Parameter Control for Evolving Video Game Agents

Group 42 Evolutionary Computing, Task II: Generalist Agent

October 12, 2020

Xiaojin Gu
2654391

Eui Yeon Jang
2569512

Tim van Loenhout
2525199

Lando de Weerdt
2652415

1 INTRODUCTION

Video and strategy games provide interesting challenges for developing artificial intelligence agents, requiring them to learn a variety of skills for a successful game-play. Whereas past approaches have focused on developing scripted agents [11], more and more machine learning approaches are taken to develop agents. While there is much attention on developing deep learning game-playing agents, such as deep neural networks playing Atari games [14], it has been demonstrated that agents developed using evolutionary algorithms (EAs) can achieve competitive performance [15].

EAs consist of various components and parameters. They are also dynamic in nature, where different parameter values may be optimal during different stages of the process [7, 8]. Parameter control modifies the value of the parameter throughout the evolutionary process, mitigating the possible sub-optimal performance brought on by static parameter values.

The goal of this research is to investigate how different parameter control methods on the mutation operator of an EA affect the performance and the genetic diversity of generalist neural game-playing agents. We focus specifically on deterministic and self-adaptive parameter control on the standard deviation of the Gaussian mutation operator.

2 BACKGROUND

The downsides of a static mutation step size is that 1) evolution may benefit from different mutation step sizes during different points in the evolutionary process [8], and 2) it ignores any information from the fitness landscape [2].

To alleviate the first problem, one could set the mutation step size based on a deterministic rule that states what the step size must be at each generation. Usually, a time-varying schedule is used [8], where the step size is altered by some rule after a set number of generations are passed.

In Eiben et al. [8], a simple update rule for deterministic parameter control is defined as

$$\sigma(t) = 1 - 0.9 \cdot \frac{t}{T}, \tag{1}$$

where t is the current generation number and T is the total number of generations to be performed in the entire evolutionary process. At the start of the evolutionary process, the step size is large, forcing the population to explore the environment more, while in later stages they exploit current maxima.

A weakness of static and deterministic mutation approaches is that mutation is applied to all individuals equally [12]. Some subsets of the population may benefit from a larger step size, while others should exploit the local environment. Such local information from the environment is ignored with static tuning or deterministic parameter control. In the self-adaptive approach, each chromosome x_j is extended with a personal mutation step size σ_j [6], which also evolves via means of its own crossover and mutation [12]. Consequently, the algorithm could still pursue exploration by increasing the diversity through altering the bad individuals, while maintaining the quality of the good individuals. A disadvantage of this strategy, however, is that parameter control is solely based on the problem knowledge as obtained by the evolutionary process [8]. According to [8], this disregards the human expertise, which could contain more global knowledge of the problem at hand and thus provide valuable insight on the proper parameter settings.

3 METHOD

We implement two variants of the genetic algorithm described in da Silva Miras de Araujo and de Franca [4]. This allows us to compare our results to the reported baseline.

The experiment is implemented using the Distributed Evolutionary Algorithm in Python [9], and the experiment is conducted using the EvoMan framework [5]. The EvoMan environment in which the agents play games is set to level 2, enemy mode to static, and rest of parameters are kept as default. The agent controller is an artificial neural network with a single hidden layer of 10 neurons. Both the hidden layer and the output layer use the sigmoid activation function [3]. The input to the network is normalised to the range $[0, 1]$. The sigmoid activation function ensures that the output of the network is also in the range $[0, 1]$, and the agents choose to take an action when the corresponding value is ≥ 0.5.

3.1 Algorithm Design

An individual is represented as a real-valued vector containing the weights and biases to the agent controller neural network. A population consists of 100 individuals, of which the values are uniformly initialised in the range $[-1, 1]$.

We evaluate each individual by having it play games against a training group of three enemies. We compute a game fitness g_i after a game against the i-th enemy in the training group,

$$g_i = 0.9 \cdot (100 - e_i) + 0.1 \cdot p_i - \log t_i, \tag{2}$$

where p_i and e_i are the energy level of player and enemy agent in one game, respectively, and t_i is the total game time steps taken against that enemy until the end of the fight.

The fitness value assigned to the individual is then computed as

$$f = \text{mean}(\{g_i\}_{i=1}^{3}) - \text{std\_dev}(\{g_i\}_{i=1}^{3}), \tag{3}$$

Figure 1: Mean population- and max-fitness at each generation for each algorithm variant with each training group.

where std_dev denotes the standard deviation. We subtract the standard deviation from the mean in order to mitigate the bias in the fitness value when there is an exceptional performance against one enemy. The fitness is to be maximised.

Parents are selected via tournament selection with tournament size of 2. Crossover is done via whole arithmetic recombination to produce a single offspring, with crossover probability $p_c = 1.0$,

$$offspring = \alpha \cdot parent_1 + (1 - \alpha) \cdot parent_2, \tag{4}$$

where α is sampled uniformly from the range $[0, 1]$. The total number of new offspring is the same as the population size. The offspring are then mutated with a Gaussian mutation operator, where each allele has the mutation probability $p_m = 0.2$. The mutation operator is initialised with mean $\mu = 0$ and standard deviation $\sigma = 1$ for all individuals, to give the same starting point.

The deterministic variant of the algorithm (DET) controls the σ of the mutation operator with respect to the number of generations during the evolution, as shown in Equation 1.

The self-adaptive variant (SELF) controls the σ_j for each individual x_j in two ways: σ_j values go through the same whole arithmetic operation as in Equation 4 and is mutated using the extended log normal rule [1],

$$\sigma_j' = \sigma_j \cdot \exp\left(\mathcal{N}(0, \tau') + \mathcal{N}_j(0, \tau)\right), \tag{5}$$

where $\tau' = c/\sqrt{2n}$ and $\tau = c/\sqrt{2\sqrt{n}}$, where n is the problem size, $c = 1$ [13], and the lower bound on σ_i is set to 0.005.

Survivor selection for the next generation is implemented via tournament selection with size 2. The algorithm evolves the population throughout 100 generations.

3.2 Experimental Setup

For each algorithm variant, we repeat the experiment 10 times. To measure the genetic diversity of the population, we measure the Manhattan distance [10] between the individual and the population mean in the vector space. Additionally, we record the mean fitness value of the population, referred to as the population-fitness, as well as the maximum fitness value, referred to as the max-fitness.

We select the best individuals from each experiment, referred to as the champions, according to their total individual gain [4]. A champion plays an additional five games against each of the eight EvoMan enemies, and the mean total gains of those games are recorded.

Table 1: Average player and enemy energy levels across five games with best champion against all enemies.

enemy	avg. player energy	avg. enemy energy
1	0.00	80.00
2	38.00	0.00
3	0.00	30.00
4	29.20	0.00
5	44.32	0.00
6	0.00	40.00
7	50.80	0.00
8	45.40	0.00

This entire process is repeated for each of the two training groups based on da Silva Miras de Araujo and de Franca [4]: Group 1 consisting of EvoMan enemies 1, 2, and 5, and Group 2 of enemies 4, 6, and 7.

4 RESULTS AND DISCUSSION

The population- and max- fitness of each algorithm variant for each training group are plotted in Figure 1. For both training groups, we observe DET outperforming SELF. We also note the population-fitness of SELF shows signs of convergence for Group 1 while DET continues to increase, to the point of overtaking the max-fitness of SELF in Group 2. We also observe in Figure 2 that champions evolved with DET show slightly better performance than SELF in terms of individual gains when fighting against all enemies, with one outlier from SELF with Group 1 showing the best result. However, performing the Welch's t-test between the two algorithms (Figure 2), we note that these differences are not statistically significant.

The majority of our champions show higher average individual gains compared to the GA10 agent with a static mutation operator in da Silva Miras de Araujo and de Franca [4]. The unpaired Welch's t-test shows that the outperformance of 32 of our champions against the baseline is statistically significant with significance level 0.05.

We notice an interesting pattern of enemies defeated by the champions. Champions evolved against Group 1 with either algorithm variant always beat enemies of Group 1, and at most one other enemy. Although they may show more consistent gains in the box plots, they are not well-rounded. Champions evolved against Group 2 show to be more robust defeating all enemies but 1 or

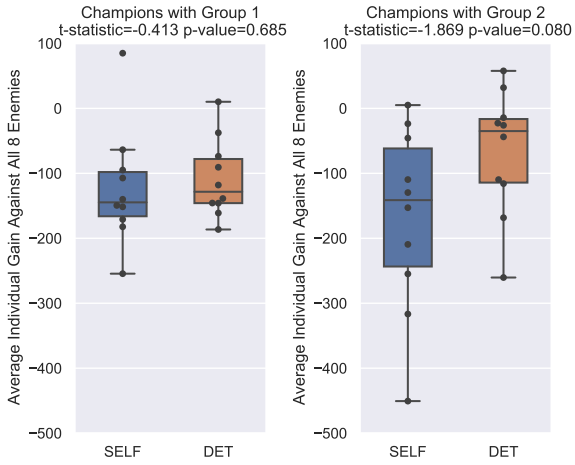

Figure 2: Boxplots of average individual gains of 10 champions per algorithm variant and training group. Each point represents the average individual gain across five games of a champion playing against all eight enemies. Welch's t-test yields t-statistic of -0.413 and -1.869, and p-values of 0.685 and 0.080 for Group 1 and Group 2, respectively.

3. We speculate that this may be due to Group 2 containing more challenging enemies [4], requiring agents with sophisticated strategies to be evolved. These results hint towards the importance of choosing the right training group. The best champion of all is one evolved against Group 2 with DET, being able to defeat five different enemies (Table 1).

Figure 4 shows the genetic diversity of the population for each algorithm variant and the population average of the standard deviation σ of the mutation operator. The genetic diversity of DET shows a decline, likely due to smaller perturbations during mutation as σ decreases over time, and the crossover operation taking the weighted averages, limiting the range of the allele values over time. This leads to individuals becoming more similar and centring around a local optimum, and as a result the population-fitness steadily increases. We hypothesise that more generations would show that the population-fitness would approximate the max-fitness.

The diversity of SELF on the other hand remains relatively stable. This is attributed to the σ values as shown in Figure 3, as they deviate only slightly from the initial values of 1, and thus add greater perturbations to the individuals. We expected a higher deviation from the initial values over time. Furthermore, we anticipated a decline, as it would be beneficial to begin fine-tuning the individuals in later generations. We posit that the value of the parameters being equally initialised may have had an unfavourable influence. Randomly initialising with varying values may benefit the evolutionary process more favourably. Therefore, we suggest further research into whether randomly initialising the σ values would improve the co-evolution. Additionally, along with the fact that the fitness values have not fully converged, experiments with more generations is warranted to observe how σ evolves over a longer period of time.

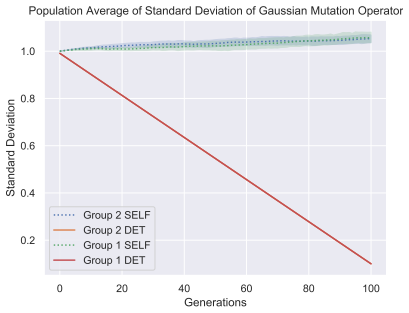

Figure 3: Population average of standard deviation for the Gaussian mutation operator at each generation.

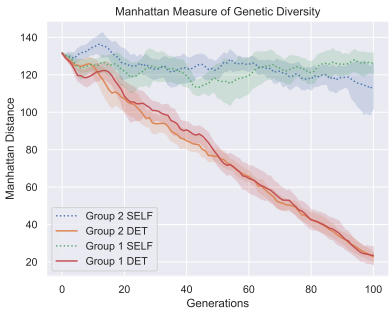

Figure 4: Genetic diversity of population at each generation measured as the Manhattan distance between an individual and the population mean for each algorithm variant and training group.

5 CONCLUSION

In this paper, we implemented two variants of an EA, DET and SELF, to examine how the different parameter control methods affect the performance and genetic diversity of generalist game-playing agents.

Our experiments show that agents evolved with DET outperform those evolved with SELF, in terms of both fitness and individual gains, at the point of termination. However, the difference in the total individual gains between the algorithm variants were not statistically significant. On the other hand, in comparison to the baseline [4], our experiments demonstrated that agents evolved with parameter control on the mutation operator outperform those evolved with static mutation operator, with statistical significance.

Moreover, our results showed a relationship between the change in standard deviation, genetic diversity, and performance. We observed that the population of SELF maintains more genetically diverse individuals than those of DET. However, failure to exploit the individuals by keeping the population diverse seem to have led to worse performing agents. We suggest further research to randomly initialise the parameter values for SELF to take better advantage of the evolutionary process. Additionally, we suggest to examine how the standard deviation of SELF changes with increased number of generations to transition from exploration to exploitation, and how that may affect the diversity and fitness over time.

REFERENCES

[1] H. Beyer and H. Schwefel. 2002. Evolution strategies - A comprehensive introduction. *Natural Computing* 1 (03 2002), 3–52. https://doi.org/10.1023/A:1015059928466

[2] Jorge Cervantes-Ojeda and Christopher Stephens. 2006. "Optimal" mutation rates for genetic search. https://doi.org/10.1145/1143997.1144201

[3] G. Cybenko. 1989. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)* 2, 4 (1 Dec. 1989), 303–314. https://doi.org/10.1007/BF02551274

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

[5] K. da Silva Miras de Araújo and F.O. de Franca. 2016. An electronic-game framework for evaluating coevolutionary algorithms. (2016). arXiv:cs.NE/1604.00644

[6] A.E. Eiben, Y. Karafotias, and E. Haasdijk. 2010. Self-Adaptive Mutation in Online, On-board Evolutionary Robotics. *Proceedings - 2010 4th IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshop, SASOW 2010* (10 2010), 147 – 152. https://doi.org/10.1109/SASOW.2010.31

[7] A.E. Eiben and J.E. Smith. 2015. *Introduction to Evolutionary Computing*. Springer. https://doi.org/10.1007/978-3-662-44874-8 Gebeurtenis: 2nd edition.

[8] A. E. Eiben, R. Hinterding, and Z. Michalewicz. 1999. Parameter Control in Evolutionary Algorithms. *Trans. Evol. Comp* 3, 2 (July 1999), 124–141. https://doi.org/10.1109/4235.771166

[9] F. Fortin, F. De Rainville, M. Gardner, M. Parizeau, and C. Gagné. 2012. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research* 13 (jul 2012), 2171–2175.

[10] P. Korosec. 2010. *New Achievements in Evolutionary Computation*. IntechOpen.

[11] S. M. Lucas and G. Kendall. 2006. Evolutionary computation and games. *IEEE Computational Intelligence Magazine* 1, 1 (2006), 10–18.

[12] S. Marsili-Libelli and P. Alba. 2000. Adaptive mutation in genetic algorithms. *Soft Computing* 4 (07 2000), 76–80. https://doi.org/10.1007/s005000000042

[13] S. Meyer-Nieberg and H. Beyer. 2007. *Self-Adaptation in Evolutionary Algorithms*. Vol. 54. 47–75. https://doi.org/10.1007/978-3-540-69432-8_3

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. (2013). arXiv:cs.LG/1312.5602

[15] D. G. Wilson, S. Cussat-Blanc, H. Luga, and J. F Miller. 2018. Evolving simple programs for playing Atari games. (2018). arXiv:cs.NE/1806.05695