Metody Systemowe i Decyzyjne
MSID project

Piotr Zatwarnicki
Album number 272641
2nd year, Applied Computer Science

May 2024

Contents

1 Introduction

Reinforcement Learning (RL) [5] has always been a fascinating area of research due to its ability to solve complex decision-making problems by allowing agents to learn from their interactions within an environment (simulation). In RL, an agent seeks to maximize its cumulative reward by making a series of decisions, receiving feedback after each action. This project aims to delve into this intriguing field by comparing various neuroevolution optimization methods within a custom car racing simulation environment.

The main goal of this project is to compare the performance of different neuroevolution algorithms in optimizing an agent's behavior in a car racing game. Neuroevolution, which evolves neural networks using evolutionary algorithms inspired by natural selection, is particularly suited for this task. Given the course's focus on optimization methods, this project is a perfect fit.

To achieve this goal, Python was selected as the primary programming language due to its popularity and the availability of machine learning libraries. A car racing environment was designed to simulate racing conditions, including track layouts and car dynamics. Following this, several neuroevolution algorithms were implemented, such as Differential Evolution [9], Evolutionary Strategy [11], Genetic Algorithm, and a Mutation-Only Genetic Algorithm [10]. Each algorithm was tested and fine-tuned to assess its ability to optimize the neural network controlling the car.

The experiments were conducted on two different racing tracks - easy and hard one, with the car starting in various positions and orientations. The performance of each algorithm was measured based on the cumulative rewards collected by the agent over a fixed number of timesteps. This provided insights into which algorithms were most effective for this specific RL task.

This project aims to shed light on the strengths and weaknesses of different neuroevolution strategies in a controlled RL environment, with potential applications in real-world scenarios. Through this work, valuable insights were gained into the design and optimization of neuroevolutionary algorithms, enhancing both theoretical understanding and practical skills in RL.

2 Environment

An environment in RL [5] problems for timestep t should first output state observations s_t. Then, an agent should analyze the input and return a decision a_t, which is passed to the environment to get s_{t+1} and the return value for a given timestep r_t based on some transition function:

$$F : (s_{t+1}, r_t) = F(s_t, a_t)$$

The interface of the environment should consist of:

- A method to get the current state of the environment.
- A method to react with the environment (transition function), given some decision a, which should return value r (s_{t+1} is not needed as it is held inside the environment).
- A method to check if the environment can still run (e.g., the agent might have lost).

The environment used in this project is a car racing game. The goal is to collect as many points as possible within a given number of timesteps. The car should go as fast as possible, but the most important issue is to avoid hitting any walls. Environment initialization includes:

- A map of boolean values indicating walls (False) and race track (True).
- Start position for the car (a tuple (x, y)).
- Start angle in degrees.
- Maximum number of timesteps.
- Other parameters that were kept constant throughout the research (see 9).

The procedure to get points from the given environment and agent is as presented in Algorithm 1. In the following section, detailed mechanisms of the elements presented in Algorithm 1 are described.

Algorithm 1 Getting reward value of an agent for a given environment

```
procedure EVALUATE_AGENT(agent, map, start_position, start_angle, max_timesteps)
    env ← Environment(map, start_position, start_angle)
    i ← 0
    reward ← 0
    while not env.finished do
        s ← get_state(env)
        a ← get_action(agent, s)
        r ← react(env, a)
        reward ← reward + r
        i ← i + 1
        if i ≥ max_timesteps then
            env.finished ← True
        end if
    end while
    return reward
end procedure
```

`get_state` presented in Algorithm 2, casts several rays looking for the track wall, then scales these values between 0 and 1, indicating how close or how far the car is from the walls in given directions. The chosen rays are:

$$env.ray\_angles = [-90, -67.5, -45, -22.5, 0, 22.5, 45, 67.5, 90]$$

There is also an input indicating the current speed of the car. Example distribution of state values for test run of environment is visible on figure 1.

Algorithm 2 Get state of the environment

```
procedure GET_STATE(environment env)
    v ← []
    for ray_angle in env.ray_angles do
        distance ← 0
        x ← env.car_x
        y ← env.car_y
        while distance < env.max_distance do
            distance ← distance + 1
            x ← distance + cos(ray_angle + env.car_angle)
            y ← distance − sin(ray_angle + env.car_angle)
            if not env.map[x, y] then
                break
            end if
        end while
        distance ← distance/env.rays_distances_scale_factor
        distance ← clip(distance, −env.ray_input_clip, env.ray_input_clip)
        v.append(distance)
    end for
    v.append(env.car_speed/env.max_speed)
end procedure
```

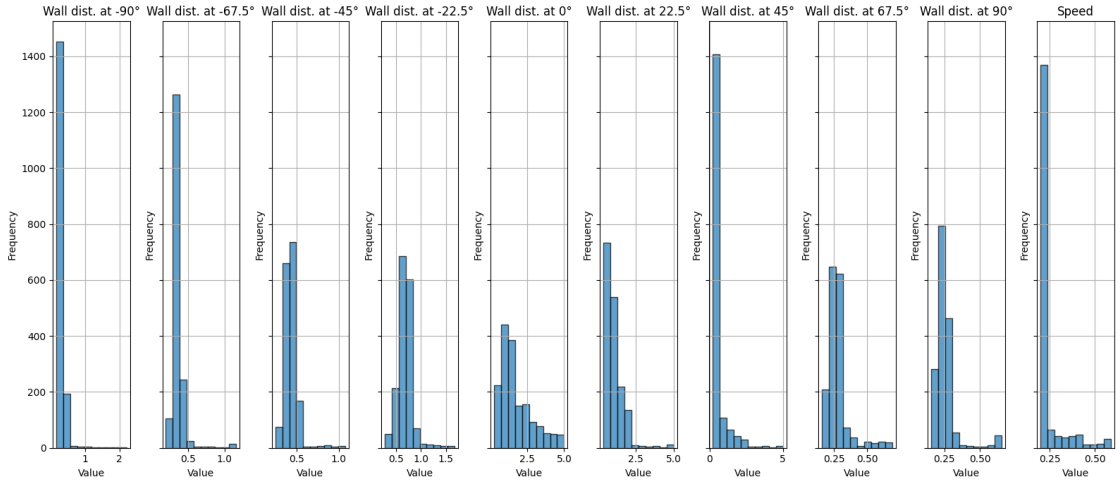

Figure 1: Example of state distribution for a single test environment run, it is evident that angles near zero tend to have higher values. This pattern suggests that the path ahead is typically unobstructed.

`get_action` takes an agent and a state, makes a forward pass of the agent's neural network, and returns a vector of 4 values. The first 3 values control steering, and the fourth controls speed. The `react` procedure presented in Algorithm 3 updates the internal variables of the environment, moves the car, calculates the reward, and checks for collisions. Figure 2 shows example of received rewards for an example run.

Algorithm 3 React to Decision and Move Car

procedure REACT(environment env, decision $decision$) ▷ Start updating internal variables
  $steering\_index \leftarrow \max\_index(decision[0:3])$
  if $steering\_index = 0$ **then**
    $env.car\_angle \leftarrow env.car\_angle + env.angle\_max\_change$
  else if $steering\_index = 1$ **then**
    $env.car\_angle \leftarrow env.car\_angle - env.angle\_max\_change$
  else
    pass
  end if
  $env.car\_speed \leftarrow \text{clip}(env.car\_speed + decision[3] \times env.max\_speed\_change, env.min\_speed, env.max\_speed)$ ▷ End updating internal variables ▷ Start moving the car
  $env.car\_x \leftarrow env.car\_x + env.car\_speed \times \cos(env.car\_angle)$
  $env.car\_y \leftarrow env.car\_y - env.car\_speed \times \sin(env.car\_angle)$ ▷ End moving the car ▷ Collision check and reward calculation
  if car_collide(env) **then**
    $env.finished \leftarrow$ True
    return $env.collision\_reward$
  else
    return $(env.car\_speed/env.max\_speed)^2$
  end if
end procedure

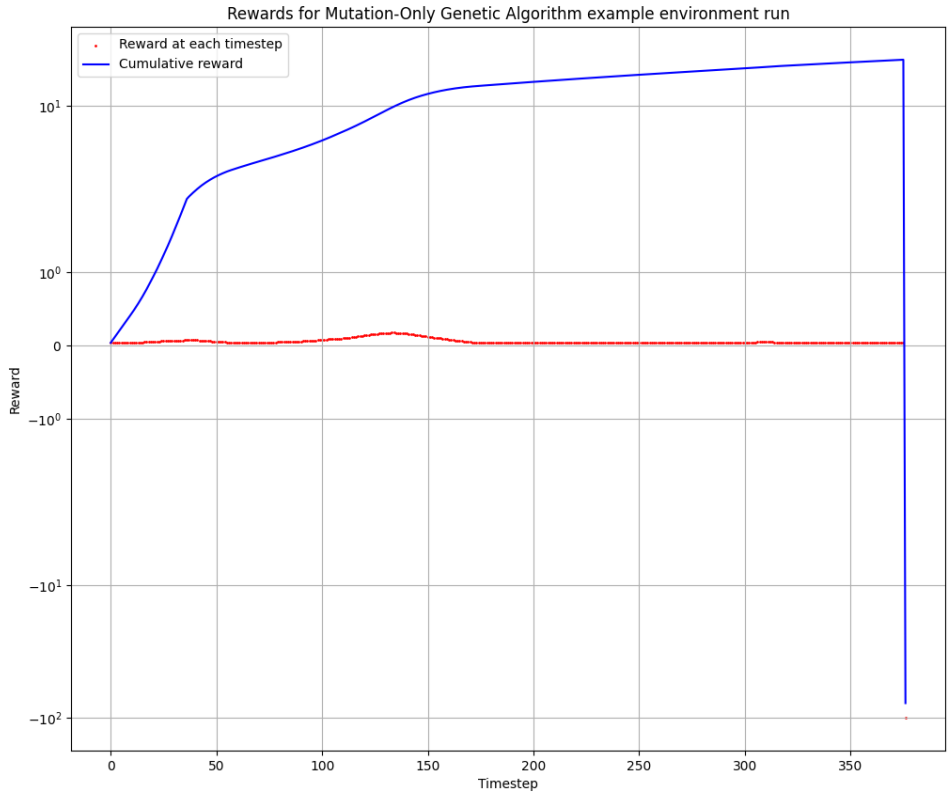

Figure 2: Example of rewards received during a single test environment run shows distinct areas where the car accelerated, earning higher rewards. Notably, in the final timestep, the car incurred a significant penalty for hitting the wall.

Collision `car_collide` checks corner points and middle points of the car's sides. If there is a wall at any point, it returns true.

Constants and reward formulae were conceptualized imperatively. Chosen values provide an interesting, challenging environment where cars are strongly rewarded for speeding, and collisions are heavily penalized. The environment constants were not adjusted to specific algorithms, allowing for an understanding of which neuroevolution algorithms are the most universal.

3 Agent

A multilayer perceptron neural network was selected as the agent to solve this problem. It takes 10 input values, 9 corresponding to distances to walls in different directions, and 1 corresponding to speed. There is a hidden layer of 64 neurons, followed by a ReLU (Rectified Linear Unit) activation layer. After activation, there is another 64-neuron layer, then again ReLU. The output layer has 4 neurons. The first 3 output neurons apply Softmax activation for turning decisions, while the fourth output neuron applies Tanh (hyperbolic tangent function) for speed control, as specified by the environment. Figure 3 presents structure of the described neural network.

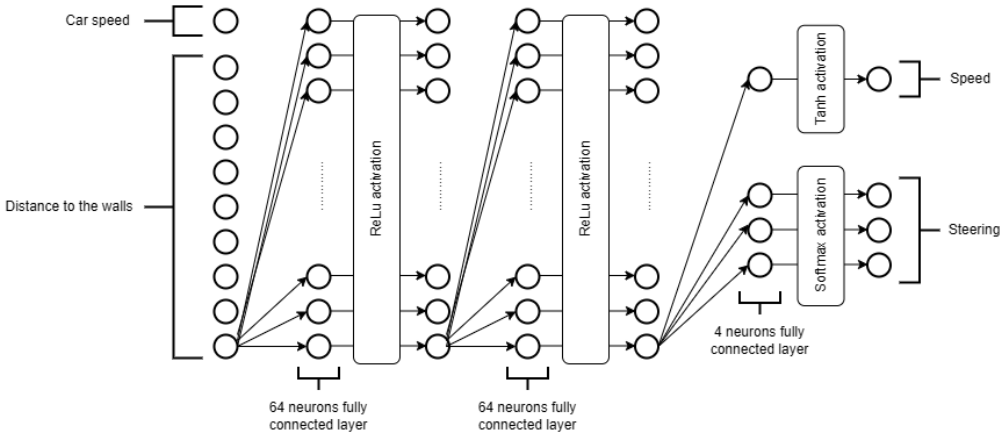

Figure 3: Agent's neural network architecture, Source: own work

As with the environment constants, the neural network structure was chosen imperatively. This architecture can solve some environments, so it was applied and kept constant throughout the research. Initial values for biases were zeros. Initial values for weights were initialized using Xavier initialization [2] where Softmax and Tanh were used, and He initialization [4] was used for layers with ReLU, as recommended by "What is Weight Initialization?" [1].

4 Implementation

Python was selected as the primary environment for implementing this project. It is a very popular and easy-to-use language with a wide variety of packages (e.g. Numpy, TensorFlow, PyTorch), making it the most popular language for machine learning preprocessing and computations.

One of the biggest limitations of Python is its threading capability. Python uses GIL (Global Interpreter Lock), which allows only one thread to process Python commands at a time. It is expected to be removed in Python 3.13 [3], but at the time of writing this paper, Python 3.13 is still in beta version, without support from most libraries. A popular workaround is using the multiprocessing module, which uses a process pool instead of a thread pool. However, it has a huge data copying overhead. Considering that neuroevolution requires many runs of small tasks with a lot of data, this was not a good solution.

Another solution is using C or C++ bindings or a Python superset that can be compiled to C, such as Cython. The latter was used in this paper. Cython can compile any Python code, ensuring superior compatibility with Python, and allows the use of `cdef` declarations, which use raw C data structures. Such code (figure 4) can run without any references to Python, allowing GIL to be released. The entire evaluation of the agent was written without GIL, enabling true parallelism.

Running the simulation without Python overhead had some drawbacks, such as the need to write the neural network from scratch. This was leveraged as an opportunity to deepen the understanding of neural network functioning. The neural network was simple and small, so implementation with raw C types without Python overhead was probably faster than using well-optimized libraries like PyTorch.

```
@cython.boundscheck(False)
@cython.wraparound(False)
@cython.nonecheck(False)
cdef inline float[:, ::1] forward(self, float[:, ::1] inputs) noexcept
    ↪ nogil:
    if inputs.shape[0] > self.output.shape[0]:
        with gil:
            self.output = np.empty([inputs.shape[0], self.output_size],
                ↪ dtype=np.float32)

    cdef float[:, ::1] weights_here = self.weights
    cdef float[::1] biases_here = self.biases
    cdef float[:, ::1] output_here = self.output[:inputs.shape[0]]
    cdef int batch_size = inputs.shape[0]
    cdef int inputs_size = weights_here.shape[0]
    cdef int output_size = output_here.shape[1]
    cdef int i, j, k

    for i in range(batch_size):
        for j in range(output_size):
            output_here[i, j] = biases_here[j]
            for k in range(inputs_size):
                output_here[i, j] += inputs[i, k] * weights_here[k, j]

    return output_here
```

Figure 4: An example of highly optimized Cython code - forward method of dense layer. Execution time of highly optimized Cython code should be similar to that of C code [7], Source: own work

5 Neuroevolution Algorithms

The neuroevolution algorithms used in this paper were chosen based on their ability to work with conventional neural networks. Therefore, popular NEAT (Neuroevolution of Augmented Topologies) [8] was not chosen for comparison.

5.1 Differential Evolution

Differential Evolution (DE) [9] is an optimization method well-suited for continuous optimization problems, including those found in RL. DE optimizes a population of candidate solutions (individuals) through operations inspired by biological evolution, such as mutation, crossover, and selection.

In this work, DE is applied to RL problems where the objective function $F(\theta)$ acts on parameters θ. DE algorithms represent the population as a set of vectors in the parameter space. The goal is to maximize the objective value $F(\theta)$ by evolving the population over successive generations.

In RL problems, $F(\theta)$ represents the stochastic return from the environment, and θ are the policy parameters. DE handles non-smoothness and high-dimensional spaces effectively, making it suitable for optimizing policies in complex environments.

DE proceeds by generating new candidate solutions through the mutation and crossover of existing individuals, and then selecting the best candidates based on their objective values. The mutation operation in DE typically involves adding a weighted difference between two population vectors to a third vector, producing a mutant vector. The crossover operation combines elements from the mutant vector and another vector to create a trial vector. If the trial vector yields a better objective value than a target vector, it replaces the target vector in the population.

The best base variant, which is used in this work, selects the best vector (the one with the highest objective value) as the base vector for mutation. This approach is chosen because our problem requires fast convergence.

The DE algorithm is presented in Algorithm 4.

Algorithm 4 Differential Evolution

Input: Population size N, scaling factor F, crossover rate CR, initial policy parameters $\{\theta_i\}_{i=1}^{N}$
for $t = 0, 1, 2, \ldots$ **do**
  for $i = 1, 2, \ldots, N$ **do**
    Select two distinct indices $r2, r3$ from $\{1, \ldots, N\} \setminus \{i\}$
    Let $r1$ be the index of the best vector in the population
    Generate mutant vector $v_i = \theta_{r1} + F(\theta_{r2} - \theta_{r3})$
    Generate trial vector u_i by crossover:
    for $j = 1, 2, \ldots, D$ **do** ▷ D is the dimension of the parameter space
      if $\text{rand}_j < CR$ or $j = j_{rand}$ **then**
        $u_{ij} = v_{ij}$
      else
        $u_{ij} = \theta_{ij}$
      end if
    end for
    Evaluate returns $F_u = F(u_i)$ and $F_\theta = F(\theta_i)$
    if $F_u \geq F_\theta$ **then**
      $\theta_i \leftarrow u_i$
    end if
  end for
end for

DE repeatedly executes these steps, evolving the population towards better solutions over time. It effectively balances exploration and exploitation through its mutation and crossover operations, allowing it to handle the complexities of RL environments [6].

5.2 Evolutionary Strategy

Evolutionary Strategy (ES) [11] generates multiple individuals each epoch by mutating the primary individual. These individuals are evaluated within the environment, and based on their performance, the gradient is estimated to adjust the parameters accordingly. This process is iteratively repeated.

In this paper, the main optimization goals are reinforcement learning (RL) problems, where the objective function $F(\theta)$ operates on parameters θ. Neuroevolution strategy algorithms represent the population using a distribution over parameters $p_\psi(\theta)$, parameterized by ψ. The aim is to maximize the expected objective value $E_{\theta\sim p_\psi}[F(\theta)]$ by adjusting ψ through stochastic gradient ascent.

In RL problems, $F(\theta)$ represents the stochastic return from the environment, with θ being the policy parameters. ES can manage non-smoothness in the environment or discrete actions by utilizing a Gaussian distribution for p_ψ. This leads to a smoothed objective function

$$E_{\epsilon\sim N(0,I)}[F(\theta+\sigma\epsilon)].$$

We then optimize θ using the gradient:

$$\nabla_\theta E_{\epsilon\sim N(0,I)}[F(\theta+\sigma\epsilon)] = \frac{1}{\sigma}E_{\epsilon\sim N(0,I)}[F(\theta+\sigma\epsilon)\epsilon].$$

This gradient is approximated with samples. The ES algorithm repeats two phases each epoch: perturbing policy parameters and evaluating them, then calculating a stochastic gradient estimate and updating the parameters [11].

The whole procedure is presented in Algorithm 5.

Algorithm 5 Evolution Strategy

1: **Input:** Learning rate α, noise standard deviation σ, initial policy parameters θ_0, population size n
2: **for** $t = 0, 1, 2, \ldots$ **do**
3: Sample noise vectors $\epsilon_1, \epsilon_2, \ldots, \epsilon_n$ from a normal distribution $N(0, I)$
4: Evaluate returns $F_i = F(\theta_t + \sigma\epsilon_i)$ for each $i = 1, \ldots, n$
5: Update policy parameters: $\theta_{t+1} \leftarrow \theta_t + \alpha\frac{1}{n\sigma}\sum_{i=1}^{n} F_i\epsilon_i$
6: **end for**

5.3 Genetic Algorithm with Scattered Crossover and Gaussian Mutation

Genetic algorithms (GAs) are robust optimization methods inspired by the process of natural evolution. They are particularly well-suited for solving complex problems in high-dimensional and non-differentiable spaces. In this section, we describe a GA variant that employs scattered crossover and Gaussian mutation for optimizing deep neural networks in reinforcement learning (RL) tasks.

Our method initializes a population of N individuals, each representing a set of policy parameters θ. The algorithm iteratively performs crossover and mutation to generate new offspring, evaluates their performance, and updates the population based on fitness. The key operations in this GA variant are scattered crossover and Gaussian mutation.

In scattered crossover, two parent vectors are randomly chosen, and their elements are recombined to create two child vectors. For each element, the child vectors inherit the corresponding element from either parent with equal probability. This method ensures a diverse mixing of genetic material, promoting exploration in the search space.

After crossover, the child vectors are mutated by adding Gaussian noise. This noise is drawn from a Gaussian distribution with mean zero and standard deviation σ. This mutation step introduces random perturbations to the child vectors, enabling the algorithm to explore new regions of the parameter space.

The GA algorithm proceeds by performing K crossover operations in each generation. For each crossover, two parents are randomly selected, two children are created through scattered crossover, and the children are mutated. The fitness of the children is then evaluated, and the worse-performing parent is replaced by the better-performing child if the child's fitness exceeds the parent's fitness.

The detailed algorithm is presented in Algorithm 6.

Algorithm 6 Genetic Algorithm with Scattered Crossover and Gaussian Mutation

Input: Population size N, number of crossovers per generation K, mutation standard deviation σ, initial policy parameters $\{\theta_i\}_{i=1}^{N}$
for $t = 0, 1, 2, \ldots$ **do**
  Evaluate returns $F(\theta_i)$ for each $i = 1, \ldots, N$
  for $k = 1, 2, \ldots, K$ **do**
    Randomly select two distinct parents θ_{p1} and θ_{p2} from the population
    Generate two children θ_{c1} and θ_{c2} using scattered crossover:
    for $j = 1, 2, \ldots, D$ **do** ▷ D is the dimension of the parameter space
      if $\text{rand}_j < 0.5$ **then**
        $\theta_{c1j} = \theta_{p1j}$ and $\theta_{c2j} = \theta_{p2j}$
      else
        $\theta_{c1j} = \theta_{p2j}$ and $\theta_{c2j} = \theta_{p1j}$
      end if
    end for
    Mutate children by adding Gaussian noise: $\theta_{c1} \leftarrow \theta_{c1} + \mathcal{N}(0, \sigma^2)$ and $\theta_{c2} \leftarrow \theta_{c2} + \mathcal{N}(0, \sigma^2)$
    Evaluate returns $F_{c1} = F(\theta_{c1})$ and $F_{c2} = F(\theta_{c2})$
    if $F_{c1} > F(\theta_{p1})$ or $F_{c2} > F(\theta_{p2})$ **then**
      if $F_{c1} > F_{c2}$ **then**
        Replace θ_{p1} with θ_{c1} if $F_{c1} > F(\theta_{p1})$
      else
        Replace θ_{p2} with θ_{c2} if $F_{c2} > F(\theta_{p2})$
      end if
    end if
  end for
end for

5.4 Mutation-Only Genetic Algorithm

Deep Neuroevolution, as demonstrated in [10], employs genetic algorithms (GA) as a viable alternative for training Deep Neural Networks (DNNs) in reinforcement learning (RL). This method harnesses evolutionary principles to optimize neural network parameters through selection and mutation, operating effectively in high-dimensional and non-differentiable spaces.

In this work, a mutation-only GA-based approach is applied to RL problems, where the objective function $F(\theta)$ acts on the policy parameters θ. Unlike traditional GAs that use both crossover and mutation to generate offspring, this method focuses solely on mutation to introduce variations in the population.

Mutation-only GA optimizes a population of candidate solutions by iteratively creating and evaluating new generations of solutions, selecting the most fit individuals, and propagating their genetic information through mutation. The algorithm (Algorithm 7) proceeds as follows: it initializes a population of neural network parameters, evaluates their performance in the environment, and generates offspring through mutation. The parent and offspring populations are then concatenated, and the top-performing individuals are selected to form the next generation. This process is repeated for a predetermined number of generations or until a convergence criterion is met.

The mutation operation involves perturbing the parameters of the neural network by adding a small, random change to each parameter. This random change is typically drawn from a Gaussian distribution with a mean of zero and a specified standard deviation. The magnitude of these

changes can be controlled by adjusting the mutation rate, which determines the standard deviation of the Gaussian distribution.

The mutation-only GA algorithm is presented in Algorithm 7.

Algorithm 7 Mutation-Only Genetic Algorithm for Deep Neuroevolution

Input: Population size N, mutation rate M, initial policy parameters $\{\theta_i\}_{i=1}^N$
for $t = 0, 1, 2, \ldots$ **do**
    Evaluate returns $F(\theta_i)$ for each $i = 1, \ldots, N$
    Initialize offspring population $\{\theta_i'\}_{i=1}^N$
    for $i = 1, 2, \ldots, N$ **do**
        Generate mutant vector $\theta_i' = \theta_i + M \cdot \mathcal{N}(0, I)$
    end for
    Concatenate parent and offspring populations $\{\theta_i\}_{i=1}^N \cup \{\theta_i'\}_{i=1}^N$
    Sort the combined population by fitness $F(\theta)$
    Select the top N individuals for the next generation
end for

This mutation-only GA algorithm emphasizes the role of mutation in exploring the parameter space and evolving the population towards optimal solutions. By focusing solely on mutation, it simplifies the evolutionary process while still maintaining the ability to effectively navigate complex and high-dimensional search spaces.

6 Conducted Experiments

Test cases in this research consist of 2 maps for car racing environment. On each map (figures 5 and 6), the car is initialized twice - once at the start point in one direction, and another time in the opposite direction. Final agent fitness is the sum of all rewards collected by the agent in 2 runs - in opposite directions.

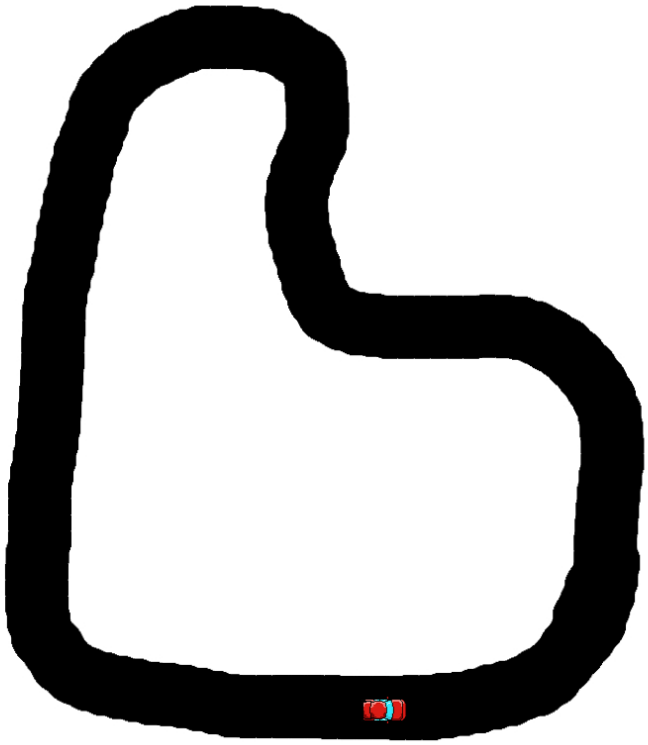

Figure 5: First racing track - simple one. The car is on its starting position, Source: own work

Figure 6: Second racing track - hard one. The car is on its starting position, Source: own work

All of the used algorithms are prone to metaparameters. Therefore, for each of them, some possible values were chosen based on literature and the author's prior knowledge. From these values, all possible combinations of tuples were constructed and for each tuple, 3 tests were performed with different random seeds.

Each run of the environment was processed with a constant number of maximum timesteps throughout the research, which was 10,000. It was observed that it makes possible driving a whole loop at least once at minimum speed.

Usually, in evolutionary computations, methods are compared based on the number of fitness evaluations, because evaluating fitness takes the most time. Usually, performing other actions is a minor overhead, therefore the number of fitness evaluations is our metric in this research. A maximum number of 100,000 fitness evaluations was set, because of limited computational resources. Each method was forced to stop if it exceeds this limit of evaluations.

Algorithm 8 presents the procedure of collecting research data.

Algorithm 8 Experiment Procedure

Initialize:
  Set $max\_timesteps \leftarrow 10,000$
  Set $max\_fitness\_evaluations \leftarrow 100,000$
  Define list of algorithms A
  For each algorithm $a \in A$
    For each metaparameter m of algorithm a
      Define list of possible values for m
  Construct all possible tuples of metaparameter values for each algorithm
for each algorithm $a \in A$ **do**
    for each tuple of metaparameters t for algorithm a **do**
      for $i = 1$ to 3 **do**
        Record results for run of algorithm with with random seed and maximum evaluations $max\_fitness\_evaluations$ and environment with $max\_timesteps$
      end for
    end for
end for

7 Analysis of Results

In this section, we will present and discuss the results of the individual algorithms, followed by a comprehensive comparison.

7.1 Differential Evolution

Differential Evolution's experiments were conducted with given metaparameters:

```
"Differential_Evolution": {
    "population": [100, 1000],
    "cross_prob": [0.9, 0.5],
    "diff_weight": [0.8, 0.5],
},
```

Results for the first track are visible in figure 7. Results for the second, harder track are visible in figure 8. A population size of 100 in general was better than 1000 - it seems that quick convergence was important in this problem. Blue (population: 100, cross_prob: 0.5, diff_weight: 0.5) and orange (population: 1000, cross_prob: 0.9, diff_weight: 0.8) were the most universal from all metaparameters - they were one of the best for both tracks.

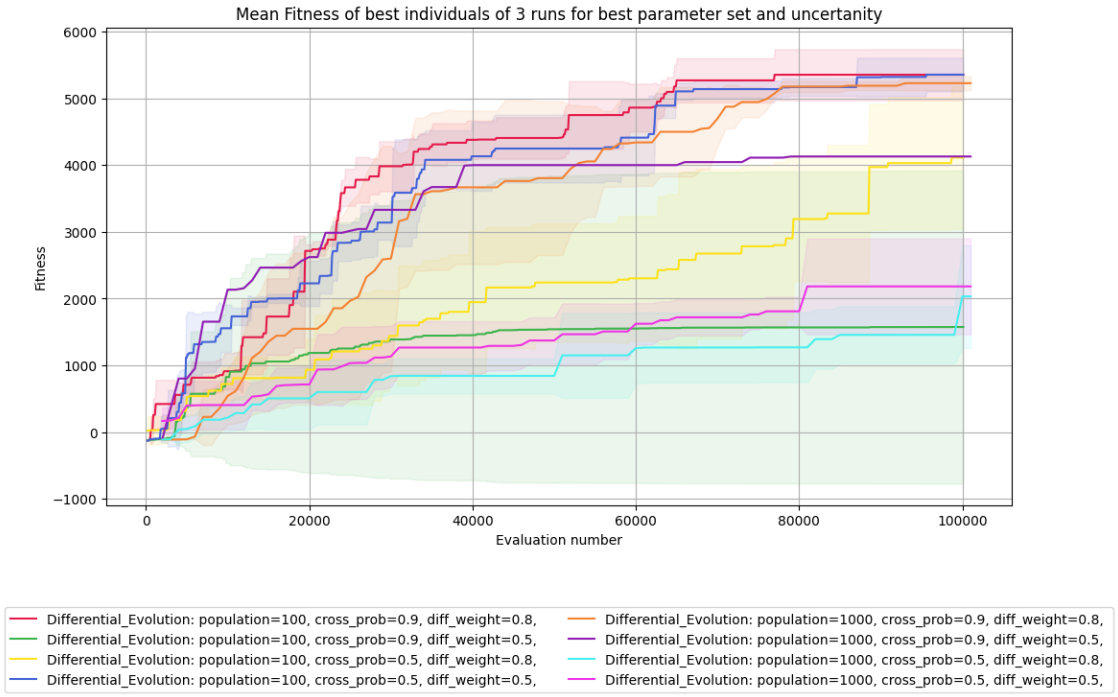

Figure 7: First track. Results of Differential Evolution for all combinations of metaparameters. It is the mean of 3 runs with standard deviation, Source: own work

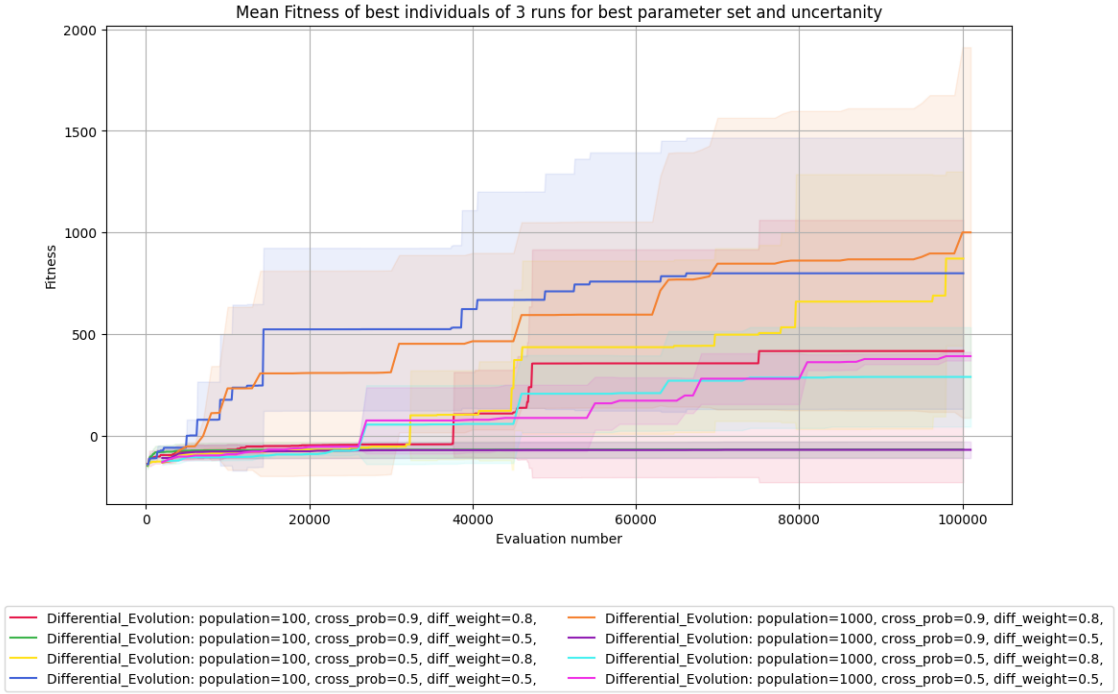

Figure 8: Second track. Results of Differential Evolution for all combinations of metaparameters. It is the mean of 3 runs with standard deviation, Source: own work

7.2 Evolutionary Strategy

Evolutionary Strategy's experiments were conducted with given metaparameters:

```
"Evolutionary_Strategy": {
    "permutations": [1000],
    "sigma_change": [0.01, 0.001],
    "learning_rate": [0.1, 0.01, 0.001],
}
```

Results for the first track are visible in figure 9. Results for the second, harder track are visible in figure 10. In general, Evolutionary Strategy did not succeed on any track. It is suspected that the fitness landscape of this environment is extremely non-linear, and therefore small changes in the Evolutionary Strategy might fail in gradient approximation. It was checked that even very small changes in parameters can ruin the whole result - one wrong decision can result in the end of the environment, therefore all possible further rewards are lost. Additionally, Evolutionary Strategy has one main individual which is updated - even if some individuals suddenly do very well (figure 9), their achievement is lost.

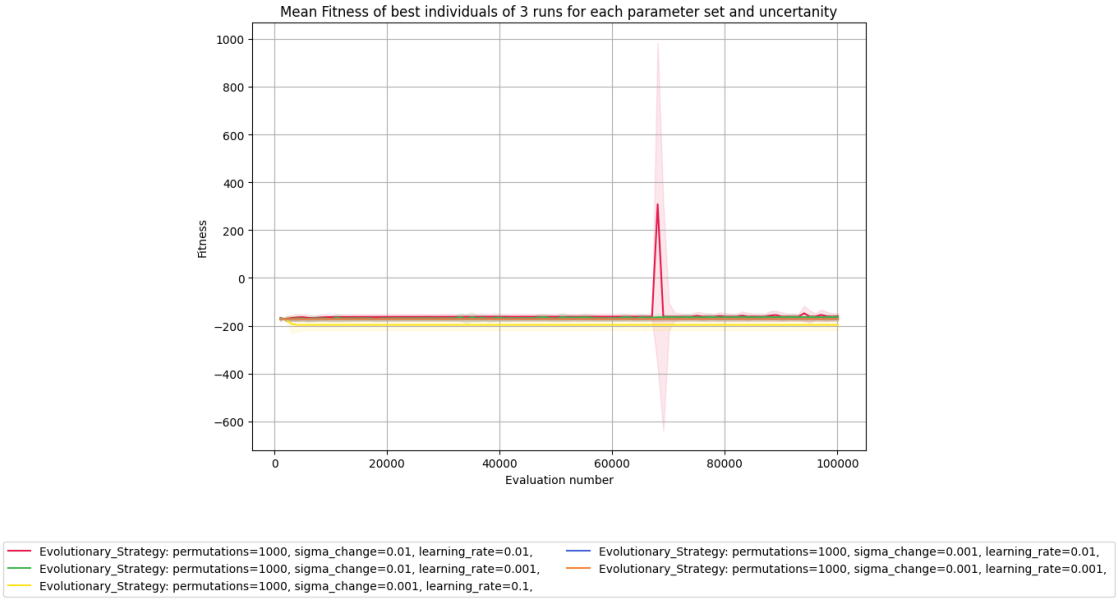

Figure 9: First track. Results of Evolutionary Strategy for all combinations of metaparameters. It is the mean of 3 runs with standard deviation, Source: own work

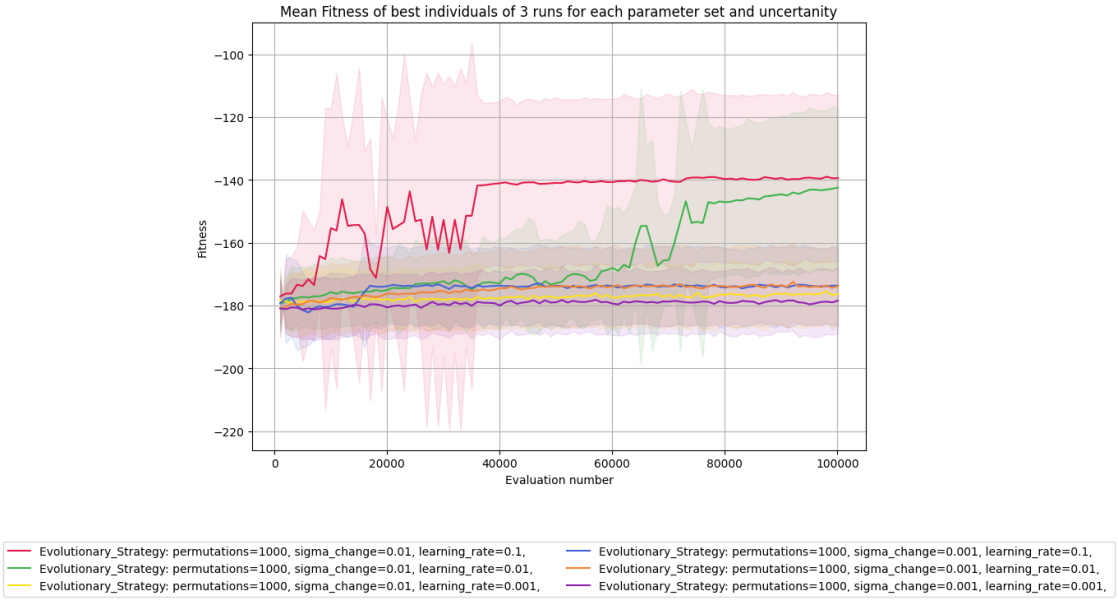

Figure 10: Second track. Results of Evolutionary Strategy for all combinations of metaparameters. It is the mean of 3 runs with standard deviation, Source: own work

7.3 Genetic Algorithm

Genetic Algorithm's experiments were conducted with given metaparameters:

```
"Genetic_Algorithm": {
    "crosses_per_epoch": [99],
    "population": [200, 1000],
    "mutation_factor": [0.1, 0.01, 0.001],
}
```

The Genetic Algorithm did better than Evolutionary strategy, but worse than Differential Evolution. The Genetic Algorithm with a traditional combinatorial crossover scatter operator seems to be the wrong choice for this environment. Also, a more greedy approach could be beneficial. Nevertheless, the best results were achieved for a population of 200 and a mutation factor of 0.1 (red line). A small population size increased convergence and a high mutation rate let the algorithm leave local optima.

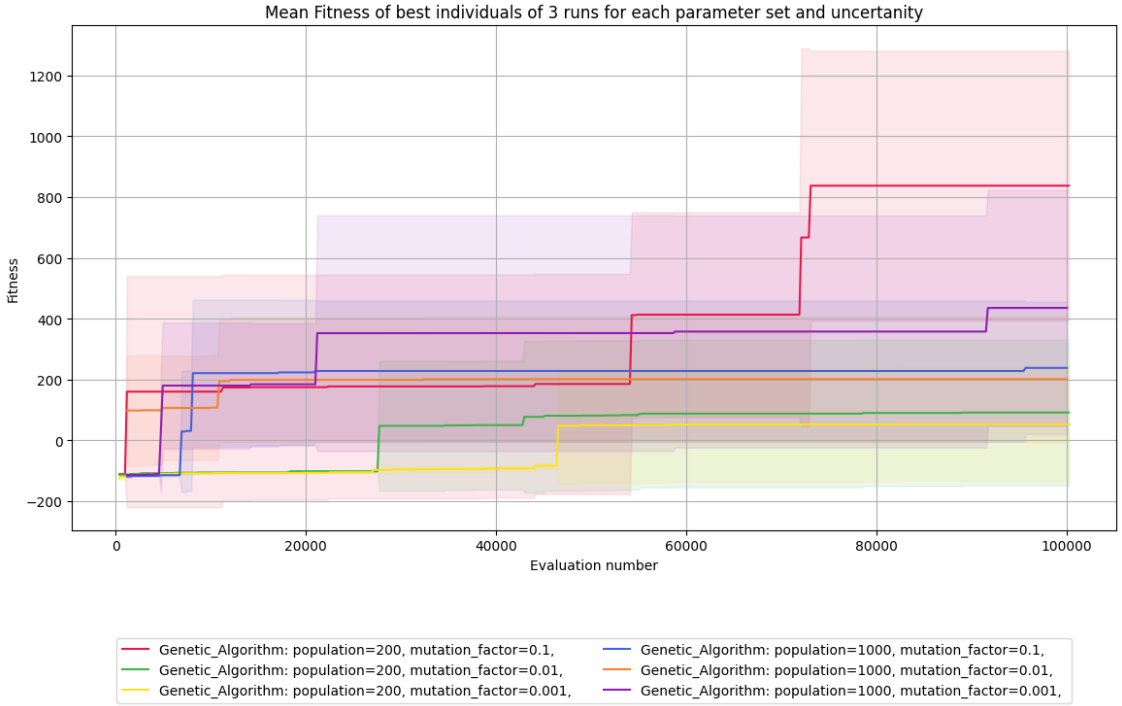

Figure 11: First track. Results of Genetic Algorithm for all combinations of metaparameters. It is the mean of 3 runs with standard deviation, Source: own work

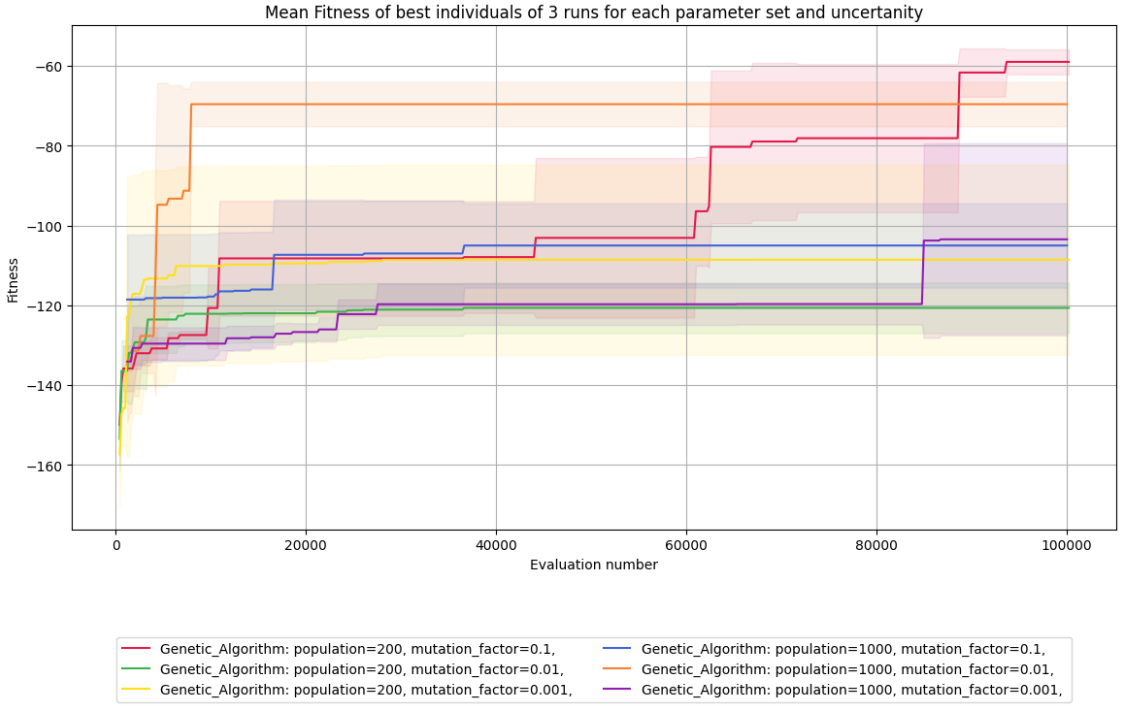

Figure 12: Second track. Results of Genetic Algorithm for all combinations of metaparameters. It is the mean of 3 runs with standard deviation, Source: own work

7.4 Mutation-Only Genetic Algorithm

Mutation-Only Genetic Algorithm's experiments were conducted with given metaparameters:

```
"Evolutionary_Mutate_Population": {
    "population": [100, 300, 1000],
    "mutation_factor": [0.1, 0.01, 0.001],
},
```

It was named *Evolutionary_Mutate_Population* in implementation, because Mutation-Only Genetic Algorithm is in fact a kind of evolutionary algorithm. It is easily visible, that mutation factor of 0.1 lead to the best results on both tracks regardless of population size. It seems that a higher mutation factor enabled individuals to jump out from local optima. Furthermore, smaller population sizes were beneficial - it means that fast convergence introduced by a smaller gene pool was more important than exploration capabilities of big populations.

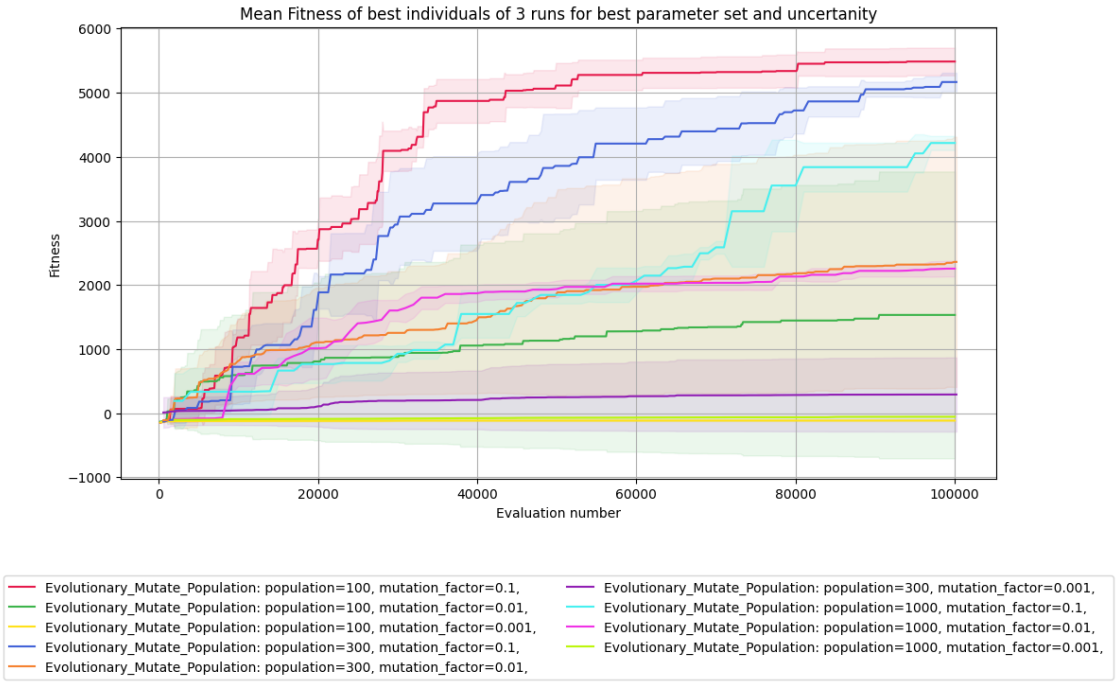

Figure 13: First track. Results of Mutation-Only Genetic Algorithm for all combinations of metaparameters. It is the mean of 3 runs with standard deviation, Source: own work

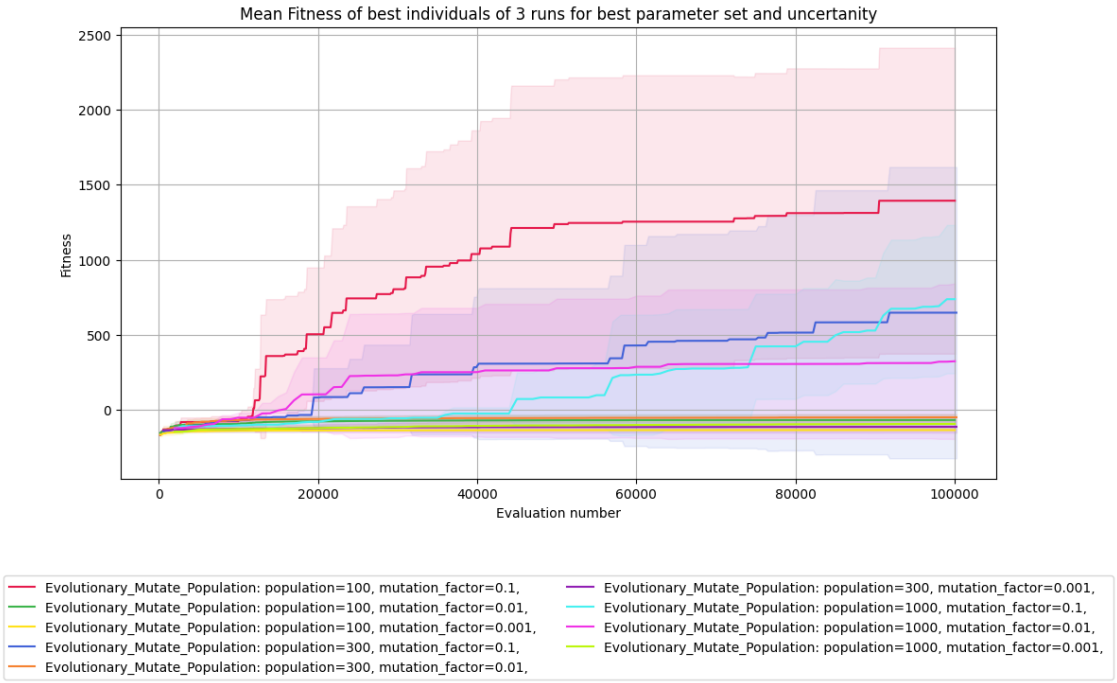

Figure 14: Second track. Results of Mutation-Only Genetic Algorithm for all combinations of metaparameters. It is the mean of 3 runs with standard deviation, Source: own work

7.5 Algorithms Comparison

For each algorithm, the set of parameters that led to the highest mean fitness of the best individuals were chosen and compared against each other. On the first track, Mutation-Only Genetic Algorithm (named *Evolutionary_Mutate_Population*) and Differential Evolution were almost identical, with MOGA having a slight edge. Both Genetic Algorithm and Evolutionary Strategy perform poorly, even though it is a simple track.

On the second track, again MOGA and Differential Evolution were best, this time MOGA won with a higher difference. Other algorithms performed very poorly.

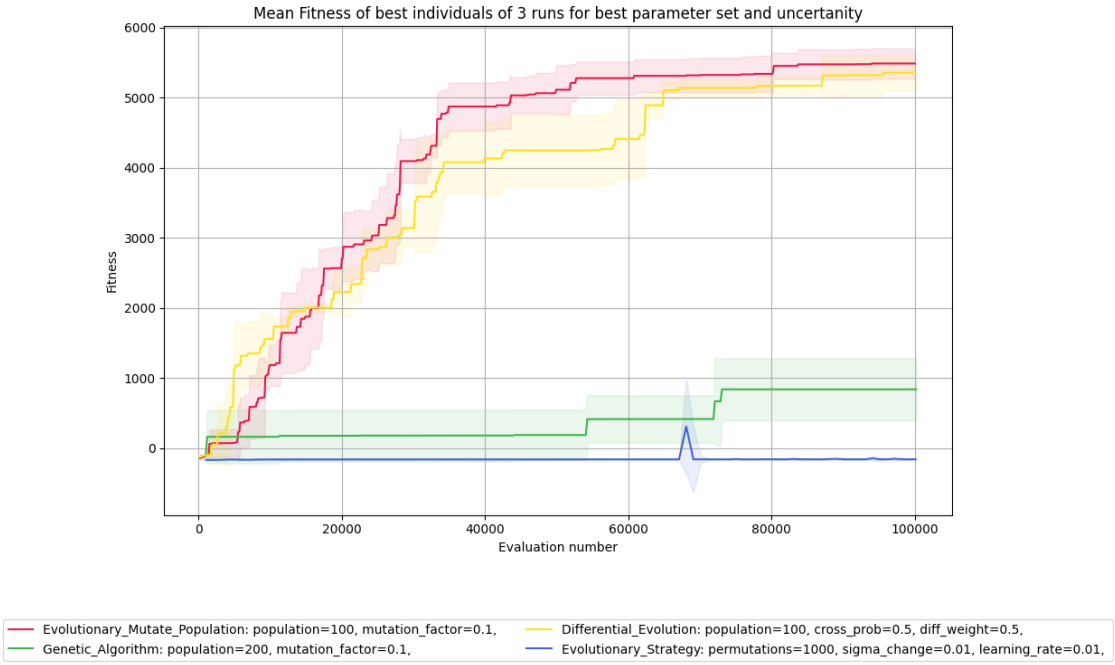

Figure 15: First track. Best set of parameters for each algorithm, Source: own work

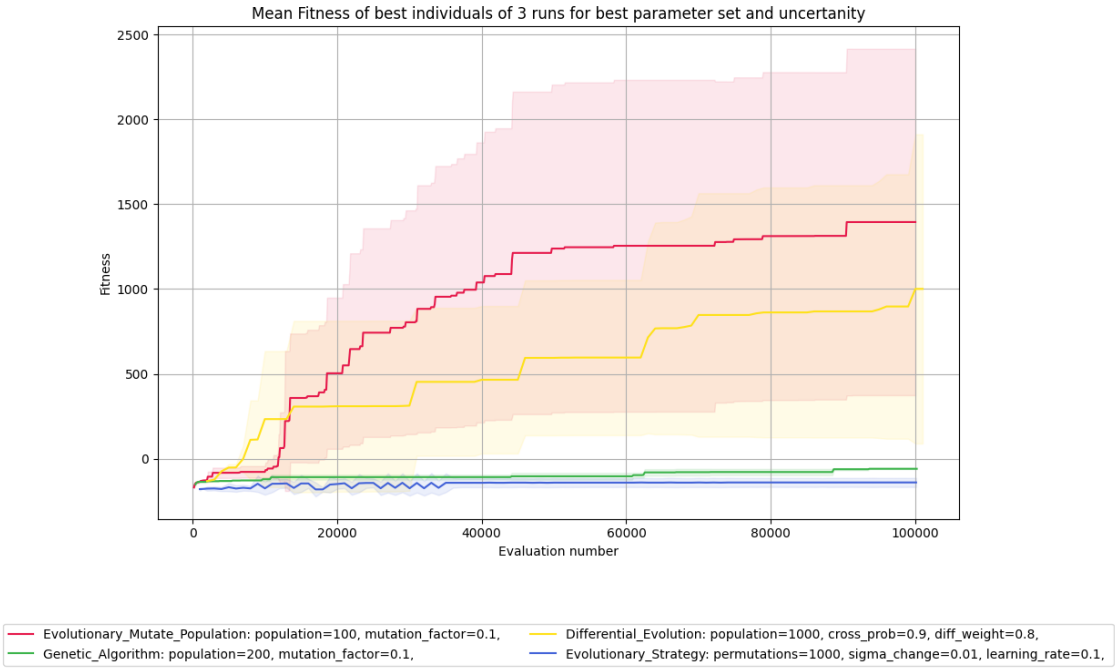

Figure 16: Second track. Best set of parameters for each algorithm, Source: own work

8 Summary

Presented project, part of MSID coursework, explored the application of various neuroevolution optimization methods in a custom car racing simulation environment. Using Python and its comprehensive libraries, a framework was developed to evaluate the effectiveness of Differential Evolution, Evolutionary Strategy, Genetic Algorithm, and Mutation-Only Genetic Algorithm in optimizing the performance of a neural network-driven racing agent.

The results showed that both greedy variant of Differential Evolution and the Mutation-Only Genetic Algorithm performed exceptionally well, consistently achieving high performance across various tracks and initial conditions. This suggests that neuroevolutionary approaches are highly effective for optimizing complex Reeinforcement Learning (RL) tasks like autonomous car racing.

Given more time, the project could further explore advanced neural network architectures, experiment with competitive RL techniques, like on-policy and off-policy RL [5], and introduce more challenging and varied tracks. These enhancements would provide a deeper understanding of the scalability and robustness of neuroevolutionary algorithms in even more complex RL environments.

Overall, the project not only met its primary goal of comparing different neuroevolution strategies but also resulted in the creation of a flexible and extensible framework for RL research. This framework, complete with reusable code and comprehensive documentation, sets a solid foundation for future studies in neuroevolution and RL, enabling further exploration and innovation in these exciting fields.

References

[1] Francesco Franco. What is weight initialization?, 2020. URL https://medium.com/@francescofranco_39234/what-is-weight-initialization-a58606f62513. Accessed: 2024-05-16.

[2] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. pages 249–256, 2010. URL http://proceedings.mlr.press/v9/glorot10a.html.

[3] Sam Gross. Pep 703 – making the global interpreter lock optional in cpython, 2023. URL https://peps.python.org/pep-0703/. Accessed: 2024-05-16.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. pages 1026–1034, 2015. doi: 10.1109/ICCV.2015.123. URL https://ieeexplore.ieee.org/document/7410480.

[5] OpenAI. Spinning up in deep reinforcement learning. URL https://spinningup.openai.com/en/latest/spinningup/rl_intro.html#key-concepts-and-terminology.

[6] K. Price, R. Storn, and J. Lampinen. *Differential Evolution: A Practical Approach to Global Optimization*. Springer, 2005.

[7] Paul Ross. The performance of python, cython and c on a vector, 2014. URL https://notes-on-cython.readthedocs.io/en/latest/std_dev.html. Accessed: 2024-05-16.

[8] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 2002. URL https://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf. Accessed: 2024-05-16.

[9] R. Storn and K. Price. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11:341–359, 1997.

[10] Felipe Such, Vivek Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

[11] Xi Chen Ilya Sutskever Tim Salimans, Jonathan Ho. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017. URL https://arxiv.org/abs/1703.03864.

9 Appendix A

```
"environment": {
    "name": "Basic_Car_Environment",
    "universal_kwargs": {
        "angle_max_change": 1.2,
        "car_dimensions": (30, 45),  # width, height
        "initial_speed": 1.2,
        "min_speed": 1.2,
        "max_speed": 6,
        "speed_change": 0.04,
        "rays_degrees": (-90, -67.5, -45, -22.5, 0, 22.5, 45, 67.5, 90),
        "rays_distances_scale_factor": 100,
        "ray_input_clip": 5,
        "collision_reward": -100,
    }
},
"neural_network": {
    "input_normal_size": 10,
    "out_actions_number": 4,
    "normal_hidden_layers": 2,
    "normal_hidden_neurons": 64,
    "normal_activation_function": "relu",
    "last_activation_function": [("softmax", 3), ("tanh", 1)],
}
```

Constants used throughout research.