Learning to Overtake in TORCS Using Simple Reinforcement Learning

Daniele Loiacono, Alessandro Prete, Pier Luca Lanzi, Luigi Cardamone

***Abstract*— In modern racing games programming non-player characters with believable and sophisticated behaviors is getting increasingly challenging. Recently, several works in the literature suggested that computational intelligence can provide effective solutions to support the development process of NPCs. In this paper, we applied the Behavior Analysis and Training (BAT) methodology to define a behavior-based architecture for the NPCs in The Open Racing Car Simulator (TORCS), a well-known open source racing game. Then, we focused on two major overtaking behaviors: (i) the overtaking of a fast opponent either on a straight stretch or on a large bend; (ii) the overtaking on a tight bend, which typically requires a rather advanced braking policy. We applied simple reinforcement learning, namely Q-learning, to learn both these overtaking behaviors. We tested our approach in several overtaking situations and compared the learned behaviors against one of the best NPC provided with TORCS. Our results suggest that, exploiting the proposed behavior-based architecture, Q-learning can effectively learn sophisticated behaviors and outperform programmed NPCs. In addition, we also show that the same approach can be successfully applied to adapt a previously learned behavior to a dynamically changing game situation.**

I. Introduction

Racing games represent one of the most popular and successful genres. Modern racing games provide an immersive experience based on stunning representations of the racing environment and realistic physics engines that accurately model all the elements of the car dynamics. In this context, the behavior of the non-player characters (NPC), i.e., the cars controlled by the computer, is becoming more and more sophisticated to meet the increasingly high expectations of the human players. As a consequence, the programming of the artificial intelligence for car racing games is getting more and more challenging. In this respect, computational intelligence can provide effective solutions to support the development of more sophisticated and believable NPC. In the literature, several works [1], [2], [3], [4], [5], [6] investigated the application of computational intelligence in the domain of racing games and reported encouraging results. Computational intelligence has also been applied to commercial car games [7].

In our previous works [8], [9], [10], we applied advanced methods of computational intelligence (on-line and off-line neuroevolution) to learn complete drivers for TORCS. In this work, we followed a completely different approach and applied very simple reinforcement learning, Q-learning [11], to learn a challenging driving behavior (overtaking) which represents a component of a more complex behavior-based architecture. At first, we analyzed the controllers available in TORCS and, by applying the Behavior Analysis and Training (BAT) methodology [12], we structured a complete driver for TORCS as a behavior-based architecture organized in four abstraction layers, each one comprising several interdependent subtasks (or behaviors). Then, we applied Q-learning [13], a simple but widely used reinforcement learning approach, to develop a sophisticated overtaking behavior which we integrated in our behavior-based architecture. We focused on the overtaking behavior since: (i) it is very important to develop a successful driver [10]; (ii) it is one of the weakest point of the controllers provided with TORCS; (iii) most of the previous computational intelligence applications to TORCS focused on improving the driving performance rather than on improving the overtaking behavior [14].

We tested our approach by applying it to tackle two major overtaking situations: (i) the overtaking on a fast opponent either on a straight stretch or on a large bend; (ii) the overtaking on a tight bend which typically requires a rather advanced brake policy. We performed several experiments with different configurations and compared the performance of a driver using our learned overtake policy against the same driver using the overtake policy implemented by `Berniw` — of one of the best drivers provided with TORCS. Our results suggest that, through a suitable task decomposition, even a rather simple learning algorithm like Q-learning can effectively learn sophisticated behaviors and outperform programmed NPCs. In addition, the proposed approach can successfully learn the target behavior in different conditions (e.g., using different car aerodynamics models), without requiring any ad-hoc adjustments.

II. Related Work

The application of computational intelligence to car racing games has been investigated in several works. Pyeatt and Howe [15] applied reinforcement learning to learn racing behaviors in RARS, an open source racing simulator, precursor of The Open Racing Car Simulator (TORCS) [16] used in this work. More recently, evolutionary computation techniques have been applied to improve the performance of a motocross game AI [1], to optimize the parameters in a sophisticated F1 racing simulator [2], and to evolve a neural network to predict crashes in car racing simulator [17]. Togelius and Lucas probably produced most of the work in this area applying neuroevolution to different car racing applications [3], [4], [5], [6]. In particular, they applied neuroevolution to learn controllers both for radio-controlled car models and for a simpler two-dimensional racing simulator.

Daniele Loiacono, Alessandro Prete, Pier Luca Lanzi, and Luigi Cardamone are with Politecnico di Milano, P.zza Leonardo da Vinci 32, 20100, Milano; email: {loiacono,lanzi,cardamone}@elet.polimi.it, alessandro.prete@polimi.it

978-1-4244-8126-2/10/$26.00 ©2010 IEEE

They also investigated several sensory models (e.g., first person based, third person based) and studied the generalization capabilities of the evolved controllers. Cardamone et al. [9] applied on-line neuroevolution to learn a controller for TORCS.

Computational intelligence has also been applied to commercial racing games. In Codemasters' *Colin McRae Rally 2.0* (`http://www.codemasters.com`), a neural network is used to drive a rally car [7]. Instead of handcrafting a large and complex set of rules, a feed-forward multilayer neural network is trained to follow the ideal trajectory, while the other behaviors (e.g., the opponent overtaking or the crash recovery) are programmed [7]. In Microsoft's *Forza Motorsport*, the player can train his own *drivatars*, i.e., controllers that learn a player's driving style and can take its place in the races.

Recently, several simulated car racing competitions have been organized during major conferences (e.g., CEC-2007, WCCI-2008, CIG-2008, CIG-2009, and GECCO-2009) which attracted the interest of new researchers. The competition held at CEC-2007 was based on the 2D simulator developed by Togelius [6]. However, since WCCI-2008, a client/server extension of TORCS has been adopted (see [18] for a description of the most recent distribution). The results of the car racing competition held at CEC-2007 and CIG-2007 are summarized in [19] while the results of the competition organized during WCCI-2008 are discussed in [20], [14]. In addition, we refer the reader to [21], [22], [9], [23], [24], [25], [26], [27] for detailed descriptions of some of the controllers that took to these competitions.

III. Reinforcement Learning

Reinforcement learning is defined as the problem of an *agent* that has to learn how to perform a task through *trial and error interactions* with an unknown *environment* which provides feedback in terms of numerical *reward* [11]. The agent and the environment interact continually. At time t the agent senses the environment to be in state s_t; based on its current sensory input s_t the agent selects an action a_t in the set A of the possible actions; then action a_t is performed in the environment. Depending on the state s_t, on the action a_t performed, and on the effect of a_t in the environment, the agent receives a *scalar reward* r_{t+1} and a new state s_{t+1}. The agent's goal is to *maximize* the amount of reward it receives from the environment *in the long run*, or *expected payoff* [11]. For this purpose, the agent computes an action-value function $Q(\cdot,\cdot)$ that maps state-action pairs (or states) into the corresponding expected return.

Q-learning [13] is perhaps the simplest and the most popular reinforcement learning algorithm which uses a simple *look-up* table (i.e., the Q-table) to store the action-value function $Q(\cdot,\cdot)$. It works as follows. At the beginning, $Q(s,a)$ is initialized either with random or default values. Then, at each time step t ($t > 0$), when the agent senses the environment to be in state s_t, and receives reward r_t for performing a_{t-1} in state s_{t-1}, the position $Q(s_{t-1}, a_{t-1})$ of the Q-table, is updated as,

$$Q(s_{t-1}, a_{t-1}) \leftarrow Q(s_{t-1}, a_{t-1}) + \beta(\Delta_t - Q(s_{t-1}, a_{t-1}))$$

where β is the learning rate, Δ_t is computed as $r_t + \gamma \max_{a \in A} Q(s_t, a)$, and γ is the discount factor [11].

When the agent has to choose which action to perform it faces an *exploration/exploitation dilemma* [11] — a well-known issue that is widely studied in the reinforcement learning community. On the one hand, the agent might decide to *exploit* its current knowledge and choose an action that it expects to produce the best outcome in the current situation. On the other hand, the agent also needs to learn and thus it might decide to *try* something new, a different action, to enrich its knowledge and maybe discover a better action. To deal with this issue, the simplest and mostly used approach consists in using an *ε-greedy exploration strategy*: at each time step, the agent selects a random action with probability ε while, with probability $1-\varepsilon$, it selects the most promising action (according to its current knowledge).

The parameter ε works as a handle that the agent can use to adjust the tradeoff between exploration and exploitation. As the learning proceeds, the exploitation should be reduced and the amount of learning performed should be decrease. Accordingly, both the learning rate β and the exploration rate ε are usually adapted during learning as follows,

$$\beta(s,a) = \frac{\beta_0}{1 + \delta_\beta n(s,a)}$$

$$\varepsilon_N = \frac{\varepsilon_0}{1 + \delta_\varepsilon N}$$

where $n(s,a)$ is the number of updates for $Q(s,a)$, N is the number of learning episodes, δ_β and δ_ε are used to tune the adaptation of the learning rate and the exploration rate.

IV. The Open Racing Car Simulator

The Open Racing Car Simulator (TORCS) [16] is a state-of-the-art open source car racing simulator which provides a sophisticated physics engine, full 3D visualization, several tracks and models of cars, and different game modes (e.g., practice, quick race, championship, etc.). The car dynamics is accurately simulated and the physics engine takes into account many aspects of racing cars such as traction, aerodynamics, fuel consumption, etc.

Each car is controlled by an automated driver or *bot*. At each control step, a bot can access the current game state, which includes several information about the car and the track as well the information about the other cars on the track, and can control the car using the gas/brake pedals, the gear stick, and steering wheel. The game distribution includes several programmed bots which can be easily customized or extended to build new bots.

V. Learning Driving Behaviors for TORCS

In our previous works, we applied neuroevolution and imitation learning to develop complete drivers for TORCS [21], [22], [9], [23], [24], [25], [26], [27]. Although such approaches proved effective in producing competitive drivers (which could win the simulated car racing at WCCI-2008 and reach the 4th place in the 2009 Simulated Car Racing championship), yet they are potentially limited. Accordingly, in this work, we targeted the development of a complete driver using a structured architecture organized in abstraction

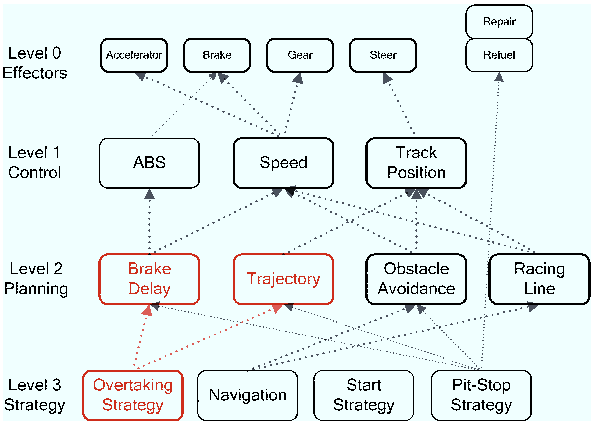


Fig. 1. Structured behavior generated from the first behavior analysis. The red boxes represents the behavior analyzed in this work, overtaking, and its two subtasks considered, trajectory and break delay.

layers, each one consisting of several behaviors which could be either programmed or learned. In particular, we focused on one of the most important behaviors for car racing, overtaking, and applied simple reinforcement learning, namely tabular Q-learning, to implement it.

As the very first step, we analyzed all the programmed drivers (or *bots*) distributed with TORCS, how their structure and the behaviors they implement. Then, we applied a process inspired to the Behavior Analysis and Training (BAT) methodology [12], and devised the architecture shown in Figure 1.

The architecture has four abstraction layers that identify (i) high-level behaviors which operate at a strategic level, (ii) behaviors which represent macro-actions needed for planning, (iii) tasks which implement middle-level control (e.g., ABS), and (iv) elementary actions on the actuators. In Figure 1, boxes identify behaviors; an arrow connecting a behavior A to a behavior B indicates that A depends on B. The *Strategy* level consists of complex tasks based on a high-level analysis of the current situation. For example, the overtaking strategy has to take into account whether the surpass will take place during a straight stretch or (so as to activate a behavior to follow the best *trajectory*, exploiting the drag effect) or during a bend (so as to activate a behavior for *brake delay*). The *Planning* level consists of tasks that represents macro-actions (e.g., brake delay, trajectory, and obstacle avoidance) which are used by higher-level behaviors. For example, the current car trajectory has to be modified both during overtaking and during navigation. The *Control* level consists of low-level control behaviors such as speed control (responsible of changing the car speed using the accelerator, the brake and the gear), track position (responsible of changing the car position through steering actions), and the Antiblock Braking System (ABS) which available in most TORCS bots. Finally, the *Effectors* level consists of simple actions that control the actuators.

VI. Learning to Overtake

In this work, we focused on the overtaking behavior, one of the most important high-level components needed in car racing. In our BAT analysis, we broke the behavior down into two main tasks which we implemented as two separate modules: *Trajectory* and *Brake Delay*. The *Trajectory* behavior deals with the challenging task of overtaking, on a straight stretch, an opponent driving at a high speed. Accordingly, the driver has to learn how to exploit the drag effect of the opponent car in front so as to gain speed and complete the overtaking. Note that, we did not consider the simpler case of a much slower opponent since, in our BAT decomposition, we view such a case as a navigation problem that is managed using the obstacle avoidance module (see Figure 1). The *Brake Delay* behavior deals with the overtaking of an opponent in a tight bend when the driver might need to delay the braking action to successfully surpass an opponent. These two behaviors are combined in the *Overtaking Strategy* component which is responsible to activate the most adequate behavior in each situation using a simple heuristic. For all the other behaviors in our architecture, we used implementations available in the Berniw bot, one of the fastest driver available in the TORCS distribution.

VII. Overtaking SubTask 1: Trajectory

At first, we focused on the challenging task of overtaking, on a straight stretch, an opponent driving at high speed. Accordingly, our driver must learn how to exploit the drag effect to gain speed so as to complete the overtaking. For this purpose, our driver should drive close behind the opponent and, when the wake effect results in a significant speed increase, our driver must change direction and perform the overtake. During this sequence of actions, our driver should also avoid collisions and going off-track.

We considered three cases: (i) the opponent follows a fixed trajectory and the standard aerodynamics available in TORCS is used; (ii) the opponent follows a fixed trajectory, but TORCS is used with a modified aerodynamics which increases the drag effect; finally, (iii) the opponent continuously modifies its trajectory and TORCS standard aerodynamics is used.

For each case, we applied simple tabular Q-learning to learn the overtaking policy and compared the performance of Q-learning against the performance of the same driver using the hand-coded policy available in the Berniw bot (one of the fastest driver available in TORCS).

A. Sensors and Effectors

To apply tabular Q-learning to learn how to overtake, we need to define the problem state-action space. For this task, the Trajectory behavior receives information about the driver relative position to the track edges, the relative position of the opponent, and the speed difference between the two cars. This information is provided by four sensors which describe the behavior state space (Table I). The $dist_y$ sensor returns the distance between the controlled car and the opponent along the track axis. The $dist_x$ sensor returns the distance between the controlled car and the opponent along the direction orthogonal to the track axis. $dist_x$ is negative when the opponent is to the left of the driver; it is positive when the opponent is to the right of the driver. The pos sensor returns the distance between the driver and the track

TABLE I

SENSORS USED FOR THE OVERTAKING TASKS. FOR EACH SENSOR ARE REPORTED (I) THE RANGE OF THE SENSOR VALUES AND (II) THE DISCRETIZATION APPLIED TO REPRESENT THE SENSORS VALUES IN A LOOKUP TABLE.

Name	Range	Representation
$dist_y$	[0, 200]	{ [0,10) [10,20) [20,30) [30,50) [50,100) [100,200] }.
$dist_x$	[-25,25]	{ [25,-15) [-15,-5) [-5,-3) [-3,-1) [-1,0) [0,1) [1,3) [3,5) [5,15) [15,25] }.
pos	[-10, 10]	{ [-10,-5) [-5,-2) [-2,-1) [-1,0) [0,1) [1,2) [2,5) [5,10] }.
Δ_{speed}	[-300, 300]	{ [-300,0) [0,30) [30,60) [60,90) [90,120) [120,150) [150,200) [200,250) [250,300] }.

axis, measured along the direction orthogonal to the track axis: pos is negative when the driver is on the right side of the track; it is positive the driver is on the left side of the track. Finally, the Δ_{speed} sensors returns the speed difference between the driver and the opponent. Since Q-learning cannot be applied to real-values, all these sensors are discretized and mapped into the intervals reported in Table I.

The Trajectory behavior only controls the steering wheel (represented by the *Track Position* behavior at the control level) while the gas pedal and the brake are controlled by other components (mainly the *Navigation* behavior which controls both obstacle avoidance and the following of the best racing line). Accordingly, Trajectory can perform three main high-level actions: it can move one meter to the left, it can keep its position, or it can move one meter to the right. These three actions are encoded respectively as -1, 0, and 1. Note that, these steering actions work at a higher level than what has always been used thus far (e.g., [21], [22], [9], [23], [24], [25], [26], [27]), therefore Q-learning actions will result in smooth changing of trajectories.

B. Design of Experiments

Each experiment consists in a number of learning episodes. Each episode starts in a random configuration with (i) the opponent car, controlled by the `Berniw` bot, on a straight stretch of the track; (ii) our car positioned behind the opponent car in a randomly generated frontal distance value from the opponent car, and a randomly generated lateral distance. The learning episode ends when the car goes off road, collides with the other car or when the overtake is successfully accomplished (i.e., when the car distance from the opponent is 0 meters). To apply Q-learning, we defined the following reward function:

$$r_t = \begin{cases} 1 & \text{if goal was reached} \\ -1 & \text{if car crashed} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

The control time chosen for this subtask is of 10 TORCS's game ticks, each one corresponding to 0.2 real seconds in the simulated environment.

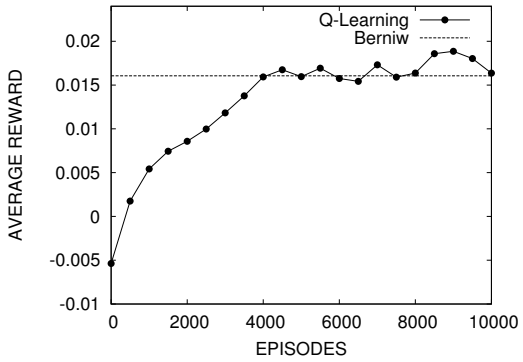


Fig. 2. Q-learning applied to the Overtaking subtask - Case 1.

C. Case 1: Opponent on Fixed Trajectory and Standard Aerodynamics

In the first experiment, we applied Q-learning to implement the Trajectory behavior with the following parameters setting: $\beta_0 = 0.5$, $\gamma = 0.95$, $\delta_\beta = 0.05$, $\varepsilon_0 = 0.5$, $\delta_\varepsilon = 0.0005$. The experiment was run for 10000 episodes.

Figure 2 compares the average reward per step achieved by the Q-learning against the reward obtained by the programmed overtaking policy of `Berniw` (computed as an average over 10000 episodes). As the learning curve shows, Q-learning needs less than 1000 learning episodes to achieve a positive average reward and approximately 5000 learning episodes to reach the performance of `Berniw`.

The analysis of the overtaking policy learned by Q-learning after 10000 episodes showed a rather sophisticated behavior: the bot is able to successfully overtake in different situations and it is also able to exploit drafting to overtake more effectively the opponent. The learned overtaking policy is also very reliable in that it is almost always able to overtake successfully the opponent without mistakes. Table II compares the performances of the learned overtaking policy to the ones programmed in `Berniw`. Two performance measures are reported: (i) the time (in seconds) required to reach the goal, i.e., the time required to overtake the opponent, and (ii) the highest speed (in kilometers per hour) achieved during the overtake. Results shows that the overtaking policy learned with Q-learning is able to complete the overtake with an average time that is slightly shorter than the one required by `Berniw`. We applied the Wilcoxon's rank-sum test to the data in Table II to check whether the reported differences are statistically significant. The analysis showed that the differences are statistically significant with a 99% confidence for time to complete the overtake as well as for the highest speed achieved during the overtake.

It should be noted that, although this setup might be considered rather simple the learned policy results in a significant performance advantage. In fact, even what appears as a small difference of 0.3 seconds is not negligible in car racing. In addition, the analysis of the highest speed achieved during the overtaking suggests that the learned policy exploits the drag reduction much better than `Berniw`.

TABLE II

COMPARISON OF THE OVERTAKING POLICY LEARNED WITH Q-LEARNING AND THE OVERTAKING POLICY PROGRAMMED IN `Berniw`. STATISTICS ARE AVERAGES OVER 200 OVERTAKING EPISODES.

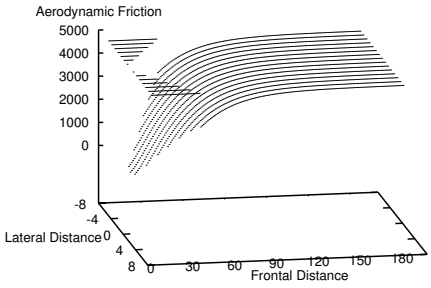
Overtaking Policy	**Overtake Time** ($\mu \pm \sigma$)	**Highest Speed** ($\mu \pm \sigma$)
Q-learning	15.12 $\pm$1.08	205.57 $\pm$1.66
`Berniw`	15.44 $\pm$1.11	197.44 $\pm$1.91

D. Case 2: Opponent on Fixed Trajectory and Challenging Aerodynamics

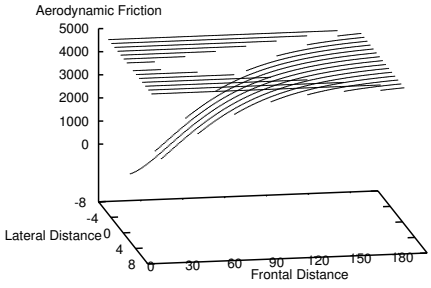
In the second experiment, we tested whether Q-learning could learn to overtake with a different, more challenging, aerodynamics. Accordingly, we modified the aerodynamic drag model of TORCS and applied Q-learning with the same setup used in the previous experiment. Figure 3a shows how the aerodynamic drag behind a car is modeled in TORCS. As can be noted, there is a large area (approximately 20 degrees spread) that benefits from drag reduction. We modified the original aerodynamics to make it more challenging by reducing the area that benefits by the drag effect. Figure 3b shows the modified aerodynamic drag model that has a rather small area affected from a drag reduction (approximately 5 degrees spread); note however, that the modified drag can be perceived even at a higher distance from the preceding car. Accordingly, the new aerodynamic model would result in a more challenging overtaking problem.

Q-learning was applied for 10000 episodes with the same parameters setting used in the previous experiment. Figure 4 shows the average reward per step achieved by the Q-learning over the learning episodes. We also report the reward per step achieved by the programmed overtaking policy of `Berniw` (computed as an average over 10000 episodes). Similarly to the previous experiment, Q-learning requires less than 1000 episodes to reach a positive average reward and approximately 5000 episodes to reach the performance of `Berniw`. However, in this experiment, Q-learning clearly outperforms `Berniw` policy at the end of the learning process. This is not surprising as Q-learning is able to adapt to the different drag model. Nevertheless, the analysis of the resulting overtaking policy (videos are available at `http://home.dei.polimi.it/loiacono/loiacono10learning.html`) shows that the learned overtaking policy features a very effective drafting behavior.

Table III compares the performance of the overtaking policy learned with Q-learning to the one of `Berniw` both in terms of average time to complete the overtake and in terms of highest speed achieved during the overtake. The results show that the overtaking policy learned by Q-learning is able to outperform the `Berniw`'s strategy both in terms of time and in terms of speed, Finally, we applied the Wilcoxon's rank-sum to data in Table III to see whether the differences are statistically significant. The analysis showed that the differences are statistically significant with a 99% confidence for both the performance measures.



(a)



(b)

Fig. 3. Aerodynamic drag behind a car: (a) original model and (b) new model.

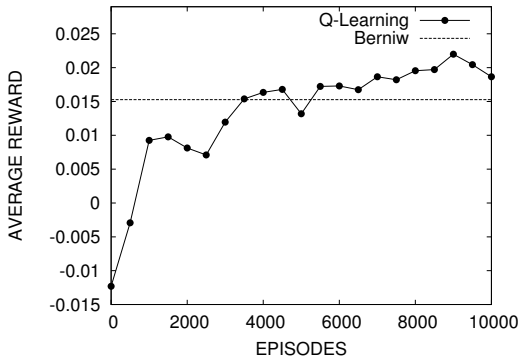


Fig. 4. Q-learning applied to the Overtaking subtask - Case 2.

TABLE III

COMPARISON OF THE OVERTAKING POLICY LEARNED WITH Q-LEARNING AND THE OVERTAKING POLICY PROGRAMMED IN `Berniw` WITH A DIFFERENT DESIGN OF THE AERODYNAMIC DRAG MODEL. STATISTICS ARE AVERAGES OVER 200 OVERTAKING EPISODES.

Overtaking Policy	**Overtake Time** ($\mu \pm \sigma$)	**Highest Speed** ($\mu \pm \sigma$)
Q-learning	14.79 $\pm$1.08	198.98 $\pm$4.83
`Berniw`	16.05 $\pm$1.21	190.04 $\pm$1.90

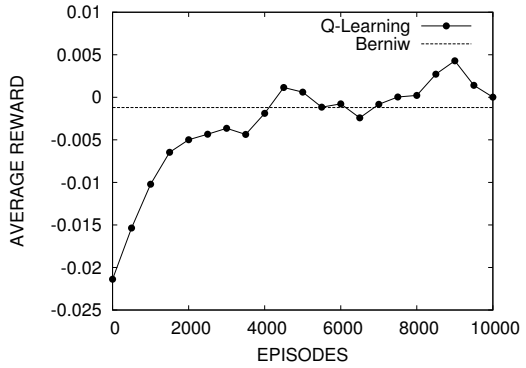


Fig. 5. Q-learning applied to the Overtaking subtask - Case 3.

TABLE IV
COMPARISON OF THE OVERTAKING POLICY LEARNED WITH Q-LEARNING AND THE OVERTAKING POLICY PROGRAMMED IN `Berniw` AGAINST AN IMPROVED OPPONENT. STATISTICS ARE AVERAGES OVER 200 OVERTAKING EPISODES.

Overtaking Policy	Overtake Time ($\mu \pm \sigma$)	Highest Speed ($\mu \pm \sigma$)	Success Rate
Q-learning	20.87 $\pm$2.72	203.57 $\pm$6.16	90.16%
`Berniw`	21.27 $\pm$2.48	201.41 $\pm$7.36	70.12%

E. Case 3: Opponent on Random Trajectory and Standard Aerodynamics

In the last experiment on the Trajectory behavior, we considered a more challenging opponent which, instead of following the middle line, as in the previous experiments, or the best racing line, as usually done by programmed bots, it randomly changes its position on the track and its speed (selecting it among Km/h 150 and 180 Km/h)

Figure 5 compares the average reward per step achieved by Q-learning against the the reward achieved by the programmed overtaking policy of `Berniw` (computed as an average over 10000 episodes). Results confirm the finding of the previous experiments: Q-learning learns an overtaking policy tailored for the new opponent behavior. In contrast, `Berniw`'s policy is not able to cope with the new opponent's behavior.

Table IV compares the performances of Q-learning and `Berniw`. In addition to the usual performance measures, the table also shows the percentage of successfully completed overtakings. The results show that the policy learned by Q-learning outperforms the policy of `Berniw` both in terms of time required to complete the overtake and in terms of speed achieved. It is worth noting that, when facing a more challenging opponent, the overtaking policy learned by Q-learning is able to complete a significantly higher number of overtakings. In fact, the Wilcoxon's rank-sum test on the data in Table IV shows that the reported differences are statistically significant with a 99% confidence for all the performance measures.

VIII. OVERTAKING SUBTASK 2: BRAKE DELAY

In this section, we take into account another driving task in which a car has to overtake an opponent in a tight bend, using brake delay. This task is managed by a specific behavior, called *Brake Delay* (Figure 1), and may activate on top of the normal overtaking policy that is usually applied to deal with opponent overtaking. With standard overtaking behaviors, it is very difficult to accomplish an overtake when the cars are approaching a turn: being the target speed generally controlled by a general navigation behavior, the car slows down just before the incoming turn as it does when there are no opponents to overtake. In contrast, a human driver would typically delay his braking action to overtake the opponent just before approaching the turn. Note that, the Brake Delay behavior needs to be very accurate, since delaying the braking action too much would easily result in running out of the track.

A. Sensors and Effectors

To define the state space for the Brake Delay task, we introduced a new sensor that provides information about the relative position of the next turn: the $dist_{turn}$ sensor returns the distance between the current position of the car and the next turn along the track axis. Then, to complete the state space definition, we used the $dist_y$ and Δ_{speed} sensors introduced in Section VII-E. Please notice that the state space does not include the $dist_x$ and pos sensors, as the Brake Delay behavior is not in charge of choosing the trajectory to follow during the overtake and, thus, it does not need any information either about the current position on the track or about lateral distance of the opponent.

To apply the tabular Q-learning to the continuous state space defined above, we mapped $dist_y$ and Δ_{speed} on the same discrete intervals previously defined (see I), while $dist_{turn}$ was mapped into the following six intervals { [0 1) [1 2) [2 5) [5 10) [10 20) [20 50) [50 100) [100 250] }.

Concerning the action space, the Braking Delay behavior can only affect the brake pedal, while the control of steering wheel is controlled by other components. In particular, the Brake Delay is built on top of two components at the control level, the *ABS* and *Speed*, and can performance two high-level action: it can either inhibit the braking action or leave the decision of braking to the lower level components. These two actions are respectively encoded as 1 and 0.

B. Design of Experiments

Each experiment consists in a number of learning episodes. Each episode starts from a specific configuration that represents an opponent overtake while approaching a turn. Although the initial configuration is not completely random as in the previous experiments, it is not even fully deterministic. In fact, during the simulation, we allowed small random variations in the initial configuration that, despite being very small, significant affect the situation faced by the driver. The learning episode ends when the car goes off road, collides with the other car or when the overtake is successfully accomplished, i.e., when the car is one meter ahead the opponent.

To apply Q-learning, we defined the following reward

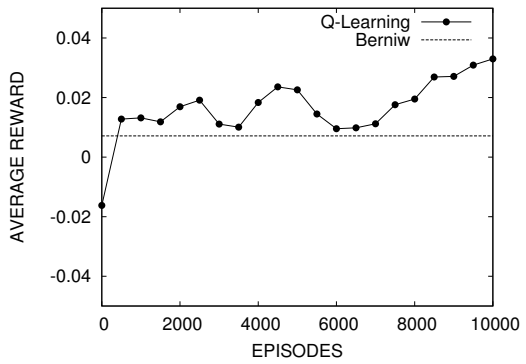


Fig. 6. Q-learning applied to the Brake Delay subtask.

function:

$$r_t = \begin{cases} 1 & \text{if the goal was reached} \\ -1 & \text{if the car go off road} \\ 0 & \text{otherwise} \end{cases}$$

As in the previous experiments, the control time chosen for this subtask is equal to 10 standard simulation ticks in TORCS, corresponding to 0.2 real seconds in the simulated environment.

C. Experimental Results

We applied Q-learning to learn the Brake Delay task using the same parameters setting of the previous experiments: $\beta_0 = 0.5$, $\gamma = 0.95$, $\delta_\beta = 0.05$, $\varepsilon_0 = 0.5$, $\delta_\varepsilon = 0.0005$. The experiment was run for 10000 episodes. Figure 6 compares the average reward per step achieved by the Q-learning against the reward obtained by `Berniw`. As in the previous experiments, the average reward achieved by `Berniw` is computed as an average over 10000 episodes. As the learning curve shows, Q-learning quickly reaches the performance of `Berniw` and clearly outperforms it at the end of the learning process. This result is not surprising, as `Berniw` does not implement any kind of brake delay strategy. Table V compares the average outcome of the overtake performed using Berniw, the Brake Delay behavior learned with Q-learning and a completely random Brake Delay behavior. Each overtake is recorded as a *success* if it is accomplished by the end of the approached turn, as a *crash* if the car runs out of the track or hits the opponent, or as a *failure* otherwise. Statistics are computed as an average over 1000 episodes. The Wilcoxon's rank-sum applied to the data in Table V shows that the differences between Q-learning and `Berniw` are statistically significant with a 99% confidence. As expected, results show that the behavior learned by Q-learning leads to a success rate much higher than the one achieved by `Berniw`. However, the learned behavior seems also to be slightly more risky as it results in a small probability of crashing greater than zero (5.2%). A video showing the learned behavior is available at at `http://home.dei.polimi.it/loiacono/loiacono10learning.html`. Finally, it is worthwhile to stress that the performance of a completely random Brake Delay behavior are very poor with a success rate equal to 0.96%. This suggests that the Brake Delay is a rather challenging problem where the driver has to choose carefully and timely when to delay the braking action and when to not.

TABLE V
COMPARISON OF THE BRAKE DELAY POLICY LEARNED WITH Q-LEARNING AND THE OVERTAKING POLICY OF `Berniw`. STATISTICS ARE COMPUTED OVER 1000 EPISODES.

Policy	Success	Failure	Crash
Q-learning	88.1%	6.7%	5.2%
`Berniw`	18.8%	81.2%	0%
Random	0.96%	15.2%	75.2%

IX. ON-LINE LEARNING FOR CHANGING CONDITIONS

Computational intelligence provides several ways to introduce adaptive behaviors in modern computer games so as to offer an improved game experience. For instance, methods of computational intelligence can be used to implement in-game adaptation to changing situations or to users preferences. In [9], on-line neuroevolution has been applied to tune an existing driving policy to different situations.

In the last set of experiments, we applied on-line learning to the *Brake Delay* behavior to develop an overtaking strategy which could dynamically adapt to changing conditions. We compared the performance of the typical fixed policy against the performance of the adaptive policy. For this purpose, we set up an experiment where the wheel friction would dynamically change to simulate the wheel consumption that usually happens during a real race. As the wheel friction decreases, the previously learned policy is no longer effective: when the friction is low, it is necessary to brake much before otherwise the car is too fast to remain on track. So, it is necessary to dynamically adapt the braking policy using on-line learning.

A. Design of Experiments

The sensors and effectors used in this experiment are the same used in Section VIII. The experimental setup is similar to the one of Section VIII. The only difference is the dynamic changing of the wheel friction which in this case was progressively decreased. During an experiment, the wheel friction started at its default value; every 1000 episodes the frictions was decreased of the 3.75%. An experiment ended when the wheel friction was around 85% of the original value.

B. Experimental Results

In this set of experiments, Q-Learning started with a Q-Table learned in the previous set of experiments and it is applied to the new changing task with the following parameters: $\alpha_0 = 0.4$, $\gamma = 0.95$, $\delta_\alpha = 0.0$, $\epsilon_0 = 0.01$, $\delta_\epsilon = 0.0$. The learning process was stopped after 12000 episodes.

Figure 7 compares the average reward achieved by the adaptive policy against the reference non-adaptive policy. This suggests that, when the wheel friction is similar to the initial value, the overhead of the learning process is acceptable. When the wheel consumption is significant, and the friction is low, the fixed policy is no longer effective.

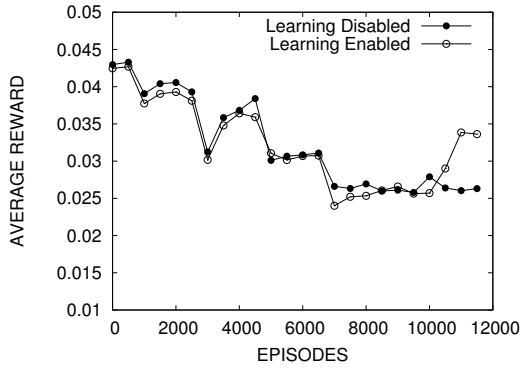


Fig. 7. Learning Disabled policy vs Learning Enabled policy for the Brake Delay task with decreasing wheel friction.

In contrast, the adaptive policy can learn to perform well to the new conditions and significantly outperforms the fixed policy.

X. Conclusions

In this paper, we applied simple reinforcement learning, namely Q-learning [11], to develop an advanced overtaking policy which we integrated into a behavior-based architecture implementing a complete driver for TORCS. The architecture we developed using the Behavior Analysis and Training (BAT) methodology [12] is organized in four abstraction layers each one comprising several interdependent behaviors. The overtaking behavior was broke down into two subtasks that deal with separate situations: (i) the overtaking of fast opponents on straight stretches or large bends; (ii) the overtaking on tight bends, which requires an advanced braking strategy. We performed several experiments to test the proposed approach in different situations including cases in which in-game learning is applied to adapt the overtaking strategy to dynamically changing wheel consumption. Our results show that even simple Q-learning can produce competitive overtaking behaviors which can outperform the strategy employed by one of the best drivers available in TORCS distribution. The experiments with dynamic wheel consumption also demonstrate that Q-learning can be successfully applied to adapt a previously learned strategy to a changing situation, outperforming static strategies when the change becomes significant. Videos showing the learned and the programmed behavior are available at `http://home.dei.polimi.it/loiacono/loiacono10learning.html`

Future research directions include the implementation of different behaviors as well as the use of different approaches (e.g., neuroevolution) to implement overtaking behaviors.

References

[1] B. Chaperot and C. Fyfe, "Improving artificial intelligence in a motocross game," in *IEEE Symposium on Computational Intelligence and Games*, 2006.

[2] K. Wloch and P. J. Bentley, "Optimising the performance of a formula one car using a genetic algorithm," in *Proceedings of Eighth International Conference on Parallel Problem Solving From Nature*, 2004, pp. 702–711.

[3] J. Togelius and S. M. Lucas, "Evolving robust and specialized car racing skills," in *Proceedings of the IEEE Congress on Evolutionary Computation*, 2006.

[4] ——, "Evolving controllers for simulated car racing," in *Proceedings of the Congress on Evolutionary Computation*, 2005.

[5] ——, "Arms races and car races," in *Proceedings of Parallel Problem Solving from Nature*. Springer, 2006.

[6] J. Togelius, "Optimization, imitation and innovation: Computational intelligence and games," Ph.D. dissertation, Department of Computing and Electronic Systems, University of Essex, Colchester, UK, 2007.

[7] J. Hannan, "Interview to jeff hannan," 2001, http://www.generation5.org/content/2001/hannan.asp. [Online]. Available: http://www.generation5.org/content/2001/hannan.asp

[8] L. Cardamone, "On-line and off-line learning of driving tasks for the open racing car simulator (torcs) using neuroevolution," Master's thesis, Politecnico di Milano, 2008.

[9] L. Cardamone, D. Loiacono, and P.-L. Lanzi, "On-line neuroevolution applied to the open racing car simulator," in *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, May 2009, pp. 2622–2629.

[10] L. Cardamone, D. Loiacono, and P. L. Lanzi, "Evolving competitive car controllers for racing games with neuroevolution," in *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. New York, NY, USA: ACM, 2009, pp. 1179–1186.

[11] R. S. Sutton and A. G. Barto, *Reinforcement Learning – An Introduction*. MIT Press, 1998.

[12] M. Dorigo and M. Colombetti, *Robot Shaping: An Experiment in Behavior Engineering*. MIT Press/Bradford Books, 1998.

[13] C. Watkins, "Learning from delayed reward," Ph.D. dissertation, 1989.

[14] D. Loiacono, J. Togelius, P. L. Lanzi, L. Kinnaird-Heether, S. M. Lucas, M. Simmerson, D. Perez, R. G. Reynolds, and Y. Saez, "The wcci 2008 simulated car racing competition," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2008.

[15] L. D. Pyeatt and A. E. Howe, "Learning to race: Experiments with a simulated race car," in *Proceedings of the Eleventh International Florida Artificial Intelligence Research Society Conference*. AAAI Press, 1998, pp. 357–361.

[16] "The open racing car simultaor website," http://torcs.sourceforge.net/. [Online]. Available: http://torcs.sourceforge.net/

[17] K. O. Stanley, N. Kohl, R. Sherony, and R. Miikkulainen, "Neuroevolution of an automobile crash warning system," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2005)*, 2005.

[18] D. Loiacono, L. Cardamone, and P. L. Lanzi, "Simulated car racing championship 2009: Competition software manual," Dipartimento di Elettronica e Informazione – Politecnico di Milano, Tech. Rep. 2009.04, 2009.

[19] J. Togelius, S. M. Lucas, H. D. Thang, J. M. Garibaldi, T. Nakashima, C. H. Tan, I. Elhanany, S. Berant, P. Hingston, R. M. MacCallum, T. Haferlach, A. Gowrisankar, and P. Burrow, "The 2007 ieee cec simulated car racing competition," *Genetic Programming and Evolvable Machines*, vol. 9, no. 4, pp. 295–329, 2008.

[20] D. Loiacono and J. Togelius, "Competitions @ wcci-2008: simulated car racing competition," *SIGEVOlution*, vol. 2, no. 4, pp. 35–36, 2007.

[21] D. Perez, G. Recio, Y. Saez, and P. Isasi, "Evolving a fuzzy controller for a car racing competition," in *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, Sept. 2009, pp. 263–270.

[22] D. T. Ho and J. Garibaldi, "A fuzzy approach for the 2007 cig simulated car racing competition," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2008.

[23] L. Cardamone, D. Loiacono, and P. L. Lanzi, "Evolving competitive car controllers for racing games with neuroevolution," in *GECCO*, F. Rothlauf, Ed. ACM, 2009, pp. 1179–1186.

[24] J. Munoz, G. Gutierrez, and A. Sanchis, "Controller for torcs created by imitation," in *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, Sept. 2009, pp. 271–278.

[25] M. V. Butz and T. Lönneker, "Optimized sensory-motor couplings plus strategy extensions for the TORCS car racing challenge," *IEEE Symposium on Computational Intelligence and Games*, vol. CIG 2009, pp. 317–324, 2009.

[26] E. Onieva, D. A. Pelta, J. Alonso, V. Milanés, and J. Perez, "A modular parametric architecture for the torcs racing engine," *IEEE Symposium on Computational Intelligence and Games*, vol. CIG 2009, pp. 256–262, 2009.

[27] M. Ebner and T. Tiede, "Evolving driving controllers using genetic programming," in *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, Sept. 2009, pp. 279–286.