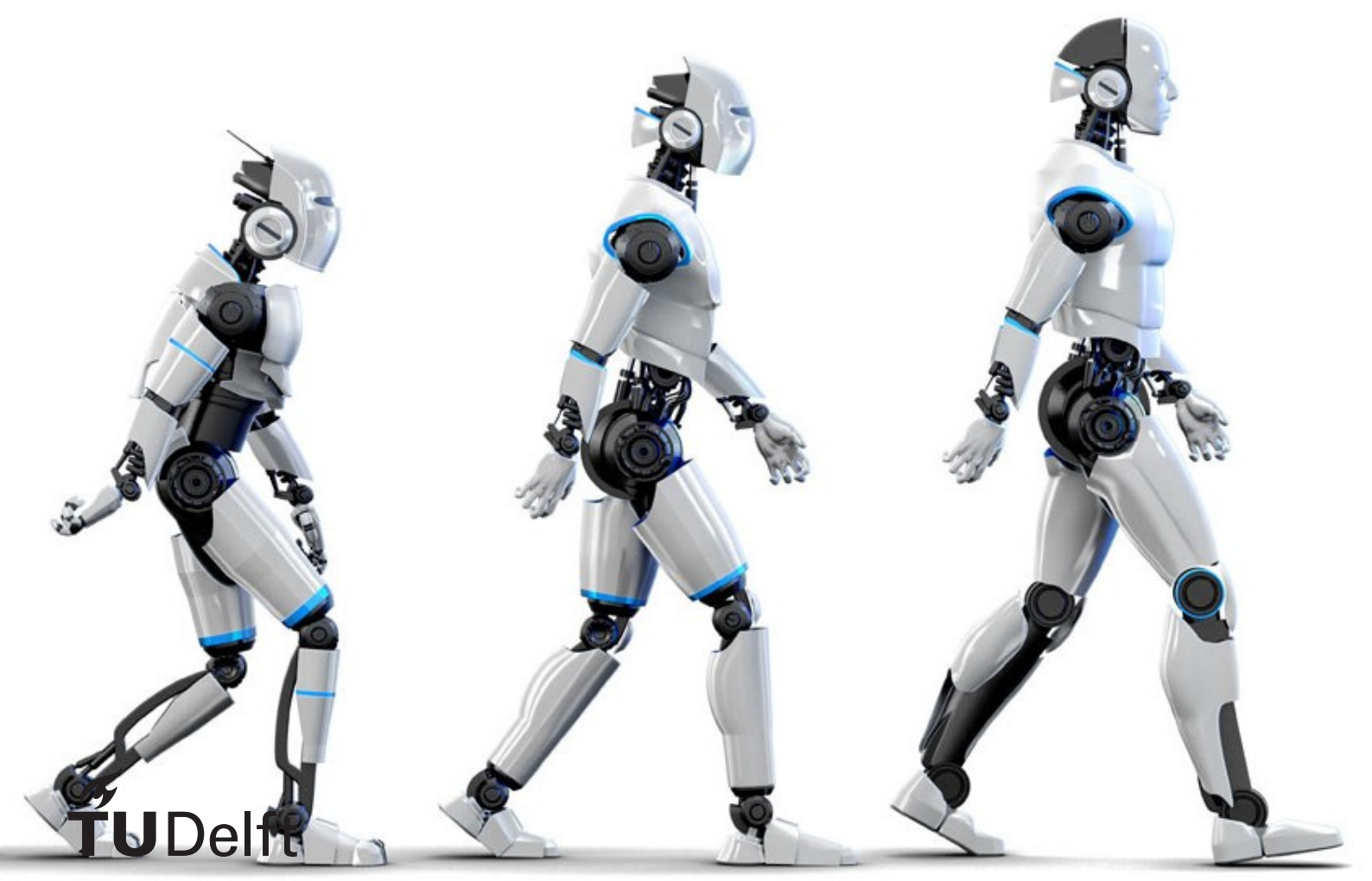
Evolutionary Robotics Assignment

AE4350 Bio-Inspired Intelligence & Learning for Aerospace Applications

4667638 Tzanetos, George

September 2, 2022



TU Delft

Contents

1

Introduction

In recent decades, machine learning is in the forefront of innovation for science. The ways experts apply, are based upon focusing on the superior traits of both machines and humans, and joining them to achieve solutions previously thought unattainable. With time, the field progresses in not only effectiveness, but also variety of methods. One can attribute this diversity to the ambition of attacking newer problems with Machine Learning every day. At this moment, there is not little shortage of methods and definitely not little shortage of problems to be faced. One large chapter of learning is Bio-inspired learning, the idea behind which is explained by its name. What better way to plan a solution against a problem by observing the natural world, and the ways it has adjusted it self to different types of adversity for millions of years? A Bio-Inspired approach to structuring Machine learning algorithms is, on its own, a wide front, with many variations of strategy and applications. The idea of one of perhaps the most popular Artificial Intelligence method, Neural Networks, is in itself a form of bio-mimicry, and specifically of the human brain. Some more examples include Swarming, Self-Supervised Learning, Sensory Networks, and more. The one, however, that this report is focused on is Evolutionary Robotics.

Evolutionary Robotics applies natural principles such as selection, variation, and heredity as guides to the design of robots with embodied intelligence [1]. The idea of application of evolutionary principles is not new, and was first since in the late 1990s, where the chapter of Evolutionary Computing proved useful to optimization problems. Primarily, Evolutionary Algorithms incorporate the ideas of populations and generations. The further the algorithm progresses in generation number, the better the results are expected to be. Among each generation there is a certain chosen number of individuals that constitute a population. Like in the natural world, and the principles of Evolution, the individuals that perform the best at a certain task among the population will be favoured to help create the next generation. In other words, this principle is most commonly known as the "survival of the fittest" principle. This of course means that there needs to be a measure for the evaluation process of each generation. That is a scoring number usually referred to as the fitness, and it is proportional to the performance of the algorithm. For the creation of the next generations, many methods apply. After selection, mating can be incorporated between two fit individuals to create offspring, and then mutation.

The assignment that this report will attempt to tackle is the Cart Pole problem created by the Open AI Gym[2]. The objective of the whole website is to have various open source games or challenges, for people to apply their Artificial Intelligence algorithms, and train them. The Cart Pole is essentially a classical control problem, where a small wagon moves left and right, has a pole that is fixed vertically on top of it and is free to rotate about the hinge. The objective is for the wagon to move appropriately, so that it balances the pole vertically, without moving out of frame, and for a fixed amount of time. Tthe controls acceptable by the algorithm are -1 for left and 1 for right, and not fractional numbers in between. chapter 3, displays how the above is dealt with, followed by a sensitivity analysis in chapter 4, and a few concluding remarks in chapter 5.

2

Methodology

This particular Evolutionary Algorithm, starts from the general approach and proceeds with appropriate modifications to the task, as expected. It has been briefly explained that each generation evolves based on the performance of the previous one. However, the first one has no predecessor, so unavoidably it is a randomly selected population of genomes (individuals). From then on, each generation is evaluated, after which, selection and variation occurs. When the desired number of generations is created, the algorithm is terminated. A schematic of the general process of Evolutionary Robotics, is shown in Figure 2.1.

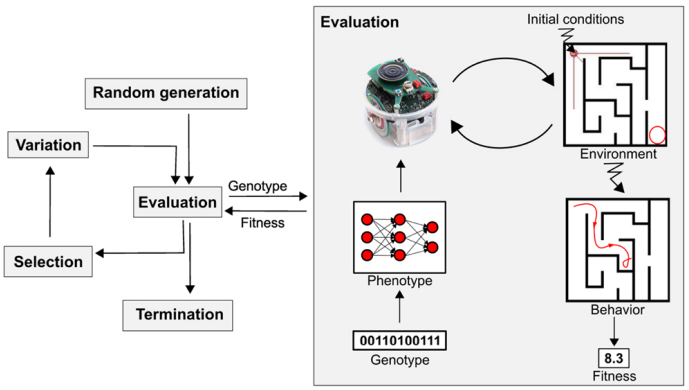


Figure 2.1: General Scheme of an Evolutionary Robotics approach [1]

The following subsections, explain in detail the gray box action, as well as the ones of the inner loop. It has also been decided, for the sake of having a baseline of comparison, that the step count will be fixed to 500 and the generations to 30, in order to get a high enough level of results, over the timeframe of the assignment.

2.1. Agent

For the inner loop to function, there needs to be present a controller that actually moves the robot, or in this case, the simulated cart of Figure 2.2.

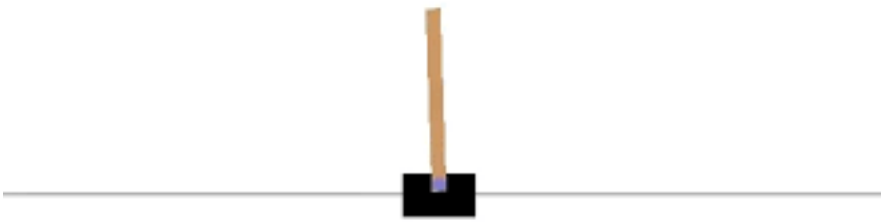


Figure 2.2: CartPole environment snapshot https://gym.openai.com/envs/CartPole-v1/

It has been observed that neural networks generally provide a good controller for Evolutionary Robotics[1], also referred to as an agent. Here, a non-linear neural network is chosen, mathematically described by Equation 2.1.

$$y = f\left(\sum_{i=0}^{n} w_i \cdot x_i\right) + b \tag{2.1}$$

Sometimes the equation can simply be a sum of each neuron's weights w_i times the input values x_i, followed by a bias b. In this network, this in encased by an activation function f, making the output equation non-linear. This is a necessary adjustment for performance, since the observations of the agent are also non-linear, so its only appropriate. The activation function typically chosen is the sigmoid or logistic function, shown in Equation 2.2.

$$\sigma = \left(\frac{1}{1 + e^{-x}}\right) \tag{2.2}$$

Regarding the structure itself of the neural network, it takes conventional form once again, since the focus of this assignment is on the evolution of it. It has 4 layers, which are the input nodes, the two hidden layers and the output. The first and last are dictated by the environment of the Cart Pole by the Open AI Gym[2]. Namely, the observation space, which is what the agent perceives, has four input nodes: the cart's position, its velocity, the pole's angle, and its angular velocity. The action space, which is how the agent can act, contains two output nodes: move left (-1), or right (1). Finally, the hidden layers are free to design, and in most cases of neural network layouts, they are a product of trial and error. Therefore, the typical route is taken here, in which both hidden layers contain 8 nodes. Now that the structure of the network is clear, it is time to evolve the agent, generation by generation.

2.2. Evaluation

This part essentially is the core of how the model is trained towards the completion of the task. Essentially in evaluation, each genome of every population is judged on its performance. That is, the values of it are passed as weights and biases into the neural network (genotype to phenotype). In python form, this is the part of the evaluate() function of the NN_agent class, and where the corresponding weight and biases are added to their nodes. The evaluation phase, in other words is the center of the inner loop, and what assists each generation to evolve and vary, so that it shows better results than previous ones.

2.3. Selection

Selection is perhaps the factor that will be the most influential in how much better the next generation is. Namely, when the population is evaluated and its traits are listed, it is time to actually see who is the "fittest". For each genome, a reward is given for every timestep the pole is kept upright. The sum of rewards of a single game is the corresponding genome's fitness. Using the bisect_right function the index of each of two parents is obtained, both of which are passed through variation until the appropriate size of the new population is reached.

2.4. Variation

The final stage in the repeated algorithm is variation. At this point, the genomes of a generation have had their performance evaluated and they have been selected. Obviously for the population to evolve, new genomes have to be created, and so far there are only the best of the same previous ones available. In variation the concepts of mating and mutation are used to resolve that. Mating, also known as crossover, occurs in one of two ways. The first is achieved by dividing two individuals at a randomly generated point along the genome, and crossing over the divided parts to create offspring, specifically two new genomes. The one applied here is taking the two genomes and getting averaging the value of their weights and biases to get one child. Mutation in the natural world is the alteration in the genetic sequence of an organism, and occurs relatively rarely in the event of exposure to mutagens. Similarly here, for the sake of diversity of individuals, their code, or genotype, will be changed slightly, and at a low rate of occurrence. Although it is something, dangerous and undesirable in nature, here it is something effective, and helpful, since with mating alone, it is possible to converge to a small group of genomes and alternate between them, and so this generally helps escaping local optima in performance problems.

3

Results

In this section, the finalized outcomes of the algorithm are presented and discussed. As it has been explained in chapter 2, an evolutionary algorithm can take various forms. This particular one incorporates mutation and mating for the part of variation.

Figure 3.1, displays the outcome of the algorithm with the best possible set of parameters set. Specifically, this evolution features a population of 40 genomes per generation, and a mutation rate of 0.01%. The two tools that indicate the performance are the maximum fitness and the average fitness achieved over each of the 30 generations. The maximum indicator quickly rises to 500 and maintains its position. This is not a given as it will be shown later on, but in this case, it shows that enough genomes achieve the goal of 500 from the initial generations, that, even though some of them might be mutated to produce a less effective individual, there will be other ones that are mated to produce a winner. On the other hand, the average indicator does not ever reach 500, but achieves a close result to that and is an overall more informative one of the generation as a whole. A clear evolvability can be seen, and especially in the first 10 generations, producing already an average result of [1]. In generation [2] of this version, the best performing genome of the entire assignment is seen, with an average fitness of [3]. The corresponding weights, biases, and general properties of the top genome are laid out in .

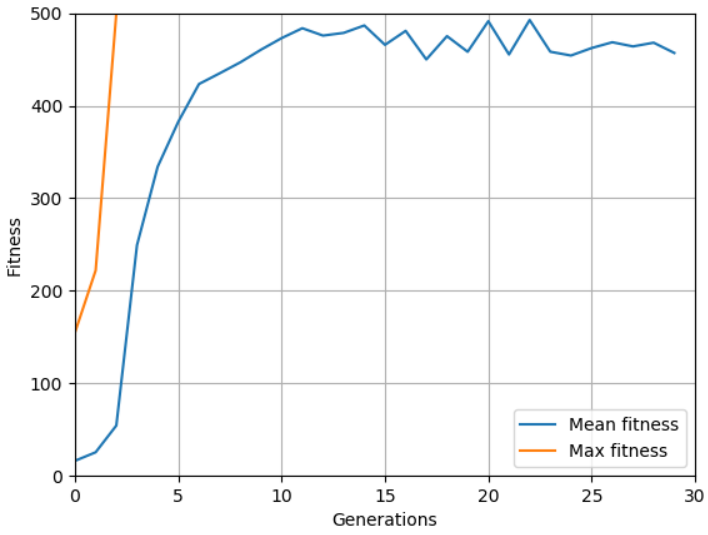


Figure 3.1: Performance with 40 generations and mutation rate 0.01%

In order to be able to assess the performance of this algorithm, one must certainly take into account the random generation in various parts of it. That is, it cannot be said that a mutation rate is more effective than another one, if at the same time, there has been generated a completely random initial population. For that to be avoided to an extent, the pseudo-randomness has to be exploited, and so seed control must be applied. In Appendix A,

[1] RESULT
[2] GEN
[3] AVG

it can be noticed that in numerous locations, the seed variable is either called or changed. It is changed because there must still be different random numbers, inside a for-loop for example, but it has to be the same ones every time the script is run. Therefore, whenever there is a random function executed, either in the definitions, or as part of an if statement, or in a loop, the seed is set right before, and changed right after. However, as mentioned, the randomness is avoided to an extent. In Figure 3.4, two similar, but not identical performance results are visible. The population of all 30 generations in these two is identical, after the pseudo-randomness is controlled, but the output is still somewhat different. This can be attributed to the embedded randomness of the CartPole environment. In line 66 of the source code created by Open AI Gym, it is said that "All observations are assigned a uniformly random value in ‘(-0.05, 0.05)‘"[2], meaning that the result of the same non-random script will not give exactly the same result every time it is run. This, however is not a problem, as the objective is still achieved. Even though this is the case, the differences due to randomness are still small enough to be able to observe the differences that result from parameter changes such as population size or mutation rate.

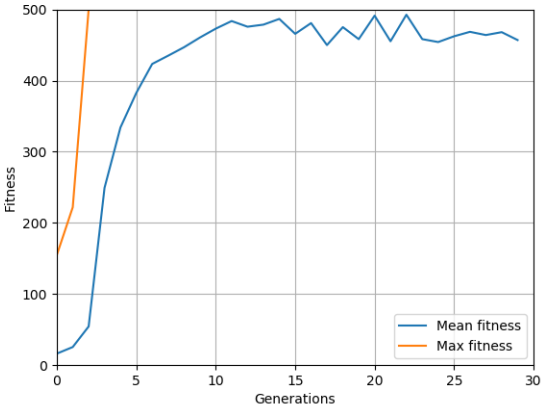


Figure 3.2

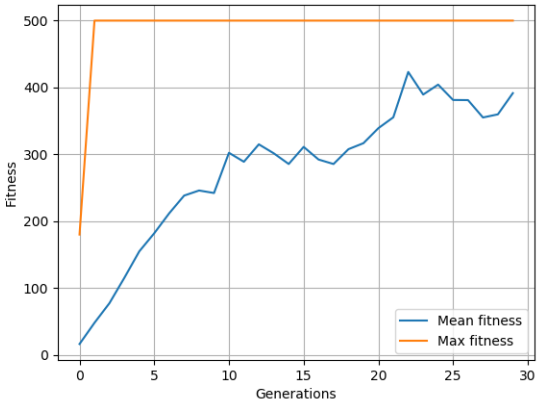


Figure 3.3

Figure 3.4: Performance of the algorithm under identical parameters

That being said, the algorithm has shown obvious evolvability, with the maximum fitness reaching the set ceiling of 500, from the fourth generation and the average fitness fluctuating about a steadily increasing average line. As it will be shown further, the parameters set for these results were the optimal ones. Th mutation rate was set to 0.01 and the population number to 30. For variation, a random genome is inserted into a new population until the desirable size is reached. However, it is still set to have a virtual fitness within the fitness range of the previous population, but not the top one. This way, every generation knocks the worst performing genomes out of the list, but not the best one. This is the reason why, once the max fitness of 500 is reached for a generation, it will usually stay that way for the following ones, as seen in Figure 3.3. Sometimes, however, for one of the generations, the max fitness might drop slightly, and then find its way bac to 500. This does not happen because the best performing genome was not selected, but because it was mutated.

4

Sensitivity

In this chapter, a conventional sensitivity analysis is displayed. Like most other optimization problems, it is a multi-dimensional one. As designed from the beginning, a simple result where an algorithm reaches the desired 500 steps was not the sole objective. It was programmed to be an algorithm with multiple control parameters to be tuned so that the optimal outcome is seen. The step and generation numbers, can indeed be varied too, but those are simply variables that are related to the results dimensions. With more steps to be reached, a successful algorithm would just get there in a few more generations, for example. Therefore, these were fixed to 500 and 30 respectively, so that all other results coming from the other parameter variations were compared under similar circumstances. Specifically, the population size and mutation rate are the ones that were manipulated.

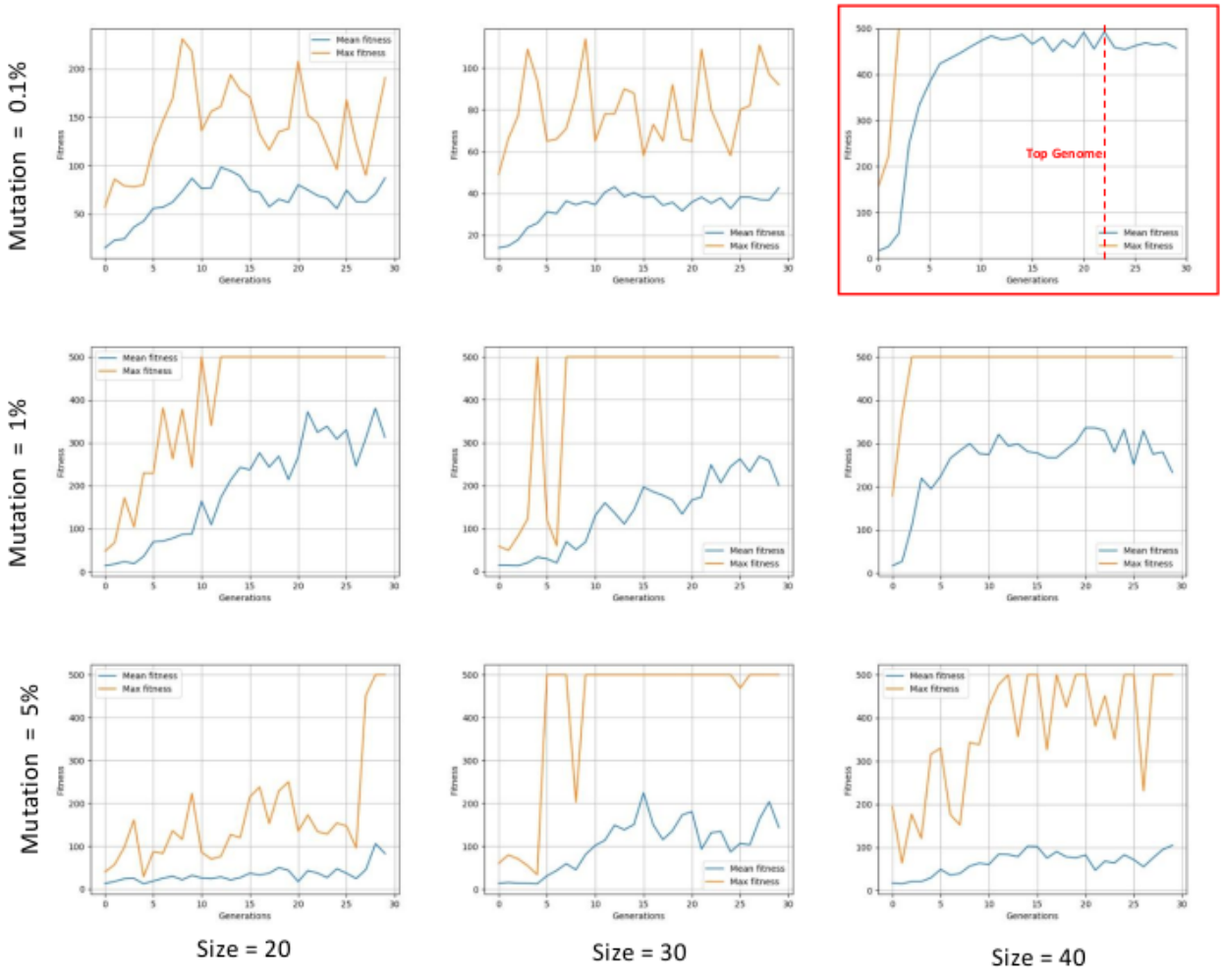


Figure 4.1: Sensitivity Analysis over mutation and population size

It is obvious from Figure 4.1, that the algorithm performance indeed fluctuates both with regards to mutation and to population size. The population number is more of an intuitive factor for sensitivity as one would easily imagine that a change from 20 to 30 in every generation is much more of an effect than a change from 1% to 0.1% in mutation rate. However, the array can show that for all 3 columns, there is significant reduction in results value when moving from 1% to 5%, and similarly from 1% to 0.1% for the cases of 20 and 30 individuals. Note the axis ranges in the first two graphs of the top row.

It is indeed a counter-intuitive result to have the best genome in the top right graph. Exempting that one, the best results relative to mutation only is seen in the middle row (1%) and the best result relative to population size is seen in the first column. These are indications of extreme sensitivity, and potentially a local optimum. That is, when it comes to the other classical control problems, the objectives are momentum-based, so there is an optimal solution to achieve the goal. Here, the goal is to keep the pole standing for a timeframe set by the programmer, and that can be as long as one wants, and even then, the cart might be doing it in a moving pattern that is not the ideal one. So undoubtedly there is a large field for discovery available concerning the CartPole problem.

5

Conclusion

This project has been an exploration of the potential of Bio-Inspired Intelligence and Learning methods, and specifically of Evolutionary Robotics while applying it to one of the classical control problems of Open AI Gym, the CartPole problem. While many different variants of Evolutionary Algorithms exist, the one created here is a collective inspiration of other ones that have showed success in classical control. As a controller of the cart balancing the pole above it, a basic Neural Network agent is chosen, with 2 hidden layers of 8 nodes each. Machine Learning problems are usually a case in which a lot of blind exploration occurs, until some acceptable result is found, after which the structure is going to be used as base for optimization. Similarly now, many structures were tried, both in the Neural Network part, when trying different size combinations for the hidden layers, but also for the evolutionary part, when trying different methods of creating a new generation, as seen from other algorithms. At last, however, the method presented was the one that deemed the challenge of controlling the cart to balance the vertical pole for 500 steps, possible.

It was pointed out in chapter 3, that a pseudo-randomness is still present in the results, even after controlling the seed in the evolution of the algorithm. It is due to the CartPole environment of Open AI Gym. Other classical control environments there do not have this feature, as objects there begin at a stationary state. Therefore, if the same algorithm from this project was applied to them, after adjustment of the observation and action spaces, the results would have been the same every time it was run. Therefore, it can be recommended that for future work, and for repetitively consistent results, that a similar environment to the one of Open AI Gym be created, in which the vertical pole starts from the same small angle in every observation. This is possible, since these environments are open-source ones, and can be modified, but it was out of the scope of this assignment.

On the other hand, however, the randomness of the population generation and the neural network setup have been controlled successfully, so that a reliable sensitivity analysis could be formulated, since the difference in results due to environment randomness, was small enough so that difference in results due to parameter variations was visible. Varying the mutation and population number, provided some significant insight in the abrupt behavior of the algorithm. While having a single set of genomes perform at an average of 493 and maximum 500 fitness is near perfect. But the overall view of the sensitivity field has been rather unpredictable. For this, it can be recommended that a more complicated reward system is set up in which, not only a reward is given for every step keeping the pole up, but a displacement-based one, so that the cart does not swing from left to right like it was observed to do at instances.

The last recommendation that can be given for future work, would be to investigate the layout of the hidden layers of the neural network. it is vital that they are not treated as a black box entity producing results at times. The path from observation to action must be looked at, divided into sub observations of some sort, and make a layer attacking each sub-section with a number of sub-tasks. Then, the results of this assignment can be taken from Appendix B, and the problem might potentially be solved even more effectively.

Bibliography

[1] Joemar J. Bancifra, Tarlac State University, Tarlac City, Philippines, and https://orcid.org/0000-0003-0641-1305. Supervisory practices of department heads and teachers' performance: Towards a proposed enhancement program. APJAET - Journal ay Asia Pacific Journal of Advanced Education and Technology, pages 25–33, September 2022.

[2] OpenAI. Gym/cartpole.py at master ů openai/gym, Aug 2022. URL `https://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py`.

A

Code

```
import gym
from matplotlib import pyplot as plt
import random, bisect, timeit
import numpy as np

# Start timer
start = timeit.default_timer()

# Set initial seed
seed=10

# Agent
class NN_agent :

    def __init__(self, n_nodes):

        global seed

        self.weights = []
        self.biases = []

        self.fitness = 0
        self.nodes = n_nodes

        for i in range(len(n_nodes) - 1):
            np.random.seed(seed)
            self.weights.append(np.random.uniform(low=-1, high=1, size=(n_nodes[i], n_nodes[i+1])).tolist())
            seed+=1
            np.random.seed(seed)
            self.biases.append(np.random.uniform(low=-1, high=1, size=(n_nodes[i+1])).tolist())
            seed+=1

    #evaluation
    def evaluate(self, input):

        output = input
        for i in range(len(self.nodes)-1):
            output = np.reshape(np.matmul(output, self.weights[i]) + self.biases[i], (self.nodes[i+1]))
        return np.argmax(sigmoid(output))

# Sigmoid activation function
def sigmoid(x):
    return 1.0/(1.0 + np.exp(-x))

# Evolutionary algorithm
class Population :

    def __init__(self, n_individuals, p_mut, n_nodes):

        self.n_nodes = n_nodes
        self.population = [NN_agent(n_nodes) for i in range(n_individuals)]
        self.pop_size = n_individuals
        self.m_rate = p_mut
```

```
    '''
    Selection: Two randomly-selected parents are chosen, before being passed into createOffspring().
    This is done in a while-loop, until the population size is reached.

    '''
    def newGen(self):

        global seed

        total_fitness = [0]
        new_gen = []
        for i in range(len(self.population)):
            total_fitness.append(total_fitness[i]+self.population[i].fitness)

        while(len(new_gen) < self.pop_size):
            random.seed(seed)
            r1 = random.uniform(0, total_fitness[len(total_fitness)-1] )
            seed+=1
            random.seed(seed)
            r2 = random.uniform(0, total_fitness[len(total_fitness)-1] )
            seed+=1
            nn1 = self.population[bisect.bisect_right(total_fitness, r1)-1]
            nn2 = self.population[bisect.bisect_right(total_fitness, r2)-1]
            new_gen.append(self.createOffspring(nn1, nn2))
        self.population.clear()
        self.population = new_gen

    '''
    Variation: Two for-loops, one for weights, one for biases. In each, first they are passed
    through an if-probability for mutation, or else mating.

    '''

    def createOffspring(self, parent1, parent2):

        global seed

        offspring = NN_agent(self.n_nodes)

        for i in range(len(offspring.weights)):
            for j in range(len(offspring.weights[i])):
                for k in range(len(offspring.weights[i][j])):
                    random.seed(seed)
                    if random.random() < self.m_rate:
                        seed+=1
                        random.seed(seed)
                        offspring.weights[i][j][k] = random.uniform(-1, 1)
                        seed+=1
                    else:
                        offspring.weights[i][j][k] = (parent1.weights[i][j][k] + parent2.weights[i][
    j][k])/2.0
                        seed+=1

        for i in range(len(offspring.biases)):
            for j in range(len(offspring.biases[i])):
                random.seed(seed)
                if random.random() < self.m_rate:
                    seed+=1
                    random.seed(seed)
                    offspring.biases[i][j] = random.uniform(-1, 1)
                    seed+=1
                else:
                    offspring.biases[i][j] = (parent1.biases[i][j] + parent2.biases[i][j])/2.0
                    seed+=1

        return offspring

# Parameters
STEPS = 500
GENS = 30
POPULATION = 40
MUTATION = 0.001

# Set up environment, agent and initial population
env = gym.make('CartPole-v1')
```

```
observation = env.reset()
dim_in = env.observation_space.shape[0]
dim_out = env.action_space.n
pop = Population(POPULATION, MUTATION, [dim_in, 8, 8, dim_out])

# Plot lists
MAXFIT = []
AVGFIT = []

# Assign initial genome as top until the real top one is found
TopGenome = pop.population[0]

# Algorithm loop
for gen in range(GENS):

    max = 0
    avg = 0

    # Generation loop
    for genome in pop.population:
        Reward = 0

        # Genome loop
        for step in range(STEPS):
            env.render()
            action = genome.evaluate(observation)
            observation, reward, done, info = env.step(action)
            Reward += reward
            if done:
                observation = env.reset()
                break

        genome.fitness = Reward
        avg += genome.fitness

        if genome.fitness > max :
            max = genome.fitness


    avg/=pop.pop_size

    print("Generation : %3d |  Avg Fitness : %4.0f  |  Max Fitness : %4.0f  " % (gen+1, avg, max))

    MAXFIT.append(max)
    AVGFIT.append(avg)

    # Top Genome check
    if TopGenome.fitness < pop.population[np.argmax(AVGFIT)].fitness:
        TopGenome = pop.population[np.argmax(AVGFIT)]
    pop.newGen()

env.close()

# Plotting
plt.figure();
plt.plot(range(GENS), AVGFIT)
plt.plot(range(GENS), MAXFIT)
plt.xlabel('Generations')
plt.ylabel('Fitness')
plt.legend(['Mean fitness', 'Max fitness'])
plt.xlim([0,30])
plt.ylim([0,500])
plt.grid()
plt.show()

# Top Genome
print(TopGenome.weights)
print(TopGenome.biases)

#Timer
stop = timeit.default_timer()
print('Time: ', stop - start)
```

Listing A.1: EA.py

B

Top Genome

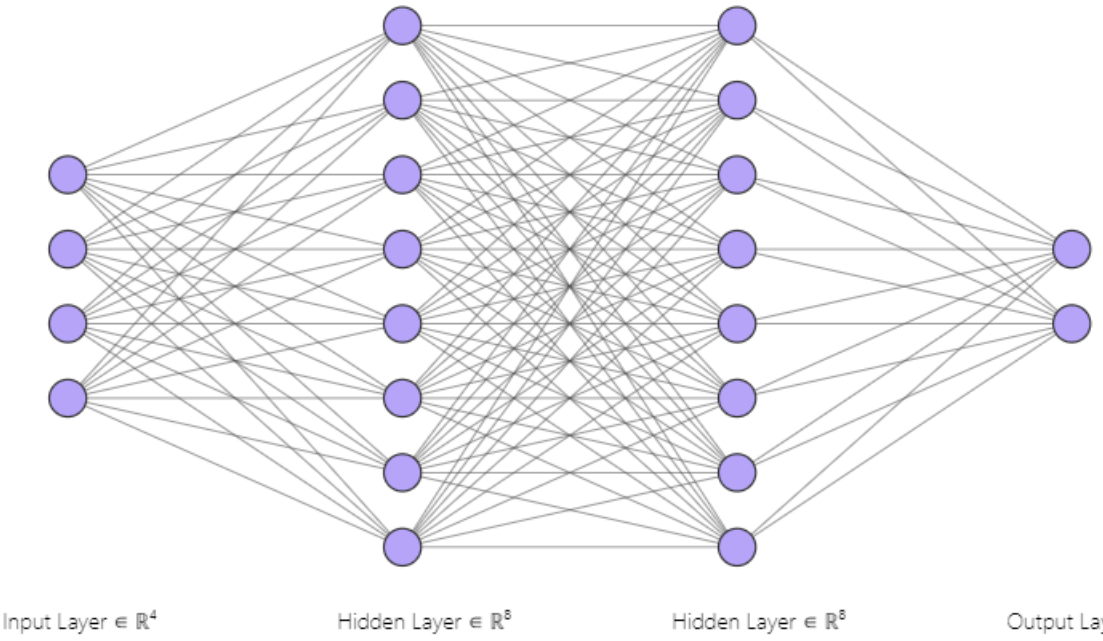


Figure B.1: Network architecture

```
# Edge Set 1
[[[0.7899145225752489, -0.310346393992303, 0.3701160968630164, 0.4271565485568958,
    -0.3665830192005328, 0.8729535392130019, -0.05276471734736954, -0.20713746411558076],
  [-0.35189131122854933, -0.6479485367238209, 0.1278004682531848, 0.47490231603740396,
    0.0421966831817584, -0.881969853424043, 0.6460321044567969, 0.224068628407349 42],
  [-0.7785980574933138, 0.24444803176803442, -0.3851441061528542, -0.4518530205629654,
    0.016517042197178755, 0.9585845446296206, -0.7478065244925494, 0.2871779554816003],
  [0.5055857148539518, -0.5560715844393955, -0.3196193022330691, 0.7712076959330867,
    -0.9246045052198519, -0.5863057638351339, -0.25562438426188994, -0.9579043707068096]],
# Edge Set 2
 [[-0.15237837706094437, 0.083485533804313, 0.37710979543547385, -0.3078339496928817,
    -0.12194199153158225, 0.17809817651045812, 0.307324345875965 13, -0.7794683286780997],
  [-0.7141946900640985, 0.7483547133353952, 0.12790022461061368, -0.8618474734218442,
    -0.37616132115559964, 0.2255612463197647, 0.3080829605341622, 0.8038346322531482],
  [-0.8138326755891261, 0.8488994453396077, -0.6357977438513804, -0.9019221636459929,
    -0.49451165600204505, -0.6346269826780928, -0.11777281386531868, 0.4767940363071894],
  [-0.4166710205030597, -0.4333045627659764, -0.740246623913245, 0.29141727794323335,
    -0.8908860558218257, -0.4369136693818243, -0.31317896305136417, -0.56684282178399],
  [-0.7436227897410816, -0.16954849480417922, 0.21443386404191367, -0.03583421211865767,
    -0.9887378248731638, -0.04670633459088647, -0.8116643472513227, -0.05216229095860703],
  [-0.2104849589622979, -0.45487377102604376, 0.3839812163176979, 0.25750016961743283,
    0.08149246563999468, 0.8491087144753189, -0.42876999078211386, 0.14657327699289313],
```

```
  [-0.9316513841919429, 0.1517806142196254, -0.16448342470471666, 0.8527257167970814,
    0.32533546377169986, 0.6959482738019833, -0.6708319524612909, 0.015085515957484086],
  [0.6232239562521102, -0.7982817933010373, 0.15957720797891217, -0.8143614197510185,
    -0.632495965072875, 0.9807747102688182, -0.5891467247586089, -0.26546011552517124]],
# Edge Set 3
 [[0.26199996521153235, 0.7646086395721123],
  [-0.482896678719678, -0.37175523975956115],
  [0.8717685729025171, 0.4900332658245832],
  [-0.9868935499144418, -0.9604676978508375],
  [0.3730390468331892, 0.9113564235926939],
  [0.002979741412528547, 0.27065244092249174],
  [0.7490056133618779, 0.6922345896917412],
  [0.47177758712576745, 0.31613228174386165]]]
```

Listing B.1: Weights

```
# Edge Set 1
[[-0.7450570896161752, 0.16113607880485992, 0.28453631891521947, -0.09587126445220684,
    -0.7656474766916621, 0.6509698605079595, 0.2648444900528355, 0.4294339405974468],
# Edge Set 2
 [-0.8059500010006504, 0.477114414062217, 0.3575676273742803, -0.504915322169442,
    0.5589234804087511, -0.05907311721816133, -0.25626845798316733, 0.3391807145659296],
# Edge Set 3
 [0.56298808165764, 0.16841823217108653]]
```

Listing B.2: Biases

```
Mutation rate: 0.001
Population size: 40
Generation number: 22
Fitness: 493
Runtime: 10231.56s
```

Listing B.3: Properties