# Reinforcement Learning Algorithms for MDPs

Csaba Szepesvári

June 7, 2010*

**Abstract**

Reinforcement learning is a learning paradigm concerned with learning to control a system so as to maximize a numerical performance measure that expresses a long-term objective. The goal in reinforcement learning is to develop efficient learning algorithms, as well as to understand the algorithms' merits and limitations. In this article we focus on a few selected algorithms of reinforcement learning which build on the powerful theory of dynamic programming.

**Keywords:** *reinforcement learning; Markov Decision Processes; temporal difference learning; stochastic approximation; function approximation; stochastic gradient methods; least-squares methods; Q-learning; actor-critic methods; policy gradient; natural gradient*

## 1 Overview

Reinforcement learning (RL) refers to both a learning problem and a subfield of machine learning. As a learning problem it refers to learning to control a system so as to maximize some numerical value which represents a long-term objective. A typical setting where reinforcement learning operates is shown in Figure 1: A controller receives the controlled system's state and a reward associated with the last state transition. It then calculates an action which is sent back to the system. In response, the system makes a transition to a new state and the cycle is repeated. The problem is to learn a way of controlling the system so as to maximize the total reward. The learning problems differ in the details of how the data is collected and how performance is measured.

In this article we assume that the system that we wish to control is stochastic. Further, we assume that the measurements available on the system's state are detailed enough so that the the controller can avoid reasoning about how to collect information about the state. Problems with these characteristics are best described in the framework of Markovian Decision Processes (MDPs). The standard approach to 'solve' MDPs is to use dynamic programming, which transforms the problem of finding a good controller into the problem of finding a good value function. However, apart from the simplest cases when the MDP has very few states and actions, dynamic programming is infeasible. The RL algorithms that

*Last update: June 25, 2010

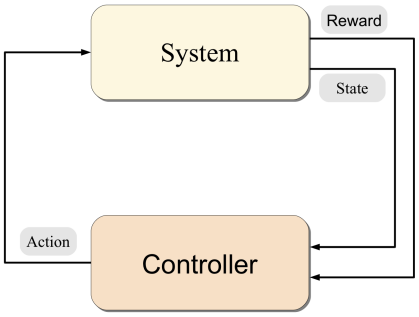


Figure 1: The basic reinforcement learning scenario

we discuss here can be thought of as a way of turning the infeasible dynamic programming methods into practical algorithms so that they can be applied to large-scale problems.

There are two key ideas that allow RL algorithms to achieve this goal. The first is to use samples to compactly represent the dynamics of the control problem. This is important for two reasons: First, it allows one to deal with learning scenarios when the dynamics is unknown. Second, even if the dynamics is available, exact reasoning that uses it might be intractable on its own. The second key idea behind RL algorithms is to use powerful function approximation methods to compactly represent value functions. The significance of this is that it makes dealing with large, high-dimensional state- and/or action-spaces possible. What is more, the two ideas fit nicely together: Samples may be focused on a small subset of the spaces they belong to, which clever function approximation techniques might exploit. It is the understanding of the interplay between dynamic programming, samples and function approximation that is at the heart of designing, analyzing and applying RL algorithms.

The interested reader is invited to learn more about RL by consulting the rich literature. The first survey that appeared on RL is due to Kaelbling et al. [46], which was followed by the publication of the books by Bertsekas and Tsitsiklis [14] and Sutton and Barto [85]. The book by Bertsekas and Tsitsiklis [14] discusses the theoretical foundations, while the book by Sutton and Barto [85] focuses on presenting the core ideas in an accessible manner. More recent books that are devoted in part or in whole to the subject include those by Gosavi [41], Cao [25], Bertsekas [9, 10], Powell [69], Chang et al. [27], Busoniu et al. [24] and Szepesvári [90]. Online, periodically updated summaries are provided by Bertsekas [11] and Szepesvári [89].

The rest of this article is organized according to Figure 1. After introducing our notation (Section 2), we consider value prediction problems (Section 3). We describe TD($\lambda$) and its recent improved versions, along with least-squares methods. In the next section (Section 4) we discuss of a few algorithms for control, most notably Q-learning and fitted Q-iteration, which can be thought of as implementing value iteration. This is followed by a description of an actor-critic algorithm, which can be thought of as implementing policy iteration. Due to the lack of space the discussion of (direct) policy search methods is not included.

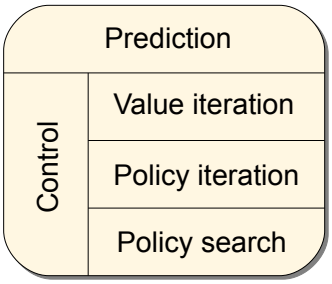


Figure 2: Types of reinforcement problems and approaches. The article gives examples of algorithms for all approaches except for (direct) policy search.

# 2 Preliminaries

We assume that the reader is familiar with Markov Decision Processes. Accordingly, the purpose of this section is to introduce the notation used in the rest of the article.
We consider controlled stochastic process with Markovian dynamics:

$$\begin{aligned} X_{t+1} &= f_1(X_t, A_t, D_{t+1}), \\ R_{t+1} &= f_2(X_t, A_t, D_{t+1}), \quad t = 0, 1, \dots . \end{aligned} \tag{1}$$

Here $X_t \in \mathcal{X}$ is the state of the system at time $t$, $A_t \in \mathcal{A}$ is the action taken by the controller after $X_t$ was observed, $D_{t+1}$ is a disturbance term (taking values in some Euclidean space) and $R_{t+1} \in \mathbb{R}$ is the reward received.[1] The disturbance sequence $(D_t; t \geq 1)$ is a sequence of independent, identically distributed (i.i.d.) random variables. The goal of the controller is to maximize the expected total discounted reward, or *expected return*, irrespective of the initial state:

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \,|\, X_0 = x\right] \to \max!, \quad x \in \mathcal{X}.$$

Here $0 < \gamma < 1$ is the so-called *discount factor*. The process $((X_t, A_t, R_{t+1}); t \geq 0)$ is called a *Markov Decision Process* (MDP).[2] In the applications $f$ is usually a smooth function of its arguments.
The equations (1) specify an MDP in the so-called state-space form. An equivalent definition can be given in terms of a transition probability kernel $\mathcal{P}_0$ which assigns to each state-action pair $(x, a) \in \mathcal{X} \times \mathcal{A}$ a probability measure over $\mathcal{X} \times \mathbb{R}$. Denoting the latter by $\mathcal{P}_0(\cdot\,|x, a)$, the connection between the two forms is given by

$$\mathcal{P}_0(U|x, a) = \mathbb{P}\left(f(x, a, D) \in U\right),$$

where $f(x, a, d) = (f_1(x, a, d), f_2(x, a, d))$ and $D$ is a random variable whose distribution is the same as that of the distribution underlying $D_t$.

[1] Here we assumed that all actions are admissible in all states. Sometimes it is necessary to restrict the set of actions admissible is restricted to a subset of the actions in a state-dependent manner. The results and algorithms presented here extend to this case essentially without any change.

[2] The algorithms described below has been extended to other than the expected total discounted reward, such as the expected total reward (i.e., no discounting), or the average reward criteria. The discounted criterion is selected because it allows a simpler presentation of the main ideas.

For simplicity, consider the case when the action set $\mathcal{A}$ is countable. A general policy determines the actions chosen as a function of history. The *value* of state $x$ under a policy $\pi$ is the expected return given that the policy is started in state $x$, while the *action-value* of a state-action pair $(x, a)$ is the expected return given that the process is started from state $x$, the first action is $a$ after which the policy $\pi$ is followed. The value of state $x$ under policy $\pi$ is denoted by $V^\pi(x)$, while the action-value of the pair $(x, a)$ under policy $\pi$ is denoted by $Q^\pi(x, a)$.

The *optimal value* of a state is the value of the best possible expected return that can be obtained from that state. It is denoted by $V^*(x)$: $V^*(x) = \sup_\pi V^\pi(x)$. Similarly, the optimal value of a state-action pair is $Q^*(x, a) = \sup_\pi Q^\pi(x, a)$. A policy is called *optimal* if $V^\pi(x) = V^*(x)$ holds for all states $x \in \mathcal{X}$. A policy is called *stationary* if the actions selected at any step depend only on the last state. A stationary policy can thus be specified by specifying a distribution $\pi(\cdot|x)$ over actions for each state of the state space. Under mild regularity assumptions (e.g., assuming that $\sup_{x,a} |\mathbb{E}[f_2(x, a, D)]| < +\infty$ and e.g. when $\mathcal{A}$ is finite) an optimal stationary policy exists and in fact any stationary policy $\pi$ that satisfies $\sum_{a\in\mathcal{A}} \pi(a|x)Q^*(x, a) = \max_{a\in\mathcal{A}} Q^*(x, a)$ for every state $x \in \mathcal{X}$ is optimal. Given some function $Q : \mathcal{X} \times A \to \mathbb{R}$, a maximizer of $Q(x, \cdot)$ is called a *greedy action* with respect to (w.r.t.) $Q$. A policy $\pi$ is greedy w.r.t. $Q$ if it is greedy w.r.t. $Q$ in every state $x \in \mathcal{X}$.

## 2.1 Dynamic programming algorithms

*Value-* and *policy-iteration* are the standard algorithms of dynamic programming to find optimal (or near-optimal) policies in MDPs.

*Value iteration* generates a sequence of value functions $V_{k+1} = T_1^* V_k, \quad k \geq 0$, where $V_0$ is arbitrary. Thanks to Banach's fixed-point theorem, $(V_k; k \geq 0)$ converges to $V^*$ at a geometric rate. Here the operator $T_1^* : B(\mathcal{X}) \to B(\mathcal{X})$ is defined by $T_1^* V(x) = \sup_{a\in\mathcal{A}} \mathbb{E}[f_2(x, a, D) + \gamma V(f_1(x, a, D))]$, where $B(\mathcal{X})$ is the set of bounded functions over $\mathcal{X}$. The operator $T_1^*$ be shown to be a $\gamma$-contraction w.r.t. the supremum norm.

Value iteration can also be used in conjunction with action-value functions, in which case it takes the form $Q_{k+1} = T_2^* Q_k, \quad k \geq 0$, which again converges to $Q^*$ at a geometric rate. Here the operator $T_2^* : B(\mathcal{X} \times \mathcal{A}) \to B(\mathcal{X} \times \mathcal{A})$ is defined by $T_2^* Q(x, a) = \mathbb{E}[f_2(x, a, D) + \gamma \sup_{a'\in\mathcal{A}} Q(f_1(x, a, D), a')]$ and is again a contraction. Both $T_1^*$ and $T_2^*$ are called *Bellman-optimality operators*.

It can be shown that once $V_k$ (or $Q_k$) is close to $V^*$ (resp., $Q^*$), a policy that is greedy with respect to $V_k$ (resps., $Q_k$) will be close-to-optimal. In particular, the following bound is known to hold [e.g., 82, Corollary 2]:

$$V^\pi(x) \geq V^*(x) - \frac{2}{1-\gamma} \|Q - Q^*\|_\infty, \quad x \in \mathcal{X}. \tag{2}$$

Here $\pi$ is any policy that is greedy policy w.r.t. $Q$ and $Q$ is an arbitrary real-valued function over the state-action space.

Let us now consider *policy iteration*. Fix an arbitrary initial policy $\pi_0$. At iteration $k > 0$, compute the action-value function underlying $\pi_k$ (this is called the policy evaluation step). Next, given $Q^{\pi_k}$, define $\pi_{k+1}$ as a policy that is greedy with respect to $Q^{\pi_k}$ (this is called the policy improvement step). The steps when $\pi_{k+1}$ is computed from $\pi_k$ define an update of *policy iteration*.

After $k$ iterations, policy iteration gives a policy not worse than the policy that is greedy w.r.t. to the value function computed using $k$ iterations of value iteration provided that the two procedures were started with the same initial value function. However, the computational cost of a single update in policy iteration is much higher (because of the policy evaluation step) than that of one update in value iteration.
For more information on MDPs consult Bertsekas and Shreve [13] or Puterman [71].

# 3 Value prediction problems

Value-prediction is the problem of predicting the value of states (or state-action pairs) under a stationary policy $\pi$. The data is given in the form of a series of transitions, $\mathcal{D} = ((X_t, R_{t+1}, Y_{t+1}); t = 0, 1, \ldots)$, where $(Y_{t+1}, R_{t+1}) = f(X_t, A_t, D_{t+1})$ and $A_t \sim \pi(\cdot|X_t)$. The goal is to learn a predictor for $V^\pi(\cdot)$. Here, the process generating $X_t$ is either left unspecified, or $Y_{t+1} = X_{t+1}$, i.e., the data is given in the form of a single trajectory. When $X_t$ has a limiting distribution which corresponds to the stationary distribution of $\pi$ (which is assumed to exist), we talk about an *on-policy setting*, otherwise we talk about an *off-policy setting*. The off-policy setting comes up for example when one uses a simulator and time-to-time the states $X_t$ are reset to specific states of interest. The on-policy setting comes up when such "resetting" of the state is not available, as in the case when learning happens while interacting with a real system.

## 3.1 TD($\lambda$) with linear function approximation

When the state space is large, it is infeasible to keep an estimate of the values of all states (the array storing the values may not fit into the memory). One way to get around this problem is to use a function approximation method, a special case of which is when a *linear function approximation method* is used. Such a method approximates the value at a state $x$ by first calculating some features $\varphi(x) \in \mathbb{R}^d$ and then linearly combining these with some weights $\theta$:

$$V^\pi(x) \approx \theta^\top \varphi(x).$$

The features are usually designed (picked) by hand so that they capture the properties of the state that the user thinks are important for predicting the values. Designing the features usually needs prior knowledge, though one can always resort to some general constructions, like tile coding or radial basis functions. Once the features are selected we are left with finding the weights so that $V^\pi$ is well-approximated. This is the job of the learning algorithms. Although it might seem a lot to ask the user to select the features, that the weights are tuned automatically is a major achievement whose significance should not be underestimated.
The TD($\lambda$) algorithm of Sutton [83] (see also, Sutton 84) is one method that allows one to tune the weights in an incremental manner. The pseudocode of the update rule for TD($\lambda$) is shown as Algorithm 1. The routine shown must be called for each sampled transition. The name of the algorithm is derived from the abbreviation of temporal differences and it comes from line 1, where $\delta$, which can be thought of as a temporal difference error is computed. In addition to the data underlying a transition, the method expects the current weights $\theta$, and the current *eligibility traces* $z$ as inputs. Here, both $\theta$ and $z$ are $d$-dimensional vectors. The routine then returns the updated values of these vectors.

**Algorithm 1** The function implementing the TD($\lambda$) algorithm with linear function approximation. This function must be called after each transition.

**function** TDLambdaLinFApp($X, R, Y, \theta, z$)
**Input:** $X$ is the last state, $Y$ is the next state, $R$ is the immediate reward associated with this transition, $\theta \in \mathbb{R}^d$ is the parameter vector of the linear function approximation, $z \in \mathbb{R}^d$ is the vector of eligibility traces
1: $\delta \leftarrow R + \gamma \cdot \theta^\top \varphi[Y] - \theta^\top \varphi[X]$
2: $z \leftarrow \varphi[X] + \gamma \cdot \lambda \cdot z$
3: $\theta \leftarrow \theta + \alpha \cdot \delta \cdot z$
4: **return** $(\theta, z)$

The tuning parameters of TD($\lambda$) are $\lambda \in [0, 1]$ and the step-size $\alpha \geq 0$. In practice, one often uses $d$ step-sizes, one for each component of the weight vector. The size of the step-sizes should match the expected magnitude of the feature components. The $\lambda$ parameter determines the amount of "bootstrapping". When the dynamics is well-preserved by the feature extraction method (e.g., $\varphi(Y_{t+1})$ and the reward can be well predicted given $\varphi(X_t)$ and the action) then a small value of $\lambda$ is appropriate (and the algorithm can be thought of as implementing an incremental form of value iteration). In the opposite case, $\lambda \approx 1$ is more appropriate and the algorithm can be thought of as implementing a Monte-Carlo algorithm. In general, increasing $\lambda$ increases the variance but decreases the bias. The best value of $\lambda$ is problem dependent and is usually found by trial and error in small scale experiments before the algorithm is run on a larger chunk of data. When $\lambda = 0$, the algorithm is also called TD(0), while it is also called TD(1) when $\lambda = 1$.
Almost sure convergence of the weights is guaranteed provided that *(i)* the stochastic process $(X_t; t \geq 0)$ is an ergodic Markov process whose stationary distribution $\mu$ is the same as the stationary distribution of the policy $\pi$; and *(ii)* the step-size sequence satisfies the so-called Robbins-Monro (RM) conditions [97; 14, p. 222, Section 5.3.7].[3] When the first condition is not met, i.e., under off-policy sampling, convergence is no longer be guaranteed, but, in fact, the parameters may diverge (see, e.g., 14, Example 6.7, p. 307). This is true for linear function approximation when the distributions of $(X_t; t \geq 0)$ do not match the stationary distribution of $\pi$. The TD($\lambda$) method has a version that is suitable for tuning the parameters of nonlinear function approximation methods, such as neural networks. However, this standard version might diverge (see, e.g., 14, Example 6.6, p. 292). For further examples of instability, see Baird [8], Boyan and Moore [21].

### 3.1.1 Gradient temporal difference learning

In this section we present two algorithms which overcome the instability of TD($\lambda$) while retaining its incremental nature and the computational efficiency of the updates.
The key is to introduce an objective function and derive a gradient method from it. For simplicity, we consider the case when $\lambda = 0$. Define the objective function

$$J(\theta) = \mathbb{E}\left[\delta_{t+1}(\theta)\varphi_t\right]^\top \mathbb{E}\left[\varphi_t \varphi_t^\top\right]^{-1} \mathbb{E}\left[\delta_{t+1}(\theta)\varphi_t\right], \tag{3}$$

[3] The RM conditions state the following: if $(\alpha_t; t \geq 0)$ are the step-sizes used in time step $t$ then $\sum_{t=0}^{\infty} \alpha_t = \infty$ and $\sum_{t=0}^{\infty} \alpha_t^2 < \infty$. They are satisfied by e.g. $\alpha_t = 1/t$. In practice, people often use small constant step-sizes or adaptive step-size selection methods.

where

$$\begin{aligned}\delta_{t+1}(\theta) &= R_{t+1} + \gamma V_\theta(Y_{t+1}) - V_\theta(X_t) \\ &= R_{t+1} + \gamma\theta^\top\varphi'_{t+1} - \theta^\top\varphi_t, \\ \varphi_t &= \varphi(X_t), \\ \varphi'_{t+1} &= \varphi(Y_{t+1}).\end{aligned} \tag{4}$$

Here, for simplicity we assumed that $(X_t; t \geq 0)$ is a stationary process, so the expectation in (3) is well-defined.
When TD(0) converges, it converges to a unique vector $\theta^*$ that satisfies

$$\mathbb{E}\left[\delta_{t+1}(\theta^*)\varphi_t\right] = 0. \tag{5}$$

In this case, the minimizer of $J$ will coincide with $\theta^*$. Therefore, it suffices to design an algorithm to minimize $J$. Accordingly, from now on let us redefine $\theta^*$ as the minimizer of $J$ (for simplicity, assume that the minimizer is unique).
Taking the gradient of $J$ we get

$$\nabla_\theta J(\theta) = -2\mathbb{E}\left[(\varphi_t - \gamma\varphi'_{t+1})\varphi_t^\top\right] w(\theta), \tag{6}$$

where

$$w(\theta) = \mathbb{E}\left[\varphi_t\varphi_t^\top\right]^{-1}\mathbb{E}\left[\delta_{t+1}(\theta)\varphi_t\right].$$

Let us introduce two sets of weights: $\theta_t$ to approximate $\theta_*$ and $w_t$ to approximate $w(\theta_*)$. In GTD2 ("gradient temporal difference learning, version 2"), the update of $\theta_t$ is chosen to follow the negative stochastic gradient of $J$ based on (6) assuming that $w_t \approx w(\theta_t)$, while the update of $w_t$ is chosen so that for any fixed $\theta$, $w_t$ would converge almost surely to $w(\theta)$. The pseudocode of the resulting update, which Sutton et al. [87] called the GTD2 update, is shown as Algorithm 2. This algorithm needs two sets of step-sizes $(\alpha, \beta)$.
Sutton et al. [87] have shown that (under the standard RM conditions on the step-sizes and some mild regularity conditions) the weight-sequence $(\theta_t)$ updated with GTD2 converges to the minimizer of $J(\theta)$ almost surely. However, unlike in the case of TD(0), convergence is guaranteed independently of the distribution of $(X_t; t \geq 0)$. At the same time, the update of GTD2 costs only twice as much as the cost of TD(0).
One can arrive at another gradient method if the gradient of $J$ is written as

$$\nabla_\theta J(\theta) = -2\Big(\mathbb{E}\left[\delta_{t+1}(\theta)\varphi_t\right] - \gamma\mathbb{E}\left[\varphi'_{t+1}\varphi_t^\top\right] w(\theta)\Big).$$

This suggests replacing the update in line 5 by

$$\theta \leftarrow \theta + \alpha \cdot (\delta \cdot f - \gamma \cdot a \cdot f'),$$

giving rise to the TDC ("temporal difference learning with corrections") update of Sutton et al. [87]. Here the update of $w_t$ must use larger step-sizes than the update of $\theta_t$: $\alpha_t = o(\beta_t)$. This makes TDC a member of the family of the so-called *two-timescale stochastic approximation algorithms* [15, 16]. If, in addition to this, the standard RM conditions are also satisfied by both step-size sequences, $\theta_t \to \theta_*$ holds again almost surely [87].

**Algorithm 2** The function implementing the GTD2 algorithm. This function must be called after each transition.

**function** GTD2($X, R, Y, \theta, w$)

**Input:** $X$ is the last state, $Y$ is the next state, $R$ is the immediate reward associated with this transition, $\theta \in \mathbb{R}^d$ is the parameter vector of the linear function approximation, $w \in \mathbb{R}^d$ is the auxiliary weight

$f \leftarrow \varphi[X]$
$f' \leftarrow \varphi[Y]$
$\delta \leftarrow R + \gamma \cdot \theta^\top f' - \theta^\top f$
$a \leftarrow f^\top w$
$\theta \leftarrow \theta + \alpha \cdot (f - \gamma \cdot f') \cdot a$
$w \leftarrow w + \beta \cdot (\delta - a) \cdot f$
**return** $(\theta, w)$

More recently, these algorithms have been extended to nonlinear function approximation [59] and to the $\lambda > 0$ case [58].

Note that although these algorithms are derived from the gradient of an objective function, they are not true stochastic gradient methods in the sense that the expected weight update direction can be different from the direction of the negative gradient of the objective function. In fact, these methods belong to the larger class of pseudo-gradient methods. The two methods differ in how they approximate the gradients. It remains to be seen whether one of them is better than the other.

### 3.1.2 LSTD: Least-squares temporal difference learning

As said earlier, in the limit, when TD(0) converges it converges to the solution of (5). Given a finite sample

$$\mathcal{D}_n = ((X_0, R_1, Y_1), (X_1, R_2, Y_2), \ldots, (X_{n-1}, R_n, Y_n)),$$

one can approximate (5) by

$$\frac{1}{n} \sum_{t=0}^{n-1} \varphi_t \, \delta_{t+1}(\theta) = 0. \tag{7}$$

Plugging in $\delta_{t+1}(\theta) = R_{t+1} - (\varphi_t - \gamma \varphi'_{t+1})^\top \theta$, we see that this equation is linear in $\theta$. In particular, if the matrix $\hat{A}_n = \frac{1}{n} \sum_{t=0}^{n-1} \varphi_t (\varphi_t - \gamma \varphi'_{t+1})^\top$ is invertible, the solution is simply

$$\theta_n = \hat{A}_n^{-1} \hat{b}_n, \tag{8}$$

where $\hat{b}_n = \frac{1}{n} \sum_{t=0}^{n-1} R_{t+1} \varphi_t$. If inverting the matrix can be afforded (e.g., the dimensionality of the features is not too large and the method is not called too many times) this method can give a better approximation to $\theta^*$ than TD(0) or some other incremental first-order method since the latter are negatively impacted by the eigenvalue spread of the matrix $A = \mathbb{E}\left[\hat{A}_n\right]$. The idea of directly computing the solution of (7) is due to Bradtke and Barto [22], who call the resulting algorithm *least-squares temporal difference* learning or LSTD. Using the terminology of stochastic programming, LSTD can be seen to use *sample average approximation* [78]. In the terminology of statistics, it belongs to the so-called $Z$-estimation family of procedures [e.g., 54, Section 2.2.5].

**Algorithm 3** The function implementing the RLSTD algorithm. This function must be called after each transition. Initially, $C$ should be set to a diagonal matrix with small positive diagonal elements: $C = \beta\, I$, with $\beta > 0$.

**function** RLSTD$(X, R, Y, C, \theta)$
**Input:** $X$ is the last state, $Y$ is the next state, $R$ is the immediate reward associated with this transition, $C \in \mathbb{R}^{d\times d}$, and $\theta \in \mathbb{R}^d$ is the parameter vector of the linear function approximation
$f \leftarrow \varphi[X]$
$f' \leftarrow \varphi[Y]$
$g \leftarrow (f - \gamma f')^\top C$ ▷ $g$ is a $1 \times d$ row vector
$a \leftarrow 1 + gf$
$v \leftarrow Cf$
$\delta \leftarrow R + \gamma \cdot \theta^\top f' - \theta^\top f$
$\theta \leftarrow \theta + \delta \,/\, a \cdot v$
$C \leftarrow C - v\, g \;/\, a$
**return** $(C, \theta)$

Using the Sherman-Morrison formula, one can derive an incremental version of LSTD, analogously to how the recursive least-squares (RLS) method is derived in adaptive filtering [103]. The resulting algorithm is called "recursive LSTD" (RLSTD) [22] and is shown as Algorithm 3. The computational complexity of one update is $O(d^2)$. When the number of samples $n$ is large, it can be faster to use the direct form of LSTD that uses matrix inversion. Boyan [19] extended LSTD to incorporate the $\lambda$ parameter of TD($\lambda$) and called the resulting algorithm LSTD($\lambda$). (Note that for $\lambda > 0$ to make sense one needs $X_{t+1} = Y_{t+1}$; otherwise, the TD errors do not telescope). The recursive form of LSTD($\lambda$), RLSTD($\lambda$), has been studied by Xu et al. [104] and (independently) by Nedič and Bertsekas [61].
An alternative to LSTD (and LSTD($\lambda$)) is $\lambda$-*least squares policy evaluation* ($\lambda$-LSPE for short) due to Bertsekas and Ioffe [12], which implements a form of fitted value iteration.

**Comparing least-squares and TD-like methods.** To compare the two approaches fix some time $T$ available for computation and assume that samples are cheap to obtain (e.g., an efficient simulator is available). In time $T$, the least-squares methods are limited to process a sample of size $n \approx T/d^2$, while the lightweight methods can process a sample of size $n' \approx nd$. Let us now look at the precision of the resulting parameters. Assume that the limit of the parameters is $\theta_*$. Denote by $\theta_t$ the parameter obtained by (say) LSTD after processing $t$ observations and denote by $\theta'_t$ the parameter obtained by a TD-method. Then, one expects that $\|\theta_t - \theta_*\| \approx C_1 t^{-\frac{1}{2}}$ and $\|\theta'_t - \theta_*\| \approx C_2 t^{-\frac{1}{2}}$. Thus,

$$\frac{\|\theta'_{n'} - \theta_*\|}{\|\theta_n - \theta_*\|} \;\approx\; \frac{C_2}{C_1}\, d^{-\frac{1}{2}}. \tag{9}$$

Hence, if $C_2/C_1 < d^{1/2}$ then the lightweight TD-like method will achieve a better accuracy, while in the opposite case the least-squares procedures will perform better. As usual, it is difficult to decide this *a priori*. As a rule of thumb, based on (9) we expect that when $d$ is relatively small, least-squares methods might be converging faster, while if $d$ is large then the lightweight, incremental methods will give better results. Notice that this analysis is not

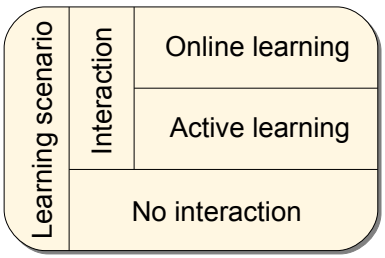


Figure 3: Types of reinforcement problems

specific to reinforcement learning methods, but it applies in all cases when an incremental lightweight procedure is compared to a least-squares-like procedure and samples are cheap (for a similar analysis in a supervised learning problem see, e.g., 18). One attempt to marry the advantages of the computational efficiency of incremental methods and the statistical efficiency of least-squares methods is described by Geramifard et al. [39].

# 4 Control

Figure 4 shows the basic types of control learning problems. The first criterion that the space of problems is split upon is whether the learner can actively influence the observations. In case she can, we talk about *interactive learning*, otherwise one is facing a *non-interactive learning* problem. Interactive learning is potentially easier since the learner has the additional option to influence the distribution of the sample. However, the goal of learning, as we will see it soon, is usually different in the two cases, making these problems incomparable in general.

In the case of non-interactive learning the natural goal is to find a good policy given the observations. A common situation is when the sample is fixed. For example, the sample can be the result of some experimentation with some physical system that happened before learning started. In machine learning terms, this corresponds to *batch learning*. (Batch learning problems are not to be confused with batch learning methods, which are the opposite of incremental, a.k.a. recursive or iterative methods.) Since the observations are uncontrolled, the learner working with a fixed sample has to deal with an off-policy learning situation. In other cases the learner can ask for more data (i.e., when a simulator is used to generate new data). Here the goal might be to learn a good policy as quickly as possible.

Consider now interactive learning. One possibility is that learning happens while interacting with a real system in a closed-loop fashion. A reasonable goal then is to optimize *online performance*, making the learning problem an instance of *online learning*. In online learning the key is to explore in a clever manner. Online performance can be measured in different ways. A natural measure is to use the sum of rewards incurred during learning. If this is offset by the expected sum of rewards under the optimal policy, one gets the notion of regret. Auer et al. [6] describes a state-of-the-art algorithm, UCRL2, that works in finite MDPs and gives an overview of previous results. An alternative cost measure is the number of times the learner's future expected return falls short of the optimal return, i.e., the number of times the learner commits a "mistake" Kakade et al. [48]. Szita and Szepesvári [92] reviews

the literature on this and describes a state-of-the-art algorithm, MorMax, that commits a polynomial number of "mistakes" in finite MDPs. Another possible goal is to produce a well-performing policy as soon as possible (or find a good policy given a finite number of samples), just like in non-interactive learning. As opposed to the non-interactive situation, however, here the learner has the option to control the samples so as to maximize the chance of finding such a good policy. This learning problem is an instance of *active learning*. The $E^3$-algorithm of Kearns and Singh [50] explores an unknown MDP and stops when it knows a good policy for the state *just visited*. $E^3$ needs a polynomial number of interactions and uses poly-resources in the relevant parameters of the problem before it stops. In a follow-up work, Brafman and Tennenholtz [23] introduced the R-max algorithm which refines the $E^3$ algorithm and proved similar results.

There exists only a few experimental studies in online learning. Jong and Stone [45] proposed a method that can be interpreted as a practical implementation of the ideas in Kakade et al. [48], while Nouri and Littman [64] experimented with multi-resolution regression trees and the so-called fitted Q-iteration algorithm (reviewed in the next section). The main message of these works is that explicit exploration control can indeed be beneficial.

When a simulator is available, the learning algorithms can be used to solve *planning problems*. In planning online learning becomes irrelevant and the algorithms' running time and memory requirements become the primary concern. Kearns et al. [49], Szepesvári [88], Kocsis and Szepesvári [51] and Chang et al. [27] discuss some ideas for planning.

When the state or action space is large, one needs to resort to function approximation. Therefore, in what follows we focus on the core algorithmic problem of learning good controllers in large spaces when a function approximation method is used and neglect the exploration issue. With this we do not want to imply that clever exploration methods are unimportant, but we merely make the point that the clever methods will need to work with efficient methods that use function approximation.

## 4.1 Direct methods

Direct methods aim at approximating the optimal action-value function directly. One class of methods, two members of which are reviewed here, can be thought of as implementing a form of value iteration with action-value functions.

### 4.1.1 $Q$-learning with function approximation

Assume that we are given features $\varphi(x,a) \in \mathbb{R}^d$ of state-action pairs. The problem is to find weights $\theta \in \mathbb{R}^d$ such that $Q^*(x,a) \approx \theta^\top \varphi(x,a)$ holds for all $(x,a) \in \mathcal{X} \times \mathcal{A}$. The $Q$-learning algorithm by Watkins [102], whose update is shown as Algorithm 4, can be thought of as the extension of TD($\lambda$) to this case.

Although $Q$-learning is widely used in practice, little can be said about its convergence properties. In fact, since TD($\lambda$) is a special case of this algorithm (when there is only one action for every state), just like TD($\lambda$), this update rule will also fail to converge when off-policy sampling or nonlinear function approximation is used (cf. Section 3.1).

**Fitted $Q$-iteration** Fitted $Q$-iteration is an instance of the generic fitted value iteration recipe. Given the previous iterate, $Q_t$, the idea is to form a Monte-Carlo approximation to

**Algorithm 4** The function implementing the $Q$-learning algorithm with linear function approximation. This function must be called after each transition.

**function** QLEARNINGLINFAPP($X, A, R, Y, \theta$)
**Input:** $X$ is the last state, $Y$ is the next state, $R$ is the immediate reward associated with this transition, $\theta \in \mathbb{R}^d$ parameter vector
$\delta \leftarrow R + \gamma \cdot \max_{a' \in \mathcal{A}} \theta^\top \varphi[Y, a'] - \theta^\top \varphi[X, A]$
$\theta \leftarrow \theta + \alpha \cdot \delta \cdot \varphi[X, A]$
**return** $\theta$

**Algorithm 5** The function implementing one iteration of the fitted $Q$-iteration algorithm. The function must be called until some criterion of convergence is met. The methods PREDICT and REGRESS are specific to the regression method chosen. The method PREDICT($z, \theta$) should return thepredicted value at the input $z$ given the regression parameters $\theta$, while REGRESS($S$), given a list of input-output pairs $S$, should implement a regression algorithm that solves the regression problem given by $S$ and returns new parameters that can be used in PREDICT.

**function** FITTEDQ($D, \theta$)
**Input:** $D = ((X_i, A_i, R_{i+1}, Y_{i+1}); i = 1, \ldots, n)$ is a list of transitions, $\theta$ are the regressor parameters
$S \leftarrow []$ ▷ Create empty list
**for** $i = 1 \to n$ **do**
  $T \leftarrow R_{i+1} + \max_{a' \in \mathcal{A}} \text{PREDICT}((Y_{i+1}, a'), \theta)$ ▷ Target at $(X_i, A_i)$
  $S \leftarrow \text{APPEND}(S, \langle (X_i, A_i), T \rangle)$
**end for**
$\theta \leftarrow \text{REGRESS}(S)$
**return** $\theta$

$T^*Q_t\,(x, a)$ at select state-action pairs and then learn a regressor based on the so-created dataset. Algorithm 5 shows the corresponding pseudocode. This method must be called repeatedly in a loop until convergence.

It is known that fitted $Q$-iteration might diverge unless a special regressor is used [8, 21, 96]. Ormoneit and Sen [65] suggest to use kernel averaging, while Ernst et al. [36] suggest using tree based regressors. These are guaranteed to converge (say, if the same data is fed to the algorithm in each iteration) as they implement local averaging and as such results of Gordon [40], Tsitsiklis and Van Roy [96] are applicable to them. Riedmiller [72] reports good empirical results with neural networks, at least when new observations obtained by following a policy greedy with respect to the latest iterate are incrementally added to the set of samples used in the updates. That the sample is changed is essential if no good initial policy is available, i.e., when in the initial sample states which are frequently visited by "good" policies are underrepresented (a theoretical argument for why this is important is given by Van Roy [100] in the context of state aggregation).

Antos et al. [5] and Munos and Szepesvári [60] prove finite-sample performance bounds that apply to a large class of regression methods that use empirical risk minimization over a fixed space $\mathcal{F}$ of candidate action-value functions. Their bounds depend on the *worst-case*

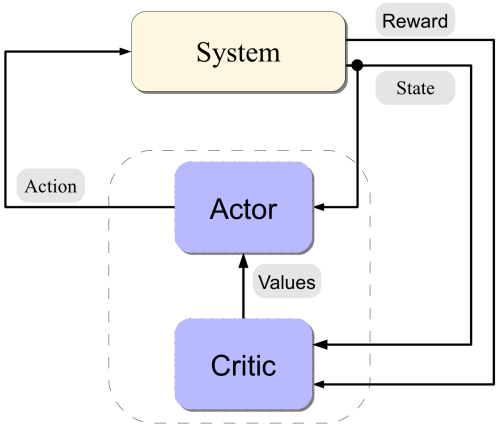


Figure 4: The Actor-Critic Architecture

*Bellman error* of $\mathcal{F}$:

$$e_1^*(\mathcal{F}) = \sup_{Q \in \mathcal{F}} \inf_{Q' \in \mathcal{F}} \|Q' - T^*Q\|_\mu ,$$

where $\mu$ is the distribution of state-action pairs in the training sample. That is, $e_1^*(\mathcal{F})$ measures how close $\mathcal{F}$ is to $T^*\mathcal{F} \stackrel{\text{def}}{=} \{T^*Q \,|\, Q \in \mathcal{F}\}$. The bounds derived have the form of the finite-sample bounds that hold in supervised learning, except that the approximation error is measured by $e_1^*(\mathcal{F})$. Note that in the earlier-mentioned counterexamples to the convergence of fitted-value iteration, $e_1^*(\mathcal{F}) = \infty$, suggesting that it is the lack of flexibility of the function approximation method that causes divergence.

## 4.2 Actor-critic methods

Actor-critic methods implement generalized policy iteration. Remember that policy iteration works by alternating between a complete policy evaluation and a complete policy improvement step. When using sample-based methods or function approximation, exact evaluation of the policies may require infinitely many samples or might be impossible due to the restrictions of the function-approximation technique. Hence, reinforcement learning algorithms simulating policy iteration must change the policy based on incomplete knowledge of the value function.

Algorithms that update the policy before it is completely evaluated are said to implement *generalized policy iteration* (GPI). In GPI there are two closely interacting processes of an actor and a critic: the actor aims at improving the current policy, while the critic evaluates the current policy, thus helping the actor. The interaction of the actor and the critic is illustrated on Figure 4 in a closed-loop learning situation.

### 4.2.1 Implementing a critic

The job of the critic is to estimate the value of the current target policy of the actor. This is a value prediction problem. Therefore, the critic can use any value prediction method. Since the actor needs action values, the algorithms are typically modified so that they estimate action values directly. When TD($\lambda$) is appropriately extended, the algorithm known as

**Algorithm 6** The function implementing the SARSA($\lambda$) algorithm with linear function approximation. This function must be called after each transition.

**function** SARSALAMBDALINFAPP($X, A, R, Y, A', \theta, z$)
**Input:** $X$ is the last state, $A$ is the last action chosen, $R$ is the immediate reward received when transitioning to $Y$, where action $A'$ is chosen. $\theta \in \mathbb{R}^d$ is the parameter vector of the linear function approximation, $z \in \mathbb{R}^d$ is the vector of eligibility traces
1: $\delta \leftarrow R + \gamma \cdot \theta^\top \varphi[Y, A'] - \theta^\top \varphi[X, A]$
2: $z \leftarrow \varphi[X, A] + \gamma \cdot \lambda \cdot z$
3: $\theta \leftarrow \theta + \alpha \cdot \delta \cdot z$
4: **return** $(\theta, z)$

SARSA($\lambda$) is obtained due to Rummery and Niranjan [74]. The algorithm got its name from its use of the current **S**tate, current **A**ction, next **R**eward, next **S**tate, and next **A**ction. The pseudocode of the corresponding update rule is shown as Algorithm 6.
Being a TD-algorithm, the resulting algorithm is subject to the same limitations as TD($\lambda$) (cf. Section 3.1), i.e., it might diverge in off-policy situations. It is, however, possible to extend GTD2 and TDC to work with action values (and use $\lambda > 0$) so that the resulting algorithms would become free of these limitations. For details consult [58].

### 4.2.2 Implementing an actor

Policy improvement can be implemented in two ways: One idea is moving the current policy towards the greedy policy underlying the approximate action-value function obtained from the critic. Another idea is to perform gradient ascent directly on the performance surface underlying a chosen parametric policy class. Here we describe a method based on the latter idea.
Consider a smoothly parameterized policy class $\Pi = (\pi_\omega; \omega \in \mathbb{R}^{d_\omega})$ of stochastic stationary policies. When the action space is finite a popular choice for $\Pi$ is to use the so-called *Gibbs policies*:

$$\pi_\omega(a|x) = \frac{\exp(\omega^\top \xi(x,a))}{\sum_{a' \in \mathcal{A}} \exp(\omega^\top \xi(x,a'))}, \qquad x \in \mathcal{X}, a \in \mathcal{A}.$$

Here $\xi : \mathcal{X} \times \mathcal{A} \to \mathbb{R}^{d_\omega}$ is an appropriate feature-extraction function. On the other hand, if the action space is a subset of a $d_{\mathcal{A}}$-dimension Euclidean space, a popular choice is to use Gaussian policies when given some parametric mean $g_\omega(x,a)$ and covariance $\Sigma_\omega(x,a)$ functions, the density specifying the action-selection distribution under $\omega$ is defined by

$$\pi_\omega(a|x) = \frac{1}{\sqrt{(2\pi)^{d_{\mathcal{A}}} \det(\Sigma_\omega(x,a))}} \exp\Big(-(a - g_\omega(x,a))^\top \Sigma_\omega^{-1}(x,a)\,(a - g_\omega(x,a)\Big).$$

Care must be taken to ensure that $\Sigma_\omega$ is positive definite. Often, for simplicity, $\Sigma_\omega$ is taken to be $\Sigma_\omega = \beta I$ with some $\beta > 0$.
Given $\Pi$, formally, the problem is to find the value of $\omega$ corresponding to the best performing policy:

$$\operatorname*{argmax}_\omega \rho_\omega = ?$$

**Algorithm 7** An actor-critic algorithm that uses compatible function approximation and SARSA(1).

**function** SARSAACTORCRITIC

1: $\omega, \theta, z \leftarrow 0$
2: $A \leftarrow a_1$
3: **repeat**
4: $\quad (R, Y) \leftarrow \text{EXECUTEINWORLD}(A)$
5: $\quad A' \leftarrow \text{DRAW}(\pi_\omega(Y, \cdot))$
6: $\quad (\theta, z) \leftarrow \text{SARSALAMBDALINFAPP}_\omega(X, A, R, Y, A', \theta, z)$ $\quad \triangleright$ Use $\lambda = 1$ and $\alpha \gg \beta$
7: $\quad \psi \leftarrow \frac{\partial}{\partial \omega} \log \pi_\omega(X, A)$
8: $\quad v \leftarrow \text{SUM}(\pi_\omega(Y, \cdot) \cdot \theta^\top \varphi[X, \cdot])$
9: $\quad \omega \leftarrow \omega + \beta \cdot \left(\theta^\top \varphi[X, A] - v\right) \cdot \psi$
10: $\quad X \leftarrow Y$
11: $\quad A \leftarrow A'$
12: **until** `True`

Here, the performance, $\rho_\omega$, is usually the expected return of policy $\pi_\omega$ with respect to the initial distribution over the states.

One possible actor-critic algorithm is shown in Algorithm 7. A special assumption in this method is that the features $\psi_\omega$ used by SARSA($\lambda$) called in line 6 are *compatible* with $\pi_\omega$ in the sense that $\psi_\omega(x, a) = \frac{\partial}{\partial \omega} \log \pi_\omega(x, a)$. Note that the features depend on the value of $\omega$, which, in the pseudocode is signified by the presence of $\omega$ as a subindex to SARSALAMBDALINFAPP. Konda and Tsitsiklis [53] proved that (under some regularity conditions) $\liminf_{t\to\infty} \nabla_\omega \rho_{\omega_t} = 0$ holds almost surely if the average-cost version of SARSA(1) is used to update $\theta_t$. They have also shown that if SARSA($\lambda$) is used and $m_\lambda = \liminf_{t\to\infty} \nabla_\omega \rho_{\omega_t}$ then $\lim_{\lambda\to 1} m_\lambda = 0$.

Another interesting algorithm is obtained if one replaces the update in line 9 with

$$\omega \leftarrow \omega + \beta \cdot \theta,$$

which is called the *natural actor-critic* (NAC) update. Under some mild conditions, this algorithm can be seen to implement a stochastic pseudo-gradient algorithm, and thus the previous results extend to it.

Interestingly, the NAC update can result in an even faster convergence than the previous rule. The reason is that $\theta_*(\omega)$ can be shown to be a so-called *natural gradient* [4] of $\rho_\omega$. This was first noted by Kakade [47]. It is believed that following a natural gradient generally improves the behavior of gradient ascent methods. This is nicely demonstrated by Kakade [47] on a simple two-state MDP, where the "normal" gradient is very small in a large part of the parameter space, while the natural gradient behaves in a reasonable manner. Other positive examples were given by Bagnell and Schneider [7], Peters et al. [67] and Peters and Schaal [66].

# 5 For further exploration

Inevitably, due to space constraints, this review must miss a large portion of the reinforcement learning literature.

One topic of particular interest not discussed is efficient sampling-based planning [49, 88, 51, 27]. The main lesson here is that off-line planning in the worst-case can scale exponentially with the dimensionality of the state space [28], while online planning (i.e., planning for the "current state") can break the curse of dimensionality by amortizing the planning effort over multiple time steps [76, 88].
Other topics of interest include the linear programming-based approaches [31, 32, 33], dual dynamic programming [101], techniques based on sample average approximation [78] such as PEGASUS [63], online learning in MDPs with arbitrary reward processes [37, 106, 62], or learning with (almost) no restrictions in a competitive framework [44].
Other important topics include learning and acting in partially observed MDPs [for recent developments, see, e.g., 57, 95, 73], learning and acting in games or under some other optimization criteria [56, 43, 91, 17], or the development of hierarchical and multi-time-scale methods [34, 86].

## 5.1 Applications

The numerous successful applications of reinforcement learning include (in no particular order) games (e.g., Backgammon [94] and Go [79]), applications in networking (e.g., packet routing [20], channel allocation [81]), applications to operations research problems (e.g., targeted marketing [3], maintenance problems [42], job-shop scheduling [107], elevator control [30], pricing [75], vehicle routing [70], inventory control [26], fleet management [80]), learning in robotics (e.g., controlling quadrupedales [52], humanoid robots [67], or helicopters [2]), and applications to finance (e.g., option pricing [98, 99, 105, 55]). For further applications, see e.g. the lists at the URLs

- `http://www.cs.ualberta.ca/~szepesva/RESEARCH/RLApplications.html` and
- `http://umichrl.pbworks.com/Successes-of-Reinforcement-Learning`.

## 5.2 Software

There are numerous software packages that support the development and testing of RL algorithms. Perhaps the most notable of these are the RL-Glue and RL-Library packages. The RL-Glue package available from `http://glue.rl-community.org` is intended for helping to standardize RL experiments. It is a free, language-neutral software package that implements a standardized RL interface [93]. The RL-Library (`http://library.rl-community.org`) builds on the top of RL-Glue. Its purpose is to provide trusted implementations of various RL testbeds and algorithms. The most notable other RL software packages are CLSquare,[4] PIQLE,[5] RL Toolbox,[6] JRLF[7] and LibPG.[8] These offer the implementation of a large number of algorithms, testbeds, intuitive visualizations, programming tools, etc. Many of these packages support RL-Glue.

[4] `http://www.ni.uos.de/index.php?id=70`
[5] `http://piqle.sourceforge.net/`
[6] `http://www.igi.tugraz.at/ril-toolbox/`
[7] `http://mykel.kochenderfer.com/?page_id=19`
[8] `http://code.google.com/p/libpgrl/`

# References


[1] S. J. Russell A. Prieditis, editor. *Proceedings of the 12th International Conference on Machine Learning (ICML 1995)*, San Francisco, CA, USA, 1995. Morgan Kaufmann. ISBN 1-55860-377-8.

[2] P. Abbeel, A. Coates, M. Quigley, and A. Y. Ng. An application of reinforcement learning to aerobatic helicopter flight. In Schölkopf et al. [77], pages 1–8. ISBN 0-262-19568-2.

[3] N. Abe, N. K. Verma, C. Apté, and R. Schroko. Cross channel optimized marketing by reinforcement learning. In W. Kim, R. Kohavi, J. Gehrke, and W. DuMouchel, editors, *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 767–772, New York, NY, USA, 2004. ACM. ISBN 1-58113-888-1.

[4] S. Amari. Natural gradient works efficiently in learning. *Neural Computation*, 10(2): 251–276, 1998.

[5] A. Antos, R. Munos, and Cs. Szepesvári. Fitted Q-iteration in continuous action-space MDPs. In Platt et al. [68], pages 9–16.

[6] P. Auer, T. Jaksch, and R. Ortner. Near-optimal regret bounds for reinforcement learning. *Journal of Machine Learning Research*, 11:1563—1600, 2010.

[7] J. A. Bagnell and J. G. Schneider. Covariant policy search. In G. Gottlob and T. Walsh, editors, *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 1019–1024, San Francisco, CA, USA, August 9–15 2003. Morgan Kaufmann.

[8] L. C. Baird. Residual algorithms: Reinforcement learning with function approximation. In A. Prieditis [1], pages 30–37. ISBN 1-55860-377-8.

[9] D. P. Bertsekas. *Dynamic Programming and Optimal Control*, volume 1. Athena Scientific, Belmont, MA, 3 edition, 2007.

[10] D. P. Bertsekas. *Dynamic Programming and Optimal Control*, volume 2. Athena Scientific, Belmont, MA, 3 edition, 2007.

[11] D. P. Bertsekas. Approximate dynamic programming (online chapter). In *Dynamic Programming and Optimal Control*, volume 2, chapter 6. Athena Scientific, Belmont, MA, 3 edition, May 13 2010. URL `http://web.mit.edu/dimitrib/www/dpchapter.pdf`.

[12] D. P. Bertsekas and S. Ioffe. Temporal differences-based policy iteration and applications in neuro-dynamic programming. LIDS-P-2349, MIT, 1996.

[13] D. P. Bertsekas and S.E. Shreve. *Stochastic Optimal Control (The Discrete Time Case)*. Academic Press, New York, 1978.

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

[15] V. S. Borkar. Stochastic approximation with two time scales. *Systems & Control Letters*, 29(5):291–294, 1997.

[16] V. S. Borkar. *Stochastic Approximation: A Dynamical Systems Viewpoint*. Cambridge University Press, 2008.

[17] V. S. Borkar and S. P. Meyn. Risk-sensitive optimal control for Markov decision processes with monotone cost. *Mathematics of Operations Research*, 27(1):192—209, Jan 2002. URL `http://mor.journal.informs.org/cgi/content/abstract/27/1/192`.

[18] L. Bottou and O. Bousquet. The tradeoffs of large scale learning. In Platt et al. [68], pages 161–168.

[19] J. A. Boyan. Technical update: Least-squares temporal difference learning. *Machine Learning*, 49:233–246, 2002.

[20] J. A. Boyan and M. L. Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *NIPS-6: Advances in Neural Information Processing Systems: Proceedings of the 1993 Conference*, pages 671–678. Morgan Kauffman, San Francisco, CA, USA, 1994.

[21] J. A. Boyan and A. W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. Touretzky, and T. Leen, editors, *NIPS-7: Advances in Neural Information Processing Systems: Proceedings of the 1994 Conference*, pages 369–376, Cambridge, MA, USA, 1995. MIT Press.

[22] S. J. Bradtke and A. G. Barto. Linear least-squares algorithms for temporal difference learning. *Machine Learning*, 22:33–57, 1996.

[23] R. I. Brafman and M. Tennenholtz. R-MAX - a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3: 213–231, 2002.

[24] L. Busoniu, R. Babuska, B. De Schutter, and D. Ernst. *Reinforcement Learning and Dynamic Programming Using Function Approximators*. Automation and Control Engineering Series. CRC Press, 2010.

[25] X. R. Cao. *Stochastic Learning and Optimization: A Sensitivity-Based Approach*. Springer, New York, 2007.

[26] H. S. Chang, M. C. Fu, J. Hu, and S. I. Marcus. An asymptotically efficient simulation-based algorithm for finite horizon stochastic dynamic programming. *IEEE Transactions on Automatic Control*, 52(1):89–94, Jan 2007.

[27] H. S. Chang, M. C. Fu, J. Hu, and S. I. Marcus. *Simulation-based Algorithms for Markov Decision Processes*. Springer Verlag, 2008.

[28] C. S. Chow and J. N. Tsitsiklis. The complexity of dynamic programming. *Journal of Complexity*, 5:466–488, 1989.

[29] W. W. Cohen and H. Hirsh, editors. *Proceedings of the 11th International Conference on Machine Learning (ICML 1994)*, San Francisco, CA, USA, 1994. Morgan Kaufmann. ISBN 1-55860-335-2.

[30] R. H. Crites and A. G. Barto. Improving elevator performance using reinforcement learning. In D. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *NIPS-8: Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference*, pages 1017–1023, Cambridge, MA, USA, 1996. MIT Press.

[31] D. P. de Farias and B. Van Roy. The linear programming approach to approximate dynamic programming. *Operations Research*, 51(6):850–865, 2003.

[32] D. P. de Farias and B. Van Roy. On constraint sampling in the linear programming approach to approximate dynamic programming. *Mathematics of Operations Research*, 29(3):462–478, 2004.

[33] D. P. de Farias and B. Van Roy. A cost-shaping linear program for average-cost approximate dynamic programming with performance guarantees. *Mathematics of Operations Research*, 31(3):597–620, 2006.

[34] T. Dietterich. The MAXQ method for hierarchical reinforcement learning. In J. W. Shavlik, editor, *Proceedings of the 15th International Conference on Machine Learning (ICML 1998)*, pages 118–126, San Francisco, CA, USA, 1998. Morgan Kauffmann. ISBN 1-55860-556-8.

[35] T. G. Dietterich, S. Becker, and Z. Ghahramani, editors. *Advances in Neural Information Processing Systems 14*, Cambridge, MA, USA, 2001. MIT Press.

[36] D. Ernst, P. Geurts, and L. Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6:503–556, 2005.

[37] E. Even-Dar, S. M. Kakade, and Y. Mansour. Experts in a Markov decision process. In L. K. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems 17*, pages 401–408, Cambridge, MA, USA, 2005. MIT Press.

[38] J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, editors. *Proceedings of the 17th European Conference on Machine Learning (ECML-2006)*, 2006. Springer.

[39] A. Geramifard, M. H. Bowling, M. Zinkevich, and R. S. Sutton. iLSTD: Eligibility traces and convergence analysis. In Schölkopf et al. [77], pages 441–448. ISBN 0-262-19568-2.

[40] G. J. Gordon. Stable function approximation in dynamic programming. In A. Prieditis [1], pages 261–268. ISBN 1-55860-377-8.

[41] A. Gosavi. *Simulation-based optimization: parametric optimization techniques and reinforcement learning*. Springer Netherlands, 2003.

[42] A Gosavi. Reinforcement learning for long-run average cost. *European Journal of Operational Research*, 155(3):654–674, Jun 2004. doi: 10.1016/S0377-2217(02)00874-3.

[43] M. Heger. Consideration of risk in reinforcement learning. In Cohen and Hirsh [29], pages 105–111. ISBN 1-55860-335-2.

[44] Marcus Hutter. *Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability*. Springer, Berlin, 2004. URL `http://www.hutter1.net/ai/uaibook.htm`. 300 pages, http://www.idsia.ch/~marcus/ai/uaibook.htm.

[45] Nicholas K. Jong and Peter Stone. Model-based exploration in continuous state spaces. In Ian Miguel and Wheeler Ruml, editors, *7th International Symposium on Abstraction, Reformulation, and Approximation (SARA 2007)*, volume 4612 of *Lecture Notes in Computer Science*, pages 258–272, Whistler, Canada, July 18-21 2007. Springer. ISBN 978-3-540-73579-3.

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

[47] S. Kakade. A natural policy gradient. In Dietterich et al. [35], pages 1531–1538.

[48] S. Kakade, M. J. Kearns, and J. Langford. Exploration in metric state spaces. In T. Fawcett and N. Mishra, editors, *Proceedings of the 20th International Conference on Machine Learning (ICML 2003)*, pages 306–312. AAAI Press, 2003. ISBN 1-57735-189-4.

[49] M. J. Kearns, Y. Mansour, and A. Y. Ng. Approximate planning in large POMDPs via reusable trajectories. In S. A. Solla, T. K. Leen, and K. R. Müller, editors, *Advances in Neural Information Processing Systems 12*, pages 1001–1007. MIT Press, Cambridge, MA, USA, 1999.

[50] M.J. Kearns and S.P. Singh. Near-optimal reinforcement learning in polynomial time. *Machine Learning*, 49(2–3):209–232, 2002.

[51] L. Kocsis and Cs. Szepesvári. Bandit based Monte-Carlo planning. In Fürnkranz et al. [38], pages 282–293.

[52] N. Kohl and P. Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *Proceedings of the 2004 IEEE International Conference on Robotics and Automation*, pages 2619–2624. IEEE, 2004.

[53] V. R. Konda and J. N. Tsitsiklis. On actor-critic algorithms. *SIAM J. Control and Optimization*, 42(4):1143–1166, 2003. doi: 10.1137/S0363012901385691. URL `http://link.aip.org/link/?SJC/42/1143/1`.

[54] M. R. Kosorok. *Introduction to Empirical Processes and Semiparametric Inference*. Springer, 2008.

[55] Y. Li, Cs. Szepesvári, and D. Schuurmans. Learning exercise policies for american options. In *Proc. of the Twelfth International Conference on Artificial Intelligence and Statistics, JMLR: W&CP*, volume 5, pages 352–359, 2009.

[56] M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. In Cohen and Hirsh [29], pages 157–163. ISBN 1-55860-335-2.

[57] M. L. Littman, R. S. Sutton, and S. P. Singh. Predictive representations of state. In Dietterich et al. [35], pages 1555–1561.

[58] H. R. Maei and R. S. Sutton. GQ($\lambda$): A general gradient algorithm for temporal-difference prediction learning with eligibility traces. In E. Baum, M. Hutter, and E. Kitzelmann, editors, *Proceedings of the Third Conference on Artificial General Intelligence*, pages 91–96. Atlantis Press, 2010.

[59] H.R. Maei, Cs. Szepesvári, S. Bhatnagar, D. Silver, D. Precup, and R.S. Sutton. Convergent temporal-difference learning with arbitrary smooth function approximation. In *NIPS-22*, pages 1204–1212, 2010.

[60] R. Munos and Cs. Szepesvári. Finite-time bounds for fitted value iteration. *Journal of Machine Learning Research*, 9:815–857, 2008.

[61] A. Nedič and D. P. Bertsekas. Least squares policy evaluation algorithms with linear function approximation. *Discrete Event Dynamic Systems*, 13(1):79–110, 2003.

[62] G. Neu, A. György, and Cs. Szepesvári. The online loop-free stochastic shortest-path problem. In *COLT-10*, 2010.

[63] A. Y. Ng and M. Jordan. PEGASUS: A policy search method for large MDPs and POMDPs. In C. Boutilier and M. Goldszmidt, editors, *Proceedings of the 16th Conference in Uncertainty in Artificial Intelligence (UAI'00)*, pages 406–415, San Francisco CA, 2000. Morgan Kaufmann. ISBN 1-55860-709-9.

[64] A. Nouri and M.L. Littman. Multi-resolution exploration in continuous spaces. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, pages 1209–1216, Cambridge, MA, USA, 2009. MIT Press.

[65] D. Ormoneit and S. Sen. Kernel-based reinforcement learning. *Machine Learning*, 49: 161–178, 2002.

[66] J. Peters and S. Schaal. Natural actor-critic. *Neurocomputing*, 71(7–9):1180–1190, 2008.

[67] J. Peters, S. Vijayakumar, and S. Schaal. Reinforcement learning for humanoid robotics. In *Humanoids2003, Third IEEE-RAS International Conference on Humanoid Robots*, pages 225—230, 2003.

[68] J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors. *Advances in Neural Information Processing Systems 20*, Cambridge, MA, USA, 2008. MIT Press.

[69] W. B. Powell. *Approximate Dynamic Programming: Solving the curses of dimensionality.* John Wiley and Sons, New York, 2007.

[70] S. Proper and P. Tadepalli. Scaling model-based average-reward reinforcement learning for product delivery. In Fürnkranz et al. [38], pages 735–742.

[71] M.L. Puterman. *Markov Decision Processes — Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, 1994.

[72] M. Riedmiller. Neural fitted Q iteration – first experiences with a data efficient neural reinforcement learning method. In J. Gama, R. Camacho, P. Brazdil, A. Jorge, and L. Torgo, editors, *Proceedings of the 16th European Conference on Machine Learning (ECML-05)*, volume 3720 of *Lecture Notes in Computer Science*, pages 317–328. Springer, 2005. ISBN 3-540-29243-8.

[73] S. Ross, J. Pineau, S. Paquet, and B. Chaib-draa. Online planning algorithms for POMDPs. *Journal of Artificial Intelligence Research*, 32:663–704, 2008.

[74] G. A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Cambridge University Engineering Department, 1994.

[75] P. Rusmevichientong, J. A. Salisbury, L. T. Truss, B. Van Roy, and P. W. Glynn. Opportunities and challenges in using online preference data for vehicle pricing: A case study at General Motors. *Journal of Revenue and Pricing Management*, 5(1): 45–61, 2006.

[76] J. Rust. Using randomization to break the curse of dimensionality. *Econometrica*, 65: 487–516, 1996.

[77] B. Schölkopf, J. C. Platt, and T. Hoffman, editors. *Advances in Neural Information Processing Systems 19*, Cambridge, MA, USA, 2007. MIT Press. ISBN 0-262-19568-2.

[78] A. Shapiro. Monte Carlo sampling methods. In *Stochastic Programming, Handbooks in OR & MS*, volume 10. North-Holland Publishing Company, Amsterdam, 2003.

[79] D. Silver, R. S. Sutton, and M. Müller. Reinforcement learning of local shape in the game of Go. In M. M. Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 1053—1058, 2007.

[80] H. P. Simão, J. Day, A. P. George, T. Gifford, J. Nienow, and W. B. Powell. An approximate dynamic programming algorithm for large-scale fleet management: A case application. *Transportation Science*, 43(2):178–197, 2009.

[81] S. P. Singh and D. P. Bertsekas. Reinforcement learning for dynamic channel allocation in cellular telephone systems. In M. C. Mozer, M. I. Jordan, and T. Petsche, editors, *NIPS-9: Advances in Neural Information Processing Systems: Proceedings of the 1996 Conference*, pages 974–980, Cambridge, MA, USA, 1997. MIT Press.

[82] S. P. Singh and R. C. Yee. An upper bound on the loss from approximate optimal-value functions. *Machine Learning*, 16(3):227–233, 1994.

[83] R. S. Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts, Amherst, MA, 1984.

[84] R. S. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3(1):9–44, 1988.

[85] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. Bradford Book. MIT Press, 1998.

[86] R. S. Sutton, D. Precup, and S. P. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112: 181–211, 1999.

[87] R. S. Sutton, H. R. Maei, D. Precup, S. Bhatnagar, D. Silver, Cs. Szepesvári, and E. Wiewiora. Fast gradient-descent methods for temporal-difference learning with linear function approximation. In A. P. Danyluk, L. Bottou, and M. L. Littman, editors, *Proceedings of the 26th Annual International Conference on Machine Learning (ICML 2009)*, volume 382 of *ACM International Conference Proceeding Series*, pages 993—1000, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-516-1.

[88] Cs. Szepesvári. Efficient approximate planning in continuous space Markovian decision problems. *AI Communications*, 13:163–176, 2001.

[89] Cs. Szepesvári. Reinforcement learning algorithms for MDPs – a survey. Technical Report TR09-13, Department of Computing Science, University of Alberta, 2009.

[90] Cs. Szepesvári. *Reinforcement Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2010.

[91] Cs. Szepesvári and M. L. Littman. A unified analysis of value-function-based reinforcement-learning algorithms. *Neural Computation*, 11:2017–2059, 1999.

[92] I. Szita and Cs. Szepesvári. Model-based reinforcement learning with nearly tight exploration complexity bounds. In S. Wrobel, J. Fürnkranz, and T. Joachims, editors, *Proceedings of the 27th Annual International Conference on Machine Learning (ICML 2010)*, ACM International Conference Proceeding Series, New York, NY, USA, 2010. ACM.

[93] B. Tanner and A. White. RL-Glue: Language-independent software for reinforcement-learning experiments. *Journal of Machine Learning Research*, 10:2133–2136, 2009.

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

[95] M. Toussaint, L. Charlin, and P. Poupart. Hierarchical POMDP controller optimization by likelihood maximization. In D. A. McAllester and P. Myllymäki, editors, *Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence*, pages 562–570. AUAI Press, 2008. ISBN 0-9749039-4-9.

[96] J. N. Tsitsiklis and B. Van Roy. Feature-based methods for large scale dynamic programming. *Machine Learning*, 22:59–94, 1996.

[97] J. N. Tsitsiklis and B. Van Roy. An analysis of temporal difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42:674–690, 1997.

[98] J. N. Tsitsiklis and B. Van Roy. Optimal stopping of Markov processes: Hilbert space theory, approximation algorithms, and an application to pricing financial derivatives. *IEEE Transactions on Automatic Control*, 44:1840–1851, 1999.

[99] J. N. Tsitsiklis and B. Van Roy. Regression methods for pricing complex American-style options. *IEEE Transactions on Neural Networks*, 12:694–703, 2001.

[100] B. Van Roy. Performance loss bounds for approximate value iteration with state aggregation. *Mathematics of Operations Research*, 31(2):234–244, 2006.

[101] T. Wang, D. J. Lizotte, M. H. Bowling, and D. Schuurmans. Stable dual dynamic programming. In Platt et al. [68].

[102] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, 1989.

[103] B. Widrow and S.D. Stearns. *Adaptive Signal Processing*. Prentice Hall, Englewood Cliffs, NJ, 1985.

[104] X. Xu, H. He, and D. Hu. Efficient reinforcement learning using recursive least-squares methods. *Journal of Artificial Intelligence Research*, 16:259–292, 2002.

[105] H. Yu and D.P. Bertsekas. Q-learning algorithms for optimal stopping based on least squares. In *Proceedings of the European Control Conference*, 2007.

[106] J. Y. Yu, S. Mannor, and N. Shimkin. Markov decision processes with arbitrary reward processes. *Mathematics of Operations Research*, 2009. to appear.

[107] W. Zhang and T. G. Dietterich. A reinforcement learning approach to job-shop scheduling. In C. R. Perrault and C. S. Mellish, editors, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 1114–1120, San Francisco, CA, USA, 1995. Morgan Kaufmann.