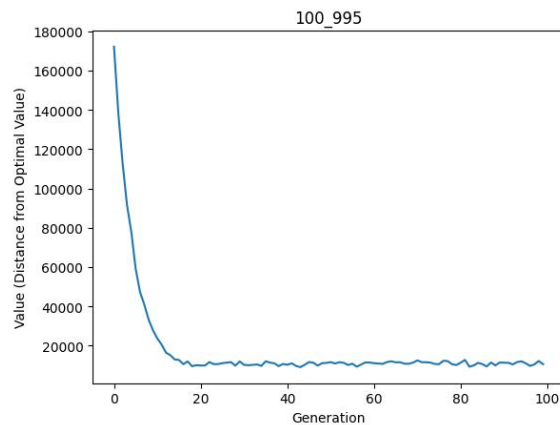
Results

For this knapsack problem instance, I ran my EDA implementation 5 times with seeds [1,2,3,4,5]. The mean and standard deviation of top results for each dataset across the 5 seeds is shown below:

```
100_995
mean: 1592.2
std: 377.69744505357727
10_269
mean: 295.0
std: 0.0
23_10000
mean: 9767.0
std: 0.0
```

As can be seen these are remarkably close to the optimal values. In fact we can see that both for the 10_269 datasets and the 23_10000 datasets they hit it every iteration. The anomaly is only present for the 100_995 dataset which I had a tough

time trying to get to converge on any value that wasn't infinite (which is why I have changed my fitness function and used such a large population size). Interestingly this is not reflected in the convergence curves which I have also drawn the convergence curve of the EDA implementation for each knapsack problem instance averaged over the runs. On the x axis is the number of generations, and on the y axis is the absolute error between the optimal values and the individual values.



A possible reason for this variability is that both over and undershoots will be represented as bumps upwards on this graph as it is absolute error. Additionally a possible reason it is not seen for the 100_995 dataset is the significantly large value that 100_995 starts at compared with its desired value.

Another option to smooth this curve would be to use elites which I did not - I kept the algorithm fairly basic as this was that seemed to be needed. Elites could possibly allow me to use a smaller population size, however.

3 Cooperative Co-evolution Genetic Programming

Objectives

- Design appropriate fitness functions required by cooperative co-evolution algorithms for solving symbolic regression problems and study the algorithm performance;

Problem Domain

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

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

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:

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

Solution

I developed a Cooperative Co-evolution GP (CCGP) to solve this symbolic regression problem. Other than how The CCGP contains two sub-populations, one for $f_1(x)$ and the other for $f_2(x)$, this solution stuck fairly closely to my initial implementation for symbolic regression. I also utilized the deep library for CCGP and stuck closely to exemplars they have produced. As such

Terminal and Function set: The Terminal and Function set was mapped as follows:

```
"add" <-- x + y
"sub" <-- x - y
"mul" <-- x * y
"neg" <-- x * -1
"div" <-- x / y (y > 0)
"sin" <-- sin(x)
"inv" <-- x ** -1
"sqrt" <-- x ** 0.5
"pow2" <-- x ** 2
"RAN" <-- random(0, 100)
```

These were chosen as virtually any equation can be created from some combination of these, and incorporated all of the operators in our equation.

The **Fitness** function used here was a co-operative one, and indeed evaluates differently for each of the sub populations. For both, however, the function returns the mean-squared error of the individual generated. That is to say, in the objective space ($-10 < x < 10$) every x value is tested for each individual as well as the regression problem function itself. The difference is then squared and summed. This program aims to minimise this error and so the lowest error will be selected for the next generation.

However, because all we know is that this function piecewise, the actual metric each sub pop is measured against changes for ($x > 0$) being evaluated against $(1/x + \sin x)$, and ($x \leq 0$) being evaluated against $(2x + x^2 + 3)$. This is because of the new information we have gleaned which tells us that these domains act independently from each other and so each is trained independently.

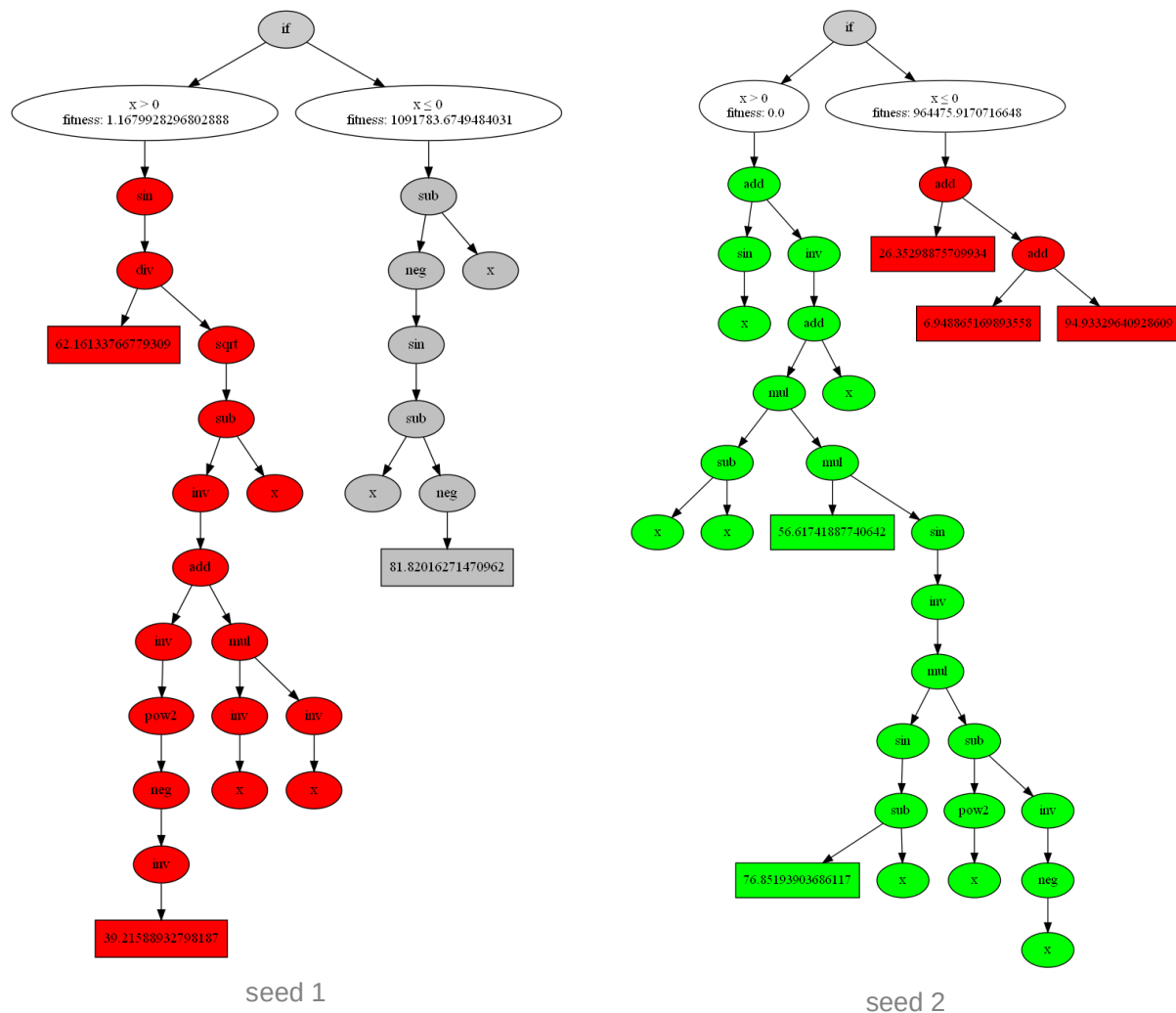
The **sub population size** was set to 1000 as guided in the DEAP tutorial with the **Max Tree Depth** again being capped at 17, as suggested in *John R. Koza, "Genetic Programming: On the Programming of Computers by Means of Natural Selection", MIT Press, 1992, pages 162-169.*

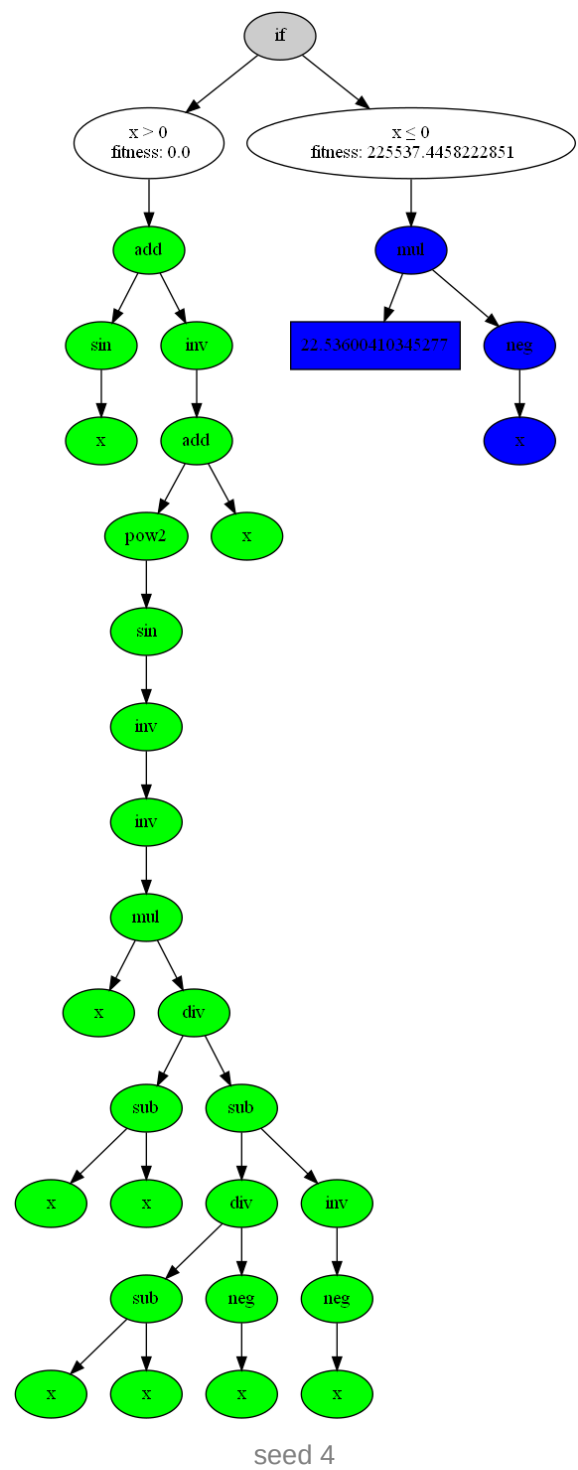
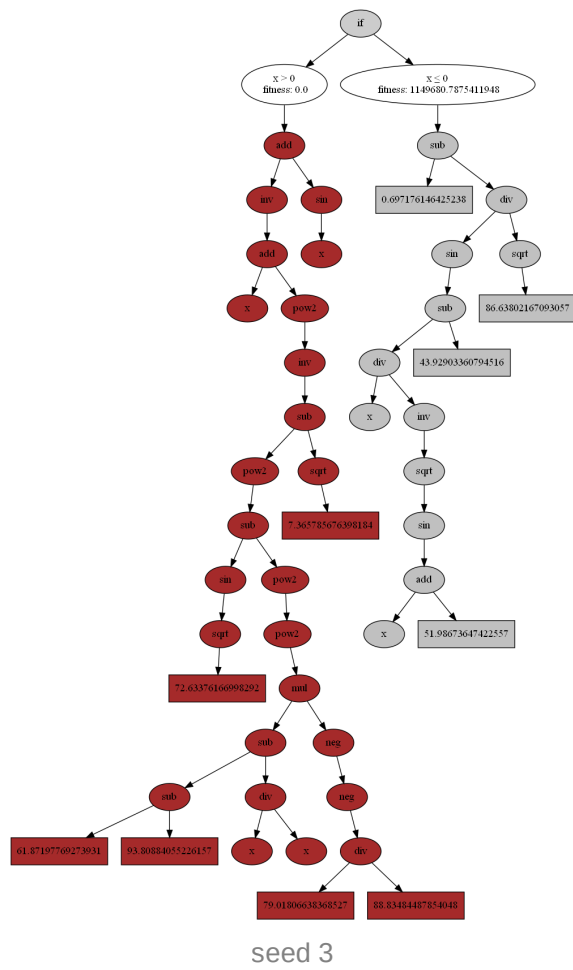
My **termination criteria** was set at max number of generations, being 150 as at this point most errors had converged to 0.

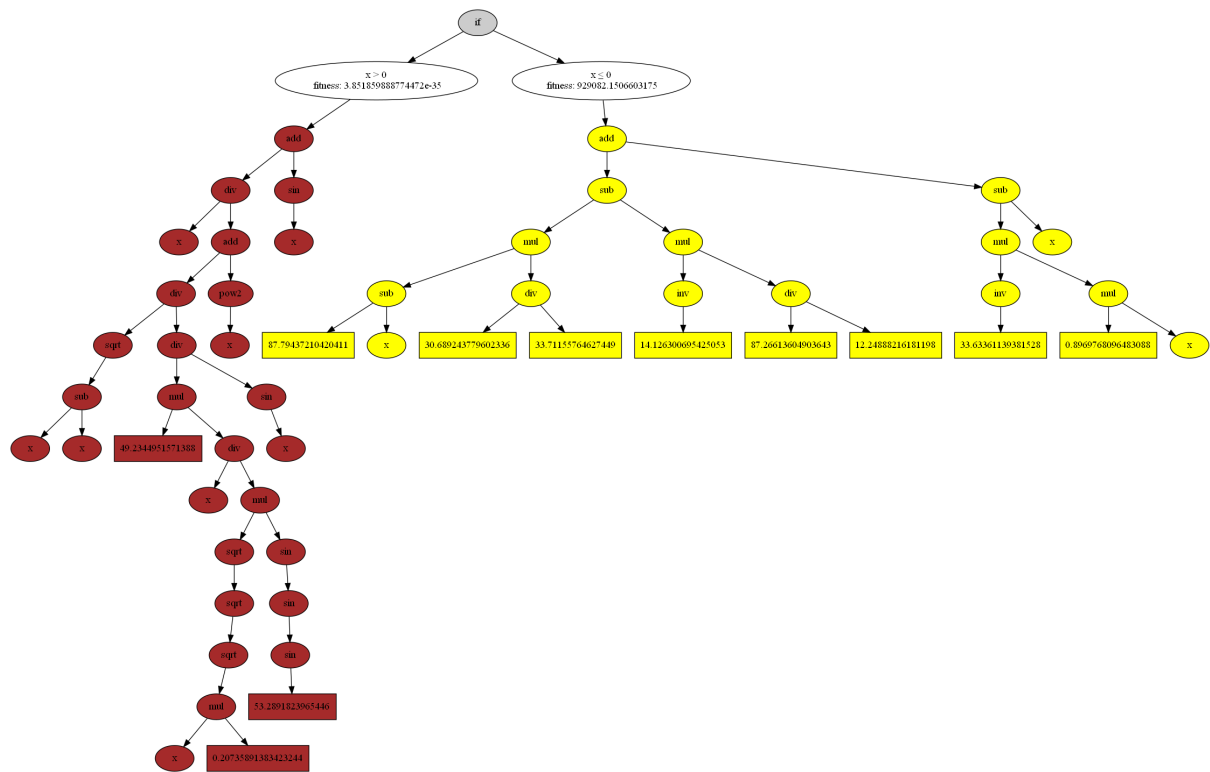
My **Mutation** and **Crossover rates** were set to 0.2 and 0.8 respectively.

Results

I ran the implemented CCGP for 5 times with different random seeds. The best programs for each sub population is represented below for each seed.







seed 5

Across all seeds $x < 0$ managed to converge consistently, whereas the performance for $x \geq 0$ was far more variable and required significant tweaking to converge at all. Interesting after removing the \cos operator the sizes of the trees increased significantly, however the performance of the $x \geq 0$ function improved dramatically so I felt it was a worthwhile trade off.

4 Genetic Programming for Image Classification

Objectives

- Explore genetic programming algorithms for computer vision applications (image classification);

Problem Domain

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. 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”. There are two steps that you need to perform to achieve this goal, as described in subsequent subsections.

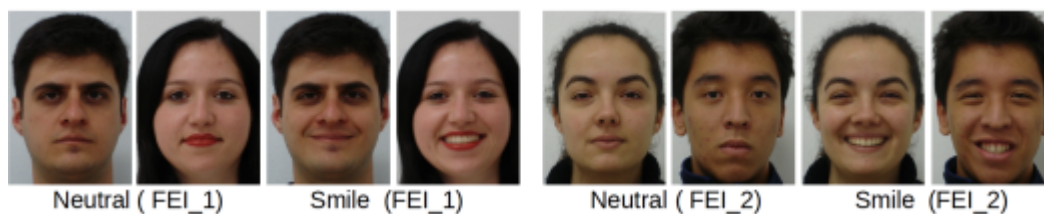


Figure 1: Example images from the FEI 1 and FEI 2 datasets.

Solution

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 contains training 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 you pattern files. Comma Separated Values (CSV) format is a good choice; it can easily be converted to other formats. 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 Naïve 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).

Results

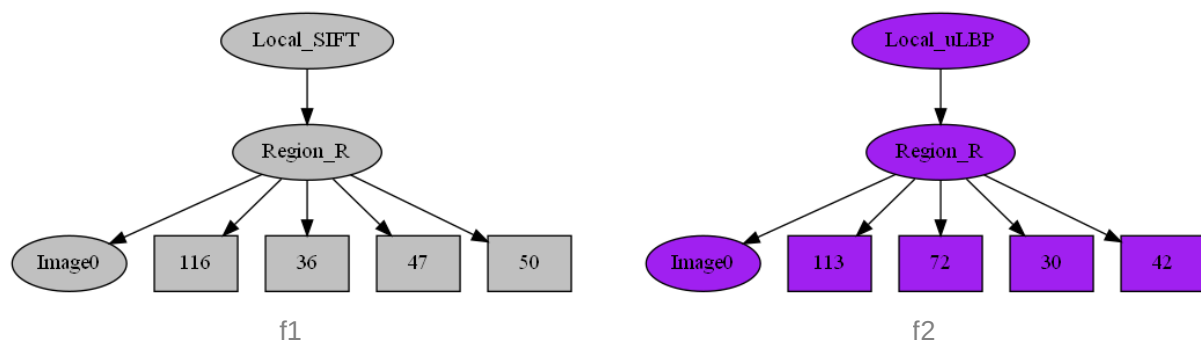
For this Evaluation I choose the f1 score over the accuracy metric as it combines the precision and recall of a classifier into a single metric by taking their harmonic mean (also... one of the datasets is called f1). The results of this are below.

We can see that the f1 dataset performed significantly better than on the f2 dataset with very similar train and test results for both. However the f2 dataset took almost a second less to test.

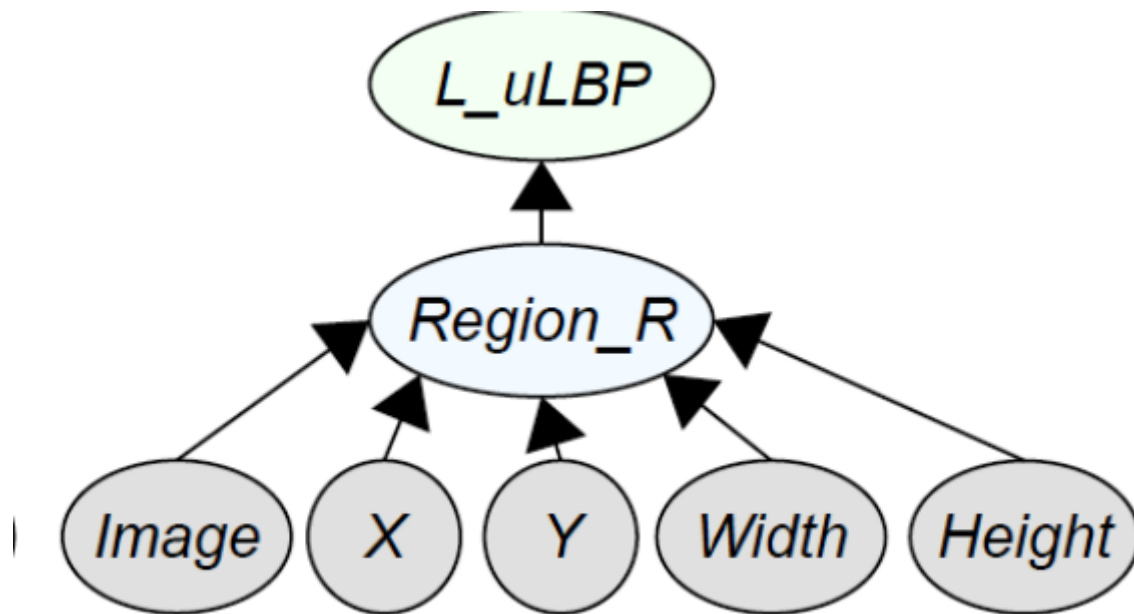
```
f1:
Best individual Local_SIFT(Region_R(Image0, 116, 36, 47, 50))
Test results (0.9795918367346939, 0.98)
Train time 1461.265625
Test time 1.359375

f2:
Best individual Local_uLBP(Region_R(Image0, 113, 72, 30, 42))
Test results (0.8510638297872342, 0.86)
Train time 1409.859375
Test time 0.484375
```

The GP trees obtained can be seen below (f1 on the left and f2 on the right).



Interestingly both opted to use local features of the image in classification suggesting that only small parts of the images were being used. that being said the parts that were used made up significant portions of the image (with the images being size **180x130**).



This is especially more noticeable for the f1 result which, unsurprisingly, both uses more of the image and has a better result.

Interestingly, however, despite this, we can see that f1 is using SIFT which is **Scale-invariant feature detection** which suggests that the size of the zone being used shouldn't matter as it will simply identify interest points to base classification off of. By contrast, f2 is using LBP or **Local Binary Pattern** which that labels the pixels of an image by thresholding the neighborhood of each pixel and considers the result as a binary number, and is supposedly very efficient at it.

On possible way to improve the accuracy across both datasets would be to increase the maximum depth of the tree which would provide more pathways for the GP to use to classify images.

5 Evolution Strategies for Training Neural Networks

Objectives

- Study the use of evolution strategies algorithms for training neural networks

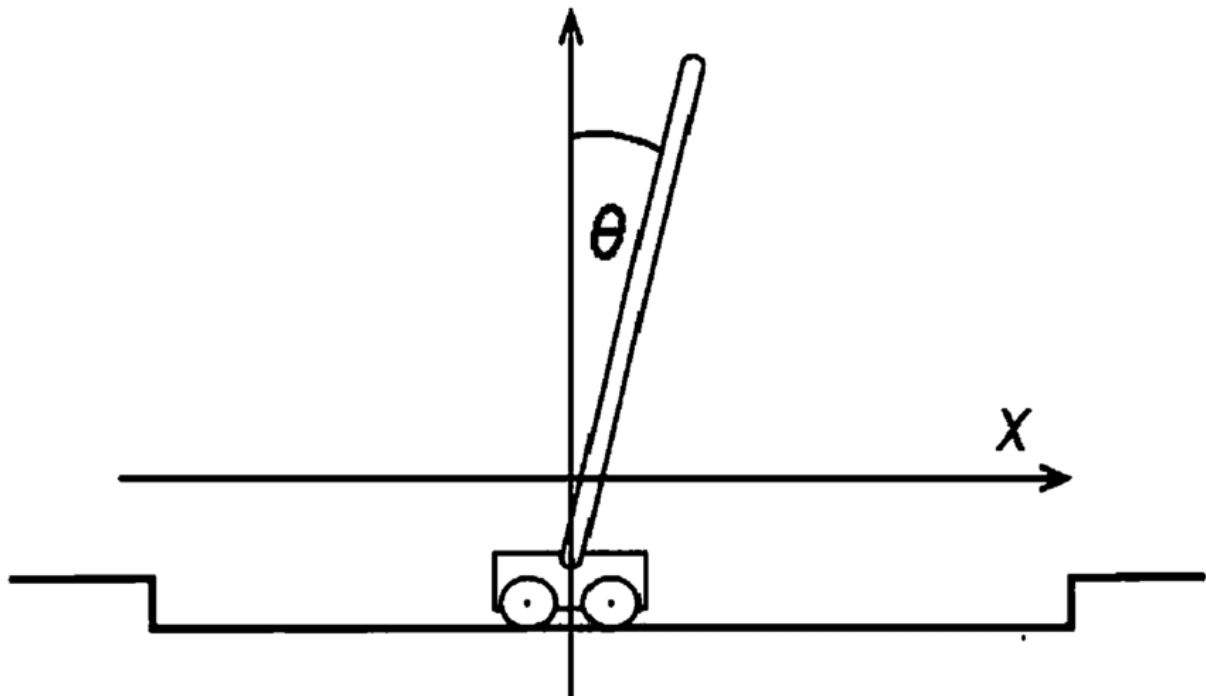
Problem Domain

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.

Solution

The Cartpole problem, or Inverted Pendulum problem, (as implemented by open AI) is a control problem that simulates a cart moving frictionlessly along one dimension with a pole attached to it above its pivot point as depicted below:

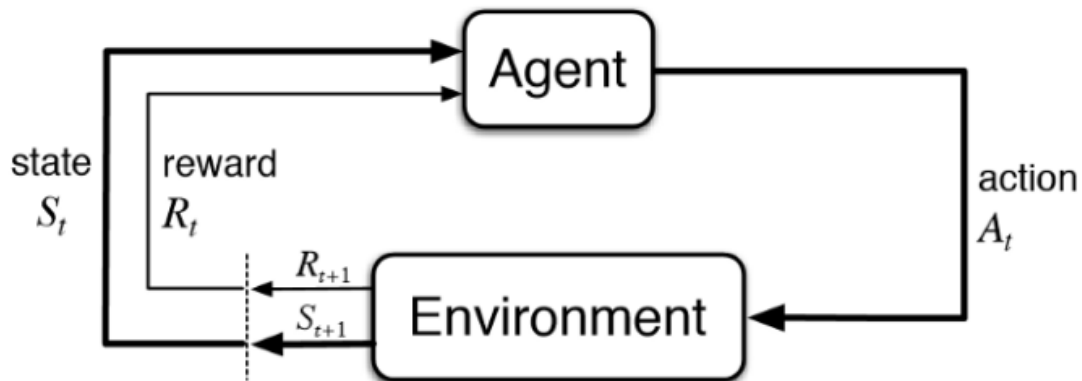


https://www.researchgate.net/figure/Cart-pole-balancing-problem-The-force-applied-to-the-cart-is-atF-where-01-at-1-is_fig2_6421220

It is unstable as the pole will move differently in response to the motion of the cart, and will also affect the motion of the cart. If the absolute angle of the pole exceeds 12° it is considered to have “fallen over” and if the absolute position of the cart exceeds 2.4m it is considered to have fallen off the cliff“. It is often tackled as a reinforcement learning problem where the goal is to balance the cart, and here we will be discussing a solution to this implemented through neural nets.

This implementation was based on top of OpenAI’s gym which is a collection of environments to develop and test RL algorithms and is built on a Markov chain

model as depicted below:



In this implementation, the **state representation** of the cart is a 4-dimensional vector that represents the cart position, the cart velocity, the pole angle, and the pole velocity at tip, with the **discrete action space** being represented as a 2-dimensional vector which contains the force applied to the cart (ie moved left or right).

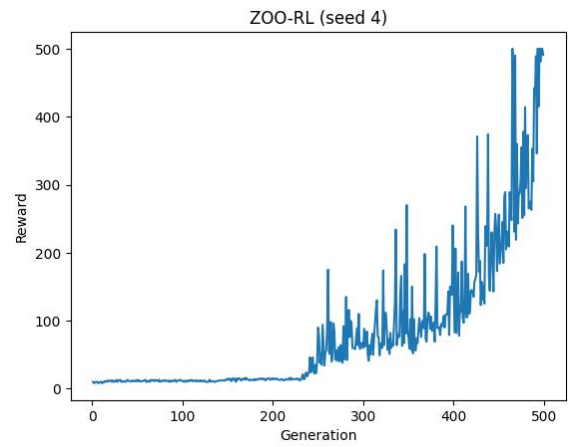
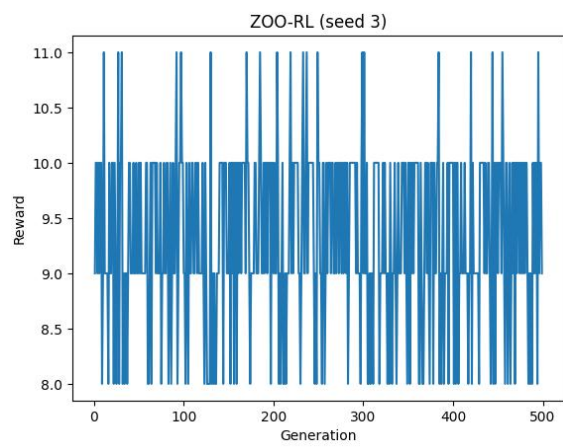
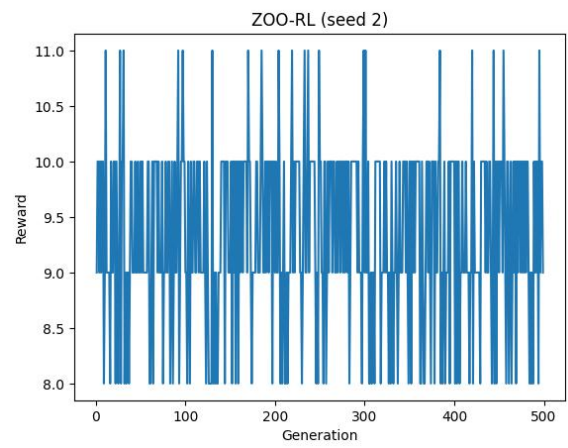
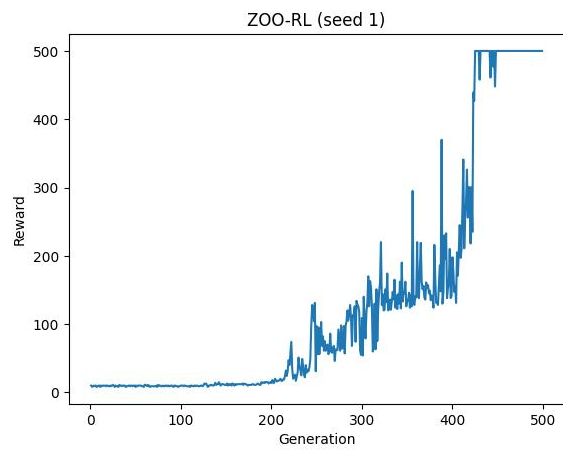
The **Architecture** of this experiment is a 4-dimensional input layer with a 16 dimensional, hidden layer and 2-dimensional output layer. The activation function used at the hidden layer is tanh and at the output layer, it is softmax. I have used a learning rate of 0.001 which decays by a factor of 0.9999 and a sigma value of 0.01 which decays by 0.999 per run. The initial population size is 100 and the **maximum number of episodes** is the number of generations, being 500. Here the fitness function is being maximized.

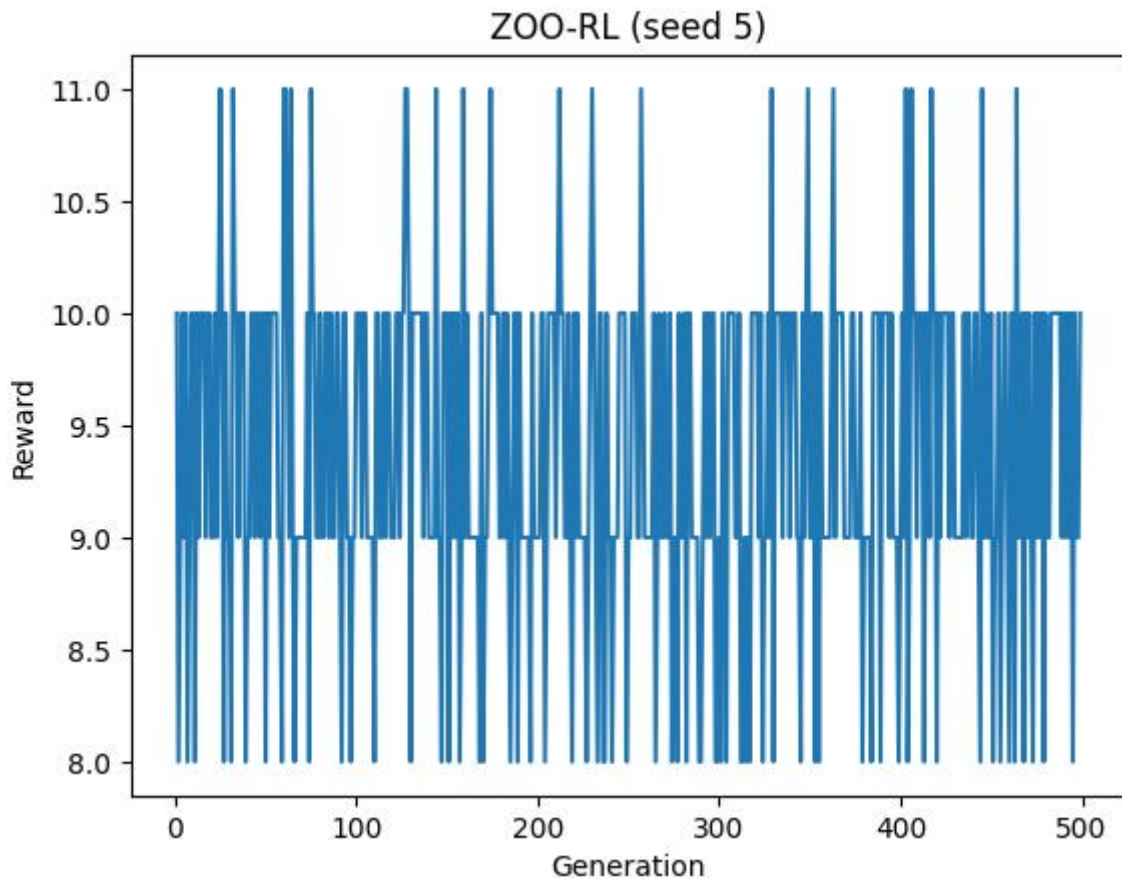
Evaluation

The **reward function** is 1 for every step taken, including the termination step. **The criteria for successful learning** is when the average reward is greater than or equal to 195.0 over 100 consecutive trials and is capped at 500.

Results

The converge for the average results across generations can be seen below:





As can be seen seed 2,3 and 5 failed to exceed a reward value of greater than 11 - indicating that these failed to train across the 5 generations. Both seeds 1 and 4, however, managed to reach the maximum fitness of 500, with both converging after 400 generations, albeit only briefly for seed 4 before returning just before the 500th generation.

Based on our success criteria only seed 1 was successful for 100 or more consecutive episodes.