CRICOS
00213J

ENN582: Reinforcement Learning and Optimal Control

Reinforcement Learning

Daniel E. Quevedo[‡]

VERSION 1.

Abstract

The purpose of these notes is to present methods to present basic methods for the design of optimal controllers using learning from data. A more comprehensive exposition can be found in books, such as [7].

1 Key Learning Objectives

In conjunction with the practical, you will learn to

- apply the temporal-difference prediction method
- formulate controllers using Sarsa and Q-learning

[‡]Please report errors within this document to daniel.quevedo@qut.edu.au

2 Introduction

The algorithms we have examined so far for calculating control policies make extensive use of system models. In many practical applications good system models are not available. A traditional way of control design in such scenarios involves two steps:

1. Carry out experiments and identify a system model from data. This is called *system identification* and many interesting methods exist; see, e.g., [6, 4].
2. Use the identified system model to design the control policy, e.g., based on Model Predictive Control or Dynamic Programming.

Instead of the above design procedure, we will next outline methods that provide control policies from data but without explicitly using a complete system model. Within the dynamic programming framework, such methods are often called *Neurodynamic Programming* or *Reinforcement Learning*. For simplicity, we will restrict our focus to systems with finite state and action spaces. As we shall see, these lead to tabular solution methods.

Reinforcement learning controllers (also called "agents") interact with the system ("environment") by implementing control values, and observing stage costs (or "rewards") and state values. The situation is depicted in Figure 1.

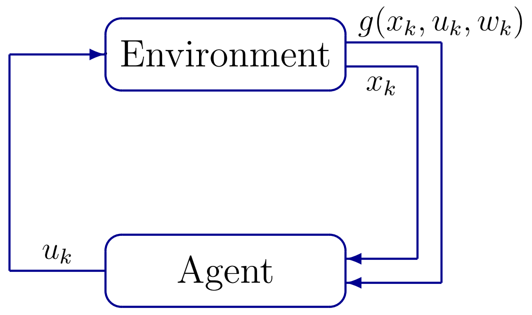

Figure 1: Reinforcement learning agent interacting with the environment

3 Temporal Difference Predictions

Before developing learning-based controllers, we will first present an algorithm that evaluates the performance of a given policy $\pi = \{\mu, \mu, \dots\}$ from data. We will focus on stationary stochastic control scenarios with an infinite horizon cost function. Thus, starting from $x_0 = x$, the cost is given by:

$$J_\mu(x) = \lim_{N\to\infty} \underset{w_k}{E}\left\{ \sum_{k=0}^{N-1} \alpha^k g(x_k, \mu(x_k), w_k) \right\}. \tag{1}$$

Assuming that a system model, say $x_{k+1} = f(x_k, u_k, w_k)$, is available, this cost can be computed using Bellman's Equation for μ:

$$J_\mu(x) = \underset{w}{E}\big\{g(x, \mu(x), w) + \alpha J_\mu\big(f(x, \mu(x), w)\big)\big\}, \quad \forall x \in \mathbb{X}. \tag{2}$$

This equation shows that $g(x, \mu(x), w) + \alpha J_\mu\big(f(x, \mu(x), w)$ is an unbiased estimate of $J_\mu(x)$.

If no system model is available, then (2) can be used to estimate $J_\mu(x)$ from empirical state and stage cost data

$$\big(x_k, g(x_k, \mu(x_k), w_k), x_{k+1}\big). \tag{3}$$

In particular, starting from an arbitrary function J, the one-step temporal-difference (TD) learning algorithm[1] repeatedly evaluates the policy μ to obtain a cost $g(x_k, \mu(x_k), w_k)$ and update the function for the old state x_k using the rule:[2]

$$J(x_k) \leftarrow J(x_k) + \delta_k\big(g(x_k, \mu(x_k), w_k) + \alpha J(x_{k+1}) - J(x_k)\big), \tag{4}$$

where $\delta_k > 0$ is the *learning rate*. By iterating (4), the function $J(x)$ will typically converge to the policy cost $J_\mu(x)$, for all $x \in \mathbb{X}$.

In (4), the term $g(x_k, \mu(x_k), w_k) + \alpha J(x_{k+1})$ is called the *TD target*. It is computed by using the data samples in (3) and serves to approximate the expectation in (2), while using $J(x_k)$ instead of $J_\mu(x)$. The quantity

$$g(x_k, \mu(x_k), w_k) + \alpha J(x_{k+1}) - J(x_k)$$

is called *TD error*. This value quantifies the difference between the prior estimate $J(x_k)$ and the revised estimate $g(x_k, \mu(x_k), w_k) + \alpha J(x_{k+1})$.

4 Action-value functions

In order to incorporate learning in optimal control problems, it is convenient to introduce *action-value* functions. For any given policy μ, its action-value function $Q_\mu(x, u)$ quantifies the cost incurred when at state x, the control u is applied, and the policy μ is used thereafter. In view of (1) we have:

$$Q_\mu(x, u) = \underset{w}{E}\big\{g(x, u, w) + \alpha J_\mu\big(f(x, u, w)\big)\big\}. \tag{5}$$

We note that the optimal action-value function is given by

$$Q^*(x, u) = \min_\mu Q_\mu(x, u), \quad \forall x, u \tag{6}$$

so that

$$J_{\mu^*}(x) = \min_{u \in \mathbb{U}} Q^*(x, u), \quad \forall x \in \mathbb{X}.$$

[1]There also exist multi-step TD methods, see [7].

[2]This type of recursion constitutes a *Robbins-Monro algorithm*, which underlies a number of methods used in *Stochastic approximation*.

The optimal action-value function (also called *Q-factor*) $Q^*(x,u)$ quantifies the infinite horizon discounted cost incurred when at state x, the action u is applied and the optimal policy μ^* is used for all subsequent states. The optimal policy μ^* can be directly extracted from

$$\mu^*(x) = \arg\min_{u\in\mathbb{U}} Q^*(x,u). \tag{7}$$

It is worth noting that in view of Bellman's Equation (for the optimal policy)

$$J_{\mu^*}(x) = \min_{u\in\mathbb{U}} \underset{w}{E}\big\{g(x,u,w) + \alpha J_{\mu^*}\big(f(x,u,w)\big)\big\}, \quad \forall x \in \mathbb{X} \tag{8}$$

we obtain the **Q-Bellman Equation**:

$$Q^*(x,u) = \underset{w}{E}\big\{g(x,u,w) + \alpha J_{\mu^*}\big(f(x,u,w)\big)\big\} = \underset{w}{E}\big\{g(x,u,w) + \alpha \min_{u\in\mathbb{U}} Q^*(f(x,u,w),u)\big\}. \tag{9}$$

This expression is widely used in reinforcement learning algorithms.

5 Sarsa Algorithm

Similar to the TD method for learning the state-value function $J_\mu(x)$ for a policy μ, one can use temporal differences to learn the action-value function $Q_\mu(x,u)$. To be more specific, (5), provides

$$Q_\mu(x,u) = \underset{w}{E}\big\{g(x,u,w) + \alpha Q_\mu\big(f(x,u,w), \mu(f(x,u,w))\big)\big\}$$

which motivates the Sarsa prediction algorithm:

$$Q(x_k,u_k) \leftarrow Q(x_k,u_k) + \delta_k\big(g(x_k,u_k,w_k) + \alpha Q(x_{k+1},u_{k+1}) - Q(x_k,u_k)\big), \tag{10}$$

where $u_k = \mu(x_k)$, $u_{k+1} = \mu(x_{k+1})$ and $\delta_k > 0$ is the, possibly time-varying, learning rate.

Typically, iterations of (10) starting from arbitrary intial values[3] will converge to the action value-function $Q_\mu(x,u)$ by using the[4] data quintuples $\big(x_k, \mu(x_k), g(x_k,\mu(x_k),w_k), x_{k+1}, \mu(x_{k+1})\big)$.

To use the sarsa prediction method for learning-based control, one can combine it with a policy improvement step, akin to the one used in Policy Iteration. An important difference to the way policy improvement was used in PI is that when learning one needs to ensure that the state space and action space are explored sufficiently. Thus, in addition to *exploiting* the available knowledge about the system and choose the action that apparently minimises the cost, one also needs to *explore* the system to acquire new knowledge. This $\underline{\text{exploration-exploitation tradeoff}}$ is fundamental in learning-based decision making.

A common way to incorporate exploration in policy improvement is based on the ε-greedy concept, wherein exploration occurs with a probability ε_k chosen by the user. Taking into account the current

[3]For terminal states, one needs to choose a zero initial action-value function.
[4]**S**tate, **A**ction, **R**eward, **S**tate, **A**ction

action-value estimate in (10), we obtain (at time k and given state $x_k = x$ and exploration rate $\varepsilon = \varepsilon_k$):

$$u = \begin{cases} \min_{u\in\mathbb{U}} Q(x,u) & \text{with probability } 1-\varepsilon \\ \text{a random action } u \in \mathbb{U} & \text{with probability } \varepsilon. \end{cases} \tag{11}$$

There are no general rules on how to choose the exploration rate ε_k. Typically, it is chosen to decrease over time.

Algorithm 1 Sarsa controller

```
Initialise: $Q(x,u)$, for all $x \in \mathbb{X}$ and $u \in \mathbb{U}$.
  Measure the current state value $x$
  Calculate $u$ by using the $\varepsilon$-greedy policy in (11)
  loop
      Implement the control value $u$
      Observe the state cost $g(x,u,w)$
      Wait one time step and measure the next state $x'$          ▷ Next state (discrete-time control)
      Calculate $u'$ by using $x'$ in the $\varepsilon$-greedy policy in (11)     ▷ Next control value
      Update the Q-table as per (10):          ▷ Updates for the old state and action

          $Q(x,u) \leftarrow Q(x,u) + \delta\big(g(x,u,w) + \alpha Q(x',u') - Q(x,u)\big).$

      $x \leftarrow x'$
      $u \leftarrow u'$
  end loop
```

The resulting controller is given in procedural form in Algorithm 1. It continuously estimates $Q_\mu(x,u)$ for the current policy and measured data and then improves this policy. Note that the Q-function update is based on the system states measured and the ε-greedy control values, that are also implemented. All other Q-functions remain unchanged. If all state-action pairs are visited infinitely often and the policy converges in the limit to the greedy policy, Sarsa converges to the optimal policy and Q-factors $Q^*(x,u)$. This can be achieved, for example, by setting $\varepsilon_k = 1/k$ and a learning rate satisfying[5]

$$\delta_k > 0, \quad \sum_{k=0}^{\infty} \delta_k = \infty, \quad \sum_{k=0}^{\infty} \delta_k^2 < \infty. \tag{12}$$

If a constant exploration rate ε is chosen, then the Q-factors provided by Sarsa will converge to those of the ε-greedy policy in (11).

Figure 2 illustrates the algorithm behaviour when applied to the Rover example. The learning rate is fixed at $\delta = 0.1$, whereas the exploration rate is chosen as $\varepsilon_k = \max(1/k, 0.01)$. Our design reflects the fact that, as time progresses the system needs to be explored less. To avoid this decaying effect to stop learning too quickly, from $k = 100$ onwards, ε_k is kept at 0.01. The controller is trained over 500 *Episodes*, each containing 1,000 time steps. By passing on the Q-factors at the end of each episode to the next, and re-setting the exploration rate, this assists with the learning process.

[5]For example, one can choose $\delta_k = 1/k$.

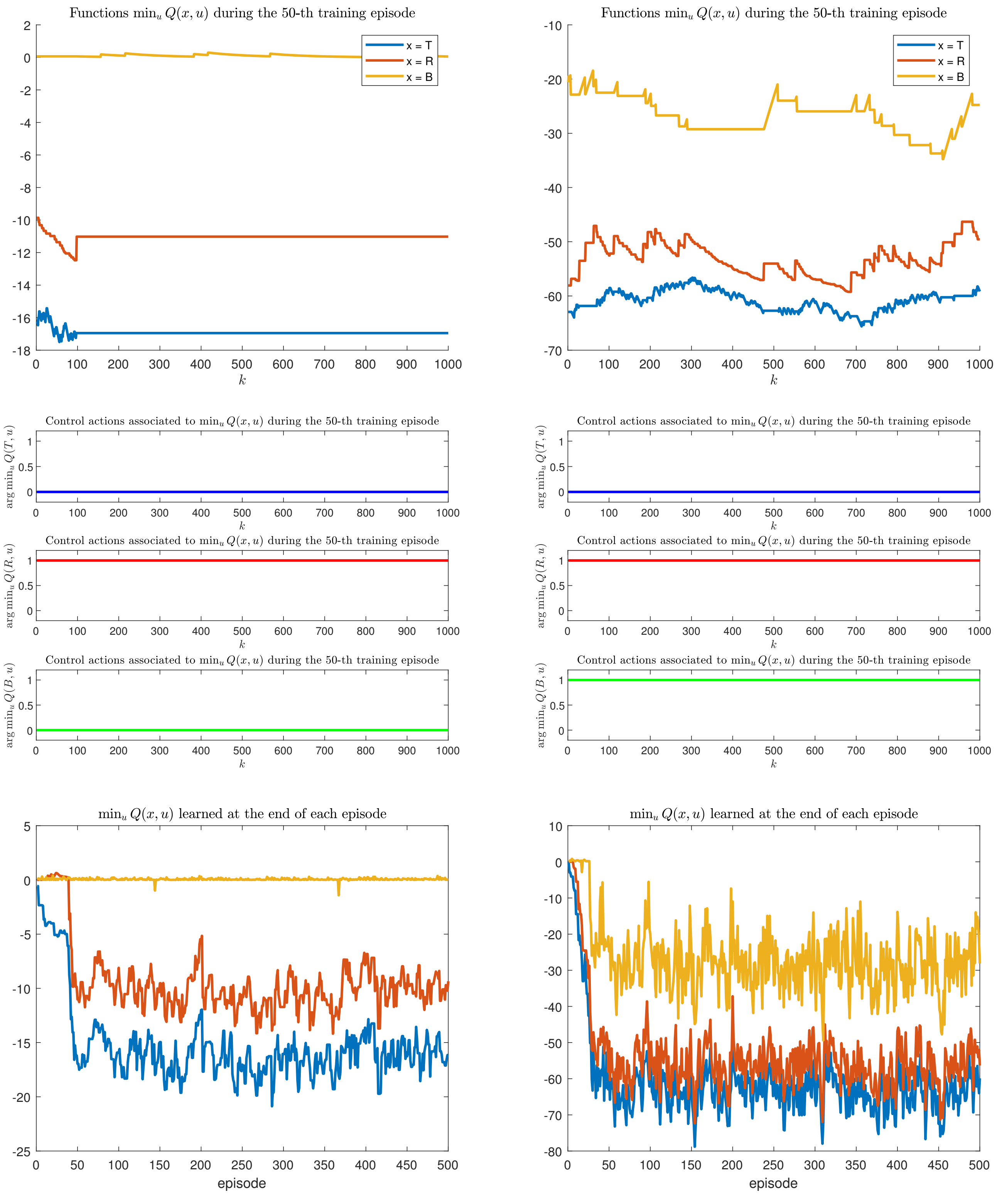

Figure 2: Sarsa for the Rover example: $\alpha = 0.9$ (left), $\alpha = 0.98$ (right)

6 Q-learning

As an alternative to the sarsa predictions in (10), one can also update the Q-table using the greedy control values (instead of the ε-greedy ones):

$$Q(x_k, u_k) \leftarrow Q(x_k, u_k) + \delta_k\big(g(x_k, u_k, w_k) + \alpha \min_{u \in \mathbb{U}} Q(x_{k+1}, u) - Q(x_k, u_k)\big). \tag{13}$$

In view of (9), given arbitrary intial values[6] this recursion will tend to directly approximate the optimal action-value function $Q^*(x,u)$, if the learning rate is chosen as per (12) and the state and action spaces are explored sufficiently. This exploration requirement is commonly achieved by implementing the ε-greedy control policy in (11). The resulting Q-learning controller is outlined in Algorithm 2. Figure 3 illustrates the behaviour of applying the Q-learning algorithm to the Rover example and with parameters chosen as for the Sarsa algorithm example examined before.

```
Algorithm 2 Q-learning controller
Initialise: $Q(x,u)$, for all $x \in \mathbb{X}$ and $u \in \mathbb{U}$.
  loop
      Measure the current state value $x$
      Calculate $u$ by using the $\varepsilon$-greedy policy in (11)
      Implement the control value $u$
      Observe the state cost $g(x,u,w)$
      Wait one time step and measure the next state $x'$        ▷ Next state (discrete-time control)
      Update the Q-table as per (13):                          ▷ Updates for the old state and action
            $Q(x,u) \leftarrow Q(x,u) + \delta\big(g(x,u,w) + \alpha \min_{u \in \mathbb{U}} Q(x',u) - Q(x,u)\big).$
      $x \leftarrow x'$
  end loop
```

Note that the Q-learning controller does not implement the policy used when updating the Q-factors. Such classes of algorithms are classified as *off-policy* methods. Instead, Q-learning estimates the total discounted cost for state-action pairs assuming a greedy policy were followed thereafter, despite the fact that an ε-greedy policy is implemented. In contrast, the Sarsa controller updates the Q-table using the control values implemented at each step. Thus, the Sarsa controller is an *on-policy* algorithm.

For problems with large state and action spaces, Q-learning (as well as Sarsa) requires significant computational resources. To overcome this issue, one can use parameterised approximations of the optimal action-value functions, $\hat{Q}(x,u;r) \approx Q^*(x,u)$ and tune the parameter vector r by minimising the associated (temporal difference) error

$$\mathop{E}_{w_k}\Big\{\big(g(x_k,u_k,w_k) + \alpha \min_{u \in \mathbb{U}} Q(x_{k+1}, u; r^{\ell-1}) - Q(x_k, u_k; r^{\ell})\big)^2\Big\}.$$

This is typically done using gradient descent algorithms. In particular, approximation architectures with neural networks underpin Deep Reinforcement Learning, which has been a breakthrough in a number of practical applications; see, e.g., [5].

[6]For terminal states, one needs to choose a zero initial action-value function.

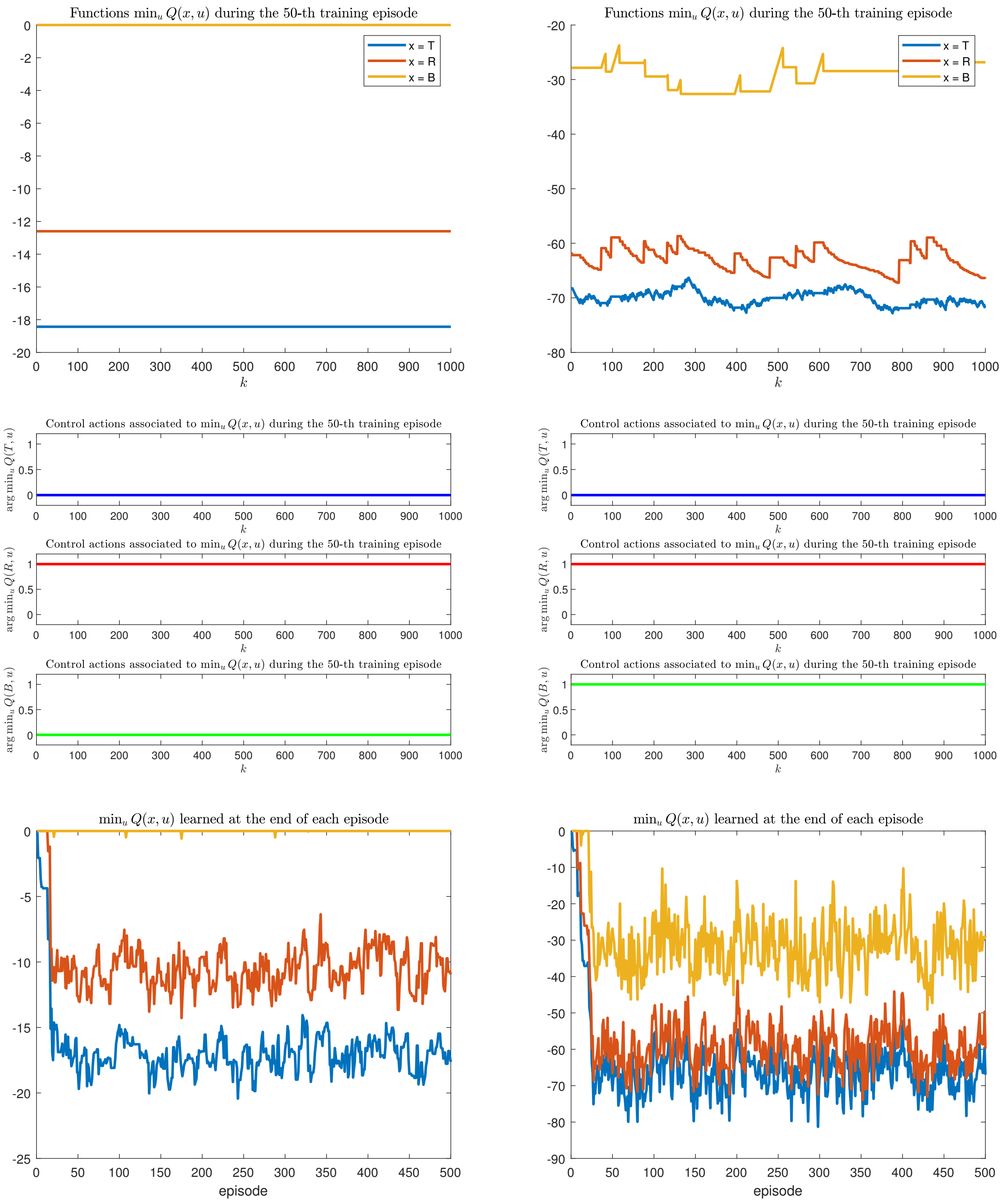

Figure 3: Q-learning for the Rover example: $\alpha = 0.9$ (left), $\alpha = 0.98$ (right)

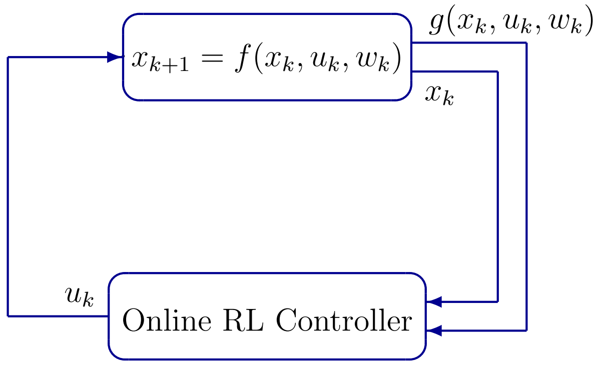

Figure 4: Online learning of control actions

7 Concluding remarks

We have shown how temporal difference prediction ideas can be used in reinforcement learning. The algorithms presented require careful tuning and can be used as a basis for more sophisticated ones. To develop high performance algorithms in practical applications, it is important to incorporate expert knowledge in a suitable manner, e.g., through choice of features or heuristics. One should also avoid trying to learn what one already knows, for example parts of the system dynamics $f(x, u, w)$.

Whilst convergence of some learning algorithms can at times be ensured, this does not directly guarantee stability of the control system. Stability of the control system relates to the state trajectory $\{x_k\}$ and is a notoriously difficult problem, especially when the controller learns whilst acting on the system, see Figure 4. The main difficulties stem from the fact that the learning dynamics and the system dynamics are coupled and that discounted cost functions do not necessarily ensure stability; see, e.g., [3].

References

[1] D. P. Bertsekas, *Reinforcement Learning and Optimal Control.* Athena Scientific, 2019.

[2] D. P. Bertsekas, *Rollout, Policy Iteration, and Distributed Reinforcement Learning.* 2020.

[3] A. S. Leong, A. Ramaswamy, D. E. Quevedo, H. Karl, and L. Shi, "Deep reinforcement learning for wireless sensor scheduling in cyber–physical systems," *Automatica*, Mar. 2020.

[4] L. Ljung, *System Identification. Theory for the User.* Prentice Hall, 2nd ed., 1998.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 529–533, 2015.

[6] T. B. Schön, A. Wills, and B. Ninness, "System identification of nonlinear state-space models," *Automatica*, vol. 47, pp. 39–49, Jan. 2011.

[7] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* MIT Press, 2018.