Hindawi Publishing Corporation
Advances in Artificial Intelligence
Volume 2013, Article ID 492852, 10 pages
http://dx.doi.org/10.1155/2013/492852

Research Article

A Novel Reinforcement Learning Architecture for Continuous State and Action Spaces

Víctor Uc-Cetina

Facultad de Matemáticas, Universidad Autónoma de Yucatán, Periférico Norte Tablaje 13615, Apartado Postal 192, C.P. 97119 Mérida, Yucatán, Mexico

Correspondence should be addressed to Víctor Uc-Cetina; uccetina@uady.mx

Received 3 December 2012; Revised 22 February 2013; Accepted 1 April 2013

Academic Editor: Farouk Yalaoui

We introduce a reinforcement learning architecture designed for problems with an infinite number of states, where each state can be seen as a vector of real numbers and with a finite number of actions, where each action requires a vector of real numbers as parameters. The main objective of this architecture is to distribute in two actors the work required to learn the final policy. One actor decides what action must be performed; meanwhile, a second actor determines the right parameters for the selected action. We tested our architecture and one algorithm based on it solving the robot dribbling problem, a challenging robot control problem taken from the RoboCup competitions. Our experimental work with three different function approximators provides enough evidence to prove that the proposed architecture can be used to implement fast, robust, and reliable reinforcement learning algorithms.

1. Introduction

Applying reinforcement learning (RL) to solve real-world robotic problems is certainly not so common nowadays mainly because most RL methods require several training episodes to learn an optimal policy. This condition supposes having a robot performing a task several thousand times, as it learns through reinforcement learning. In addition to the time required for the training process, we must also consider the time we must spend calibrating sensors and actuators, and the possible damage the robots may suffer. Therefore, one common approach is to first try to solve difficult problems with continuous states and actions in simulated environments, where even the noise of real sensors and actuators can be simulated.

In this paper we propose a novel RL architecture for continuous state and actions spaces. Such an architecture was tested with a difficult control problem in the official simulator of the RoboCup [1]. The Robot World Cup or RoboCup for short is an international tournament taking place every year since 1997, each year in a different country. The RoboCup is known up to date as a standard and challenging problem for artificial intelligence and robotics. The most important goal of RoboCup is to advance the overall technological level of society, and as a more pragmatic goal to achieve the following.

By mid-twenty-first century, a team of fully autonomous humanoid robot soccer players shall win the soccer game, complying with the official rule of the FIFA, against the winner of the most recent World Cup.

One of the competitions in this tournament is the simulation league. In this category two teams of eleven virtual soccer players each play for ten minutes. The main advantage of this league is that it allows us to focus more on higher level concepts and less on the hardware problems related to working with real robots. In general, the simulator provides a challenging testbed due to its nondeterministic behavior with real-time demands and semistructured conditions. The robot dribbling problem, a challenging control problem taken from this competition, is perfect for our purpose. It is difficult to solve and it requires handling continuous states and actions. Solutions to the dribbling problem using reinforcement learning were first provided by Gollin [2].

As we have already mentioned, RoboCup has become a popular testbed for new artificial intelligence methods in general and for machine learning methods in particular. For instance, Riedmiller and Gabel [3] have been working on

the application of reinforcement learning to solve problems in the RoboCup, especially in the simulation league. Other recent research works related to RL and robot soccer are presented by Cherubini et al. [4] and Leng and Lim [5]. In the former, the authors compare two learning algorithms based on policy gradient to solve the humanoid walking gait prob lem, which is not a trivial issue addressed in humanoid robotic soccer. In the latter, a simulation testbed is introduced and it is used to analyze the effectiveness of different RL algorithms, specially in a competitive and cooperative learning framework, involving several goal-oriented agents. Some other researchers have proposed simplified versions of the RoboCup simulator. This is the case of Stone and his research group, who proposed the keepaway domain [6].

Reinforcement learning methods for problems with continuous state and action spaces have become more and more important, as an increasing number of researchers try to solve real-world problems. In a recently published work, Montazeri et al. [7] present a novel algorithm based on growing self-organizing maps, which is shown to be effective in solving the continuous state-action problem in RL. However, among the most promising RL methods for continuous state and action spaces are the ones based on the actor-critic architecture [8]. Crites and Barto [9] introduced an actor-critic algorithm that is equivalent to Q-learning constrained by a particular exploration strategy. In this method, Q-values are encoded within the policy and value function of the actor and critic. In general, it updates the critic only when the most probable action is performed from any given state, and it rewards the actor taking into account the relative probability of the action that was executed. The authors provided a convergence proof for the case where the state and action sets are finite. Algorithms based on the standard actor-critic architecture are structured in 2 main modules. One module known as the actor which implements a policy that maps states to actions, and a second module known as the critic which attempts to estimate the value of each state in order to provide useful feedback to the actor. In such methods the actor adapts to the critic and the critic adapts to the actor. Learning in both modules is obtained through the computation of the temporal difference error.

In the next section, we provide a basic background on reinforcement learning and the standard actor-critic architecture. Then, in Section 3 we introduce our proposed A^2C architecture and one RL algorithm based on it. Section 4 gives details about our experimental work, as well as some discussions about the main results we obtained. Finally, Section 5 presents our conclusions and gives suggestions for possible extensions of our research work.

2. Background

In reinforcement learning [8, 10, 11], the central idea is that of an agent learning to accomplish a goal through its interaction with an environment. Such a problem is commonly approached using the Markov decision process (MDP) framework [12–15]. The agent interacts with the environment several times and gather, information about the rewards obtained and the states visited, after performing different actions in different states.

Formally, a Markov decision process (MDP) is a tuple (S, A, P, γ, R), where

(i) S is a set of states,

(ii) A is a set of actions,

(iii) $P(s_{t+1} \mid s_t, a_t)$ are the state transition probabilities for all states $s_t, s_{t+1} \in S$ and actions $a \in A$,

(iv) $\gamma \in [0, 1)$ is a discount factor,

(v) $R : S \times A \rightarrow \mathfrak{R}$ is the reward function.

The MDP dynamics is follows. An agent in state $s_t \in S$ performs an action a_t selected from the set of actions A. As a result of performing action a_t, the agent receives a reward with expected value $R(s_t, a_t)$, and the current state of the MDP transitions to some successor state s_{t+1}, according to the transition probability $P(s_{t+1} \mid s_t, a_t)$. Once in state s_{t+1} the agent chooses and executes an action a_{t+1}, receiving reward $R(s_{t+1}, a_{t+1})$ and moving to state s_{t+2}, The agent keeps choosing and executing actions, creating a path of visited states $s_t, s_{t+1}, s_{t+2}, \ldots$ and obtaining the following rewards:

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \cdots \quad (1)$$

The reward at timestep t is discounted by a factor of γ^t. By doing so, the agent gives more importance to those rewards obtained sooner. In an MDP, we try to maximize the sum of expected rewards obtained by the agent

$$E\left[R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \cdots\right]. \quad (2)$$

A policy is defined as any function $\pi : S \rightarrow A$ mapping states to the actions. A value function $V(s)$ for a policy π is defined as the expected sum of discounted rewards, obtained by performing always the actions provided by π as

$$V^{\pi}(s) = E\left[R(s_0, \pi(s_0)) + \gamma R(s_1, \pi(s_1)) + \gamma^2 R(s_2, \pi(s_2)) + \cdots \mid s_0 = s, \pi\right]. \quad (3)$$

V^{π} is the expected sum of discounted rewards that the agent would receive if it starts in state s and takes actions given by π. Given a fixed policy π, its value function V^{π} satisfies the following Bellman equation:

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} P\left(s' \mid s, \pi(s)\right) V^{\pi}\left(s'\right). \quad (4)$$

The optimal value function is defined as

$$V^{*}(s) = \max_{\pi} V^{\pi}(s). \quad (5)$$

This function gives the best possible expected sum of discounted rewards that can be obtained using any policy π. Using (4) and (5), we can obtain the Bellman equation for the optimal value function as

$$V^{*}(s) = \max_{a \in A}\left[R(s, a) + \gamma \sum_{s' \in S} P\left(s' \mid s, a\right) V^{*}\left(s'\right)\right]. \quad (6)$$

The Bellman equation is fundamental in the design of RL algorithms. In general, using the data collected by the agent, RL algorithms compute estimates of the state or action value function.

Value functions could be stored using lookup tables. However, interesting problems have a large number of states and actions, sometimes, an infinite number of them. For such cases, we need to replace the lookup table with a function approximator. Using a function approximator naturally complicates the learning process, since we need to deal with more parameters. However, it is the most effective way that we have so far to deal with the curse of dimensionality. Another promising way to deal with complex state and action spaces is based on the exploitation of temporal abstractions, through the use of hierarchical reinforcement learning methods, as explained by Barto and Mahadevan [16].

One of the main developments in reinforcement learning was the introduction of the temporal difference (TD) methods, which are a class of incremental learning procedures specialized for prediction problems [17]. They are driven by the error or difference between temporally successive predictions of the states. Learning occurs whenever there is a change in the prediction over time.

The simplest TD method known as TD(0) updates the estimate of the value function, after going from state s to state s' and receiving the reward r, using the following rule:

$$V_{t+1}(s) \longleftarrow V_t(s) + \alpha\left[r + \gamma V_t\left(s'\right) - V_t(s)\right]. \quad (7)$$

Based on this rule, several popular RL methods such as SARSA [8] and the actor-critic [8] methods were developed. SARSA is an on-policy temporal difference control algorithm which continually estimates the state-action value function Q^π for the behavior policy π, and at the same time changes π toward greediness with respect to Q^π. If the policy is such that each action is executed infinitely often in every state, every state is visited infinitely often, and it is greedy with respect to the current action-value function in the limit, and then by decaying α, the algorithm converges to Q^* [18].

Actor-critic methods are TD methods that have a separate memory structure to explicitly represent the policy independent of the value function. The policy structure is known as the actor, because it is used to select actions, and the estimated value function is known as the critic, because it criticizes the actions made by the actor. Learning is always on-policy: the critic must learn about and critique whatever policy is currently being followed by the actor. The critique takes the form of a TD error. This scalar is the only output of the critic and guides the learning occurring in both actor and critic as illustrated in Figure 1.

Typically, the critic is a state-value function. After each action selection, the critic evaluates the new state to determine whether things have gone better or worse than expected. That evaluation is the TD error as

$$\delta_t = r_{t+1} + \gamma V\left(s_{t+1}\right) - V\left(s_t\right), \quad (8)$$

where V is the current value function implemented by the critic. This TD error can be used to evaluate the action just selected, the action a_t taken in state s_t. If the TD error is positive, it suggests that the tendency to select a_t should be strengthened for the future, whereas if the TD error is negative it suggests that the tendency should be weakened. Under batch updating, TD converges deterministically to a single answer independent of the step-size parameter α, when α is sufficiently small [8].

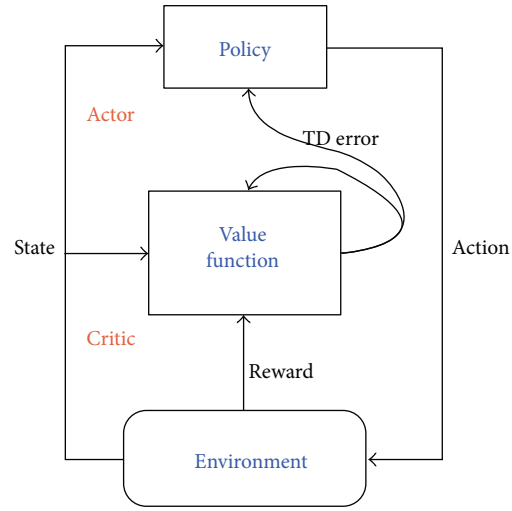

Figure 1: The actor-critic architecture.

Actor-critic methods have two significant apparent advantages.

(i) They require minimal computation in order to select actions. Consider a case where there are an infinite number of possible actions, for example, a continuous-valued action. Any method learning just action values must search through this infinite set in order to pick an action. If the policy is explicitly stored, then this extensive computation may not be needed for each action selection.

(ii) They can learn an explicitly stochastic policy; that is, they can learn the optimal probabilities of selecting various actions.

3. Architecture and Algorithm

The A^{2}C architecture, illustrated in Figure 2, is the result of our search for a robust reinforcement learning architecture specifically designed to tackle intelligent control problems with continuous state and action spaces, where the computation of the final policy must be performed very fast.

The key part of this architecture design is the idea of distributed policy learning, which allows the agent to learn and store the information of the final policy using only two modules, instead of only one. Sometimes trying to use one structure to store the policy of a complex reinforcement learning problem is simply not enough. We need to distribute the amount of information required to store the policy in more than one module. Therefore, the proposed architecture increases the number of modules used to store the final policy

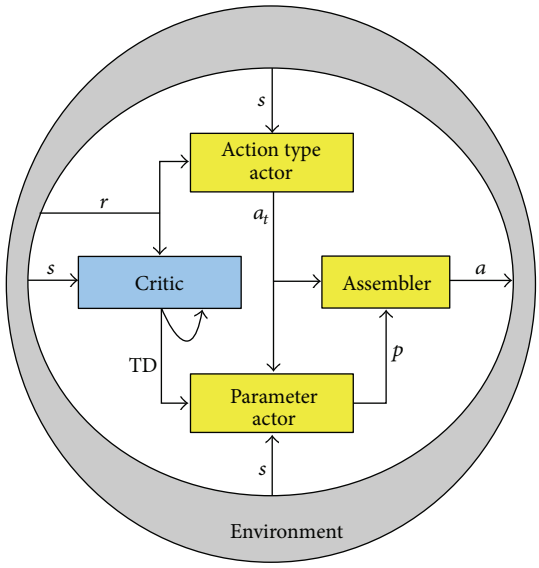

FIGURE 2: The A^{2}C architecture.

from one to two: the *action type actor* and the *parameter actor*. During the learning phase, a third module known as the assembler is used to assemble one action type and one vector of parameters into one executable action. The functions of each of the three modules are as follows.

(i) Action type actor learns from the scalar reward r which action type a_t is the best one to be executed by the agent in the next time stage.

(ii) Parameter actor learns from the temporal difference error TD which parameter vector p is the best for the action type a_t provided by the action type actor.

(iii) Assembler takes the action type a_t and the parameter vector p and assembles the action a to be executed by the agent.

The action type actor implements SARSA learning [8]. This module focuses on learning the best action type at each moment. Given that the number of actions is finite and small, the learning process in this module is fast and robust, even when the states are expressed as continuous vectors.

The parameter actor learns from the temporal difference error computed after each execution of an action. At each time, the parameter action suggests what it believes is the best parameter for the action type chosen by the action type actor. Then, it observes the TD error generated after applying the action formed by the action type and the parameters. If the TD error is greater than zero, it means that the action type and the parameters vector selected are good, and its selection in the future must be reinforced. In this algorithm reinforcing the use of a specific parameters vector means that we should train the function approximators with the supervised training example (s, p), where s is the current state and p is the vector of parameters.

If the TD error is zero or less than zero, it means that one out of three possibilities is happening: (1) the action type selected is incorrect and the parameter vector is correct; (2) the action type selected is correct and the parameter vector is incorrect; (3) both the action type and the parameter vector are incorrect. Only in the first case we should reinforce the selection of such parameters vector; however, it is impossible to determine in which of these three cases we have fallen. Therefore, we simply jump to the next state without experimenting any learning within the parameter actor.

Note that the parameter actor is responsible only for the selection of the parameters; however, it is evaluated taking into account both the action type and the parameters. Under this condition, it is only possible to guarantee the correct learning of the parameters if we can guarantee that the action type actor will eventually learn the correct action types; otherwise, the parameter actor will fail to learn the right parameters.

As in the standard actor-critic architecture, we also employed the typical critic module, which learns from the temporal difference error TD the value function $V(s)$ used to evaluate the quality of each state s. This evaluation is used by the critic itself to improve its estimation of $V(s)$ and it is also used by the parameter actor to improve its estimation of the best parameters for the action types.

To implement our architecture, a number of function approximators are required. The number of function approximators is determined by the number of action types and the number of parameters required by each action type. Figure 3 illustrates the function approximators used in our experimental work. We employed 4 function approximators for the action type actor, 7 for the parameter actor, and 1 for the critic.

The algorithm we propose to implement our architecture is the SARSA A^{2}C.

(1) Initialize the *action type actor*, the *parameter actor,* and the *critic*. This step refers to randomly choosing the initial values of all the parameters used by all the function approximators.

(2) From current state s, select the best action type a_t and parameter vector $\vec{p}_{a_t}$. To select the best action type, we simply evaluate all the function approximators used by the action type actor, with the current state s, and we pick the action type whose function approximator gives the greatest evaluation. Once we have selected the best action type for the current state s, we evaluate the function approximators assigned to that action type, to get $\vec{p}_{a_t}$.

(3) Assembly action a with action type a_t and parameter vector $\vec{p}_{a_t}$. This step is a plain call to the code function used internally by our agent to get ready to execute the chosen action.

(4) For each training episode, use learning rate α and discount factor γ to do the following.

(a) Execute action a and observe next state s' and the scalar reward r.

(b) Compute the TD error as $\varepsilon = [r + \gamma\widehat{V}(s')] - \widehat{V}(s)$, where $\widehat{V}(s)$ is the state value function stored by the critic.

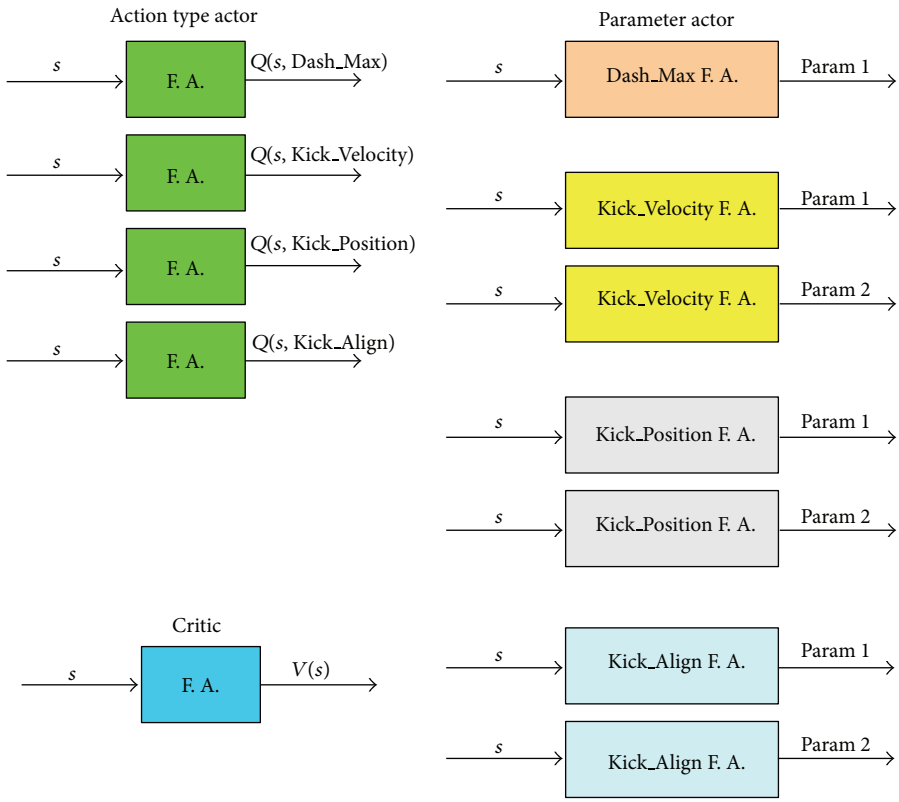

Figure 3: 12 function approximators used to implement the action type actor, the parameter actor, and the critic.

(c) Update the critic using the TD error with $\widehat{V}(s) \leftarrow \widehat{V}(s) + \alpha\varepsilon$.

(d) If $(\varepsilon > 0)$ then reinforce the use of $\vec{p}_{a_t}$ by retraining the function approximator of action type a_t with the example $(s, \vec{p}_{a_t})$.

(e) From the next state s', compute the next action type a'_t and parameter vector $\vec{p}_{a'_t}$.

(f) Assemble the next action a' with action type a'_t and parameter vector $\vec{p}_{a'_t}$.

(g) Update the action type actor with $Q(s, a_t) \leftarrow Q(s, a_t) + \alpha[r + \gamma Q(s', a'_t) - Q(s, a_t)]$, where $Q(s, a_t)$ is the state-action value function implemented as a function approximator.

(h) Update the current state with $s \leftarrow s'$ and the current action with $a \leftarrow a'$.

4. Experimental Results

In the RoboCup simulation league, one of the most difficult skills that the robots can perform is dribbling. Dribbling can be defined as the skill that allows a player to run on the field while keeping the ball always within its kickable margin, as illustrated in Figure 4. In order to accomplish this skill, the player must alternate *dash* and *kick* actions.

There are three factors that make this skill a difficult one to accomplish.

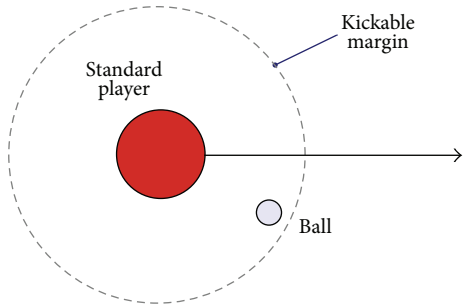

Figure 4: The dribbling problem.

(1) The simulator adds noise to the movement of objects and to the parameters of commands.

(2) Since the ball must remain close to the robot without collisioning with it, and at the same time it must be kept in the kick range, the margin for error is small.

(3) The most challenging factor is the use of heterogeneous players during competitions. Using heterogeneous players means that for each game the simulator generates seven different player types at startup, and the eleven players of each team are selected from this set of seven types. Given that each player type has different "physical" capacities, an optimal policy learned with one type of player is simply suboptimal

when followed by another player of different type. In theory, the number of player types is infinite.

Due to these three reasons, a good performance in the dribbling skill is very difficult to obtain. Up to date, even the best teams perform only a reduced number of dribbling sequences during a game. Most of the time the ball is simply passed from one player to another.

4.1. Desired Characteristics of the Solution. The final policy we are looking for must fulfill the following conditions if it is to be used during competitions.

(i) The final policy, seen as a function of the current state, must be easy to evaluate. During competitions, many decisions must be taken by the team, such as computing positions, velocities, and accelerations, and deciding what kicks and dashes must be performed to keep the possession of the ball or to recover it. All these computations must be done as fast as possible so the state does not change much before the actions are executed; otherwise, the results are suboptimal or completely useless in the best case. In the worst case, the adversary team might score many goals against us. Dribbling is only a minor part of these bunch of computations and we must assure that its execution is as fast as possible.

(ii) It must have a high performance with heterogeneous players. Solving the dribbling problem with standard players could be achieved easily. However, getting a competitive performance with the heterogeneous players is a different story.

(iii) The generation of such policy must be done with few manual adjustments. The RoboCup Simulator is being continuously improved, if the learning algorithm depends strongly on some parameters of the players or the server, and those parameters are removed in the future due to major changes in the simulator software, then major changes will be required in the learning algorithm to keep it useful, especially if we require several manual adjustments.

(iv) Performance at least around 20 meters with high reliability is required. This is a self-imposed condition by our team based on the experience accumulated during several years of experimentation and competitions. We have seen that after 20 meters the policies that push the players to run faster can be successful with fast players, but terrible with the slow ones who tend to lose the ball very often.

4.2. Experiments. The state is seen by the player as a parameter vector which consists of the following 10 variables: (1) player decay, (2) dash power rate, (3) kickable margin, (4) kick rand, (5) ball position x—player position x, (6) ball position y—player position y, (7) ball velocity x—player velocity x, (8) ball velocity y—player velocity y, (9) player velocity x, and (10) player velocity y. The first 4 variables are some of the parameters that define a type of player, and for this problem, they were the most useful during our experimentation. The other 6 variables are needed to specify the current physical state of the ball and the player.

In terms of the reward function, the following is applied. If the ball is in the way of the player, the player receives only a punishment of −10. If the first case is not true, we check if the ball is in the border of the kickable margin which means that the player is about to lose the ball, and in that case the player receives only a punishment of −10. If the first and second cases are not true and we check the third case. If the ball is very close, to the player, there is the possibility of collisioning, and in that case the player receives only a punishment of −10. If none of these three cases is true, then the player receives a positive reward r = player position x + ball position x + [50 ∗ (player velocity x + ball velocity x)], where 50 is a weight parameter chosen experimentally.

The results presented next were obtained under the following experimental settings.

(i) Both the player and the ball are initially in movement.

(ii) The player is placed at the center of the field with a random velocity vector v_p and the ball is placed in any position that falls within the kick range of the player with a random velocity vector v_b.

(iii) The player has 4 actions: Dash Max, Kick Velocity, Kick Position, and Kick Align.

(iv) At each training episode we let the player try 35 actions at most.

(v) If the ball goes out of the kickable margin of the player, the current training episode is finished.

(vi) During training and testing, a new set of players is generated every 7 episodes. In this way we basically train with a different player each episode. By doing so, we pretend to improve the generalization of our learned model.

We used 3 different function approximators to implement the actors and the critic.

(i) Multilayer perceptrons.

(ii) Multilayer perceptrons with one layer of radial basis functions [19].

(iii) Arrays of radial basis functions.

In Figure 5, we see the performance of the SARSA A^2C algorithm using only multilayer perceptrons with 3 different numbers of hidden units: 2, 10, and 20. After 50,000 training episodes, all three configurations of the networks managed to learn a policy that allows a performance of 15 meters at least, being the configuration with 2 hidden units slightly better with a performance of 16 meters. From the graph, we can also see that as we increase the number of hidden units, the learning curve becomes less smooth. Based on these results, the configuration of the multilayer perceptron with 2 hidden units was the most reliable.

We modified the multilayer perceptrons networks adding a layer of radial basis functions between the input and the

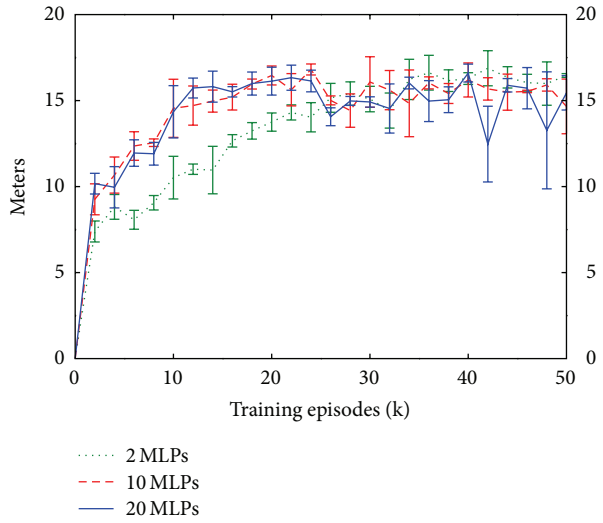

Figure 5: Learning curves of the SARSA A^{2}C algorithm using different numbers of multilayer perceptrons.

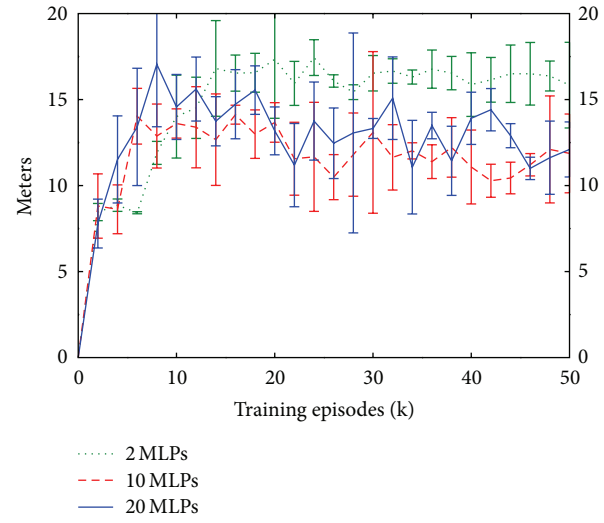

Figure 6: Learning curves of the SARSA A^{2}C algorithm using different numbers of multilayer perceptrons and 10 radial basis functions for each multilayer perceptron.

hidden layer that works as a filter layer. Each input unit was connected to 10 gaussian radial basis functions. The idea was to try to localize the learning. The resulting learning curves are shown in Figure 6. We can see that in the case of using 10 and 20 hidden units, the performance obtained is not more than 12 meters, being worse than that obtained using only the multilayer perceptrons without radial basis functions. It is also clear that the learning curves become less smooth. However, we can notice that the configuration of 2 hidden units keeps its performance of 16 meters and it is able to reach this performance in less training time than the simple multilayer perceptron network. The network with radial basis function using 2 hidden units gets its best performance in 15,000 training episodes; meanwhile, the network without radial basis functions needs at least 40,000 episodes.

For the third implementation of the SARSA A^{2}C algorithm, we used n arrays of gaussian radial basis functions, one array per input. In Figure 7, we can see the learning curves obtained with 10, 30, and 50 radial basis functions. The final performance of 25 meters is much better than that obtained with the previous function approximators. This performance is obtained in less than 10,000 training episodes, which is very fast compared to the results shown in the two previous graphs. Also, we can notice that as we increase the number of radial basis functions, the learning curves need a little more time to go up. All three learning curves are smooth and with small variance, which means that we are finding very reliable policies.

Figure 8 shows the best learning curves found together with each function approximator. It is clear that the best performance in terms of training time and reliability is obtained using only radial basis functions.

4.3. Final Policy Performance. Table 1 shows a comparison of the two best reinforcement learning methods found so far to solve the dribbling problem.

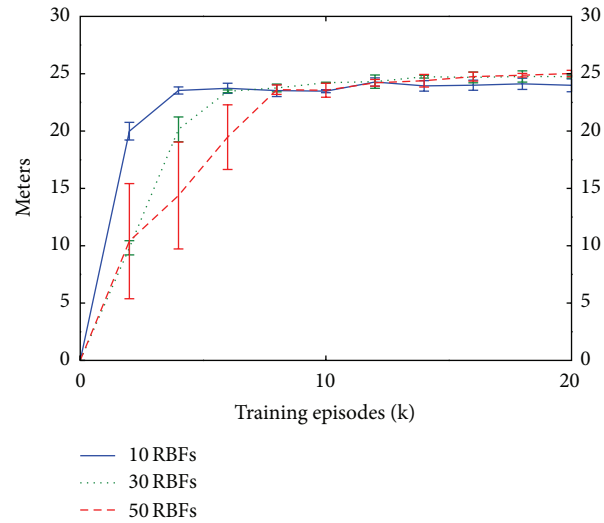

Figure 7: Learning curves of the SARSA A^{2}C algorithm using different numbers of radial basis functions.

Table 1: Comparison of the best policies for the dribbling problem.

	SARSA A^{2}C	$Q(\lambda)$-learning
Algorithm type	Actor-Critic	$Q(\lambda)$-learning
Function approx.	RBFs	CMACs
States	Continuous	Continuous
Actions	Continuous	Discrete
Total learning time	10 minutes	24 hours 30 minutes
Average distance	25.45 meters	29.21 meters
Maximum distance	36.23 meters	39.0 meters

From Table 1, we can see the following. Both methods employed linear function approximators. The states are handled continuously by both methods. The SARSA A^{2}C

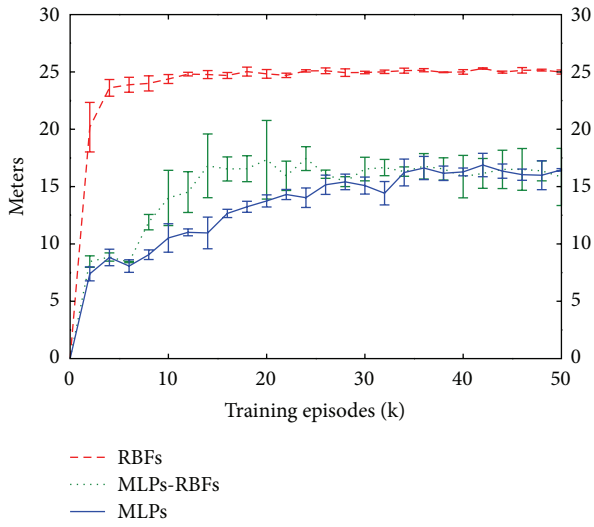

Figure 8: Learning curves of the SARSA A^{2}C algorithm using three different function approximators: radial basis functions, multilayer perceptrons, and multilayer perceptrons with a layer of radial basis functions.

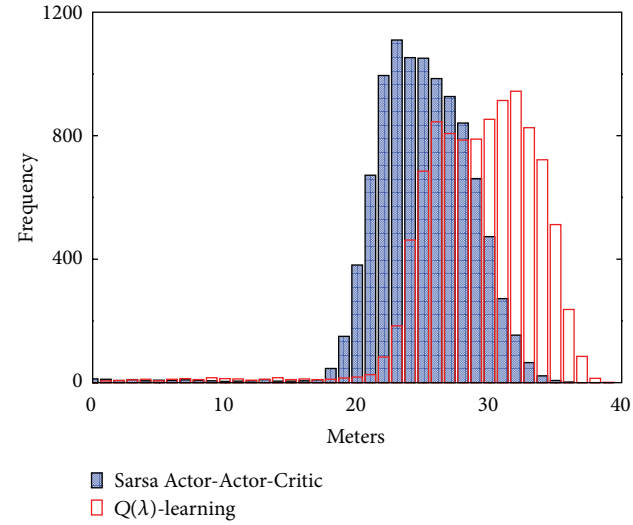

Figure 9: Comparison of the reliability of the policies found with the SARSA Actor-Actor-Critic algorithm and the $Q(\lambda)$-learning algorithm.

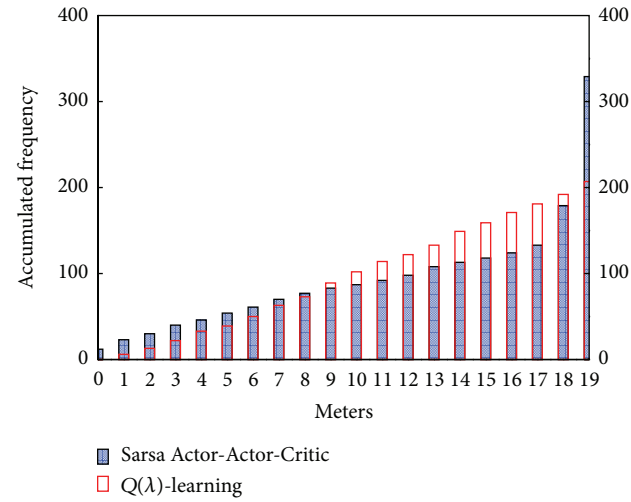

Figure 10: Accumulated frequency: Comparison of the reliability of the policies found with the SARSA Actor-Actor-Critic algorithm and the $Q(\lambda)$-learning algorithm.

method handles the actions continuously; meanwhile, in $Q(\lambda)$-learning, a finite set of actions with continuous parameters is employed. However, it is important to consider that in $Q(\lambda)$-learning, a finite set of actions is generated randomly at the beginning of the training process. Then, an algorithm is used during 24 hours to discover and delete the less useful actions. After this reduction process, the final set of actions contains about 30 action with continuous parameters. At this point, the policy learning process is started with the reduced set of actions. The policy learning process takes around 30 minutes.

In terms of average distance and maximum distance, $Q(\lambda)$-learning is superior to SARSA A^{2}C. However, we must realize that a more important factor to consider in the final policy performance is the reliability. By reliability we mean how much we can trust our policy during a game. In other words, we are interested in knowing how often the player will manage to run at least a given number of meters before it loses the ball due to a wrong action selection. Therefore, in order to study the reliability of the policies found with the SARSA A^{2}C algorithm in comparison with $Q(\lambda)$-learning, we performed the following experiment. Using one of the best policies obtained, we let 10,000 different players run with the ball from the center of the soccer field. We wrote down how many meters each player managed to run before losing the ball. Then, using this statistics we plotted the graphs in Figures 9 and 10.

From Figure 9, we can see that with $Q(\lambda)$-learning policy most of the players managed to run at least 20 meters; meanwhile, with the SARSA A^{2}C policy they run at least 18 meters. Now, if we make a closeup of this graph and focus on the first 20 meters as an accumulated frequency histogram, the image we see is the one in Figure 10. In this figure, something interesting is evident. With $Q(\lambda)$-learning policy, the players lose the ball more often than with the SARSA A^{2}C policy. Notice that a perfect policy will not allow the player to lose the ball before running at least 20 meters. And if we were working with a perfect policy, all the frequency columns from meter 0 to 19 would be of size 0, meaning that the players always run at least 20 meters. We can see that from meter 0 to 8, $Q(\lambda)$-learning policy is more reliable, and between meters 9 and 18, the SARSA A^{2}C policy is more reliable.

In competitions, the time required to compute each of the decisions taken by the players must be reduced as much as possible. Once the final policy has been learned, during competition we need to compute the following.

(1) We need to compute he type of action to be executed and the parameters of such action. To select

the action, we need to compute the Q values corresponding to each type of action. The cost of this computation is $4(c_1 + c_2 + \cdots + c_n)$, where n is the number of radial basis functions and c_i is the cost of computing the output of the gaussian function r_i and multiplying this output by the corresponding weight w_i. We multiply the cost by 4, since we have to perform these operations for each type of action.

(2) Then, we must find the maximum of such 4 Q values. We need to perform 3 comparisons to find the greatest Q value. Therefore, the cost is $3c$, where c is the cost of each logic comparison.

(3) Finally, we need to compute the parameters of the chosen type of action. In the worst case we would need to compute the values of two parameters. The cost of this computation is $2(c_1 + c_2 + \cdots + c_n)$, where n is the number of radial basis functions and c_i is the cost of computing the output of the gaussian function r_i and multiplying this output by the corresponding weight w_i. We multiply the cost by 2, since we have to perform these operations for each parameter of the selected action.

5. Conclusions and Future Work

We have introduced the Actor-Actor-Critic architecture and explained its benefits in problems with continuous state and action spaces. Based on such architecture, we presented the SARSA Actor-Actor-Critic algorithm or SARSA A^2C for short. Such an algorithm generate reliable policies for the dribbling problem in short training times. We have also presented and described the experimental results obtained when applying the SARSA A^2C algorithm to the most difficult test scenario considered in this research. Finally, we have made a comparison of our method and the current best method to solve the dribbling problem.

We have shown through experimentation that an algorithm with two actors and one critic, where the action type actor is trained using Q values through SARSA learning and the parameter actor and the critic are trained using the standard temporal difference error, is capable of learning policies with good performance and the learning process is fast and completely reliable.

We have also seen that the array of radial basis functions works better than the multilayer perceptron and the multilayer perceptron with one layer of radial basis functions.

The results obtained for a specially challenging task in the RoboCup framework suggest that the implementation of our A^2C architecture may also lead to outstanding performance in other RL problems with continuous state and action spaces.

Among our future work involving the A^2C architecture, we consider the solution of other problems with continuous state and action spaces, the development of other algorithms based on it, and the study of ways of implementing hierarchical A^2C methods.

Acknowledgments

The author would like to express his gratitude to Professor Hans-Dieter Burkhard for his advice and support, and Ralf Berger for his experimental work with the $Q(\lambda)$ algorithm.

References

[1] RoboCup, "The goals of robocup.robocup federation on," 2001, http://www.robocup.org/about-robocup/objective/.

[2] M. Gollin, *Implementation einer bibliothek für reinforcement learning und anwendung in der robocup simulationsliga [M.S. thesis]*, Humboldt University of Berlin, Berlin, Germany, 2005.

[3] M. Riedmiller and T. Gabel, "On experiences in a complex and competitive gaming domain: reinforcement learning meets RoboCup," in *Proceedings of the 3rd IEEE Symposium on Computational Intelligence and Games (CIG '07)*, pp. 17–23, April 2007.

[4] A. Cherubini, F. Giannone, L. Iocchi, M. Lombardo, and G. Oriolo, "Policy gradient learning for a humanoid soccer robot," *Robotics and Autonomous Systems*, vol. 57, no. 8, pp. 808–818, 2009.

[5] J. Leng and C. P. Lim, "Reinforcement learning of competitive and cooperative skills in soccer agents," *Applied Soft Computing Journal*, vol. 11, no. 1, pp. 1353–1362, 2011.

[6] P. Stone, G. Kuhlmann, M. E. Taylor, and Y. Liu, "Keepaway soccer: from machine learning testbed to benchmark," in *RoboCup-2005: Robot Soccer World Cup IX*, I. Noda, A. Jacoff, A. Bredenfeld, and Y. Takahashi, Eds., pp. 93–105, Springer, Berlin, Germany, 2006.

[7] H. Montazeri, S. Moradi, and R. Safabakhsh, "Continuous state/action reinforcement learning: a growing self-organizing map approach," *Neurocomputing*, vol. 74, no. 7, pp. 1069–1082, 2011.

[8] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, The MIT Press, Cambridge, Mass, USA, 1998.

[9] R. H. Crites and A. G. Barto, "An actor/critic algorithm that is equivalent to q-learning," in *Advances in Neural Information Processing Systems*, 1995.

[10] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: a survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

[11] V. Heidrich-Meisner, M. Lauer, C. Igel, and M. Riedmiller, "Reinforcement learning in a nutshell," in *Proceedings of the 15th European Symposium on Artificial Neural Networks*, 2007.

[12] R. A. Howard, *Dynamic Programming and Markov Processes*, The MIT Press, Cambridge, Mass, USA, 1960.

[13] S. Ross, *Introduction to Stochastic Dynamic Programming*, Academic Press, New York, NY, USA, 1983.

[14] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-Dynamic Programming*, Athena Scientific, 1996.

[15] M. Puterman, *Markov Decision Processes*, John Wiley & Sons, New York, NY, USA, 1994.

[16] A. G. Barto and S. Mahadevan, "Recent advances in hierarchical reinforcement learning," *Discrete Event Dynamic Systems*, vol. 13, no. 1-2, pp. 41–77, 2003.

[17] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine Learning*, vol. 3, no. 1, pp. 9–44, 1988.

[18] S. Singh, T. Jaakkola, M. L. Littman, and C. Szepesvári, "Convergence results for single-step on-policy reinforcement-learning algorithms," *Machine Learning*, vol. 38, no. 3, pp. 287–308, 2000.

[19] V. Uc-Cetina, "Multilayer perceptrons with radial basis functions as value functions in reinforcement learning," in *Proceedings of the European Symposium on Artificial Neural Networks*, 2008.