Evolutionary Computing

Task 1: Specialist Agent

Group 23

Kleio Fragkedaki
Student number: 2729842

Matilda Knierim
Student number: 2700374

Carol Rameder
Student number: 2747982

Robin Stöhr
Student number: 2750340

1 INTRODUCTION

Evolutionary Algorithms (EA) are based on biological evolution, leading to the creation of an artificial population that evolves over several generations. In order to maximize the fitness of a population, the methods of selection, recombination, and mutation are used. Moreover, EAs are not reliant on function gradient information where optima can be found in a discrete, non-linear and non-deterministic process.

One crucial aspect in designing a high-quality EA is to increase population diversity with variation of chromosomes. Diversity ensures a sufficient novelty of the population over the time of generations, and therefore avoids local optima and premature convergence. However, increasing diversity might come at the cost of quality within a population due to negative changes in the chromosomes and a resulting lower fitness. Hence, a dynamic combination between high diversity that escapes local optima and low diversity that ensures progress by fine-tuning the solutions specifically tailored to individual chromosomes might give the best optimization results.

One mutation method that was found to dynamically change diversity is the self-adaptive mutation [5]. This method is adjusting the mutation specifically to the individual gene, leading to a tailored mutation for each chromosome. In comparison, the Gaussian mutation changes chromosomes by a random value from a draw of the Gaussian distribution around the mean 0 and a standard deviation of sigma. Based on the difference of these methods, this paper aims to answer the research question whether an EA with a self-adaptive mutation method can be seen as superior in terms of fitness performance in comparison to a standard EA using a standard Gaussian mutation. In order to investigate the research question, we compare two EAs with different mutation methods in the EvoMan environment [3] using the evolutionary computation framework DEAP [2].

EvoMan [3] is a video game computational intelligence framework in which an agent, equipped with an arm cannon, fights against different robots in several environments. Movements of the agent include moving to the right or left, shoot, jump, and interrupt the jump to fall to the ground. The enemy robots are equipped with different weapons and movements in order to hurt the agent. Both, agent and robot, start with 100 energy points that decrease once they are hit by the other's projectiles. The game ends when one of the energy levels turns zero. For our experiment, the agent, controlled by the EAs, fights against the enemies Airman, *Metalman*, and *Quickman*.

2 RELATED WORK

Self-adaption is defined according to Eiben [1] as the dynamic changes of parameters during evolution based on the inclusion of parameters within an genetic encoding that is exposed itself to evolutionary process and pressures. Teo [5] reported that their self-adaptive Gaussian-based mutation method dramatically outperformed a standard algorithm with a standard Gaussian mutation method. Accordingly, the self-adaptive mutation enabled the algorithm to escape local optima in a highly deceptive fitness landscape and hence successfully located global optimal solutions. To extend this finding to a different environmental set-up, this research aims to investigate the comparison between the self-adaptive mutation method and a standard Gaussian mutation method in the environment of the EvoMan framework [4].

3 ALGORITHM DESCRIPTION

Two EAs are used to train the AI agent that plays the game. The trainable parameters, the weights and the biases, of the single-layer Neural Network represent the genotype of each individual. Thus, each one of them represents a configuration of the model, that can be evaluated accordingly to its performance during a game. Both of our algorithms use the standard genetic operators for evolution. We use fitness proportionate selection for deciding which individuals are chosen for reproduction. Therefore, the optimal features are propagated to the next generation, most likely leading to an improvement.

Crossover represents the reproduction process, in which the genetic information of the parents is combined. The resulted offspring represent the population of the next generation. Mutation consists of random tweaks of the offspring, which provides the diversity of the population. By doing this, the search space of our problem is well explored.

Initially, every gene in the genotype of an individual from the first generation consists of a random float number from the [-1,1] interval. Therefore, the algorithm starts with no human knowledge about the game or environment.

In the main loop of the algorithm, for each new generation, we select the parents for breeding green using the tournament selection and then perform the multi-parent crossover for obtaining the offspring. To create diversity, one of the two mutation methods (self-adaptive and Gaussian) is applied to the offspring for each algorithm. We do not use the Survival Selection operator, as both used algorithms are based on a constant number of individuals in a population that produce the same number of offspring for the next generation. The results consist of the fitness value of the current

population, that will be used in the selection process of the next generation.
The quality of individuals is measured using the so-called *fitness*. In our setup, we are using the provided fitness function of the framework, defined as

$$fitness = 0.9 * (100 - e_e) + 0.1 * e_p - \log t \quad (1)$$

In the stated above equation, e_e represents the enemy's energy and e_p the player's energy, with values in the interval $e_e, e_p \in [[0, 100]]$. The parameter t denotes the number of time steps until the game is finished.

The *gain*, used for selecting the best individuals after creating all generations, is a simplified fitness functions, not taking into account any weights or the time. The equation goes as follows:

$$gain = e_p - e_e \quad (2)$$

4 EXPERIMENTAL SETUP

In this setup section, the individual mutation, crossover and selection algorithms used will be explained in more detail. The Gaussian mutation and the selection algorithm are taken from the DEAP framework [2]. We refer to the evolutionary algorithm using the Gaussian mutation as EA1 and the algorithm using the self-adaptive mutation as EA2.

4.1 Mutation Algorithm

Both of our algorithms use a population size of $p = 100$ and a generation size of $g = 30$. In accordance to our research question, the only specific method changed in our two EAs is the mutation method. The EvoMan framework provided us with a controller class calls *demo_controller*, which uses a neural network to control the agent. The weights of this neural network are the chromosomes of the individuals.

Algorithm 1: Gaussian Mutation

For each chromosome, a value is drawn using a Gaussian distribution, with the mean $\mu = 0.0$ and with a standard deviation $\sigma = 0.1$. This value is then added to the chromosome with a probability of indpb = 0.1. The mean of the mutation was to set to zero so that the mutation is able to alter the old chromosome value in both directions with an equal probability. With a standard deviation of 0.1 the random draw from the Gaussian distribution will have a standard deviation of $\sqrt{0.1} \approx 0.32$, which in the test runs together with a mutation chance of 10% led to the best results.

Algorithm 2: Uncorrelated mutation with n σ's

In the second mutation algorithm, each Individual has n additional parameters $\sigma_1 \dots \sigma_n$, one for each chromosome $x_1 \dots x_n$. In each mutation step, the sigma values are updated first, as seen in equation (3), and the corresponding chromosome x values are mutated in relation to their sigma values equation (4).

$$\sigma_i' = \sigma_i \cdot e^{\tau' \cdot N(0,1) + \tau \cdot N_i(0,1)} \quad (3)$$

$$x_i' = x_i + \sigma_i' \cdot N_i(0, 1) \quad (4)$$

The constant parameters τ and τ' of equation (3) represent learning rates, with τ being a coordinate wise learning rate and τ' a general one. The values are set to:

$$\tau' = 1/\sqrt{2n}$$

$$\tau = 1/\sqrt{2\sqrt{n}}$$

$N(0, 1)$ represents a draw from the normal distribution around the center 0 and a standard deviation of 1. Every sigma value has a lower bound of $\epsilon_0 = 0.05$, so that the minimum *expected* change of the old chromosome is at least 0.05 in either direction. We also use those bound values for the random initialization of the sigma values, with each value being randomly chosen form the interval $\sigma_i \in [\epsilon_0, 2\epsilon_0]$. The independent sigma parameters are passed on to the offspring the same way the chromosomes are.

4.2 Crossover

The crossover used in both EAs is the *Multi-parent uniform crossover*, with four parents creating four offspring. In this uniform crossover, four of the selected parents are taken. For each child, each chromosome is copied from one of the four parents with an equal probability. The number of parents is set to four, to keep balance between the integrity and the diversity of the chromosomes.

4.3 Selection

For the selection of the mating parents, a *Tournament Selection* with a size of tournsize = 20 is used. In this selection, 20 out of the 100 individuals are selected at random. The best individual based on the evaluated fitness value is then chosen to be one of the mating parents. With a tournament size of 20, we aim to achieve a favorable mix of diversity and excellence. A too large tournament size yields the risk that the best performing individuals will be selected too often, which could damage the diversity. On the other hand, a too small tournament size might increase the randomness which individuals will be selected for mating. This could lead to good individuals being ruled out and bad ones creating the new offspring. The test runs indicate that 20 is a good middle ground with a general population size of 100.

5 EXPERIMENTAL RESULTS

Two experiments were conducted using the different EAs mentioned in the previous section. The created EAs were tested in the EvoMan Framework [3] against three different enemies, the *Airman*, the *Metalman*, and the *Quickman*.

Each EA runs 10 times for each enemy, while the best individual of every run is being tested 5 times. The best individual is selected based on the highest individual gain (equation 2) among all generations.

In order to evaluate the EAs, two kinds of plots were created. The line plots represent the average of mean and maximum fitness (equation 1) of each generation and their standard deviations, while the box plots depict the individual gain of the best individuals arising from each *training_run*. Each individual is tested for accuracy

reasons 5 times, and the gain represented in the plot is the mean of those five *testing* runs.

The three Figures depict the fitness and gain values of the agent that is controlled by the two EAs in the fight against three different enemies. In the first fight, depicted in **Figure 1**, it can be seen that the average fitness of EA2 is slightly poorer compared to EA1. Moreover, the individual gain of EA2 has a higher variance than EA1. In the fight against the Metalman enemy, the fitness performance did not differ significantly between both algorithms. As in the fight depicted in **Figure 2**, the individual gain of EA1 is much more concentrated than for EA2. The fight against Quickman, **Figure 3**, shows the greatest differences between the fitness values. The EA2 shows a worse fitness performance than the EA1 with a higher standard deviation. Also, the box plot emphasizes a high range in individual gain for EA2 in comparison to EA1. Furthermore, some agents controlled by EA2 had a negative individual gain.

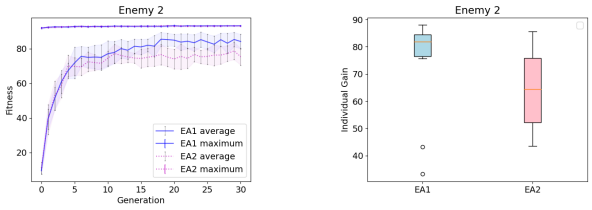

Figure 1: Airman Enemy

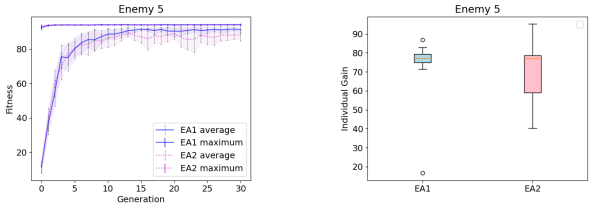

Figure 2: Metalman Enemy

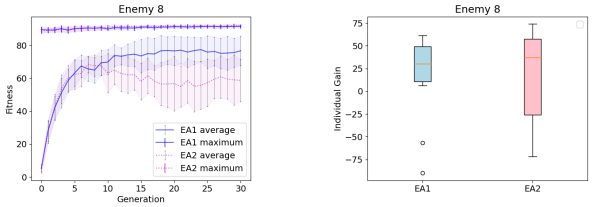

Figure 3: Quickman Enemy

6 ANALYSIS AND DISCUSSION

Based on the experimental results, the average fitness value of both algorithms rises rapidly during the first five generations when facing any of the enemies, indicating that the evolution process works successfully.

Regarding the fitness performance, the main difference between the two EAs is the average fitness obtained over the generations. EA1 achieves higher fitness on average than EA2 regardless of the chosen enemy. Additionally, while testing the best individual of each training run, the average individual gain for the EA1 is consistently high and there is no large variation, whilst the EA2 fluctuates between a high range of different values. This means that the best individual of EA2 is not performing coherently in every experiment. These surprising findings emphasize that EA2 is not performing superior in comparison to EA1.

Furthermore, the plots show that high maximum fitness values are reached even from the first generation and remain constant throughout the evolution process. High maximum values in early generations could possibly be explained by a low performance of the enemy or the high population size that is used.

Moreover, the standard deviation of both EA fitness values is considerably higher for the Quickman enemy comparing to the other two enemies. A possible reason for this could be the structure of the environment for each game, considering that the Airman and Metalman enemy have a plain ground on their map, while the ground of Quickman yields some obstacles. These obstacles bring diversity that could slow down the evolution process and potentially cause a high difference between highly fit individuals and weaker ones. The similar development of both algorithms is an argument that the environment causes the high variations and not the differing mutation methods.

One possible reason for the weaker results of EA2 might be a missing survival selection. Since the self-adaptive mutation relies on the combination of all sigma values of an individual to achieve the best results, a multi-parent crossover might lead to a disconnection of coherent sigmas. Moreover, since the parents are completely replaced with the offspring, groups of well-functioning sigmas might get destroyed in the process of recombination and lost in the genetic pool. Therefore, for future research, it might be interesting to investigate how a different crossover method and an inclusion of survival selection effects the comparison between EAs using self-adaptive mutation and Gaussian mutation.

7 CONCLUSION

This research investigated the difference of fitness and gain performance between an EA with a self-adaptive mutation method and an EA with a Gaussian mutation method. Taken together, we can conclude that the self-adaptive mutation in our set-up is not superior, and in some cases even weaker than the Gaussian mutation. As discussed, this interesting finding could be based on the multi-parent crossover method and missing survival selection.

8 GROUP WORK SET-UP

We split our work into two sections, 1) setting up the algorithm and code and 2) writing the report. We set up the groundwork for our EA together and then split the tasks for the fine-tuning. Robin coded the multi-parent crossover and self-adaptive mutation method, Kleio coded the line and box plots, and Carol and Matilda fine-tuned the specific parameters. For the report, we divided the writing evenly in four parts.

REFERENCES

[1] A. E. Eiben and James E. Smith. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer, 2003, pp. 1–300. ISBN: 978-3-662-05094-1.

[2] Félix-Antoine Fortin et al. "DEAP: Evolutionary Algorithms Made Easy". In: *Journal of Machine Learning Research* 13 (July 2012), pp. 2171–2175.

[3] Fabricio Olivetti de Franca et al. *EvoMan: Game-playing Competition*. 2020. arXiv: 1912.10445 [cs.AI].

[4] Karine da Silva Miras de Araújo and F. O. D. França. "Evolving a generalized strategy for an action-platformer video game framework". In: *CEC*. 2016.

[5] J. Teo. "Self-adaptive mutation for enhancing evolutionary search in real-coded genetic algorithms". In: *2006 International Conference on Computing & Informatics* (2006), pp. 1–6.