A Comparison of Algorithms Playing EvoMan

1st Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

Eugen **Croitoru**
Dept. of Computer Science
"Al. I. Cuza" University
Iasi, Romania
eugennc@uaic.ro

***Abstract*—This paper describes a comparison between algorithms for evolving agents for the game Evoman. We have tried a cascade ensemble method, starting with a fast algorithm, and exploiting the found solutions with a slower algorithm. The algorithms used for the first stage are Q-learning, Genetic Algorithms and Particle Swarm Optimization. All are searching for the optimal weights of a fixed-structure neural network. Both the GA and the PSO algorithms were tested with sparse and iterative evaluations. The best explorative algorithm was the sparse PSO. The exploitative algorithm we have used is the Proximal Policy Optimization algorithm. Using PPO with random initialization led to better results than any ensemble method, and often, better results than the upper bound provided in the problem statement. However, we also show significant overfitting in our approach.**

***Index Terms*—game-playing agent, Artificial Intelligence, EvoMan, Genetic Algorithm, Reinforcement Learning, Q-learning, Neuroevolution, Particle Swarm Optimization, Proximal Policy Optimization**

I. Introduction

This paper presents our solution for the "Evoman: Game-playing Competition for WCCI 2020" [1]. We train an agent playing the 2D shooting game Evoman [2]. We have tried an ensemble cascade method with two stages: For the first stage we have tested algorithms that either found acceptable solutions quickly either had a high explorative bias. The algorithms we considered for this stage are Q-Learning [6], Genetic Algorithms [7] and Particle Swarm Optimization [8]. The structure of the ANN was fixed, and the optimal weights were searched. Both the GA and the PSO were tested with a sparse and an iterative evaluation approach.

From these quickly-found starting points, we sought to refine the results using a slower Proximal Policy Optimization [10] algorithm. While PSO was the best first-stage algorithm, we found that a random weight initialization worked best with PPO. This fixed-topology, generic PPO is sometimes (4 ouf of 8 opponents) able to surpass the specialized NEAT upper bound provided in the problem description.

We also find significant overfit. The problem requires selecting 4 opponents for training, leaving the other 4 for testing. Our method performs very well on the first 4, and worse on the rest. Additionally, the best agent at the end of training is surpassed, in testing, by an earlier-trained solution.

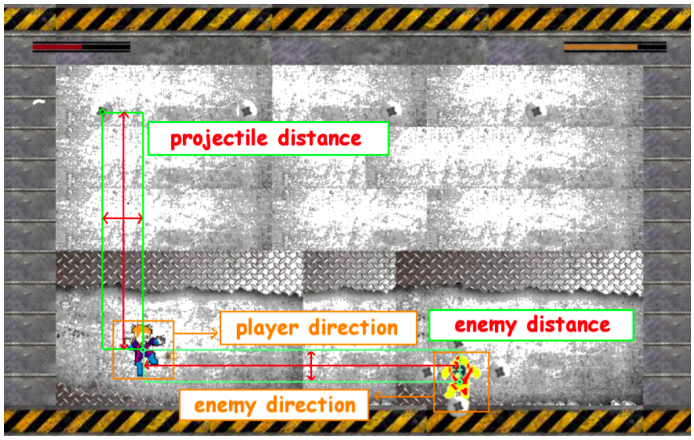

Fig. 1. Sensors available for the player [2].

II. Problem Description

A. Environment

EvoMan [2], [3] is a framework for testing competitive game-playing agents. This framework is inspired by Mega Man II [5], the game created by Capcom. EvoMan is a 2D shooting game where the player controls an agent playing against an opponent. The agent will collect information about the environment through 20 sensors (Fig 1):

- 16 correspond to horizontal and vertical distances to a maximum of 8 different opponent projectiles.
- 2 correspond to the horizontal and vertical distance to the enemy.
- 2 describe the directions the player and the enemy is facing

The 5 actions which the agent may take are: walk left, walk right, shoot, press jump, release jump.

The player and opponent start with 100 life points. On each hit receives, their life points decrease. The player wins by reducing the opponent's life points to 0, and lose if their life points are reduced to 0 first.

In the original Capcom game the player would have to beat 8 opponents and acquire their weapons as they are defeated. The additional difficulty of EvoMan comes from the fact that the player has to defeat all the opponents using only the starting weapon. Each opponent can be fought on a specified difficulty level. The difficulty level is an integer greater or equal than

1 which is translated into a factor for the damage taken and damage given by the player, the higher the difficulty level the lower the damage given and the higher the damage taken. The framework is freely available[1]. There is also an extensive documentation available[2].

B. Problem

Defeating an opponent is relatively easy when using specialized models against a specific enemy. The authors of the EvoMan framework trained multiple specialized agents against each opponent. After playing a game, the formula for the gain by which an agent is evaluated is:

$$gain = 100.01 + player\_life - enemy\_life \qquad (1)$$

The best possible value for gain is 100.01, for a player which defeats the opponent without getting hit. The highest final gain is 185.67(as a harmonic average of 8 values) [2] and was obtained with the NEAT [9] algorithm.

The problem we are trying to solve is a generalization of the one above. We are to train an agent against 4 enemies and then test its performance against all 8. The final score metric of an agent is the harmonic mean of the gains against all enemies. The combination of four enemies we are to train the agent is not fixed, but left as a free choice. The difficulty level for solving this problem is 2 for all enemies in training and testing, but we did some exploratory work with greater levels of difficulty.

The specifications above, like the number of enemies chosen for training or the difficulty level, come directly from a competition organized by the creators of the framework [1].

III. Approach

Our intention was to develop a cascade ensemble method where a fast method could provide a better starting point to a slower algorithm. The algorithm we expected to obtain a decent gain fast is a classic Q-learning [6] algorithm. Two more exploratory algorithms we have considered are Genetic Algorithms [7] and Particle Swarm Optimization [8]. The slower algorithm, intended to exploit the starting points provided by the first algorithm in the cascade, is a Proximal Policy Optimization [10] algorithm.

To find the best first-stage algorithm we have made the following changes (which had no impact in the training and testing for the final solution to the problem, other than guiding the choice of algorithm):

- we have increased the difficulty from the default 2 up to 5.
- the evaluation is made only on the second opponent due to the varied environment.
- the gain function was modified as in Karine Miras' analysis [11] to:

$$fitness = 0.9 \cdot (100 - enemy\_life) + 0.1 \cdot player\_life - ln(nr\_of\_game\_timesteps) \qquad (2)$$

The best possible value for the fitness is 100.

The best exploratory algorithm would be trained on four enemies and used in cascade with PPO without the changes mentioned in this section.

A. Q-Learning

The algorithm we expected to obtain a decent gain fast is a classic Q-learning [6] algorithm with Artificial Neural Networks.

We used an ANN to predict the reward function for each possible move (left, right, shoot, press jump, release jump) from a given state. We then take the action with the highest predicted reward. The input of this Neural Network is composed of the current game sensors, the previous 2 game sensor inputs, and the previous 2 moves taken.

The ANN used for experiments has 2 hidden layers with 32 neurons each. Each layer has l2 regularization applied to it, each weight has a decay of 0.01, and each neuron has a Logistic Sigmoid activation function. After each predicted move, we update the neural network using backpropagation. We compute the reward as the difference between the current score and the previous score:

$(prev\_enemy\_health - curr\_enemy\_health) \cdot 0.8 + (curr\_player\_health - prev\_player\_health) \cdot 0.2$

A game ends when either the agent or the enemy lose all life. We have trained the agent on 5000 games. The (arithmetic) average number of frames per game for the best model is 287.

B. Evolving Neural Network Weights with Genetic Algorithms

We used Genetic Algorithms to evolve the weights of fixed-topology ANNs.

1) Sparse Reward Genetic Algorithm: The second algorithm is a sparse reward Neuroevolution [4] algorithm. The reward is defined as "sparse" because an agent doesn't find out how well it's doing until the end of the game, with no feedback during the game. An individual represents the binary encoding of the weights of a Neural Network.

The ANN's role is to predict the next move using the current game sensors, the previous 2 game sensor inputs, and the previous 2 moves. We use 2 hidden layers with 32 neurons for each individual.

We start with randomly initialized ANN weights and then represent them as a bitstring. The weights used for the ANN were values $\in [-2, 2]$, with a precision of 6 digits.

Since we are representing the individuals as bitstrings we were able to apply a Simple Genetic Algorithm [7].

[1]https://github.com/karinemiras/evoman_framework

[2]https://github.com/karinemiras/evoman_framework/blob/master/evoman1.0-doc.pdf

The configurations we have used for the genetic algorithm is:

- population size: 50
- number of generation: 500
- crossover rate: 0.7
- mutation rate: $\{0.008, 0.1\}$
- elitism: 1

We have tried two experiments with different mutation rates in order to observe if a high mutation rate can lead to good results for the problem, since the role of this GA was chiefly exploratory in our ensemble.

The solution of the genetic algorithm is the best individual from the last generation.

2) Iterative Genetic Algorithm: The next algorithm is an iterative neuroevolution. It is "iterative" because the agents are trained on a small number of game steps first and then the number of game steps slowly increases.

After a number of generations we increase the number of game timesteps the agents are allowed to train on. The fitness function was scaled based on the number of game timesteps in a way that if an agent training on x game timesteps will always have a fitness lower than an agent training on y game timesteps if x is lower than y.

C. Searching Neural Network Weights with Particle Swarm Optimization

We have used Particle Swarm Optimization [8] to search for the weights of a fixed-topology ANN maximize the fitness function defined at (2). We used the same ANN topology as with the GA.

The configurations used for the PSO algorithm are:

- population size: 30
- number of iterations: 200
- cognitive weight: $\{0.4, 0.8, 1.5\}$
- social weight: $\{0.8, 0.4, 3\}$
- inertia weight: {constant 1, decreasing from 1 by 0.0035 every epoch}

The same separation between a sparse and an iterative search was made in the case of PSO, as in the case of the GA.

We also tried to search with PSO using an uniformly decreasing inertia weight, starting at 1 and ending at 0.3.

D. Proximal Policy Optimization

We have devoted significant computational resources to a PPO [10] algorithm, expecting it to exploit and refine solutions. For this stage we have used the default difficulty of 2, the training was done on 4 opponents and the evaluation was done as described in the Problem Description (Section II).

The configuration we have used for the PPO is:

- randomly initialized weights
- hidden layer sizes: $(64, 64)$
- steps per epoch: 10000
- epochs: 3000
- gamma: 0.99
- clip_ratio: 0.2
- pi_lr: 3e-4
- vf_lr: 1e-3
- train_pi_iterations: 80
- train_v_iterations: 80
- lambda: 0.97
- target_KL: 0.01

E. Particle Swarm Optimization Cascaded With Proximal Policy Optimization

The output weights of a neural network resulted from a PSO search is the starting agent for PPO. In order to solve the generalized problem we have made the following changes from the PSO from the exploratory stage:

- The sizes of the hidden layers were increased from (32, 32) to (64, 64).
- It is trained on four opponents, not only one.
- The fitness function is the one from the Problem Description (Section II).

The same configuration was used for PPO as in the case of the PPO with random weights initialization.

IV. Experimental Investigation

A. Q-Learning

The Q-learning algorithm was the starting point of our comparison. Even when trained and evaluated against the same opponent, it would lose every game while inflicting almost no damage.

B. Genetic Algorithms

The iterative genetic algorithm leads to much better results than Q-learning (Fig. 2).

We have shown that a mutation rate of 0.008 leads to better results than a mutation rate of 0.1 (Fig. 3).

The results of the sparse GA and the iterative GA are not significantly different (Fig. 4).

C. Particle Swarm Optimization

Both PSO algorithms lead to much better results than either of the sparse and iterative GA. The iterative PSO lead to worse results than the sparse PSO. In Fig. 4 we can see that the best first-stage algorithm is the sparse PSO.

The time advantage of iterative PSO vs sparse PSO is not significant (Fig. 5).

We have also searched for good PSO weights for our problem. In the next stages we have used a cognitive weight of 0.4 and a social weight of 0.8 due to its higher variance in results (Fig. 6).

We have also tried two PSO inertia weight updates (Fig. 7). We continued with the constant inertia weight approach.

After deciding what the starting model should be, we have tested its performance (Fig. 8) and run time (Fig. 9) on multiple difficulties.

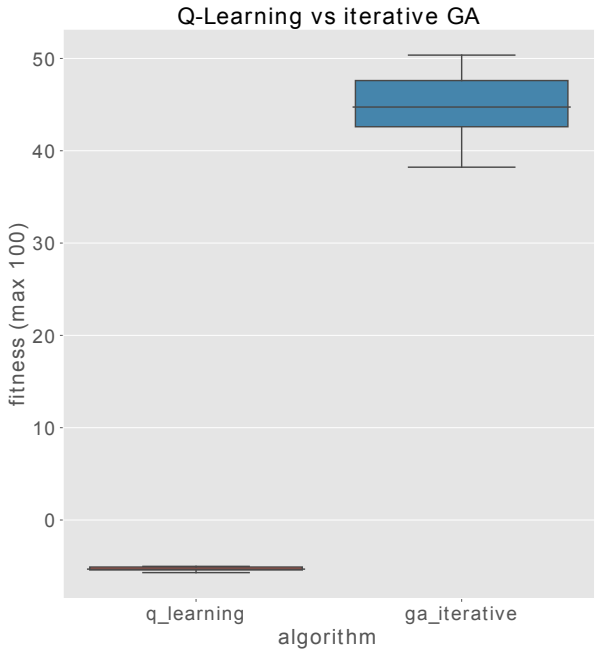

Fig. 2. The fitness comparison between Q-learning and iterative genetic algorithms.

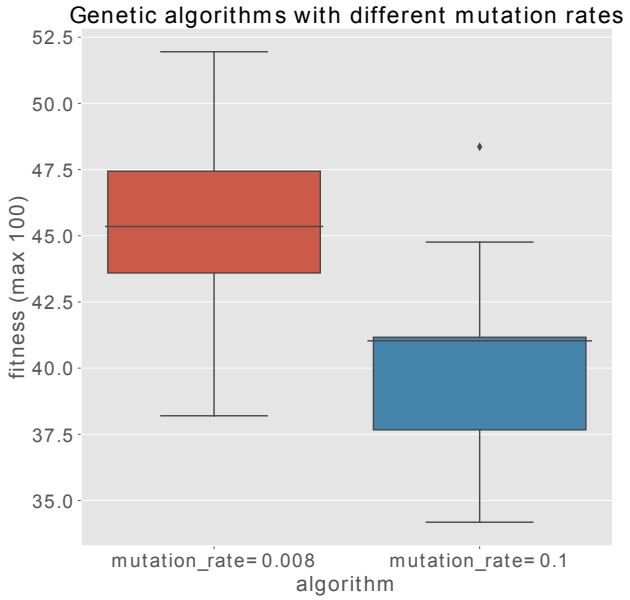

Fig. 3. The fitness comparison between sparse genetic algorithms with mutation rate of 0.1 and 0.008.

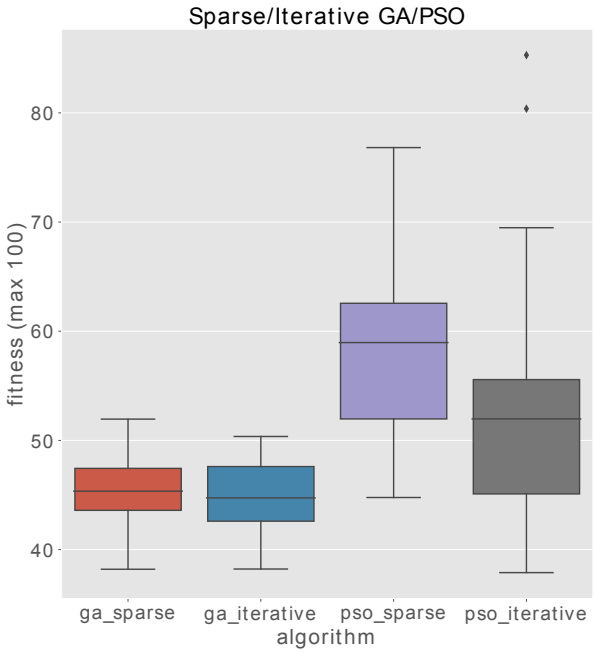

Fig. 4. The fitness comparison between the iterative and the sparse approached, both for the genetic algorithms and the PSO.

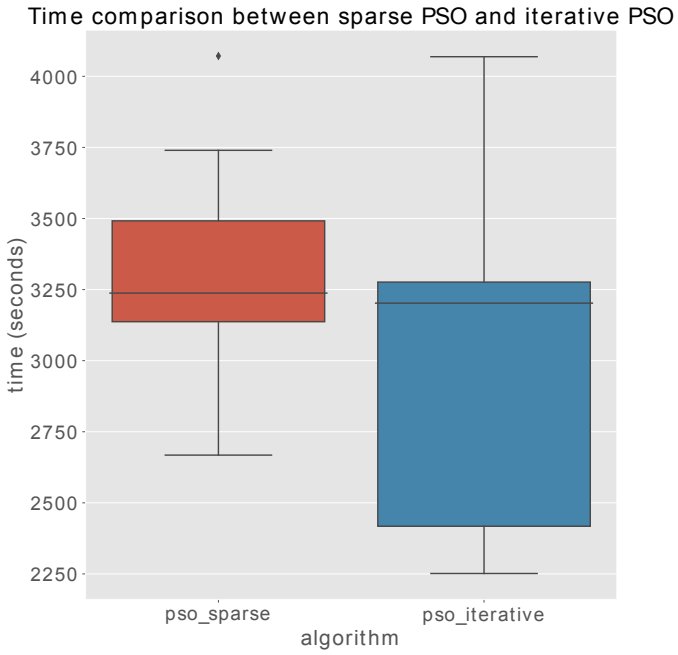

Fig. 5. The time comparison between sparse and iterative PSO.

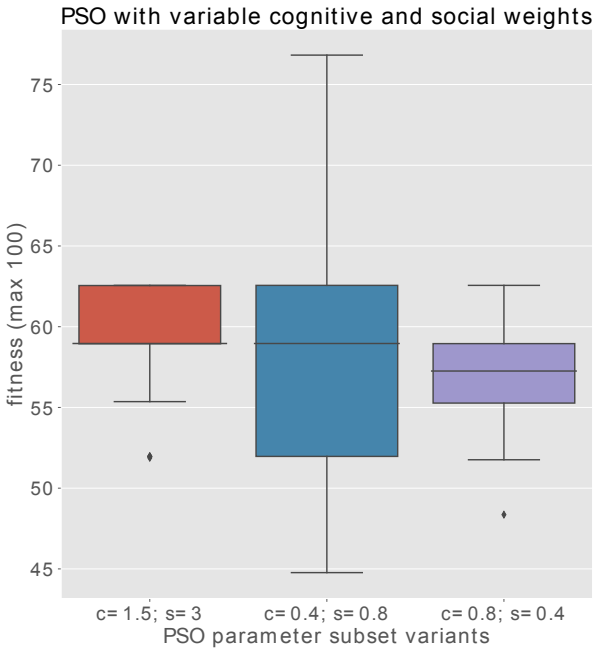

Fig. 6. The fitness comparison between sparse PSO with different cognitive and social weights.

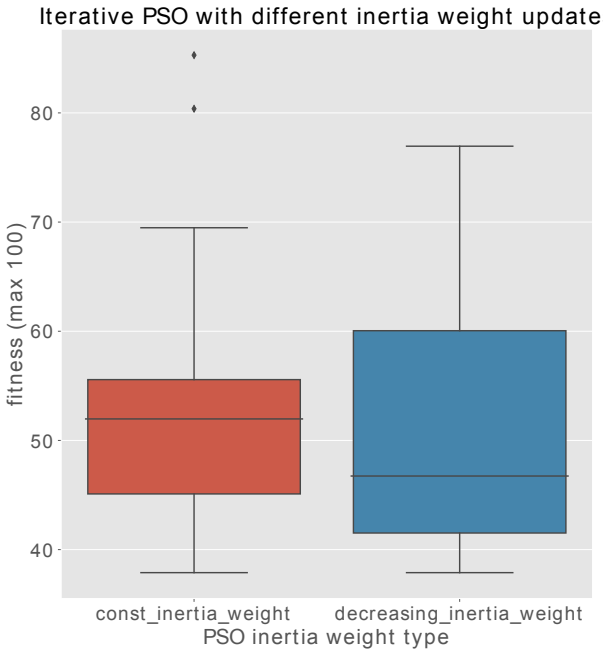

Fig. 7. The fitness comparison between PSO with constant inertia weight and with uniformly decreasing in time inertia weight.

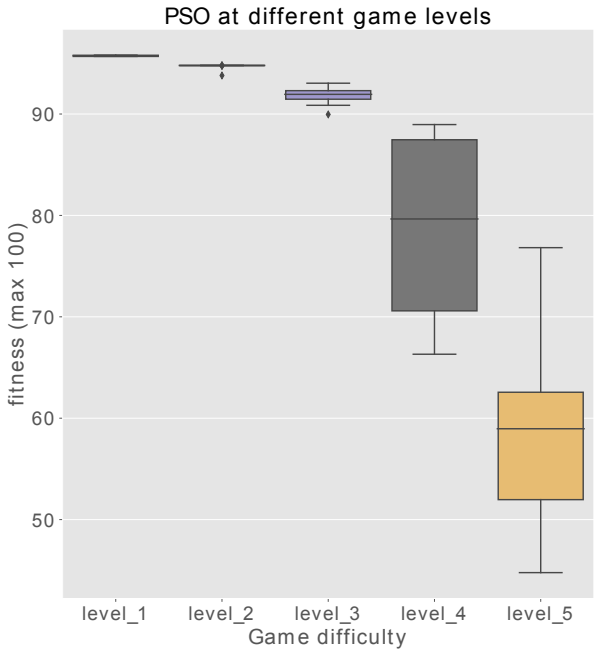

Fig. 8. The fitness comparison of PSO against different game difficulty levels.

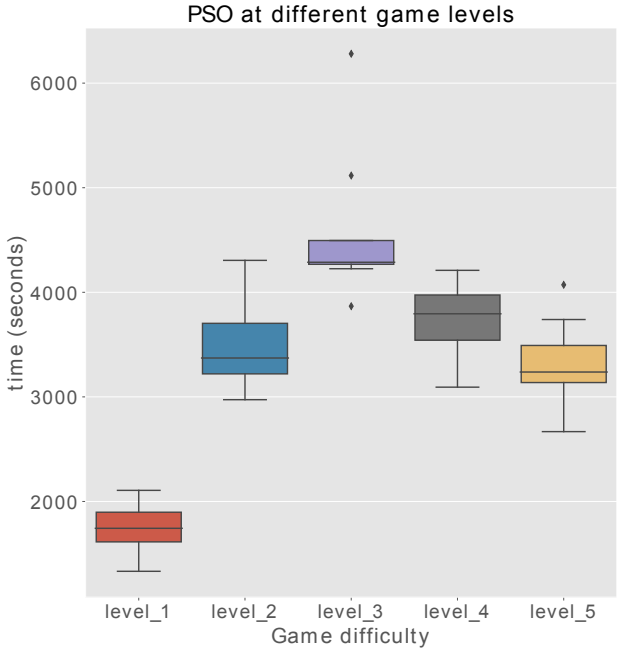

Fig. 9. The time comparison of PSO at different game difficulty levels.

V. Results

For evaluation of an agent, we ran 30 games against each opponent, leading to 8 averages (one per opponent), of which

we computed the harmonic mean which is the final result of the agent.

A. Best Combination of Opponents for Training

We searched for the best 4 opponents to serve as a basis for generalization. The first subset considered was $\{1, 2, 6, 7\}$, which was chosen empirically after manually playing against every opponent and choosing the ones which exhibited the most general behaviors our agents could learn, in our opinion. The gain on this subset is 125.37. Due to limited time, instead of searching all other subsets, we sought to invalidate our choice: by adding other opponents to the subset, we thought they could have increased gains, and guide us to choose another starting subset.

TABLE I
RESULTS FOR VARIOUS OPPONENTS WHEN STARTING WITH A PRE-TRAINED PPO MODEL ON ENEMIES $\{1, 2, 6, 7\}$

Train opponents	gain (harmonic mean)
$\{1,2,6,7\} \cup \{3,4,5,8\}$	44.16
$\{1,2,6,7\} \cup \{3,4,6,7\}$	75.3
$\{1,2,6,7\} \cup \{1,3,6,7\}$	110.19
$\{1,2,6,7\} \cup \{2,3,6,7\}$	136.1
$\{1,2,6,7\} \cup \{2,4,6,7\}$	85.22

The only combination that lead to a better score than 125.37 was $\{2, 3, 6, 7\}$. We ran an experiment with PPO with random initialization with the combination of train enemies $\{2, 3, 6, 7\}$ and we obtained a final gain of 99.97. This means that we did not find a better combination of training opponents than $\{1, 2, 6, 7\}$, so it is the one we used for the following experiments.

B. Random Initialization PPO vs PSO Cascading PPO

The best PSO configuration was used in cascade before the PPO in 4 runs. PPO with random initialization was ran 3 times. The small number of runs is due to the long training time. The worst result of the PPO with random initialization is far better than the best result of PSO cascaded with PPO (Fig. 10). Our explanation for getting worse results when using PSO+PPO is that the search space is vast, and we haven't allowed PSO enough exploration time. Another possible explanation would be that the landscape is misleading when it comes to searching for generalist versus specialized solutions. This can be more easily explained in game-playing terms: the game-playing strategies which are learned quickly might not be only suboptimal, but antithetic to the optimal game-playing strategies.

We could try solving this problem by giving more exploratory power to the PSO, but we did not experiment with this due to the lack of time and good performance of the PPO.

C. PPO in the best configuration

Considering the results above, we decided that PPO with random initialization and training enemies $\{1, 2, 6, 7\}$ would be the best configuration. We ran 3000 epochs (chosen *a priori*) in this configuration, saving all the models along the way after every 250 epochs (since we anticipated overfit). *After* looking at the testing results, we noticed that the final *test* gain is greater after 2000 epochs than after 3000 epochs (Fig. 11). This means that there is overfitting during the training. We can conclude that by stopping the PPO algorithm after 2000 epochs instead of 3000 epochs we can end up with more generalized agents that perform better on average against all opponents.

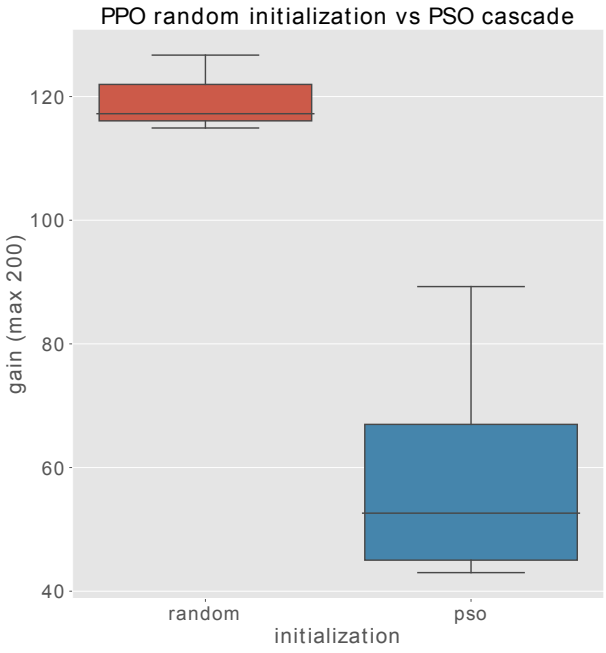

Fig. 10. Gain comparison between PPO with random initialization and with PSO cascading.

D. Best Train Agent vs Best Tested Agent

The best train gain was obtained in the first run of the PPO with random initialization. The train gain was computed as the harmonic mean of the averages per opponent of the 30 games played against the enemies $\{1, 2, 6, 7\}$. The highest train gain is 198.4. When computed against all opponents, this agent has an overall gain of 117.22. Out of the three PPO with random initialization runs, the highest final gain is 127.01 (but they had lower training gains, due to overfit). After the experiments were concluded we also looked at the overall gains of the agents before 3000 epochs (Fig. 12). For the runs 2 and 3 the snapshots of the models were saved every 250 epochs. In the first run these periodic snapshots start after 2000 epochs. When looking at the results of the intermediary agents we observed that there are some high peaks in test gain. The highest test gain was observed during the third run after 1750 epochs and it has a value of 137.18.

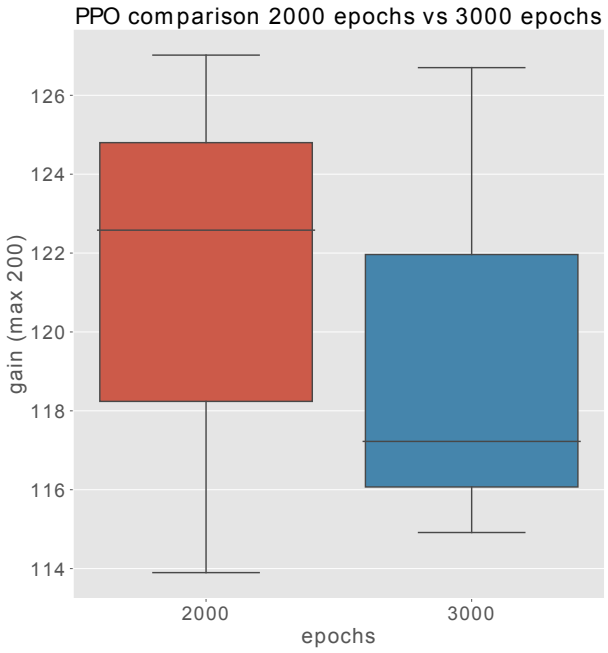

Fig. 11. Gain comparison between PPO after 2000 epochs and after 3000 epochs.

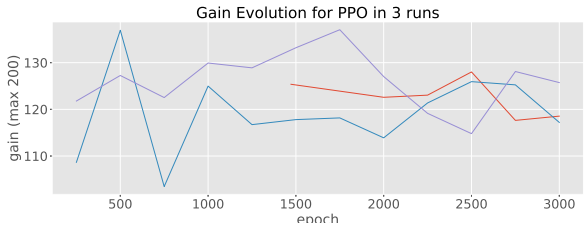

Fig. 12. Gain Evolution for 3 PPO runs, each point is the harmonic average of 30 evaluations

E. Comparison with the Upper Bound

With the PPO algorithm trained on the opponents $\{1, 2, 6, 7\}$ we have beat the best specialized models from the original paper which were given as upper bounds [2]. While the NEAT approach giving the upper bound is specialized on each opponent (thus, 8 specialized solutions for 8 opponents), a single PPO solution trained on 4 opponents manages to enlarge this bound, on the 4 opponents it trained on.

Looking at the average survival rates for our algorithm, we can also see near-perfect survival rates against the training opponents, but also some generalization. Opponents 3 and 4, however, were very difficult, while opponents 5 and 8 apparently had behaviors similar to those we trained on.

F. Time Analysis

The tests were run on an i7 4750HQ processor, on 3 threads. The median game time during training is 28 seconds. The average number of frames per game is 287.

TABLE II
SPECIALIZED NEAT VS GENERALIZED PPO (GAIN)

Opponent	**PPO**				**Specialized** NEAT
	Run 1 3000 ep best train	***Run 2*** 3000 ep	***Run 3*** 3000 ep	***Run 3*** 1750 ep best test	
1	198.41	199.07	199.81	199.61	190.01
2	199.74	199.27	190.34	189.14	194.01
3	58.94	46.67	70.27	85.01	180.01
4	58.34	66.5	64.11	80.17	194.01
5	172.61	165.65	174.61	158.83	194.01
6	195.73	196.95	195.85	194.37	173.01
7	199.79	195.05	193.27	191.17	177.01
8	122.17	145.49	141.89	140.61	186.01
$\{1, 2, 6, 7\}$	198.4	197.57	194.75	193.49	183.09
$\{1, \ldots, 8\}$	117.22	114.91	127.01	137.18	185.67

TABLE III
SPECIALIZED NEAT VS GENERALIZED PPO (PERCENTAGE OF GAMES WON)

Opponent	**PPO**			
	Run 1 3000 ep best train	***Run 2*** 3000 ep	***Run 3*** 3000 ep	***Run 3*** 1750 ep best test
1	100	100	100	100
2	100	100	100	100
3	3.33	0	3.33	46.66
4	0	6.66	0	6.66
5	100	96.66	100	100
6	100	100	100	100
7	100	93.33	100	100
8	80	100	96.66	90

VI. CONCLUSIONS

After observing that the sparse methods were better than the iterative ones, PPO with random initialization is better than PSO cascaded with PPO and PPO with random initialization trained on four opponents leads to better train results than specialized agents trained with NEAT we can conclude that the problem space is in such a way that the methods with a greedy bias are moving away from the global optimum.

REFERENCES

[1] Evoman: Game-playing Competition for WCCI 2020, http://pesquisa.ufabc.edu.br/hal/Evoman.html
[2] Fabricio Olivetti de Franca, Denis Fantinato, Karine Miras, A.E. Eiben and Patricia A. Vargas. "EvoMan: Game-playing Competition" arXiv:1912.10445
[3] de Araújo, Karine da Silva Miras, and Fabrício Olivetti de França. "An electronic-game framework for evaluating coevolutionary algorithms." arXiv:1604.00644 (2016).
[4] Floreano, D., Dürr, P. & Mattiussi, C. Neuroevolution: from architectures to learning. Evol. Intel. 1, 47–62 (2008). https://doi.org/10.1007/s12065-007-0002-4
[5] M. MEGA, "Produced by capcom, distributed by capcom, 1987," System: NES.
[6] Watkins, C.J.C.H., Dayan, P. Q-learning. Machine Learning 8, 279–292 (1992)
[7] Holland J.H., Genetic Algorithms and Adaptation. Adaptive Control of Ill-Defined Systems, 1984, Volume 16 ISBN 978-1-4684-8943-9

[8] Kennedy, J.; Eberhart, R. (1995). ”Particle Swarm Optimization”. Proceedings of IEEE International Conference on Neural Networks. IV. pp. 1942–1948.

[9] Kenneth O. Stanly; Rist Miikkulainen (2002). ”Evolving Neural Networks through Augmenting Topologies”. Evolutionary Computation, Volume 10, Issue 2, Summer 2002, p.99-127, https://doi.org/10.1162/106365602320169811

[10] John Schulman, Filip Wolski, Prafulla Dhariwal, Alex Radford, Oleg Klimov (2017) ”Proximal Policy Optimization Algorithms”, arXiv:1707.06347v2

[11] Karine Miras, Evoman, https://karinemirasblog.wordpress.com/portfolio/evoman/