# Neuroevolution for General Video Game Playing

Spyridon Samothrakis, Diego Perez and Simon Lucas
School of Computer Science and Electronic Engineering
University of Essex, Colchester CO4 3SQ, UK
ssamot@essex.ac.uk, dperez@essex.ac.uk, sml@essex.ac.uk

***Abstract*—General Video Game Playing (GVGP) allows for the fair evaluation of algorithms and agents, as it minimiz es the ability of an agent to exploit apriori knowledge in the form of game specific heuristics. In this paper we compare four different, but straightforward, evolutionary algorithms in the new learning stream of the GVGP competition. We show that ”crappy results”**

## I. Introduction

We have to say somewhere that this is the first time this is done for GVG

I'm not sure if this is up to date with the experiments actually run

Learning how to act in unknown environments is often termed the “Reinforcement Learning” problem[**?**]. The problem encapsulates the core of artificial intelligence and has been studied widely under different contexts. Broadly speaking, an agents tries to maximize some long term notion of utility or reward by selecting appropriate actions at each state. Classic RL assumes that state can somehow be identified by the agent in a unique fashion (i.e. the environment has the Markov Property).

A possible way of attacking the problem in the general sense Neuroevolution[**?**]. Neuroevolution adapts adapts the weights of a local or global function approximator (in the form of a neural network) in order to maximize reward. The appeal of neurovolution over traditional gradient-based Reinforcement kerning methods is two fold. First, the gradient in RL problems might be unstable and/or hard to approximate, thus requiring extensive hyper-parameter tuning in order to get any performance. Secondly, the tournament-ranking schemes used by evolutionary approaches are robust to outliers, as they are effectively calculations of medians rather than means. Thirdly, classic RL algorithms (at least their critic-only versions [**?**] can be greatly impacted by the lack of a perfect markov state. Sensor aliasing is a known and common problem in RL under function approximation.

On the other hand the lack of direct gradient information limits the possible size of the parameters of the function approximator to be evolved. If provided with the correct setup and hyperparameters, RL algorithms might be able to perform at superhuman level in a number of hard problems.

In practical terms, research into RL has often been coupled with the use of strong heuristics. General Game Playing allows for completely arbitrary games, making the easy application of temporal based methods non-trivial. On the other hand, modern evolutionary algorithms require minimum fine-tuning and can help greatly with providing a great baseline. In this paper we perform experiments using evolutionary algorithms to learn actors in the following setup.

- CMA-ES using e-greedy policy and linear function approximator
- CMA-ES using e-gredy policy and neural network
- CMA-ES using softmax policy and a linear function approximator
- CMA-ES using softmax policy and a neural network

We will see specifics of this setup later. In order to make this research easy to replicate and interesting in its own right, the environmental substrate is provided by a the new track of the General Game Playing AI Competition. We evaluate our players in this setup.

The paper is structured as follows: blah

## II. Related Research

Everything is so well documented

Include here definitions of GGP, GVGP, first attempts. Also GVGAI competition with the planning track (2014)

## III. The GVGAI Framework

This section describes the framework and games employed in this research.

### A. The framework

The General Video Game Playing competition and framework (GVG-AI) is built on VGDL (Video Game Description Language), a framework designed by Ebner et al. [1] and developed by Tom Schaul [2] for GVGP in Python (*py-vgdl*). In VGDL, there are several *objects* that interact in a two-dimensional rectangular space, with associated coordinates, with the possibility of moving and colliding with each other. VGDL defines an ontology that allows the generation of real-time games and levels with simple text files.

The GVG-AI framework is a Java port of py-vgdl, re-engineered for the GVG-AI Competition by Perez et al. [3], which exposes an interface that allows the implementation of controllers that must determine the actions of the player (also referred to as the *avatar*). The VGDL definition of the game is **not** provided, so it is the responsibility of the agent to determine the nature of the game and the actions needed to achieve victory.

The original version of this framework provided a *forward model*, that allowed the agent to simulate actions on the current state to devise the consequences of the available actions. In this case, the agents counted on $40ms$ of CPU time to decide an action on each game cycle. This version of the framework was employed by the organizers of the competition to run what is now called the "planning track" of the contest, celebrated on the Computational Intelligence in Games (CIG) conference in Dortmund (Germany) in 2014. A complete description of this competition, framework, rules, entries and results has been published in [3].

2015 will feature a new track for this competition, named "learning track". In this track, controllers do **not** have access to forward model, so they cannot foresee the states reached after applying any of the available actions of the game. In contrast, they will be offered with the possibility of playing the game many times, with the aim of allowing the agents to learn how to play each given game.

Information is given regarding the state of the game to the controllers, at three different stages: initialization, at each game step (*act* call, called repeatedly, until the game is finished) and termination. Table I summarizes the information given to the agent and at which stages:

### *B. Games*

Games description

The table with the games is exactly the same...

Maybe I can put a different picture here, not to be repetitive with last CIG.

## IV. Background

Everything is so powerful

### *A. Approaches*

Everything is so cool

## V. Experiments

Everything took so long

## VI. Conclusions

Everything is so pretty

## Acknowledgment

This work was supported by EPSRC grant EP/H048588/1.

## References

[1] M. Ebner, J. Levine, S. Lucas, T. Schaul, T. Thompson, and J. Togelius, "Towards a Video Game Description Language." *Dagstuhl Follow-up*, vol. 6, pp. 85–100, 2013.

[2] T. Schaul, "A Video Game Description Language for Model-based or Interactive Learning," in *Proceedings of the IEEE Conference on Computational Intelligence in Games*. Niagara Falls: IEEE Press, 2013, pp. 193–200.

[3] D. Perez, S. Samothrakis, J. Togelius, T. Schaul, S. Lucas, A. Couëtoux, J. Lee, C.-U. Lim, and T. Thompson, "The 2014 General Video Game Playing Competition," *IEEE Transactions on Computational Intelligence and AI in Games*, p. (to appear) DOI: 10.1109/TCIAIG.2015.2402393, 2015.

| **Information** | **Description** | **Possible Values** | **Init** | **Act** | **End** |
|---|---|---|---|---|---|
| | | **Game Information** | | | |
| **Score** | Score of the game. | $[0..N] \in \mathbb{N}$ | ✓ | ✓ | ✓ |
| **Game tick** | Current game tick. | $[0..1000] \in \mathbb{N}$ | ✓ | ✓ | ✓ |
| **Game winner** | Indicates the result of the game for the player. | $\{won, lost, ongoing\}$ | ✓ | ✓ | ✓ |
| **Is game over** | Indicates if the game is over. | $\{true, false\}$ | ✓ | ✓ | ✓ |
| **World's dimension (width)** | Width of the level (in pixels). | $[0..N] \in \mathbb{R}$ | ✓ | | |
| **World's dimension (height)** | Height of the level (in pixels). | $[0..N] \in \mathbb{R}$ | ✓ | | |
| **Block size (*bs*)** | Number of pixels each grid cell has. | $[0..N] \in \mathbb{N}$ | ✓ | | |
| | | **Actions Information** | | | |
| **Actions** | Available actions of the game. | A subset of $\{nil, up, down, left, right, use\}$ | ✓ | | |
| | | **Avatar Information** | | | |
| **Avatar position ($x$)** | $x$ coordinate (in pixels) of the avatar's position. | $[0..N] \in \mathbb{R}$ | ✓ | ✓ | ✓ |
| **Avatar position ($y$)** | $y$ coordinate (in pixels) of the avatar's position. | $[0..N] \in \mathbb{R}$ | ✓ | ✓ | ✓ |
| **Avatar orientation ($x$)** | $x$ coordinate of the avatar's orientation. | $\{-1, 0, 1\}$ | ✓ | ✓ | ✓ |
| **Avatar orientation ($y$)** | $y$ coordinate of the avatar's orientation.. | $\{-1, 0, 1\}$ | ✓ | ✓ | ✓ |
| **Avatar speed** | Speed of the avatar (pixels/tick). | $[0..N] \in \mathbb{N}$ | ✓ | ✓ | ✓ |
| **Avatar last action** | Last action played by the avatar. | One of $\{nil, up, down, left, right, use\}$ | ✓ | ✓ | ✓ |
| **Avatar resources** | Collections of pairs $< K, V >$, where $K$ is the type of resource and $V$ is the amount owned. | $\{< K_1, V_1 >, < K_2, V_2 >, ...\}\ ; K_i, V_i \in \mathbb{N}$ | ✓ | ✓ | ✓ |
| | | **Grid Information (for each type of sprite)** | | | |
| **Sprite type** | Sprite identifier. | $[0..N] \in \mathbb{N}$ | ✓ | ✓ | ✓ |
| **Grid Bitmap** | Presence array of the sprite type on the grid. 1 means presence, 0 means absence. | Array $\{0, 1\}$ of size $[width/bs, height/bs]$ | ✓ | ✓ | ✓ |

TABLE I: Information given to the controllers for action decision. The controller of the agent receives this information on the three different calls: *Init* (at the beginning of the game), *Act* (at every game cycle) and *End* (when the game is over). Pieces of information only sent on the *Init* call are constant throughout the game played.

| Game | Description | Score | Actions |
|---|---|---|---|
| **Aliens** | Similar to traditional Space Invaders, Aliens features the player (avatar) in the bottom of the screen, shooting upwards at aliens that approach Earth, who also shoot back at the avatar. The player loses if any alien touches it, and wins if all aliens are eliminated. | • 1 point is awarded for each alien or protective structure destroyed by the avatar.<br>• $-1$ point is given if the player is hit. | LEFT, RIGHT, USE. |
| **Boulderdash** | The avatar must dig in a cave to find at least 10 diamonds, with the aid of a shovel, before exiting through a door. Some heavy rocks may fall while digging, killing the player if it is hit from above. There are enemies in the cave that might kill the player, but if two different enemies collide, a new diamond is spawned. | • 2 points are awarded for each diamond collected, and 1 point every time a new diamond is spawned.<br>• $-1$ point is given if the avatar is killed by a rock or an enemy. | LEFT, RIGHT, UP, DOWN, USE. |
| **Butterflies** | The avatar must capture butterflies that move randomly around the level. If a butterfly touches a cocoon, more butterflies are spawned. The player wins if it collects all butterflies, but loses if all cocoons are opened. | • 2 points are awarded for each butterfly captured. | LEFT, RIGHT, UP, DOWN. |
| **Chase** | The avatar must chase and kill scared goats that flee from the player. If a goat finds another goat's corpse, it becomes angry and chases the player. The player wins if all scared goats are dead, but it loses if is hit by an angry goat. | • 1 point for killing a goat.<br>• $-1$ point for being hit by an angry goat. | LEFT, RIGHT, UP, DOWN. |
| **Frogs** | The avatar is a frog that must cross a road, full of tracks, and a river, only traversable by logs, to reach a goal. The player wins if the goal is reached, but loses if it is hit by a truck or falls into the water. | • 1 point for reaching the goal.<br>• $-2$ points for being hit by a truck. | LEFT, RIGHT, UP, DOWN. |
| **Missile Command** | The avatar must shoot at several missiles that fall from the sky, before they reach the cities they are directed towards. The player wins if it is able to save at least one city, and loses if all cities are hit. | • 2 points are given for destroying a missile.<br>• $-1$ point for each city hit. | LEFT, RIGHT, UP, DOWN, USE. |
| **Portals** | The avatar must find the goal while avoiding lasers that kill him. There are many portals that teleport the player from one location to another. The player wins if the goal is reached, and loses if killed by a laser. | • 1 point is given for reaching the goal. | LEFT, RIGHT, UP, DOWN. |
| **Sokoban** | The avatar must push boxes so they fall into holes. The player wins if all boxes are made to disappear, and loses when the timer runs out. | • 1 point is given for each box pushed into a hole. | LEFT, RIGHT, UP, DOWN. |
| **Survive Zombies** | The avatar must stay alive while being attacked by spawned zombies. It may collect honey, dropped by bees, in order to avoid being killed by zombies. The player wins if the timer runs out, and loses if hit by a zombie while having no honey (otherwise, the zombie dies). | • 1 point is given for collecting one piece of honey, and also for killing a zombie.<br>• $-1$ point if the avatar is killed, or it falls into the zombie spawn point. | LEFT, RIGHT, UP, DOWN. |
| **Zelda** | The avatar must find a key in a maze to open a door and exit. The player is also equipped with a sword to kill enemies existing in the maze. The player wins if it exits the maze, and loses if it is hit by an enemy. | • 2 points for killing an enemy, 1 for collecting the key, and another point for reaching the door with it.<br>• $-1$ point if the avatar is killed. | LEFT, RIGHT, UP, DOWN, USE. |

TABLE II: Games in the training set of the GVGAI Competition, employed in the experiments of this paper. All games also include the possibility of using the NIL action.