Task 2: Generalist Agent

Evolutionary Computing - Group 84 - 18/10/2021

Jaimie Rutgers
Vrije Universiteit Amsterdam
Amsterdam, Netherlands
2598649

Martijn Wesselius
Vrije Universiteit Amsterdam
Amsterdam, Netherlands
2599666

Ekaterina Geraskina
Vrije Universiteit Amsterdam
Amsterdam, Netherlands
2636212

Dominic Istha
Vrije Universiteit Amsterdam
Amsterdam, Netherlands
2600140

1 INTRODUCTION

Environments with large action and state spaces and unmodelled constraints offer challenging optimization problems [18]. Examples of such optimization problems in complex environments can be modeled by video games. Video games, therefore, offer a popular platform to evaluate the performance of algorithms on complex optimization problems [3, 10, 14, 18]. A promising class of algorithms shown to be able to perform well on a variety of video games are Evolutionary Algorithms [2, 11, 16, 17, 20]. Evolutionary algorithms are able to evolve autonomous agents, that can deal with the complex environments of video games, without the interference of human design [18].

The goal of this research is to examine the difference between an Evolutionary Algorithm that optimizes a specific set of parameters and an Evolutionary Algorithm that optimizes the whole distribution of parameters. Their performance is evaluated on the EvoMan framework, proposed by de Araújo and de França [3], which offers a complex video game environment for testing Evolutionary Algorithms. As cited in Paolo Pagliuca [13], the work of Lehman [7] and Lehner [8], suggests that optimization over a distribution of parameters results in solutions that are more robust to parameter variation. This is associated with more robust solutions in varying environmental conditions. Therefore, it is hypothesized that Evolutionary Algorithms optimizing the distribution of parameters may offer superior performance in complex environments like video games.

To test this hypothesis, we will implement the NeuroEvolution of Augmenting Topologies (NEAT) method, proposed by Stanley and Miikkulainen [19], to optimize a specific set of parameters. This method simultaneously optimizes the weights and topology of an Artificial Neural Network, but does not optimize the parameters over a distribution. NEAT is chosen due to its superior performance in our previous work [15], as well as the work of de Araujo and de Franca [4]. In comparison, we examine an Evolutionary Algorithm that does optimize the entire distribution of the parameters; the Exponential Natural Evolution Strategy (xNES), proposed by Glasmachers et al. [5]. xNES was chosen, because Pagliuca and Nolfi [12] demonstrated its robustness compared to other Evolutionary Algorithms on changing environments on three different tasks.

We begin in Section 2 by describing the xNES and NEAT methods and their underlying principles. Next, we will present the experimental setup and motivate the parameter choices. In Section 3, we will analyse and compare the results of xNES and NEAT. In Section 4, we will discuss the experiment and results and present our concluding remarks. Section 5 briefly states the contributions of the authors to this report.

2 METHODS

2.1 Exponential Natural Evolution Strategies (xNES)

xNES [5] is an Evolution Strategy that uses natural gradients with an exponential parametrization. It is an extension to Exact Natural Evolution Strategies (eNES) [22], which is again an extension to Natural Evolution Strategies (NES) [21]. In this section we will explain the idea of NES and how xNES works. For the various strategies introduced in eNES to overcome the difficulties in NES, we refer to Yi et al. [22].

Natural Evolution Strategies. Natural Evolution Strategies are well-grounded, evolution-strategy inspired black-box optimization algorithms, which instead of maintaining a population of search points, iteratively update parameters of a search distribution [21]. In its standard form, NES uses Gaussian distributions with a fully adaptive covariance matrix. In NES, we collect the mean $\mu \in \mathbb{R}^{d\times d}$ and $A \in \mathbb{R}^{d\times d}$, the square root of the covariance matrix (such that $AA^T = C$), where d is the number of weights and C the covariance matrix. $x = \mu + \sigma B \cdot z$ then transforms a standard normal vector $z \sim \mathcal{N}(0, I)$ into the sample $x \sim \mathcal{N}(\mu, C)$ with $B = \frac{A}{\sigma}$ and $\sigma = \sqrt[d]{|det(A)|}$. Evaluation is done by shaping the fitness values into utilities and sorting these values. Using the density function of $\mathcal{N}(\mu, C)$ a Monte-Carlo estimate of the expected utility gradient can be obtained, see Glasmachers et al. [5]. Updates of parameters are then based on the *natural gradient* [1] instead of the stochastic gradient of the expected fitness, by using natural gradient ascent. In each generation, the whole population is replaced and no selection mechanism is used. By using utility values, the state of the algorithm is encoded in the search distribution and not in the population.

Exponential parametrization and natural coordinates. xNES extends eNES by using an exponential parametrization. This is done to ensure that the updated covariance matrix still holds the property that $AA^T = C$. An alternative would be to update a factor A of C, but this could eventually result in undesired oscillations [5].

Updates. After computing fitness values, converting them to utility values and ranking these values, the parameters of the distribution are updated. In short, the gradients with respect to μ, the step size σ and the transformation matrix B are computed as G_δ, G_σ and G_B, respectively. These gradients are then used, together with the corresponding learning rates $\eta_\mu, \eta_\sigma, \eta_B$ to update the distribution parameters μ, σ and B. For sake of brevity, all update formulas have been ommitted from this report, but can be found in Glasmachers et al. [5]. After the update, the xNES algorithm starts over by sampling from the Gaussian distributed and transforming this sample with its updated distribution parameters.

2.2 NeuroEvolution of Augementing Topologies (NEAT)

NEAT is an evolutionary algorithm that simultaneously optimizes the weights and topology of a Neural Network. Unlike xNES, NEAT can evolve the number of hidden nodes in a neural network, which allows NEAT to optimize and complexify solutions. NEAT combines four key features to efficiently evolve increasingly complex solutions over time: mutation, crossover, speciation and incrementally growing solutions from a minimal structure. First, through genetic mutation, solutions with both varying weights and different topologies are explored. Second, NEAT introduces a simple solution for performing crossover between topologically different solutions, by keeping track of the chronology of every gene in the system. Next, NEAT speciates the solutions, to effectively protect innovation and explore the search space in multiple directions. Finally, NEAT is initialized with a uniform population and only introduces structural

changes when necessary, which allows NEAT to limit the number of searches through weight dimensions, significantly increasing the efficiency. For a more detailed description of NEAT, we refer to our previous work [15] or the original paper introducing NEAT [19].

2.3 Experimental setup

The algorithms are executed for two training sets of enemies, which will be referred to as an experiment. Following de Araujo and de Franca [4], we use one set of two enemies, consisting of enemies 2 and 4, given the combination of their high fitness and gain. The second set will be a set of three enemies, consisting of enemies 1, 2 and 5, because enemy 2 was a recurrent good enemy from their considered pairs and (1, 5) was the pair with the highest fitness. Both algorithms are executed for 100 generations, such that we can make fair comparisons to the work of de Araujo and de Franca [4]. This process is repeated 10 times for both xNES and NEAT, resulting in 10 runs. For each generation and each of the 10 runs, we store the mean and maximum fitness of the population to evaluate the algorithms. The fitness per iteration is expressed by the following equation:

$$\text{fitness} = \sum_i e_{p,i} - e_{e,i} - \log t_i \tag{1}$$

where e_e, i and e_p, i are the energy measures of the enemy and player for the game against enemy i respectively, with $e_e, e_p \in [0, 100]$. t_i is the number of time steps the game took against enemy i. A final fitness value is given by summing over the results per enemy. This fitness function combines the total gain $(e_{p,i} - e_{e,i})$ with $log(t)_i$ to penalize long lasting solutions, inspired by de Araujo and de Franca [4]. Since this research also has a competition associated with it, which will be evaluated on the gain, we aimed for a fitness function that is primarily based on the gain. Next, we run the best solution per run 5 times on the validation set consisting of all enemies to account for stochasticity. These results are evaluated using the individual gain, which is expressed by $g_i = e_p - e_e$. A t-test is then applied to test for a significant difference in mean individual gains between the two algorithms. Finally, for our overall best solution, we will present the player and enemy energies per enemy level in a table.

The xNES algorithm was implemented in Python. The algorithm aimed to optimize the weights for the required 10 hidden node Neural Network of the Evoman framework. For the algorithm, μ was initialized using a random uniform [-1,1] initialization, following Pagliuca and Nolfi [12]. A was created as an identity matrix with the dimension equal to the length and which gives the input to compute B and σ, see Section 2.1. The learning rates $\eta_\mu, \eta_\sigma, \eta_B$ were all set to the default values as presented in Table 1 in Glasmachers et al. [5]. The same holds for the population size, which gives a population size of 20. For the utility function we also use the default value in [4], which is in fact the weighting scheme of the CMA-ES algorithm [6]. To induce exploration of the algorithm, all parameters were reset to their default values in case the maximum fitness did not improve for 15 generations.

The NEAT algorithm for the experiments conducted, is implemented using the NEAT-Python library [9]. The individuals in the population are all feed-forward neural networks in our experiments. The configurations of NEAT are the same for the two experiments against the different sets of enemies. The parameter choices for NEAT are primarily motivated by our previous work and findings [15], except for a minor adjustment to the elitism parameters and minimum specie size. The elitism of different species is increased to 3, while the elitism within a specie and minimum specie size are decreased to 1 and 3, respectively. The elitism of different species is increased, to ensure higher variation in the population for this more complex task. The other parameters are decreased to improve efficiency.

3 RESULTS

Figures 1 and 2 show the average and standard deviation over the 10 runs of the mean and maximum fitness, per generation. For both sets of enemies we observe that mean fitness of xNES is considerably higher than that of NEAT. Moreover, Figure 2 shows that the standard deviation range of the mean fitness for xNES actually overlaps with the range of the standard deviation of the maximum fitness for both xNES and NEAT. It shows that the gap between the maximum and mean fitness for xNES is much smaller, than for NEAT. NEAT naturally contains a higher degree of variation in the solutions, since it explores a wide range of topologies simultaneously, some of which are more promising than others.

Additionally, both graphs show that during the first 25 generations the maximum fitness for xNES increase quickly, whilst the NEAT maximum fitness shows a more subtle improvement. Around the 25th generation, the curves cross for the first time. Then, in Figure 1 the curves stay close to each other for the rest of the 100 generations. Both curves eventually end up at a fitness score of around 30. In Figure 2 we observe the same behaviour until around generation 60, after which the maximum fitness of NEAT starts rising again. Maximum fitness scores in Figure 2 for both algorithms are significantly higher than in Figure 1, ending up at approximately 100. Comparisons to the results of de Araujo and de Franca [4] are hard to make, as they present no results on gain for the training group specifically, but either a total gain on *all* enemies or a *fitness* on the training group. The latter is hard to compare since we used a different fitness function

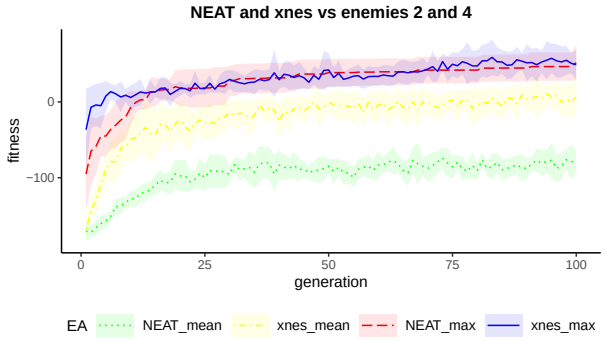

Figure 1: Performance of NEAT and xNES vs. enemies 2 and 4

The two EAs are also tested on all eight enemies of the EvoMan framework [3]. Figure 3 shows the distribution of individual gain

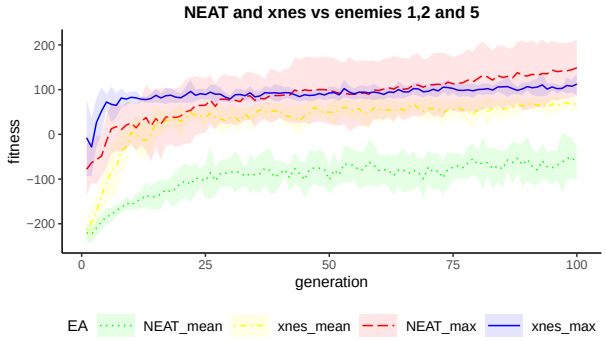

Figure 2: Performance of NEAT and xNES vs. enemies 1,2 and 5

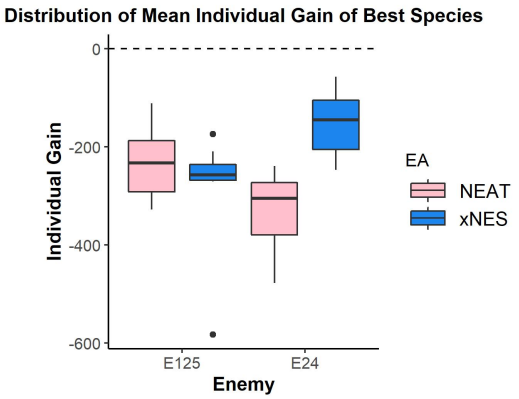

Figure 3: Individual gains for best runs

Table 1: Results of best solution against all enemies

Enemy	1	2	3	4	5	6	7	8
Player Energy	0	74	0	0	57	0	0	0
Enemy Energy	70	0	60	70	0	80	30	90

for the best runs. Both algorithms did not succeed in defeating all enemies for either of the two training sets, as is visible by the location of the boxplots under the threshold of 0. Additionally, one can notice that the standard deviation bars are considerably smaller for xNES than that of NEAT.

For enemies 2 and 4, xNES(μ = -153.64, σ = 64.36) outperforms NEAT (μ = -335.01, σ = 84.10) with a significant gap in performance (t = -5.42, p-value < 0.001). For enemies 1, 2 and 5 both Algorithms perform equally poor, with one solution of xNES almost reaching -600. Visually NEAT (μ = -230.18, σ = 73.26) seems to perform slightly better than xNES (μ = -276.63, σ = 111.64). Significance in the performance difference was carried out with the Wilcoxon test, instead of the t-test, because the distribution of the xNES results did not pass the Shapiro-Wilk test for normality. (W =0.63, p-value < 0.001). The Wilcoxon test shows that this difference is not significant (W = 56, p-value = 0.68).

Table 1 shows the performance of our best solution against all enemies in terms of remaining player and enemy energy. The best weights were found in run 5 of the xNES method trained on enemies 1, 2, 5, where it showed a fitness of 178. The performance is rather bad as the best solution only wins from enemies 2 and 5 on which it was trained. It loses from all other enemies, which leads us to believe that the xNES algorithm is overfitting on enemies 2 and 5. However, the fitness seems to be still increasing in Figure 2, which gives reason to believe the optimum has not yet been reached.

Lastly, let us compare our results with the general strategy results of de Araujo and de Franca [4]. Their NEAT algorithm scored a gain of -73 and -143, when trained on enemies (2,4) and (1,2,5), respectively. This is significantly higher than our gains obtained, training on the same sets of enemies, see Figure 3. When looking at the number of enemies defeated, our best solution, xNES trained on enemies (1,2,5), was only able to beat enemies 2 and 5, as shown in Table 1. de Araujo and de Franca [4] were able to evolve NEAT to defeat enemies 1, 2 and 5, outperforming our xNES solution. Their best solution trained on enemies (1,2,5), a LinkedOpt algorithm with 10 hidden nodes, was even able to defeat 5 out of the 8 enemies.

4 DISCUSSION

The goal of this research was to examine the difference between an Evolutionary Algorithm that optimizes a specific set of parameters (NEAT), and an Evolutionary Algorithm that optimizes the whole distribution of parameters (xNES). To conclude, xNES only outperformed NEAT in terms of acquired gain on the set of enemies 2 and 4. Gain on enemies 1, 2 and 5 was roughly equivalent, as were the mean fitness values after 100 iterations on both sets of enemies. The relatively high mean fitness values, low standard deviations and the ease with which xNES finds good fitness values, leads to believe that xNES is a slightly better performing Evolutionary Algorithm than NEAT when it comes to designing an ANN for a generalist agent. However, the performance of xNES itself based on gain is not optimal.

It must be mentioned unfortunately, that at a very late stage of this research, it became clear that the results of xNES are not 100% correct. Recall that the weights are constructed according to the following transformation $x = \mu + \sigma B \cdot z$. Our implementation did not store the weights of the best solution, nor all parameters μ, σ and B (such that we could sample weights), but only μ. This means that Figure 3 and Table 1 are based on the vector μ and not actual weights. Nevertheless, the weights are heavily correlated with μ and $\mathbb{E}[x] = \mu$. Furthermore, we expect σ and B to become very small in late generations, as the algorithm becomes more certain about its sample transformation. Eventually, this implies that μ is a good representation of x and that the results could only improve when incorporating σ and B as well.

5 CONTRIBUTIONS

Jaimie Rutgers and Martijn Wesselius implemented the xNES method, Dominic Istha implemented the NEAT method. Ekaterina visualized and tested the results. The report itself is the result of the combined efforts of all authors.

REFERENCES

[1] Shun-Ichi Amari and Scott C Douglas. 1998. Why natural gradient?. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, Vol. 2. IEEE, 1213–1216.

[2] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. 2019. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680* (2019).

[3] Karine da Silva Miras de Araújo and Fabrício Olivetti de França. 2016. An electronic-game framework for evaluating coevolutionary algorithms. *arXiv preprint arXiv:1604.00644* (2016).

[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] Tobias Glasmachers, Tom Schaul, Sun Yi, Daan Wierstra, and Jürgen Schmidhuber. 2010. Exponential natural evolution strategies. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*. 393–400.

[6] Nikolaus Hansen, Sibylle D Müller, and Petros Koumoutsakos. 2003. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES). *Evolutionary computation* 11, 1 (2003), 1–18.

[7] Chen J. Clune J. Stanley K. O. Lehman, J. 2018. ES is more than just a traditional finite-difference approximator. In *In Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO, 450–457.

[8] Ben Lehner. 2010. Genes confer similar robustness to environmental, stochastic, and genetic perturbations in yeast. In *PloS one*, Vol. 5(2). e9035.

[9] Alan McIntyre, Matt Kallada, Cesar G. Miguel, and Carolina Feher da Silva. [n. d.]. neat-python. https://github.com/CodeReclaimers/neat-python. ([n. d.]).

[10] Risto Miikkulainen, Bobby D Bryant, Ryan Cornelius, Igor V Karpov, Kenneth O Stanley, and Chern Han Yong. 2006. Computational intelligence in games. *Computational Intelligence: Principles and Practice* (2006), 155–191.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[12] Paolo Pagliuca and Stefano Nolfi. 2019. Robust optimization through neuroevolution. *PloS one* 14, 3 (2019), e0213193.

[13] Stefano Nolfi Paolo Pagliuca. [n. d.]. Robust optimization through neuroevolution. *PLoS ONE* 14, 3 ([n. d.]).

[14] Sebastian Risi and Julian Togelius. 2015. Neuroevolution in games: State of the art and open challenges. *IEEE Transactions on Computational Intelligence and AI in Games* 9, 1 (2015), 25–41.

[15] Jaimie Rutgers, Martijn Wesselius, Ekaterina Geraskina, and Dominic Istha. 2021. *Task 1: Specialist Agent.* Technical Report. Vrije Universiteit Amsterdam.

[16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484–489.

[17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362, 6419 (2018), 1140–1144.

[18] Kenneth O Stanley, Bobby D Bryant, and Risto Miikkulainen. 2005. Real-time neuroevolution in the NERO video game. *IEEE transactions on evolutionary computation* 9, 6 (2005), 653–668.

[19] Kenneth O Stanley and Risto Miikkulainen. 2002. Efficient evolution of neural network topologies. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*, Vol. 2. IEEE, 1757–1762.

[20] Gerald Tesauro. 1995. Temporal Difference Learning and TD-Gammon. *Commun. ACM* 38, 3 (March 1995), 58–68. https://doi.org/10.1145/203330.203343

[21] Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. 2014. Natural evolution strategies. *The Journal of Machine Learning Research* 15, 1 (2014), 949–980.

[22] Sun Yi, Daan Wierstra, Tom Schaul, and Jürgen Schmidhuber. 2009. Stochastic search using the natural gradient. In *Proceedings of the 26th Annual International Conference on Machine Learning*. 1161–1168.