Online Adaptation of Computer Games Agents: A Reinforcement Learning Approach

GUSTAVO DANZI DE ANDRADE
HUGO PIMENTEL SANTANA
ANDRÉ WILSON BROTTO FURTADO
ANDRÉ ROBERTO GOUVEIA DO AMARAL LEITÃO
GEBER LISBOA RAMALHO

Universidade Federal de Pernambuco (UFPE) – Centro de Informática (CIn)
Av. Prof. Luís Freire, s/n, Cidade Universitária, CEP 50740-540 – Recife/PE/Brazil
{gda,hps,awbf,argal,glr}@cin.ufpe.br

Abstract

Designing the behavior of non-player characters that challenges the human player adequately is both a key feature and a big concern in computer games development. This work presents a reinforcement learning (RL) based technique to build intelligent agents that automatically control the game difficulty level, adapting it to the human player's skills in order to improve the gameplay. The technique is applied to a fighting game, Knock'em, to provide empirical validation of the approach.

Key-words:

Reinforcement learning, user adaptation, intelligent agents, fighting computer games.

1 Introduction

The quality of a computer game is influenced by different concepts, such as its graphical interface, background history, input interface, and, particularly, non-player characters (NPCs) artificial intelligence [17]. In particular, an entertaining opponent should neither be invincible nor easily defeatable. It should behave roughly at the same human player level, challenging him and increasing the gameplay.

A traditional way to develop artificially intelligent agents is to use pre-programmed scripts, in which fixed rules are defined during the development of the game, representing agent behavior. A typical rule in a fighting game would state "*punch opponent if he is reachable, chase him, otherwise*". As game complexity increases, this technique results in a lot of rules, which are error-prone. Moreover, the resulting agent does not adapt to user skills, acting similarly against beginning or experienced players, as well as repeating the same old tactics even after a long term experience. This way, human players can easily defeat computer opponents by always exploiting faults in such foreseeable agent behavior. Although it is still possible for a game to offer static difficulty levels, players must choose it at its beginning, remaining tied to this level until the end. Clearly, this traditional approach harms the gameplay.

Designing non-player characters behavior involves two distinct, yet inter-related, problems: building the agent initial intelligence, and providing mechanisms for online adaptation to the human player behavior. While scripts have shown to be impracticable to deal with complex knowledge as well as dynamic adaptation, machine learning is a natural way to address these problems. In fact, this is a traditional approach to design learning and adaptive systems [18].

Some commercial games already use machine learning [1]. The techniques range from fuzzy logic systems [2] to multilayer perceptron neural networks [3]. However, although reinforcement learning is quite popular in academic AI, it's still uncommon at game AI community.

This paper presents a novel approach to the construction, evolution and adaptation of intelligent agents in computer games. We combine reinforcement learning (RL) and challenge functions in an original way to explore RL properties that are overseen by conventional RL applications. Challenge functions map a game state into a value which specifies how easy the game is perceived by human users. Based in this value, the game difficulty level can be increased or decreased, by choosing more or less adequate actions, respectively. This approach provides agents that are able to challenge very experienced human players, but still adapt its behavior to novice users. In order to evaluate the approach, we applied it to Knock'em [14], a real time fighting game we have developed with a simple Artificial Intelligence (AI).

Next Section revises previous user adaptation work into computer games. Section 3 summarizes reinforcement learning concepts and its applications in games. Section 4 describes our original approach to address the problem. Section 5 presents Knock'em and how the previous section concepts were applied in the game. Section 6 shows the experimental results, and Section 7 concludes and provides future directions to the work.

2 User Adaptation in Computer Games

The task of building adaptive agents in computers games is addressed in some recent works.

Some techniques have been proposed to model human opponent behavior. Opponent modeling is useful to discover how to defeat him. Developers have been using genetic classifier systems [11], decision trees [12], and dynamic scripting [13]. These works apply machine learning techniques aiming to create players that can beat all possible opponents.

Other works are more concerned with developing mechanisms to dynamically adapt game level to user skills. Hunicke and Chapman [19] control the game environment in order to increase or decrease the difficulty. For example, if the game turns to be too much difficult, the player gets more weapons, recover life points faster or face fewer opponents. As this approach does not change opponent behavior, it turns to become quite artificial and predictable to human players.

Demasi and Cruz [10] explore the concept of challenge function with genetic algorithms to achieve user adaptation. They build agents intelligence through genetic algorithms, keeping alive agents that best fit game difficulty.

This is an innovative, indeed. However, it suffers from some problems. Aiming to speed up the learning process, this approach uses some pre-defined models (agents with good genetic features) to guide the evolution. As so, agent learning is bounded by the best pre-defined model, beyond which learning becomes quite uncontrollable, harming the technique application for very skilled users or users with uncommon behavior. Furthermore, this approach does not keep agent history, but only the current best fit to human player. If the human change from a newbie player to an experienced one, the agent will have to gradually evolve again toward a good generation, requiring human users to play a lot of games against easy agents.

Our approach to address user adaptation of computer game agents uses reinforcement learning as machine learning technique, and explores some of its properties in an innovative way.

3 Reinforcement Learning in Games

Reinforcement learning (RL) creates agent's intelligence based only in its interaction with the environment. In contrast to supervised learning, it does not use examples of desired behavior, but only a reward signal that indicates how good (or bad) an action was in a given context.

3.1 The RL Framework

RL is often characterized as a problem of "learning what to do (how to map situations into actions) so as to maximize a numerical reward signal" [16]. Formally, in the reinforcement learning framework, we have an agent that sequentially makes decisions in an environment. At each step, the agent percepts the current state s from a finite set S, and chooses an action a from a finite set A, leading to a new state s'. The information encoded in s should summarize all present and past relevant sensations. Each state-action pair (s,a) has a reward signal $R(s,a)$ feedback to the agent when action a is executed at state s. Implicitly, this reward signal must determine the agent objective, as it is the only feedback to guide the desired behavior.

The main goal is to maximize a long-term performance criterion, called return, which represents the expected value of future rewards. The agent then tries to learn an *optimal policy* π^* which maximizes the expected return. A policy is a function $\pi(s)\rightarrow a$ that maps state perceptions into actions. We can define the *action-value function*, $Q^{\pi}(s,a)$, as the expected return when starting from state s, performing action a, and then following π thereafter. If the agent can learn the optimal action-value function $Q^*(s,a)$, an optimal policy can be constructed greedily: for each state s, the best action a is the one that maximizes Q.

As previously stated, reinforcement learning is a learning problem. One traditional algorithm for solving it is Q-Learning [16]. It consists in iteratively computing the Q values for state-action pairs, using the following update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma.V(s') - Q(s,a)]$$

in which V(s') = $\max_a$ Q(s',a), α is the learning rate and γ is a discount factor that gives more importance to near rewards, differing it from results of far executed actions.

The Q-Learning algorithm can be easily implemented through dynamic programming, using a bidimensional matrix, called Q-Table, representing the Q function. Table values are updated accordingly to the previous rule.

It is guaranteed that this algorithm converges to the optimal *Q* function in the limit, under the standard stochastic approximation conditions. It is worth noticing that no prior knowledge about the process dynamics is necessary.

The feature of not using specific domain knowledge, combined with the fact that a teacher is not necessary, make reinforcement learning naturally applicable at complex and diverse domains, such as computer games.

3.2 Previous Work

A traditional successful reinforcement learning application is Tesauro Backgammon player [4], which reached first class players level using little backgammon specific knowledge. Other successful RL players are Samuel checkers [5] and a Go player that performs better than traditional computer Go players [6]. However, these RL players act in turn-based games, in which the environment do not change while the agent is choosing his action. In real time games, the time processing requirements are a new problem to be addressed.

A particular domain commonly used to test new artificial intelligence techniques is Robocup [7], in which reinforcement learning was combined with methods to increase learning speed [8] and reduce problem complexity [9]. These techniques are also easily applicable into games domain.

4 The proposed approach

Our approach to develop intelligent adaptive agents combines Q-Learning with a challenge function, as proposed by Demasi and Cruz [10], and explores some properties of the learned policy. Given an state *s*, Q-Learning estimates *Q(s,a)*, the quality of executing action *a* at state *s*. Standard Q-Learning applications use the best Q value to determine the action to be executed. In the computer games domain, it means keeping the agent acting as eficient as possible. As this is not our objective, we allow the agent to choose any possible action, accordingly to the challenge function.

In principle, as any RL-based agent, the agent chooses the best actions for each situation and keeps learning the player behavior in order to improve its performance. However, according to the value of the challenge function, i.e. the difficult the player is facing, the agent can choose better or worse actions. For a given situation, if the game level is too hard, the agent does not choose the best action in the Q-Table. Instead, it chooses the second best one, the third, and so on, until its performance is as good as the player's. Similarly, if the game level becomes too easy, it starts to choose actions one level above. Figure 1 shows a possible configuration for an agent acting in its second best level.

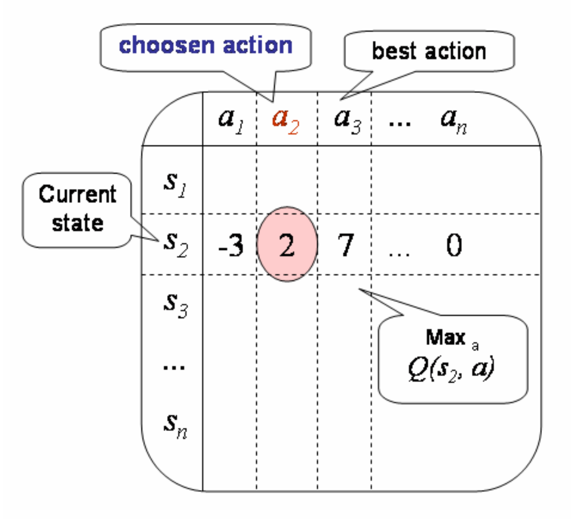

Figure 1: An agent acting at the second level.

This approach means to use the order relation naturally defined in a given state by each action's Q-value, which is automatically built during the learning process. As these values estimate the individual quality of each possible action, it turns out to be possible to control the agent's total quality, i.e. its game playing level.

It is important to notice that this technique changes only the action choice procedure, while the learning process, which means the updates at the Q values, is the same as standard Q-Learning applications.

Our approach apparently has a drawback. Since machine learning techniques require thousands of training iterations to achieve good performance, it could not be possible to learn a competitive behavior in real time. To deal with this, we use an offline learning phase, where a general initial behavior is learned by the agent. Moreover, to keep the learning speed at online phase as fast as possible, we use strategies to reduce the problem complexity.

The problem complexity is directly related to states and actions space size. Reducing states space size can be done by discretizing continuous variables and coding abstract characteristics. The first strategy means not only to transform real values at the nearest integer, but to code values that are representative to agent perception. In a first person shooter game, for example, the opponent distance can be coded simply as inside or outside the gun reach area (supposing the shot damage is not influenced by the distance), so the state space size is reduced preserving the agent's perception quality. The second strategy is to code environment abstract features. For a soccer player agent learning to dribble opponent, it would not mean to code players directions (right and left), but their relative directions (matching and opposite), so the agent needs only to learn to move at opponent opposite direction (learns one state-action pair), and not specifically going left when opponent goes right, and vice-versa (two state-action pairs).

Reducing actions space size can be done by coding full moves [9]. Moves are sequences of atomic actions with a common objective. For a soccer player, the action *retrieve the ball* would be the composition of the following atomic actions: change agent direction, run to the opponent, and catch the ball.

A special design feature of a reinforcement learning agent is the quality of reward signals. As this is the way to guide agent objectives, a natural design decision for computer games is to give positive rewards when the agent wins the game and negative ones otherwise. Although this approach correctly represents agent objectives, it excessively delays the learning process, demanding several iterations until the impact of first actions at game final result are learned by the agent. An alternative approach would give rewards as soon as possible, based in performance measurements for a running game (won and lost pieces, life difference or shooting accuracy, for example).

5 Case Study

As a case study, the concepts stated at previous chapters were implemented in Knock'em [14], a real time fighting game where two fighters are faced into an enclosure for bullfighting. This class of games is represented by successful commercial series, like Capcom Street Fighter and Midway Mortal Kombat [15]. Figure 2 shows a screenshot of the game.

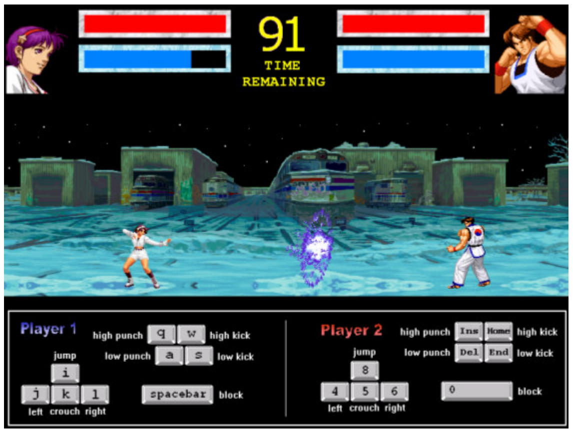

Figure 2: Knock'em screenshot.

The main objective of the game is to beat the opponent. A fight ends when the life points of some player (initially, 100 points) are turn to zero, or after one minute of fighting. The winner is the fighter which has the higher remaining life. The environment is a bidimensional arena in which horizontal moves are free and vertical moves are possible through jumps. The possible attack actions are to punch (strong or fast), to kick (strong or fast), and to launch fireballs. Punches and kicks can also be deferred in the air, during a jump. The defensive actions are blocking or crouching. While crouching, it is also possible for a fighter to punch and kick the opponent. The fighter "mana", which is reduced after a magic attack, is continuously refilled during time at a fixed rate.

The fighters artificial intelligence is implemented as a reinforcement learning task. As so, it is necessary to code the agents perceptions, possible actions and reward signal. The state representation (agent perceptions) is represented by the following tuple:

$$S = (S_{agent}, S_{opponent}, D, M_{agent}, M_{opponent}, F)$$

S$_{\textbf{agent}}$ stands for the agent state (stopped, jumping, or crouching). **S**$_{\textbf{opponent}}$ stands for opponent state (stopped, jumping, crouching, attacking, jumping attack, crouching attack, and blocking). **D** represents opponent distance (near, medium distance and far away). **M** stands for agent or opponent mana (sufficient or insufficient to launch one fireball). Finally, **F** stands for enemies' fireballs configuration (near, medium distance, far away, and non existence).

These attributes were chosen because of their impact in fighter performance. The agent possible states represent the ones in which the agent can effectively make decisions (i.e. change its state). The opponent state is important to perceive his attacks (which the agent must defend) and for detecting situations where he is vulnerable. Opponent distance is relevant to percept the difference between punches executed far away from those when the opponent is in a reachable distance. Mana is important to know if the agent (or the opponent) can launch fireballs anytime or should wait for mana refilling. Fireballs configuration aims to inform how the agent must act (defend or deviate) regarding the magic attacks.

The agents' possible actions are the ones possible to all fighters: punching and kicking (strong or fast), coming close, running away, jumping, jumping to close, jumping to escape, launching fireball, blocking, crouching and keep stopped.

The reinforcement signal is based in the difference of life caused by the action (life taken out from opponent minus life lost by the agent). As a result, the agent reward is always in the range [-100, 100]. Negative rewards mean bad performance, because the agent lost more life than was taken from the opponent, while positive rewards are the desired agent objective. This measure is representative of the agent objective because a fight winner is determined by its ending life points.

Finally, the challenge function used is based in the reinforcement signal. As positive rewards indicate the agent is winning and negative ones indicate that it is losing, we expect that rewards near zero indicate that the two fighters are acting in the same level. Therefore, we empirically stated the following challenge function:

$$f(agent) = \begin{cases} easy, if \quad r(agent) < -10 \\ difficult, if \quad r(agent) > 10 \\ medium, \ otherwise \end{cases}$$

6 Experimental Results

To evaluate the effectiveness of our approach, we implemented the developed concepts in Knock'em.

In all experiments some parameters were fixed. The learning rate was fixed in 50% and the reward discount rate in 90%. Although the game has different fighters with different attributes (skills and limitations), the experiments were fixed to only one of them.

Before being evaluated, the reinforcement learning agents were trained against a random fighter during 500 fights. We compared the performance of three distinct agents: a traditional state-machine (script-based agent), a traditional reinforcement learning (playing as best as possible), and the adaptive agent (implementing the proposed approach).

The evaluation scenario consists of a series of fights against different opponents, simulating the diversity of human players strategies: a state-machine (static behavior), a random (unforeseeable behavior) and a traditional RL agent (intelligent and with learning skill). Each agent being evaluated plays 30 fights against each opponent. The performance measurement is based in the final life difference in each fight. Positive values represent that the evaluated agent wins, and negative ones that the agent loses. These values are graphically displayed beyond.

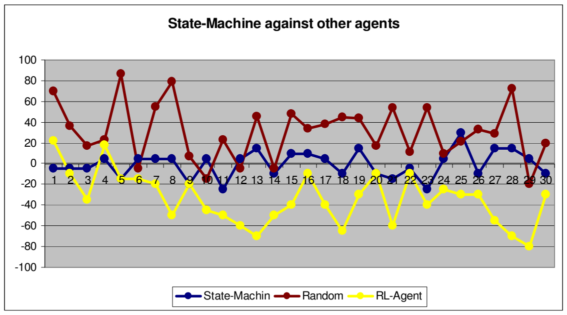

Figure 3: State-machine agent's performance

Figure 3 shows the state-machine (SM) agent performance against each of the others agents. The positive values of the red points show that the agent beats almost always a random opponent. The blue points show that two state-machine fighters have a similar performance while fighting against each other. The negative yellow points show that the RL agent almost always beats the state-machine, and the life difference increases as it learns to deal with the static state-machine behavior.

Figure 4 shows the traditional RL agent performance. Analyzing as above, we can conclude that the RL agent beats quite easy the state-machine and the random players. However, as random players do not have a foreseeable behavior, the RL agent fights better against state-machine opponents, learning a policy that maximizes the result against the SM strategy.

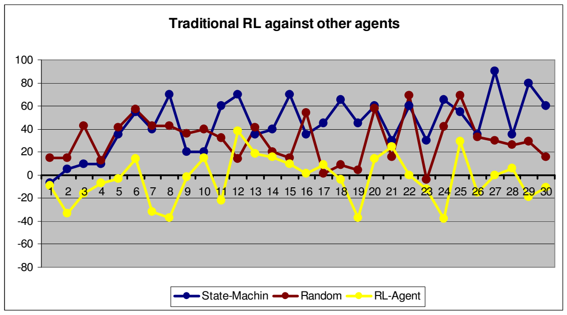

Figure 4: Traditional RL agent's performance

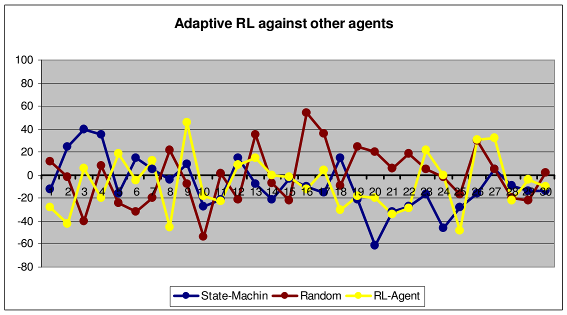

Figure 5: Adaptive RL agent's performance

Figure 5 show the adaptive RL agent performance. Although this agent has the same capabilities as traditional RL, because their learning algorithms are the same, the adaptive mechanism forces him to act at the same opponent level. As a result, the agent performance varies between wins and losses, independently of opponent's skills. The average performance of the agent shows that most of the fights end with a small difference of life, meaning that both fighters had similar performance. Table 1 shows the average life difference for each agent.

Table 1: Average life difference

	State-Mach.	*Trad. RL*	*Adaptive RL*
SM	-0,50	44,10	-8,57
Random	30,76	30,67	-0,67
RL	-34,16	-3,36	-7,10

These results indicate the effectiveness of our approach. Although the adaptive agent could easily beat their opponents, the difficulty level is adapted so it acts nearly the opponent, interleaving wins and loses.

7 Conclusions

This work presented an original approach to construct agents that dynamically adapt their behavior in order to keep the game in a difficulty level adequate to the current user skills. The developed technique combines reinforcement learning [16] with challenge functions [14], and uses RL properties to define an order relation into the quality of the agent possible actions. The approach was successfully applied to a real time fighting game.

Since this work's experiments were restricted to computer agents, a future work is to extend the experiments to human users. Since the main objective is to create intelligent agents that enhance the gameplay, it is necessary to check whether the agents are really entertaining for humans. Therefore, we intend to perform experiments in the future involving human players.

Another direction for future work is testing different offline learning strategies. As online learning is an expensive process, it is important that the initial agents are sufficiently skilled to deal with a broader range of users.

8 References

1. Woodcock, S. *The Game AI Page: Building Artificial Intelligence into Games*, http://www.gameai.com (04/01/2004)

2. Demasi, P., Cruz, A. *Aprendizado de Regras Nebulosas em Tempo Real para Jogos Eletrônicos*. II Workshop Brasileiro de Jogos e Entretenimento Digital, 2003.

3. Spronck, P., Sprinkhuizen-Kuyper, I., Postma, E. *Improving Opponent Intelligence through Machine Learning*. Proceedings of the Fourteenth Belgium-Netherlands Conference on Artificial Intelligence (eds. Hendrik Blockeel and Marc Denecker), pp. 299-306. 2002.

4. Tesauro, G. *TD-Gammon, a self-teaching backgammon program, achieves master-level play*. Neural Computation, 6(2): 215-219, 1994.

5. Samuel, A. *Some studies in machine learning using the game of checkers*. II-Recent progress. IBM Journal on Research and Development, 11:601-617, 1967.

6. Abramson, M., Wechsler, H. *Competitive Reinforcement Learning for Combinatorial Problems*. Int. Joint Conference on Neural Net-works, Washington, DC, 2001.

7. Robocup. *RoboCup Official Site*, http://www.robocup.org/ (01/04/2004).

8. Vasilyev, A., Kapishnikov, A., Sukov, A. *Quick Online Adaptation with Reinforcement of Simulated Soccer Agents*. RoboCup'2003 International Symposium. In press, 2003.

9. Riedmiller, M., Merke, A., Meier, D., Hoffmann, A., Sinner, A., Thate, O., Ehrmann, R. *Karlsruhe Brainstormers – A Reinforcement Learning approach to robotic soccer*. RoboCup-00: Robot Soccer World Cup IV, LNCS, Springer.

10. Demasi, P. *Estratégias Adaptativas e Evolutivas em Tempo Real para Jogos Eletrônicos*. Rio de Janeiro. Dissertação de Mestrado. UFRJ/IM/NCE, 2003.

11. Meyer, C., Ganascia, J-G, Zucker, J-D. *Learning Strategies in Games by Anticipation*. Proceedings of the fifteenth International Joint Conference on Artificial Intelligence, IJCAI'97. Morgan Kaufman Editor, 1997.

12. Ramon, J., Jacobs, N., Blockeel, H. *Opponent modeling by analyzing play*. Third International Conference on Computers Games (CG'02), Workshop on Agents in Computer Games, 2002.

13. Spronck, P., Sprinkhuizen-Kuyper, I., Postma, E. *Online Adaptation of Computer Game Opponent AI*. Proceedings of the 15th Belgium-Netherlands Conference on Artificial Intel-ligence, pp. 291-298. University of Nijmegen, 2003.

14. Andrade, F., Andrade, G., Leitão, A., Furtado, A., Ramalho, G. *Knock'em: Um Estudo de Caso de Processa-mento Gráfico e Inteligência Artificial para Jogos de Luta*. II Workshop Brasileiro de Jogos e Entretenimento Digital, 2003.

15. Klov, *Killer List of Videogames*. Coin-Op Museum. http://www.klov.com/ (03/04/2004).

16. Sutton, R., Barto A. *Reinforcement Learning: An Introduction*. Cambridge, MA. 1998.

17. Spronck, P., Sprinkhuizen-Kuyper, I., Postma, Eric. *Online Adaptation of Game Opponent AI in Simulation and in Practice*. Proceedings of the 4th International Conference on Intelligent Games and Simulation (GAME-ON 2003), pp. 93-100. EUROSIS, Belgium, 2003.

18. Mitchell, T. *Machine Learning*. McGraw Hill, 1997.

19. HUNICKE, Robin, CHAPMAN, Vernell. *AI for Dynamic Difficulty Adjustment in Games*. Challenges in Game Artificial Intelliigence, AAAI Workshop. AAAI Press 2004.