Learning and evolving combat game controllers

Luis Peña, Sascha Ossowski *
* Universidad Rey Juan Carlos
Mostoles, Spain
{*luis.pena,sascha.ossowski*}*@urjc.es*

Jose M. Peña†
†Universidad Politécnica de Madrid
Madrid, Spain
jmpena@fi.upm.es

Simon M. Lucas‡
‡University of Essex
Essex, United Kingdom
sml@essex.ac.uk

***Abstract*—The design of the control mechanisms for the agents in modern video games is one of the main tasks involved in the game design process. Designing controllers grows in complexity as either the number of different game agents or the number of possible actions increase. An alternative mechanism to hard-coding agent controllers is the use of learning techniques. This paper introduces two new variants of a hybrid algorithm, named WEREWoLF and WERESARSA, that combine evolutionary techniques with reinforcement learning. Both new algorithms allow a group of different reinforcement learning controllers to be recombined in an iterative process that uses both evolution and learning. These new algorithms have been tested against different instances of predefined controllers on a one-on-one combat simulator, with underlying game mechanics similar to classic arcade games of this kind. The results have been compared with other reinforcement learning controllers, showing that WEREWoLF outperforms the other algorithms for a series of different learning conditions.**

I. Introduction

Although the graphical quality of a video game is one of its major selling points, the success of a game also depends heavily on other aspects such as the story script or the playing experience. A key factor of the playing experience is the behavior of the agents in the game.

Traditionally, agent controllers were programmed using ad hoc implementations, mainly based on finite state machines (FSM). More recently approaches such as Behavior Trees [1] are increasingly used because they are better equipped to deal with more complex behaviors, and are relatively easy for human designers to use. Nonetheless, this hard-coded development, when the number of agents and number of actions per agent are large, makes the design of these controllers time-consuming and error-prone. An alternative mechanism to avoid the hard-coding of agent controllers is the use of learning mechanisms to develop these controllers. These learning mechanisms can be applied during a development phase of the game, making agents learn the best sequence of actions under different circumstances. Reinforcement learning (RL) is one of the most interesting paradigms in this domain [2]. So far these learning methods have not been very widely used within video games, but this is at least partly due to the unpredictable speed and quality of the learning process. We believe that when learning can be done reliably and quickly it will naturally find wide application within video games.

In addition to being potentially useful for developing game agent controllers, the research described in this paper is also interesting from a machine learning perspective. In this view, we simply use the game environment as a test-bed to see which approach can best learn to beat opponents in a challenging fighting game.

This paper presents an alternative method to combine evolutionary algorithms and Q-Learning (a well-known RL algorithm) for stochastic games. The results, using a video game scenario, show that the combination of learning and evolving Q-learner matrices improves the learning rate, independently of the number of learning episodes.

The organization of the rest of the paper is as follows: Section II briefly reviews the related work on the reinforcement learning techniques. Section III presents the main contribution of this article, the combination of evolutionary algorithms and RL methods, applying two different evolutionary approaches. Section IV describes the experimental environment and the game rules used as framework. The results of these experiments are reported in section V. Finally, section VI summarizes the conclusions derived from this work.

II. Related Work

A. Reinforced Learning Techniques

Reinforcement Learning (RL) is a machine learning technique applied to model the behavior of an agent that has sensor mechanisms to perceive the state of an environment and obtains rewards from the actions it performs within this environment. RL is widely applied to solve a broad range of problems [3]. One of the typical problems that are solved by RL algorithms are Markov Decision Processes (MDPs). MDPs model environments where actions performed by the agent make the state of the environment transition to some other state with a certain probability, making the transitions non-deterministic. A similar case is Stochastic Games (SGs) in which multiple agents select actions and the next state and rewards depend on the joint action of all the agents. SGs are a natural model for many video games, including the fighting game studied in this paper.

1) SARSA and Q-Learner: The use of RL in fighting video games was introduced by [4], as a possible approach for the automatic creation of fighting strategies. This study showed that learning agents found interesting policies capturing the behaviour of the opponents, evaluating the strategy according to specific reward functions. This work also demonstrated that the application of well known algorithms (SARSA and Q-Learner) can be used as learning mechanisms suitable for this kind of scenarios. The SARSA algorithm

is an on-policy temporal difference learning algorithm for MDPs [5]. It is based on the estimation of the expected reward of a given state s for a given action a, denoted by $Q(s,a)$ (also know as Q-value). This estimation is continuously updated according to the following equation:

$$Q(s,a) \Leftarrow (1-\alpha)Q(s,a) + \alpha\left[r + \gamma Q(s',a')\right] \quad (1)$$

where $0 \leq \alpha \leq 1$ is a learning rate parameter (determining how fast the state-action pair is updated) and $0 \leq \gamma < 1$ is the discount factor for future rewards (indicating the influence of the reward from new state-action pair into the reward of the original state-action pair). The update equation states that for a given state-action pair $(s,a) \in S \times A$ the new state-action value is obtained by adding a small (depending on α) correction to the old value. The correction is the difference between the immediate reward r increased by the discounted future state-action value $\gamma Q(s',a')$ and the old state-action value $Q(s,a)$.

SARSA (s,a,r,s',a') is an on-policy learning algorithm in the sense that it estimates the value of the same policy that it is using for control. Q-Learning [6] constitutes an off-policy alternative to SARSA and replaces the term $\gamma Q(s'a')$ by $\gamma \arg\max_{a' \in A(s')} Q(s',a')$ in the above equation 1. This provides a separation of the policy being evaluated from the policy used for control. It leaves the update equation like this:

$$Q(s,a) \Leftarrow (1-\alpha)Q(s,a) + \alpha\left[r + \gamma arg\,\max_{a' \in A(s')} Q(s',a')\right] \quad (2)$$

Although SARSA and Q-Learning are suitable approaches to deal with MDPs, video games do not fully match MDP characteristics, as mentioned previously It is important, in many cases, to represent the complete set of transitions that may occur derived from the presence of two or more agents over the same environment. Stochastic games (SGs) are a natural multi-agent extension of MDPs, and have also been studied within RL [7], [8].

2) WoLF: Despite RL algorithms (such as Q-Learning) being appropriate to deal with MDPs, they are less appropriate, in theory, for SGs, due to the multi-agent aspects [9]. Thus, some variants of these algorithms have been successfully applied in these SG scenarios. "Policy Hill Climbing" (PHC) and "Win or Learn Fast" (WoLF) [10] are extensions to the Q-Learning algorithm particularly designed to deal with stochastic scenarios with multiple agents i.e. with SGs.

Both PHC and WoLF maintain a learning rate in the form of a selection probability for each state-action pair. The main difference is that, in PHC, this learning rate is constant while WoLF changes this learning rate depending on whether it is winning or losing. Intuitively, the algorithm tries to learn quickly when it is losing and more slowly when it is winning. To determine whether the algorithm is winning or losing, the current policy's payoff is compared with that of the average policy over time.

Algorithm 1: WoLF Algorithm

begin
  Let α, $\delta_l > \delta_w$ be learning rates, Initialize
  $Q(s,a) \leftarrow 0, \quad \pi(s,a) \leftarrow \frac{1}{|A|}, \quad C(s) \leftarrow 0$
  while *Finalization state not reached* **do**
    From state s select action a with probability $\pi(s,a)$
    Q-values are updated observing reward r and next state s',
    $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$
    Update estimate of average policy, $\bar{\pi}$,
    $C(s) \leftarrow C(s) + 1 \quad \forall\, a' \in A, \bar{\pi}(s,a') \leftarrow$
    $\bar{\pi}(s,a') + \frac{1}{C(s)}(\pi(s,a') - \bar{\pi}(s,a'))$
    Update $\pi(s,a)$ and constrain it to a legal probability distribution

$$\pi(s,a) + \begin{cases} \delta & if\, a = \\ & \quad argmax_{a'}(Q(s,a')) \\ \frac{-\delta}{|A|-1} & \text{otherwise} \end{cases}$$

    where,

$$\delta = \begin{cases} \delta_w & if \sum_a \pi(s,a)Q(s,a) \\ & \quad > \sum_a \bar{\pi}(s,a)Q(s,a) \\ \delta_l & \text{otherwise} \end{cases}$$

  end
end

In addition to Q-values, the algorithm also maintains the current mixed policy ($\pi(s,a)$). This policy controls the probability of selecting a given action during the learning phase. It is updated by increasing the probability of selecting the best performing action according to a learning rate δ_l, that is applied when the algorithm is losing, and δ_w, that is used when the algorithm is winning, with $\delta_l > \delta_w$. Algorithm 1 provides a detailed description of this policy.

B. Evolutionary Techniques

Evolutionary techniques have already been considered as a complementary mechanism for training learning algorithms. A representative approach was the NeuroEvolution of Augmenting Topologies (NEAT) [11], that have attracted great interest in the video game community. A practical example was the video game called Neuro-Evolving Robotic Operatives (NERO) that extends NEAT to work in real-time.

Indeed, the combination of evolutionary strategies and reinforcement learning has mainly been addressed towards the use of optimization algorithms to adjust connection weights in neural networks. There are only few references on the use of evolutionary techniques to complement reinforcement learning algorithms based on Q-Learning or similar approaches (usually named policy-space approaches). A first reference appeared in Moriarty et al.'s work [12]. In this work, a preliminary evolutionary algorithm for reinforcement learning (named EARL) is put forward. EARL evolves a chromosome with same size of the number of states, and the value of each of the genes would be the action to perform at this state. EARL focuses on deterministic policies.

In [13], evolved neural networks (using NEAT), were combined with Q-Learning algorithms to search function approximations. Using neural networks in this way is rather

different to the approach we adopt in this paper, though it would be interesting future work to compare the two approaches.

More recently, in [14], the authors introduce a reinforcement learning algorithm with a hierarchical evolutionary mechanism to evolve adaptive action value tables. The algorithm evolves several Q-Learning parameters, such as state discretization data, learning rate α, discount factor γ and searching rate (non deterministic action selection).

1) Estimation of Distribution Algorithms: Estimation of Distribution Algorithms (EDAs) were introduced in the 90s [15]. In general terms, EDAs are similar to Genetic Algorithms, but their main characteristic is the use of probabilistic models to extract information from the current population (instead of using crossover or mutation operators) in order to create a new and presumably better population.

As in the case of other evolutionary algorithms, EDAs create multiple solutions or individuals in a population. This population evolves from one generation to the next by estimating the probability distribution of a set of individuals (usually the best individual from the past generation), then sampling the induced model (without using crossover or mutation operators). Based on the probabilistic model considered, three main groups of EDAs can be distinguished: univariate models, which assume that variables are marginally independent; bivariate models, which accept dependencies between pairs of variables; and multivariate models, in which there is no limitation on the number of dependencies. The complexity of the different EDA approaches is usually related to the probabilistic model used, and the ability of that model to identify and represent the (in)dependencies among the variables. Detailed information about the main characteristics of EDAs, as well as the different algorithms that belong to this family, can be found in [16].

In this study, we focus on the Univariate Marginal Distribution Algorithm for Gaussian Models (UMDAg) [17]. This algorithm considers no dependencies between the variables involved in the problem. It is assumed that the joint density function follows an n-dimensional normal distribution, which is factorized by a product of one-dimensional and independent normal densities.

2) Differential Evolution Algorithms: Differential Evolution (DEs) algorithms were proposed by Rainer Storn and Kenneth Price in 1995 [18]. DEs are also a specific type of evolutionary algorithms that use an alternative recombination operator. Given a population in the generation i, each individual in this population $x_{i;j}$ is selected for recombination. The selected vector receives the name of objective vector. Three other vectors, x_{r_1}, x_{r_2}, and x_{r_3} are then randomly selected. They are all different to the objective vector and different to each other. These four vectors are then combined to obtain a new vector candidate to replace the objective vector:

$$v_{i+1;j} = x_{r_1} + F(x_{r_2} - x_{r_3}) \tag{3}$$

First, vectors x_{r_2} and x_{r_3} are subtracted and scaled according to a F factor. Finally, the result vector from the previous step and vector x_{r_1} are added. The final result vector of the mutation phases is known as the donor vector.

Once the donor vector is obtained, it is combined with the original vector $x_{i;j}$ by means of a crossover operation. A usual crossover method is the binomial crossover, which randomly selects, component by component, either from the original vector or from the donor vector, producing the new individual $u_{i+1;j}$. Finally, $x_{i;j}$ is replaced by $u_{i+1;j}$ if and only if the new individual has a better fitness value.

A more detailed survey of DE can be found at [19].

III. WEREWoLF ALGORITHM

WEREWoLF was first introduced in [20] as an evolutionary mechanism to combine multiple reinforcement learners (originally WoLF learners) using genetic algorithms. In this paper we extend it, applying two different evolutionary techniques. The current paper also includes an improved experimental process, comparing the learned controllers with various fixed controllers. Additionally, we also compare both WoLF and SARSA as the core learner models.

In WEREWoLF, the chromosome representing each individual is the encoding of the Q-values (and probabilities matrices, in the case of WoLF). A population of several learners is maintained at the same time. Each learner is trained for a fixed number of episodes at each generation. After a complete generation is finished, the different learners are combined. Each learner has a fitness value equal to the average of the aggregated reward obtained in each of the episodes (the sum of all the rewards of the actions selected in that episode). The combination of learners is performed by combining their matrices, which are then inserted into the target learner (after the necessary normalization of the probabilities, in the case of WoLF controllers). A brief explanations of the WEREWoLF is given by Algorithm 2.

For the present work, two evolutionary algorithms are considered as the combination mechanism of learners: Differential Evolution Algorithm (DE) and Estimation of Distribution Algorithms (EDA), in particular UMDA.

IV. SIMULATION FRAMEWORK

The framework used for the evaluation is based on a more complex environment called vBATTLE [21], which is a video game framework developed in Java. In vBATTLE the engine is designed as an independent light-weight event-driven simulator with a decoupled visualization tool. The simplified version used for this experiment is restricted to the one-on-one melee combat engagement.

The main elements of this simulator are the combatants. Each of the two combatants (intended as the bodies) are managed by a different controller (intended as minds), which must decide the action to make at each step. The combatants are engaged in a hand to hand combat which finishes after the death of one of the combatants or it is declared as draw if a certain amount of time passed without a result.

An example combatant is included in the Appendix, but these are the main components of a combatant:

Algorithm 2: WEREWoLF Algorithm

Initialization: **begin**
  Let P_0 be an initial population of WoLF instances of fixed size S (initialized as in the case of standard WoLF)
  $\forall_i w_i \in P_0$ initialize $res(w_i) \leftarrow 0$ and $cnt(w_i) \leftarrow 0$
  Let $g \leftarrow 0$ be the generation counter
  Let $c \leftarrow 0$ be the execution counter
end
//The evaluation step is repeated every time this algorithm is selected for evaluation
Evaluation Step (using the contrast controller w^c) **begin**
  $c \leftarrow c + 1$, execution counter
  Select w_i from the population P_g
  Execute a contest of w_i against the contrast controller w^c having rew as the final reward for w_i
  Update the result value for w_i, $res(w_i) \leftarrow res(w_i) + rew$ and $cnt(w_i) \leftarrow cnt(w_i) + 1$
  Update w_i during the learning process of the contest in the population P_g
  if $c \mod C_{GEN} = 0$, *the execution counter is multiple of the executions per generation* **then**
    $\forall_i w_i \in P_g$ compute individual fitness $fit(w_i) \leftarrow \frac{res(w_i)}{cnt(w_i)}$
    Apply genetic operators over P_g and produce the new offspring population of P_{g+1} and $g \leftarrow g + 1$
    $\forall_i w_i \in P_g$ initialize $res(w_i) \leftarrow 0$ and $cnt(w_i) \leftarrow 0$
  end
end

- A combatant has two different counters which represent his health (called *HP*) and his energy (called *EP*), the different actions that are carried out by each combatant can produce changes at the counters of both combatants.
- Each of the combatants has a set of actions that he can execute. The actions are classified as `REST`, `ATTACK` and `DEFENCE` actions. Each combatant has a different detailed interpretation of each named action in order to produce interesting asymmetric control strategies, which depend on the combatant being controlled and on the opponent combatant.
- Each action has two basic counters: Action Points (*AP*) which represent the amount of time that takes from the declaration of the action and the execution or finalization of the action, and the Exhaustion Points (*EP*) which is the amount of energy consumed (restored in the case of the REST action) to the action, applied when the action concludes.
- The ATTACK actions have the probability of hit, and the description of the basic damage that deals in the case of hitting. The damages are computed with three counters: Health Damage (*HD*), Exhaustion Damage (*ED*) and Stun Damage (*SD*). The *HD* and *ED* are subtracted from the counters of the target when the attack succeeds. The *SD* is transformed in certain amount of time which the target is inactive, loosing his declared action.
- The DEFENCE actions also have a probability of blocking. The defences are active since they are declared and protect the actor as long as they do not finish. If during their duration an attack is blocked the amount of *HD* and *ED* damage is reduced by a factor called Reduction (*Red*). There are some defences that can also block the *SD* damage that are marked as *UnStun*.

Algorithm 3: Simple Battle Manager Engine

begin
  Let $B, R \in \bar{C}$ two combatants with $A_B = \{A_{B_1}, \ldots, A_{B_n}\}, A_R = \{A_{R_1}, \ldots, A_{R_m}\} \in A$ the sets of actions for B and R.
  while *Finalization state not reached* **do**
    Let $seg_i = \{A_j, \ldots\}$ next segment of Actions and $ini \in \mathbb{N}$ the next instant counter
    RecoverEnergy(B, R, ini)
    RecoverStun(B, R, ini)
    for A_j **do**
      if A_j *is Attack* **then**
        Resolve$(A_j, S, T) \quad \rightarrow D_j \in \bar{D}$
      end
      EnergyModification(A_j, S)
    end
    ApplyDamages$(\bar{D})$
    if B *or* R *has no action declared* **then**
      DeclareAction$(c \in \bar{C})$
    end
  end
end
Where
RecoverEnergy(B, R, ini) adds an amount of energy to the combatant counter depending of the ini difference between segments
RecoverStun(B, R, ini) updates the counter of time remaining to finish the stun time for each combatant, if applicable
Resolve$(A_j, S, T) \quad \rightarrow D_j$
begin
  if A_j *hits* **then**
    HitGrade$(A_j) \rightarrow G_{off}$
    if T *has a defence declared* **then**
      DefenceGrade$(A_{T_i}) \rightarrow G_{def}$
    end
    DamageCalculus$(G_{off} - G_{def}, A_{T_i}) \rightarrow D_j$
  end
  else
    $0 \rightarrow D_j$
  end
  return D_j
end

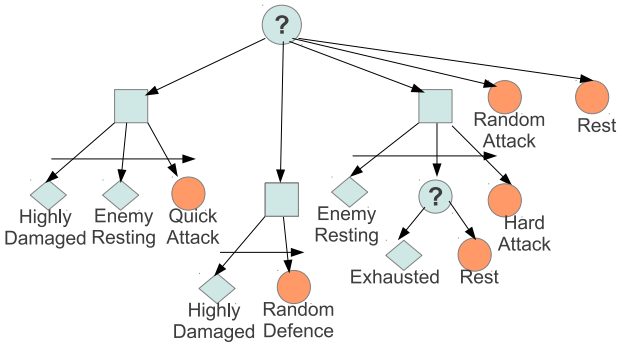
The general description of the game engine is explained at the Algorithm 3. The main features of this game are: (1) the actions are declared, in many cases, simultaneously by the two combatants, (2) the probability of success in the actions, and (3) the order of execution of the actions can be altered, because there are certain attacks (those which can inflict stun damage) that can delay the execution of the target's actions. These features make it an interesting game to study that captures many of the most important underlying features of fighting games.

V. Experimental Results

The experimental set up is based on the contest between two combatants, one with a fixed strategy and another using learning to adapt his own strategy over time. For this experiment the combatant characteristics are the same for both of the combatants to be fair and to make the result only dependent on the strategy and not on the combatant capabilities; the profile used is the one included at the Appendix.

Twelve (12) different controllers were built for the static strategies, with different rules or mechanisms. The controllers built are the following:

- A random controller, which randomly selects actions from the nine possible ones (E_RAND).
- A rule based engine, with mixed strategies which tries to exploit the time between the declaration and execution to select fast actions (E_SMART).
- Ten controllers based on Behaviour Trees [1], [22], shown in Figures 1, 2 and 3.



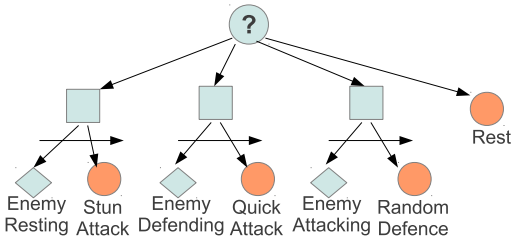


Figure 1. Behaviour Trees. Top: E_BT_OFF Down: E_BT_DEF

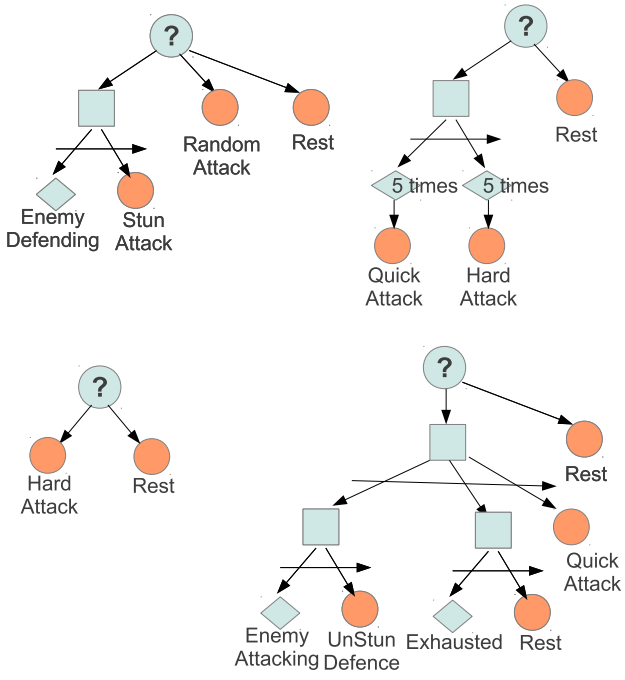


Figure 2. Behaviour Trees Top to Down, Left to Right: E_BT_ALL, E_BT_COMBO, E_BT_HARD and E_BT_COWARD

For the learning controllers, six different alternatives were proposed:

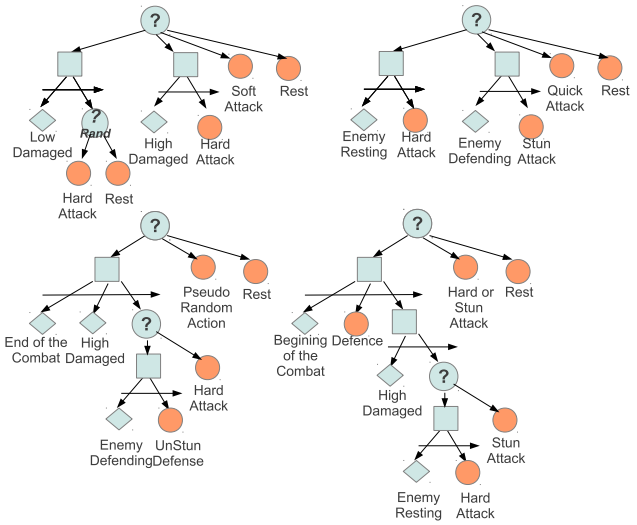


Figure 3. Behaviour Trees Top to Down, Left to Right: E_BT_ENERGY, E_BT_STUN, E_BT_FINAL and E_BT_TIMER

- A SARSA controller with parameters $\epsilon = 0.1, \gamma = 1.0$.
- A WoLF controller with parameters $\epsilon = 0.1, \alpha = 0.4, \gamma = 0.4, \sigma_l = 0.6$ and $\sigma_w = 0.2$.
- A WEREWoLF controller with DE evolutionary strategy and the same parameters of the aforementioned WoLF.
- A WEREWoLF controller with EDA evolutionary strategy and the same parameters of the aforementioned WoLF.
- A WERESARSA controller with DE evolutionary strategy and the same parameters of the aforementioned SARSA.
- A WERESARSA controller with EDA evolutionary strategy and the same parameters of the aforementioned SARSA.

All the "WERE controllers" (WEREWoLF and WERESARSA) use a fixed size population of 10 individuals which are combined to produce the new populations using the two different schema (DE and EDA). Fitness evaluation is the average reward along 20 episodes. DE-specific parameters are: DE/rand/1/bin model, CR=0.5, F=0.5. EDA-specific parameters are: UMDAg model with elitism.

For all of the controllers the environment is represented by the following state variables:

- The enemy action declared (a value in the interval $[0, 8]$).
- The segment of action when the enemy action will be executed. These values are discretised into a set of bins depending on the APs required by the different actions the learning controller can do. Thus, each of the bins represents one or more possible actions a controller can perform before the opponent's action is completed. The values of this variable depend on the combatant profile, using the one depicted at the Appendix the range is $[0, 6] = \{[0-6APs], [7-10APs], [11-12APs], [13-15APs], 16APs, [17-20APs], 21+APs\}$

- The percentage of remaining HPs of the learning combatant discretised into four quartiles. The values are in $[0, 3]$
- The relative difference of HPs between the two combatants. If the static combatant has less HPs than the learning combatant the variable is set to 0, otherwise it is 1.

The possible states result from the cross product of these variables (504 possible states) plus four additional states:

- One final state where the combatant controlled by the learner is dead.
- One final state where the combatant controlled by the static strategy is dead.
- One state for the case where learning combatant is so exhausted that he can not make any other action.
- One state for the case where combatant with the static strategy is so exhausted that can not make any other action.

The complete representation of the environment produce a set of 508 different states. This state space representation is used by all of the controllers. And the action space is fixed by the combatant profile having 9 different actions.

Each step of the simulation obtains a reward derived by the current environment state. The reward function is the same for all of the experiments:

$$r = \alpha \Delta\Delta HP\% + (1-\alpha)\Delta\Delta EP\% + W\dot{(}1-\tau) \qquad (4)$$

Where,

$$\Delta\Delta HP\% = \Delta MyHP\% - \Delta EnemyHP\%,$$
$$\Delta MyHP\% = MyHP\%_t - MyHP\%_{t-1}, \quad \text{Same for EP\%}$$
$$\tau = \frac{APs_t}{APs_{max}}$$

α is set to 0.95 and W is 200 if $EnemyHP < 0$, or -200 if $MyHP < 0$, or 0 otherwise.

Each experiment is repeated 25 times (for each combination of one static and one learning controller). Each evaluation is performed as follows: first, a learning phase (out of 10000, 25000 and 50000 episodes), and finally, a evaluation phase (of 1000 episodes). Each episode is carried out until one of the combatants has lost (reaching 0 HPs) or a fixed time of 10000 APs is reached (resulting as a draw combat). The final comparison among all the learning controllers are based on the percentage of victories in the evaluation phase against each of the static controllers.

A. Comparative results

In order to provide a proper statistical validation of the results, the distribution of all of the results was first compared with the Friedman test to detect significant differences among the algorithms. For the case of the 10000 episodes scenario, a value of 28.95 was obtained for the chi-squared statistic, which corresponds with a $p-value$ of $2.37E-05$, a value of 33.57 ($p-value$ of $2.90E-06$) and a value of 34.14 ($p-value$ of $2.23E-06$), for 25000 and 50000 episodes respectively. In the three scenarios, these results confirm the existence of significant differences between the algorithms. According to this test, the algorithms were ranked as shown in Table I, where, WEREWoLF-DE algorithm obtained the best results for the three scenarios. In addition, the table also shows the average win ratio for each of the algorithms. Then two post-hoc methods (Holm and Hochberg) were used to obtain the adjusted p-values for each comparison between the control algorithm (WEREWoLF-DE) and the remaining algorithms. The *Wilcoxon signed rank* test was also used for comparing the results, adjusting the obtained $p-values$ to take into account the Family-Wise Error Rate (FWER) when conducting multiple comparisons. The results of these tests are reported in Table III and show, for all of them, that there is statistical evidence to state that WEREWoLF-DE is significantly better than the remaining algorithms (for all the cases according to Wilcox test and in a vast majority of the cases according to Holm and Hochberg procedures).

Table I
AVERAGE RANKING AND AVERAGE WIN RATIO AGAINST ALL CONTROLLERS (10000, 25000, 50000 EPISODES)

	Ranking			Avg Win Ratio		
	10k	25k	50k	10k	25k	50k
WEREWoLF-DE	1.42	1.33	1.58	0.35	0.41	0.44
WERESARSA-DE	2.75	2.66	2.50	0.26	0.26	0.27
SARSA	3.42	3.08	2.75	0.23	0.24	0.24
WEREWoLF-EDA	3.83	4.17	4.33	0.15	0.16	0.12
WERESARSA-EDA	4.58	4.92	4.83	0.19	0.17	0.16
WoLF	5.00	4.83	5.00	0.19	0.22	0.21

To summarize the general performance of the controllers we can look at the Figure 4, which represents the average win ratio against each of the controllers (in the particular case of 50000 episodes).

In a detailed study of the performance of the algorithms, a particular case behaves differently from the rest of the experiments: The most defensive controller (E_BT_DEF) prefers a draw against a lose, tending to consume the time, in this particular case the SARSA-based controllers reach more victories, but with an average reward lower than the reward obtained by the WoLF-based controllers (see Table II). These results show that the WoLF algorithm, in those problems where the reward function punishes the wrong steps, tends to perform a more cautious exploration of the environment, becoming a more conservative controller than SARSA algorithm. This particular behavior is positive in this case, according to the selected evaluation criteria.

Table II
AVERAGE REWARD WITH 50000 EPISODES AGAINST E_BT_DEF

	Avg Rw	StdDev Rw	Win Ratio
WoLF	-2.8	22.99	5E-05
WEREWoLF-DE	-4.6	29.27	3E-04
SARSA	-162.6	58.9	0.02
WEREWoLF-EDA	-87.3	100.19	0.02
WERESARSA-EDA	-59.76	153.89	0.26
WERESARSA-DE	-12.19	151.46	0.34

Another interesting study is the evolution of the fitness value. Individuals obtain this fitness value as the average

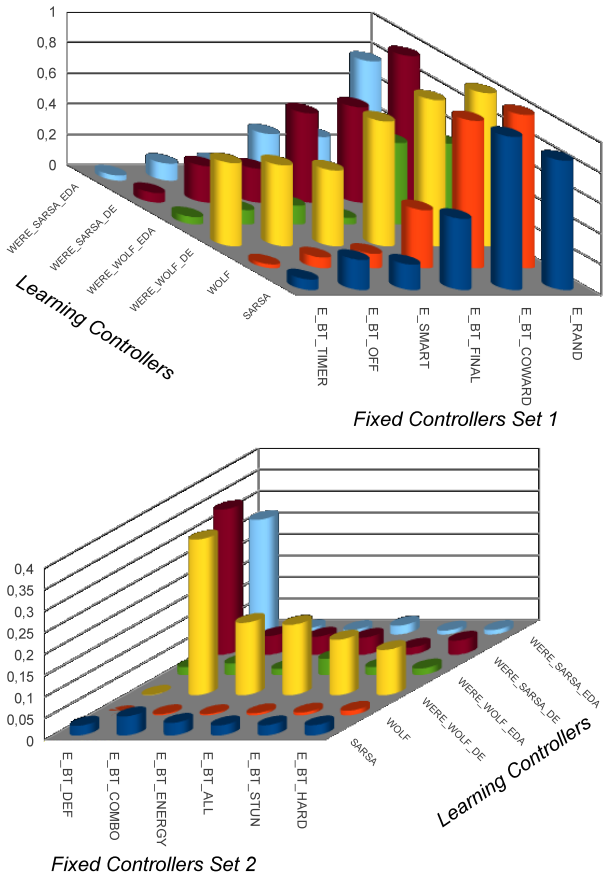


Figure 4. Average Win Ratio with 50000 episodes

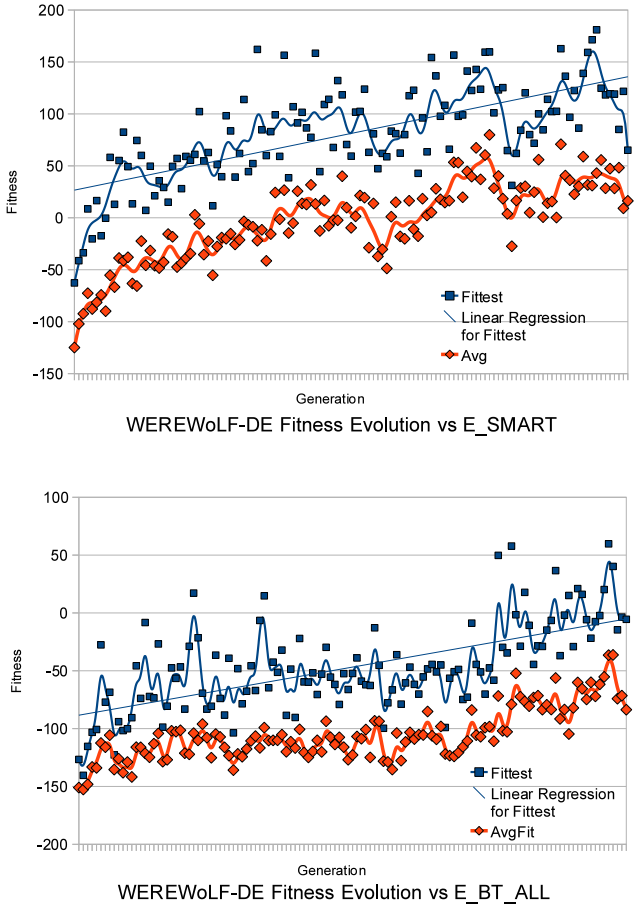


Figure 5. WEREWoLF-DE Fitness evolution vs E_BT_ALL and E_SMART (50000 episodes)

of their performance on each of the episodes they consume during a given generation. This performance is the sum of all the rewards granted by all the actions in that episode. Although the evolution is not continuously incremental due to the randomness of the problem (actions have only a random chance of success), WEREWoLF-DE shows an incremental trend along the different generations (for instance, the results shown by the Figure 5).

Finally, it is clear also that DE search exploits much better the mechanism for combining multiple learners in this evolutionary framework. The results are consistent for both WoLF and SARSA controller cores and along multiple length of the experiments (10000 episodes to 50000 episodes). The poor performance of EDA variants can be explained by the reduced population (compared with the size of the chromosome), this was also a reason why other state-of-the-art optimization techniques, such as CMAES, were not considered. DE proves to be a powerful optimization alternative in high-dimensionality problems.

VI. Conclusions

We implemented and tested two hybrid learning algorithms that combine evolutionary algorithms with reinforcement learning. We compared the controllers learned using these algorithms with several other controllers, and analysed the results using standard statistical tests. The results show the improved performance of reinforcement learning algorithms when combined with evolutionary techniques. This experimentation has been carried out in environments with large state spaces and with a set of complex actions in which Were-Hybrid algorithms outperforms the exploration of the standard reinforcement learning mechanisms.

Despite the improvement of the learning algorithm, this work has highlighted certain interesting facts. The representation of the state space and the construction of the reward function is, like in any RL problem, complex and critical to the success of the experiment. The results show that the application RL to this kind of video game scenario is highly dependent on the environment variables used for the description of the state and the reward function, that must provide enough information to guide the learning process. In future work, we want to apply Function Approximators to try to represent with more fidelity the environment and extract all the possible knowledge from it in order to steer the learning more effectively.

Acknowledgments

This work was supported in part by the Spanish Ministry of Science and Innovation through the projects *AT*

Table III
STATISTICAL VALIDATION 10000, 25000 AND 50000 EPISODES (WEREWOLF-DE IS THE CONTROL ALGORITHM)

WEREWoLF-DE vs.	p-values of the different tests Holm	Hochberg	Wilcox
10000 episodes			
WERESARSA-DE	$8.09E-02$	$8.09E-02$	$1.71E-02^{\surd}$
WERESARSA-EDA	$1.35E-04^{\surd}$	$1.35E-04^{\surd}$	$6.10E-03^{\surd}$
SARSA	$1.77E-02^{\surd}$	$1.77E-02^{\surd}$	$4.88E-04^{\surd}$
WEREWoLF-EDA	$4.67E-03^{\surd}$	$4.67E-03^{\surd}$	$4.88E-04^{\surd}$
WoLF	$1.35E-05^{\surd}$	$1.35E-05^{\surd}$	$7.32E-04^{\surd}$
Wilcox p-value with FWER: WEREWoLF-DE vs. All			$2.48E-02^{\surd}$
25000 episodes			
WERESARSA-DE	$8.09E-02$	$8.09E-02$	$1.05E-02^{\surd}$
WERESARSA-EDA	$1.35E-05^{\surd}$	$1.35E-05^{\surd}$	$2.44E-03^{\surd}$
SARSA	$4.39E-02^{\surd}$	$4.39E-02^{\surd}$	$4.88E-04^{\surd}$
WEREWoLF-EDA	$6.23E-04^{\surd}$	$6.23E-04^{\surd}$	$4.88E-04^{\surd}$
WoLF	$1.84E-05^{\surd}$	$1.84E-05^{\surd}$	$2.44E-04^{\surd}$
Wilcox p-value with FWER: WEREWoLF-DE vs. All			$1.41E-02^{\surd}$
50000 episodes			
WERESARSA-DE	$2.53E-01$	$2.30E-01$	$1.34E-02^{\surd}$
SARSA	$2.53E-01$	$2.30E-01$	$1.22E-03^{\surd}$
WoLF	$3.85E-05^{\surd}$	$3.85E-05^{\surd}$	$1.71E-03^{\surd}$
WERESARSA-EDA	$8.35E-05^{\surd}$	$8.35E-05^{\surd}$	$3.42E-03^{\surd}$
WEREWoLF-EDA	$9.52E-04^{\surd}$	$9.52E-04^{\surd}$	$4.88E-04^{\surd}$
Wilcox p-value with FWER: WEREWoLF-DE vs. All			$2.02E-02^{\surd}$

$\surd$ means that there are statistical differences with significance level $\alpha = 0.05$

(Grant CONSOLIDER CSD2007-0022, INGENIO 2010) and *OVAMAH* (Grant TIN2009-13839-C03-02).

REFERENCES

[1] A. Champandard, "Behavior trees for next-gen game AI," in *Game Developers Conference*, 2007.

[2] M. Pfeiffer, "Machine learning applications in computer games," Master's thesis, Institute for Theoretical Computer Science, Graz University of Technology, 2003.

[3] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research 4*, pp. 235–285, 1996.

[4] T. Graepel, R. Herbrich, and J. Gold, "Learning to fight," in *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*, 2004.

[5] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," vol. 9, no. 5, 1998.

[6] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 272–292, 1992.

[7] C. Claus and C. Boutilier, "The dynamics of reinforcement learning in cooperative multiagent systems," in *AAAI '98/IAAI '98: Proc. 15th national/10th Artificial Intelligence/Innovative Applications of Artificial Intelligence*. Menlo Park, CA, USA: American Association for Artificial Intelligence, 1998, pp. 746–752.

[8] J. Hu and M. P. Wellman, "Multiagent reinforcement learning: Theoretical framework and an algorithm," in *In Proc. of the 15th International Conference on Machine Learning*, 1998, pp. 242–250.

[9] M. Bowling and M. Veloso, "An analysis of stochastic game theory for multiagent reinforcement learning," School of Computer Science. Carnegie Mellon University. Course Report, October 2000.

[10] M. Bowling and M. Veloso, "Multiagent learning using a variable learning rate," *Artificial Intelligence*, vol. 136, pp. 215–250, 2002.

[11] K. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002.

[12] D. Moriarty, A. Schultz, and J. Grefenstette, "Evolutionary algorithms for reinforcement learning," *Journal of Artificial Intelligence Research*, vol. 11, pp. 241–276, 1999.

[13] S. Whiteson and P. Stone, "Evolutionary function approximation for reinforcement learning," *Journal of Machine Learning Research*, vol. 7, pp. 877–917, 2006.

[14] M. Yoshikawa, T. Kihira, and H. Terai, "Q-learning based on hierarchical evolutionary mechanism," *WSEAS Transactions on Systems and Control*, vol. 3, no. 3, pp. 219–228, 2008.

[15] H. Mühlenbein and G. Paaß, "From recombination of genes to the estimation of distributions i. binary parameters," *PPNS IV*, vol. 1141, pp. 178–187, 1996.

[16] P. Larrañaga and J. A. Lozano, Eds., *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*. Boston, MA: Kluwer, 2002.

[17] P. Larrañaga, R. Etxeberria, J. A. Lozano, and J. M. Peña, "Optimization in continuous domains by learning and simulation of gaussian networks," in *Proceedings of the 2000 Genetic and Evolutionary Computation Conference*, A. S. Wu, Ed., Las Vegas, Nevada, USA, 2000, pp. 201–204.

[18] R. Storn and K. Price, "Differential evolution - a simple and efficient adaptive scheme for global optimization over continuous spaces," 1995.

[19] S. Das and P. N. Suganthan, "Differential evolution: A survey of the state-of-the-art," *IEEE Trans. Evolutionary Computation*, vol. 15, no. 1, pp. 4–31, 2011.

[20] L. Peña, J.-M. Peña, S. Ossowski, and P. Herrero, "Evolving Q-learners for stochastic games: Study on video game agent controllers," in *World Automation Congress (WAC)*, 2010.

[21] L. Peña, S. Ossowski, and J.-M. Peña, "vBattle: A new framework to simulate medium-scale battles in individual-per-individual basis," in *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, Sept. 2009, pp. 61–68.

[22] R. J. P. Duran, *Java Behaviour Trees, User Guide*, http://sourceforge.net/projects/jbt/, September 2010.

APPENDIX

Table IV
COMBATANT DESCRIPTION EXAMPLE

Brutus			**HPs**	250	**EP**	150
Rest			**AP**		**EP**	
Rest10			10		-25	
Attack	**AP**	**EP**	**HD**	**ED**	**SD**	**% Hit**
Quick	6	13	11	5	0	0.65
Stun	12	19	11	10	20	0.55
Hard	16	25	33	15	10	0.40
Accurate	12	16	10	5	0	0.85
Dummy	20	25	5	5	5	0.45
Defense	**AP**	**EP**	**Red**	**UnStun**	**% Block**	
Reduction	10	5	0.25	false	0.30	
UnStun	15	8	0.65	true	0.60	
Sure	20	4	0.75	false	0.85	