Evolutionary Computing

Task I: Specialist Agent

October 4, 2019

Team 38
Mathijs Maijer - mmr479
Esra Solak - esk410
Kasper Nicholas - kns280

1 INTRODUCTION

Machine learning techniques are often applied to create autonomous agents capable of learning to play games on par with, or even better than, humans [1, 2]. The agent learns through the measurement of results after several events occur and changes its behaviour accordingly. An issue that often arises in machine learning, and even more so in its application in video games, is the over-fitting or over-specialisation of agents. This means that the agent tends to work very well in a few, specific scenario's in the game, rather than being good at the game in general. Obtaining a generalised game playing agent thus becomes a challenge and requires adequate training data depending on the task at hand.

An important adaptation of machine learning in games is the technique called neuroevolution (NE). Neuroevolution uses evolutionary algorithms (EA) to train artificial neural networks (ANN). Over the last two decades the application of NE in games was researched extensively[3, 5, 6]. Using video games as a test bed proved to be an efficient method to research NE and other AI methods compared to existing test beds such as mobile robotics [5]. A distinction can be made between EAs that evolve the weights of the ANN, the topology of the ANN or a combination of the two. The focus of this paper is on the evolution of the neural weights only, and is done through EvoMan, a NE framework with a video game as a test bed[2].

The Evoman framework, based on the 1987 Mega Man game, was created and extensively researched by Da Silva and Olivetti in 2016 [1]. The standard choice for obtaining optimal neural weights is gradient descent, however, rather than a sequence of pairs of inputs and desired outputs (supervised learning), the nature of the fitness function is indicative [1]. An evolutionary algorithm is used instead to evolve the neural network weights towards the optimal solution(s). The multimodal search space of the neural weights further indicates that different solutions can lead to equally good playing strategies[5]. The implementation of an EA that deals with local optima and overfitting of the ANN is thus of importance.

This paper aims to research the difference between the $\mu + \lambda$- and the μ, λ evolutionary strategies (ES). This is done by means of the EvoMan framework and two EAs to solve the provided tasks; in this case defeating three different enemies provided by the EvoMan framework. The evolutionary strategies will be applied to specialist agents that are optimised for solving one particular task, in this case one enemy. The ES are implemented by means of the DEAP package in Python[4]. Subsequent results are then compared to each other and to literature in order to assess the quality of the implemented algorithms for optimising player strategies in video games.

2 METHODS

In order to understand the complete methodology that was applied in our research, a brief overview of the EvoMan framework as in [1] is given. Thereafter, we briefly elaborate upon NE and our implementation of two EAs and their respective properties.

2.1 The EvoMan Framework

The EvoMan framework consists of the player agent and several different enemy agents. The main character's goal is to defeat each of these enemies by firing projectiles at them. Each enemy fires back and moves according to a standard set of rules. Both the player and enemy start off with 100 life points that decrease whenever they get hit with the opponent's projectiles. The player also gets hurt when it collides with an enemy. The game ends when the energy goes to zero for one of the involved agents.

In order to assess the state of the game we record data from twenty sensors after discrete time steps. There is a player controller present, namely an ANN, that maps the values of these sensors to actions the player can use. The sensors are assigned weights in the ANN and are considered the genotype of our EA. Sensors record the vertical and horizontal distances for both agents, the direction for both agents, and x and y coordinates of a maximum of eight projectiles at a time. The values from every sensor are then mapped to one or more out of four actions, hence, the finished network can be considered as the phenotype in our EA. The player agent can move, shoot, jump and interrupt its jump by falling to the ground.

Three enemies were employed in our experiments:

Flashman (enemy 1) follows a player in the game, can freeze the player's movement and shoot eight projectiles at a time.

AirMan (enemy 2) stands still on either side of the game and shoots six projectiles spread across the screen. After several attacks it moves to the opposite side and attacks the player again.

WoodMan (enemy 3) slowly moves towards the player whilst shooting four vertical projectiles from above. The EvoMan framework, including the enemies, is programmed through the Python packages Pygame and Numpy.

2.2 Neuroevolution

As was stated in the introduction the use of gradient descent for obtaining optimal weights in our neural network is unfeasible due to the lack of correct input-out tuples that are required for supervised learning. Instead two evolutionary strategies, $\mu + \lambda$ and μ, λ, are applied by measuring the quality of the outcome after a sequence of inputs and outputs [5]. The network weights are evolved according to three different operations:

(1) Offspring of a given population of solutions is done by means of a **crossover** operator on pairs.
(2) Random samples α from a uniform distribution are used to **mutate** solutions. The offspring x' then becomes:

$$x' = \alpha x_1 + (1 - \alpha) x_2, \tag{1}$$

where x_1, x_2 are chosen solutions.
(3) In the $\mu + \lambda$ case the next generation's parents are selected from both the parents and offspring. Conversely, μ, λ only considers the offspring for the next generation's parents. The parameter μ dictates the number of parents selected for the next generation, whereas λ dictates the number of offspring that is created.

Parameter	Value
Population size	100
Crossover Prob.	0.5
Mutation Prob.	0.2
μ	100
λ	200
Num. Generations	30

Table 1: Parameter values for the $\mu + \lambda$ and μ, λ algorithms.

It should be noted that evolutionary strategies are less prone to get stuck in local optima due to the stochastic elements that are included. The fitness function we optimise is derived from p and e, the current energies levels of the player and enemy respectively. The fitness function is now defined as

$$f = 0.9 \cdot (100 - e) + 0.1 \cdot p - \log t, \tag{2}$$

with t the total number of time steps required to end the fight.

Finally Both $\mu + \lambda$ and μ, λ are implemented through the DEAP package in python. Subsequent parameter values for both algorithms are initialised according to table 1, and are loosely based on the parameter values used in [2]. A limit of 30 generations was due to computational limitations and the lack of significant improvement in results for more generations. Every strategy was trained on each enemy independently, training specialist agents that are optimised to provide only a solution to one specific enemy. Next to that, every training, or experiment, was repeated ten times.

3 RESULTS

This section shows and analyses the results acquired from the methods explained in the previous section.

Figure 1 shows the average fitness per generation averaged over 10 experiments for both strategies. The standard deviation of the average fitness over the 10 experiments is shown as error bars for each generation. It can be seen that after thirty generations, the plot seems to converge. The figures for enemy 1 and 3 show a little deviation around the average value, but the results for enemy 2 show that the average fitness converges after around 5 generations, with very small deviation.

The average life of the population averaged over 10 experiments is shown for each generation in figure 2. It can be seen that the standard deviation is quite big. For enemy 1 more so for the $\mu + \lambda$ strategy. For the second and third enemy, the standard deviation is more or less equal between strategies, while it seems the $\mu + \lambda$ strategy performs slightly better, but also has a bigger standard deviation.

Figure 3 shows the best solutions found for each strategy for all 10 experiments among the whole population during training. The fitness acquired for these ten best solutions are shown in box-plots for all three enemies. For the first enemy, both strategies perform quite similar, except that the μ, λ strategy has two outliers. For the second enemy, the $\mu + \lambda$ strategy spans a much larger range of fitness values found compared to the μ, λ strategy, but the μ, λ strategy seems to perform better consistently. For the third enemy it seems like the $\mu + \lambda$ outperforms the μ, λ strategy more consistently.

Looking at figure 1, it can be seen that the average fitness is generally higher for the $\mu + \lambda$ strategy, except for enemy two, where both strategies seem to perform almost equally. When looking at figure 2, this thought would be reinforced, as both the average life and standard deviation for both strategies seem to remain quite equal for enemy two. But actually, if figure 3 is considered, it can be seen that the best solutions gathered over the ten experiments differ for both strategies. Table 2 shows the results for a statistical T-Test performed on the data used to create figure 3. These results show that the difference between best found solutions is actually significant for enemy two if $\alpha = 0.05$ is used. The difference in performance between the strategies is not significant for enemy one or enemy three.

Enemy 1	Enemy 2	Enemy 3
T: 0.822	T: -2.420	T: 1.565
p: 0.422	p: 0.026	p: 0.134

Table 2: Statistical T-test outcome per enemy for the lists of best solutions regarding fitness found over 10 independent experiments for each strategy

4 DISCUSSION

Considering the results, at first glance it seems that the $\mu + \lambda$ strategy outperforms the μ, λ strategy. The average values are generally higher, but due to the high variance, no statistical significant differences were found to support this claim. In contrast, for enemy 2, the μ, λ strategy has performed significantly better in providing the best solution regarding fitness. The sole difference in these two strategies, is the way the offspring are selected. These results could be explained with the idea that the $\mu + \lambda$ strategy has a higher pool of individuals to choose from to use as new parents, which produces a higher variance, while the μ, λ strategy is selectively better in easier environments where it can more consistently produce better solutions.

Furthermore, a basic GA was trained with the same neural network topology, but with different parameters for the genetic algorithm[2]. Compared to both GAs trained in this paper, it performs considerably worse in general. Only for the first enemy are the averages of the best scores almost equal to the compared GA, but both strategies combined with the parameters used in this paper outperform the GA in the mentioned paper by solely looking at the highest fitness obtained for all three enemies. Next to this, even though run times are not mentioned, the solutions provided in this paper are found during 30 generations, in contrast to the 100 generations of the compared GA.

This paper only considers specialist agents, but the real challenge lies in training generalist agents, where one model can be trained on multiple enemies, providing one solution that can solve multiple problems. It might be interesting to see if the $\mu + \lambda$ and μ, λ strategies show different results when being used to train a generalist agent.

Furthermore, from the results of this paper, it can be seen that evolutionary algorithms can provide very strong solutions and can be used to 'train' neural networks without the use of back-propagation or any other gradient descent methods. This is especially helpful in situations where gradient descent can not be

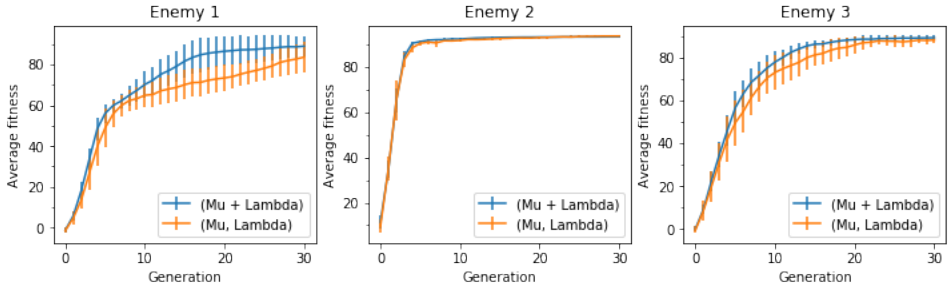

Figure 1: Average fitness of the population per generation averaged over 10 experiments for each enemy and evolutionary strategy.

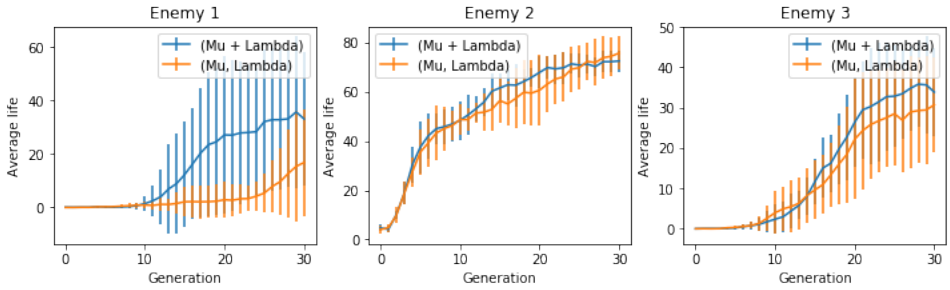

Figure 2: Average life of the population per generation averaged over 10 experiments for each enemy and evolutionary strategy.

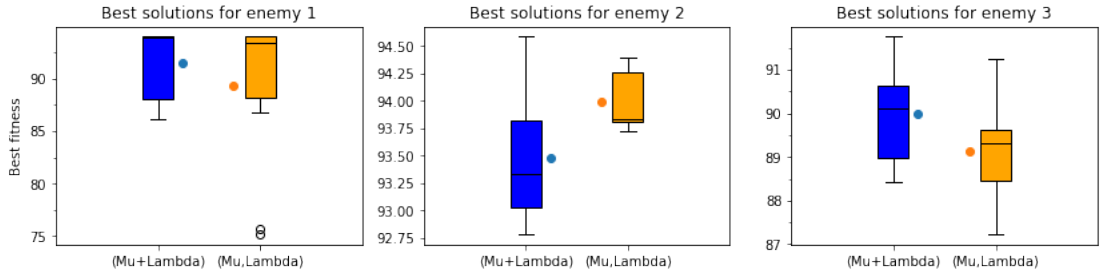

Figure 3: Box-plots for highest fitness for each of the 10 independent runs per strategy. The arithmetic mean of the highest fitness over the 10 runs is displayed as a circle next to the corresponding strategy.

applied. Future research could attempt to combine both the traditional training of neural networks and evolutionary algorithms, in environments where gradient descent can also be used, by either using back propagation and using an evolutionary algorithm to find new optima after the training has already converged, or to 'pre-train' a neural network and then using back-propagation to find optimal solutions that can not be achieved by relying on a single technique.

REFERENCES

[1] 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).

[2] 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.

[3] Daniel Jallov, Sebastian Risi, and Julian Togelius. 2016. EvoCommander: A novel game based on evolving and switching between artificial brains. *IEEE Transactions on Computational Intelligence and AI in Games* 9, 2 (2016), 181–191.

[4] De Rainville, Félix-Antoine Fortin, Marc-André Gardner, Marc Parizeau, Christian Gagné, et al. 2014. DEAP: enabling nimbler evolutions. *ACM SIGEVOlution* 6, 2 (2014), 17–26.

[5] 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.

[6] Julian Togelius. 2014. How to run a successful game-based AI competition. *IEEE Transactions on Computational Intelligence and AI in Games* 8, 1 (2014), 95–100.