Optimizing Neural Architecture Search using Evolutionary Algorithms

Mireia Planas
s3686140
`m.planas.lisbona@umail.leidenuniv.nl`
& *FilipoBonni*
s3628590
`f.boni@umail.leidenuniv.nl`

December 7, 2022

1 Introduction

In this assignment, we address the problem of Neural Architecture Search (NAS) through two different types of evolutionary algorithms: *Genetic Algorithm* and *Evolutionary Strategy.* The NAS technique aims to learn and construct a well-performing neural network architecture automatically. In this task, we will work with the NAS-Bench-101 API, which provides a dataset that maps neural architectures to their training and evaluation metrics [1].

In the NAS-bench-101 framework, the search for neural network topologies is restricted to small, feedforward structures called cells. The architectures considered are constructed following the scheme in Figure 1. The outer skeleton of the architecture stacks each cell three times, followed by a downsampling layer in which the height and width of the image are halved via max-pooling, and the channel count is doubled. The pattern is repeated three times, followed by a global average pooling and a final dense softmax layer. The initial layer of the model is a stem consisting of a 3×3 convolution with 128 output channels [1].

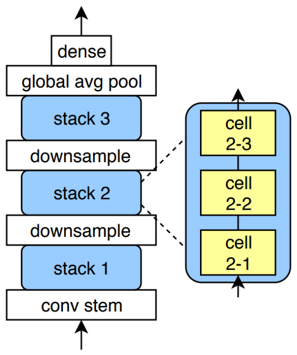

Figure 1: Outer skeleton of a NAS-bench-101 model [1].

The step where meta-learning occurs is the design of the cells. The search space consists of all possible directed acyclic graphs on V nodes, with each node having one of L labels corresponding to an operation. Two of the vertices are already assigned with the labels IN and OUT, representing the input and output nodes of the cell. This setup presents the following restrictions:

- The possible operations in a vertex are: 3×3 convolution, 1×1 convolution and 3×3 max-pool. Hence, we set $L = 3$.
- The maximum amount of vertices is 7, i.e. $V \leq 7$.
- The maximum amount of edges is 9.

In order to encode the solution to our problem, we will represent the cell's graph by its adjacency matrix and the labels of the nodes by a list. This is:

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

$$\text{Operations at the vertices } = [\text{INPUT}, x_{22}, x_{23}, x_{24}, x_{25}, x_{26}, \text{OUTPUT}]$$

Next, we can group all the variables together on a 26 dimensional vector $\mathbf{x}$, with $x_i \in \{0, 1\},\ i \in 1, ..., 21$ and $x_i \in \{0, 1, 2\},\ i \in 22, ..., 26$. With this setup we can proceed to apply the evolutionary algorithms that we will describe in the following section.

2 Algorithms

2.1 Genetic algorithm

The Genetic Algorithm approach allows us to solve discrete optimization problems based on natural selection, the process that drives biological evolution. The genetic algorithm operates on solutions to a problem specified through bitstrings. At each step the algorithm selects individuals from the current population to be parents and uses them to produce children for the next generation. When creating a new generation, the children undergo a mutation operation to generate variations that can potentially lead to improvements. An overview of the basic setting for a genetic algorithm is given in Algorithm 1

2.1.1 Genetic Algorithms theory

Algorithm 1: A framework of Genetic Algorithm

```
Input  : Population size $\mu$
         Mutation rate $p_m$
         Crossover probability $p_c$
         Number of offsprings generated $\lambda$
         Budget $B = 5000$
Output : Best found individual $x^*$
         Best found fitness $f(x^*)$
1  $t \leftarrow 0$
2  Initialize$(P(t))$ through random sampling
3  Evaluate$(P(t)); B = B - \mu;$
4  $f^* = \max_{x \in P(t)} f(x)$
5  $x^* = \operatorname{argmax}_{x \in P(t)} f(x)$
6  while $B > 0$ do
7      $P'(t) \leftarrow$ Mating selection$(P(t), \lambda)$
8      $P''(t) \leftarrow$ Crossover$(P'(t), p_c)$
9      $P'''(t) \leftarrow$ Mutate$(P''(t), p_m)$
10     $P''''(t) \leftarrow$ Filter out non valid$(P'''(t))$
11     $P(t+1) \leftarrow$ Environmental selection$(P''''(t))$
12     $f^*_{t+1} = \max_{x \in P(t+1)} f(x)$
13     $x^*_{t+1} = \operatorname{argmax}_{x \in P(t+1)} f(x)$
14     if $f^*_{t+1} > f$ then
15         $f^* = f^*_{t+1}$
16         $x^* = x^*_{t+1}$
17     $t \leftarrow t + 1$
18 end
19 return $x^*, f^*$
```

Mating selection

The mating selection is the function in charge of selecting the parents that will be involved in giving birth to the new generation. The parameter offspring size (λ) will control how many parents will be selected and will also decide the size of the offspring. In general, genetic algorithms don't use environmental selection, and they set $\mu = \lambda$. However, we considered lifting this restriction in order to give more flexibility to the algorithm. We have implemented a proportional selection mechanism in which the probability of a parent being chosen is proportional to its "shifted" fitness. In mathematical terms, we can express the method as follows:

$$f_i' = f_i - \min_{i \in 1 \ldots \mu} f_i$$

$$p_i = \frac{f_i'}{\sum_{i=1}^{\mu} f_i'}$$

Through shifting the fitness values, we overcome the problem of negative probabilities, and we also ensure that we give a higher probability of being chosen to individuals with high fitness values, even if all fitness values are similar.

Crossover

The crossover method consists of combining two parents to generate two children that combine parts from both parents. The application of crossover is conditioned by the crossover probability p_c. At each iteration, we will decide whether or not to apply it with probability p_c. There are three main approaches to implementing the crossover:

- **1-point crossover:**

  We will choose a random index of the bitstring and split the parents at that crossover point. Next, we will create two children by exchanging the parents' tails.

- **n-point crossover:** We will choose n random crossover indices and split the parents at those points. Finally, we create the children by alternating from parent to parent at each crossover index.

- **Uniform crossover:** For each index $i \in \{1, ..., l\}$, we will decide with uniform probability if child 1 receives the bit from parent 1 and child 2 receives the bit from parent 2 or the other way around.

In Figure 2 we present a diagram exemplifying the three types of crossover.

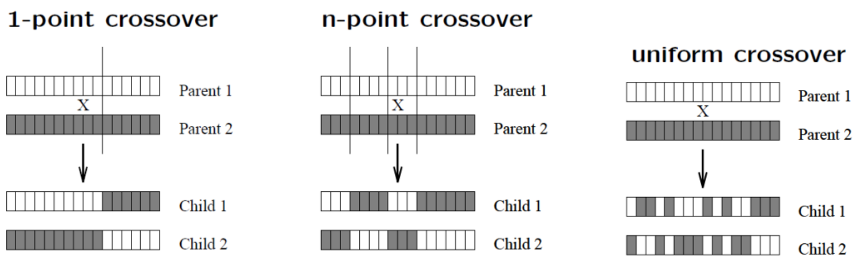

Figure 2: Crossover methods graphical representation.

We have decided to implement the three types of crossover and experiment with them in the hyperparameter random search in order to see which one gives the best results.

Mutation

The mutation operator consists of altering each gene with a probability given by p_m. The values that make sense for the mutation probability are $1/l \leq p_m \leq 1/2$. This is because we want the mutation to alter at least one bit and, at most half of the individual on average. A mutation rate of $p_m = 1/2$ corresponds to a random generation of the offspring. In the approach that we propose, we use a decreasing mutation rate schedule in which at each iteration we multiply the current mutation by 0.98. In this way, we can perform more exploration at the early stages of the optimization and more exploitation at the end of the process.

Environmental selection

Since we have introduced the possibility of having offspring of a different size than the parents' population, we also need to include environmental selection in our pipeline. This method selects the next generation given the previous population and its offspring. The two canonical methods to perform environmental selection are the following:

- $(\mu + \lambda)$ **Selection**

  After the parents' population of size, μ has generated an offspring of size λ, we will consider all the individuals together, and we will choose the ones with better fitness in a deterministic way. This method guarantees monotony, i.e., the fitness of the individuals will only increase with the generations.

- (μ, λ) **Selection**

  In this approach, the offspring will be considered alone, and its best individuals will form the next generation. In this case, deteriorations are possible since the best fitness value of a generation could be worse than the best in the previous one. In our problem, since some of the offspring individuals can be invalid solutions to the NAS problem, we will complete the next generation with individuals from the parents' population in case we don't have enough valid children to fill the population size.

2.1.2 Experimental setup

Encoding/decoding

In canonical Genetic Algorithms, the individuals are represented by bitstrings $\mathbf{a} \in \{0,1\}^l$. We will call the space of bitstrings over which the algorithm operates the **genotype space**. In general, in discrete problems like the one we are facing, the solutions are not initially encoded as bitstrings. Instead, each problem will present a specific space of solutions that we will call the **phenotype space**. In our case, a phenotypical solution is represented by a 26 dimensional discrete vector $\mathbf{x}$. In particular, we have that $x_i \in \{0,1\},\ i \in [1...21]$ and $x_i \in \{0,1,2\},\ i \in [22...26]$.

Before applying the genetic algorithm, we need to introduce an encoding/decoding scheme to convert phenotype representations to genotype individuals. The main issue we need to address is how to encode the information of the last 5 elements of $\mathbf{x}$ into a bitstring. To do so, we will interpret these last five elements as a 5 digit number in base 3. In order to convert them into a bitstring, we will just need to express such a number in base 2. We performed the conversion by going from base 3 to base 10 and then from base 10 to base 2. When converting a 5 digit number in base 3 to base 10, the biggest number we can obtain is $(22222)_3 = (242)_{10}$. In order to be able to represent these numbers in base 2, we need at least 8 bits. Indeed, the highest number that we can represent with a bitstring of length 8 is $(256)_{10}$. Hence, all numbers of 5 digits in base 3 can be represented with a bitstring of length 8, but not the other way around. In Section 2.1.2 we will explain how we solve this problem. When given a phenotype, the encoding procedure will consist of converting the 26 dimensional vector $\mathbf{x}$ to a 29 dimensional bitstring $\mathbf{y}$ where the first 21 elements of $\mathbf{y}$ will be the same as in $\mathbf{x}$ and the last 8 bits will be the binary representation of the 5 last digits of $\mathbf{x}$.

Population initalization

The first step of the genetic algorithm is to produce an initial population in the genotype space. However, in order to not waste the budget, we want to make sure that the individuals generated are valid phenotypes for the NAS problem. To do so, we built a random sampling method that generates μ bitstrings of dimension 29, all of them fulfilling the following constraints:

- **Existance of a corresponding phenotype:**

  As we said in the previous section, all numbers of 5 digits in base 3 can be represented with a bitstring of length 8, but not the other way around. For that reason, after generating a random bitstring of size 29, we check that the last 8 digits are the binary representation of a number smaller than $(243)_{10}$. In other words, the last 8 bits should represent a number with 5 or fewer digits in base 3.

- **Validity of the phenotype:**

  Due to the complexity of the problem we are dealing with, the phenotype individuals will not always represent valid neural network architectures. To avoid wasting resources on invalid solutions, we will check that the phenotypes corresponding to the randomly generated genotypes represent a valid solution to the NAS problem. A function checking this validity is already implemented in the provided code so we will make use of it.

In order to make sure we have the right amount of valid individuals, we will generate the random genotypes one by one and only add them to the population if they fulfill the conditions described above. We will stop generating more individuals once we have generated μ valid genotypes.

Filtering out invalid individuals

The initialization of the population is not the only step where invalid phenotypes could be generated. After generating an offspring population (i.e. after applying mating selection, crossover, and mutation), we may end up with invalid individuals that need to be filtered out before being evaluated in order to save budget resources. For that reason, we check the validity of the generated offspring according to the two requirements described above. Finally, we evaluate the filtered offspring population and feed it to the environmental selection method.

Genetic Algorithm parameters overview

Here we present an overview of the possible values of the parameters that were considered to build the algorithm:

1. l: dimension of the genotypes with $l = 29$.
2. μ: population size with $\mu \in \{50, ..., 1000\}$.
3. λ: number of offsprings generated by the recombination with $\lambda \in [50, 1000]$.
4. p_m: mutation rate with $p_m \in [1/l, 1/2]$.
5. p_c: Crossover probability with $p_c \in [0, 1]$.
6. *Crossover method*: uniform, one-point, n-points.
7. *Crossover points* $\in \{2, ..., 6\}$: only used if we selected n-points crossover.

2.2 Evolutionary Strategy

2.2.1 Evolutionary Strategy Theory

The Evolutionary Strategy allows solving optimization problems by representing the possible solutions using d dimensional arrays called individuals. At each step, the group of individuals called the population undergoes a series of operations called recombination, mutation, and environmental selections which have the goal of generating new and possibly better solutions to the problem. The proposed solutions represented by the individuals are evaluated using the $objective\ function$(or $fitness$), which is the function to be optimized.
We present the structure of the algorithm that we used in Algorithm 2. In the following sections, we will explain its components.

Algorithm 2: A framework of Evolutionary Strategy

```
Input  : Population size $\mu$
         Number of offsprings generated $\lambda$
         Number of parents used in the recombination $\beta$
         Step size $\tau$
         Step size $\tau'$
         Budget $B = 5000$
Output : Best found individual $x^*$
         Best found fitness $f(x^*)$
1  $t \leftarrow 0$
2  Initialize($P(t)$) through random sampling
3  Evaluate($P(t)$); $B = B - \mu$;
4  $f^* = \max_{x \in P(t)} f(x)$
5  $x^* = \operatorname{argmax}_{x \in P(t)} f(x)$
6  while $B > 0$ do
7      $P'(t) \leftarrow$ Recombination($P(t), \lambda, \beta$)
8      $P''(t) \leftarrow$ Mutation($P'(t), \tau, \tau'$)
9      $P'''(t) \leftarrow$ Filter out non valid($P''(t)$)
10     $P(t+1) \leftarrow$ Environmental selection($P'''(t)$)
11     $f^*_{t+1} = \max_{x \in P(t+1)} f(x)$
12     $x^*_{t+1} = \operatorname{argmax}_{x \in P(t+1)} f(x)$
13     if $f^*_{t+1} > f$ then
14         $f^* = f^*_{t+1}$
15         $x^* = x^*_{t+1}$
16     $t \leftarrow t + 1$
17 end
18 return $x^*, f^*$
```

Basics

The main characteristic of the evolutionary strategy algorithm is that it can work with a real-valued search space $\mathbb{R}^d$. Secondly, the most important feature is the self-adaptation of the strategy parameters. This means that some of the parameters with which the algorithm is initialized change accordingly to the performance evaluation.

Sigma type

There are different ways of representing an individual in the evolutionary strategy algorithm for a d dimensional problem. The representation depends on the format of the parameter σ. We used and present two of them.

- **Singular** σ: The individual **a** is composed by a d dimensional array, which represents the set of values of the individual, and by a parameter σ which will be used for the mutation.

$$a = ((x_1, ..., x_d), \sigma)$$

- **Individual** σ: The individual **a** is composed by a d dimensional array, which represents the set of values of the individual, and by a d dimensional array of σ_i values, one for each dimension, which will be used for the mutation.

$$a = ((x_1, ..., x_d), (\sigma_1, ..., \sigma_d))$$

Every component of the individuals shown is affected by mutation, recombination, and environmental selection.

Mutation

The mutation is a step of the algorithm that modifies the individual. It is based on random sampling from a normal distribution. It changes accordingly to the individual representation.

- **Singular** σ: It is characterized by a parameter τ. The steps are the following:
  1. Before mutation: $a = ((x_1, ..., x_d), \sigma)$
  2. $\sigma' = \sigma exp(N(0, \tau^2))$ with $N(0, \tau)$ being a random number sampled from a normal distribution with mean 0 and standard deviation τ.
  3. $x'_i = x_i + N(0, \sigma'^2)$ $\forall i = 1, .., d$ with $N(0, \sigma'^2)$ being a random number sampled from a normal distribution with mean 0 and standard deviation σ'.

4. After mutation: $a' = ((x'_1, ..., x'_d), \sigma')$

- **Individual** σ_i: It is characterized by the parameters τ and τ'. The steps are the following:

  1. Before mutation: $a = ((x_1, ..., x_d), (\sigma_1, ..., \sigma_d))$
  2. $\sigma'_i = \sigma_i exp(N(0, \tau'^2) + N_i(0, \tau^2))$
  3. $x'_i = x_i + N_i(0, \sigma'^2_i)\ \forall i = 1, .., d$
  4. After mutation: $a' = ((x'_1, ..., x'_d), (\sigma'_1, ..., \sigma'_d))$

Recombination

The recombination steps consist of considering 2 or more individuals (parents) of the population and combining them to create a new individual (child). In our case, the parents are selected randomly from the population. The two types of recombination considered are the following.

- **Discrete**: For each index $i \in 1, ..., d$, we decide with uniform probability if the child receives the x_i value from parent 1 or from parent 2. In Figure 3 we present a diagram that explains how this kind of recombination work.

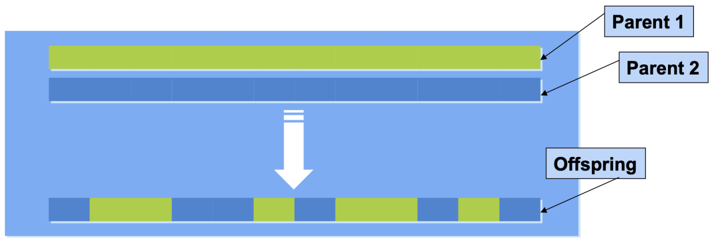

Figure 3: Discrete recombination scheme

- **Intermediate**: For each index $i \in 1, ..., d$ the value of x_i of the children is the mean of the x_i of the parents. In Figure 4 we present a diagram that explains how this kind of recombination work.

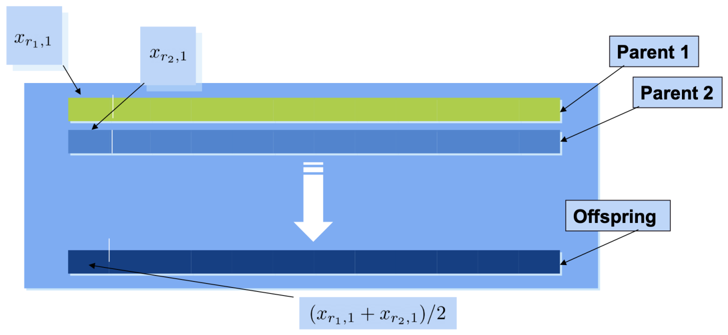

Figure 4: Intermediate recombination scheme

The recombination operation is applied to both x_i and σ_i (or σ in case of singular σ) values.

Environmental Selection The environmental selection in the evolutionary strategy algorithm applies to the individuals in the way already explained for the genetic Algorithms.

2.2.2 Evolutionary strategy Experimental Setup

Encoding/Decoding

Each individual of the population x is a 26-dimensional string with $x_i \in \{0, 1\}$ if $i \in [1..21]$ and $x_i \in \{0, 1, 2\}$ if $i \in [22..26]$ and it is evaluated in this form. This is a problem because the evolutionary strategy algorithm works with floating numbers while the elements of the offspring are integers. In order to overcome this problem we applied the following encoding and decoding strategy:

- The population is initialized to individuals $x \in \mathbb{R}^{26}$ where $x_i \in [0, 2], \forall i \in \{1, ..., 21\}$ and $x_i \in [0, 3], \forall i \in \{22, ..., 26\}$.
- After mutation, which is the only operator that can create individuals out of the specified ranges, we will redirect the resulting individuals to the desired range by performing the operations: $(x_i \textbf{ mod } 2)$ if $i \in \{1, ..., 21\}$ and $(x_i \textbf{ mod } 3)$ if $i \in \{22, ..., 26\}$.
- For calculating the fitness and the validity of the individuals we want to get back the corresponding discrete representation. For that reason, we take the floor of the individual elementwise. This will give us discrete individuals in the desired ranges.

Population Intialization

Each element x_i of the offspring x is generated by sampling from a uniform distribution. If $i \in [1..21]$ then the range of values of the uniform distribution is $[0, 2]$. If $i \in [22..26]$ then the range of values of the uniform distribution is $[0, 3]$.

The initializtion value of σ is $\frac{||x||_2}{\sqrt{d}}$ with d as dimension of the problem (26). If we work with the case of individual sigmas σ_i for each x_i the value is repeated d times in order to build a $d - dim$ array.

Filtering out invalid individuals

As explained in Section 2.1.2, among the possible offspring that can be generated some of them are not considered valid for the problem analyzed. Thus in the implementation of the algorithm, we need to check if the offspring generated are valid. We perform the check in two steps of the algorithm:

1. **Population initialization**: When a new offspring is generated we check if it is valid before adding it to the initial population. In the case an offspring is not valid it is discarded. We keep generating new offspring until we have a number of valid offspring equal to the fixed population size. Then the offspring are evaluated.
2. **After the Mutation**: With the recombination and the mutation new offspring are generated. Before evaluating them and before performing the environmental selection we check if the offsprings are valid. In the case an offspring is not valid it is discarded.

Evolution Strategy parameters overview

Here we present an overview of the possible values of the parameters that were considered to build the algorithm:

1. μ: population size with $\mu \in \{20, .., 300\}$.
2. λ: number of offspring generated by the recombination. If the *environmental selection* is 'plus' we have $\lambda \in \{20, .., 300\}$. If the *environmental selection* is 'comma' we have $\lambda = \mu + k$ with $k \in \{50, .., 100\}$.
3. *mutation*: singular σ, individual σ_i.
4. *recombination*: discrete, intermediate.
5. β: number of parents involved in the recombination. $\beta \in \{2, 3, 4, 5\}$.
6. *environmental selection*: comma, plus.
7. τ_0: if the mutation type is *singular* $\tau_0 = \frac{1}{\sqrt{d}}$, if the mutation type is *individual* $\tau_0 = \frac{1}{\sqrt{2d}}$ where d is the dimension of the problem (26).
8. $\alpha(multiplicative\ factor\ of\ \tau_0)$: we scale the value of τ_0 presented by multiplying it with a factor. $\alpha \in [0.1, 5]$, i.e. $\tau = \alpha\tau_0$.
9. τ_0':used if the mutation type is *individual*. $\tau_0' = \frac{1}{\sqrt{2\sqrt{d}}}$ where d is the dimension of the problem (26).
10. $\alpha'(multiplicative\ factor\ of\ \tau_0')$: As for $\tau_\prime$ we scale the value of τ_0' by multiplying it with a factor. $\alpha' \in [0.1, 5]$, i.e. $\tau' = \alpha'\tau_0'$.

3 Experimental Results

3.1 Experimental method

We run each algorithm with a budget of $B = 5000$, and for each set of parameters, we perform 20 runs. Since during the first trial and error experiments, we did not observe any particular improvement in the performance when using a specific set of the parameters we decided to perform a meta-random search among the parameters and methods involved. For each pipeline (GA and ES) we performed 100 runs of meta-random search. Then for each set of parameters the individual with the best fitness value, x^*, the best fitness value $f^* = f(x^*)$ and the mean of the fitness values obtained in each of the 20 runs of the algorithm f^*_{mean} are returned. As a final step, we order the parameter configurations by the value of f^*_{mean} and f^*, and we consider the best 3 for our analysis.

3.2 Evaluation metrics

In order to evaluate the performance of the two algorithms we use the following metrics:

- f^*: the best fitness value obtained over the 20 runs.
- f^*_{mean}: mean of the best fitness values obtained in each of the 20 runs. Its evolution through the number of function evaluations can be shown in the Expected Target Value (ETV) plot.
- *ERT* (expected running time): given a target value ϕ the ERT of the algorithm A for hitting ϕ is:

  $$ERT(A, \phi) = \frac{\sum_{i=1}^{r} min\{t_i(A, \phi), B\}}{\sum_{i=1}^{r} \mathbb{1}\{t_i(A, \phi) < \infty\}}$$

  Where r is the number of independent runs of A, B is the maximum budget, $t_i(A, \phi)$ is the running time, i.e. the number of evaluations to hit the target, with ϕ $t_i(A, \phi) = \infty$ if none of the solutions is better than ϕ, and $\mathbb{1}$ is the indicator function. For our experiments, we considered a range of target values that is $\phi \in [0.94, 1]$.
- ECDF (empirical cumulative distribution function): given a set of targets $\Phi = \{\phi_i \in \mathbb{R} | i \in \{1, 2, .., m\}\}$ and a set of budgets $T = \{t_j \in \mathbb{N} | j \in \{1, 2, .., B\}\}$ for an algorithm A, the ECDF value of A at the budget t_j is the fraction of (run, target)-pairs (r, ϕ_i) that satisfy that run r of the algorithm A finds a solution that has fitness at least as good as ϕ_i within the budget t_j. For our experiment, the set of targets Φ is the range $[0.94, 1]$ with a stepsize of 0.001.
- AUC for the ECDF curve: Given a set of targets $\Phi = \{\phi_i \in \mathbb{R} | i \in \{1, 2, ..., m\}\}$ and a set of budgets $T = \{t_i \in \{1, 2, ..., B\} | j \in \{1, 2, ..., z\}\}$,

the AUC $\in [0,1]$ (normalized over B) of algorithm A on problem P is the area under the ECDF curve of the running time over multiple targets. For maximization, it reads

$$AUC(A,P,\Phi,T) = \frac{\sum_{h=1}^{r}\sum_{i=1}^{m}\sum_{j=1}^{z} \mathbb{1}\{\phi_h(A,P,t_j) \geq \phi_i\}}{r \cdot m \cdot z}$$

where r is the number of independent runs of A and $\phi_h(A,P,t)$ denotes the value of the best solution of A evaluated within its first t evaluations of run h.

- ECDF single target: It is the ECDF when just a single target ϕ is considered. In our analysis, we considered 0.94 as the target value because it is the minimal value of fitness we want to achieve in the optimization.
- AUC single target: It is the AUC associated with the ECDF single target.

3.3 Genetic Algorithm meta-random search results

After running the meta-random search on the genetic algorithm hyperparameters, we achieved the results shown in Table 1. We can observe that the best-found fitness is the same in the three best runs, but run 68 is the best in terms of f^*_{mean}. Run 68 has the highest AUC_{st}(AUC single target) thus, it achieves the target 0.94 a higher proportion of times. However, run 28 has notably better results for the AUC metric.

Table 1: Metrics for the best configurations found by the meta-random search in the genetic algorithm.(rs = random seed)

run_{ID}	rs	f^*	f^*_{mean}	AUC	AUC_{st}	x^*
68	633	0.950554	0.948453	0.118948	0.98960	[1,1,1,1,1,1,0,0,1,1,0,1,0,0,0,1,0,0,0,1,0,2,2,2,2,1]
28	210	0.950554	0.948075	0.127476	0.98756	[1,1,1,1,0,1,1,0,0,0,0,0,1,1,0,1,0,0,1,1,0,2,2,2,2,0]
64	801	0.950554	0.947948	0.118983	0.98756	[1,0,1,1,1,1,0,0,1,0,0,1,0,0,0,0,1,0,1,0,1,2,0,2,2,2]

To further analyze the behavior of each configuration, we plot the expected runtime, the expected target value, and the ECDF curve in Figure 5. In the top-left plot, we observe that the lowest curve is the one corresponding to run 68. This means that the configuration used in that run achieves better fitness values for a lower amount of budget. Looking at the top-right plot, we get to the same conclusion of the table. Indeed run 68 hits the highest f^*_{mean} value even though the lines are closer together and it is more difficult to observe the differences. Finally, analyzing the bottom plot, we can see that run 28 gets a higher ECDF curve. This means that it is able to achieve high targets in a higher proportion of runs. This result can also be observed in Table 1 where we show that run 28 has the highest AUC score.

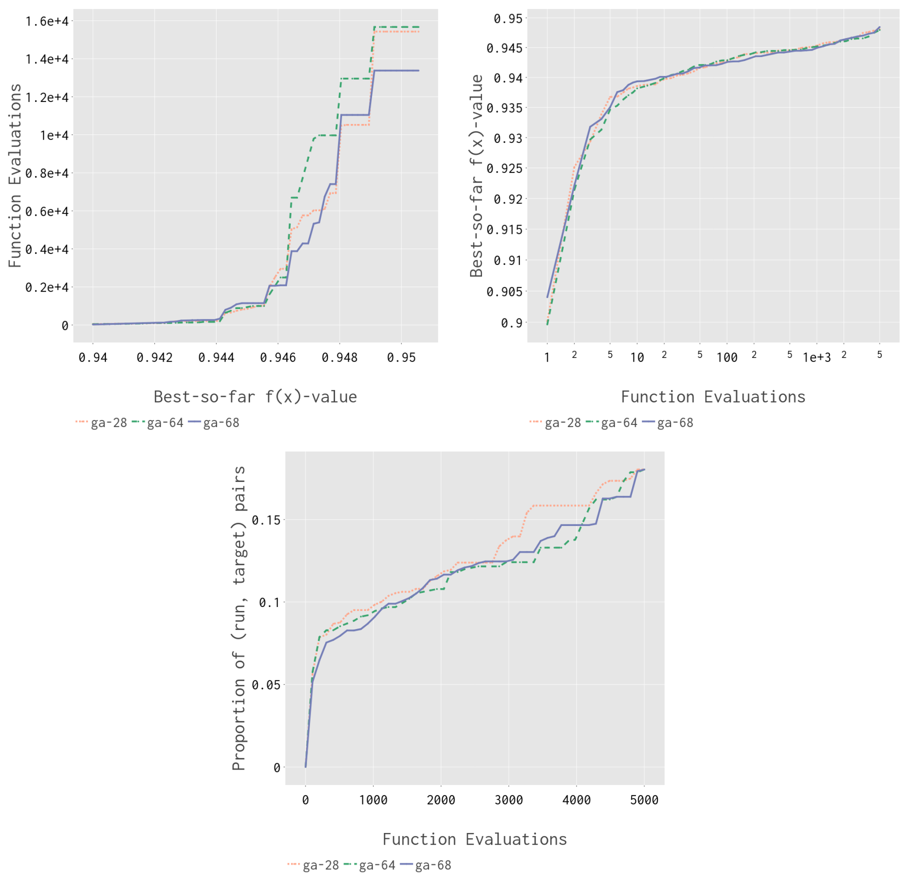

Figure 5: Evaluation figures for the 3 best genetic algorithms. (Top-Left) Expected runtime curves. (Top-Right) Expected target value (x-axis is in *log* scale). (Bottom) Aggregated Empirical Cumulative Distribution (ECDF).

We have chosen run 28 to be the best-found configuration for the genetic algorithm. In Section 4 we justify this choice. Later in this section, we will compare the results from this run to the ones of the best evolutionary strategy and the random search. The hyperparameters chosen in run 28 are presented in Table 2.

We can see that the best-performing configuration presents a rather high population and offspring sizes (253 and 187). This happens because a big population has a higher probability of being close to an optimum since the search space is highly explored. Having an offspring size lower than the population size is unexpected behavior, but since the budget restriction is on the function evaluations and not on the iterations, it should not matter too much. We also see that the chosen mutation rate is more or less twice the ratio $1/l$, with l being

Table 2: Parameters of run 28 of genetic algorithm

μ	λ	*mutation rate*	*crossover probability*
253	187	0.07052	0.21361
crossover method	*crossover points*	*environmental selection*	
one point	1	plus	

the dimension of the individuals. That means that the mutation will change on average two bits in every iteration. The crossover probability is around 0.2, so we will perform crossover around once every 5 times. Having chosen the "plus" environmental selection means that the best fitness will never decrease.

3.4 Evolutionary Strategy meta-random search results

The results of the meta-random search on the evolutionary strategy are the following.

Table 3: Metrics for the best configurations found by the meta-random search in the evolutionary strategy. (rs = random seed)

run_{ID}	rs	f^*	f^*_{mean}	AUC	AUC_{st}	x^*
38	137	0.950554	0.950386	0.125219	0.98756	[1,1,1,1,0,1,1,0,0,1,0,0,1,0,0,1,1,0,1,1,0,2,2,2,2,0]
82	814	0.950554	0.950374	0.125186	0.98960	[0,1,1,1,1,1,0,0,0,1,0,0,0,1,0,1,0,0,1,0,1,1,2,2,2,2]
94	112	0.950554	0.950214	0.136435	0.98858	[1,1,1,1,0,1,0,0,1,0,0,1,0,0,0,1,0,0,1,1,0,2,2,2,2,2]

From the table 3 it is possible to see that the value of f^* is the same for all three runs, while run 38 is the best in terms of f^*_{mean} even if the difference with run 82 is not significant. If we consider the AUC, run 94 is the best. On the other hand run, 82 has the best performance for the AUC_{st} (AUC single target). Thus, it achieves the target 0.94 a higher amount of times.

In figure 6 we analyze the results of the other metrics.

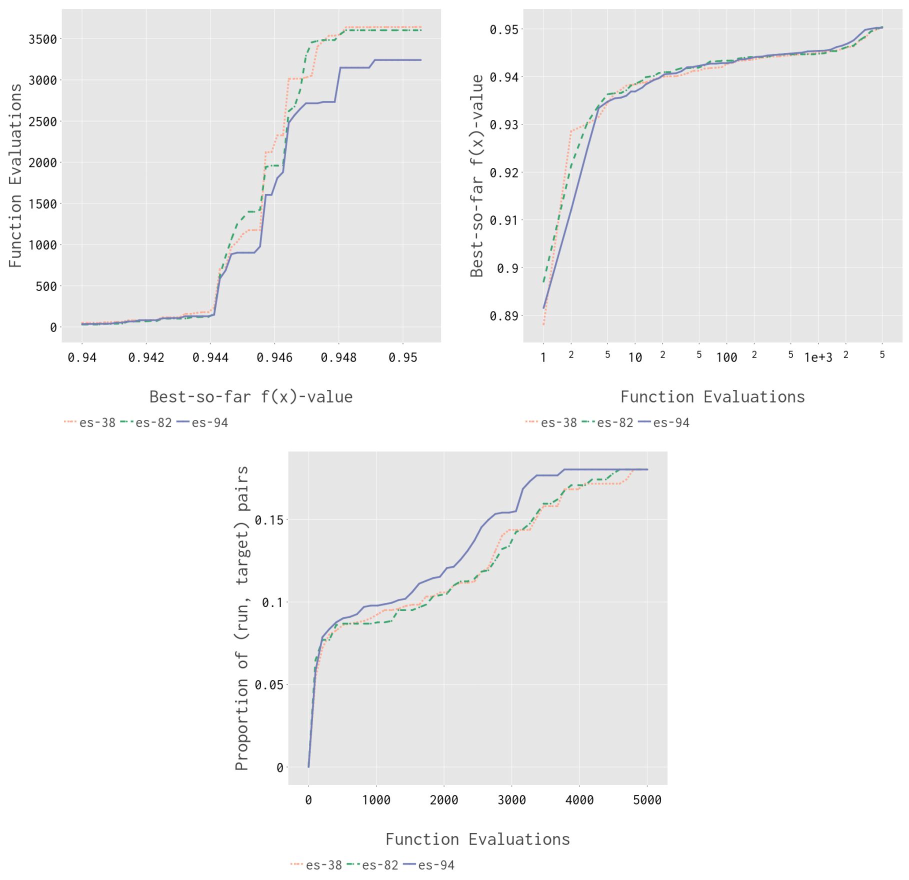

Figure 6: Evaluation figures for the 3 best evolutionary strategies. (Top-Left) Expected runtime curves. (Top-Right) Expected target value (x-axis is in *log* scale). (Bottom) Aggregated Empirical Cumulative Distribution (ECDF).

From the top-left plot, it is possible to see that run 94 has the best ERT with a value that does not overcome 3500 as the other two do. This means that run 94 is faster than the other two in hitting better fitness values. The Expected target value plot shows that the final f^*_{mean} is very similar for all the runs as already shown in table 3. The ECDF shows that run 94 has a higher curve and thus the plot confirms what is already shown in the 3 with the AUC value: run 94 is the best if considering that metric. This means that it is able to achieve high targets in a higher proportion of runs.

From the discussion that is possible to read in Section 4 we choose as best configuration run 94. The results of this run are then compared to the genetic algorithm and the random search. The parameters of this run are presented in table 4.

As we can see in the table, the chosen population and offspring sizes are again

Table 4: Parameters of run 94 of evolutionary strategy

μ	λ	*mutation type*	*recombination type*
120	112	singular	discrete
τ	α	β	*environmental selection*
0.72888	3.71658	2	plus

quite big. We were a bit surprised that the selected mutation type is "singular", meaning we have the same step size for all dimensions. We can also see that the recombination type is discrete, with only two parents involved. The value of $\alpha = 3.71$ implies that the value of τ is almost 4 times the default value τ_0. This means that the values of sigma will mutate quite a lot. Finally, the chosen environmental selection method has been the "plus".

3.5 Algorithm comparison

Table 5: Metrics for the best configuration of each algorithm and the random search.

run_{ID}	f^*	f^*_{mean}	AUC	AUC_{st}	x^*
GA-28	0.950554	0.948075	0.127476	0.98756	[1,1,1,1,0,1,1,0,0,0,0,0,1,1,0,1,0,0,1,1,0,2,2,2,2,0]
ES-94	0.950554	0.950214	0.136435	0.98858	[1,1,1,1,0,1,0,0,1,0,0,1,0,0,0,1,0,0,1,1,0,2,2,2,2,2]
RS	0.948952	0.946090	0.127116	0.98909	[1,1,1,1,1,1,0,0,0,1,0,1,1,0,0,0,0,0,1,0,1,2,2,2,2,2]

In table 5 we compare the final results of the three algorithms. Analyzing the values of the table, we can see that the evolutionary algorithms outperform the random search. Both evolutionary strategy and genetic algorithm have better f^*_{mean} value, meaning that on average the evolutionary algorithms find solutions with better fitness values. In addition, the value of the optimal fitness f^* of the evolutionary algorithms is higher than in the random search. The two evolutionary algorithms also have a higher AUC value with the evolutionary strategy being the best. It is interesting to notice that random search has the best AUC_{st}. Thus a higher number of runs for the random search hit the target of 0.94.

Table 5 also shows that the evolutionary strategy finds solutions with a higher fitness on average (f^*_{mean}). In addition, the evolutionary strategy has a higher proportion of runs hitting a higher target, as shown by the AUC value.

We continue the comparison of the algorithms by plotting the ERT, the ECDF, and the Expected target value.

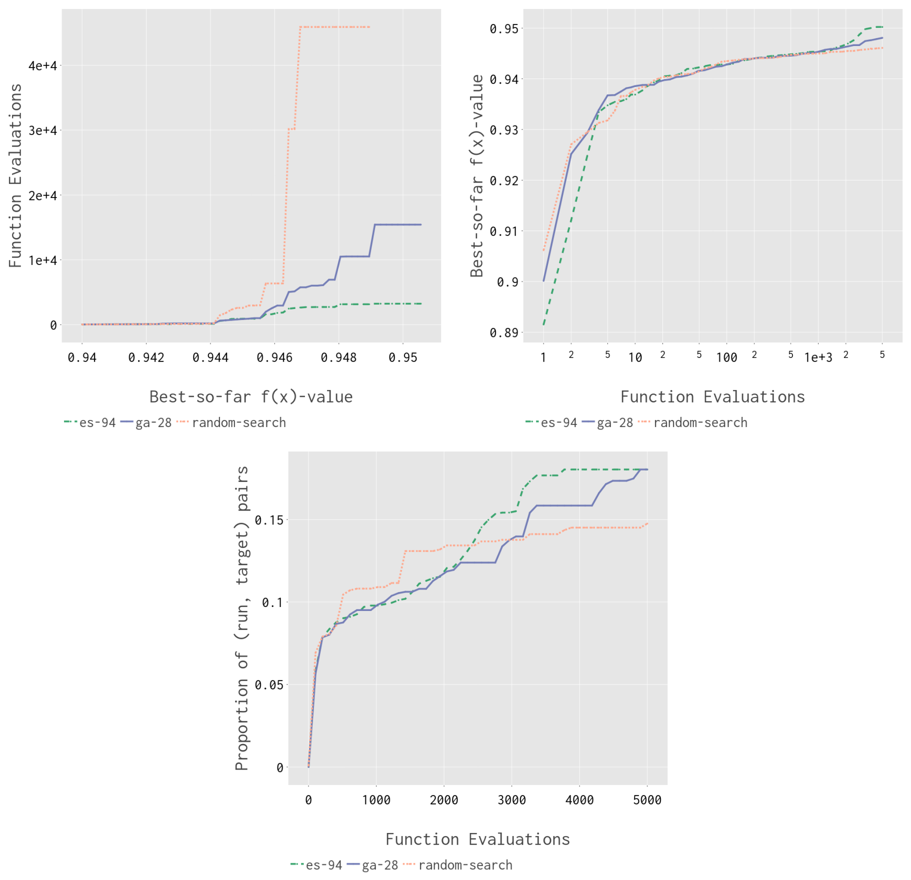

Figure 7: Evaluation figures for the best genetic algorithm, the best evolutionary strategy, and the random search. (Top-Left) Expected runtime curves. (Top-Right) Expected target value(x-axis is in *log* scale). (Bottom) Aggregated Empirical Cumulative Distribution (ECDF).

The top right and bottom plots confirm the results of the table 5. Indeed the top right plot shows that on average the fitness found by the evolutionary algorithms is higher than the one of random search with the evolutionary strategy having the best results. It is interesting to notice that in the first runs the evolutionary strategy has poor performances with respect to the random search and the genetic algorithm. In the ECDF plot, the evolutionary strategy curve hits higher values resulting in higher AUC, as shown in table 5.

The ERT plot is the most interesting plot. Indeed the random search needs a large number of function evaluations to achieve good fitness values. In addition, the optimal fitness f^* found is lower than that of the evolutionary algorithms. The ERT of the evolutionary strategy is lower than the one for the genetic algorithm, meaning that the evolutionary strategy needs a lower number of function evaluations to achieve good fitness values.

4 Discussion and conclusions

4.1 Run selection Genetic Algorithm

Amongst the three runs that we have compared, we want to choose the best-performing one. Since all of them get to the same optimal fitness, we need to rely on other metrics to make that decision. We have chosen these metrics to be f^*_{mean}, AUC, AUC_{st} and ERT. We have found that run 28 performs notably well on the AUC metric while it does a decent job at f^*_{mean} and AUC_{st}. Even though its ERT is far from the one in run 68, we have considered the other metrics to be more significant. For this reason, the selected configuration is the one in run 28.

4.2 Run selection Evolutionary strategy

In order to select the best run we analyzed the global performances of the runs for all the metrics. At first, it is important to remember that the f^* is the same for all the runs so in any case, the final solution x^* will have the best-found fitness values. Even if run 38 has the highest f^*_{mean} this run has poor performances if we consider the metrics AUC, AUC_{st}, and ERT. Thus we discard it as a candidate. When runs 82 and 94 are compared using the f^*_{mean} and AUC values it is possible to see that run 94 outperforms run 82 for AUC while 82 has a better f^*_{mean} value. In addition run 94 has the lowest ERT. Even if both runs 94 and 82 have pros and cons we consider run 94 as the best since its f^*_{mean} is not significantly distant from the best-found value of run 38 while the AUC values of runs 38 and 82 are not close to the one of run 94. In addition the low ERT of run 94 guarantees better performances with fewer function evaluations.

4.3 Three Algorithms comparison

From the experimental results, it is possible to conclude that the random search is the algorithm that performs the worse. It achieves lower f^* and f^*_{mean} values than the evolutionary algorithms. In addition to these results, the ERT of the random search is very high with respect to the ones of genetic algorithm and evolutionary strategy as shown in the plot of figure 7.

Even if the final optimal fitness values f^* of the evolutionary strategy and the genetic algorithm are the same, we can identify the evolutionary strategy as the best algorithm. It has a higher f^*_{mean}; thus on average, it hits better optima than the genetic algorithm. It also has a higher proportion of runs that hit higher fitness values, as shown by the AUC metric. Finally, it also hits better fitness values with fewer function evaluations as demonstrated in the ERT plot, resulting in the best f^*_{mean} and ERT tradeoff.

4.4 Conclusion

In this assignment, we have worked in depth with two different approaches for optimizing the NAS problem, Genetic Algorithms and Evolutionary Strategies. We have explored multiple options for all the methods involved in these algorithms and their hyperparameters. Although the differences in performance were not big, we are confident to conclude that the Evolutionary Strategy has achieved the best results.

References

[1] Chris Ying, Aaron Klein, Esteban Real, Eric Christiansen, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. *CoRR*, abs/1902.09635, 2019.