Assignment 1

Ananta Shahane
s2900718
`a.a.shahane@umail.leidenuniv.nl`

December 8, 2022

1 Introduction

In this assignment, we implement a *Genetic Algorithm* and an *Evolution Strategy* on a neural architecture search problem.

1.1 Problem Statement

We are given a neural network architecture to search. This architecture has total 7 nodes, which needs to be connected in different ways and 7 operation nodes. In these operation nodes, one of them is `INPUT` other is `OUTPUT` node, and there are total 3 distinct kind of filters, that needs to be applied in 5 remaining ways. Giving us the operations as:

$$[\text{INPUT} f_1, f_2, f_3, f_4, f_5, \text{OUTPUT}] \tag{1}$$

These filters are `conv1x1-bn-relu`, `conv3x3-bn-relu` and `maxpool3x3`. The node connection configuration is given by adjacency matrix as stated below:

$$\text{Adjacency matrix} = \begin{vmatrix} 0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \\ 0 & 0 & x_7 & x_8 & x_9 & x_{10} & x_{11} \\ 0 & 0 & 0 & x_{12} & x_{13} & x_{14} & x_{15} \\ 0 & 0 & 0 & 0 & x_{16} & x_{17} & x_{18} \\ 0 & 0 & 0 & 0 & 0 & x_{19} & x_{20} \\ 0 & 0 & 0 & 0 & 0 & 0 & x_{21} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{vmatrix} \tag{2}$$

For this problem statement we represent this problem as a 26 parameter search problem given by X where $x_i \in X \forall i \in \{0, 1, 2, \ldots, 26\}$ such that:

$$x_i \in \begin{cases} \{0,1\} & 0 \leq i \leq 20 \\ \{0,1,2\} & 21 \leq i \leq 25 \end{cases}$$

Hence now we can see that the problem is to find an optimal solution for the neural network we have a search space of size:

$$S = 2^{21} \times 3^5$$

Start with discussing how the genetic algorithm was implemented followed by the result that it yielded. Then we move on to evolutionary stratagy and it's results.

2 Genetic Algorithm

In genetic algorithm, we are dealing a bit string representation of the given data, and since we are in neural architecture we are dealing with the adjacency matrix for first 21 elements of the vector, but for the last 5 elements, since we have 3 choices, we need to encode it. I decided to go for the the encoding showed in Table 1. Hence now we are dealing with a bit-string of size 31. I used the tradational genetic algorithm as show in Algorithm 1. I started by coding some stubs that acts as a translation layer between `nasbench` and the Genetic Algorithms. These stubs are `GetMatrix(vector)`

Table 1: Translation Used.

DNA	RNA
00	Random
01	0
10	1
11	2

Algorithm 1: A framework of Genetic Algorithm

```
Input       : Parent Population size $\mu$
              Offspring population size $\lambda$
              Genetic Algorithm type $\{(\mu+\lambda), (\mu,\lambda)\}$
              Crossover probability $p_c$
              Mutation Rate $p_x$
Termination : The algorithm terminates when, budget $B$ runs out.
1 Parent ← Initialization(firstGeneration);
2 for $i = 1$ to $B$ do
3     Score ← Grading(Parents);
4     Parent ← Selection(parent, Score);
5     Offspring ← Crossover(Parent);
6     Offspring ← Mutation(Offspring);
7     Parent ← PopulationSelection(Genetic Algorithm Type, Parent, Offspring)
8 end
```

and `GetFilters(vector)`, both of them take the vector of size 26 as stated in introduction and returns a Matrix of form shown in equation (2) and filter settings shown in equation (1). These are rather trivial implementations so I will not go much deeper in their algorithm here.

2.1 Parent Selection

I decided to experiment with the first generation. There are the first generation of kind below:

$$G_1 = \text{Random vector of type } X \text{ in introduction.}$$

$$G_2 = \prod_{i=0}^{20} 0^i 10^{20-i-1} \prod_{i=1}^{5} random(0,1,2)$$

$$G_3 = \prod_{i=0}^{20} 1^i 01^{20-i-1} \prod_{i=1}^{5} random(0,1,2)$$

$$G_4 = \prod_{i=0}^{20} 0^i 1^{20-i} \prod_{i=1}^{5} random(0,1,2)$$

$$G_5 = \prod_{i=0}^{20} 1^i 0^{20-i} \prod_{i=1}^{5} random(0,1,2)$$

For each first generation the population size was fixed to 21 and then I ran the genetic algorithm. During the experiments I found out that G_2 parent Initialization resulted in all 21 individuals of first generation being invalid. Which means that the first few generations were wasted in atleast getting valid results. G_4 and G_5 is not any better as they had 11 invalid cases. While random never made more that 8 invalid individuals in first generation. So I stuck to G_1, for rest of the implementation. The algorithm for this is also trivial and will not be discussed.

2.2 Grading

The grading of the individual in the population is handeled by the `nasbench` framework It returns the “fitness" of the individual by stating if it is valid or not, and if it is valid, it returns the accuracy of the architecture. This is stated in Algorithm 2.

Algorithm 2: Scoring Algorithm

```
Input       : Population of current generation $X$
              Nasbench interface **nas**
Termination : The algorithm terminates when all individuals in the population are evaluated.
1  Scores $\leftarrow \phi$;
2  for $x_i \in X$ do
3      matrix $\leftarrow$ GetMatrix($x_i$);
4      config $\leftarrow$ GetFilters($x_i$);
5      if nas(matrix, config) is Valid then
6          Scores $\leftarrow$ Scores $\cup$ nas.Evaluation(matrix, config)
7      end
8      else
9          Scores $\leftarrow$ Scores $\cup$ 0
10     end
11 end
12 return Scores
```

2.3 Selection

For the selection process, I decided to resort to the roulette wheel algorithm, that takes the list of candidates, their scores and the number of selections required and returns a set of individuals of corresponding size. They may repeat, but the probability of them being in this return set is proportional to their fitness. Let us consider an example, say we have a population consisting of the set $S = \{a, b, c, d, e, f, g, h, i, j\}$ with corresponding fitness $f = \{0.1, 0.4, 0.5, 0.8, 0.6, 0.4, 0.2, 0.1, 0.9, 0.7\}$. Now to make a roulette table we start with an array say $r = [0]$. Then for each individual we append a value equal to sum of fitness of all the previously considered individuals. Hence:

$$r \leftarrow r \cup lastElement(r) + f[i]$$

This gives an a roulette table $r = \{0, 0.1, 0.5, 1.0, 1.8, 2.4, 2.8, 3.0, 3.1, 4.0, 4.7\}$ Now we can take a random number $c \in [0, 1]$ multiply by the sum of fitness, and search in r the index between which it fits. If it fits between index i and $i + 1$, we select individual number i from population. This works really well as for both it just take $\mathcal{O}(n)$ computation time and space. For this very example we can see the performance of this approach in Figure 1.

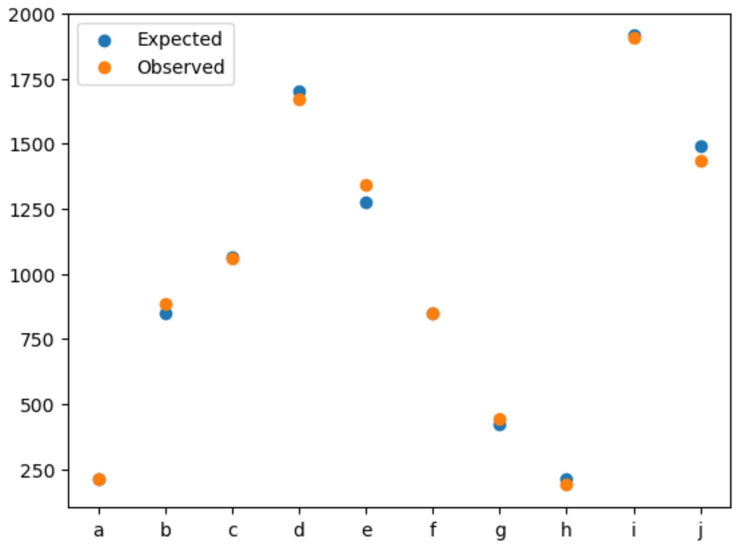


Figure 1: Selection Approach Performance.

The algorithm for mate selection is given below by Algorithm 3 which also takes into consideration the possiblility of whole population being invalid as seen when selecting first generation population G_2.

Algorithm 3: Selection Algorithm

```
Input       : Population of current generation $X$
              Score array of current population **Score**
              Offspring Population Selection Size $\lambda$
              Size of $X$ and size **Scores** needs to match.
Termination : The algorithm terminates when all individuals in the new population are selected.
rouletteTable ← $0$;
total ← $0$;
for $i \in \{0, \ldots count(X)\}$ do
    rouletteTable ← rouletteTable ∪ lastElement(rouletteTable) + Scores[i];
end
if total == $0$ then
    return Random Choices from $X$ of size $\lambda$;
end
newPopulation ← $\phi$;
luckyGuys ← $\lambda$ Random numbers in range $[0, 1]$;
for luckyGuy ∈ luckyGuys do
    luckyGuy ← luckyGuy × total;
    i ← 0;
    while !(rouletteTable[i] <= luckyGuy ∧ rouletteTable[i+1] > luckyGuy) do
        i ← i + 1;
    end
    newPopulation ← X[i] ∪ newPopulation;
end
return newPopulation
```

2.4 Crossover

Crossover is basically the process of “birthing”, here we take two parents and pull features from them to create an offspring. It is the “exploitation" part of the Genetic Algorithm, There are two distinct types of crossover namely

1. n-point crossover
2. Uniform crossover

In n point crossover, for the parents of length l, $1 \leq n < l - 1$. When n is 1 we call in 1-point crossover and when $n = l - 1$ we get a uniform crossover. Consider two parents p_1 and p_2, when we do a 1 point crossover, we pick a random point in p_1, which is the same point in p_2, then we copy everything after that point from p_2 into p_1. For $n > 1$ we create a new offspring o and for each point of crossover we deterministically switch between p_1 and p_2 to copy into o. For uniform crossover, we randomly pick each feature from p_1 and p_2 to be copied into o. For experimental purposes, I decided to let the program have an option to select between the two. So I came up with the algorithm 4

This is just one iteration of crossover, I use this algorithm on all adjacent parents, and on first and last parent, preserving the current population.

2.5 Mutation

Mutation is a process of randomly flipping a bit in a bit string. This simulates the random mutation in the nature, and hence falls under the “exploration” part of the Genetic Algorithm. Theoretically $p_m = 1/l$ is an optimal mutation rate, so we will use it as default. This is because if we keep getting closer to 1 we end up flipping more and more bits, giving us Morte Carlo Search, which is worse case scenario. Now we discuss the algorithm itself given in 5. Of course this is just for one candidate, so we can run it on each candidate.

2.6 Population Selection

Now, in order to decide which one of the algorithms to use between (μ, λ) or $(\mu + \lambda)$. The population selection method is used to choose among them. This algorithm takes two populations p_1 and p_2, where p_1 is previous two generations, and p_2 is the population of current generation. Hence we get the Algorithm 6

Algorithm 4: Crossover Algorithm
Only upto 10 **crossover points allowed for** n **point crossover**

```
Input       : Parent 1 $p_1$
              Parent 2 $p_2$
              Crossover Points $x_p$
              Crossover probability $p_x$
Termination : The algorithm terminates when crossover is finished

parentFlag ← True;
offspring ← $\phi$;
if $x_p \notin \{1, 2, \ldots, 10\}$ then
    for $i \in$ length($p_1$) do
        if randomNumber $\leq p_x$ then
            parentFlag ← !parentFlag;
        end
        if parentFlag then
            offspring ← offspring $\cup p_1[i]$;
        end
        else
            offspring ← offspring $\cup p_2[i]$;
        end
    end
end
else
    CrossoverPoints ← $x_p$ unique random numbers in range $(0, len(p_1) - 1)$ in ascending order;
    previousIndex ← $0$;
    for CrossoverPoint ∈ CrossoverPoints do
        if parentFlag then
            offspring ← offspring $\cup p_1$[previousIndex, CrossoverPoint];
        end
        else
            offspring ← offspring $\cup p_2$[previousIndex, CrossoverPoint];
        end
        parentFlag ← !parentFlag;
        previousIndex ← CrossoverPoint;
    end
end
return offspring
```

Algorithm 5: Mutation Algorithm

```
Input       : Candidate $x$
              Mutation probability $p_m$
Termination : The algorithm terminates when mutation is finished

for featureBit $\in x$ do
    if randomNumber $< p_m$ then
        featureBit ← !featureBit;
    end
end
return x
```

Now we just combine all the Algorithms [2, 3, 4, 5, 6] into 1, and we get our genetic algorithm.

Algorithm 6: Population Selection Algorithm

Input : Population of previous two generations: p_1
Population of current generation: p_2
Population Selection method: "," or "+"
Parent Population Size: μ

Termination : The algorithm returns (μ, λ) or $(\mu + \lambda)$ population

```
if selectionMethod == "," then
    return $p_2$;
end
else if selectionMethod == "+" then
    if length of $p_1$ > $\mu$ then
        return $p_1$ - (first $\lambda$ elements of $p_1$) + $p_2$;
    end
    return $p_1$ + $p_2$;
end
```

3 Evolution Strategy

For evolutionary strategy representation of the given problem statement as a 2–D problem search. Since we have a connection graph of 21 dimentions, and 5 permutations of 3 filters. We will have a search space of $(2^{21} \times 3^5)$ which turns out to be a (2097152×243) for all $(x, y) \in \mathbb{Z}^2$. Since the search space is not in $\mathbb{R}$ we need a translation layer between the problem and it's evolutionary representation. Hence I will present the encoded/decode algorithms 7 8.

Algorithm 7: Encode Algorithm

Input : Number $x \in \mathbb{Z}$
Number $y \in \mathbb{Z}$

Termination : The algorithm terminates when encoding is complete.

```
$B_x \leftarrow 0^{21}$;
$B_y \leftarrow 00000$;
for $i \in [0, 20] \forall i \in \mathbb{Z}$;
 do $b_i$ is bit at location $i$ in $B_x$
    $b_{20-i} \leftarrow x \mod 2$;
    $x \leftarrow x/2$;
end
for $i \in [0, 4] \forall i \in \mathbb{Z}$;
 do $b_i$ is bit at location $i$ in $B_y$
    $b_{20-i} \leftarrow x \mod 3$;
    $x \leftarrow x/3$;
end
return $(B_x, B_y)$
```

I begin with my implementation with the routine 1 presented in Figure 2 and would like to discuss each of the implementations. For initialisation, I took the the center of the search space that is at $(1048576, 121)$ and picked μ at variance around this value of 8 along x and 3 along y-axis. This gives us the distribution density as shown in Figure 3. The σ are set to mean 0 and variance of 1 followed by $\sigma = 0.7071 + |\sigma|$, both for individual and single σ implementation. This is because radius of circle formmed by the disrribution is $\sigma\sqrt{n}$ for n dimensions, and I need the radius to be bigger than 1 for exploration, otherwise the algorithm stagnates. For evaluating, the algorithm is copied from the provided boiler plate so it won't be discussed here.

3.1 Mutation

In mutation, there are two different implementations that I used, one is for single–σ and other one is individual–σ. We first discuss the single-σ, here each individual is represented as

$$a = ((x, y), \sigma) \tag{3}$$

Algorithm 8: Decode Algorithm

```
Input       : Bitstring of length 5 $B_x \in (0,1)^5$
              Bitstring of length 21 $B_y \in (0,1)^{21}$
Termination : The algorithm terminates when decoding is complete.
1  $X \leftarrow 0$;
2  for $i \in [0, 20] \forall i \in \mathbb{Z}$;
3   do $b_i$ is bit at location $i$ in $B_x$
4      $X \leftarrow X + b_i \times 2^i$
5  end
6  $Y \leftarrow 0$;
7  for $i \in [0, 4] \forall i \in \mathbb{Z}$;
8   do $b_i$ is bit at location $i$ in $B_y$
9      $Y \leftarrow Y + b_i \times 3^i$
10 end
11 return $(X, Y)$
```

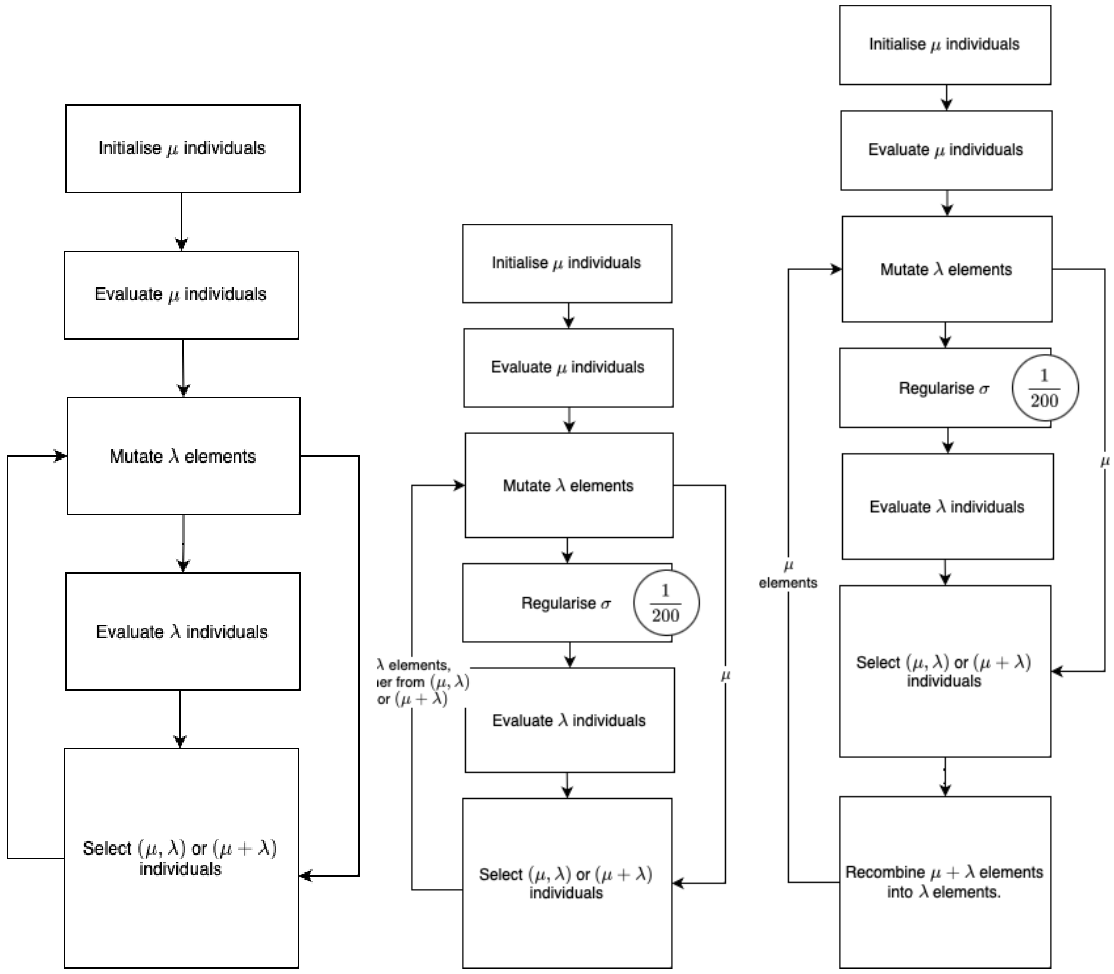

Initialise μ individuals
Evaluate μ individuals
Mutate λ elements
Evaluate λ individuals
Select (μ, λ) or $(\mu + \lambda)$ individuals
Initialise μ individuals
Evaluate μ individuals
Mutate λ elements
Regularise σ
$\frac{1}{200}$
λ elements, ier from (μ, λ) or $(\mu + \lambda)$
μ
Evaluate λ individuals
Select (μ, λ) or $(\mu + \lambda)$ individuals
Initialise μ individuals
Evaluate μ individuals
Mutate λ elements
Regularise σ
$\frac{1}{200}$
μ
μ elements
Evaluate λ individuals
Select (μ, λ) or $(\mu + \lambda)$ individuals
Recombine $\mu + \lambda$ elements into λ elements.

Figure 2: Different Routines used in the Evolutionary Strategy development

Where x is numeral representation of the connection graph, y is numeral representation of the filter permutations and σ is the step size, which we can visualise as the maximum radius of probable mutation for atleast 50% of the mutations. In individual mutation we first mutate this step size as

$$\sigma' = \sigma e^{\mathcal{N}(0, \tau_0)} \tag{4}$$

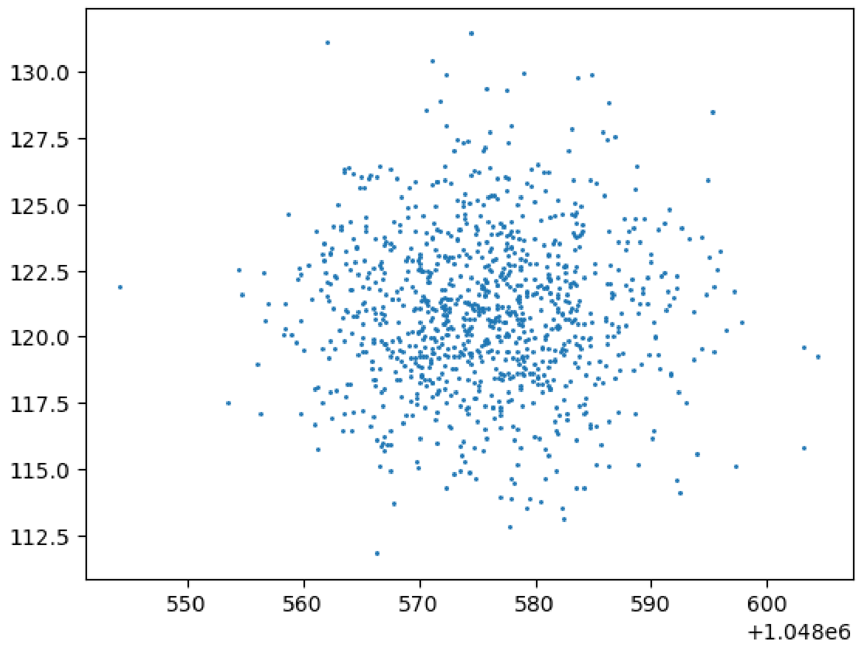


Figure 3: Initialisation Distribution Density

Here $\mathcal{N}(0, \tau_0)$ is a random number the normal distribution of mean 0 and deviation τ_0. Here we use $\tau_0 = 1/\sqrt{2}$ as suggested by Schwefel $(1/\sqrt{n})$. Then we use the so found step size to mutate the individual as shown:

$$x' = x + \mathcal{N}(0, \sigma) \tag{5.1}$$

$$y' = y + \mathcal{N}(0, \sigma) \tag{5.2}$$

However this mutation has a chance of going out of range, hence we have to implement a margin, that works as in equation 6.

$$x'' = \begin{cases} 41943 & x < 0 \\ x' & 0 \leq x < 2097152 \\ 2055208 & x \geq 2097152 \end{cases} \tag{6.1}$$

$$y'' = \begin{cases} 4 & x < 0 \\ y' & 0 \leq x < 243 \\ 238 & x \geq 243 \end{cases} \tag{6.2}$$

This gives us the margin mechanism, that pushes back the mutations into the range and with big enough sigma with global intermediate recombination encourages the population to stay within the bounds. This is because the "centre of mass" of λ population with be oriented a little inwards always. As for the *individual–σ* implementation, we have a few changes in equation 3,4, and 5. These changes are shown in equation 7,8, and 9.

$$a = ((x, y), (\sigma_x, \sigma_y)) \tag{7}$$

$$\sigma'_x = \sigma_x e^{\mathcal{N}(0,\tau)+\mathcal{N}(0,\tau')} \tag{8.1}$$

$$\sigma'_y = \sigma_y e^{\mathcal{N}(0,\tau)+\mathcal{N}(0,\tau')} \tag{8.2}$$

$$i' = i + \mathcal{N}(0, \sigma_i[i \in x, y] \tag{9}$$

This mutation also goes through equation 6 since it can also go out of range. Hence we get the algorithms 9, 10.

Algorithm 9: Single σ Mutation Algorithm

```
Input       : Individual $(x, y)$ in range $[[0, 2097152], [0, 242]]$
              $\sigma$ step-size of individual
              Offspring population size $\lambda$
              Mutation rate $\tau_0$
Termination : The algorithm terminates on successful mutation into $\lambda$ elements.
1  $\lambda_{pop} \leftarrow \phi$;
2  $\sigma_\lambda \leftarrow \phi$;
3  for $i \in [0, \lambda]$ do
4      $\sigma_i \leftarrow \sigma e^{\mathcal{N}(0,\tau_0)}$;
5      $\sigma_\lambda \leftarrow \sigma_\lambda \cup \sigma_i$;
6      $x_i \leftarrow \mathcal{N}(x, \sigma_i)$;
7      $y_i \leftarrow \mathcal{N}(y, \sigma_i)$;
8      if $x_i < 0$ then
9          $x_i \leftarrow 41943$
10     end
11     else if $x_i > 2097151$ then
12         $x_i \leftarrow 2055208$
13     end
14     if $y_i < 0$ then
15         $y_i \leftarrow 4$;
16     end
17     else if $x_i > 242$ then
18         $x_i \leftarrow 238$;
19     end
20     $\lambda_{pop} \leftarrow \lambda_{pop} \cup (x_i, y_i)$
21 end
22 return $(\lambda_{pop}, \sigma_\lambda)$
```

3.2 Selection

For selection algorithm, I considered implementing an elitist stratagy, but it seemed as a bad idea as it has really high selection pressure. The selection pressure was equal to

$$\tau^* = \frac{\ln \lambda}{\ln(\lambda/\mu)}$$

Hence I went back to trustee roulette slection algorithm as shown in 3.

3.3 Problem and Regularization

This algorithm however tends to have a bit of problem, it tends to to stop searching and stagnate at various points for a really long time, I first implemented this algorithm as shown in the first of Figures in 2. Looking at the output, it seems like the σ in both individual and single–σ implement is getting really low, and since the contour of the problem can be seen as a stair-case with step size (1×1), if σ gets too small, it can get suck at that place. You can visualise it as a small ball moving around on the step in stair case with the steps much bigger than the ball.

To counter this problem, since I observed that the algorithms stagnates at around 200 iterations, I decided to run a regularizer that refreshes the sigma of the algorithm to the initial state. This regularizer can be given as:

$$\sigma = |\mathcal{N}(0, 4)| + 0.7071 \tag{I}$$

I just chose the standard daviation of 4 on random, as it will give a radius of about 6.6. Which seems to be a good trade-off of good exploration and avoiding valling off the edge as discussed earlier.

Algorithm 10: Individual σ Mutation Algorithm

```
Input       : Individual $(x, y)$ in range $[[0, 2097152], [0, 242]]$
              $(\sigma_x, \sigma_y)$ step-size of individual
              Offspring population size $\lambda$
              Mutation rate $\tau$ and $\tau'$
Termination : The algorithm terminates on successful mutation into $\lambda$ elements.
$\lambda_{pop} \leftarrow \phi$;
$\sigma_\lambda \leftarrow \phi$;
for $i \in [0, \lambda]$ do
    $\alpha \leftarrow \mathcal{N}(0, \tau)$;
    $\sigma_{i,x} \leftarrow \sigma_x e^{\alpha + \mathcal{N}(0, \tau')}$;
    $\sigma_{i,y} \leftarrow \sigma_y e^{\alpha + \mathcal{N}(0, \tau')}$;
    $\sigma_\lambda \leftarrow \sigma_\lambda \cup (\sigma_{i,x}, \sigma_{i,y})$;
    $x_i \leftarrow \mathcal{N}(x, \sigma_{i,x})$;
    $y_i \leftarrow \mathcal{N}(y, \sigma_{i,y})$;
    if $x_i < 0$ then
        $x_i \leftarrow 41943$
    end
    else if $x_i > 2097151$ then
        $x_i \leftarrow 2055208$
    end
    if $y_i < 0$ then
        $y_i \leftarrow 4$;
    end
    else if $x_i > 242$ then
        $x_i \leftarrow 238$;
    end
    $\lambda_{pop} \leftarrow \lambda_{pop} \cup (x_i, y_i)$
end
return $(\lambda_{pop}, \sigma_\lambda)$
```

3.4 Recombination

In recombination we are using global intermediate recombination, which is basically the average of all the individual as shown in equation 10.

$$x = \frac{1}{\lambda} \sum_{i=1}^{\lambda} x_i \tag{10.1}$$

$$y = \frac{1}{\lambda} \sum_{i=1}^{\lambda} y_i \tag{10.2}$$

As for the σ values, I decided to let the sigma also get averaged, but this doesn't seem like a good idea. I though of implementing a weighted average, that tries to set the σ values in the range of $[1, 10]$, but was late to implement it. Doing this would have allowed me to get rid of regularizer allowing for really accurate values. So the final algorithm for recombination is given by 11. Once this algorithm was implemented it I created the over arching evolutionary strategy as shown in the third from left in Figure 2. It stopped being completely random as in archetecture 2, but it still has the slowing down issue. I think this is because now that we are taking the average, we need to do some tuning of variance in equation (0).

4 Experimental Results

4.1 Genetic Algorithm

For the genetic algorithm, to get a baseline behaviour, we start with the following parameters:

- Mutation Rate = $1/32$

Algorithm 11: Recombination Algorithm

```
Input       : Population $[(x, y)]$ each in range $[[0, 2097152], [0, 242]]$
              $[(\sigma_x, \sigma_y)]$ step-sizes (Individual–$\sigma$) or,
              $[\sigma]$ step-sizes (One–$\sigma$)
Termination : The algorithm terminates on successful global intermediate recombination.
1  $\mu_{pop} \leftarrow \phi$;
2  $\sigma_\mu \leftarrow \phi$;
3  IndividualFlag $\leftarrow true$;
4  $X \leftarrow 0$;
5  $Y \leftarrow 0$;
6  $\sigma_X \leftarrow 0$;
7  if (Dimensionality of $\sigma$) $= 1$ then
8      IndividualFlag $\leftarrow false$;
9      $\sigma_Y \leftarrow 0$;
10 end
11 for $(x_i, y_i) \in [(x, y)]$ do
12     $X \leftarrow X + x_i$;
13     $Y \leftarrow Y + y_i$;
14 end
15 $X \leftarrow X/count([(x, y)])$;
16 $Y \leftarrow Y/count([(x, y)])$;
17 if IndividualFlag then
18     for $(\sigma_{x,i}, \sigma_{y,i}) \in [(\sigma_x, \sigma_y)]$ do
19         $\sigma_X \leftarrow \sigma_X + \sigma_{x,i}$;
20         $\sigma_Y \leftarrow \sigma_Y + \sigma_{y,i}$;
21     end
22     return $(\mu_{pop}, [\sigma_X, \sigma_Y])$
23 end
24 else
25     for $(\sigma_{x,i}, \sigma_{y,i}) \in [(\sigma_x, \sigma_y)]$ do
26         $\sigma_X \leftarrow \sigma_X + \sigma_{x,i}$;
27     end
28     return $(\mu_{pop}, \sigma_X)$
29 end
```

- Crossover Rate = $1/2$
- Crossover Style = Uniform
- Population Size = 26

Then I decided to see how the change in population size affects the performance of algorithm. I thought the larger number of population would improve the the probability of the poor fitness individual making it through the `MatingSelection` in the code. The result of larger population can go in one of two directions. Due to less fit individuals surviving for more generation, the progression can be slow, but it can also reduce the probability of the search settling in a local minima. Due to computation time[1] required, I decided to limit the population size limited to $\mu = \{21, 26, 31, 100\}$. Looking at their expected runtime in Figure 4, we can see clearly that the larger population has lower expected runtime. The final ERT at most accurate point of each algorithm is shown in table 2. For reference the random search algorithm provided in the project file scored 47042@0.94.

4.1.1 Crossover Experiment

Since I have made a very modular Genetic Algorithms with a lot of tunable parameters, I decided to play aroud a bit with these parameters. Since we are working on finding a connection graph in the neural archetecture, I thought the

[1] I tried to multi thread the algorithm 2, with different data structures to ensure data is always locally available, but it just yielded a lift in performance by just 3% even on Apple M1 (it has 12MB cache.) So I reverted it back to single threaded.

Table 2: Expected Runtime Table

	Expected Runtime at best fit		Best f Evaluation	
μ, λ	(μ, λ)	$(\mu + \lambda)$	(μ, λ)	$(\mu + \lambda)$
21	47930.5	243707.42	0.9505	0.9505
26	22894	347951.15	0.9471	0.9505
31	31395.67	234799	0.9505	0.9505
100	30142.33	98787.15	0.9505	0.9505

Table 3: Area Under Curve

	Target f is 0.94	
μ, λ	(μ, λ)	$(\mu + \lambda)$
21	0.999959483643959	0.990305046681312
26	0.99994085045348	0.991325464851718
31	0.999944533758575	0.990305046681312
100	0.999974866859354	0.989794837596109

parameter that is going to have the most significant affect on the performance is the number crossover points. I plotted the area under the curve keeping all but crossover parameters locked to following:

- Mutation Rate = $1/32$
- Crossover Rate = $1/2$
- Crossover Style = Uniform or n-point $\forall n \in [1, 10], n \in \mathbb{Z}$
- Population Size = 31.

On plotting the ECDF graph for a target of 0.95 we get the area under the curve table 4. We can see that for (μ, λ) GA, we get better performance in $2, 7$ and 10-point crossover compared to uniform crossover. The 7-point crossover is berforming best, maybe because we are working on a 7×7 connection matrix. But since this is also not true in $(\mu + \lambda)$ algorithm plots, I am not so sure, could be by chance that it is working best.[2]

Table 4: Area Under Curve

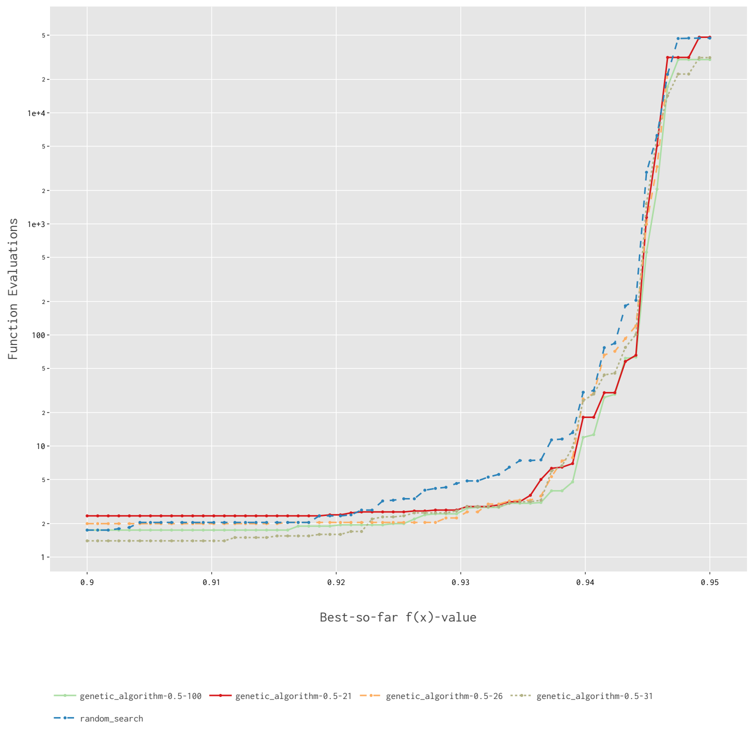
	Target f is 0.95	
Crossover points	(μ, λ)	$(\mu + \lambda)$
Uniform	0.612841760979622	0.724487495191779
1	0.517003287686308	0.428571091260029
2	0.678567191229168*	0.431971404824706
3	0.440473160878826	0.416510828799868
4	0.535710601068842	0.535249331220214
5	0.555389555454896	0.466603511936876
6	0.518549353748919	0.65101882761842
7	0.689233921569486*	0.679589790086823
8	0.435856955185868	0.472301889041967
9	0.608839267905166	0.72448798889301*
10	0.660345698348395*	0.762388558446376*

4.2 Evolutionary Strategy

For the evolutionary strategy, I started to build the program with architectures in Figure 2 and would like to discuss their performances here. For comparison purposes I used the following configuration on all of the as:

- $\mu = 1$

[2] Since plotting these graphs took really long time, I will include the IOHAnalyser data with the project.



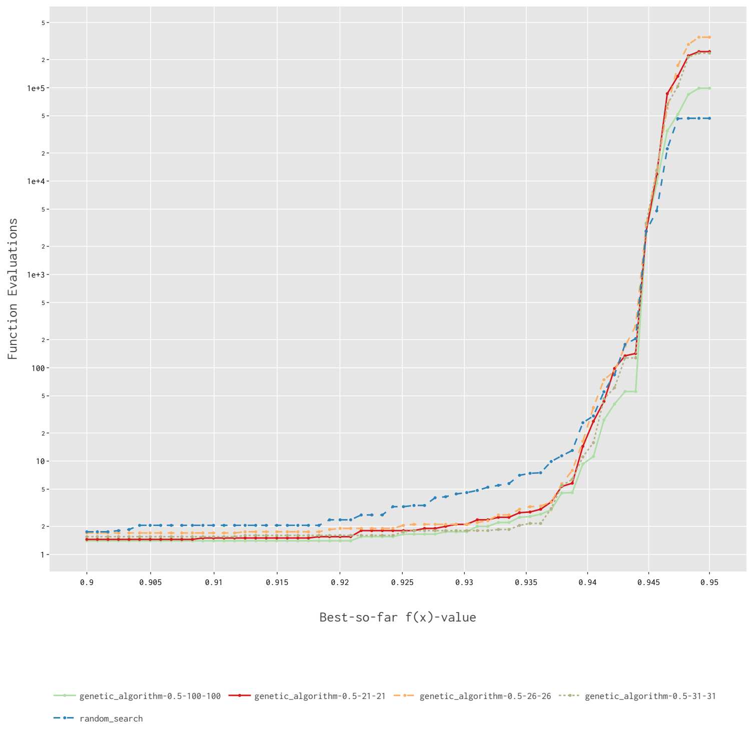
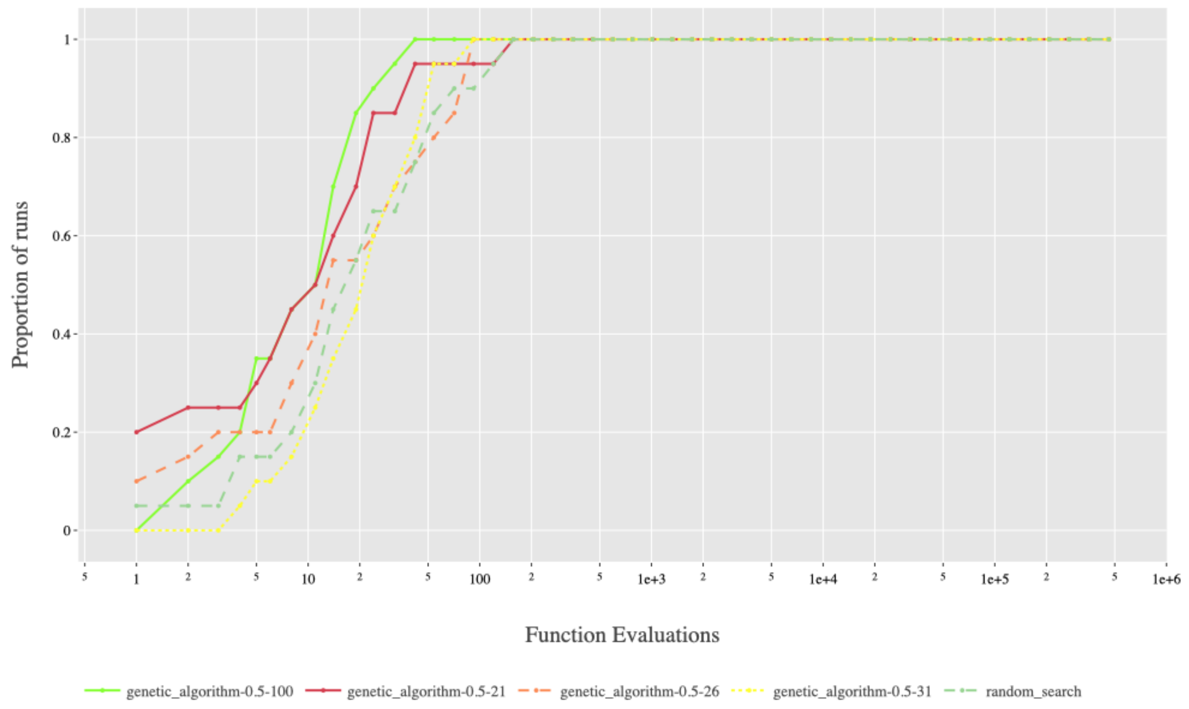


Figure 4: Expected runtime of aforementioned algorithms with different population size **top** (μ, λ) **bottom** $(\mu + \lambda)$.

- $\lambda = 32$
- $\tau_0 = 0.7071$
- Individual–σ



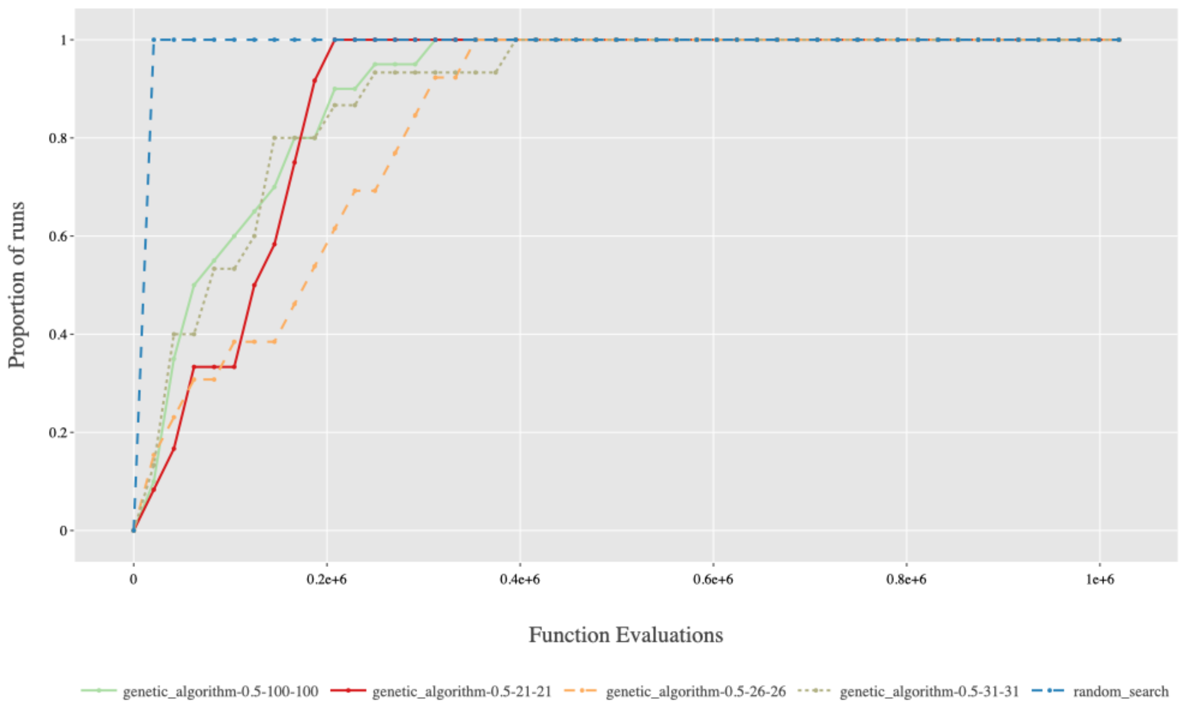
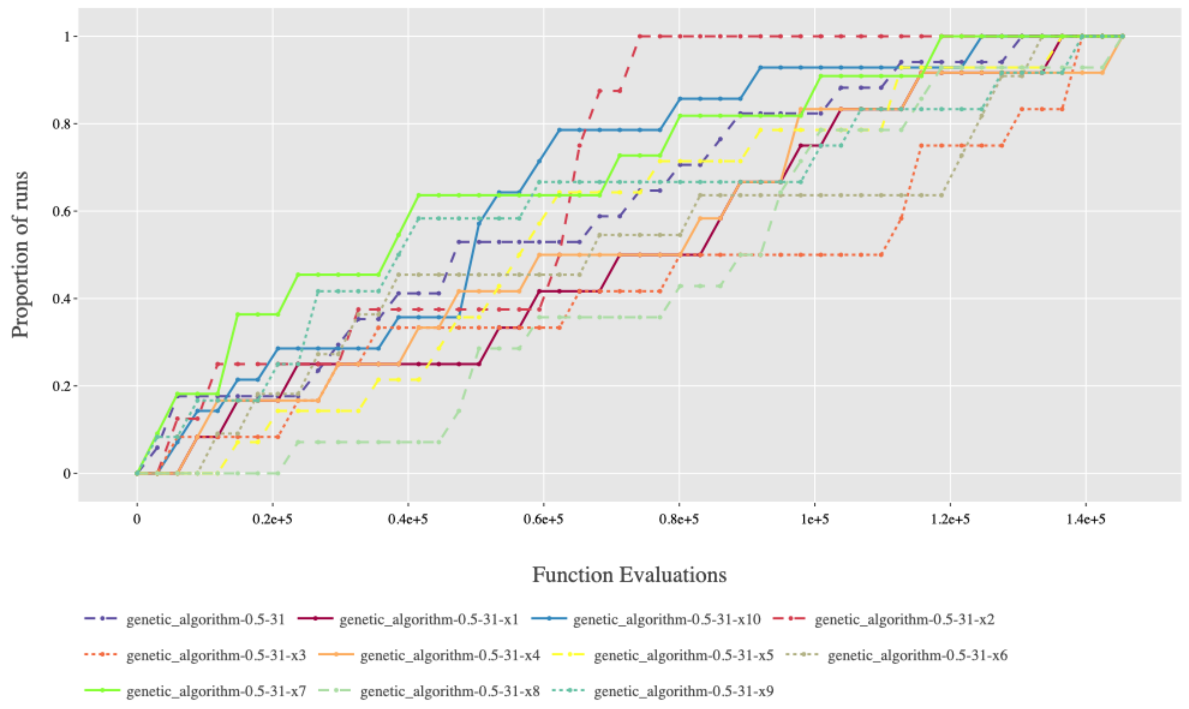


Figure 5: The ECDF curve for the (μ, λ) (top) and $(\mu + \lambda)$ (bottom) GA, for different population sizes.

They all perform in a very stpper way, with most of the progress made when the sigmas are regularised. As we can see that ever since introducing the Regularisation, the area under the curve for all (μ, λ) is better than $(\mu + \lambda)$ and by a long shot. This holds up with the theory we studied in class, as when you remove parent from equation of selection, we are less likely ignore the improvement in σ, causing an improvement in the performance as shown in table 5. So now I



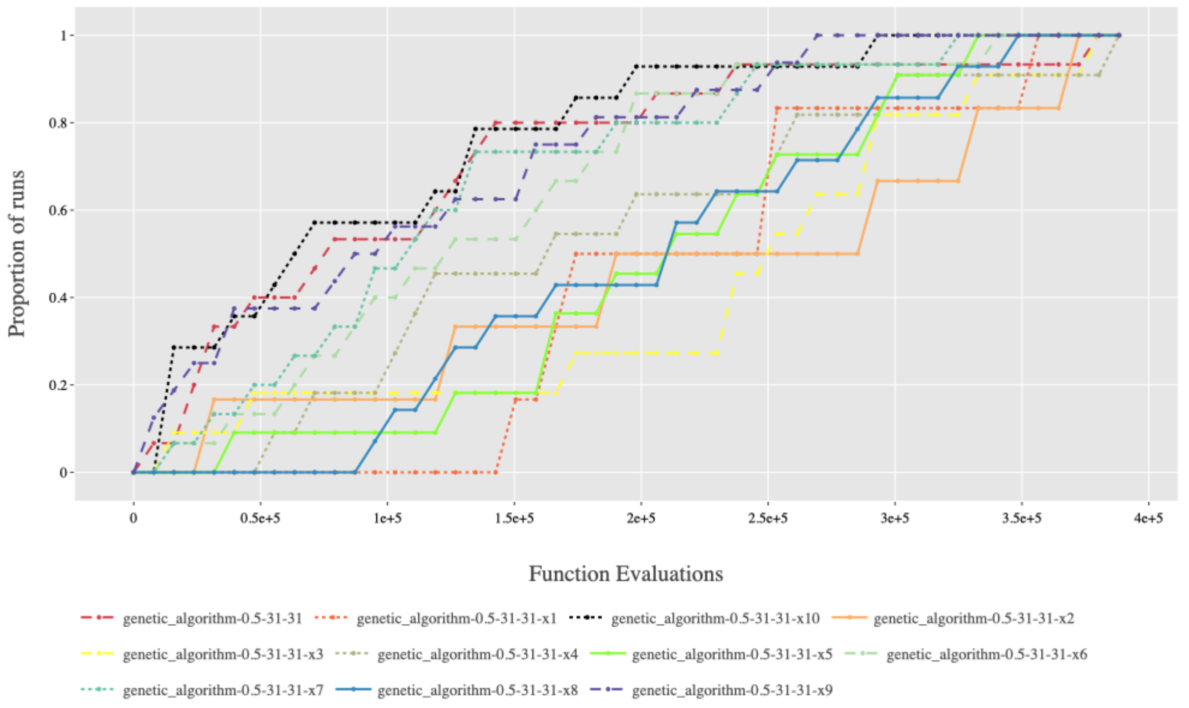


Figure 6: ECDF Curves for Crossover performance comparison, (μ, λ) on top, and $(\mu + \lambda)$ at the bottom. The f_{traget} was set to be 0.95 in both cases.

will focus on trying to optimise the regularizer function to get best performance from archetecture 3. I tried to do two different things,

1. Add multiplier to increase the equation I. This is to tune the size of the *ball* discussed in earlier visulaisation.

Table 5: Area Under Curve for archetectures in Figure 2

	Target f is 0.943	
Archetecture	(μ, λ)	$(\mu + \lambda)$
1	0.859404674513599	0.968307317632152
2	0.975503641232263	0.973462799369354
3	0.928564278385345	0.792511282397791

2. Add multiplier to the number of selection. This is because we want more individuals with better fitness appearing multiple times. Sinve we are using global intermediate recombination, larger population with larger number of fit individuals can pull the λ towards those fitter individuals.

It seems that the second modification performs much better, I ran architecture 3 on the aforementioned modification The area under the curve for the normal one is 0.928564278385345, the first modification is 0.831895214034185 for $2\times$ regularizer; 0.938768929042215 for $4\times$ regularizer and $1.6\times$ selections of next generation. So for all further experiments, I will be using $2 \times (\mu, \lambda)$ selection. [3]

Now we discuss the performance of the algorithm for different λ, and discuss their performance. The expected runtime of the the EA are shown in Figure 7, and the corresponding ECDF plots are in Figure 8.

The area under the curve are as show in table 6, as we can see the $(1, 2)$ performs the best, and the this is due to our recombination function choice. We will discuss how we can remedy this stagnation further in the section 5. Another interesting thing is as the population grows, the lead by (μ, λ) over $(\mu + \lambda)$ grows much larger.

Table 6: Area Under Curve

	Target f is 0.94	
λ	$(1, \lambda)$	$(1 + \lambda)$
2	0.989795190296269	0.987751412029136
4	0.989795190296269	0.98877179226266
8	0.989795190296269	0.974486307986579
16	0.988661412641013	0.965982763691477
32	0.970662514999919	0.938091833952673
64	0.989795190296269	0.960313984616342
128	0.981291790314647	DNF
256	0.926019715771807	DNF

5 Discussion and Conclusion

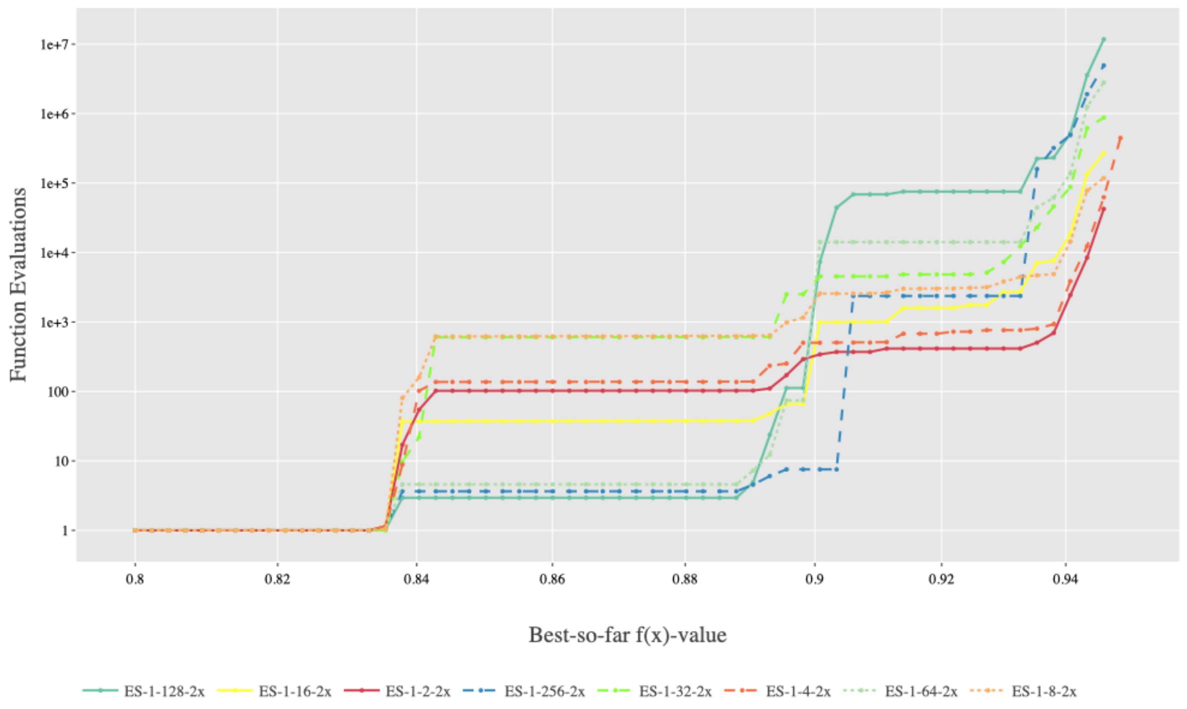
It seems that the Genetic Algorithm is the better fit for this problem, since it is a mostly a bit string search problem, a perfect fit for GA, while trying to fit this problem into Evolutionary Strategy takes a lot of experiemntation and visulaisation. While it provides a nice terrain on 2-d search space, the evolutionary strategy also needs a few hacks to atleast work properly as discussed in both sections Evolutionary Strategy and Experimental Results.

While we are comparing the two, it is also interesting to note that for Genetic Algorithm $(\mu + \lambda)$ implementations generally ouperformed their (μ, λ) while it was completely opposite in case of evolutionary strategy. Also, the Evolutionary Strategy has a stagnation problem while the Genetic Algorithm has not encountered such problems.

Future Work

I did not implement a multi μ recombination method, because I wasn't sure what to do with the sigma values, it will be really interestng to see how multi μ ES will work when they are initialised wit really large variance. In a normal problem with gradient and not many invalid potential individuals, these inital parents may diverge or converge depending on the gradient.

[3]The selection process can be tuned further, but due to time constraints, I chose $2\times$.



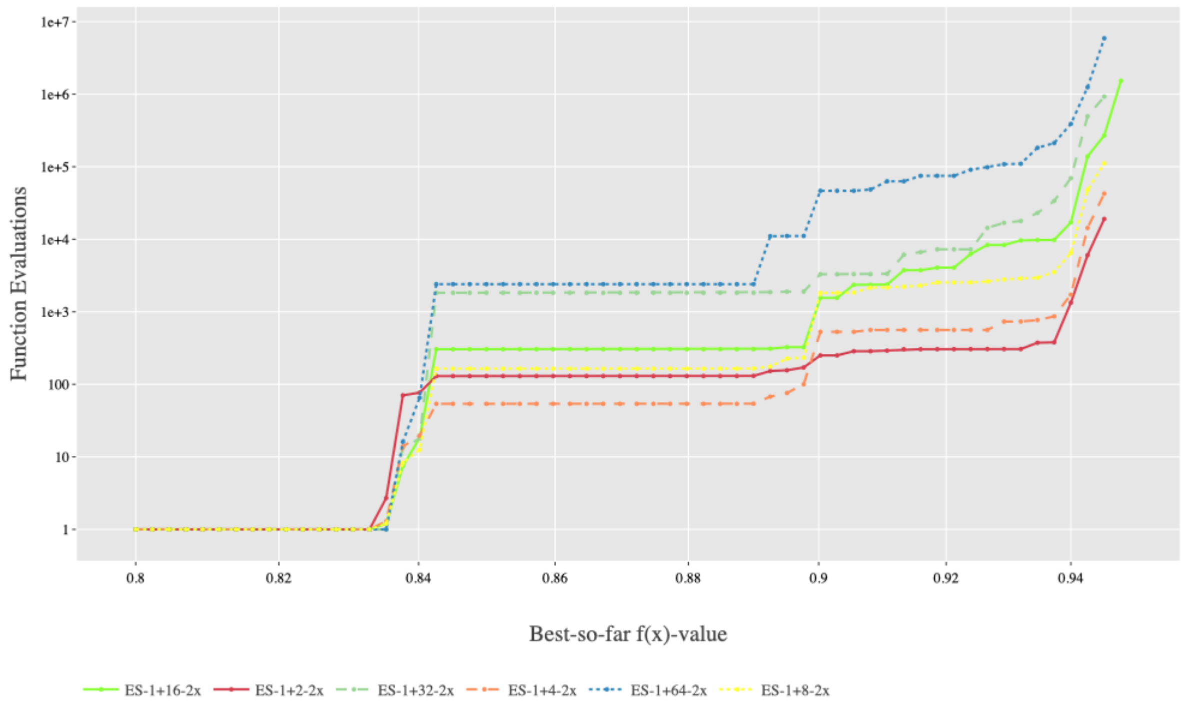
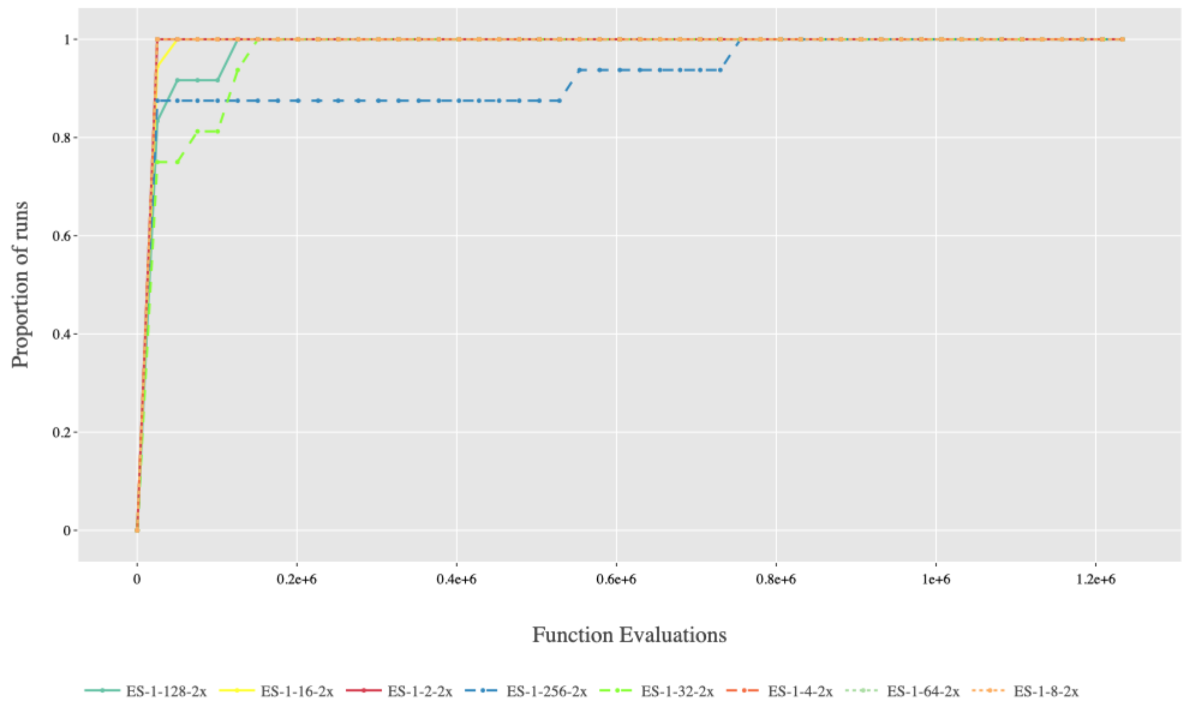


Figure 7: Expected runtime for, (μ, λ) on top, and $(\mu + \lambda)$ at the bottom. The f_{traget} was set to be 0.95 in both cases.

For future work, I'd like to see what happens when the one implements a weighted recombination. I regrettable ran out of time but the idea of this implementation can be seen as in equation 11.

$$x = \frac{1}{\lambda \sum f_i} \sum_{i=1}^{\lambda} x_i \times f_i \tag{11.1}$$



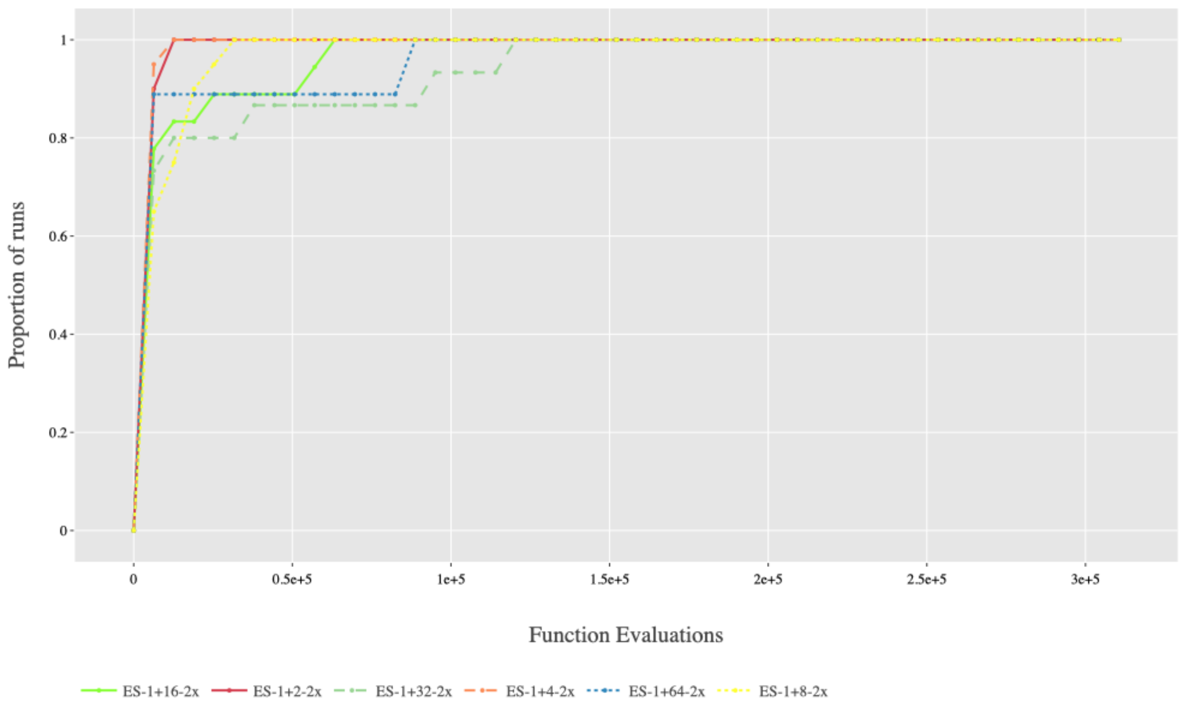


Figure 8: ECDF plots for (μ, λ) on top, and $(\mu + \lambda)$ at the bottom. The f_{traget} was set to be 0.94 in both cases.

$$y = \frac{1}{\lambda \sum f_i} \sum_{i=1}^{\lambda} y_i \times f_i \tag{11.2}$$

This will allow us to drop selecting some n–times more than $\mu + \lambda$ individuals in Selection process [3], hence saving us precious compute time.

There is also a way to get Genetic Algorithm, not sure if it will be objectively better performing, but I'd like to see what happens if I added a new "do nothing" or $\mathbb{I}$ filter, so that 00 in 1 is $\mathbb{I}$. Since it removes the last random element the algorithm, it will be interesting to see how the performance changes.

It would be also really interesting if we are not using $\mathbb{I}$ what will happen to setup the genetic algorithm of adjacency matrix part and Evolutionary algorithm on permuting the filter. Then setting a feedback loop on both side. I do not expect better performance since neither of these problems are gradient problems where evolutionary strategy shines, but it's just purely out of interest.