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

3rd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

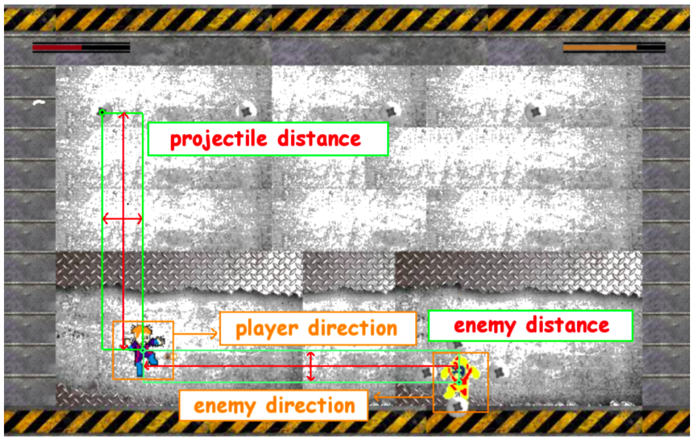

Fig. 1. Sensors available for the player.

***Abstract*—Abstract is last.**

***Index Terms*—game-playing agent, artificial intelligence, EvoMan, genetic algorithm, reinforcement learning, q-learning, neuroevolution, particle swarm optimization, proximal policy optimization, ppo**

I. Introduction

Introduction is also last.

II. Problem Description

A. Environment

EvoMan [1], [2] is a framework for testing competitive game-playing agents. This framework is inspired by Mega Man II [4], 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 actions which the agent may take are:

- walk left
- walk right
- jump
- shoot
- release of the jump

The lives of the player and the enemy start at 100. Everytime one of them gets hit, their life deplenishes. Whoever's life reaches 0 loses the game.

In the original 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. The framework is freely available[1] and it is currently compatible with Python 3.6 and 3.7. There is also an extensive documentation available[2].

B. Problem

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

$$gain = 100.01 + player\_life - enemy\_life$$

The highest final gain is 185.67 [1] and it was obtained with the NEAT [8] algorithm.

The problem we are trying to solve is a generalization of the one above. We are to train an agent against four enemies and then test its performance against all eight. 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.

III. Approach

Our intention was to develop a cascade ensemble method where the resulting model of an algorithm which has a high exploratory bias is the starting point of an algorithm with high exploitative bias. The exploratory algorithms we have considered are Q-Learning [5], genetic algorithms [6] and particle swarm optimization [7]. The exploitative algorithm we used was Proximal Policy Optimization [9].

To find the best exploratory algorithm we have made the following changes:

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

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

- we have increased the difficulty from the default 2 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 [10] to:

$$\mathbf{0.9 \cdot (100 - \mathit{enemy\_life}) + 0.1 \cdot \mathit{player\_life} - \mathit{ln}(\mathit{nr\_of\_game\_timesteps})}$$

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

A. Q-Learning

The first algorithm is a classic Q-learning [5] algorithm with neural networks.

We used a neural network to predict the reward function for each possible move (left, right, shoot, jump, release of 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 and the previous 2 game sensors with the moves taken at that specific point.

The neural network used for experiments has 2 hidden layers with 32 neurons each. Each layer has l2 regularization applied with a weight decay of 0.01 and sigmoid activation. After each predicted move, we updated the neural network using backpropagation, using as input the game sensors and as output the true reward function. A game ends when either the agent or the enemy lose all life. We have trained the agent on 5000 games. The average number of frames per game for the best model is (todo: count in game steps).

B. Evolving Neural Network Weights with Genetic Algorithms

We used genetic algorithms to evolve the weights of neural networks with fixed structures.

1) Sparse Reward Genetic Algorithm: The second algorithm is a sparse reward neuroevolution [3]. 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 is represented as the weights of a neural network.

The neural network role is to predict the next move using the current game sensors and the previous 2 game sensors with the moves taken at that specific point. We used 2 hidden layers with 32 neurons for each individual.

We start with randomly initialized neural network weights and then represent them as a bitstring. The weights used for the neural network were values between 2 and -2 with a precision of 6 digits.

For evaluation, the bitstring is transformed into the weights of the neural network and a game is played, from which we can obtain the individual fitness.

Since we are representing the individuals as bitstrings we were able to apply a simple genetic algorithm [6].

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 whether a very high mutation rate can lead to good results for the problem.

The solution of the genetic algorithm is the best individual from the last generation. Since we have used an elitism of 1, this means that the solution of the genetic algorithm is the best individual ever evaluated.

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 [7] to search for the weights of a neural network of a fixed structure which maximize the fitness defined in this section. We used the same neural network configuration as in the case of genetic algorithms.

The configurations used for the particle swarm optimization [7] 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, uniformly decreasing from 1 to 0.3 (decreasing 0.0035 every iteration)}

The same separation between an sparse and an iterative search was made in the case of particle swarm optimization, as in the case of the genetic algorithms.

We also tried to search with pso using an uniformly decreasing inertia weight with respect to the percent of the iterations passed, having values from 1 to 0.3. This was tried to make the PSO focus even more on exploitation, rather than exploration.

D. Proximal Policy Optimization

PPO [9] is the algorithm we have used for exploitation. 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(

The neural network is randomly initialized.

The configuration we have used for the PPO is:

- hidden layers: (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 (

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.
We have shown that a mutation rate of 0.008 leads to better results than a mutation rate of 0.1 (Fig.
The results of the sparse genetic algorithm and the iterative genetic algorithm are not significantly different (Fig.

C. Particle Swarm Optimization

Both particle swarm optimization algorithms lead to much better results than either of the sparse and iterative genetic algorithms. The iterative PSO lead to worse results than the sparse PSO. In Fig.
The time advantage of iterative PSO vs sparse PSO is not significant (Fig.
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.
We have also tried two PSO inertia weight updates (Fig.
After deciding what the starting model should be, we have tested its performance (Fig.

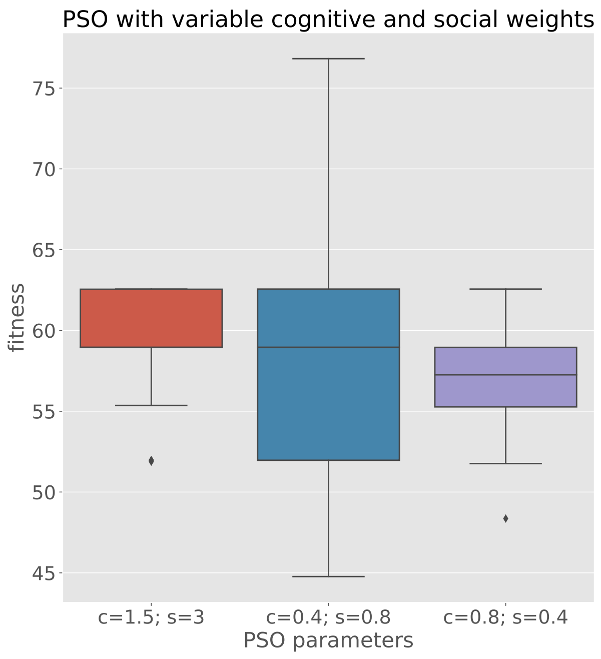

Fig. 2. The fitness comparison between sparse PSO with different cognitive and social weights.

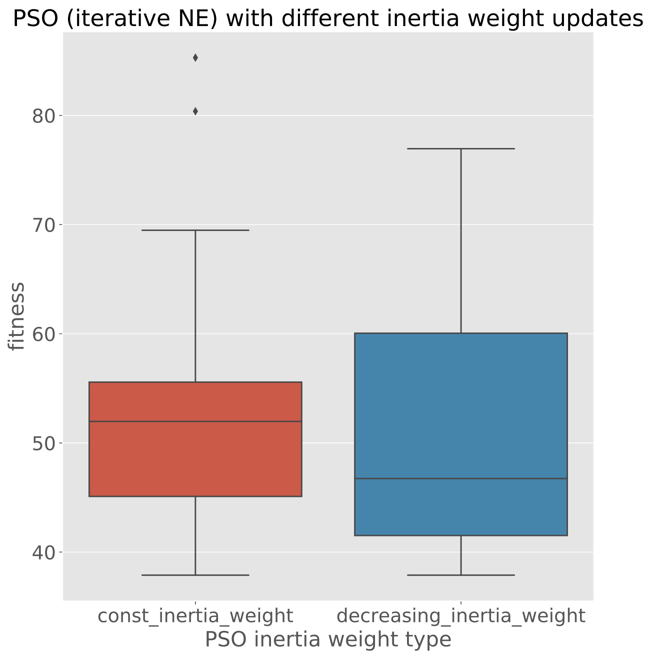

Fig. 3. The fitness comparison between PSO with constant inertia weight and with uniformly decreasing in time inertia weight.

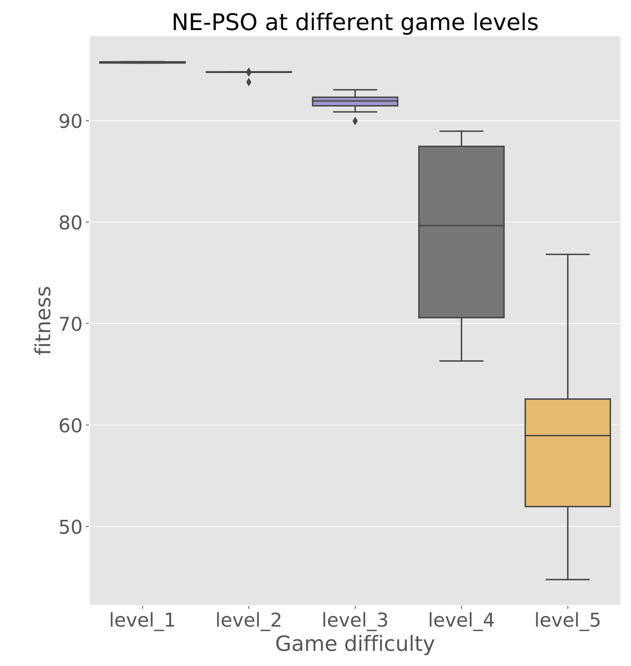

Fig. 4. The fitness comparison of PSO against different game difficulty levels.

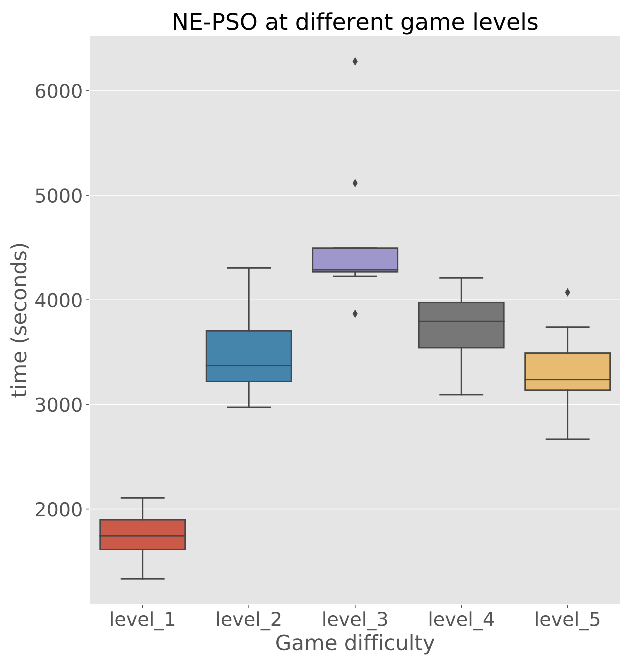

Fig. 5. The time comparison of PSO at different game difficulty levels.

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 opponents that lead to the best results. The first combination of train opponents we considered was $\{1, 2, 6, 7\}$ because (todo: explain). We observed its final gain as 125.37. We realized a set of exploratory experiments where we would search for other combinations of train opponents, but starting with the resulting model of PPO trained for 1000 epochs on the combination of opponents $\{1, 2, 6, 7\}$. If by starting with a pre-trained model another combination of train enemies does not lead to better results, then we can believe that the respective combination does not lead to better results than $\{1, 2, 6, 7\}$.

TABLE I
Various train opponents for pre-trained PPO (2000 epochs)

Train opponents	gain (harmonic mean)
$\{3, 4, 5, 8\}$	44.16
$\{3, 4, 6, 7\}$	75.3
$\{1, 3, 6, 7\}$	110.19
$\{2, 3, 6, 7\}$	136.1
$\{2, 4, 6, 7\}$	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 train 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 worst result of the PPO with random initialization is higher than the best result of PSO cascaded with PPO (Fig.

C. *PPO 2000 epochs vs PPO 3000 epochs*

The final gain is greater after 2000 epochs than after 3000 epochs (Fig.

D. *Best Tested Agent*

The highest gain for a tested agent is 137.18. It was obtained after 1750 epochs of one of the PPO runs with random initialization. Only against opponents 3 and 4 this agent sometimes loses.

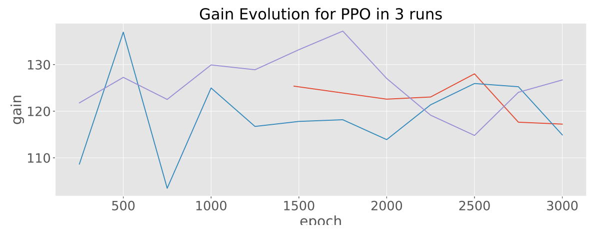

Fig. 6. Gain Evolution for PPO in 3 runs.

TABLE II
BEST AGENT AGAINST ALL OPPONENTS

Opponent	**Average gain (30 games)**
1	199.61
2	189.143
3	85.01
4	80.17
5	158.83
6	194.37
7	191.17
8	140.61

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 [1]. NEAT specialized

TABLE III
SPECIALIZED NEAT VS GENERALIZED PPO ON TRAIN OPPONENTS

Opponent	**Specialized NEAT**	**Generalized PPO**
1	190.01	199.81
2	194.01	190.34
6	173.01	195.85
7	177.01	193.27

The generalized PPO was trained on the opponents {1, 2, 6, 7}

TABLE IV
SPECIALIZED NEAT VS GENERALIZED PPO

Opponent	**NEAT**	**PPO Run 1**	**PPO Run 2**	**PPO Run 3**	**PPO Best**
1	190.01	198.41	199.07	199.81	199.61
2	194.01	199.74	199.27	190.34	189.14
3	180.01	58.94	46.67	70.27	85.01
4	194.01	58.34	66.5	64.11	80.17
5	194.01	172.61	165.65	174.61	158.83
6	173.01	195.73	196.95	195.85	194.37
7	177.01	199.79	195.05	193.27	191.17
8	186.01	122.17	145.49	141.89	140.61

F. Time Analysis

Since the match time is relevant, the machine on which the games were run is also relevant. The tests were run on an i7 4750HQ processor. The training was done on 3 threads. All computations were done on the CPU. The median game time during training is 28 seconds. (todo: include median nr of epochs at testing)

REFERENCES

[1] Fabricio Olivetti de Franca, Denis Fantinato, Karine Miras, A.E. Eiben and Patricia A. Vargas. ”EvoMan: Game-playing Competition” arXiv:1912.10445

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

[3] 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

[4] M. MEGA, ”Produced by capcom, distributed by capcom, 1987,” System: NES.

[5] Watkins, C.J.C.H., Dayan, P. Q-learning. Machine Learning 8, 279–292 (1992)

[6] Holland J.H., Genetic Algorithms and Adaptation. Adaptive Control of Ill-Defined Systems, 1984, Volume 16 ISBN 978-1-4684-8943-9

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

[8] 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

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

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