EvoMan - Framework 1.0

Documentation

Karine Miras

karine.smiras@gmail.com

January 31, 2016

Contents

1 Introduction

EvoMan framework is a platform for testing optimization algorithms, that has been developed in Federal University of ABC in Santo Andre / Sao Paulo / Brazil, during the research of a masters dissertation.

This document intends presenting an overview of the research results, but mainly providing details about how to use the platform for making experiments similar to the ones presented.

2 EvoMan Framework

The framework code can be downloaded in: https://github.com/karinemiras/evoman_framework

2.1 Platform structure

EvoMan was developed in Python using pygame library, and provides **8** predator-prey games (Fig. 1) inspired in the electronic game **MegaMan II**, and is expected to simulate a dinamic environment, i.e., that does not present always the same behaviour.

2.1.1 Controlling sprites

Both prey (player) and predator (enemy) may be controlled by an Artificial Intelligence **algorithm** (agent), or assume its **default** behaviour.

For the **player**, the default behaviour is waiting for commands from some input device. For the **enemy**, the default behaviour is an static attack composed by fixed rules, different for each enemy.

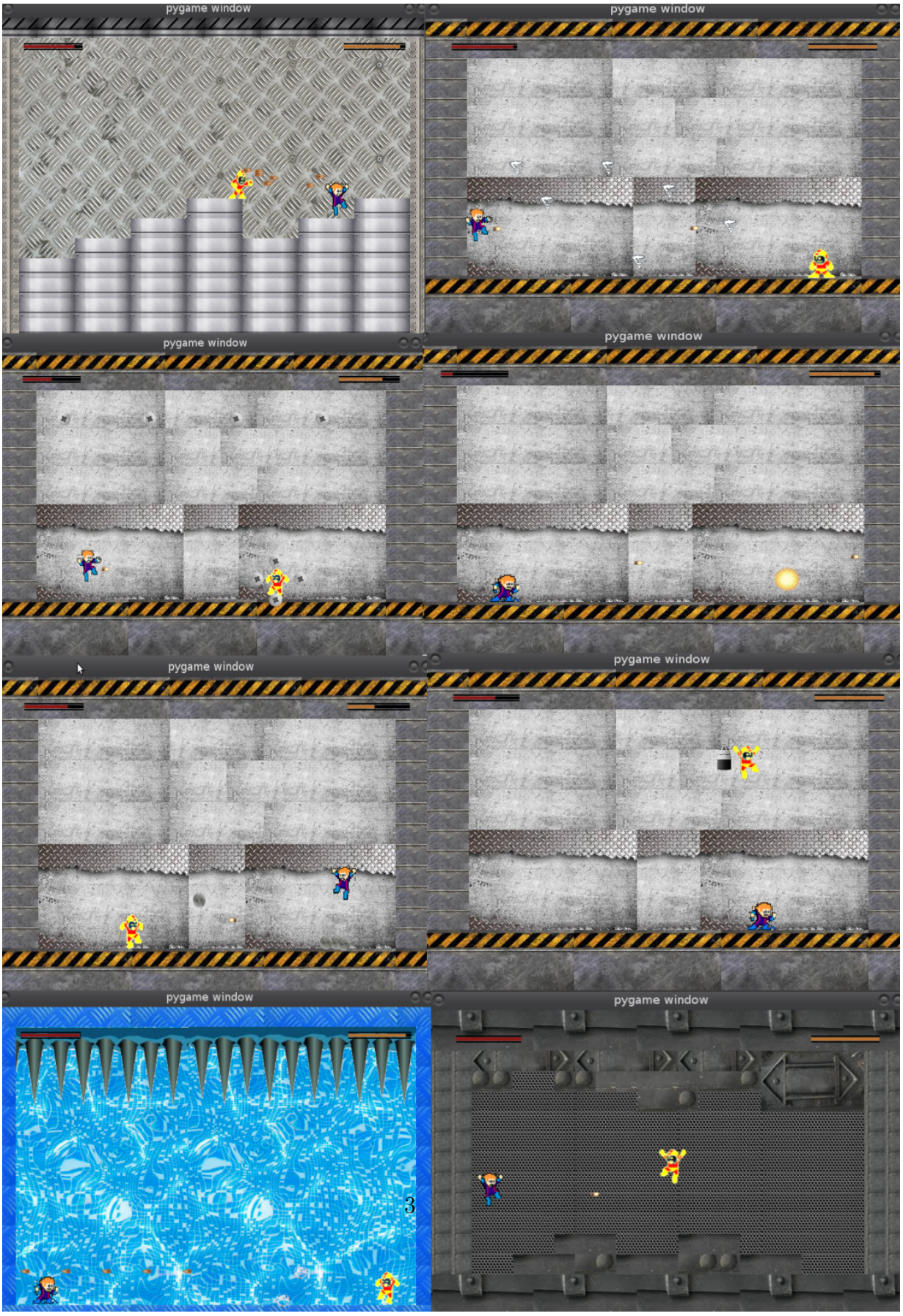

Figure 1: Games

Combining the controlling methods for enemy and player gives us interesting simulation **modes**, as shown in Figure 2.

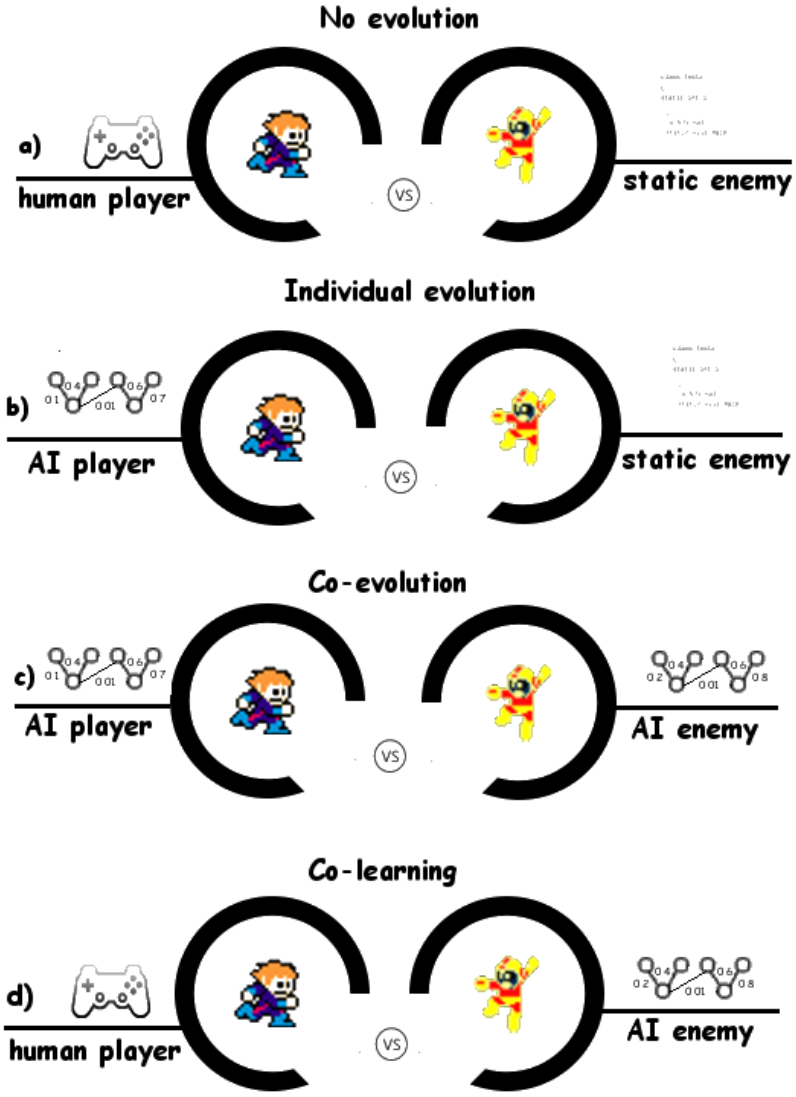

Figure 2: Simulation modes

(a) **No-evolution**: runs as normal the game way for testing, controlling player by human-operated input device, and enemies by static attacks defined as fixed rules.

(b) **Individual evolution**: for simulating player evolution, controlling the player

by a user-provided algorithm, and enemies by static attacks defined as fixed rules.

(c) **Co-evolution**: for simulating both player and enemy evolutions, controlling them by user-provided algorithms.

(d) **Co-learning**: for training algorithm against human, controlling the player by human input-device and enemy by user-provided algorithm.

2.1.2 Training objectives

When training an agent, it is possible to choose between **single** or **multiple** objectives (Fig. 3).

If you choose single objective, you will be training an **specialist** agent against one enemy. If you choose multiple objective, you will be training a **generalist** agent, fighting against more than one enemy in the same training phase.

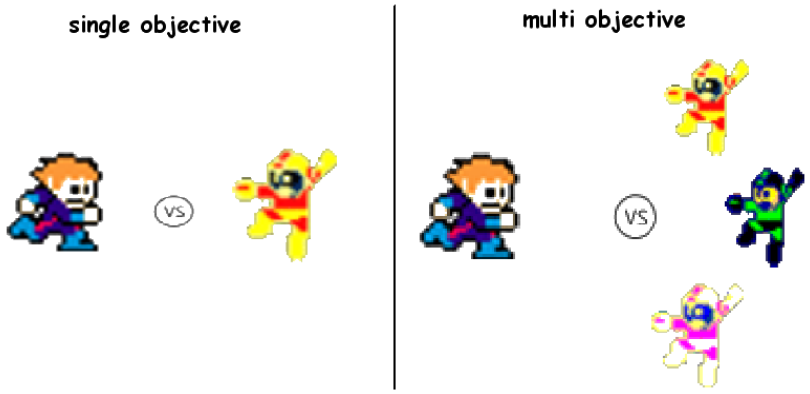

Figure 3: Training objectives

2.1.3 Sensors

The controllers have access to **20 sensors** (Fig. 4). They are: 16 for distances from projectiles; 2 for distances from enemy; 2 for sprites' directions.

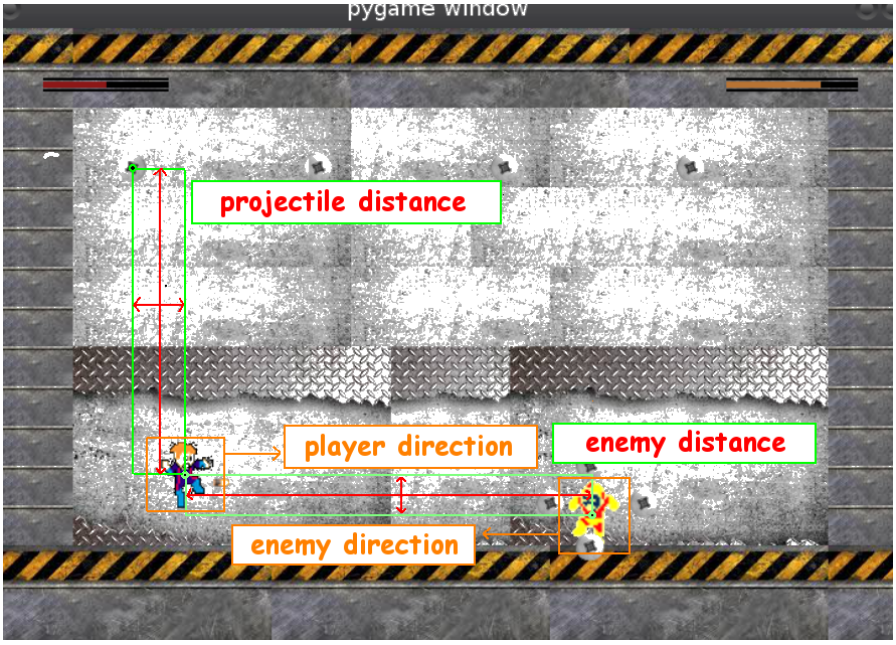

Figure 4: Sensors for sprites

2.1.4 Running an experiment

Using the framework for running experiments can be done as depicted in Figure 5. The optimization algorithm must look for solutions to the problem (single or multiple games), and everytime a solution is tested, it runs the game loop.

During the game loop, agent controllers receive the sensors state at each game time step, and take decisions for all its possible actions.

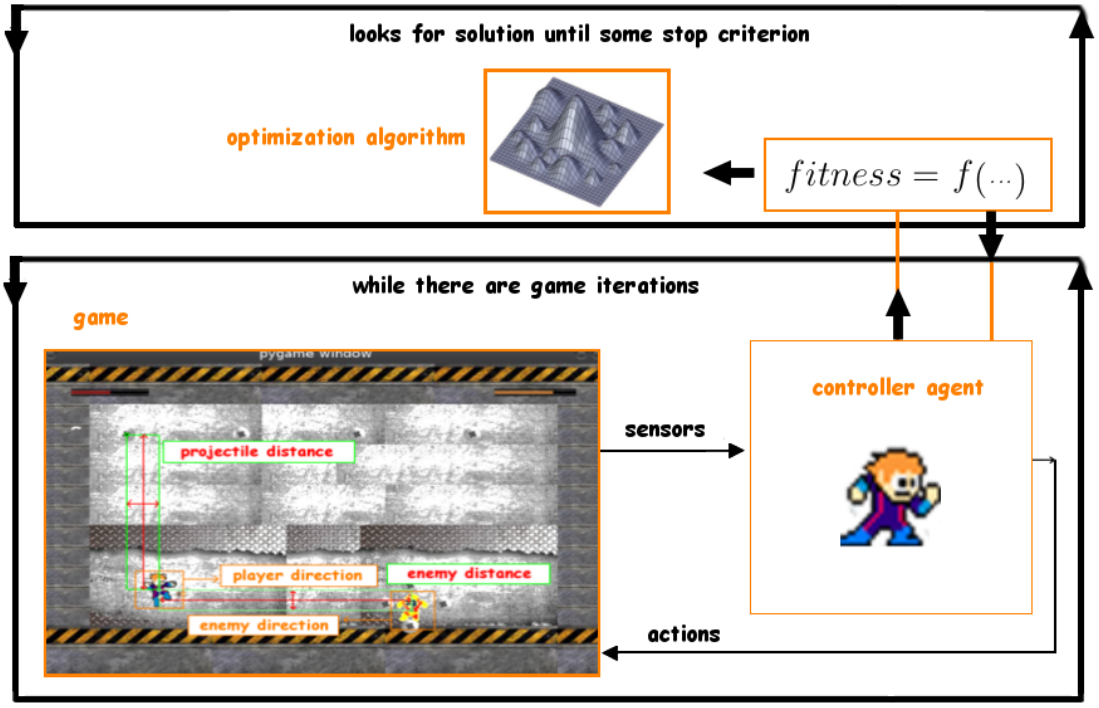

Figure 5: Experiment stream

2.2 Classes

2.2.1 Environment class

Environment is the main class of the framework. It may be instantiated as it is or implemented as needed.

When instantiating the class with no new implementations and providing no parameters, it will use default fitnesses and configuraion.

```
from environment import Environment
env = Environment()
env.play()
```

Figure 6: Environment class simple instantiation

Parameters

The list below describes all parameters for this class. Parameters are named and assume default values when not provided in the instantiation.

- **multiplemode**: whether objective is multiple or single. Multiple objective demands informing multiple games in the 'enemies' parameter. It may assume [yes] for multiple mode or [no] for single mode. Default value is [no].
- **enemies**: games list. Must be an array from 1 to 8 positions containing numbers from 1 to 8. Default value [[8]].
- **loadplayer**: excludes player sprite from the game, useful for testing. May assume [yes] for keeping player or [no] for excluding player. Default value is [yes].
- **loadenemy**: excludes enemy sprite from the game, useful for testing. May assume [yes] for keeping player or [no] for excluding player. Default value is [yes].
- **level**: integer number for defining game difficulty level. The greater the level, harder the game is. Default value [1].
- **playermode**: mode for controlling player. May be [human] for controlling by keyboard/joystick, or [ai] for controlling by algorithm. Default value [ai].
- **enemymode**: mode for controlling enemy. May be [static] for controlling by fixed rules, or [ai] for controlling by algorithm. Default value [ai].
- **speed**: game run velocity. May be [normal] for 30 frames a second or [fastest] for the maximum speed possible in pygame. Default value [normal].
- **inputscoded**: applies several transformations to agent sensors, increasing the number of variables. May be [yes] for coding and [no] for not coding. Default value [no].
- **randomini**: positions enemy in random initial positions in screem randomly each game run. May be [yes] for varying position and [no] for fixed position. Default value [no].

- **sound**: turn on game sound. May be [on] for sound on and [off] for sound off. Sound only works in player human mode. Default value [on].
- **contacthurt**: defines what sprite is injured in body contact. May be [player] for injuring player or [enemy] for injuring enemy. Default value [player].
- **logs**: turns on printing logs. May be [on] for printing or [off] for not printing. Default value [on].
- **savelogs**: turns on exporting logs to file evoman_logs.txt. May be [yes] for exporting or [no] for not exporting. Default value [yes].
- **timeexpire**: integer number for defining game run maximum duration. Useful for avoiding unnecessary long game runs. The greater the time, longer the run is. Default value [3000].
- **overturetime**: integer number for defining time for starting and finishing a game. Overture only happens in player human mode. The greater the time, longer the overture takes. Default value [100].
- **clockprec**: precision of time measuring. Pygame's time counter is not totally accurate, which means games will not present exactelly the same behaviour every run, as time counting is used for changing game frames. This is useful for simulating an uncertain environment. Value may be [low] for using the low precision function clock.tick() function or [medium] for using the medium precision function clock.tick_busy_loop(). Default value [low].
- **player_controller**: a Controller class object representing the controller for player sprite. Default value is an object of the original Controller class, where a default method is implemented taking random actions for the sprite.
- **enemy_controller**: a Controller class object representing the controller for enemy sprite. Default value is an object of the original Controller class, where a default method is implemented taking random actions for the sprite.

Methods

- **update_solutions**(self, solutions): receives a list containing the experiment results found so far and saves it in the current game state. The user may store anything wanted in it.

- **update_parameter**(self, name, value): any parameter must be updated using this method. 'name' is the name of the parameter and 'value' is de new value it will assume.

- **get_num_sensors**(self): returns the number of sensors configured for the controllers.

- **save_state**(self, name): saves the current game state, containing solutions and configured parameters in files evoman_solstate_'name' and evoman_paramstate_'name'.txt respectively, in which 'name' is the name to be given to the state.

- **load_state**(self, name): loads a saved game state as current, setting parameters and solutions backup, where 'name' is the name of the state saved.

- **get_playerlife**(self): returns the number of energy points kept by the player in the last run.

- **get_enemylife**(self): returns the number of energy points kept by the enemy in the last run.

- **get_time**(self): returns the duration of the last run, in game steps.

- **play**(self, pcont=None, econt=None): executes a game run using the solutions given as 'pcont'(player) and 'econt'(enemy) for the controllers. When using the default controllers these parameters are not necessary, as agents will take random actions.

  When in single objective mode it returns: fitness, player life, enemy life and game run time, in this order as single values. When in multi objective mode it returns: fitness, player life, enemy life and game run time, in this order as 4 lists containing values of each game (lists are ordered by the order of enemies parameter provided).

- **fitness_single**(self): this method should be **implemented** and replaced to define a different fitness function for training agents in single objective mode. The default fitness function is expressed by Equation 1.

$$fitness = \gamma * (100 - e_e) + \alpha * e_p - \log t \tag{1}$$

where e_e, e_p are the energy measures of enemy and player respectively, varying from 0 to 100; t is number of time steps of game run; and constants γ and α assume values 0.9 and 0.1 respectively.

- **cons_multi**(self, values): this method should be **implemented** and replaced to define a different fitness function for training agents in multiple mode. The default function is expressed by Equation 2.

  This method must consolidate all fitnesses of single objective solutions into one only fitness. 'values' is a list with the fitnesses calculated by fitness_single(self) method for every game configured in the experiment.

$$fitness' = \bar{f} - \sigma \tag{2}$$

where $fitness'$ is the consolidated *fitness* by $\bar{f}$ and σ, that are the mean and the standard deviation of fitnesses against the m enemies chosen for training.

2.2.2 Controller class

This is the class to be implemented when defining a controller structure.

The **control**(self, params,cont) method should be to **implemented** and replaced to define the agent controller structure. If the you do not implement this method, the agent will use the default method, taking random actions.

The 'params' param received is a numpy array containing all the 20 sensors' values, to be read at each game time step.

When implementing control methods for the controllers, the returned variables

at each game time step must be booleans, and the number of variables may vary for each sprite:

- The player controller may take **5 actions**: walk left, walk right, jump and shoot.
- The enemies 1, 2, 3 and 4 controllers may take **4** actions, defined as attack 1 until attack 4.
- The enemy 5 and 6 controller may take **3** actions, defined as attack 1 until attack 3.
- The enemy 7 and 8 controller may take **6** actions, defined as attack 1 until attack 6.

3 Dissertation summary

Title: Neuroevolution for the construction of a generic strategy with EvoMan environment.

University library link for dissertations and theses (in portuguese):
http://biblioteca.ufabc.edu.br/index.html

3.1 Objectives

Problem: generalizing learning for multiple problems
Application: electronic games

General Video Game Playing (GVGP) is a recent research field that aims creating autonomous generic controllers able to play different games.

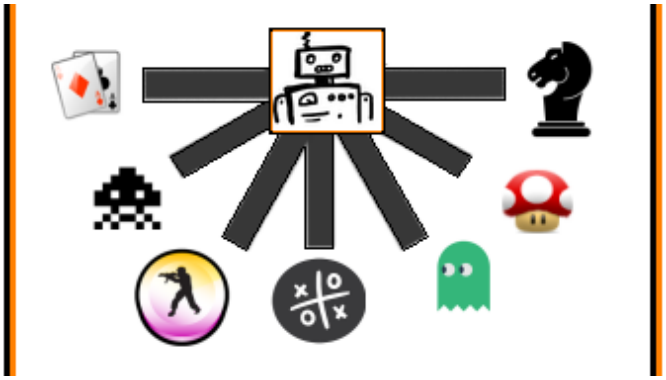

Figure 7: GVGP

Objective 1 - Creating a testing environment that permits training autonomous agent controllers using electronic games in different simulation modes.

Objective 2 - Testing NeuroEvolution algorithms to create specialist (single game) and generalist (generalist) agents using EvoMan.

3.2 Methodology

For assessing solutions found by the optimization algorithm, it is necessary to define a success measure, usually called fitness.

The fitness defined for single objective solutions was the default one. Also for multi objective solutions a default fitness was used. The greater the fitness, better the solution is.

The optimization method used was NeuroEvolution, that trains and defines neural networks using evolutionary algorithms. The method was applied in two ways:

- Evolving only network weights for perceptron and multilayer (10 and 50 neurons): Genetic Algorithms and LinkedOpt algorithm
- Evolving network weights and topology: NEAT algorithm

The **success** criterion for solutions was:

- **Effectiveness** is to zero the power of the enemy
- **Efficiency** is the amount of energy preserved by player

Solutions can be **classified** as:

- **Bad solutions** do not win the game (unsuccessfull)
- **Medium solutions** win the game preserving less than 50 points of energy (successfull)
- **Good solutions** win the game preserving more than 50 points of energy (successfull)

3.3 Results

For specialist solutions, some games presented greater challenge than others, but all games were shown to be beatable (Fig. 8).

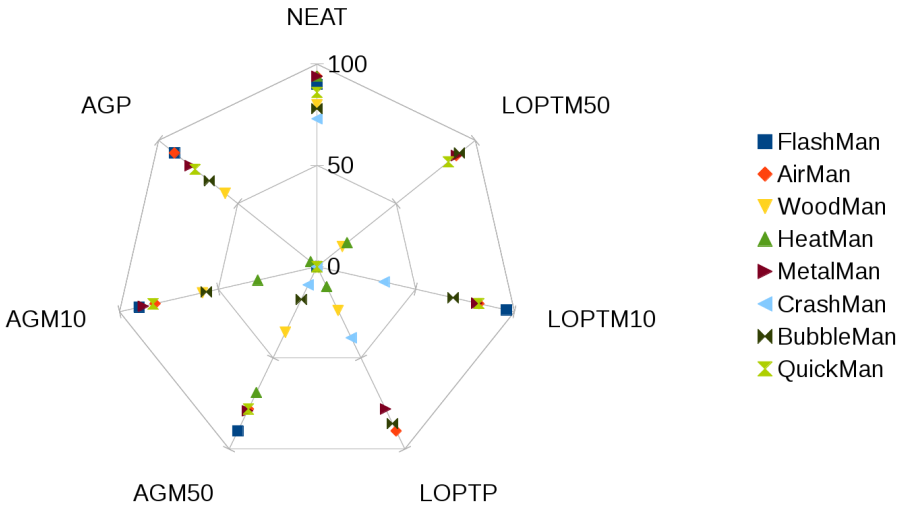

Figure 8: Algorithms performance by game for single objective solutions.

For generalist solutions, several groups of games were composed and tested. From the total of 8 games, all (28) possible pairs of games were combined and

tested using NEAT algorithm as a reference, and from the best pairs, some triplet were also tested.

To realize a final generalization test for generic agent controllers, a measure called Gain was used, as shown in Figure 9. The measure goes from -800 to 800, and the greater it is, better is the generalition power of the agent.

$$g = \sum_{i=1}^{n} p_i - \sum_{i=1}^{n} e_i$$

Figure 9: Gain measure

where e is the energy of the enemy, from 0 to 100; p is the energy of the player, from 0 to 100; n=8, is the total number of games in the test.

The best groups obtained by NEAT were:

- pairs: (2, 4) (2 6) (7, 8)
- trios: (2, 6, 5) (6, 5, 1) (2, 5, 1) (6, 7, 4)

These best groups were tested using the other 2 algorithms also. The best final training group found was: 7 and 8. (Fig 11)

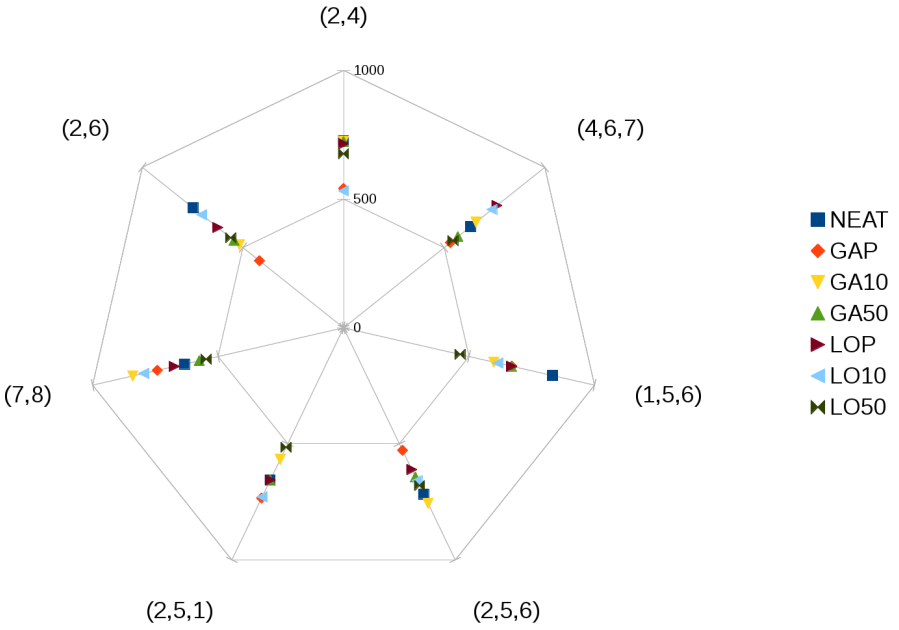

Figure 10: Algorithms performance by game for multiple objective solutions.

The group reached the greatest Gain value (40), by the Genetic Algorithm with a neural network having a hidden layer with 10 neurons. (Fig. 10)

Also this group beated the greatest number of enemies (5 games), by the Genetic Algorithm with a perceptron neural network.

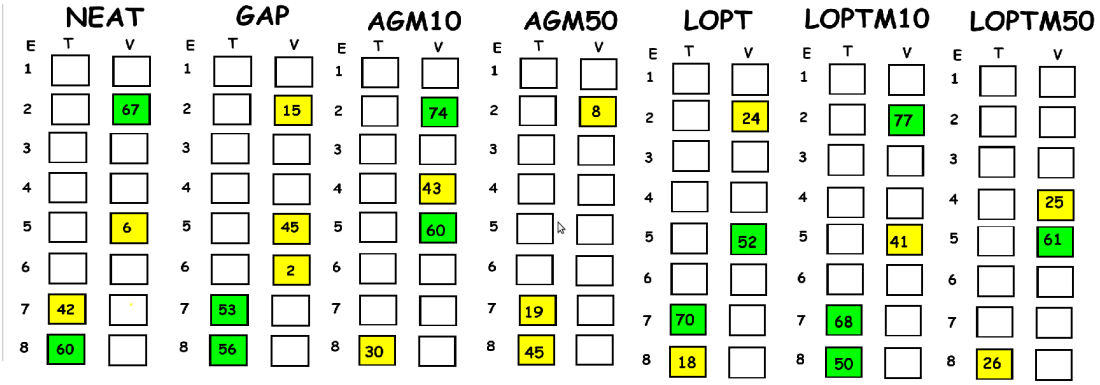

Figure 11: Algorithms performance by game for multiple objective solutions, final energy measures.

Best generalist solutions videos:

youtu.be/w97qitczuL8
youtu.be/vQ_xCShU1gQ
youtu.be/ic68VCjMIMI

4 Demos

4.1 Dummy demo

File: dummy_demo.py

Runs single game with default configuration environment.

4.2 Specialist agent

File: controller_specialist_demo.py
Runs best solutions found by a Genetic Algorihm, for each one of the enemies separately.

4.3 Generalist agent

File: controller_generalist_demo.py
Runs best solution found by a Genetic Algorihm, for all enemies, in general way.

4.4 Individual evolution

File: optimization_individualevolution_demo.py
Runs an evolution process for individual evolution of the player agent controller, using Genetic Algorithms.

4.5 Coevolution

optimization_coevolution_demo.py
Runs an evolution process for co-evolution of player's and enemy's agents, using Genetic Algorithms.

5 Future: framework 2.0

- Create a Game class, making it possible to add new games to the framework enrvironment. The class shall present a group of methods that may allow and help constructing 2D games. This way, games will no longer need to be predator-prey style or even platform.
- Create an Experiment class, providing a set of methods that might help in neuroevolution experiments.