Evoman: Training the player to beat the enemy with evolutionary algorithms

Task I: Specialist agent, Group 33, Evolutionary Computing 2022

Barbara Brocades Zaalberg, 2719394
Universiteit van Amsterdam
Amsterdam, Netherlands
barara.bz@outlook.com

Eva Biesot, 2777979
Vrije Universiteit Amsterdam
Amsterdam, Netherlands
embiesot@hotmail.com

Godelieve ten Have, 2748616
Vrije Universiteit Amsterdam
Amsterdam, Netherlands
gmmtenhave@gmail.com

Patrick de Jong, 2703867
Vrije Universiteit Amsterdam
Amstelveen, Netherlands
p9.de.jong@student.vu.nl

Stijn Kok, 2749616
Vrije Universiteit Amsterdam
Leiden, Netherlands
stijnkok123@gmail.com

KEYWORDS

Evolutionary Algorithm, fitness, population, mutation, cross-over, recombination

ACM Reference Format:
Barbara Brocades Zaalberg, 2719394, Eva Biesot, 2777979, Godelieve ten Have, 2748616, Patrick de Jong, 2703867, and Stijn Kok, 2749616. 2022. Evoman: Training the player to beat the enemy with evolutionary algorithms: Task I: Specialist agent, Group 33, Evolutionary Computing 2022. In *Proceedings of Evolutionary Computing 2022 (EC 2022)*. Amsterdam, Netherlands, 5 pages. https://doi.org/10.1109/CEC.2016.7743938

EC 2022, Sep 5th–Oct 28th, 2022, Amsterdam, Netherlands
© 2022 Association for Computing Machinery.
ACM ISBN 978-3-662-44873-1…$35.00
https://doi.org/10.1109/CEC.2016.7743938

1 INTRODUCTION

Biological processes are a significant source of inspiration in problem solving and computer science[5]. One such biological process is evolution. In evolutionary computing, the biological processes and concepts of evolution are translated into problem solving algorithms that evolve by themselves to find solutions to specified computational problems. In these problems, scoring functions like fitness are evaluated and optimized through multiple generations of parent solutions and offspring. Evolution in populations is generally simulated by using functions to mimic biological processes, such as: *Mutation, Cross-over* and *Selection.* During the selection process, two or more individuals are chosen to create offspring, while mutation and cross-over are used to introduce random, possibly new, solutions in the offspring.

Evoman is a game playing framework copying the original Megaman game, designed by De Araujo et. al[4], that can be used to test evolutionary algorithms (EAs). The game consists of a player agent and an enemy agent which both have a limited set of actions (namely move left, move right, jump, release jump and shoot) that they can take to interact with the environment for the purpose of winning the game. A player wins the game, thus ending it, when the other player is out of energy. To learn the sequence of actions necessary to beat his opponent, our player will use a neuroevolution algorithm [7]. A neuroevolution algorithm is a artificial neural network where the networks structure, parameters and/or rules are determined by an EA. In our case, the network structure is fixed, but the weights between nodes will be optimized using two variations of an EA. The variation will be in the way that the algorithm selects the offspring for the new population. This results in the following research question:

What is the difference in performance of a Neuroevolution algorithm when applying a steady-state model compared to a generational model during the selection phase in the Evoman framework?

How well a strategy found by a player agent performs, is measured by its fitness. This fitness is expressed by the amount of energy of both the enemy and the player, together with the amount of time to reach the end of the game, see 2.1. The objective function of an EA for this game would be to maximize the players energy while minimizing that of the enemy while also minimizing the time to end the game.

It's unclear from literature which of the two selection methods is empirically 'better' than the other while optimizing neural networks[1][8]. For this reason, this paper aims to test which of the two methods proves superior within this framework, using neural networks to control a gameplaying agent. As literature states [6], we do hypothesize that a steady-state algorithm will reach convergence more steadily, but slower, and that a generational algorithm will be faster, but more oscillated.

In the next section the framework will be discussed, and the difference between the algorithms (steady-state vs generational) will be explained. In section 3 the results of the experiments are shown and finally, in section 4 the results will be discussed and the research question will be answered.

2 METHODS

2.1 Framework

The framework which is used to run the game can be downloaded from the GitHub provided by De Araujo et. al[3]. The player is controlled by a neural network, which is also provided in this framework by "`democontroller.py`". For all general network architecture choices, see the paper by Araujo and de Franca[4]. The weights of the network will evolve under the process of mutation, cross-over and selection, while being evaluated using the following fitness function:

$$fitness = (100 - e)^{\gamma} - (100 - p)^{\beta} - ((\sum_{i=1}^{(\vec{t})} 100 - p_i)/t)^{\alpha} \quad (1)$$

With: e : energy measure of the enemy and p : the energy measure of the player, both in the range from 0 to 100 and t : the number of time steps that was necessary to end the game. $\gamma = 1$, $\beta = 2$ and $\alpha = 2$ are importance weights. To introduce variation between a certain number of generations, 25% of the population with the lowest fitness is replaced by random new individuals. This is referred to as doomsday.

The effectiveness of both variations of the algorithm was tested on 3 out of 8 available enemy players. Both variations will be trained on these enemies individually, thus creating a 'specialist' agent for that certain enemy. The enemies that have been chosen are: *Heatman*, *Crashman* and *Bubbleman.* Araujo et. al[4] played against these enemies with a neural network with a hidden layer of 10 nodes, resulting in fitness scores of 30, 0 and 56, respectively. Thus, these 3 enemies are relatively difficult to beat, which is why they are chosen for this experiment. Although their implementation had a good score towards *Heatman*, most algorithms had a lower score towards that enemy.

The weights of the neural network that are evaluated in this EA are seen as the individuals controlling the player, represented as a vector. The neural networks in the first generation are initialized with a random set of weights.

The population is defined as a generation of solutions (I.e chromosomes), being the weights that define the neural network to control the player.

To select the parents, a tournament selection with size 2 is used, meaning two individuals are randomly chosen from the population and the fittest one will become a parent. Two tournaments are used to get two parents which will generate two children. Tournament size 2 is chosen, to keep the selection pressure low to maintain diversity in the population[2].

Variety in the offspring of the parents is needed, otherwise evolution would not be happening and the algorithm would be stuck in one type of solution. Hence, the children of the parents need to be different than an exact copy of their parents, which is why variation operators like crossover and mutation are applied. There are several option for mutations. In De Araujo et. al[3], they apply mutation by sampling from a uniform distribution with the upper and lower bound the same as the problem domain. They do not apply mutation to all chromosomes, they kept the mutation probability at 20%, thus in 20% of the cases a mutation will take place. This is called a uniform mutation. To vary from De Araujo et. al, nonuniform mutation is used in this report. There still is a mutation rate during nonuniform mutation that makes sure only some chromosomes will be mutated. But instead of replacing the whole value during mutation, a value drawn from a Gaussian distribution with mean zero and a self chosen standard deviation (σ), is added to the original value of the mutated weight. The standard deviation influences the probability of drawing very large values and should be chosen carefully. Based on the book of Eiben et. al[5], a mutation rate of 20% is chosen. Since the value of the weights are bounded between [-1,1] it is not helpful to set sigma too high, causing to often end up in one of the bounds. Two thirds of the samples drawn will be between [-σ,+σ][5]. Based on this and runs with trial and error, a σ of 0.1 is used for the remainder of this paper.

For survivor selection, Tournament selection with tournament size 5 is used. Size 5 is used because for selection, the selection pressure should be higher to ensure that the best solutions are kept. Because the goal is to compare a generational model with a steady-state model, two types of survivor selections are used. For one, tournament selection is used to select survivors from a pool of both parents and offspring. For the other, tournament selection is used to select survivors out of a pool of just the offspring. For the second model, every individual only lives one generation, whereas in the first model individuals can survive for multiple generations.

2.2 Algorithms

2.2.1 Steady-state. The steady-state selection method uses two assigned parents in the population to create new offspring, which it then replaces the bad solutions with, at every cycle (generation). So from the population consisting of parents (100) and children (100), the next generation of 100 individuals is chosen based on tournament selection. This means the population is changing slowly but steadily.

2.2.2 Generational. In a generational algorithm, there will be 200 children afterwhich all parents will be deleted from the population. After this, 100 individuals are selected for the next generation based on tournament selection again.

2.3 Experimental set-up

For both the steady-state and generational algorithms, the programme is ran 10 times against 3 different enemies. This results in 6 variations and 60 runs where in each run, the EA loops for 30 generations. Every generation consists of a population of size 100. For each variation, the fitness is plotted over generations, showing the mean fitness with standard deviation. The same is done for the best individual out of each generation (see Figures 1, 2 and 3). A boxplot is also made showing the variation of the best individuals over all 10 runs (see Figure 4). This is done by taking the overall best individual out of each run, and running that testing that individual for 5 additional times. With these, take the mean, and plot the means out of all 10 runs into a boxplot per enenmy and per EA. To test whether the algorithms have a difference in fitness, a T-test was used ('scipy.stats.ttest_ind') to determine any significance.

3 RESULTS

In this section, the evolutions and strategies are described which have evolved out of the EA's for the three chosen enemies. In all generational plots, a dip is obeserved in generation 10 and 20, due to the *doomsday* function.

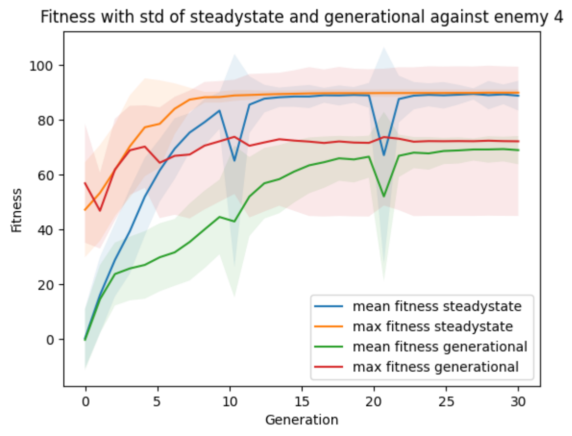

Figure 1: Results of steady state and generational algorithms for enemy 4 (*Heatman*), with standard deviations. Graph shows fitness on the X-axis, generations on Y-axis.

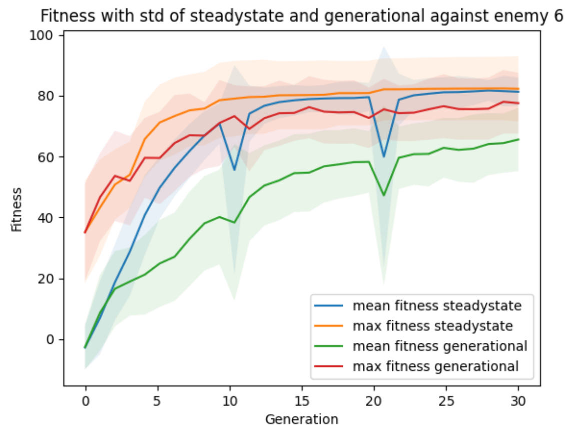

Figure 2: Results of steady state and generational algorithms for enemy 6 (*Crashman*), with standard deviations. Graph shows fitness on the X-axis, generations on Y-axis.

3.1 Evolutions and strategies

For *Heatman*, also called enemy 4 (see Figure 1), the mean of the steady state algorithm quickly overtakes the generational best solutions in fitness, whilst maintaining a relatively low standard deviation. The mean scores were significantly different ($p = 0.0298$). The best solutions produced by the two different algorithms take a different approach. In the steady state solution, the player jumps very low and shoots to the right continuously in a fixed spot. If the enemy comes close, the player jumps high and keeps jumping and shooting to the right. The players beats the enemy with 89 remaining life points with a time of 1038. In the generational solution the player keeps moving towards the enemy until it is a specific distance away, while constantly jumping high and shooting. The player ends up with 92 life points, beating the enemy with a time of 982.

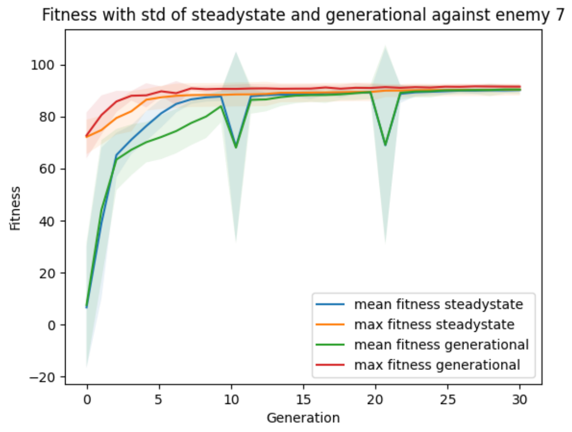

Figure 3: Results of steady state and generational algorithms for enemy 7 (*Bubbleman*), with standard deviations. Graph shows fitness on the X-axis, generations on Y-axis.

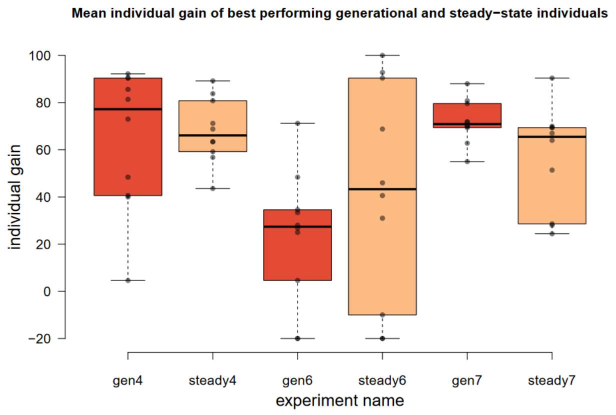

Figure 4: The mean individual gain of the best individuals across all generations, sampled 10 times per condition. Every data point is a mean of 5 simulations using one individual. The red boxes show runs that used a generational method, the orange boxes show runs that used a steady state method.

For *Crashman*, also called enemy 6 (see Figure 2), the patterns look almost the same as for enemy 4. The steady state scores slightly lower compared to enemy 4, and the best solutions for generational slightly higher. The steady state still outperforms generational in best solutions. The mean scores were not significantly different (*p = 0.0766*). However, the optimal strategy evolved by the two algorithms was very different. The steady state player stays mostly in the right corner. If the enemy jumps to the player, it avoids the dropped projectile and the enemy by jumping while shooting two aimed bullets. The player does not lose any life points this way, but takes a long time to beat the enemy (1548). The best strategy evolved by the generational algorithm consists of the player constantly jumping and shooting to the right in a specific spot. This results in the enemy not jumping over and barely touching the player. The player beats the enemy more quickly (738), but ends up with 71 life points.

For *Bubbleman*, also called enemy 7 (see Figure 3), both algorithms converge to almost the same fitness for the mean and best solutions. The mean scores were/were not significantly different (*p = 0.6827*). This is also reflected in the evolved playing strategy: the best solutions for both algorithms evolved a strategy that fired a few shots first, followed by a very high jump to stay at the same height as the enemy while it jumped, while constantly firing shots to do as much damage as possible. Both strategies resulted in a play time of 136, but in the steady state solution the player had more remaining life points (90, while the generational approach resulted in 88 life points).

The best solutions for both the steady state and generational algorithms tested against each enemy showed good solutions, generally with a fitness 90 (see Figure 4). There was no significant difference in individual gain between algorithms across different runs for enemy 4, 6 or 7 (*p=0.7508, p= 0.2945, p=0.0629*, respectively).

4 DISCUSSION AND CONCLUSION

As all combinations of the algorithms versus enemies have achieved a good fitness of higher than zero, it can be concluded that the general framework of our algorithm works well. Furthermore, the enemies chosen were largely considered the hardest to beat, according to De Araujo et. al. The steady state algorithm beats the generational algorithm in terms of stability and speed to convergence. It quickly attains a slightly higher fitness over generational, although it depends on which enemy it's focusing on. With enemy 7, it seems to be easier to evolve a general strategy to beat it, despite De Araujo et. al describing trouble with the effect on the jump action. For enemies 4 and 6, the algorithms evolve at a comparable pace, while ending up in slightly different score values, with 4 having a significant difference between algorithms.

Despite the generational algorithm being seemingly slower to converge compared to steady state, when comparing the scores of all best solutions it scores about the same as the steady state algorithms best scores.

Since our paper focused solely on survival selection, different methods for mutation are studied to a lesser extent. Further research could be done to look for the optimal sigma and mutation rate, or whether applying a different technique like self-adaptive mutation would be beneficial. In the current approach, we are also overfitting on the starting positions, as the enemy is static and the game is thus deterministic. Adding stochasticity to the starting positions might result in a more general strategy against an enemy.

REFERENCES

[1] Peter J Angeline, Gregory M Saunders, and Jordan B Pollack. 1994. An evolutionary algorithm that constructs recurrent neural networks. *IEEE transactions on Neural Networks* 5, 1 (1994), 54–65.

[2] Thomas Back. 1994. Selective pressure in evolutionary algorithms: A characterization of selection mechanisms. In *Proceedings of the first IEEE conference on evolutionary computation. IEEE World Congress on Computational Intelligence*. IEEE, 57–62.

[3] Karina da Silva Miras de Araujo. 2019. evoman_framework. (Jul 2019). https://github.com/karinemiras/evoman_framework

[4] Karine da Silva Miras de Araujo and Fabrício Olivetti de Franca. 2016. Evolving a generalized strategy for an action-platformer video game framework. In *2016 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 1303–1310.

[5] AE Eiben and JE Smith. 2015. Evolutionary computing: The origins. In *Introduction to Evolutionary Computing*. Springer, 13–24.

[6] Parsa Esfahanian and Mohammad Akhavan. 2019. Gacnn: Training deep convolutional neural networks with genetic algorithm. *arXiv preprint arXiv:1909.13354* (2019).

[7] Dario Floreano, Peter Dürr, and Claudio Mattiussi. 2008. Neuroevolution: from architectures to learning. *Evolutionary intelligence* 1, 1 (2008), 47–62.

[8] J David Schaffer, Darrell Whitley, and Larry J Eshelman. 1992. Combinations of genetic algorithms and neural networks: A survey of the state of the art. In *[Proceedings] COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*. IEEE, 1–37.