Neuroevolution for General Video Game Playing

Spyridon Samothrakis, Diego Perez-Liebana, Simon M. Lucas and Maria Fasli
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, mfasli@essex.ac.uk

***Abstract*—General Video Game Playing (GVGP) allows for the fair evaluation of algorithms and agents as it minimizes the ability of an agent to exploit apriori knowledge in the form of game specific heuristics. In this paper we compare four possible combinations of evolutionary learning using Separable Natural Evolution Strategies as our evolutionary algorithm of choice; linear function approximation with *Softmax* search and ϵ-*greedy* policies and neural networks with the same policies. The algorithms explored in this research play each of the games during a sequence of 1000 matches, where the score obtained is used as a measurement of performance. We show that learning is achieved in 8 out of the 10 games employed in this research, without introducing any domain specific knowledge, leading the algorithms to maximize the average score as the number of games played increases.**

I. Introduction

Learning how to act in unknown environments is often termed the "Reinforcement Learning" problem [1]. 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, but it is often the case that function approximators are used to help with state spaces that cannot be enumerated.

A possible way of attacking the problem in the general sense is Neuroevolution (for an example in games see [2]). Neuroevolution 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 Learning methods is three-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 [3]) 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 one 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 quick baseline. In this paper we perform experiments using evolutionary algorithms to learn actors in the following setup.

Among evolutionary algorithms, for problems with continuous state variables, a simple yet robust algorithm is Separable Natural Evolution Strategies (S-NES) [4]. Coupling the algorithm with different policy schemes and different approximators yields the experimental methods:

- S-NES using e-greedy policy and a linear function approximator.
- S-NES using e-greedy policy and a neural network.
- S-NES using softmax policy and a linear function approximator.
- S-NES using softmax policy and a neural network.

The motivation behind this is to check whether there is an exploration method which works better in a general sense, without misguiding the evolution. We will see specifics of this setup in later sections. In order to make this research easy to replicate and interesting in its own right, the environmental substrate is provided by the new track of the General Video Game Playing AI Competition[1]. The players evaluated in this research are tested in this setup.

The paper is structured as follows: Section II does a brief review of the current state of the art in General Game and Video Game Playing. Section III analyses the framework used in this paper for evaluating games. We describe the algorithms used in Section IV, while in Section V we present the experimental results. We conclude with a discussion in Section VI.

II. Related Research

While there has been considerable research in the topic of both Neuroevolution and General Game Playing (GGP), GGP [5] was initially mostly concerned with approaches that included access to a forward model. In this scenario, agents (or players) have the ability to predict the future to a certain extent and plan their actions, typically in domains that are more closely aligned to board games and are of a deterministic nature.

Recently, a competition termed "The General Video Game Playing Competition" [6] has focused on similar problems, but this time in the real-time game domain. The competition provided the competitors with internal models they could sample from (but not access to specific future event probabilities),

[1] www.gvgai.net

while limiting the amount of "thinking" time provided for an agent to 40ms (imposing an-almost-real-time constraint). In both contexts, the algorithms that dominated were sample based search methods [7], [6], also known as Monte Carlo Tree Search (MCTS). In the case of GVG-AI [6], which allows for stochastic environments, Open Loop MCTS methods seem to perform best. A recent paper studies the use of open loop methods, employing both tree search and evolutionary algorithms, in this domain [8].

This competition and framework has brought more research in this area during the last months, after the end of the contest. For instance, B. Ross, in his Master thesis [9], explores enhancements to the short-sight of MCTS in this domain by adding a long term goal based approach. Also recently, T. Nielsen et al. [10] employed different game playing algorithms to identify well and badly designed games, under the assumption that good games should differentiate well the performance of these different agents.

There has also been considerable effort in learning how to play games in a generic fashion outside competitions, using the Atari framework [11], which provides a multitude of games for testing. The games were tackled recently [12] using a combination of neural networks and Q-learning with experience replay. The method was widely seen as a success (though it could still not outperform humans in all games). The methods for these games operate on a pixel level and require a significant amount of training, but the results achieved when successful cover the full AI loop (from vision to actions).

This latest work (by Mihn et al.) is of special interest to this research, as it tackles the problem in a similar way to how it is done here. The agents that play those games have no access to a forward model that allows them to anticipate the state that will be found after applying an action. Learning is performed by maximizing the score obtained in the games after playing them during a number of consecutive trials.

Neuroevolution has been used for real time video games [2], General Game Playing (see [13]) and playing Atari games [14], [15], all with varying degrees of success. In the front of solving Atari games, neuroevolution showed strong results using HyperNEAT [16], a method that evolves all network parameters, including the topology of the network.

Note that the approach taken herein has the goal of just creating a baseline, not to create competitive agents. It is an ad-hoc method for neurovolution with very limited time and almost no feature engineering. If however one dedicates the right effort and fine tunes enough, we expect Reinforcement Learning (e.g., Q-learning) approaches to outperform the current set of agents.

III. The GVGAI Framework

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. [17] and developed by T. Schaul [18] for GVGP in Python (*py-vgdl*). In VGDL, there are several *objects* that interact in a two-dimensional rectangular space, with associated coordinates and 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. [6], 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, which took part during the Computational Intelligence in Games (CIG) conference in Dortmund (Germany) in 2014. A complete description of this competition, framework, rules, entries and results have been published in [6].

2016 will feature a new track for this competition, named "learning track". In this track, controllers do **not** have access to a 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 the possibility of playing the game multiple times, with the aim of allowing the agents to learn how to play each given game. Note that the competition is not limited by some of the drawbacks related to using Atari as an approach (e.g., no true stochasticity [19]) and that it allows for games that can be customised easily, and also allows for great flexibility in the choice of interface between the game andthe AI agent, which could be vision based (as with ALE) or game-object based (as with the current experiments). Thus, we think, it provides different challenges than Atari General Game Playing.

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

B. Games

The games employed for this research are the ones from the training set of the GVG-AI competition. These games are described in detail here:

1) Aliens: Similar to traditional Space Invaders, Aliens features the player (avatar) at 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 the aliens are eliminated. *Scoring:* 1 point is awarded for each alien or protective structure destroyed by the avatar, and -1 point is given if the player is hit.

2) 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

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 that are only sent on the *Init* call are constant throughout the game played.

enemies collide, a new diamond is spawned. *Scoring:* 2 points are awarded for each diamond collected, and 1 point every time a new diamond is spawned. Also, -1 point is given if the avatar is killed by a rock or an enemy.

3) 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. *Scoring:* 2 points are awarded for each butterfly captured.

4) 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. *Scoring:* 1 point for killing a goat and -1 point for being hit by an angry goat.

5) 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. *Scoring:* 1 point for reaching the goal, and -2 points for being hit by a truck.

6) 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. *Scoring:* 2 points are given for destroying a missile and -1 point for each city hit.

7) Portals: The avatar must find the goal while avoiding lasers that kill it. 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. *Scoring:* 1 point is given for reaching the goal.

8) 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. *Scoring:* 1 point is given for each box pushed into a hole.

9) 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). *Scoring:* 1 point is given for collecting one piece of honey, and also for killing a zombie. Additionally, -1 point if the avatar is killed or it falls into the zombie spawn point.

10) 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. *Scoring:* 2 points for killing an enemy, 1 for collecting the key, and another point for reaching the door with it. -1 point is given if the avatar is killed.

In order to show the diversity of these games, Table II shows the features of each one of them. In this table, the `Score System` indicates how the reward is given to the avatar by means of a numeric score. This system can be either **binary** (B) or **incremental** (I). In the former case, the score of the game will always be 0 until victory is achieved, when the reward will become 1. In the latter scenario, points are regularly awarded throughout the course of the game.

The second column, `NPC Types`, indicates the sort of other moving entities that can be found in the game. *Friendly* refers to the Non Player Characters (NPC) that do not harm the player, whereas *Enemies* refer to those NPCs that pose some hazard in the game. Additionally, one game also features more than one type of enemy (*Boulderdash*).

`Resources` indicates if the game contains sprites that can be picked up by the player or not, and `Terminations` relates to how the game can end. A game can finish either

Game	Score System	NPC Types			Resouces	Terminations			Action Set
		Friendly	Enemies	> 1 type		Counters	Exit Door	Timeout	
Aliens	I		✓			✓			A2
Boulderdash	I		✓	✓	✓		✓		A0
Butterflies	I	✓				✓			A1
Chase	I	✓	✓			✓			A1
Frogs	B						✓		A1
Missile Command	I		✓			✓			A0
Portals	B						✓		A1
Sokoban	I					✓			A1
Survive Zombies	I	✓	✓		✓			✓	A1
Zelda	I		✓		✓		✓		A0

TABLE II: Games feature comparative, showing all games employed in this study. Legend: I: Incremental; B: Binary; A0: All moves; A1: Only directional; A2: Left, right and use. Check Section III-B for a full description of the meaning of all terms in this table.

by creating or destroying one or more sprites of a given type (column *Counters*), reaching a goal (*Exit Door*) or when a timer runs out (*Timeout*). Independently, all games can finish eventually (with a loss) it the maximum number of game steps that can be played, set to 1000 for all games, is reached. This limit is introduced in order to avoid degenerate players from never finishing the game.

Finally, each one of these games provides a different set of available actions to the player. The games labelled in Table II as $a0$ provide the complete set of actions ($\{nil, up, down, left, right, use\}$). $A1$ games are those where the actions are only directional (this is, the complete set without the use action). Finally, $A2$ indicates the subset composed by $\{nil, left, right, use\}$.

It is worthwhile highlighting that the variety of these games poses a serious hazard to the playing agents. The learning algorithm needs to learn how the game has to be played in order to achieve victory and maximize score. It is important to realize that the description of the game is not passed to the agent at any time, thus the only way the agent can learn how to play the game is to analyze the game state (maybe by extracting features) and build relations between the observations and the games played.

Of course, one could try to identify the game that is being played and act consequently. However, this is of a limited research interest, as the long term objective is to find agents that can learn any game (rather than building an agent that uses a database of known games, heuristically prepared to play these). Figure 1 shows screenshots of four of the games employed in this research.

IV. Approaches

A. Learning Algorithms

As stated in the introduction, we coupled two different kinds of function approximators with two different exploration policies, and evolved their weights. Assuming input features ϕ and a set of actions A (with each action $a \in A$), we evolved a set of weights for each action w_a:

- $p_a^{linear} = w_a^0 \cdot \phi$ for the linear function approximator.
- $p_a^{nn} = w_a^0 \cdot \max(w_1 \cdot \phi, 0)$ for the rectifier neural network [20].

The types of exploration used are *Softmax* and *ϵ-greedy*. We normalise all inputs on the function approximators between -1 and 1. Notice that the output layer of weights w_a^0 for the neural network approximator is the same for all actions and set to 20 (i.e., the total number of hidden neurons). The number of neurons used is mostly an ad-hoc choice, however it is small enough to be evolved online.

Using p_a we define two different policies:

- $\pi^{\epsilon}(\phi, a) = \begin{cases} 1 - \epsilon + \epsilon/|A| & \text{if } a = \arg\max_{a \in A} p_a \\ \epsilon/|A| & \text{otherwise} \end{cases}$
- $\pi^{softmax}(\phi, a) = e^{p_a} / \sum_{b \in A} p_b$

These two policies where chosen because of their ease of implementation and their popularity - they are the two most common exploration policies. For all our experiments, ϵ is always set to 0.1. All four possible combinations of polices and function approximators are used in the experiments. Training the function approximators involves learning the weights w. We use Separable Natural Evolution Strategies (S-NES) for this [4]. Since we used at least 7 features for each action, and GVG-AI supports 4 actions, we would have a 28 dimensional problem for the linear approximator and a 220 dimensional problem for the neural network appoximator at the very minimum. Since S-NES is an evolution strategy that follows the natural gradient (which can be seen as a gradient that uses KL-divergence as a measure of distance instead of euclidean distance). It assumes that the search parameters come from a multi-variate Gaussian distribution and tries to adapt the Gaussian's mean and covariance-matrix using the natural gradient. The natural gradient allows for information on the uncertainty of the estimates to be incorporated in the gradient search, as well as making the search invariant to transformation. A further transformation on the fitness structure of the problem also takes place; using a population of samples, the fitness is transformed to $[0, 1]$ using a ranking scheme, which makes the algorithm invariant to transformations of the target function as long as these transformations do not impact ranking order. The "Separable" element of the algorithm used here refers to adapting only the diagonal elements of the covariance matrix. This speeds up learning significantly, allowing the S-NES to scale linearly to problems of arbitrary size (instead of quadratically), making the algorithm practical for our situation. For a more detailed explanation of the algorithm, the reader is

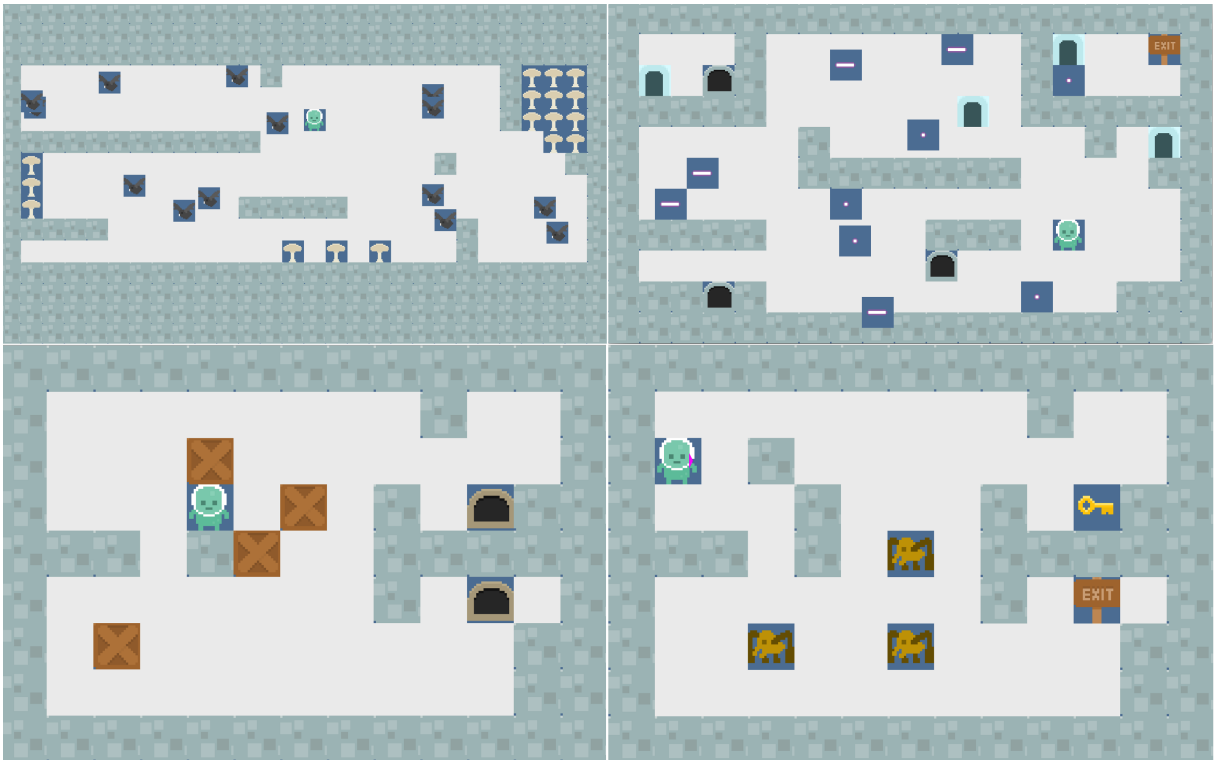


Fig. 1: Four of the ten training set games: from top to bottom, left to right, *Butterflies*, *Portals*, *Sokoban* and *Zelda*.

referred to [4].

B. Game features

All algorithms employed in this experimental study use the same set of features ϕ for learning. These features are extracted from the game state, which is received in the format explained in Section III-A and Table I. The features employed in this research are all of a numeric nature, and they are detailed next:

- Game score.
- Game tick.
- Game winner (-1 if game is still ongoing, 0 if the player lost and 1 if the player won).
- Game over (1 if game is over, 0 otherwise).
- Resources: list of the amount of resources owned by the avatar[2].
- List with the Euclidean distances to the closest sprite of each type.
- Speed of the avatar.

Note that the features are common in all games and are very general (i.e. they do not change from game to game). It is possible however that one might not want to use any features - there is indeed an option and one could use the underlying object structure directly.

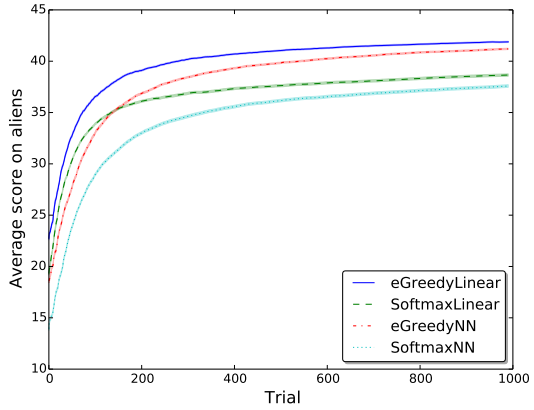
[2]In order to have a fixed number of features, the length of this list is set to 5 (the number of avatar resources can vary from game to game, from none to N types).

V. Experiments

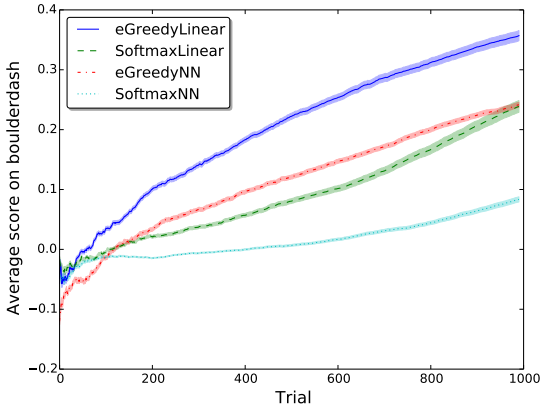
The algorithms proposed in the previous section have been tested in the games described previously in Section III-B. For each one of the 10 present games, the first available level has been used for a learning process where each algorithm played 1000 consecutive times. For each game, 50 experiments where run, and the average of the progression of the score is plotted in Figures 2 and 3.

These figures show the average of score, including the standard error (shadowed area), as it progresses with the number of matches, for each one of the games played. The first aspect to realize is that learning can be appreciated in 8 out of the 10 games employed in the experimental study. It is not possible to declare any of the algorithms as an overall best option, as it generally depends on the game played.

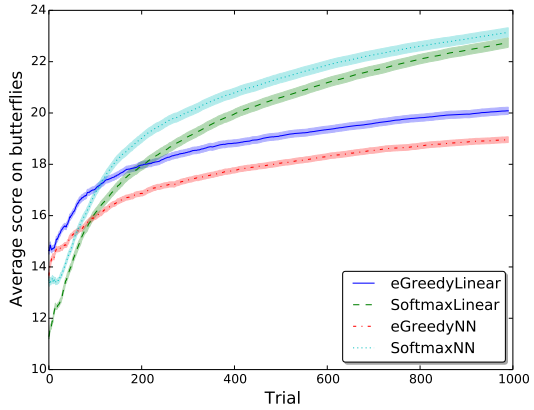
It seems clear that in some games (such as *Aliens*, *MissileCommand* or *SurviveZombies*) the algorithms with an ϵ-*greedy* policy learn better, while in others (such as *Frogs* or *Zelda*) it is the algorithms with a *Softmax* policy the ones with a better performance. For instance, in the game *Frogs*, ϵ-*greedy* algorithms are not able to survive (a score of -2 reflects that the player is killed by a truck), while *Softmax* approaches are capable of learning how to avoid these. Regarding the approximations used, even when the difference is small, the results suggest that the linear approximator works better than the neural network, for both policies tested and most of the games. A possible explanation for this is that the neural network operates in a larger search space, which can potentially inhibit its learning capabilities in the short term.



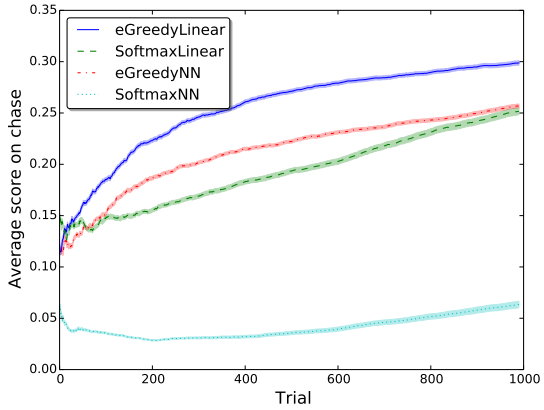
(a) Aliens



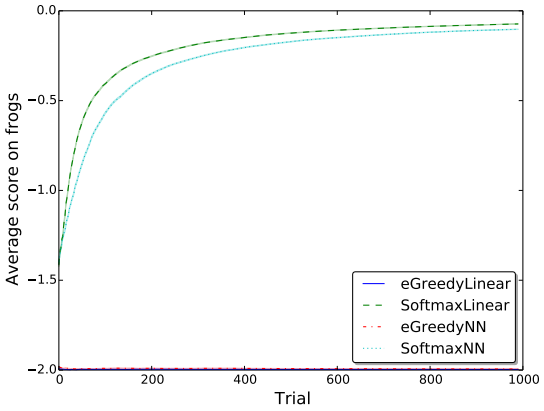
(b) Boulderdash



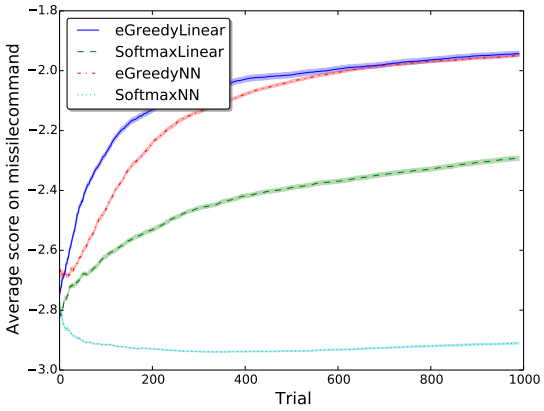
(c) Butterflies



(d) Chase



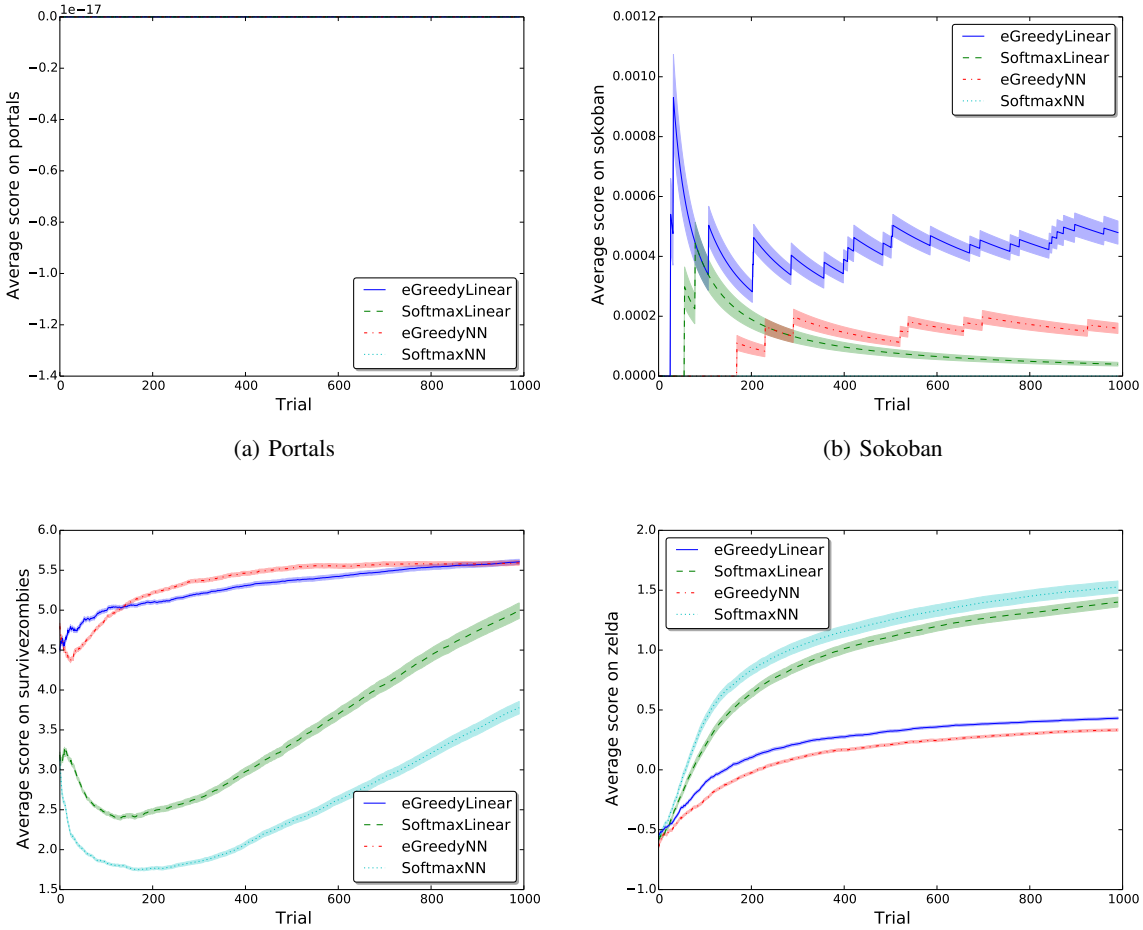
(e) Frogs



(f) Missile Command

Fig. 2: Progression of the average score during 1000 trials. Horizontal axis denotes the trial, while the vertical axis shows the average score. Shadowed area indicates the standard error of the measure. From left to right, top to bottom, the games are *Aliens*, *Boulderdash*, *Butterflies*, *Chase*, *Frogs* and *MissileCommand*.

The cases were learning cannot be achieved by any of the algorithms tested here happen in the games *Portals* (where the score is 0 through the whole process) and *Sokoban* (with an interesting jagged curve). The low performance of all



(a) Portals

(b) Sokoban

(c) Survival Zombies

(d) Zelda

Fig. 3: Progression of the average score during 1000 trials. Horizontal axis denotes the trial, while vertical axis shows the average score. Shadowed area indicates the standard error of the measure. From left to right, top to bottom, the games are *Portals*, *Sokoban*, *SurviveZombies* and *Zelda*.

algorithms in these two games can be due to different reasons.

In the case of *Portals*, the game requires a long sequence of movements (traversing portals) before reaching the goal, which is the only moment where the agent receives positive feedback. As reaching the exit is extremely unlikely, the flat landscape of rewards inhibits the agent from learning how to score points in the game. Interestingly, the hazards in the game that kill the player are well avoided in general. The results show that the time spent in the game (a measure not depicted here for clarity) is close to 60 and 100% of the maximum time allowed (for algorithms with ϵ-*greedy* and *Softmax* policies, respectively). This means that the algorithms, specially the ones with a *Softmax* policy, are able to learn very quickly how to survive in this game, albeit ending with a score of 0.

An explanation for the strange results on *Sokoban* may be related to the features used for learning. In this game, score is awarded when a box is introduced into a hole in the level. With the features extracted from the game state (see Section IV-B), this can be related to minimizing distances between the avatar and the closest box, and between the avatar and the closest hole, both at the same time. The problem is that these minimizations must happen simultaneously, having the avatar pushing in the right direction and with the presence of a hole at the other side of the box. These situations happen rarely, and it is reflected in this game's plot with picks along the curve. As the features only allow the algorithms to determine distances (and not the supplementary needed requirements), they are not able to learn properly and their performance decreases. The game of *Sokoban* has been traditionally one of the most difficult ones from the GVGAI framework and competition [6], as it requires long term plans for its completion.

It is also worthwhile highlighting that in some games, after the 1000 matches, the improvement on the score average has not stopped, which suggests that better results can be found with longer execution trials.

VI. Conclusions

This paper shows a first comparison of different policies (ϵ-*greedy* and *Softmax*) and function approximators, using an evolutionary algorithm (S-NES) as a learning scheme. The objective of the agents that implement these algorithms is to learn how to maximize the score in 10 different games, without the help of any domain specific-heuristic, something that was (at least partially) achieved. Additionally, all these games have different characteristics, which allows us to examine under what conditions learning is performed better. What we have shown is that under most scenarios, using a ϵ-*greedy* algorithm coupled with a linear function approximator is the fastest, surest way to get reasonably good results. This is not that unintuitive, as there are fewer variables to learn and a more "greedy" version of learning can take place, compared to softmax.

In this study, the algorithms tested are able to learn in most of the games tested. In particular, linear function approximators seem to perform best in this setup, although results may vary from game to game, with different algorithms providing a distinct performance. Results also suggest that games requiring a long term plan (such as *Sokoban* or *Portals*) may require a different approach for getting learning to work, one that might include substantial exploration, as the lack of any learning signal proves fatal.

Although the robustness of evolutionary computation is commendable, it might be worth attacking the same problems using more traditional, gradient-based approaches (e.g, Monte Carlo methods, SARSA/Q Learning). This would allow for the training of deeper networks which could possibly help. This could be coupled without using features at all, but rather using the raw bitplanes as input. It is also obvious that more training would be very beneficial for a number of games.

Finally, an important point to make here is that effectively testing and learning are happening on the same levels. Including more levels (and also increasing the size of the game set used for experiments) for each game will bring further details about how general the algorithms tested in this research are, and will allow for a deeper analysis on their learning abilities in the domain of General Video Game Playing. This highlights the difficulty of the domain where one has to learn extremely quickly, without much access to heuristics and/or an environmental model.

References

[1] R. S. Sutton and A. G. Barto, *Introduction to reinforcement learning*. MIT Press, 1998.

[2] K. O. Stanley, B. D. Bryant, and R. Miikkulainen, "Real-time neuroevolution in the nero video game," *Evolutionary Computation, IEEE Transactions on*, vol. 9, no. 6, pp. 653–668, 2005.

[3] S. P. Singh, T. Jaakkola, and M. I. Jordan, "Learning without state-estimation in partially observable markovian decision processes." in *ICML*, 1994, pp. 284–292.

[4] T. Schaul, T. Glasmachers, and J. Schmidhuber, "High dimensions and heavy tails for natural evolution strategies," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. ACM, 2011, pp. 845–852.

[5] M. Genesereth, N. Love, and B. Pell, "General game playing: Overview of the aaai competition," *AI magazine*, vol. 26, no. 2, p. 62, 2005.

[6] 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.

[7] H. Finnsson and Y. Björnsson, "Simulation-based approach to general game playing." in *AAAI*, vol. 8, 2008, pp. 259–264.

[8] D. Perez, J. Dieskau, M. Hünermund, S. Mostaghim, and S. Lucas, "Open Loop Search for General Video Game Playing," in *Proc. of the Conference on Genetic and Evolutionary Computation (GECCO)*, 2015, p. (to appear).

[9] B. Ross, "General Video Game Playing with Goal Orientation," Master's thesis, University of Strathclyde, September 2014.

[10] T. S. Nielsen, G. Barros, J. Togelius, and M. J. Nelson, "General Video Game Evaluation Using Relative Algorithm Performance Profiles," in *Proceedings of EvoApplications*, 2015.

[11] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.

[12] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[13] J. Reisinger, E. Bahçeci, I. Karpov, and R. Miikkulainen, "Coevolving strategies for general game playing," in *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*. IEEE, 2007, pp. 320–327.

[14] M. Hausknecht, P. Khandelwal, R. Miikkulainen, and P. Stone, "Hyperneat-ggp: A hyperneat-based atari general game player," in *Proceedings of the 14th annual conference on Genetic and evolutionary computation*. ACM, 2012, pp. 217–224.

[15] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone, "A neuroevolution approach to general atari game playing," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 6, no. 4, pp. 355–366, 2014.

[16] J. Gauci and K. O. Stanley, "Autonomous evolution of topographic regularities in artificial neural networks," *Neural computation*, vol. 22, no. 7, pp. 1860–1898, 2010.

[17] 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.

[18] 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.

[19] M. Hausknecht and P. Stone, "The impact of determinism on learning atari 2600 games," in *Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[20] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier networks," in *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics. JMLR W&CP Volume*, vol. 15, 2011, pp. 315–323.