Available online at www.sciencedirect.com
ScienceDirect
Artificial Intelligence 172 (2008) 454–482
Artificial Intelligence
www.elsevier.com/locate/artint

Restricted gradient-descent algorithm for value-function approximation in reinforcement learning

André da Motta Salles Barreto [a,*], Charles W. Anderson [b]

[a] *Programa de Engenharia Civil/COPPE, Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, Brazil*
[b] *Department of Computer Science, Colorado State University, Fort Collins, CO 80523, USA*

Received 22 May 2006; received in revised form 22 August 2007; accepted 23 August 2007

Available online 6 September 2007

Abstract

This work presents the restricted gradient-descent (RGD) algorithm, a training method for local radial-basis function networks specifically developed to be used in the context of reinforcement learning. The RGD algorithm can be seen as a way to extract relevant features from the state space to feed a linear model computing an approximation of the value function. Its basic idea is to restrict the way the standard gradient-descent algorithm changes the hidden units of the approximator, which results in conservative modifications that make the learning process less prone to divergence. The algorithm is also able to configure the topology of the network, an important characteristic in the context of reinforcement learning, where the changing policy may result in different requirements on the approximator structure. Computational experiments are presented showing that the RGD algorithm consistently generates better value-function approximations than the standard gradient-descent method, and that the latter is more susceptible to divergence. In the pole-balancing and Acrobot tasks, RGD combined with SARSA presents competitive results with other methods found in the literature, including evolutionary and recent reinforcement-learning algorithms.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Reinforcement learning; Neuro-dynamic programming; Value-function approximation; Radial-basis-function networks

1. Introduction

Sutton and Barto [77] and Kaelbling et al. [37] describe reinforcement learning as a class of problems, rather than as a set of techniques. In this paradigm, an agent must learn how to act through direct interaction with an environment. The only information available to the agent is a reinforcement signal providing evaluative feedback for the decisions made.

This rather informal definition is sufficient to explain the increasing interest in the field from the artificial intelligence and machine learning communities. First of all, this framework is a crude but appealing model of what actually happens in nature, and the analogy with animal behavior is almost irresistible [64,76]. From an engineering perspective, the reinforcement-learning paradigm is also tempting, since it transfers to the learning system the burden of figuring out *how* to accomplish a task. This way, instead of providing a set of examples with the desired behavior, as

* Corresponding author.
E-mail addresses: andremsb@lncc.br (A. da Motta Salles Barreto), anderson@cs.colostate.edu (C.W. Anderson).

0004-3702/$ – see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.artint.2007.08.001

in supervised learning, the designer is left with the much simpler task of representing a problem in terms of reward and punishment signals only. If, for example, one wanted to have a mobile robot travel from one point to another, it could be done by simply giving the robot a reward when the goal was reached.

In order for the above scenario to be true, however, several obstacles other than those in the robot's trajectory must be overcome. One of the major difficulties arises from the combination of reinforcement learning and function approximation. If one wants to solve real-world reinforcement learning tasks—where the number of possible states of the system is usually too large to allow an exhaustive exploration—it is necessary to provide the agent with the ability to generalize. Nevertheless, as is now well known (though not completely understood), the combination of reinforcement learning algorithms with function approximators can easily become unstable, and finding a feasible way to merge both paradigms is today an active area of research in machine learning.

This paper presents an attempt in this direction, namely a training method for local radial-basis function networks especially designed to be used in the context of reinforcement learning. The restricted gradient-descent (RGD) algorithm is essentially a modified version of the standard gradient-descent method, and as such it inherits both its qualities and drawbacks. However, the restrictions imposed by RGD on the application of gradient-descent's delta rule make it less prone to divergence. In the RGD algorithm the center of the radial functions are always moved toward the current state, which tends to confine them to the convex-hull formed by the training data. Also, the widths of the radial-basis functions are only allowed to shrink, and thus they can not possibly diverge to infinity. Obviously, these restrictions on the delta rule limit the trajectory of the approximator's parameter vector in the parameter space. This lost of flexibility is compensated by the on-line allocation of new hidden units, which results in a monotonically increasing of the approximation granularity.

This work is organized as follows. Section 2 presents a brief review of reinforcement learning, focusing on methods that rely on the concept of a value function to address this problem. As mentioned, the stability of such methods can be affected by the use of function approximators. This issue is discussed in more detail in Section 3. One way to alleviate the instability caused by the use of approximators is to adopt linear models operating on features extracted from the original state space. Section 4 discusses the concept of "features" used in this paper, as well as a way to select them using local radial-basis functions. Section 5 presents the RGD algorithm, which can be seen as a strategy to extract relevant features from the state space. This algorithm is applied in Section 6 to a series of computational experiments. Particularly, the performance of RGD is compared to that of the standard gradient-descent method and to several traditional reinforcement-learning algorithms using other techniques to approximate the value function. Section 7 presents an overall analysis of the experiments and a hypothesis explaining the stable behavior of the restricted gradient-descent algorithm. Finally, in Section 8 the main conclusions about this research are presented, and some possible directions for future work are discussed.

2. Reinforcement learning

The goal of the agent in reinforcement learning is to find a policy π—a mapping from states to actions—that maximizes the expected return. The return R_t is the total reward the agent receives in the long run from time t on:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{i=0}^{\infty} \gamma^i r_{t+i+1},$$

where $\gamma \in [0, 1]$ is the discount factor. This parameter determines the relative importance of the individual rewards, depending on how far in the future they are.

The rewards $r \in \Re$ are given by the environment to the agent each time it performs an action. Usually, this interaction happens in discrete steps: at each time step t, the agent selects an action $a \in A(s_t)$ as a function of the current state $s_t \in S$. The sets S and $A(s_t)$ are part of the environment and represent the possible states of the system and the available actions in each state s_t. As a response to the action a, the agent receives from the environment a reward r_{t+1} and a new state s_{t+1}. This loop is repeated indefinitely or until the agent reaches a terminal state. The interaction between agent and environment can be formalized as a Markov decision process [10,14,54].

One way to address the reinforcement-learning problem is to search for a good solution in the policy space directly. This could be done, for example, by parametrizing π and then have an evolutionary algorithm search for good policies [35,88]. Another approach is to compute an approximation of the gradient of the average reward with respect to the parametrized policy, which can be used to perform gradient ascent [7,8,73].

Another way to deal with the reinforcement learning problem is to use methods derived from dynamic programming [9,14,54]. One advantage of this approach is the fact that dynamic programming has been studied for a long time, and is now supported by a strong and well understood theoretical basis. Central to the dynamic programming approach is the concept of a *value function*. The action-value function of a given policy π associates each state–action pair (s, a) with the expected return when performing action a in state s and following π thereafter:

$$Q^{\pi}(s, a) \equiv E_{\pi}\{R_t \mid s_t = s, a_t = a\},$$

where $E_{\pi}\{\}$ denotes the expected value when following policy π. Since the notation above is widely adopted, the action-value function is usually referred to as the Q-function. Once the Q-function of a particular policy π_k is known, we can derive a new policy, π_{k+1}, which is greedy with respect to $Q^{\pi_k}(s, a)$:

$$\pi_{k+1}(s) = \arg\max_{a \in A(s)} Q^{\pi_k}(s, a). \tag{1}$$

The policy π_{k+1} is guaranteed to be at least as good as (if not better than) the policy π_k. This is the fundamental idea of the reinforcement learning algorithms based on dynamic programming: given an initial arbitrary policy π_0, compute its value function $Q^{\pi_0}(s, a)$ and then generate a better policy π_1 which is greedy with respect to this function. The next step is to compute $Q^{\pi_1}(s, a)$, use it to generate a new policy π_2, and so on. Under certain assumptions, the successive alternation of these two steps—*policy evaluation* and *policy improvement*—can be shown to converge to the optimal policy π^*, which maximizes the expected return on every state.

Of course, the above process can be executed in different levels of granularity. It is not necessary, for example, to have an exact Q-function $Q^{\pi_k}(s, a)$ to perform the policy improvement step. One can compute a rough approximation of this function and then use it to generate a new policy π_{k+1}. The improvement step itself can be performed in different levels. Eq. (1) could be used to update the policy π_k on a single state, a set of them, or on the entire state space S. The exact way the policy evaluation and policy improvement steps are performed defines the different reinforcement learning algorithms.

2.1. Computing the value function

A complete control problem can be broken into two stages: policy evaluation and policy improvement. The policy improvement step is usually easy to compute. This is especially true if $A(s_t)$ is finite and small for every s_t, in which case it is reduced to the computation of the "max" operator in (1). Therefore, much of the effort in reinforcement learning research is devoted to the evaluation problem, that is, given a policy π how to compute its value function.

Suppose the state space is a finite set. One way to do policy evaluation is to use sample trajectories to compute the average return associated with each state–action pair [5,45]:

$$Q_k^{\pi}(s, a) = \frac{1}{k} \sum_{i=1}^{k} R_t^i, \quad \text{with } s_t = s \text{ and } a_t = a, \tag{2}$$

where R_t^i are actual returns following the visits to (s, a). It can be shown that the sequence $Q_1^{\pi}, Q_2^{\pi}, \ldots, Q_{\infty}^{\pi}$ asymptotically approaches the true Q-function of π [77]. One drawback of this approach is the fact that it can be naturally applied only to tasks where the interaction between agent and environment can be broken into subsequences or "trials", as in a maze, for example, where a new trial starts every time the agent reaches the goal. We call each subsequence an *episode* [77].

Even if the task can be subdivided into episodes, when using (2) the agent does not learn anything *during* each interaction, but only between them, when the collected data can be used to update Q_k^{π}. It is possible for the agent to learn while interacting with the environment. This is because of a recursive relation between states known as the *Bellman equation* [9]:

$$\begin{aligned} Q^{\pi}(s, a) &= E_{\pi}\left\{r_{t+1} + \gamma Q^{\pi}\left(s_{t+1}, \pi(s_{t+1})\right) \mid s_t = s, a_t = a\right\} \\ &= \sum_{s'} P_{ss'}^{a}\left[R_{ss'}^{a} + \gamma Q^{\pi}\left(s', \pi(s')\right)\right], \end{aligned} \tag{3}$$

where $P_{ss'}^{a}$ is the probability of reaching state s' when in state s and performing action a and $R_{ss'}^{a}$ is the expected reward of this transition.

Eq. (3) is one of the most important results in dynamic programming, and is the core of most value-function based reinforcement-learning methods. First of all, Q^π is the unique solution for its Bellman equation, and thus solving (3) for every state–action pair corresponds to finding the true Q-function of π. Furthermore, the Bellman equation makes it possible to update the Q-values of a state s based on the estimated value of its successors. This is what Sutton and Barto [77] call *bootstrapping*, and is the basic mechanism through which an agent can learn while interacting with its environment.

If the environment's dynamics are completely known—that is, if $P^a_{ss'}$ and $R^a_{ss'}$ are given—the Bellman equations of all state–action pairs can be written as a system of linear equations whose solution is the true value function of π. One way to solve this system is to use iterative methods in which the approximation of Q^π is successively refined based on its past estimates [14,54]:

$$Q^\pi_{k+1}(s,a) = \sum_{s'} P^a_{ss'}\big[R^a_{ss'} + \gamma\, Q^\pi_k\big(s', \pi(s')\big)\big]. \tag{4}$$

The above algorithm is known as *iterative policy evaluation*. It can be applied synchronously, where the old values $Q^\pi_k(s,a)$ are kept during one iteration, or in a Gauss–Seidel style, in which the most recent available values are used to make up the targets for the updates. In both cases, however, a model of the system is required. This restriction can be removed if $P^a_{ss'}$ and $R^a_{ss'}$ are estimated from sample transitions from the environment or a generative model. This is the basic idea of the *temporal-difference* learning methods [74], whose update rule can be written as:

$$Q^\pi_{k+1}(s,a) = Q^\pi_k(s,a) + \alpha\big[r + \gamma\, Q^\pi_k\big(s', \pi(s')\big) - Q^\pi_k(s,a)\big], \tag{5}$$

where r is the reward received in one transition from state s to s' and $\alpha \in [0,1]$ is the learning rate. Temporal-difference (TD) learning is one of the central ideas of reinforcement learning. It makes it possible for the agent to learn while interacting with the environment without knowledge of its dynamics. With the use of *eligibility traces*, the TD method can update more than one Q-value at each iteration. This generates a family of algorithms known as TD(λ) [67].

Eq. (5) can be modified to deal with the complete control problem, that is, besides the evaluation step, also to perform the policy improvement step. This is done by incorporating the "max" operator into the update rule, and the resulting algorithm is called *Q-learning* [83]. If the policy improvement step is applied after each temporal-difference update—that is, if (1) and (5) are applied alternately—one has the SARSA algorithm [57,75].

The algorithms represented by (4) and (5) and their control counterparts are guaranteed to converge to the true value function Q^π if certain assumptions are respected [14,24,36,66]. One of these assumptions is that Q^π_k is stored in a lookup-table, with one entry for each state–action pair. This makes the application of the standard version of these methods to large or continuous state spaces infeasible.

3. Reinforcement learning and function approximation

In real-world reinforcement learning tasks, it is usually not possible to visit every state–action pair (s,a), and thus the agent must be able to *generalize* from its limited experience. In the context of dynamic-programming based algorithms, this means the value function must be represented by a function approximator.

Generalization from examples has been studied in artificial intelligence for many years, and there is now a great body of research on function approximation that can be borrowed from supervised learning. However, the combination of reinforcement learning and function approximators is not a straightforward task, for several reasons:

(1) In reinforcement learning, it is important that learning occurs on-line, while the agent interacts with the environment. This requires methods that are able to deal with incrementally acquired data, instead of a static training set [77,85].
(2) Also, if a bootstrapping method is adopted, the target values to be used as training examples—the right-hand side of (4) and (5), for example—are highly non-stationary. If the policy changes during the learning process, this problem can get even worse, since the distribution from which the training examples are picked may also change over time [55].
(3) As observed by Boyan and Moore [19] and Gordon [32], even if the function approximator is able to approximate the final value function Q^π, it might not be able to represent the intermediary versions Q^π_k of this function.

(4) In the complete control problem, even if a good approximation of the optimal value function Q^* is found, the approximation error may cause the resulting policy π to perform very poorly [13,58,68,89].

Despite all these difficulties, reinforcement learning with function approximators has been successfully applied in various domains. Classical examples in game playing are the Samuel's checker player [60,61] and Tesauro's TD-Gammon, a program based on reinforcement learning which is able to play backgammon near the level of the world's strongest grandmasters [79]. The reinforcement learning paradigm has also been applied to different sequential decision tasks, such as shop-schedule strategies for NASA space shuttle missions [90], elevator dispatch control [23], dynamic channel allocation in cellular telephone networks [65] and regulation of an irrigation network [34]. In robotics, reinforcement learning algorithms combined with function approximation have been used, for example, for navigation control [25,41] and to help robots to walk [12] and to play soccer [72].

The contrast between the theoretical obstacles and the practical successes in applying reinforcement learning with function approximators resulted in great interest in the area. Some of the earlier works presented discouraging results, with simple counterexamples in which the most popular methods would fail dramatically [4,19,32,81]. Later, Sutton [75] showed that some of these counterexamples could be solved when using a linear approximator and online state sampling (that is, sampling the transitions according to the current policy π). Based on this observation, Tsitsiklis and Roy [82] proved that TD(λ) with this configuration converges with probability 1. This result created a strong tendency toward the use of linear models in reinforcement learning, for which several theoretical analyses exist [53,56,63,78]. The strongest results in the current literature are those regarding the convergence of control algorithms with linear function approximators [33,39,48,49]. Among these, kernel-based reinforcement learning [48] and the least-squares policy iteration algorithm (LSPI) [39] deserve special attention, and will be discussed further in the text.

4. Features as local basis functions

As mentioned, recent theoretical and practical results in the reinforcement-learning literature consider the use of linear approximators. One way to represent the Q-function in the case of discrete actions is to have one linear model for each possible action:

$$\tilde{Q}_a^{\pi}(s) = \sum_{i=1}^{m} w_i^a \theta_i(s), \quad \text{for all } a \in A(s) \tag{6}$$

where w_i^a are the linear weights associated with action a and $\theta_i(s)$ are the *m features* representing state s (common to all $\vec{w}^a$).

The concept of feature used here is the same used by Tsitsiklis and Roy [82], and corresponds to what kind of information the agent extracts from the environment. The definition of a suitable feature space is a fundamental step for any reinforcement learning method, and is completely task-dependent. To illustrate this statement, one can compare two well known card games, blackjack and poker. While in the first one the values of the card ranks are enough to derive a good strategy, an agent unable to distinguish the suits on the cards would not perform very well playing poker.

Usually, the identification of which features should be used is not a trivial task. If one has enough knowledge about the domain of interest, it might be a good strategy to handcraft the structure of the linear approximators and to adhere as tightly as possible to the frameworks for which guarantees exist. If this is not the case, it might be desirable to automate this process, since the alternative would be an expensive trial-and-error procedure.

4.1. Local radial-basis functions

One way to extract features from the state space is to use basis functions [81]. In this way, each feature $\theta_i(s)$ is a function mapping S into $\Re$. It has been claimed in the literature that the functions θ_i should be local, that is, they should present a significant activation in only a limited region of the state space [3,69,80]. The main motivation for this is to avoid a phenomenon known as *interference* [26]. Interference happens when the update of one state–action pair changes the Q-values of other pairs, possibly in the wrong direction.

Interference is naturally associated with generalization, and also happens in conventional supervised learning. Nevertheless, in the reinforcement learning paradigm its effects tend to be much more harmful. The reason for this is twofold. First, the combination of interference and bootstrapping can easily become unstable, since the updates are no longer strictly local. The convergence proofs for the algorithms derived from (4) and (5) are based on the fact that these operators are contraction mappings, that is, their successive application results in a sequence converging to a fixed point which is the solution for the Bellman equation [14,36]. When using approximators, however, this asymptotic convergence is lost, since the update of one state–action pair (s, a) can change the Q-values of other pairs (s_i, a_j), with $s_i \neq s$ or $a_j \neq a$. In this case, the approximation errors on the points (s_i, a_j) may increase, and if $\tilde{Q}^{\pi}_{a_j}(s_i)$ are subsequently used as the target values for other updates (as in (5)), the errors are passed on, easily spreading out over the entire domain.

Another source of instability is a consequence of the fact that in on-line reinforcement learning the distribution of the incoming data depends on the current policy. Depending on the dynamics of the system, the agent can remain for some time in a region of the state space which is not representative of the entire domain. In this situation, the learning algorithm may allocate excessive resources of the function approximator to represent that region, possibly "*forgetting*" the previous stored information [85].

One way to alleviate the interference problem is to use a local function approximator. The more independent each basis function is from each other, the less severe this problem is (in the limit, one has one basis function for each state, which corresponds to the lookup-table case) [86]. A class of local functions that have been widely used for approximation is the radial basis functions (RBFs) [52]. The characteristic feature of local RBFs is the fact that their value decreases monotonically with the distance from a central point, called the *center* $\vec{c}$ of the radial function. The *width* σ of the RBF determines how fast its value drops. As first noticed by Broomhead and Lowe [21] and Poggio and Girosi [51], the radial basis function paradigm for approximation can be structured as an artificial neural network. In this case, the RBFs are the activation functions of the network's hidden units. It has been shown that, if there is no limit on the number of radial functions available and their centers and widths are adapted during training, an RBF network is a universal approximator [29].

5. Restricted gradient-descent algorithm

It is possible to use an RBF network to approximate the reinforcement-learning value function. In this case, defining the hidden layer of the network corresponds to determining the structure of the feature space wherein the linear model will operate. This is not a trivial task: the configuration of an RBF-network hidden layer requires the definition of the number of radial functions, their centers and widths.

This section presents the restricted gradient-descent algorithm, a training method for local RBF networks developed to be used in the context of reinforcement learning. This means it can operate on-line, deal with non-stationary data and adapt the network topology according to the complexity of the value function. To achieve this the RGD algorithm relies on two basic mechanisms. The first of them, discussed in Section 5.1, is the strategy adopted to allocate new units for the hidden layer. Section 5.2 presents the second ingredient of RGD, namely the mechanism through which features to be used in the value-function approximation are determined. Finally, in Section 5.3 the complete RGD algorithm is presented, and a pseudo-code for the case in which it is combined with TD learning is given.

The use of local radial functions in reinforcement learning is by no means a new idea. See, for example, [3,44, 48,58,59], just to cite a few. Instead of going over the details of each previous attempt to use RBF networks to approximate the value function, we simply point out the similarities and differences of each approach with respect to our algorithm while RGD's characteristics are presented. For a throughout review, the reader is redirected to Ratitch's PhD thesis [55].

5.1. Dynamic allocation of resources

One decision that has a strong impact on the performance of an RBF network is the number of units in its hidden layer. If there are too few units (or features), the model might not be able to represent the value function with the necessary accuracy to generate a good policy. On the other hand, an excessive number of hidden nodes makes the learning process much slower, besides potentially harming the generalization capability of the network.

It is desirable to be able to adapt the topology of the approximating model according to the complexity of the value function being approximated. This is especially true in the complete control problem, since the value function changes every time a policy-improvement step is performed. When using local RBF networks, it is possible to allocate new hidden units "on-demand", based on the pattern of activity of the network. One can, for example, place new RBFs in the regions of the state space where the approximation error is unusually large [27,50].

The allocation of new units based on the approximation error requires the definition of either a schedule or a threshold to trigger the process [59]. Usually, both values are very domain specific. Another idea is to try to "cover" the relevant areas of the state space equally, assuring that each state visited by the agent results in a minimum activation level of the RBF network. If after the presentation of a state $\vec{s}$ the activation of the network is below a specific threshold τ, a new RBF is added to the model, with its center coincident with $\vec{s}$ [46,55].[1] One advantage of using such a strategy is that the threshold τ can be defined independently of the domain, and can be seen as a way to control the overlap—and thus the interference—between RBFs.

The latter strategy has still another advantage: it can be used to guarantee a regular coverage of the state space even if the granularity of the approximator dynamically increases (that is, if the widths of the RBFs decrease over time). This is an essential characteristic that makes the adaptive process of the RGD algorithm possible, as described in the next section.

5.2. *Defining features based on the approximation error*

As noted by Sutton and Barto [77], in reinforcement learning the structure of the approximator should be related to the complexity of the value function. Different regions in the domain of this function may require different levels of granularity, and ideally the approximator's structure would reflect this requirement. A strong indication of the need for a finer granularity is a large approximation error. However, using this information to determine the network topology is not straightforward, since any error function is discontinuous with respect to the number of RBFs in the network's hidden layer. Since the gradient of the error function with respect to the number of hidden units is not computable, one can try to detect the need for a finer grain indirectly. Particularly, if the conventional gradient-descent method successively reduces the RBFs' widths, new radial functions can be added in the areas of the state space left uncovered.

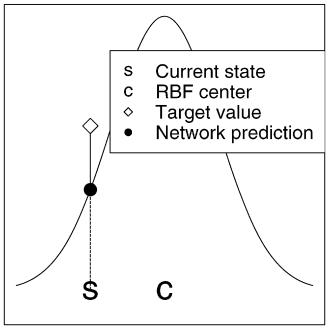
Given a target value $Q^{\pi}(s,a)$ (which can be obtained based on (2), (4) or (5)), suppose the objective is to minimize the following weighted euclidean-norm:

$$\xi_1 = \sum \rho \varepsilon^2 = \sum_{(s,a)} \rho(s,a)\big(Q^{\pi}(s,a) - \tilde{Q}^{\pi}_a(s)\big)^2, \tag{7}$$

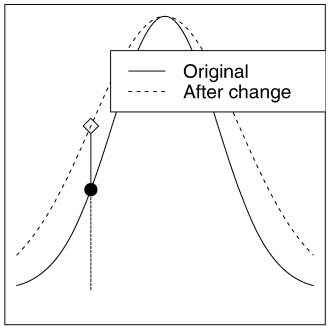
where $\rho(s,a)$ is a distribution weighting the errors of different state–action pairs. In this case, the incremental gradient-descent algorithm will change the parameters of a local RBF in two distinct ways, depending on the relationship between the approximation error ε and the RBF's output weights w_i^a. If $\varepsilon w_i^a > 0$, the width σ_i of the radial basis function will be enlarged and its center $\vec{c}_i$ will be moved toward the current state $\vec{s}$, as shown in Fig. 1. If $\varepsilon w_i^a < 0$, the opposite changes will be performed: the RBF's width will be reduced and its center moved away from the current state (Fig. 2). A more formal presentation with the update equations for the Gaussian function can be found in Appendix A.

The indiscriminate application of the updates described above can easily lead to divergence. In bootstrapping reinforcement-learning the target values used to update $\tilde{Q}^{\pi}(\cdot)$ depend on the current approximation of the Q-function. Notice, for example, how in (4) and (5) the computation of a new estimate $\tilde{Q}^{\pi}_{k+1}(\cdot)$ depends on the previous $\tilde{Q}^{\pi}_{k}(\cdot)$. So, if there is a source of error in the updates, this error will be amplified at each iteration, and the cumulative effect of this will be an unbounded growth of the approximation error. There is some evidence in the literature that one such a source of error is an exaggeration of the Q-values [32,48,56,80]. The underlying idea is that the combination of the "max" operator in (1) with the inevitable imprecision on the estimate $\tilde{Q}^{\pi}_{k}(\cdot)$ is likely to bias $\tilde{Q}^{\pi}_{k+1}(\cdot)$ upward. In the case of local RBFs, this systematic overestimation will result in a never-stopping increase of the functions' widths (see Fig. 1(b)). In fact, this phenomenon was observed in our preliminary experiments with reinforcement learning and RBF networks, and was one of the motivations for the development of the RGD algorithm.

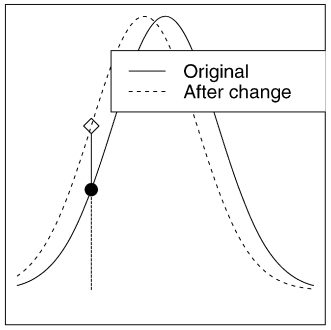
[1] We use '$\vec{s}$' instead of 's' whenever we want to emphasize that the state s is a vector, that is, $S \subset \Re^n$.



(a) Original RBF

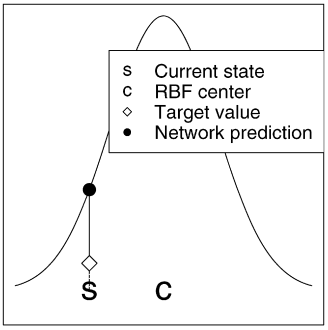


(b) Width is enlarged

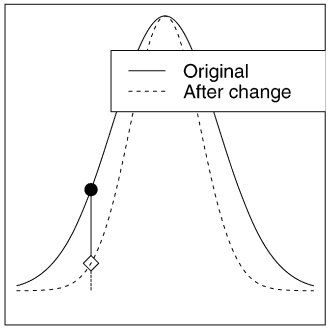


(c) Center is moved toward the current state

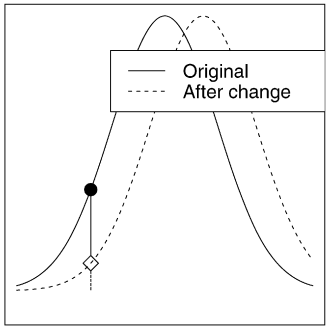
Fig. 1. Changes made when $\varepsilon w_i^a > 0$.



(a) Original RBF



(b) Width is reduced



(c) Center is moved away from the current state

Fig. 2. Changes made when $\varepsilon w_i^a < 0$.

The idea of RGD is to restrict the modifications performed by the standard gradient-descent algorithm to the RBFs' parameters. If the widths of the radial functions are only allowed to "shrink", one has an approximator whose granularity is steadily increasing. Obviously, this process will leave some areas of the state space uncovered, which can be compensated by the allocation of new hidden units. As the reduction of the RBFs' widths is error driven, the resulting procedure is a training algorithm which increases the number of features used by the linear model based indirectly on the approximation error.

5.3. Algorithm

Algorithm 1 shows a pseudo-code of RGD combined with TD(0). Notice that this is essentially the TD-learning algorithm using a linear approximator, except for the block of code detached between dotted lines. The extension for the case where $\lambda > 0$ as well as for the control algorithms is straightforward.

As shown in Algorithm 1, the first step of the RGD method is to check whether the most activated unit is below the threshold τ. If so, a new RBF θ_{m+1} is added to the network, coincident with the current state $\vec{s}$. The width σ_{m+1} of the new RBF determines the overlap between this function and its neighbors, and its definition together with τ allows for some control on the level of interference between the functions. If σ_{m+1} and τ are sufficiently small, the changes performed by the RGD algorithm on the hidden layer may be restricted to the most activated unit θ_i, since the activation level of the others will usually not be very high. The changes on the RBF's parameters depend on εw_i^a, as discussed before. If $\varepsilon w_i^a > 0$, $\vec{c}_i$ is moved towards $\vec{s}$, and its width is left unaltered; if $\varepsilon w_i^a < 0$, the width of the radial function is reduced, and no changes are made to its center.

Reducing the RBFs' widths has two desirable effects. First, it decreases the overlap between functions, which helps to maintain the locality of the model. Also, since the reduction is proportional to the approximation error, the size of the radial functions tends to reflect the shape of the value function, with narrower RBFs in the areas where it is more

loop
  initialize $\vec{s}$
  $a \leftarrow \pi(\vec{s})$ {action given by π for $\vec{s}$}
  repeat
    perform action a and observe next state $\vec{s}_2$ and reward r
    $a_2 \leftarrow \pi(\vec{s}_2)$
    $\varepsilon \leftarrow r + \gamma \tilde{Q}^{\pi}_{a_2}(\vec{s}_2) - \tilde{Q}^{\pi}_{a}(\vec{s})$ {compute TD-error}
    $\vec{w}^a \leftarrow \vec{w}^a + \alpha\varepsilon\vec{\theta}$ {update the linear weights}
    .....
    $\theta_i \leftarrow \max_j \theta_j(\vec{s})$ {find the most activated unit}
    if $\theta_i < \tau$ **then**
      allocate new hidden unit θ_{m+1}
    else
      if $\varepsilon w^a_i > 0$ **then** $\vec{c}_i \leftarrow \vec{c}_i - \beta \frac{\partial \varepsilon^2}{\partial \vec{c}_i}$ {move $\vec{c}_i$ towards $\vec{s}$}
      else $\sigma_i \leftarrow \sigma_i - \beta \frac{\partial \varepsilon^2}{\partial \sigma_i}$ {decrease σ_i}
    end if
    .....
    $\vec{s} \leftarrow \vec{s}_2$
    $a \leftarrow a_2$
  until state $\vec{s}$ is terminal
end loop

Algorithm 1. RGD-TD(0).

complex. If the learning rate β is small, moving the centers of the radial functions always in the direction of the current state tends to confine the RBFs in the convex-hull defined by the training data-points. This might also be a desirable property, since the approximator's resources will be concentrated in the regions of the state space where the data lies [69].

As mentioned before, the widths of the RBFs should be defined in order to guarantee some overlap between the functions. One way to do that is to define σ_{m+1} so that the activation of the new RBF θ_{m+1} will equal τ at the center of the most activated unit θ_i; that is:

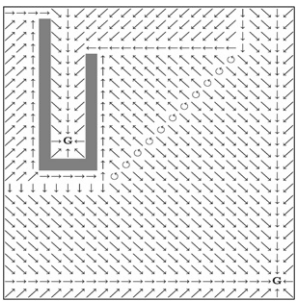
$$\sigma_{m+1} = \sigma \quad \text{such that} \quad \theta_{m+1}(\vec{c}_i; \vec{c}_{m+1}, \sigma) = \tau, \tag{8}$$

where the dependency of θ_{m+1} on the parameters $\vec{c}_{m+1}$ and σ has been emphasized. Notice that when using this strategy the width of a new RBF will depend on the widths of the previous ones: the larger the width of its neighbor θ_i, the larger σ_{m+1} will be, since the threshold τ will be "triggered" further from $\vec{c}_i$. Suppose we start with a single RBF in the network. How, then, to define the width σ_1 of this function? Clearly, an underestimated σ_1 will result in an excessive number of hidden units in the final network generated by RGD. As the RBFs' widths are constantly reducing, however, an overestimated value for this parameter should not harm the approximation. This issue is discussed in more detail in Section 6.1.

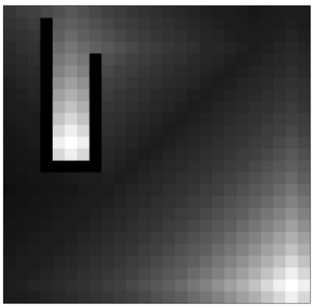
Another question that arises naturally is when to stop shrinking the RBFs: as the minimization is being made in the least-squares sense, the otherwise local minima will generate approximation residuals in both directions, and the alternate signs of εw^a_i will result in a never-stopping decrease of the RBFs' widths. The decision on when to stop the shrinking process is left to the user, who should define an appropriate stop criterion depending on the task. This is where specific domain knowledge should be incorporated to the process such as, for example, an upper bound on the number of hidden units or a threshold for the approximation error. The next section will show computational experiments using different stop criteria.

6. Computational experiments

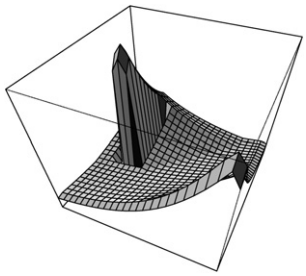
In this section we present an empirical analysis of RGD. Four tasks were used to evaluate different aspects of this algorithm. In Section 6.1 we use a simple maze to study the general behavior of RGD. Particularly, we focus on the



(a) Maze with optimal policy. The arrows represent the sum of the unit-length vectors corresponding to the optimal actions. The symbol ○ is used to denote states for which all actions have the same value-function.



(b) Value function's gray map. The brighter a state, the higher its value function. Walls are represent as black squares.



(c) Value function's landscape. The value of the internal walls are considered to be $\min_{(s,a)} Q^{\pi^*}(s,a) - 1$.

Fig. 3. Maze task.

policy evaluation problem to analyse how the two main parameters of RGD affect its performance. In Section 6.2 the results of RGD on the mountain-car task are contrasted with those of its direct "ancestor", namely the standard gradient-descent method. The latter is applied to both linear and non-linear models of different sizes, which allows us to evaluate the quality of the features selected by RGD. Finally, in Sections 6.3 and 6.4 RGD is compared with other algorithms found in the literature in more challenging control problems. We use the pole-balancing and the Acrobot tasks to check the performance of RGD against evolutionary and recent reinforcement-learning algorithms. The configuration used by RGD in the experiments is discussed in Appendix B.

6.1. Maze

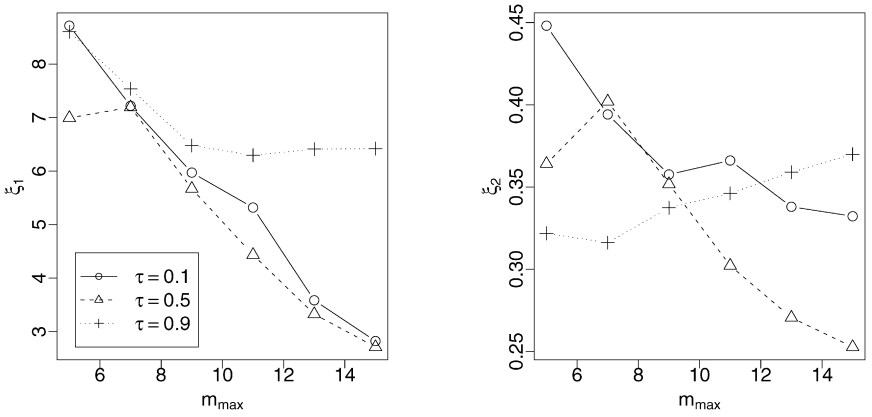
Section 5 presented several statements regarding the expected behavior of the RGD algorithm. In this section a very simple task will be used to verify these assertions. The task is a 25×25 maze presented by Menache et al. [44] and shown in Fig. 3(a). The objective in this domain is to find one of the goal states (marked with "G") as quickly as possible. The actions available at every state are `north`, `south`, `east` and `west`, corresponding to the four possible directions. The arrows in Fig. 3(a) represent the optimal policy π^* for the following reward function: each time the agent performs an action, it receives a reward of -0.5, except if it ends up in one of the two goal states, in which case it gets a reward of $+8$ and is repositioned at a random state. If the agent runs into a wall (the limits of the maze or the dark gray squares in Fig. 3(a)), it remains in the same state, but still gets a "reward" of -0.5. Figs. 3(b) and 3(c) show the corresponding state-value function[2] for the case of a discount factor $\gamma = 0.9$.

The task in this first experiment is to use the RGD algorithm to perform the policy evaluation step, that is, given π^* use Algorithm 1 to compute the corresponding value function $Q^{\pi^*}(s,a)$. In order to assess the quality of the approximation $\tilde{Q}^{\pi^*}(s,a)$ we report the measure ξ_1 defined in (7). However, it is known that even small approximation errors can result in large deviations of the approximated policy $\tilde{\pi}$ with respect to the true π, as discussed in Section 3. Since the final goal of reinforcement learning is to find a good policy (and not a good value-function approximation), we defined another metric, which can be seen as a measure of how well one can "restore" π from its approximate value function $\tilde{Q}^{\pi}(s,a)$. The metric ξ_2 is defined as follows:

$$\xi_2 = \frac{1}{|S|} \sum_s \delta\big(\pi(s), \tilde{\pi}(s)\big), \quad \text{with } \delta\big(\pi(s), \tilde{\pi}(s)\big) = \begin{cases} 1 & \text{if } \pi(s) \neq \tilde{\pi}(s), \\ 0 & \text{otherwise.} \end{cases}$$

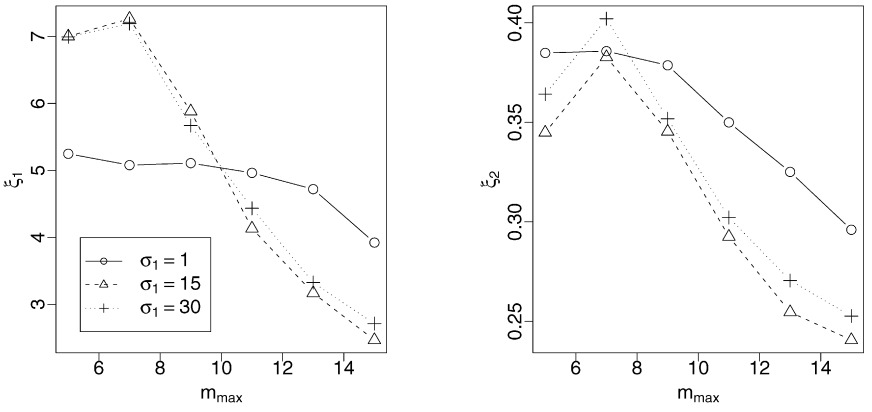
We defined a simple strategy to control the complexity of the models generated by the RGD algorithm: when the number m of units in the RBF network reaches a limit $m_{\max}$, no more changes are made to its hidden layer (that

[2] The value function of a state s is defined as $V^{\pi}(s) = \max_a Q^{\pi}(s,a)$.



(a) ξ_1 (b) ξ_2

Fig. 4. Results on the maze task with $\sigma_1 = 30$. The points correspond to the results achieved after 1,000,000 transitions. Averaged over 50 runs.



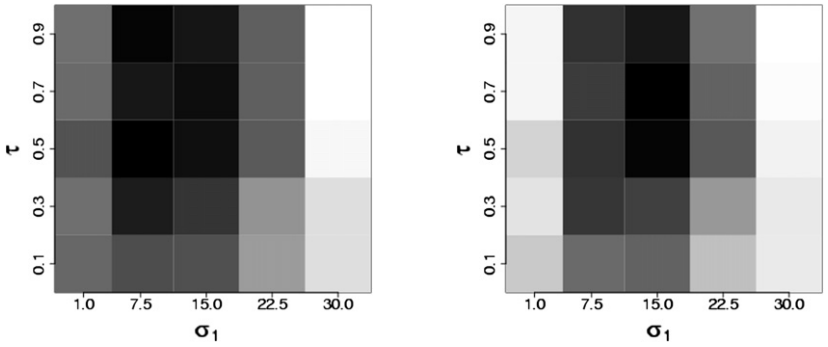
(a) ξ_1 (b) ξ_2

Fig. 5. Results on the maze task with $\tau = 0.5$. The points correspond to the results achieved after 1,000,000 transitions. Averaged over 50 runs.

is, neither new RBFs are added nor the ones already in the model have their parameters changed). Fig. 4 shows the performance of the RGD algorithm when using different values for $m_{\max}$ and τ (the initial width used in this experiment was $\sigma_1 = 30$, which leads to an activation level of the first RBF greater than 0.5 over the entire state space). The first thing to notice when analysing Fig. 4 is that the metrics ξ_1 and ξ_2 are in fact not as correlated as one would expect. Even so, note how an intermediary value of $\tau = 0.5$ generates the best results for both measures. The difference is particularly relevant when one considers ξ_2: for $\tau = 0.1$ and $\tau = 0.9$, the policies $\tilde{\pi}$ found by RGD vary between an error rate of 30% and 40% (regardless of the number of RBFs), while for $\tau = 0.5$ the quality of the policy increases monotonically with the number of hidden units. Apparently, a high value for τ increases the interference between the radial functions, while too low a value decreases the smoothness of the function computed by the RBF network.

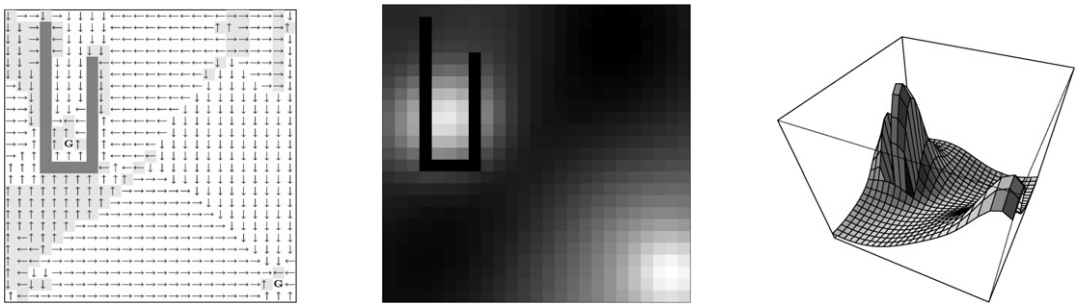
We now use the value $\tau = 0.5$ found in the last experiment to check the performance of RGD under different values for σ_1 and $m_{\max}$. If there were no limit on the number of hidden units, smaller values for the initial width σ_1 would probably generate better results, since the number of RBFs in the final network's hidden layer would increase. If the resources are limited, though, using larger widths might be a better choice. Fig. 5 makes it clear that using an initial width $\sigma_1 = 1$—which generates an activation level greater than $\tau = 0.5$ over only one state s—degrades the results of RGD almost under all values of $m_{\max}$.

Besides influencing the widths of the new RBFs, a large σ_1 also increases the number of adaptation steps of the first hidden unit, which provides an initial approximation of $Q^{\pi}(s, a)$. But how large should σ_1 be? Fig. 5 shows that



(a) ξ_1—minimum at (7.5,0.5) (b) ξ_2—minimum at (15,0.7)

Fig. 6. Gray map representing the results on the maze task with $m_{\max} = 15$. The brighter the square, the worse the performance of the RGD algorithm. The points correspond to the results achieved after 1,000,000 transitions. Averaged over 50 runs.



(a) Policy $\tilde{\pi}$ generated by the agent. The light gray areas represent states for which $\tilde{\pi}(s) \neq \pi^*(s)$.

(b) Gray map of the value function generated by the agent.

(c) Landscape of the value function generated by the agent. The value of the internal walls are considered to be $\min_{(s,a)} \tilde{Q}^{\pi^*}(s,a) - 1$.

Fig. 7. Results on the maze task after 1,000,000 transitions using $\tau = 0.5$ and $\sigma_1 = 15$.

the difference on the results when the initial width changes from 15 to 30 is not very significant. Intuitively, one can say that, if the number of transitions is large enough, the performance of the RGD algorithm will not change very much when σ_1 is above a threshold σ^*, since the RBFs will shrink down to their "right" size. The closer σ_1 is to σ^*, the faster RGD will find a reasonable solution. The exact value of σ^* is domain-dependent, and might vary with the number of transitions (obviously, if the number of transitions is fixed, too large a σ_1 will result in poor performance, since the RBFs' widths will not have enough time to adjust).

The value of the "ideal" initial width σ^* may also depend on the parameter τ, since it changes the way σ_1 affects the widths of the RBFs that will be created (see (8)). To illustrate this point, we performed several experiments in which the maximum number of RBFs was fixed at $m_{\max} = 15$ while τ and σ_1 changed. The results after 1,000,000 transitions are shown in Fig. 6. Notice that extreme values for both τ and σ_1 result in bad performance of the RGD algorithm. In general, the configurations in the center of Figs. 6(a) and 6(b) perform better than those in the edges. Even though neither of the two metrics has a minimum at the exact center, both of them present their best performances with a configuration close to $\sigma_1 = 15$ and $\tau = 0.5$.

Fig. 7 shows the result of a single execution of the RGD algorithm using $\tau = 0.5$, $\sigma_1 = 15$ and $m_{\max} = 15$. As shown in Fig. 7(a), after 1,000,000 transitions the agent's policy $\tilde{\pi}$ selects the optimal action in more than 80% of the state space. Notice that the wrong choices are concentrated in the areas where the concavity of the value function changes, particularly around the internal walls. The reason for this is clear when one observes Fig. 7(b), which shows how the high values of the states close to the goal go across the limits of the internal walls, causing states on the opposite side of the walls to have high Q-values. Even though, Fig. 7(c) shows that the approximation $\tilde{Q}^{\pi^*}(s,a)$

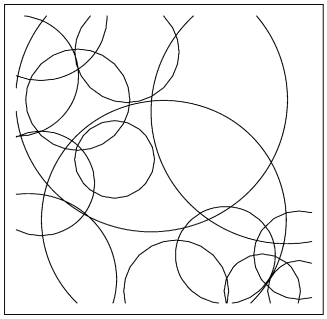


Fig. 8. Configuration of the RBFs on the maze task after 1,000,000 transitions. The parameters used were $\tau = 0.5$, $\sigma_1 = 15$ and $m_{\max} = 15$. The contours were drawn at $\theta_i(s) = \tau = 0.5$.

performed by the resulting RBF network was able to assimilate the overall characteristic of the true value function $Q^{\pi^*}(s, a)$.

An interesting issue is how exactly the approximation shown in Fig. 7(c) is constructed. It is expected that the RBF network's structure reflects the landscape of the value function, with a larger number of narrower RBFs in the areas where its value changes more abruptly. This expectation is confirmed in Fig. 8, which shows the distribution of the radial functions in the two-dimensional state space after the run described above terminated. Notice how the RGD algorithm managed to concentrate more resources of the approximator in the areas surrounding the two goals and the internal walls, where the value function is more complex.

Another point that comes up naturally regards the susceptibility of RGD to the presence of noise. In principle, one might suspect that RGD would not perform very well under stochastic environments, since the noise could cause the RBFs to ratchet down to zero width. We believe, however, that noise has the same effect on RGD as the residual errors intrinsic to the least-squares approximation. If no stop criteria are used to interrupt the shrinking of the RBFs, the widths of these functions will never stop decreasing, even if the environment is deterministic. Therefore, an adequate stop criterion would interrupt the decreasing of the radial function's widths caused by noise, just like in the deterministic case. If this is true, the sensitivity of RGD to noise is more related to the strategy used to stop the shrinking of the RBFs than to the algorithm's mechanism itself.

To test our hypothesis, we re-ran the experiment with the maze task, but now with some noise added to the environment's dynamics. Specifically, when the agent selected one of the four possible directions `north`, `south`, `east` or `west`, it was moved in the right direction with probability $1 - \eta$; with probability η (the "noise") it was randomly positioned in one of the 4 neighbor cells. Fig. 9 shows the effect of different levels of noise on the performance of RGD. Notice that, when using $m_{\max}$ as the stop criterion, the presence of noise does not affect the quality of the approximation. Actually, one can easily see that ξ_1 drops slightly when $\eta = 0.4$ and $\eta = 0.5$, probably because of the new shape of the value function (when the level of noise η is increased, the value function of neighbor states get closer to each other, since all the actions can lead to any of the neighbor cells).

6.2. Mountain car

When a stop criterion is used to interrupt the adaptation of the RBFs, the RGD algorithm operates in two distinct stages. In the first one it configures the hidden units of the RBF network, which corresponds to determining which features will be used by the output layer. After the stop criterion has been satisfied, the network's hidden layer is kept fixed, and RGD is reduced to the standard gradient-descent method applied to a linear model. An interesting question is whether the features selected by RGD in the first stage are really helpful for the value-function approximation in the second.

To investigate this issue we chose the mountain-car task [67]. In this domain the goal is to drive a car out of a valley. The challenge relies on the fact that the car's engine is not strong enough to pull it up the slope it is facing: the only way to escape is to first move away from the goal and up the other slope until enough inertia has been built up to carry the car out of the valley. This task has a continuous state space, with each state s represented by the position and velocity of the car. There are three possible actions in every state: full throttle forward, full throttle reverse and

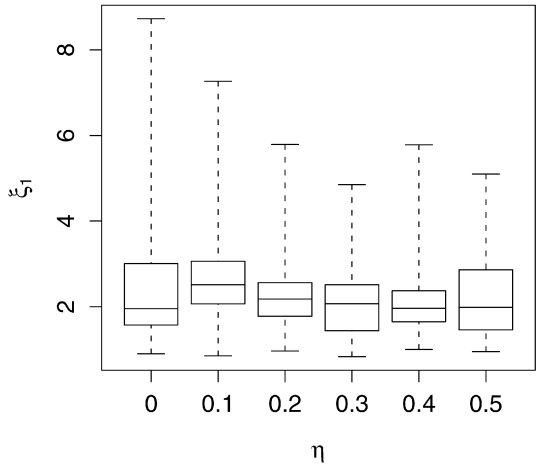


Fig. 9. Box-and-Whisker plots of the RGD results on the maze task under different noise levels. The values correspond to the approximation error achieved by the agent after 1,000,000 transitions using $\tau = 0.5$, $\sigma_1 = 15$ and $m_{\max} = 15$. The boxes in the graphics represent the median and the lower and upper quartiles. The dotted lines extend up to the extreme values. Statistics computed over 50 runs.

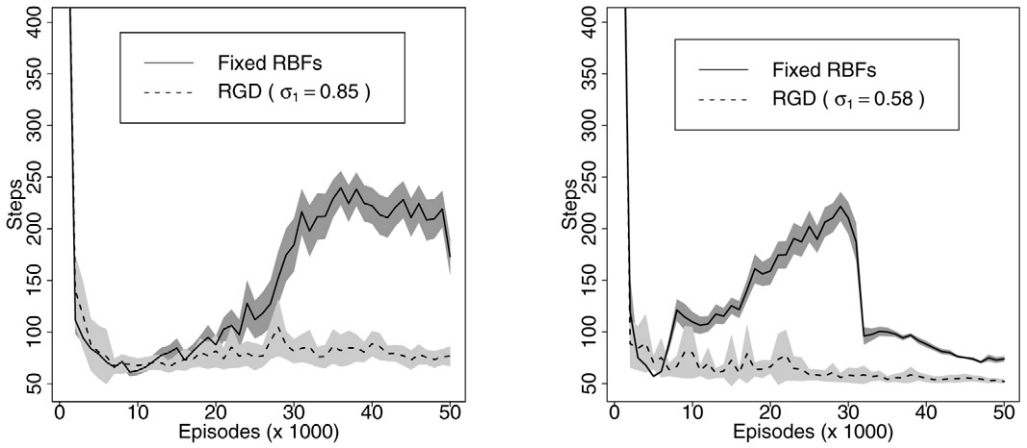
zero throttle. The agent receives a reward of -1 at every time step until it moves past its goal position at the top of the mountain, which ends the episode with a reward of 0. An episode may also terminate if the agent does not reach the goal within 1000 steps. Regardless of how the previous one has been terminated, a new episode is always started at a random position. The equations governing the system were implemented exactly as described by Sutton and Barto [77], with a discount factor $\gamma = 1$ and the input variables normalized to lie in the interval $[0, 1]$.

The mountain car is a complete control problem. To deal with this task we adopted the already mentioned SARSA(λ) algorithm using no eligibility traces (that is, $\lambda = 0$; see Appendix B for details). In order to analyse the quality of the features selected by RGD we combined SARSA with this algorithm and also with the standard gradient-descent method applied to linear RBF networks of two different sizes. The linear models had 4 and 9 fixed RBFs evenly distributed over the two-dimensional state-space. To guarantee that the comparison would be made between models of about the same size, the parameter $m_{\max}$ of the RGD algorithm was assigned the same values (recall that $m_{\max}$ is only an upper limit on the number of hidden units). The widths of the fixed radial functions were determined using (8) with an activation level of 0.5. The initial widths σ_1 used by RGD were determined as being 1.5 the widths of the fixed RBFs in the linear model of the same size. Based on the experiments of the last section, the minimum-activation threshold used by RGD to allocate new units was set as $\tau = 0.5$.

To assess the quality of the approximation constructed by both methods, we counted after each episode the number of steps taken by the agent to escape from 100 randomly-selected states (the same set of states was used for all the experiments). Fig. 10 shows these numbers for RGD and the standard gradient-descent algorithm applied to RBF networks with fixed hidden units. Notice in Fig. 10(a) how the performance of the networks with 4 fixed hidden units degenerates after episode 20,000 (observe that the shadowed regions representing the 95% confidence interval of both methods do not overlap after around episode 30,000). With the 9 fixed-units networks a different phenomenon occurs, as shown in Fig. 10(b): their performance degrades from episode 10,000 up to around episode 30,000, when it starts to improve again. The results of the RGD algorithm make it clear that, at least in this problem, its strategy to position the RBFs leads to a stable behavior. Notice how the policy generated by SARSA-RGD consistently keeps the number of steps to escape under 100, even when using only 4 RBFs in the hidden layer.

In Section 4 we mentioned that the indiscriminate application of the delta rule on the on-line gradient-descent algorithm may lead to divergence in the context of reinforcement learning. To verify this statement we repeated the experiment above with the standard gradient-descent algorithm applied to an RBF network with a tunable hidden layer. The networks were initialized exactly as before, but now the centers and widths of the RBFs were allowed to adapt.

Table 1 shows the average results achieved by RGD and the standard gradient-descent method applied to RBF networks with fixed and adjustable RBFs. The values correspond to the performance of the agent after 50,000 episodes. Notice that when using 4 units in the hidden layer the RBF network with adjustable units is unable to learn anything



(a) 4 RBFs (b) 9 RBFs

Fig. 10. Average number of steps taken to escape from 100 randomly-selected initial states of the mountain-car task. The shadowed gray regions correspond to the 95% confidence interval computed for 50 runs.

Table 1
Average number of steps taken to escape from 100 randomly-selected initial states of the mountain-car task. The values correspond to the results achieved after 50,000 episodes using SARSA(0) algorithm with the models shown. The numbers on the left refer to the number of hidden units used by the RBF networks (for the RGD algorithm this is the value used for $m_{\max}$). Averaged over 50 independent runs. "SD" stands for standard deviation and "95% CI" is the confidence interval at the 95% probability level

		Mean	Best	Worst	SD	95% CI
4	Fixed RBFs	172.91	93.58	269.30	63.52	[155.30,190.52]
	Adjustable RBFs	890.69	890.69	890.69	0.00	[890.69,890.69]
	RGD ($\sigma_1 = 0.85$)	76.65	49.33	222.34	34.30	[67.14,86.16]
9	Fixed RBFs	73.85	63.80	97.48	9.45	[71.23,76.47]
	Adjustable RBFs	205.96	46.11	890.69	305.69	[121.23,290.69]
	RGD ($\sigma_1 = 0.58$)	52.04	47.04	75.77	9.76	[49.33,54.75]

about the task, performing a number of steps near the maximum possible in all final episodes of all runs (theoretically, a failure to escape from all 100 initial states would result in an average of 1000 steps. The number 890.69 probably reflects the presence of trivial states from which the car is able to escape the valley no matter what actions are selected by the agent). When the number of hidden units is increased to 9 the performance of the model with adjustable RBFs improves, but is still worse than the other two. For both 4 and 9 units in the hidden layer the RGD algorithm presents results substantially better than those obtained by the standard gradient-descent algorithm. This difference is statistically significant at the 95% probability level.[3] Notice that the worst result of RGD corresponds to an average of 222.34 steps to escape from the 100 initial states. This is much smaller than the maximum number of steps possible, which indicates convergence to a reasonable solution in all cases.

The experiments with the mountain-car task make it clear that RGD is able to generate better results than the standard gradient-descent algorithm. Why exactly does it happen? Besides the on-demand allocation of new units, RGD has three characteristics that distinguish it from the standard gradient-descent method: the widths of the radial functions are only allowed to shrink, its centers only move towards the current state, and only the most activated unit has its parameters changed. An interesting question is whether all these features are really necessary, and how they interact with each other. To answer that, we performed a series of experiments in which each one of RGD's features was "turned on" and "off". The results are shown in Table 2. The first row corresponds to the case where RGD behaves more similarly to the standard gradient-descent algorithm, while the last one corresponds to the actual RGD.

[3] For the statistical tests, we used Student's "t-test" or the "u-test" from Mann and Whitney [42], depending on whether the data were normally distributed or not.

Table 2
Average number of steps taken to escape from 100 randomly selected initial states of the mountain-car task. The values correspond to the results achieved after 5000 episodes using SARSA(0) algorithm combined with RGD using different configurations. The parameters of the RGD algorithm were set as $\tau = 0.5$, $\sigma_1 = 0.58$ and $m_{\max} = 9$. Averaged over 50 independent runs. The columns "W", "C" and "U" indicate the restrictions used by each configuration. "W" and "C" refer to the changes performed by the algorithm to the widths and center of the RBFs, respectively. The column "U" indicates whether all or only the most activated unit is changed at each iteration. An "–" means all the changes are made indiscriminately (as in the standard-gradient algorithm); a "✓" indicates only restricted changes are made, as in RGD

	Restrictions			Mean	Best	Worst	SD	95% CI
	W	C	U					
1	–	–	–	547.88	50.48	890.69	392.76	[439.01,656.75]
2	–	–	✓	695.66	55.81	890.69	350.72	[598.45,792.87]
3	–	✓	–	548.24	48.80	890.69	408.81	[434.92,661.56]
4	–	✓	✓	742.28	50.11	890.69	314.28	[655.17,829.39]
5	✓	–	–	79.57	49.12	161.48	28.79	[71.59,87.55]
6	✓	–	✓	72.52	48.14	156.16	22.60	[66.26,78.78]
7	✓	✓	–	62.88	48.33	479.43	60.48	[46.12,79.64]
8	✓	✓	✓	61.71	47.35	137.94	19.13	[56.4,67.01]

The first thing that stands out when observing Table 2 is the big difference between the values shown in the first four rows and those in the last four ones. This makes it clear that the restriction on the changes made to the RBFs' widths is the main mechanism through which RGD achieves good performance. When considering the cases in which the RBFs are only allowed to shrink (rows 5 through 8), moving the centers of the radial functions always towards the current point seems to be a good strategy: notice how the means of rows 7 and 8 are smaller than the corresponding rows 5 and 6. Finally, moving only the most activated unit seems to increase the stability of the algorithm, as shown by the standard deviations. Notice how this value drops from row 5 to 6 and 7 to 8 (and also from row 1 to 2 and 3 to 4).

As a final observation, it should be mentioned that RGD achieved its best performance when using all three features that distinguish it from the standard-gradient algorithm; notice how the last row of Table 2 presents the best values for all the statistics shown. The differences between the mean shown in the last row and all the others are statistically significant at the 95% level, except for the 7th row, whose average number of steps is statistically identical to that of the standard RGD. This indicates that moving all the units or only the most activated one does not have a strong effect on the final results. It is important to notice, however, that the latter requires a much smaller number of operations, and thus might be a better choice.

6.3. Pole balancing

The pole-balancing problem is a classic reinforcement-learning task studied by many authors [1,6,45]. The objective here is to apply forces to a wheeled cart moving along a limited track in order to keep a pole hinged to the cart from falling over. At every time step the agent receives information regarding the position and velocity of the cart, the angle between the cart and the pole, and the angular velocity of the pole, which constitutes a 4-dimensional continuous state space. There are two actions available at every state: push the cart either left or right with a force of constant magnitude. If the pole falls past a 12-degree angle or the cart reaches the boundaries of the track the agent gets a reward of -1 and a new episode is started. At all other steps the agent receives a reward of 0.

Here we use the pole-balancing task to compare the performance of the RGD algorithm with that of other methods found in the literature. As a basis for our comparison we chose the work of Moriarty and Miikkulainen [47], in which the authors evaluate the performance of several techniques on the pole-balancing problem, including evolutionary and more traditional reinforcement-learning algorithms. More specifically, their "Symbiotic, Adaptive Neuro-Evolution" algorithm (SANE) is compared to the GENITOR system of Whitley et al. [88], the Adaptive Heuristic Critic (AHC) with both single-layer [6] and two-layer networks [2] and a lookup-table version of the Q-learning method of Watkins and Dayan [84]. The discretization of the state-space used by the single-layer AHC and the Q-learning algorithms was based on the work of Barto et al. [6], in which prior knowledge about the domain was used to partition the space into 162 non-overlapping regions. The other methods received the continuous values of the state variables. For our experiments we adopted the SARSA(λ) algorithm combined with RGD as in Section 6.2, but now with eligibility traces

Table 3
Results averaged over 50 runs on the pole-balancing task. The episodes refer to the number of attempts (or "starts") necessary to balance the pole for 120,000 steps starting with the pole straight up and the car centered in the track

	Episodes to learn				Failures	Free parameters
	Mean	Best	Worst	SD		
1-layer AHC	232	32	5003	709	0	162×2
2-layer AHC	8976	3963	41308	7573	4	35×2
Q-learning	1975	366	10164	1919	0	162
GENITOR	1846	272	7052	1396	0	35
SANE	535	70	1910	329	0	40
SARSA-RGD	411	5	924	237	0	18.6 ± 2.9

with a decay rate of $\lambda = 0.5$. The initial width $\sigma_1 = 1$ was computed so that the first RBF would have an activation level greater than $\tau = 0.5$ over the entire state space. In order to have a fair comparison, we ran our simulations on a version of the pole-balancing task implemented exactly as described by Moriarty and Miikkulainen [47].

The task in the first experiment was to find a network able to balance the pole for 120,000 time steps starting with the pole straight up and the cart centered in the track. Neither the cart nor the pole had any initial velocity. A failure was said to occur if the agent was not able to achieve the goal within 50,000 episodes. Notice that with this task it is not straightforward to use a fixed exploration strategy with SARSA (we adopted ϵ-greedy exploration with a fixed ϵ, as detailed in Appendix B). Since this algorithm does not have an explicit actor—but instead derives it from the current approximation of $Q^{\pi}(s, a)$—the only way to balance the pole for a reasonable amount of time is to reduce the exploration rate, since exploratory moves can easily end an episode. In this context, though, defining a decreasing schedule for ϵ could make the analysis of RGD's performance more difficult, since the success of this algorithm would depend on one more parameter.[4] Therefore, instead of gradually decreasing the exploration rate we simply tested the performance of the current RBF network after each episode. This test was performed without exploring or learning, very much like a "validation" step in supervised learning [62]. If the network was able to balance the pole for 120,000 steps, the run was terminated. If not, the learning process continued with a fixed ϵ-greedy exploration. Notice that with such an "external" criterion to interrupt the training process no special care had to be taken to stop the shrinking process of the RGD algorithm.

Table 3 shows several statistics regarding the number of episodes taken by each method to balance the pole. Notice how these numbers are extremely favorable to RGD: besides learning the task faster, this algorithm also presented a stable behavior, as shown by the lower standard deviation. These results are even more impressive when one considers that the other methods do not configure the topology of the approximators, using instead models whose complexity are known to be sufficient for this problem. Particularly, the 1-layer AHC—the only algorithm to learn faster than RGD on average—used a set of features constructed based on knowledge about "useful regions" of the state space [6], which in theory makes the task much easier. Notice also that the models generated by RGD were on average simpler than those used by the other algorithms, as shown by the number of free parameters associated with each method (the AHCs adopted two networks of the same size, one for the actor and one for the critic).

The success of RGD in this experiment might be a consequence of the stop criterion used by this algorithm: as the task is always initialized at the same start position, checking the current solution against the initial state corresponds to verifying if the policy derived from $\tilde{Q}(s, a)$ is able to accomplish the task. This is the same as having a validation set in supervised learning that is coincident with the final test set. Indeed, this stop criterion may generate highly specialized solutions, able to perform the task only in the restricted set of states tested. Even though, this might be a valid strategy in tasks that are initialized always in the same way, as is the case of this version of the pole-balancing problem. Notice also that this is the same stop criterion adopted by the evolutionary methods, both of them outperformed by RGD. In the next experiment we will verify how this strategy to stop the learning process extends to domains with a larger set of initial states.

The agents trained in the last experiment were able to balance the pole from one specific start position, namely: the pole straight up, the car centered and both velocities equal zero. A more interesting challenge is to learn the task

[4] The strategy adopted by Moriarty and Miikkulainen [47] to stop the Q-learning algorithm is not clear.

over a range of initial states and then try to generalize over the entire state space. Our second experiment was set up in order to reproduce this scenario: it was configured exactly as the previous one, but instead of always starting the learning process at the same state, the four input variables were randomly selected from the range of possible values. In the work of Moriarty and Miikkulainen [47] all the algorithms were interrupted when the pole had been balanced for 120,000 time steps from any initial random position. To be coherent with our decision of not decreasing the exploration rate, we generalized the stop criterion used in the last experiment: now, the process was stopped when the network being constructed by RGD was able to balance the pole for 1000 steps from n consecutive start positions, with no failure. This test was made with the model fixed and no exploratory moves.

To verify the quality of the solutions generated by each algorithm we measured their generalization ability against 100 random initial states. These values are shown in Table 4, along with the number of episodes taken by each method to achieve the goal. The first thing to note is that using only 1 start position as a criterion to stop RGD's learning process does indeed result in very specialized solutions, as discussed before. Notice that, when using this strategy, the RGD algorithm presents a reasonable learning rate, but very bad generalization. When the number of start positions used as the stop criterion is increased, the expected phenomenon occurs: the number of episodes to accomplish the task also increases, and the resulting RBF networks present much better generalization. Notice that the results of SARSA-RGD when using $n = 5$ and $n = 7$ are quite good, specially considering that many of the initial states used to assess the networks' generalization represent irrecoverable situations, that is, states from which it is impossible to balance the pole [88].

The values shown in Table 4 make it clear that the RGD algorithm takes a larger number of episodes to learn the pole-balancing task than the other algorithms. This is true even when only 1 initial state is used to stop RGD. Why does it happen? The algorithms being compared in this section can be divided in three categories, according to the feature space they work in. Q-learning and 1-layer AHC operate on a set of features pre-selected based on human knowledge about the problem. The 2-layer AHC, GENITOR and SANE extract the features used in the approximation through a mapping from the original input space to a higher dimensional hidden space of *fixed* size. The RGD algorithm belongs to a third category, in which the dimension of the hidden space is also learned. We conjecture that the difference on the number of episodes to learn the task is related to the amount of *a priori* information given to each algorithm about the feature space.

Even using less information about the feature space RGD presented the best generalization performance among the tested algorithms (except for the SARSA-RGD-1 case, in which the algorithm clearly stopped prematurely). At the 95% probability level, the difference on the generalization of the other algorithms is not statistically significant, except between 1-layer AHC (the best) and Q-learning, which presented the worst generalization among all. When comparing RGD with the others, the difference on the mean generalization is significant for $n \geqslant 5$. Notice, however, that the stop criterion used by RGD was specially designed to improve generalization, and in principle it is possible to come up with similar strategies for the other algorithms.

The values shown in Table 4 clearly indicate that RGD is a stable algorithm, at least in the pole-balancing task. Notice that for $n \geqslant 3$ this algorithm presents very low standard deviations of the generalization metric, and the worst

Table 4
Results averaged over 50 runs on the pole-balancing task. The episodes refer to the number of attempts necessary to balance the pole from a random initial position. The generalization was measured as the number of initial states out of 100 randomly-selected ones from which the trained agents were able to balance the pole for 1000 time steps. The "n" in the SARSA-RGD-n labels represent the number of states used as stop criteria (that is, the number of consecutive start positions from which the agent was supposed to balance the pole to terminate an episode)

	Episodes to learn				Generalization			
	Mean	Best	Worst	SD	Mean	Best	Worst	SD
1-layer AHC	430	80	7373	1071	50	76	2	16
2-layer AHC	12513	3458	45922	9338	44	76	5	20
Q-learning	2402	426	10056	1903	41	61	13	11
GENITOR	2578	415	12964	2092	48	81	2	23
SANE	1691	46	4461	984	48	81	1	25
SARSA-RGD-1	2844	1102	5993	1367	15	61	1	21
SARSA-RGD-3	6618	2340	14846	2670	50	68	16	12
SARSA-RGD-5	8455	2249	23666	4250	56	74	17	12
SARSA-RGD-7	11387	2976	26851	5534	58	80	40	8

network found in all 50 runs of all 3 configurations was able to balance the pole in 16% of the states tested, which is not bad when compared to the other methods. Another indication of RGD's stability is the fact that it has not failed to balance the pole in any of the 50 runs, in contrast with 1-layer AHC (3 failures) and 2-layer AHC, with 14 failures.[5] Finally, it is interesting to note that the size of the RBF networks generated by RGD are compatible with the complexity of the models used by the other methods (shown in Table 3), spanning from an average of 50 free parameters for $n = 3$ to 86 when $n = 7$.

The conclusion is that the RGD algorithm is a viable alternative in problems similar to the pole-balancing task. If one has enough information about the state-space to partition it or even to define the right structure of the approximator, any of the methods shown in the upper half of Table 4 would be expected to perform similarly. If this information is not available, RGD might be a good choice.

A final point should be mentioned regarding the performance of the evolutionary methods (GENITOR and SANE) on the experiments with the pole-balancing task. Evolutionary algorithms have always been particularly successful on this task, since the early works [88] until more recently (see [31] and references therein). So much so that the pole-balancing has become a benchmark problem in the field, and since the publication of Moriarty and Miikkulainen's work more difficult versions of the task have been proposed and successfully addressed [31,35,71]. In addition, comparisons between recent evolutionary methods and traditional value-based reinforcement learning algorithms seem to indicate a clear advantage of the first over the second [31].[6]

However, we believe part of this success is due the fact that the pole-balancing belongs to a class of control problems particularly suitable for optimization techniques. As well known, the search performed by evolutionary algorithms is based on information gathered between episodes, but not within them. This type of search is feasible in tasks like balancing a pole, where the learning process spans a large number of short episodes. However, it does not seem practical to use evolutionary methods in tasks where the episodes themselves are long, as for example in the game of chess. The methods based on temporal-difference learning, on the other hand, tend to be less sensitive to this aspect, since the learning occur both intra and inter episodes. Perhaps more importantly, in the pole-balancing problem it is possible to assess the quality of a candidate solution even if it has failed to accomplish the task (here, the quality was measured by the number of steps a solution could balance the pole for). Although many problems have a similar formulation, this is not always true. In shortest-path problems very little information can be gathered from a failure other than the reinforcement signal, and ranking the potential solutions is no longer a trivial task. In the next section we present a control problem with such a characteristic, and make the present discussion more concrete.

6.4. Acrobot

Underactuated mechanical systems have more degrees of freedom than actuators. Examples of such systems include manipulator arms on diving vessels or spacecrafts, non-rigid body systems, and balancing systems such as unicycles or dynamically stable legged robots [17]. In this section we study the Acrobot, an underactuated non-linear system that has been studied by both control [70] and machine-learning [16,75] researchers. The Acrobot is an interesting task because its dynamics are complex enough to yield challenging control problems, yet simple enough to permit a complete mathematical analysis and modeling. In our experiments we used a simulator whose equations of motion are given in [17,70,77].[7]

The Acrobot is a two-link robot arm powered at the elbow but free-swinging at the shoulder. It resembles a gymnast swinging on a bar, in which case only the joint corresponding to the gymnast's waist can exert torque (thus the name). The goal is to swing the gymnast's "feet" above the bar by an amount equal to one of the links as fast as possible (a reward of -1 is given to the agent on all time steps, with no discounting). The choice to be made at every time step is the torque applied at the second joint. Following Sutton [75], we restricted the options to three choices: positive torque of $+1$ Nm, negative torque of -1 Nm, or no torque at all. The state-space is continuous and 4-dimensional, with two

[5] Following Moriarty and Miikkulainen [47], the failure cases were not included in the statistics of Table 4.

[6] Gomez et al. focus on the problem of solving a sequence of increasingly difficult versions of the pole-balancing task, some of them not solved by value-function based methods [31]. Unfortunately, they do not report a measure of the generalization capability of their evolved networks, which seems to be a dimension in which traditional reinforcement-learning algorithms produce competitive results.

[7] We used 4th order Runge–Kutta integration with a time step of 0.005 seconds and actions chosen after every 10 time steps (the reason for such choices is given further in the text). The constants were set as in [17,75], and we did not restrict the velocities of the links.

variables representing the joint positions and two representing the joint angular velocities. All episodes are started at the stable position $\vec{s} = [0, 0, 0, 0]$, and terminate when the agent reaches the goal or a maximum of 1000 steps.

In our simulations actions were applied at a frequency of 20 Hz, contrasting with the usual choice of 5 Hz. This modification makes the task considerably harder, and also explains the larger number of steps taken by the agents to reach the goal in our experiments when compared to previous results [16,75]. Notice that the Acrobot task formulated in this way is a relatively hard shortest-path problem, and as such it presents the characteristics discussed in the last section. In particular, a "bad" agent/controller will often perform a sequence of actions that will never lead to the goal, which results in the episode being truncated at an arbitrary point. Therefore, the only information returned by an agent that has failed to accomplish the task is a "failure signal". Any optimization technique that relies on the concept of an objective function—evolutionary methods included—will have problems ranking the unsuccessful candidate solutions. In fact, unless a successful solution luckily emerges in the process (which is highly unlikely in this case), the optimization will be reduced to a random search.

In order to check the last statement, we implemented four evolutionary methods and tested them on the Acrobot task. We tried a conventional real-coded genetic algorithm [30], its steady-state version (which is very similar to the GENITOR algorithm of the last section [88]), and two evolution strategies: $(1, 10)$-ES and $(1 + 10)$-ES [15], which have recently shown excellent performance on the pole-balancing task when combined with genetic programming [87]. In all of them an individual encoded the linear weights of an RBF network with 81 Gaussian units evenly distributed over the state space (we tried 10 different levels of overlap between the functions, as detailed below). Besides the RBFs, we also adopted a constant term. As in (6), we used one linear model for each action, thus each candidate solution was a real-vector of length $3 \times 82 = 246$. The fitness of a solution was defined as $1000 - ns$, where ns is the number of steps taken by the corresponding policy to accomplish the goal. The genetic algorithms used a population of 100 individuals, and all four methods were interrupted after 10,000 evaluations had been carried out.[8] All other choices were standard [38]. Each algorithm was executed 10 times for each configuration of the hidden layer, resulting in 100 independent runs; *none* of them was able to find a *single individual* capable of achieving the goal in less than 1000 steps.[9]

Given the bad results achieved by the evolutionary methods, we proceeded to try a value-function based method, namely the least-squares policy iteration algorithm (LSPI) [39]. LSPI is an approximate policy-iteration algorithm. It extends the benefits of least-squares temporal difference (LSTD) [18,20] to the control problem. Like the second, the former makes efficient use of data and eliminates learning parameters. Unlike LSTD, LSPI does not need a model of the system to perform the policy update and can be used with data collected from any reasonable sample distribution. The LSPI algorithm enjoys good convergence properties [39] and has been applied to several problems [40].

The first step when applying LSPI to any task is to collect data in the form of transitions (s, a, r). Usually, sample trajectories are generated by exploratory policies [39]. However, using random policies in the Acrobot task tends to concentrate the data around the equilibrium state $\vec{s} = [0, 0, 0, 0]$. Sampling transitions from a random distribution is not a trivial task with the Acrobot, either. First, the variables representing the links' velocities are not bounded, and we have to define an interval from which to pick the samples. More importantly, we want the data to be concentrated in the relevant regions of the state space, which is hard to define *a priori*. In order to overcome these difficulties, we decided to implement a graphical interface and let some people "play" with the Acrobot. Based on the movements of the Acrobot on the screen, the person could choose at every time step the torque to be applied on the system, like a very simple videogame. The idea was to figure out what parts of the state space are really visited during a reasonable episode, and also to set up a baseline against which to compare the algorithms' results. Each person was asked to interact with our simulator until he/she got familiarized with the system's dynamics. After that, each person played

[8] This number was set in order to make the number of operations compatible with those performed by the other methods applied to this task. In particular, since each evaluation takes approximately $1000 \times |A| \times m$ operations, where $m = 82$ is the number of features used, the total cost of each run was of the order of magnitude of 10^9 operations.

[9] Notice that this formulation of the Acrobot task was deliberately designed to illustrate the difficulty of applying conventional optimization techniques to hard shortest-path problems. The "standard" version of the problem—in which actions are applied at a frequency of 5 Hz—is easily solved by evolutionary methods [43]. Notice also that the only information used to define the fitness of an individual was ns, the number of steps taken by it to reach the goal. It is, of course, possible to design more elaborate fitness functions [22,91], but this requires domain-specific knowledge not used by the value-function based methods.

Table 5
Results obtained by 5 different people with our Acrobot simulator. The values refer to the number of steps taken to swing the Acrobot's tip above the bar. Averaged over 10 episodes

Person	Mean	Best	Worst	SD
1	402.0	302	472	49.16
2	290.6	232	630	121.22
3	365.6	299	673	110.62
4	453.4	245	651	106.28
5	612.6	391	996	217.68

for 10 episodes and we recorded the data generated. We were helped by 5 people, which resulted in 21,242 transitions. The detailed information regarding this experiment is given in Table 5.

Our first attempt was to use the data generated by the humans' interaction with the system directly, but LSPI was not able to find any policy capable of performing the task when using this dataset. Thus, we generated a larger dataset in the following way. First, we bounded the four variables based on our human-generated dataset (that is, the limits of the intervals were defined as the minimum and maximum values present in the data for the corresponding variable). Then, we picked 10,000 states $\vec{s}_i$ evenly distributed over the hypercube defined by these intervals, and in each $\vec{s}_i$ we applied each one of the 3 actions. This resulted in a dataset with 30,000 transitions. All the results of LSPI reported here were generated using this dataset.[10]

We began our experiments with LSPI using a Gaussian RBF network with 16 hidden units located in a $2 \times 2 \times 2 \times 2$ grid, plus a constant term. Unfortunately, none of our experiments with this architecture were successful, and thus we increased the granularity of the grid to 81 RBFs. We used the same width σ for all hidden units, and tried different values for this parameter. In particular, we used (8) to define several levels of overlap τ between neighbor radial functions. We started with the intuitive values $\tau \in \{0.9, 0.7, 0.5, 0.3, 0.1\}$, but since we noticed smaller values generated better results, we also tried $\tau \in \{0.09, 0.07, 0.05, 0.03, 0.01\}$. This resulted in 10 different configurations of the hidden layer (these were the configurations used in the experiments with the evolutionary methods). LSPI was executed for 10 iterations, and since each iteration takes around $m^2 \times 30{,}000 + (m \times |A|)^3$ operations, the total number of operations performed on each run was of the order of 10^9.

Table 6 presents preliminary results of LSPI on the Acrobot task when using different levels of overlap between the RBFs. Notice that for $\tau = 0.9$, $\tau = 0.7$ and $\tau = 0.5$ none of the 10 runs was able to find a policy that could swing up the Acrobot in less than 1000 steps. This may not come as a surprise for the first two values, but we expected better results for $\tau = 0.5$. Anyway, the results improve significantly for smaller levels of interference between the RBFs, and for $\tau \in [0.05, 0.1]$ the average results obtained by LSPI are better than those achieved by 4 out of 5 people who tried our simulator. The LSPI algorithm reaches its best performance at $\tau = 0.05$, with the smallest average and standard deviation among all. We used this configuration to run extra experiments with LSPI, as will be described further in the text.

The experiments with the RGD algorithm were much simpler to do, since we did not have to define a dataset. We simply combined RGD with SARSA(0) and executed it on-line. We used ϵ-greedy exploration with a fixed $\epsilon = 0.5$. As in the previous experiment, the agent was allowed to explore during learning only, which explains such a large exploration rate. Every time we wanted to check the performance of the current RBF network we fixed $\epsilon = 0$, which corresponds to using the greedy policy with respect to the current value-function approximation. All the results of RGD refer to this setting. We adopted a decaying learning rate starting at $\alpha = 10^{-3}$ and going down to $\alpha = 10^{-6}$. The number of episodes performed during learning was defined in order to make the computational cost of SARSA-RGD compatible with LSPI's. Since in our current RGD implementation each step takes approximately $8m$ operations, and considering each episode takes at most 1000 steps, we set the number of episodes as 600 for the 16-RBF case and as

[10] We tried several other datasets as, for example, sampling a larger number of transitions as described above or merging human-generated data with transitions uniformly sampled. None of them resulted in any improvement on LSPI's results.

Table 6
Results of LSPI on the Acrobot task. Each run consisted of 10 iterations. Averaged over 10 runs

τ	Mean	Best	Worst	SD
0.90	1000.0	1000	1000	0.00
0.70	1000.0	1000	1000	0.00
0.50	1000.0	1000	1000	0.00
0.30	927.5	855	1000	76.42
0.10	353.3	257	1000	227.60
0.09	364.9	288	1000	223.44
0.07	384.8	313	1000	216.18
0.05	335.0	315	343	12.89
0.03	422.4	374	800	133.22
0.01	562.9	431	1000	170.14

Table 7
Results obtained by SARSA-RGD on the Acrobot task with several parameter configurations. The value of the hidden layer's learning rate β is given relative to α, the learning rate of the linear parameters. Results averaged over 10 runs

Parameters			$m_{\max} = 16$ RBFs				$m_{\max} = 81$ RBFs			
τ	σ_1	β	Mean	Best	Worst	SD	Mean	Best	Worst	SD
0.5	10	0.1α	471.2	295	1000	280.54	264.2	245	343	29.62
0.5	10	α	633.7	297	1000	318.28	261.1	242	283	14.69
0.5	50	0.1α	327.9	247	411	56.25	300.4	247	459	73.30
0.5	50	α	307.9	224	358	35.02	272.3	248	297	16.73
0.7	10	0.1α	647.2	275	1000	371.96	256.6	243	280	12.47
0.7	10	α	438.1	293	1000	296.16	567.6	260	1000	372.38
0.7	50	0.1α	502.6	273	1000	343.43	254.5	244	285	11.57
0.7	50	α	431.8	274	1000	299.80	253.3	230	282	15.04

3000 for the experiments with 81 hidden units.[11] We experienced with different values for τ, σ_1 and β (the hidden layer's learning rate). The results are shown in Table 7.

The first thing that stands out in Table 7 is the fact that SARSA-RGD was able to find successful policies even when using a maximum of only 16 hidden units to approximate the Q-function. We believe the strategy used by RGD to configure the hidden layer is playing an important role here, specially considering LSPI was unable to find a single successful solution using the same network architecture. Notice that for $\tau = 0.5$ and $\sigma_1 = 50$ the results obtained by SARSA-RGD with a 16-RBF network are competitive with those achieved by LSPI with 81 RBFs, and really close to the best results found by human experience. When $m_{\max} = 81$ hidden units, SARSA-RGD consistently overcomes LSPI and often the best results of Table 5. Perhaps more importantly, RGD manages to find a solution for the problem in all 10 runs of 7 out of 8 configurations.

In order to get more reliable results, we reran the experiments with the best configuration of each algorithm, now averaging over 50 executions. LSPI was applied to an RBF network with 81 hidden units and an overlap of $\tau = 0.05$ between neighbor functions. RGD used the parameters on the last row of Table 7, namely $m_{\max} = 81$, $\tau = 0.7$, $\sigma_1 = 50$, and $\beta = \alpha$. The results are shown in Fig. 11 and Table 8.

Fig. 11 shows the performance of both LSPI and SARSA-RGD over time. Notice how LSPI makes little progress after the first iteration, and after the 5th one all the 50 runs converge to about the same solution, which remains unaltered until the last iteration. At the 95% confidence level, we can say LSPI will find a policy that needs at least 332 steps to accomplish the task, as shown by the last column of Table 8. Contrasting with LSPI, RGD's results decrease monotonically until around episode 1500, and after this point the results seem to bounce around 300 steps. The confidence interval of RGD's final result is about 8.8 times wider than LSPI's, which suggests more variation on the algorithm's behavior from one run to the other. Even so, there is a clear advantage of the first algorithm over

[11] The resulting numbers of operations are of the same order of magnitude as those performed in 10 iterations of LSPI with models of the same size, namely of the order of 10^7 and 10^9, respectively.

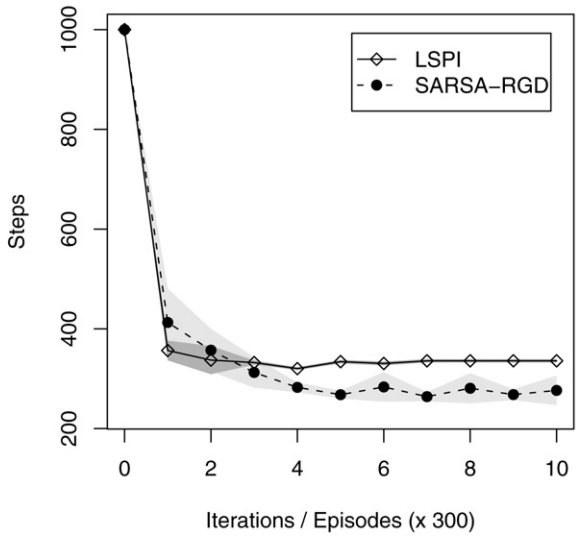


Fig. 11. Results achieved by the best configurations of LSPI and SARSA-RGD on the Acrobot task (see text for details). The shadowed gray regions correspond to the 95% confidence interval computed for 50 runs.

Table 8
Results achieved by the best configurations of LSPI and SARSA-RGD on the Acrobot task (see text for details). LSPI's numbers correspond to the results found after 10 iterations; SARSA-RGD was allowed to run for 3000 episodes (these values result in both algorithms performing a number of operations of the order of magnitude of 10^9). Averaged over 50 runs

	Mean	Best	Worst	SD	95% CI
LSPI	335.90	315	343	12.11	[332.54,339.26]
SARSA-RGD	276.56	238	1000	106.62	[247.01,306.11]

the second. Notice in Fig. 11 how RGD's curve crosses LSPI's between episodes 600 and 900 (which corresponds to the second and third iterations of LSPI), and after episode 1200 the confidence intervals do not overlap anymore. Note also that the final result of RGD represents a reduction of almost 60 steps over that of LSPI, on average. As a final point, we should mention that RGD was able to find policies able to control the Acrobot at the same level of proficiency of the most skilled people who tried our simulator.

In our experiments with the Acrobot the SARSA-RGD algorithm was able to achieve better results than LSPI performing roughly the same number of operations. We believe this happened for two main reasons. First, the way data was collected and used by both algorithms was quite different. When merged with SARSA, RGD is an on-line algorithm, and as such it actively gathers data according to its exploration strategy. Normally, the greedy action with respect to the current Q-function will be chosen more often than the others. This results in some regions of the state space being visited more often than others, which is equivalent to saying the corresponding states are assigned higher weights in the Q-function approximation (see (7)). In shortest-path problems this characteristic may be crucial, since regions of the state space "far" from the path being explored by the agent will simply be ignored. With LSPI, it is hard to define beforehand which states are important and which are not, and the usual choice is a uniform coverage of the state space. Thus, a lot of computational effort is wasted in the approximation of the Q-function in regions of the state space that will never be visited by the agent in practice.[12]

However, this fact alone might not be sufficient to explain the superior performance of SARSA-RGD when compared to LSPI. To check this out, we reran the experiments with SARSA, but now using the standard gradient-descent algorithm instead of RGD. We used an RBF network with 81 fixed hidden-units evenly distributed over the state space and tried two levels of overlap between the functions: $\tau = 0.7$ and $\tau = 0.5$. All the parameter values used in the

[12] One way to remedy this with LSPI is to resample the transitions (s, a, r) at each iteration according to the current greedy policy [39].

experiments with RGD were kept, resulting in an on-line algorithm with exactly the same exploration strategy. For each level of interference between RBFs we executed SARSA for 10 times. None of them resulted in a policy able to accomplish the task.

Another characteristic of RGD that might have helped on the Acrobot task is its strategy to define the feature space, that is, the way it configures the number of basis functions in the model, as well as its centers and widths. The results with the 16-RBF networks supports this hypothesis, since LSPI was not able to find a single solution for the problem using the same number of hidden units equally spaced over the state space. An interesting question here is: how would another non-linear method do on the Acrobot task under the same circumstances of RGD? To answer that, we performed yet another test: we reran the last experiment, but now we let the gradient descent method also configure the RBFs' centers and widths. The functions were initially positioned as in the last experiment, and again we tried two values for τ: 0.7 and 0.5. As in the experiments with RGD, we tested two values for the hidden layer's learning rate, $\beta = \alpha$ and $\beta = 0.1\alpha$. All the remaining parameters were set with the same values as before. We executed the algorithm for 10 runs with each configuration, which resulted in 40 independent runs. Again, none of the runs resulted in a successful solution.

7. Discussion

It is probably possible to improve the results of both RGD and the other algorithms on the tasks studied by using more specific configurations. We prefer instead to concentrate on the behavior of the algorithms without too much tweaking or use of domain knowledge, which in our opinion better reproduces a real-world scenario. The experiments performed in this way have shown that the RGD algorithm shares with the standard gradient-descent method one of its most desirable characteristics: generality. In all domains it was tested, RGD presented a stable behavior and was able to find reasonable solutions using several parameter configurations.

The main difference between RGD and the standard gradient-descent algorithm is in the application of the delta rule. In particular, in the RGD algorithm the widths of the RBFs are only allowed to shrink, and the centers always move towards the current state. These modifications are *conservative*, in the sense that they can not lead to divergence of the parameters to infinity. As long as a sufficiently small learning rate is adopted, the widths of the radial functions will asymptotically approach zero, and its centers will be confined to the convex hull defined in the state space by the data.

Another difference between RGD and the gradient-descent method is the possibility of allocating new hidden units. The strategy used by RGD to add and position the radial functions follows the basic philosophy of the so-called "self-organizing networks" [27,28]: 1) determine the hidden unit that is closest to the current input vector, 2) move it (and optionally its k-nearest neighbors) towards the input vector and 3) add new units on-demand, according to a pre-defined insertion criterion. One objective of such unsupervised-learning methods is dimensionality reduction [28]. In principle, if the states that come up in the agent interaction with a reinforcement-learning environment are concentrated in a low-dimensional subspace of the original state space, one can expect that RGD will restrict the RBFs to such a subspace. Notice, however, that generally this will *not* be the case, and therefore RGD is subjected to Bellman's "curse of dimensionality" [11], in the sense that the number of RBFs in the approximator will grow exponentially with the number of dimensions of the state space.

7.1. Kernel-based reinforcement learning

We believe that increasing the number of RBFs while decreasing their widths is the main mechanism through which RGD achieves its good results (see Table 2). Although derived from intuitive ideas, this strategy has surprisingly many similarities with the work of Ormoneit and Sen on "kernel-based reinforcement learning" [48], which has a more theoretical perspective. In their work, the authors show how the Q-function can be approximated by a sum of weighting kernels, which resembles a local RBF network whose functions are centered at the states $\vec{s}_i$. Ormoneit and Sen argue that this schema can be used to derive an iterative update rule similar to (4), and prove that the resulting $\tilde{Q}^\pi(s,a)$ converges in probability to the true $Q^\pi(s,a)$ as the number of sample transitions used in the approximation increases. This is a very desirable property, which the authors call *consistency*: additional training data always improve the quality of the approximation $\tilde{Q}^\pi(s,a)$, and eventually leads to optimal performance. In the reinforcement learning

context, this property is very hard to establish for conventional parametric approximators such as neural networks with a fixed architecture trained by the standard gradient-descent algorithm [48].

Besides some simple assumptions on the reward, transition and kernel functions, the only requirement for the above convergence to be true is that *the weighting kernels "shrink" with increasing sample size at an "admissible" rate*. Since in kernel-based reinforcement learning there is one kernel function for each sample transition, the last statement is equivalent to saying that the widths of the functions should decrease to zero as their number increases. Also, this reduction should not happen too fast, in order to guarantee a certain degree of overlap between neighbor functions. This is very similar to the reasoning behind RGD, though in the opposite direction: while in kernel-based reinforcement learning the number of functions determines their widths, in the RGD algorithm the opposite happens. Notice, however, that having one kernel for each sample transition is not feasible in practice (especially in on-line learning), and therefore RGD's strategy to configure the hidden layer can be regarded as a practical approach for defining the kernel functions.

It is somewhat surprising that two works following so different lines of thought have come to so similar conclusions, and in our opinion this strengthens our empirical arguments. More importantly, it opens up new interesting possibilities for future research. We believe it might be possible to fit RGD (or a slightly modified version of it) within the kernel-based reinforcement-learning framework.

8. Conclusions

The restricted gradient-descent algorithm is essentially a strategy to extract important features from the state space. If one has enough information about the problem at hand to handcraft the feature space, it is certainly a better choice to do so and use a linear model to perform the value-function approximation. On the other hand, if not much is known about the domain, the RGD algorithm may be an appealing alternative.

RGD is basically a modified version of the standard gradient-descent algorithm, and as such it inherits both its qualities and drawbacks. In particular, it is very simple and general, that is, it has wide applicability and requires minimal use of domain knowledge, as shown by the experiments. Our algorithm presents still some advantages when compared to its unrestricted form: since the changes performed by RGD are conservative, the non-linear parameters of the approximator can not diverge. Besides, RGD is able to configure the topology of the RBF networks, which may be an important feature in some situations. On the downside, we can mention the facts that RGD makes inefficient use of data when compared to least-squares methods like LSTD, and that its performance depends on the right definition of its parameters. Also, it is sensitive to the dimensionality of the state space, meaning that the size of the RBF network will usually grow exponentially with the dimension of the input space.

The study presented in this work is fundamentally an empirical analysis, intended to show the feasibility of applying the RGD algorithm to reinforcement-learning benchmark tasks. Specifically, it has been shown that this algorithm combined with SARSA presents competitive results with other methods found in the literature, including evolutionary and traditional reinforcement-learning algorithms. When merged with SARSA, RGD becomes an on-line and "on-policy" algorithm, that is, learning takes place while the agent interacts with the environment and data are collected according to the agent's actual experience. It is not difficult to think of other combinations that would give rise to algorithms with different characteristics (RGDQ-learning, for example, would be an on-line and off-policy algorithm). It is also conceivable to use a model of the environment to aid in the learning process.

A more theoretical analysis of RGD is desirable, especially regarding its connections with Ormoneit and Sen's kernel-based reinforcement learning. Like RGD, kernel-based learning relies on a local approximator with an open architecture, and in both cases an increase on the number of basis functions results in a decrease of their widths. As a theoretical framework the kernel-based approach enjoys much stronger convergence properties, but is not practical. On the other hand, RGD is practical, but lacks theoretical performance guarantees. We believe bridging the gap between them would be beneficial for both.

Acknowledgements

The first author would like to thank the support provided by the Brazilian agency "Coordenação de Aperfeiçoamento de Pessoal de Nível Superior" (CAPES), which made the current research possible. We also would like to thank

Helena, Tati, Bruninho and Carol for playing with our Acrobot simulator. Finally, we thank the reviewers from the Artificial Intelligence Journal, whose suggestions were of great value for the preparation of the final manuscript.

Appendix A. Update equations for the Gaussian function

The Gaussian function is given by:

$$\theta_i(\vec{s}) = \exp\left(-\frac{\|\vec{s} - \vec{c}_i\|^2}{\sigma_i^2}\right) \tag{A.1}$$

where $\|.\|$ denotes the Euclidean norm and $\sigma_i > 0$. The value function computed by an RBF network using m Gaussian units has the following form for a given action a:

$$\tilde{Q}_a^{\pi}(\vec{s}) = \sum_{i=1}^{m} w_i^a \theta_i(\vec{s}).$$

When using the on-line gradient-descent algorithm to minimize (7), the update rule for the RBFs' widths is:

$$\Delta_{\sigma_i} = -\frac{\partial \varepsilon^2}{\partial \sigma_i} = \varepsilon w_i^a \underbrace{4\theta_i(\vec{s})\frac{\|\vec{s} - \vec{c}_i\|^2}{\sigma_i^3}}_{\text{always } \geqslant 0}.$$

It is easy to note that the nature of the change performed by the delta rule will depend on the sign of εw_i^a, since all other terms are non-negative. If $\varepsilon w_i^a > 0$, the width will be enlarged; if $\varepsilon w_i^a < 0$, it will be reduced. Similar situation happens with the centers $\vec{c}_i$:

$$\Delta_{\vec{c}_i} = -\frac{\partial \varepsilon^2}{\partial \vec{c}_i} = \varepsilon w_i^a \underbrace{\frac{4\theta_i(\vec{s})}{\sigma_i^2}}_{\text{always } \geqslant 0}(\vec{s} - \vec{c}_i). \tag{A.2}$$

In this case, if $\varepsilon w_i^a > 0$ the center $\vec{c}_i$ will be moved toward the current state $\vec{s}$; if $\varepsilon w_i^a < 0$ it will be moved away from $\vec{s}$.

Appendix B. RGD configuration on the experiments

The RBF networks generated by RGD to approximate the value function $Q^{\pi}(s, a)$ had the structure shown in (6), that is, one output layer for each possible action a. The Gaussian function given by (A.1) was used as the hidden units' activation in all the experiments. We always started the learning process with a single unit in the network's hidden layer, and the widths of new RBFs were defined using (8). The hidden layer's learning rate β was set as 0.1α, where α is the learning rate used for the output layer (empirically, we found out that using $\beta < \alpha$ results in a more stable behavior of RGD). On the experiments with the Acrobot we also tested $\alpha = \beta$, as discussed in the text. The value of α varied among experiments. As in the maze and mountain-car tasks the number of steps to learn was not the focus of the analysis, we used a small learning rate $\alpha = 10^{-4}$. The same learning rates were used for the RBF networks with fixed and tunable hidden units in Section 6.2. In Section 6.3, on the other hand, the learning performance of RGD was compared with that of other algorithms, and thus a larger learning rate $\alpha = 10^{-1}$ was adopted. On the Acrobot task we used a decaying learning rate, as discussed in Section 6.4. No eligibility traces were used in the experiments, except in the pole-balancing task, where a decay rate of $\lambda = 0.5$ was adopted. With the SARSA algorithm an ϵ-greedy exploration strategy was used, and unless otherwise noted ϵ was set as 0.15 (which means the action presenting the largest value function was selected 85% of the time and in the remaining an action was picked uniformly at random). The value for τ and σ_1 are discussed in the text. All the parameters were determined empirically, based on a small set of preliminary experiments.

References

[1] C.W. Anderson, Learning and problem solving with multilayer connectionist systems, PhD thesis, Computer and Information Science, University of Massachusetts, 1986.
[2] C.W. Anderson, Learning to control an inverted pendulum using neural networks, IEEE Control Systems Magazine 9 (1989) 31–37.
[3] C.W. Anderson, Q-learning with hidden-unit restarting, in: Advances in Neural Information Processing Systems, 1993, pp. 81–88.
[4] L.C. Baird, Residual algorithms: Reinforcement learning with function approximation, in: International Conference on Machine Learning, 1995, pp. 30–37.
[5] A.G. Barto, M. Duff, Monte Carlo matrix inversion and reinforcement learning, in: Advances in Neural Information Processing Systems, vol. 6, Morgan Kaufmann, 1994, pp. 687–694.
[6] A.G. Barto, R.S. Sutton, C.W. Anderson, Neuronlike adaptive elements that can solve difficult learning control problems, IEEE Transactions on Systems, Man, and Cybernetics 13 (1983) 834–846.
[7] J. Baxter, P. Bartlett, Direct gradient-based reinforcement learning: I. Gradient estimation algorithms, Technical report, Research School of Information Sciences and Engineering, Australian National University, July 1999.
[8] J. Baxter, L. Weaver, P. Bartlett, Direct gradient-based reinforcement learning: II. Gradient ascent algorithms and experiments, Technical report, Research School of Information Sciences and Engineering, Australian National University, July 1999.
[9] R.E. Bellman, Dynamic Programming, Princeton University Press, 1957.
[10] R.E. Bellman, A Markov decision process, Journal of Mathematical Mechanics 6 (1957) 679–684.
[11] R.E. Bellman, Adaptive Control Processes, Princeton University Press, 1961.
[12] H. Benbrahim, J. Franklin, Biped dynamic walking using reinforcement learning, Robotics and Autonomous Systems Journal (December 1997).
[13] D.P. Bertsekas, Dynamic Programming: Deterministic and Stochastic Models, Prentice-Hall, Inc., Upper Saddle River, NJ, 1987.
[14] D.P. Bertsekas, J.N. Tsitsiklis, Neuro-Dynamic Programming, Athena Scientific, Belmont, MA, 1996.
[15] H.-G. Beyer, H.-P. Schwefel, Evolution strategies: A comprehensive introduction, Natural Computing 1 (1) (2002) 3–52.
[16] G. Boone, Efficient reinforcement learning: Model-based Acrobot control, in: International Conference on Robotics and Automation, vol. 1, IEEE Robotics and Automation Society, Albuquerque, NM, 1997, pp. 229–234.
[17] G. Boone, Minimum-time control of the Acrobot, in: International Conference on Robotics and Automation, vol. 1, IEEE Robotics and Automation Society, Albuquerque, NM, 1997, pp. 3281–3287.
[18] J.A. Boyan, Technical update: Least-squares temporal difference learning, Machine Learning 49 (2002) 233–246.
[19] J.A. Boyan, A.W. Moore, Generalization in reinforcement learning: Safely approximating the value function, in: Advances in Neural Information Processing Systems, MIT Press, Cambridge, MA, 1995, pp. 369–376.
[20] S.J. Bradtke, A.G. Barto, Linear least-squares algorithms for temporal difference learning, Machine Learning 22 (1/2/3) (1996) 33–57.
[21] D.S. Broomhead, D. Lowe, Multivariable functional interpolation and adaptive networks, Complex Systems 2 (1988) 321–355.
[22] S.C. Brown, K.M. Passino, Intelligent control for an Acrobot, J. Intell. Robotics Syst. 18 (3) (1997) 209–248.
[23] R.H. Crites, A.G. Barto, Improving elevator performance using reinforcement learning, in: Advances in Neural Information Processing Systems, vol. 8, MIT Press, Cambridge, MA, 1996, pp. 1017–1023.
[24] P. Dayan, T. Sejnowski, TD(λ) converges with probability 1, Machine Learning 14 (1994) 295–301.
[25] M. Dorigo, M. Colombetti, Robot shaping: Developing autonomous agents through learning, Artificial Intelligence 71 (1994) 321–370.
[26] J. Farrel, T. Berger, On the effects of the training sample density in passive learning control, in: American Control Conference, 1995, pp. 872–876.
[27] B. Fritzke, Growing cell structures—a self-organizing network for unsupervised and supervised learning, Neural Networks 7 (9) (1994) 1441–1460.
[28] B. Fritzke, A growing neural gas network learns topologies, in: G. Tesauro, D.S. Touretzky, T.K. Leen (Eds.), Advances in Neural Information Processing Systems, vol. 7, MIT Press, Cambridge, MA, 1995, pp. 625–632.
[29] F. Girosi, T. Poggio, Networks and the best approximation property, Technical Report AIM-1164, Massachusetts Institute of Technology Artificial Intelligence Laboratory and Center for Biological Information Processing Whitaker College, 1989.
[30] D. Goldberg, Real-coded genetic algorithms, virtual alphabets, and blocking, Technical Report IlliGAL Report 90001, Illinois Genetic Algorithms Laboratory, Dept. of General Engineering—University of Illinois, Urbana, IL, USA, 1990.
[31] F. Gomez, J. Schmidhuber, R. Miikkulainen, Efficient non-linear control through neuroevolution, in: ECML 2006: 17th European Conference on Machine Learning, Springer, Berlin, 2006.
[32] G.J. Gordon, Stable function approximation in dynamic programming, in: International Conference on Machine Learning, Morgan Kaufmann, San Francisco, CA, 1995, pp. 261–268.
[33] G.J. Gordon, Reinforcement learning with function approximation converges to a region, in: Advances in Neural Information Processing Systems, 2000, pp. 1040–1046.
[34] C. Guestrin, M. Hauskrecht, B. Kveton, Solving factored MDPs with continuous and discrete variables, in: 20th Conference on Uncertainty in Artificial Intelligence, 2004.
[35] C. Igel, Neuroevolution for reinforcement learning using evolution strategies, in: Congress on Evolutionary Computation (CEC 2003), vol. 4, IEEE Press, 2003, pp. 2588–2595.
[36] T. Jaakkola, M.I. Jordan, S.P. Singh, On the convergence of stochastic iterative dynamic programming algorithms, Neural Computation 6 (1994).
[37] L.P. Kaelbling, M.L. Littman, A.P. Moore, Reinforcement learning: A survey, Journal of Artificial Intelligence Research 4 (1996) 237–285.

[38] M. Keijzer, J.J. Merelo, G. Romero, M.G. Schoenauer, Evolving objects: A general purpose evolutionary computation library, Artificial Evolution 2310 (2002) 231–242.
[39] M.G. Lagoudakis, R. Parr, Least-squares policy iteration, Journal of Machine Learning Research 4 (2003) 1107–1149.
[40] M.G. Lagoudakis, R. Parr, M.L. Littman, Least-squares methods in reinforcement learning for control, in: SETN, 2002, pp. 249–260.
[41] L.-J. Lin, Self-improving reactive agents based on reinforcement learning, planning and teaching, Machine Learning 8 (1992) 293–321.
[42] H.B. Mann, D.R. Whitney, On a test of whether one of 2 random variables is stochastically larger than the other, Annals of Mathematical Statistics 18 (1947) 50–60.
[43] P.H. McQuesten, Cultural enhancement of neuroevolution, PhD thesis, The University of Texas at Austin, 2002.
[44] I. Menache, S. Mannor, N. Shimkin, Basis function adaptation in temporal difference reinforcement learning, Annals of Operations Research—Special Issue on the Cross Entropy Method 134 (2005) 215–238.
[45] D. Michie, R. Chambers, BOXES: An experiment on adaptivecontrol, Machine Intelligence 2 (1968) 125–133.
[46] J.D.R. Millán, D. Posenato, E. Dedieu, Continuous-action Q-learning, Machine Learning 49 (2002) 247–265.
[47] D.E. Moriarty, R. Miikkulainen, Efficient reinforcement learning through symbiotic evolution, Machine Learning 22 (1–3) (1996) 11–32.
[48] D. Ormoneit, S. Sen, Kernel-based reinforcement learning, Machine Learning 49 (2002) 161–178.
[49] T.J. Perkins, D. Precup, A convergent form of approximate policy iteration, in: Advances in Neural Information Processing Systems, vol. 15, MIT Press, Cambridge, MA, 2003, pp. 1595–1602.
[50] J.C. Platt, A resource-allocating network for function interpolation, Neural Computation 3 (2) (1991) 213–225.
[51] T. Poggio, F. Girosi, Network for approximation and learning, Proceedings of the IEEE 78 (9) (September 1990) 1481–1497.
[52] M.J.D. Powell, Radial basis functions for multivariable interpolation: A review, in: J.C. Mason, M.G. Cox (Eds.), Algorithms for Approximation, Clarendon Press, Oxford, 1987, pp. 143–167.
[53] D. Precup, R.S. Sutton, S. Dasgupta, Off-policy temporal-difference learning with function approximation, in: 18th International Conference on Machine Learning, Morgan Kaufmann, San Francisco, CA, 2001, pp. 417–424.
[54] M.L. Puterman, Markov Decision Processes—Discrete Stochastic Dynamic Programming, Wiley-Interscience, 1994.
[55] B. Ratitch, On characteristics of Markov decision processes and reinforcement learning in large domains, PhD thesis, School of Computer Science, McGill University, Montréal, 2004.
[56] S.I. Reynolds, The stability of general discounted reinforcement learning with linear function approximation, in: UK Workshop on Computational Intelligence, 2002.
[57] G. Rummery, M. Niranjan, On-line q-learning using connectionist systems, Technical Report CUED/F-INFENG/TR 166, Cambridge University—Engineering Department, 1994.
[58] P.N. Sabes, Approximating Q-values with basis function representations, in: 1993 Connectionist Models Summer School, Lawrence Erlbaum Assoc. Inc., Hillsdale, NJ, 1993.
[59] K. Samejima, T. Omori, Adaptive internal state space construction method for reinforcement learning of a real-world agent, Neural Networks 12 (1999) 1143–1155.
[60] A.L. Samuel, Some studies in machine learning using the game of checkers, IBM Journal on Research and Development 3 (1959) 211–229.
[61] A.L. Samuel, Some studies in machine learning using the game of checkers. ii—recent advances, IBM Journal on Research and Development 11 (1967) 601–617.
[62] W. Sarle, Stopped training and other remedies for overfitting, in: Proceedings of the 27th Symposium on Interface, 1995.
[63] R. Schoknecht, A. Merke, Convergent combinations of reinforcement learning with linear function approximation, in: Advances in Neural Information Processing Systems, vol. 15, MIT Press, Cambridge, MA, 2003, pp. 1579–1586.
[64] W. Schultz, P. Dayan, P.R. Montague, A neural substrate of prediction and reward, Science 275 (1997) 1593–1599.
[65] S.P. Singh, D. Bertsekas, Reinforcement learning for dynamic channel allocation in cellular telephone systems, in: Advances in Neural Information Processing Systems, vol. 9, MIT Press, Cambridge, MA, 1997, p. 974.
[66] S.P. Singh, T. Jaakkola, M.L. Littman, C. Szepesvari, Convergence results for single-step on-policy reinforcement-learning algorithms, Machine Learning 38 (3) (2000) 287–308.
[67] S.P. Singh, R.S. Sutton, Reinforcement learning with replacing eligibility traces, Machine Learning 22 (1–3) (1996) 123–158.
[68] S.P. Singh, R.C. Yee, An upper bound on the loss from approximate optimal-value functions, Machine Learning 16 (3) (1994) 227–233.
[69] W.D. Smart, L.P. Kaelbling, Practical reinforcement learning in continuous spaces, in: International Conference on Machine Learning, 2000, pp. 903–910.
[70] M.W. Spong, The swing up control problem for the Acrobot, IEEE Control Systems Magazine 15 (1995) 49–55. Reprinted in Neurocomputing: Foundation of Research.
[71] K.O. Stanley, R. Miikkulainen, Efficient reinforcement learning through evolving neural network topologies, in: GECCO 2002: Genetic and Evolutionary Computation Conference, Morgan Kaufmann, New York, 2002, pp. 569–577.
[72] P. Stone, R.S. Sutton, Scaling reinforcement learning toward RoboCup soccer, in: 18th International Conference on Machine Learning, Morgan Kaufmann, San Francisco, CA, 2001, pp. 537–544.
[73] R. Sutton, D. McAllester, S. Singh, Y. Mansour, Policy gradient methods for reinforcement learning with function approximation, in: Advances in Neural Information Processing Systems, 2000, pp. 1057–1063.
[74] R.S. Sutton, Learning to predict by the methods of temporal differences, Machine Learning 3 (1988) 9–44.
[75] R.S. Sutton, Generalization in reinforcement learning: Successful examples using sparse coarse coding, in: Advances in Neural Information Processing Systems, vol. 8, MIT Press, Cambridge, MA, 1996, pp. 1038–1044.
[76] R.S. Sutton, A.G. Barto, Time-derivative models of Pavlovian reinforcement, in: Learning and Computational Neuroscience: Foundations of Adaptive Networks, MIT Press, Cambridge, MA, 1990, pp. 497–537.
[77] R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction, MIT Press, Cambridge, MA, 1998. A Bradford Book.

[78] V. Tadić, On the convergence of temporal-difference learning with linear function approximation, Machine Learning 42 (3) (2001) 241–267.
[79] G.J. Tesauro, TD-Gammon, a self-teaching backgammon program achieves master-level play, Neural Computation 6 (2) (1994) 215–219.
[80] S. Thrun, A. Schwartz, Issues in using function approximation for reinforcement learning, in: Fourth Connectionist Models Summer School, Lawrence Erlbaum Associates, Hillsdale, NJ, 1993.
[81] J.N. Tsitsiklis, B. Van Roy, Feature-based methods for large scale dynamic programming, Machine Learning 22 (1996) 59–94.
[82] J.N. Tsitsiklis, B. Van Roy, An analysis of temporal-difference learning with function approximation, IEEE Transactions on Automatic Control 42 (May 1997) 674–690.
[83] C. Watkins, Learning from delayed rewards, PhD thesis, University of Cambridge, England, 1989.
[84] C. Watkins, P.D. Dayan, Q-learning, Machine Learning 8 (1992) 279–292.
[85] S.E. Weaver, L.C. Baird, M.M. Polycarpou, Preventing unlearning during on-line training of feedforward networks, in: International Symposium of Intelligent Control, Gaithersburg, 1998, pp. 359–364.
[86] D.A. White, D.A. Sofge, Handbook of Intelligence Control, Neural, Fuzzy and Adaptive Approaches, Van Nostrand Reinhold, New York, 1992. Chapter: Applied learning: Optimal control for manufacturing.
[87] D. Whitley, M. Richards, R. Beveridge, A. da Motta Salles Barreto, Alternative evolutionary algorithms for evolving programs: Evolution strategies and steady-state GP, in: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (GECCO 2006), vol. 1, ACM Press, Seattle, Washington, 2006, pp. 919–926. Winner best GP paper.
[88] D. Whitley, S. Dominic, R. Das, C.W. Anderson, Genetic reinforcement learning for neurocontrol problems, Machine Learning 13 (2–3) (1993) 259–284.
[89] R.J. Williams, L.C. Baird, Tight performance bounds on greedy policies based on imperfect value functions, Technical Report NU-CCS-93-14, Northeastern University, November 1993.
[90] W. Zhang, T.G. Dietterich, A reinforcement learning approach to job-shop scheduling, in: International Joint Conference on Artificial Intelligence, 1995.
[91] D.Z. Zhao, J. Yi, GA-based control to swing up an Acrobot with limited torque, Transactions of the Institute of Measurement and Control 28 (1) (2006) 3–13.