

10.1 Introduction

All of the learning algorithms described so far—except the clustering algorithms—belong to the class of *supervised learning*. In supervised learning, the agent is supposed to learn a mapping from the input variables to the output variables. Here it is important that for each individual training example, all values for both input variables and output variables are provided. In other words, we need a teacher or a database in which the mapping to be learned is approximately defined for a sufficient number of input values. The sole task of the machine learning algorithm is to filter out the noise from the data and find a function which approximates the mapping well, even between the given data points.

In reinforcement learning the situation is different and more difficult because no training data is available. We begin with a simple illustrative example from robotics, which then is used as an application for the various algorithms.

Reinforcement learning is very valuable in the field of robotics, where the tasks to be performed are frequently complex enough to defy encoding as programs and no training data is available. The robot's task consists of finding out, through trial and error (or success), which actions are *good* in a certain situation and which are not. In many cases we humans learn in a very similar way. For example, when a child learns to walk, this usually happens without instruction, rather simply through reinforcement. Successful attempts at walking are rewarded by forward progress, and unsuccessful attempts are penalized by often painful falls. Positive and negative reinforcement are also important factors in successful learning in school and in many sports (see Fig. 10.1 on page 290).

A greatly simplified movement task is learned in the following example.

Example 10.1 A robot whose mechanism consists only of a rectangular block and an arm with two joints g_y and g_x is shown in Fig. 10.2 on page 290 (see [KMK97]). The robot's only possible actions are the movement of g_y up or down and of g_x right or left. Furthermore, we only allow movements of fixed discrete units (for example,



Fig. 10.1 “Maybe next time I should start turning a bit sooner or go slower?”—Learning from negative reinforcement

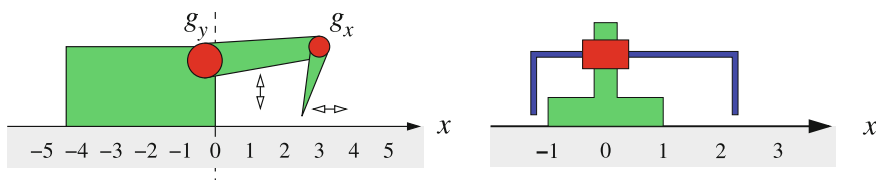


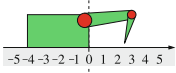
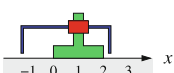
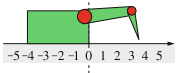
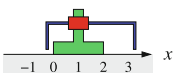
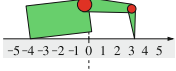



Fig. 10.2 By moving both of its joints, the crawling robot in the *left* of the image can move forward and backward. The walking robot on the *right* side must correspondingly move the frame *up* and *down* or *left* and *right*. The feedback for the robot’s movement is positive for movement to the *right* and negative for movement to the *left*

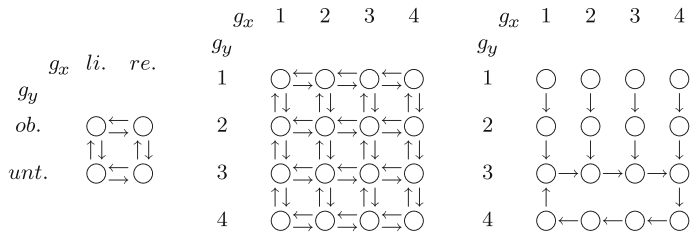
of 10-degree increments). The agent’s task consists of learning a *policy* which allows it to move as quickly as possible to the right. The walking robot in Fig. 10.2 works analogously within the same two-dimensional state space.

A successful action sequence is shown in Table 10.1 on page 291. The action at time $t = 2$ results in the loaded arm moving the body one unit length to the right. Nice animations of this example can be found through [KMK97] and [Tok06].

Before we go into the learning algorithm, we must suitably model the task mathematically. We describe the *state* of the robot by the two variables g_x and g_y for the position of the joints, each with finitely many discrete values. The state of the robot is thus encoded as a vector (g_x, g_y) . The number of possible joint positions is n_x , or respectively n_y . We use the horizontal position of the robot’s body, which can take on real values, to evaluate the robot’s actions. Movements to the right are

Table 10.1 A cycle of a periodic series of movements with systematic forward movement

Crawling robot	Running robot	Time t	State		Reward x	Action a_t
			g_y	g_x		
		0	Up	Left	0	Right
		1	Up	Right	0	Down
		2	Down	Right	0	Left
		3	Down	Left	1	Up



○

○

○

○

→

→

→

→

↑

↑

↑

↑

○

○

○

○

→

→

→

→

↑

↑

↑

↑

○

○

○

○

→

→

→

→

↑

↑

↑

↑

Fig. 10.3 The state space of the example robot in the case of two possible positions for each of the joints (*left*) and in the case of four horizontal and vertical positions for each (*middle*). In the *right image* an optimal policy is given

rewarded with positive changes to x , and movements to the left are punished with negative changes.

In Fig. 10.3 the state space for two variants of this is shown in simplified form.¹ In the left example, both joints have two positions, and in the middle example they each have four positions. An optimal policy is given in Fig. 10.3 right.

10.2 The Task

As shown in Fig. 10.4 on page 292, we distinguish between the agent and its environment. At time t the world, which includes the agent and its environment, is described by a *state* $s_t \in \mathcal{S}$. The set \mathcal{S} is an abstraction of the actual possible states

¹The arm movement space consisting of arcs is rendered as a right-angled grid.

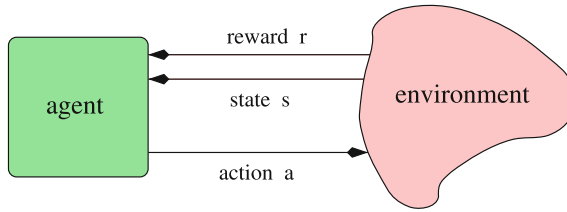


Fig. 10.4 The agent and its interaction with the environment

of the world because, on one hand, the world cannot be exactly described, and on the other hand the agent often only has incomplete information about the actual state because of measurement errors. The agent then carries out an *action* $a_t \in \mathcal{A}$ at time t . This action changes the world and thus results in the state s_{t+1} at time $t+1$. The *state transition function* δ defined by the environment determines the new state $s_{t+1} = \delta(s_t, a_t)$. It cannot be influenced by the agent.

After executing action a_t , the agent obtains immediate reward $r_t = r(s_t, a_t)$ (see Fig. 10.4). The immediate reward $r_t = r(s_t, a_t)$ is always dependent on the current state and the executed action. $r(s_t, a_t) = 0$ means that the agent receives no immediate feedback for the action a_t . During learning, $r_t > 0$ should result in positive and $r_t < 0$ in negative reinforcement of the evaluation of the action a_t in state s_t . In reinforcement learning especially, applications are being studied in which no immediate reward happens for a long time. A chess player for example learns to improve his game from won or lost matches, even if he gets no immediate reward for all individual moves. Here we can see the difficulty of assigning the reward at the end of a sequence of actions to all the actions in the sequence that led to this point (credit assignment problem).

In the crawling robot's case the state consists of the position of the two joints, that is, $s = (g_x, g_y)$. The reward is given by the distance x traveled.

A *policy* $\pi : \mathcal{S} \rightarrow \mathcal{A}$ is a mapping from states to actions. The goal of reinforcement learning is that the agent learns an optimal policy based on its experiences. A policy is optimal if it maximizes reward in the long run, that is, over many steps. But what does "maximize reward" mean exactly? We define the *value*, or the *discounted reward*

$$V^\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{i=0}^{\infty} \gamma^i r_{t+i} \quad (10.1)$$

of a policy π when we start in starting state s_t . Here $0 \leq \gamma < 1$ is a constant, which ensures that future feedback is discounted more the farther in the future that it happens. The immediate reward r_t is weighted the strongest. This reward function is the most predominantly used. An alternative which is sometimes interesting is the average reward

$$V^\pi(s_t) = \lim_{h \rightarrow \infty} \frac{1}{h} \sum_{i=0}^h r_{t+i}. \quad (10.2)$$

A policy π^* is called *optimal*, if for all states s

$$V^{\pi^*}(s) \geq V^{\pi}(s). \quad (10.3)$$

That is, it is at least as good as all other policies according to the defined value function. For better readability, the optimum value function V^{π^*} will be denoted V^* .

The agents discussed here, or their policies, only use information about the current state s_t to determine the next state, and not the prior history. This is justified if the reward of an action only depends on the current state and current actions. Such processes are called *Markov decision processes (MDP)*. In many applications, especially in robotics, the actual state of the agent is not exactly known, which makes planning actions even more difficult. The reason for this may be a noisy sensor signal. We call such a process a *partially observable Markov decision process (POMDP)*.

10.3 Uninformed Combinatorial Search

The simplest possibility of finding a successful policy is the combinatorial enumeration of all policies, as described in Chap. 6. However, even in the simple Example 10.1 on page 289 there are a very many policies, which causes combinatorial search to be associated with extremely high computational cost. In Fig. 10.5 the number of possible actions is given for every state. From that, the number of possible policies is calculated as the product of the given values, as shown in Table 10.2 on page 294.

For arbitrary values of n_x and n_y there are always four corner nodes with two possible actions, $2(n_x - 2) + 2(n_y - 2)$ edge nodes with three actions, and $(n_x - 2)(n_y - 2)$ inner nodes with four actions. Thus there are

$$2^4 3^{2(n_x-2) + 2(n_y-2)} 4^{(n_x-2)(n_y-2)}$$

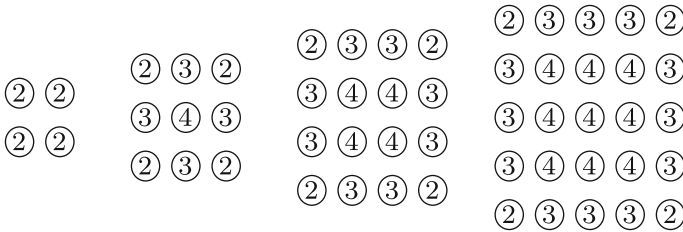


Fig. 10.5 The state space for the example with the values 2, 3, 4, 5 for n_x and n_y . The number of possible actions is given for each state in the respective circles

Table 10.2 Number of policies for differently sized state spaces in the example

n_x, n_y	Number of states	Number of policies
2	4	$2^4 = 16$
3	9	$2^4 3^4 = 5184$
4	16	$2^4 3^8 4^4 \approx 2.7 \times 10^7$
5	25	$2^4 3^{12} 4^9 \approx 2.2 \times 10^{12}$

different policies for fixed n_x and n_y . The number of policies thus grows exponentially with the number of states. This is true in general if there is more than one possible action per state. For practical applications this algorithm is therefore useless. Even heuristic search, described in Chap. 6, cannot be used here. Since the direct reward for almost all actions is zero, it cannot be used as a heuristic evaluation function.

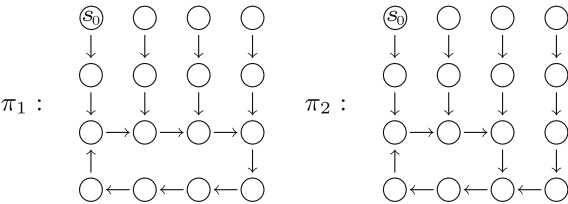
The computational cost rises even higher when we consider that (in addition to enumerating all policies), the value $V^\pi(s)$ must be calculated for every generated policy π and every starting state s . The infinite sum in $V^\pi(s)$ must be cut off for use in a practical calculation; however, due to the exponential reduction of the γ^i factors in (10.1) on page 292, this does not present a problem.

In Example 10.1 on page 289 the difference $x_{t+1} - x_t$ can be used as an immediate reward for an action a_t , which means that every movement of the robot's body to the right is rewarded with 1 and every movement of the robot's body to the left is penalized with -1 . In Fig. 10.6, two policies are shown. Here the immediate reward is zero everywhere other than in the bottom row of the state space. The left policy π_1 is better in the long term because, for long action sequences, the average progress per action is $3/8 = 0.375$ for π_1 and $2/6 \approx 0.333$ for π_2 . If we use (10.1) on page 292 for $V^\pi(s)$, the result is the following table with starting state s_0 at the top left and various γ values:

γ	0.9	0.8375	0.8
$V^{\pi_1}(s_0)$	2.52	1.156	0.77
$V^{\pi_2}(s_0)$	2.39	1.156	0.80

Here we see that policy π_1 is superior to policy π_2 when $gamma = 0.9$, and the reverse is true when $gamma = 0.8$. For $\gamma \approx 0.8375$ both policies are equally good. We can clearly see that a larger γ results in a larger time horizon for the evaluation of policies.

Fig. 10.6 Two different policies for the example



10.4 Value Iteration and Dynamic Programming

In the naive approach of enumerating all policies, much redundant work is performed, because many policies are for the most part identical. They may only differ slightly. Nevertheless every policy is completely newly generated and evaluated. This suggests saving intermediate results for parts of policies and reusing them. This approach to solving optimization problems was introduced as *dynamic programming* by Richard Bellman already in 1957 [Bel57]. Bellman recognized that for an optimal policy it is the case that:

Independent of the starting state s_t and the first action a_t , all subsequent decisions proceeding from every possible successor state s_{t+1} must be optimal.

Based on the so-called Bellman principle, it becomes possible to find a globally optimal policy through local optimization of individual actions. We will derive this principle for MDPs together with a suitable iteration algorithm.

Desired is an optimal policy π^* which fulfills (10.3) on page 293 and (10.1) on page 292. We rewrite the two equations and obtain

$$V^*(s_t) = \max_{a_t, a_{t+1}, a_{t+2}, \dots} (r(s_t, a_t) + \gamma r(s_{t+1}, a_{t+1}) + \gamma^2 r(s_{t+2}, a_{t+2}) + \dots). \quad (10.4)$$

Since the immediate reward $r(s_t, a_t)$ only depends on s_t and a_t , but not on the successor states and actions, the maximization can be distributed, which ultimately results in the following recursive characterization of V^* :

$$\begin{aligned} V^*(s_t) &= \max_{a_t} [r(s_t, a_t) + \gamma \max_{a_{t+1}, a_{t+2}, \dots} (r(s_{t+1}, a_{t+1}) + \gamma r(s_{t+2}, a_{t+2}) + \dots)] \\ &= \max_{a_t} [r(s_t, a_t) + \gamma V^*(s_{t+1})]. \end{aligned} \quad (10.5)$$

Equation (10.5) results from the substitution $t \rightarrow t + 1$ in (10.4). Written somewhat simpler:

$$V^*(s) = \max_a [r(s, a) + \gamma V^*(\delta(s, a))]. \quad (10.6)$$

This equation implies, as does (10.1) on page 292, that, to calculate $V^*(s)$, the immediate reward is added to the reward of all successor states, discounted by the factor γ . If $V^*(\delta(s, a))$ is known, then $V^*(s)$ clearly results by simple local optimization over all possible actions a in state s . This corresponds to the Bellman principle, because of which (10.6) is also called the Bellman equation.

The optimal policy $\pi^*(s)$ carries out an action in state s which results in the maximum value V^* . Thus,

$$\pi^*(s) = \operatorname{argmax}_a [r(s, a) + \gamma V^*(\delta(s, a))]. \quad (10.7)$$

From the recursion equation (10.6) on page 295 an iteration rule for approximating V^* : follows in a straightforward manner:

$$\hat{V}(s) = \max_a [r(s, a) + \gamma \hat{V}(\delta(s, a))]. \quad (10.8)$$

To begin the approximate values $\hat{V}(s)$ for all states are initialized, for example with the value zero. Now $\hat{V}(s)$ is repeatedly updated for each state by recursively falling back on the value $\hat{V}(\delta(s, a))$ of the best successor state. This process of calculating V^* is called *value iteration* and is shown schematically in Fig. 10.7. It can be shown that value iteration converges to V^* [SB98]. An excellent analysis of dynamic programming algorithms can be found in [Sze10], where, based on contraction properties of the particular algorithms (for example value iteration), convergence can be proven using Banach's fixed-point theorem.

In Fig. 10.8 on page 297 this algorithm is applied to Example 10.1 on page 289 with $\gamma = 0.9$. In each iteration the states are processed row-wise from bottom left to top right. Shown are several beginning iterations and in the second image in the bottom row the stable limit values for V^* .

We clearly see the progression of the learning in this sequence. The agent repeatedly explores all states, carries out value iteration for each state and saves the policy in the form of a tabulated function V^* , which then can be further compiled into an efficiently usable table π^* .

Incidentally, to find an optimal policy from V^* it would be wrong to choose the action in state s_t which results in the state with the maximum V^* value. Corresponding to (10.7), the immediate reward $r(s_t, a_t)$ must also be added because we

Fig. 10.7 The algorithm for value iteration

```

VALUEITERATION()
For all  $s \in \mathcal{S}$ 
     $\hat{V}(s) = 0$ 
Repeat
    For all  $s \in \mathcal{S}$ 
         $\hat{V}(s) = \max_a [r(s, a) + \gamma \hat{V}(\delta(s, a))]$ 
    Until  $\hat{V}(s)$  does not change

```

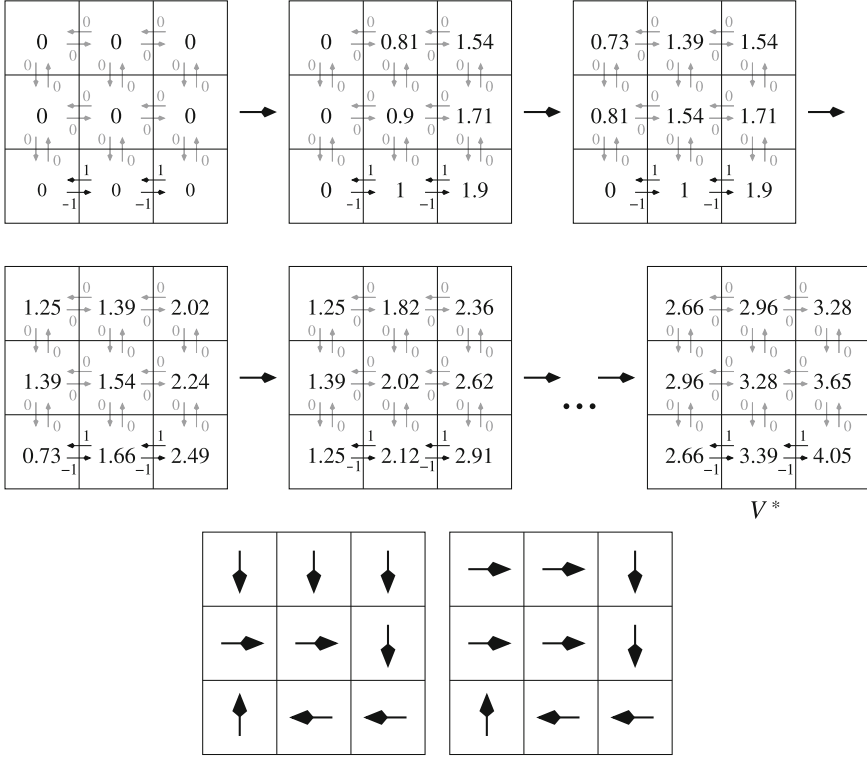



Fig. 10.8 Value iteration in the example with 3×3 states. The *last two images* show two optimal policies. The numbers next to the *arrows* give the immediate reward $r(s, a)$ of each action

are searching for $V^*(s_t)$ and not $V^*(s_{t+1})$. Applied to state $s = (2, 3)$ in Fig. 10.8, this means

$$\begin{aligned}
 \pi^*(2, 3) &= \operatorname{argmax}_{a \in \{\text{left}, \text{right}, \text{up}\}} [r(s, a) + \gamma V^*(\delta(s, a))] \\
 &= \operatorname{argmax}_{\{\text{left}, \text{right}, \text{up}\}} \{1 + 0.9 \cdot 2.66, -1 + 0.9 \cdot 4.05, 0 + 0.9 \cdot 3.28\} \\
 &= \operatorname{argmax}_{\{\text{left}, \text{right}, \text{up}\}} \{3.39, 2.65, 2.95\} = \text{left}.
 \end{aligned}$$

In (10.7) on page 296 we see that the agent in state s_t must know the immediate reward r_t and the successor state $s_{t+1} = \delta(s_t, a_t)$ to choose the optimal action a_t . It must also have a model of the functions r and δ . Since this is not the case for many practical applications, algorithms are needed which can also work without knowledge of r and δ . Section 10.6 is dedicated to such an algorithm.

10.5 A Learning Walking Robot and Its Simulation

A graphical user interface for simple experiments with reinforcement learning is shown in Fig. 10.9 [TEF09]. The user can observe reinforcement learning for differently sized two-dimensional state spaces. For better generalization, back-propagation networks are used to save the state (see Sect. 10.8). The feedback editor shown at the bottom right, with which the user can manually supply feedback about the environment, is especially interesting for experiments. Not shown is the menu for setting up the parameters for value iteration and backpropagation learning.

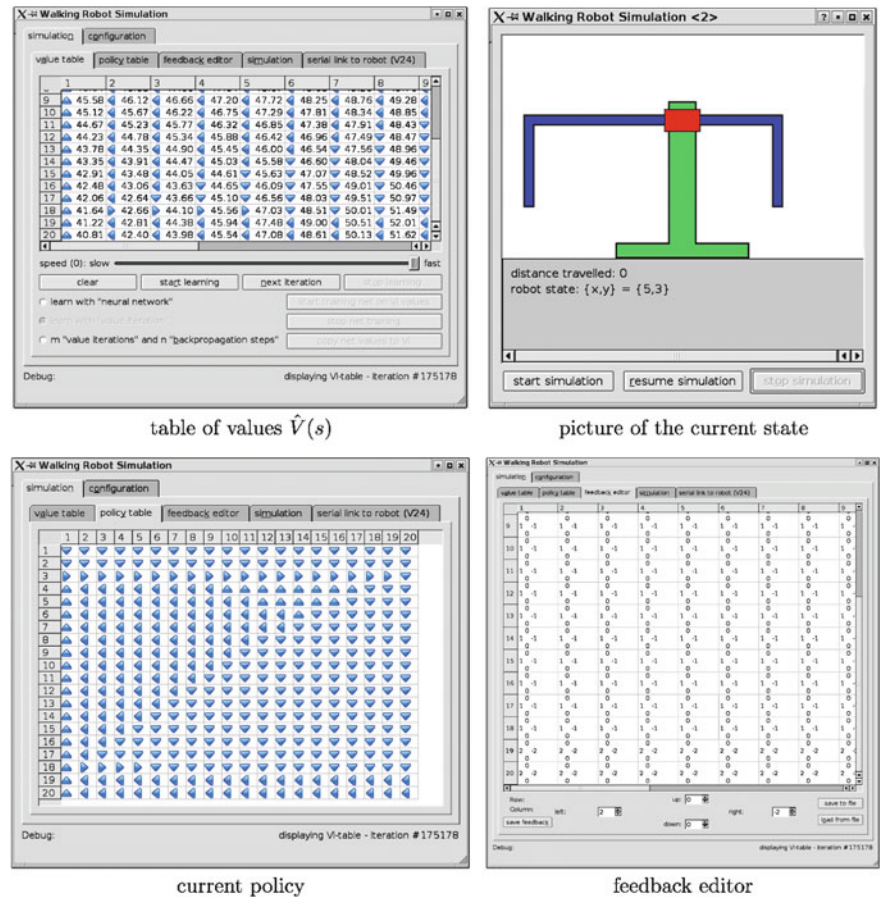


Fig. 10.9 Four different windows of the walking robot simulator

Besides the simulation, two small, real crawling robots with the same two-dimensional discrete state space were developed specifically for teaching [TEF09].² The two robots are shown in Fig. 10.10. Each moves with a servo actuator. The servos are controlled by a microcontroller or through a wireless interface directly from a PC. Using simulation software, the feedback matrix of the robot can be visualized on the PC. With this saved feedback, a policy can be trained on the PC (which computes faster), then loaded again into the robot and executed. However, the robot can also learn autonomously. For a state space of size 5×5 this takes about 30 seconds.

It is interesting to observe the difference between the simulation and the “real” robot. In contrast to the simulation, the crawler learns policies in which it never lifts its arm from the ground, but nonetheless moves forward very efficiently. The reason for this is that, depending on the surface of the underground, the tip of the “underarm” can grip the ground during backward movement, but slides through during forward movement. This effect is very sensibly perceived through the distance measuring sensors and evaluated accordingly during learning.

The robot’s adaptivity results in surprising effects. For example, we can observe how the crawler, despite a defective servo which slips at a certain angle, nonetheless learns to walk (more like hobbling). It is even capable of adapting to changed situations by changing policies. A thoroughly desirable effect is the ability, given differently smooth surfaces (for example, different rough carpets) to learn an optimal policy for each. It also turns out that the real robot is indeed very adaptable given a small state space of size 5×5 .

The reader may (lacking a real robot) model various surfaces or servo defects by varying feedback values and then observing the resulting policies (Exercise 10.3 on page 311).



Fig. 10.10 Two versions of the crawling robot

²Further information and related sources about crawling robots are available through www.hs-weingarten.de/~ertel/kibuch.

10.6 Q-Learning

A policy based on evaluation of possible successor states is clearly not useable if the agent does not have a model of the world, that is, when it does not know which state a possible action leads to. In most realistic applications the agent cannot resort to such a model of the world. For example, a robot which is supposed to grasp complex objects cannot predict whether the object will be securely held in its grip after a gripping action, or whether it will remain in place.

If there is no model of the world, an evaluation of an action a_t carried out in state s_t is needed even if it is still unknown where this action leads to. Thus we now work with an evaluation function $Q(s_t, a_t)$ for states with their associated actions. With this function, the choice of the optimal action is made by the rule

$$\pi^*(s) = \operatorname{argmax}_a Q(s, a). \quad (10.9)$$

To define the evaluation function we again use stepwise discounting of the evaluation for state-action pairs which occur further into the future, just as in (10.1) on page 292. We thus want to maximize $r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$. Therefore, to evaluate action a_t in state s_t we define in analogy to (10.4) on page 295:

$$Q(s_t, a_t) = \max_{a_{t+1}, a_{t+2}, \dots} (r(s_t, a_t) + \gamma r(s_{t+1}, a_{t+1}) + \gamma^2 r(s_{t+2}, a_{t+2}) + \dots). \quad (10.10)$$

Analogously to the approach for value iteration, we bring this equation into a simple recursive form by

$$\begin{aligned} Q(s_t, a_t) &= \max_{a_{t+1}, a_{t+2}, \dots} (r(s_t, a_t) + \gamma r(s_{t+1}, a_{t+1}) + \gamma^2 r(s_{t+2}, a_{t+2}) + \dots) \\ &= r(s_t, a_t) + \gamma \max_{a_{t+1}, a_{t+2}, \dots} (r(s_{t+1}, a_{t+1}) + \gamma r(s_{t+2}, a_{t+2}) + \dots) \\ &= r(s_t, a_t) + \gamma \max_{a_{t+1}} (r(s_{t+1}, a_{t+1}) + \gamma \max_{a_{t+2}} (r(s_{t+2}, a_{t+2}) + \dots)) \\ &= r(s_t, a_t) + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \\ &= r(s_t, a_t) + \gamma \max_{a_{t+1}} Q(\delta(s_t, a_t), a_{t+1}) \\ &= r(s, a) + \gamma \max_a Q(\delta(s, a), a'). \end{aligned} \quad (10.11)$$

What then is the advantage compared to value iteration? The old equation is only slightly rewritten, but this turns out to be exactly the right approach to a new algorithm. Instead of saving V^* , now the function Q is saved, and the agent can choose its actions from the functions δ and r without a model of the world. We still do not have a process, however, which can learn Q directly, that is, without knowledge of V^* .

From the recursive formulation of $Q(s, a)$, an iteration algorithm for determining $Q(s, a)$ can be derived in a straightforward manner. We initialize a table $\hat{Q}(s, a)$ for all states arbitrarily, for example with zeroes, and iteratively carry out

```

Q-LEARNING()
For all  $s \in \mathcal{S}, a \in \mathcal{A}$ 
     $\hat{Q}(s, a) = 0$  (or randomly)
Repeat
    Select (e.g. randomly) a state  $s$ 
    Repeat
        Select an action  $a$  and carry it out
        Obtain reward  $r$  and new state  $s'$ 
         $\hat{Q}(s, a) := r(s, a) + \gamma \max_{a'} \hat{Q}(s', a')$ 
         $s := s'$ 
    Until  $s$  is an ending state Or time limit reached
Until  $\hat{Q}$  converges

```

Fig. 10.11 The algorithm for Q-learning

$$\hat{Q}(s, a) = r(s, a) + \gamma \max_{a'} \hat{Q}(\delta(s, a), a'). \quad (10.12)$$

It remains to note that we do not know the functions r and δ . We solve this problem quite pragmatically by letting the agent in its environment in state s carry out action a . The successor state is then clearly $\delta(s, a)$ and the agent receives its reward from the environment. The algorithm shown in Fig. 10.11 implements this algorithm for Q-learning.

The application of the algorithm to Example 10.1 on page 289 with $\gamma = 0.9$ and $n_x = 3, n_y = 2$ (that is, in a 2×3 grid) is shown in Fig. 10.12 on page 302 as an example. In the first picture, all Q values are initialized to zero. In the second picture, after the first action sequence, the four r values which are not equal to zero become visible as Q values. In the last picture, the learned optimal policy is given. The following theorem, whose proof is found in [Mit97], shows that this algorithm converges not just in the example, but in general.

Theorem 10.1 *Let a deterministic MDP with limited immediate reward $r(s, a)$ be given. Equation (10.12) with $0 \leq \gamma < 1$ is used for learning. Let $\hat{Q}_n(s, a)$ be the value for $\hat{Q}(s, a)$ after n updates. If each state-action pair is visited infinitely often, then $\hat{Q}_n(s, a)$ converges to $Q(s, a)$ for all values s and a for $n \rightarrow \infty$.*

Proof Since each state-action transition occurs infinitely often, we look at successive time intervals with the property that, in every interval, all state-action transitions occur at least once. We now show that the maximum error for all entries in the \hat{Q} table is reduced by at least the factor γ in each of these intervals. Let

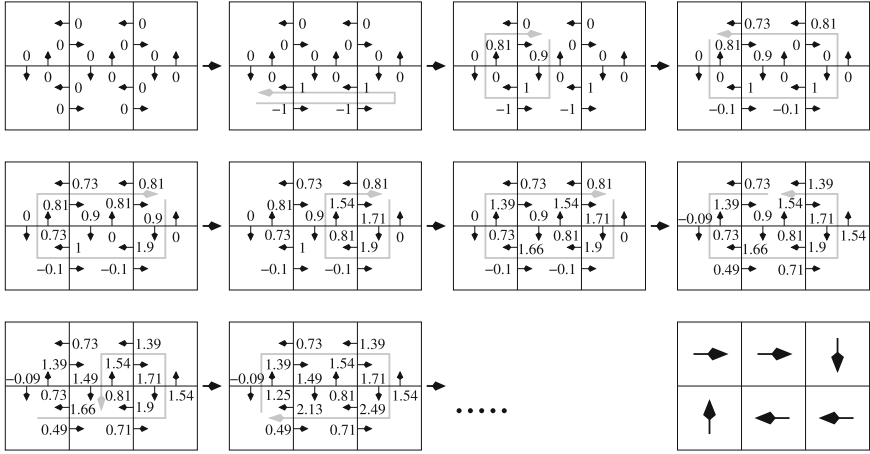


Fig. 10.12 Q-learning applied to the example with $n_x = 3$, $n_y = 2$. The gray arrows mark the actions carried out in each picture. The updated Q values are given. In the last picture, the current policy, which is also optimal, is shown

$$\Delta_n = \max_{s,a} |\hat{Q}_n(s,a) - Q(s,a)|$$

be the maximum error in the table \hat{Q}_n and $s' = \delta(s, a)$. For each table entry $\hat{Q}_n(s, a)$ we calculate its contribution to the error after an interval as

$$\begin{aligned} |\hat{Q}_{n+1}(s, a) - Q(s, a)| &= |(r + \gamma \max_{a'} \hat{Q}_n(s', a')) - (r + \gamma \max_{a'} Q(s', a'))| \\ &= \gamma |\max_{a'} \hat{Q}_n(s', a') - \max_{a'} Q(s', a')| \\ &\leq \gamma \max_{a'} |\hat{Q}_n(s', a') - Q(s', a')| \\ &\leq \gamma \max_{s''a''} |\hat{Q}_n(s'', a'') - Q(s'', a'')| = \gamma \Delta_n. \end{aligned}$$

The first inequality is true because, for arbitrary functions f and g ,

$$l |\max_x f(x) - \max_x g(x)| \leq \max_x |f(x) - g(x)|$$

and the second inequality is true because, by additional variation of the state s'' , the resulting maximum cannot become smaller. Thus it has been shown that $\Delta_{n+1} \leq \gamma \Delta_n$. Since the error in each interval is reduced by a factor of at least γ , after k intervals it is at most $\gamma^k \Delta_0$, and, as a result, Δ_0 is bounded. Since each state is visited infinitely many times, there are infinitely many intervals and Δ_n converges to zero. \square

According to Theorem 10.1 on page 301 Q-learning converges independently of the actions chosen during learning. This means that for convergence it does not matter which actions the agent chooses, as long as each is executed infinitely often. The speed of convergence, however, certainly depends on which paths the agent takes during learning (see Sect. 10.7).

10.6.1 Q-Learning in a Nondeterministic Environment

In many robotics applications, the agent's environment is nondeterministic. This means that the reaction of the environment to the action a in state s at two different points in time can result in different successor states and rewards. Such a nondeterministic Markov process is modeled by a probabilistic transition function $\delta(s, a)$ and probabilistic immediate reward $r(s, a)$. To define the Q function, each time the expected value must be calculated over all possible successor states. Equation (10.11) on page 300 is thus generalized to

$$Q(s_t, a_t) = E(r(s, a)) + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q(s', a'), \quad (10.13)$$

where $P(s'|s, a)$ is the probability of moving from state s to the successor state s' with action a . Unfortunately there is no guarantee of convergence for Q-learning in the nondeterministic case if we proceed as before according to (10.12) on page 301. This is because, in successive runs through the outer loop of the algorithm in Fig. 10.11 on page 301, the reward and successor state can be completely different for the same state s and same action a . This may result in an alternating sequence which jumps back and forth between several values. To avoid this kind of strongly jumping Q values, we add the old weighted Q value to the right side of (10.12) on page 301. This stabilizes the iteration. The learning rule then reads

$$\hat{Q}_n(s, a) = (1 - \alpha_n) \hat{Q}_{n-1}(s, a) + \alpha_n \left[r(s, a) + \gamma \max_{a'} \hat{Q}_{n-1}(\delta(s, a), a') \right] \quad (10.14)$$

with a time-varying weighting factor

$$\alpha_n = \frac{1}{1 + b_n(s, a)}.$$

The value $b_n(s, a)$ indicates how often the action a was executed in state s at the n th iteration. For small values of b_n (that is, at the beginning of learning) the stabilizing term $\hat{Q}_{n-1}(s, a)$ does not come into play, for we want the learning process to make quick progress. Later, however, b_n gets bigger and thereby prevents excessively large jumps in the sequence of \hat{Q} values. When integrating (10.14) into Q-learning, the values $b_n(s, a)$ must be saved for all state-action pairs. This can be accomplished by extending the table of \hat{Q} values.

For a better understanding of (10.14) on page 303, we simplify this by assuming $\alpha_n = \alpha$ is a constant and transforming it as follows:

$$\begin{aligned}\hat{Q}_n(s, a) &= (1 - \alpha)\hat{Q}_{n-1}(s, a) + \alpha[r(s, a) + \gamma \max_{a'} \hat{Q}_{n-1}(\delta(s, a), a')] \\ &= \hat{Q}_{n-1}(s, a) + \underbrace{\alpha[r(s, a) + \gamma \max_{a'} \hat{Q}_{n-1}(\delta(s, a), a') - \hat{Q}_{n-1}(s, a)]}_{\text{TD-error}}.\end{aligned}$$

The new Q value $\hat{Q}_n(s, a)$ can clearly be represented as the old $\hat{Q}_{n-1}(s, a)$ plus α times a correction term which is the same as the Q value's change in this step. The correction term is called the TD-error, or temporal difference error, and the above equation for changing the Q value is a special case of TD-Learning, an important class of learning algorithms [SB98]. For $\alpha = 1$ we obtain the Q-learning described above. For $\alpha = 0$ the \hat{Q} values are completely unchanged. Thus no learning takes place.

10.7 Exploration and Exploitation

For Q-learning so far, only a coarse algorithm schema has been given. Especially lacking is a description of the choice of the starting state each time and the actions to be carried out in the inner loop of Fig. 10.11 on page 301. For the selection of the next action there are two possibilities. Among the possible actions, one can be chosen randomly. In the long term this results in a uniform exploration of all possible actions or policies, but with very slow convergence. An alternative to this is the exploitation of previously learned \hat{Q} values. Here the agent always chooses the action with the highest \hat{Q} value. This results in relatively fast convergence of a specific trajectory. Other paths, however, remain unvisited all the way to the end. In the extreme case then we can obtain non-optimal policies. In Theorem 10.1 on page 301 it is therefore required that every state-action pair is visited infinitely many times. It is recommended to use a combination of exploration and exploitation with a high exploration portion at the beginning and reduce it more and more over time.

The choice of the starting state also influences the speed of learning. In the first three pictures in Fig. 10.12 on page 302 we can clearly see that, for the first iterations, only the Q values in the immediate vicinity of state-action pairs are changed by immediate reward. Starting farther away from this kind of point results in much unnecessary work. This suggests transferring prior knowledge about state-action pairs with immediate reward into starting states nearby these points. In the course of learning then more distant starting states can be selected.

10.8 Approximation, Generalization and Convergence

As Q-learning has been described so far, a table with all Q values is explicitly saved in a table. This is only possible when working with a finite state space with finitely many actions. If the state space is infinite, however, for example in the case of continuous variables, then it is neither possible to save all Q values nor to visit all state-action pairs during learning.

Nonetheless there is a simple way of using Q-learning and value iteration on continuous variables. The $Q(s, a)$ table is replaced by a neural network, for example a backpropagation network with the input variables s , a and the Q value as the target output. For every update of a Q value, the neural network is presented a training example with (s, a) as input and $Q(s, a)$ as target output. At the end we have a finite representation of the function $Q(s, a)$. Since we only ever have finitely many training examples, but the function $Q(s, a)$ is defined for infinitely many inputs, we thus automatically obtain a generalization if the network size is chosen appropriately (see Chap. 9). Instead of a neural network, we can also use another supervised learning algorithm or a function approximator such as a support vector machine or a Gaussian process.

However, the step from finitely many training examples to a continuous function can become very expensive in certain situations. Q-learning with function approximation might not converge because Theorem 10.1 on page 301 is only true if each state-action pair is visited infinitely often.

However, convergence problems can also come up in the case of finitely many state-action pairs when Q-learning is used on a POMDP. Q-learning can be applied—in both described variants—to deterministic and nondeterministic Markov processes (MDPs). For a POMDP it can happen that the agent, due to noisy sensors for example, perceives many different states as one. Often many states in the real world are purposefully mapped to one so-called *observation*. The resulting observation space is then much smaller than the state space, whereby learning becomes faster and overfitting can be avoided (see Sect. 8.4.7).

However, by bundling together multiple states, the agent can no longer differentiate between the actual states, and an action may lead it into many different successor states, depending on which state it is really in. This can lead to convergence problems for value iteration or for Q-learning. In the literature (e.g., in [SB98]) many different approaches to a solution are suggested.

Also very promising are so-called policy improvement methods and their derived policy gradient methods, in which Q values are not changed, but rather the policy is changed directly. In this scheme a policy is searched for in the space of all policies, which maximizes the cumulative discounted reward ((10.1) on page 292). One possibility of achieving this is by following the gradient of the cumulative reward to a maximum. The policy found in this way then clearly optimizes the cumulative reward. In [PS08] it is shown that this algorithm can greatly speed up learning in applications with large state spaces, such as those which occur for humanoid robots.

10.9 Applications

The practical utility of reinforcement learning has meanwhile been shown many times over. From a large number of examples of this, we will briefly present a small selection.

TD-learning, together with a backpropagation network with 40 to 80 hidden neurons was used very successfully in TD-gammon, a backgammon-playing program [Tes95]. The only immediate reward for the program is the result at the end of the game. An optimized version of the program with a two-move lookahead was trained against itself in 1.5 million games. It went on to defeat world-class players and plays as well as the three best human players.

There are many applications in robotics. For example, in the RoboCup Soccer Simulation League, the best robot soccer teams now successfully use reinforcement learning [SSK05, Robb]. Balancing a pole, which is relatively easy for a human, has been solved successfully many times with reinforcement learning.

An impressive demonstration of the learning ability of robots was given by Russ Tedrake at IROS 2008 in his presentation about a model airplane which learns to land at an exact point, just like a bird landing on a branch [Ted08]. Because air currents become very turbulent during such highly dynamic landing approach, the associated differential equation, the Navier–Stokes equation, is unsolvable. Landing therefore cannot be controlled in the classical mathematical way. Tedrake’s comment about this:

“Birds don’t solve Navier–Stokes!”

Birds can clearly learn to fly and land even without the Navier–Stokes equation. Tedrake showed that this is now also possible for airplanes.

Today it is also possible to learn to control a real car in only 20 minutes using Q-learning and function approximation [RMD07]. This example shows that real industrial applications in which few measurements must be mapped to actions can be learned very well in short time.

Real robots still have difficulty learning in high-dimensional state-action spaces because, compared to a simulation, real robots get feedback from the environment relatively slowly. Due to time limitations, the many millions of necessary training cycles are therefore not realizable. Here, besides fast learning algorithms, methods are needed which allow at least parts of the learning to happen offline, that is, without feedback from the environment.

10.10 AlphaGo, the Breakthrough in Go

Although alpha-beta pruning has been used successfully by chess computers, it cannot achieve the same success in Go programs due to the game’s large branching factor of roughly 250, as was described in Sect. 6.6.2. It has been known for some time that in Go the next move should be chosen using pattern recognition

algorithms on the current board position. Yet all prior attempts had not been very successful. This changed when Google DeepMind presented AlphaGo in [SHM⁺16]. This program uses Monte Carlo tree search (MCTS) to generate training data, deep learning to evaluate board positions and reinforcement learning to improve its strategy by playing against itself. Roughly sketched, the algorithm works as follows:

The learning process's most important goal is to learn a policy p which, for a position s , calculates the probability $p(a|s)$ of winning for every possible move a . If this probability function is found, then it will be used in the game such that when in a position s , the best of all possible moves

$$a^* = \operatorname{argmax}_a p(a|s)$$

will be chosen. Because the program should play better than its human role model in the end, it is a multi-stage process to learn a policy p that is as good as possible. First, using saved champion-level games, two different move selection functions are learned (stage 1). The stronger of the two will then be further improved using reinforcement learning (stage 2) and transformed into a position evaluation function (stage 3). Finally, the program plays by combining the resulting position evaluation function with the two move selection policies from stage 1 in a complex MCTS search process.

Stage 1: Deep Learning Based on Saved Games The KGS Go Server contains many complete Go matches by champion players with a total of thirty million board positions s and the players' respective moves $a(s)$. All of these moves are used to train a convolutional neural network (CNN).

The CNN takes a board position s as input and, after training, should return the probability $p_\sigma(a|s)$ for all legal moves a . The de facto realization of $p_\sigma(a|s)$ is a 19×19 matrix, which represents the board. The values for each of the 361 points on the board stand for the probability of winning for the corresponding move. An example for the application of such a policy function can be seen in Fig. 10.13 on page 308. For the board state s left, $p_\sigma(a|s)$ is represented in a density graph on the right. On this turn, the player may set a white stone on one of the points. For nearly all of the available points, the probability of winning is zero, but for a few interesting points, the probability is represented as a grayscale value. Apparently the most desirable place to put the stone is in the top middle in row 2, column 11.

During training, the 19×19 board state for each of the thirty million champion moves in the database is given as input to the network, and the target output is the 19×19 matrix with all values set to null and the square selected by the expert set to one.

On a holdout set of test data it has not yet seen, the complex thirteen-layer CNN network used by AlphaGo chooses the correct move up to 57% of the time, which demonstrates an impressive gain over the best performance to date of about 44% by other Go programs.

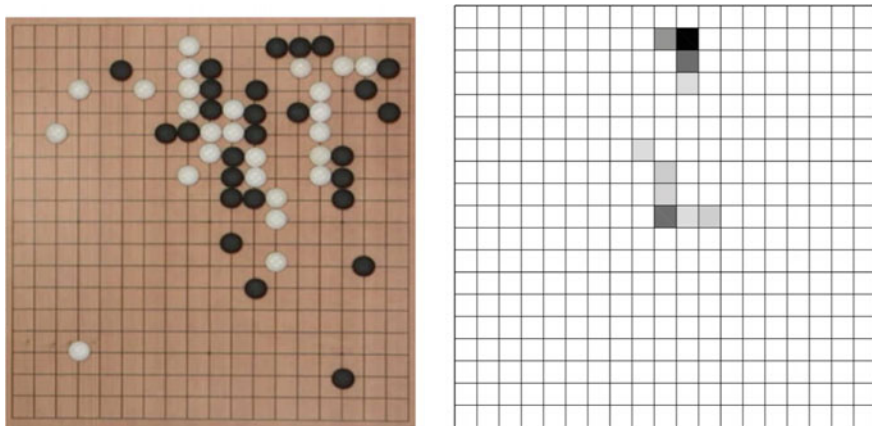


Fig. 10.13 Density graph (right) of a move probability $p_{\sigma}(a|s)$ for the player with white stones in the board state shown on the left. For each white square, $p_{\sigma}(a|s) = 0$. The darker the square, the higher the value for a move by white onto the corresponding grid point

In addition to the policy $p_{\sigma}(a|s)$, a much simpler *rollout policy* $p_{\pi}(a|s)$ is trained on the same training data with a simpler CNN. It chooses moves correctly only 24% of the time, but does calculations about 1000 times faster.

Stage 2: Improving the Learned Policy with Reinforcement Learning Next, the move selection function is improved using reinforcement learning. This second step is necessary in order for the program to play better than its human role models.

The policy $p_{\sigma}(a|s)$ learned from the database is used by AlphaGo to play against itself. After every game, the current policy $p_{\rho}(a|s)$ is improved using stochastic gradient descent. To avoid overfitting, AlphaGo does not always play against the current version, rather against a randomly selected earlier one.

Stage 3: Learning a Position's Value Next, the current policy $p_{\rho}(a|s)$ is used to train a board state evaluation function $V(s)$ using value iteration (see Sect. 10.4).

The Final AlphaGo Game Policy and its Performance AlphaGo's actual playing algorithm uses a complex MCTS algorithm in which the tree is expanded a bit from the current position. From the leaf nodes thus generated, the game is played to the end using a fast but simple rollout policy $p_{\pi}(a|s)$. The position is then evaluated using the simulated game's outcome together with the value function $V(s)$. Despite having a very good policy, the computational cost is immense because of the complexity of this MCTS algorithm. AlphaGo's highest playing ability, that was used to defeated Go grandmaster Lee Sedol of Korea 4:1, was reached on a parallel computer with 1202 CPUs and 176 GPUs.

In summary, AlphaGo represents a great milestone in the history of AI. This achievement was made possible by using deep learning and by the astounding engineering effort of a large team of experts in the various areas of machine learning.

10.11 Curse of Dimensionality

Despite success in recent years, reinforcement learning remains an active area of research in AI, not least because even the best learning algorithms known today are still impractical for high-dimensional state and action spaces due to their gigantic computation time. This problem is known as the “curse of dimensionality”.

In the search for solutions to this problem, scientists observe animals and humans during learning. Here we notice that learning in nature takes place on many levels of abstraction. A baby first learns simple motor and language skills on the lowest level. When these are well learned, they are saved and can later be called up any time and used. Translated into the language of computer science, this means that every learned ability is encapsulated in a module and then, on a higher level, represents an action. By using such complex actions on a higher level, the action space becomes greatly reduced and thus learning is accelerated. In a similar way, states can be abstracted and thus the state space can be shrunk. This learning on multiple levels is called hierarchical learning [BM03].

Another approach to modularization of learning is distributed learning, or multi-agent learning [PL05]. When learning a humanoid robot’s motor skills, up to 50 different motors must be simultaneously controlled, which results in 50-dimensional state space and also a 50-dimensional action space. To reduce this gigantic complexity, central control is replaced by distributed control. For example, each individual motor could get an individual control which steers it directly, if possible independently of the other motors. In nature, we find this kind of control in insects. For example, the many legs of a millipede are not steered by a central brain, rather each pair of legs has its own tiny “brain”.

Similar to uninformed combinatorial search, reinforcement learning has the task of finding the best of a huge number of policies. The learning task becomes significantly easier if the agent has a more or less good policy before learning begins. Then the high-dimensional learning tasks can be solved sooner. But how do we find such an initial policy? Here there are two main possibilities.

The first possibility is classical programming. The programmer provides the agent with a policy comprising a program which he considers good. Then a switchover occurs, for example to Q-learning. The agent chooses, at least at the beginning of learning, its actions according to the programmed policy and thus is led into “interesting” areas of the state-action space. This can lead to dramatic reductions in the search space of reinforcement learning.

If traditional programming becomes too complex, we can begin training the robot or agent by having a human proscribe the right actions. In the simplest case, this is done by manual remote-control of the robot. The robot then saves the proscribed action for each state and generalizes using a supervised learning algorithm such as backpropagation or decision tree learning. This so-called demonstration learning [BCDS08, SE10] thus also provides an initial policy for the subsequent reinforcement learning.

10.12 Summary and Outlook

Today we have access to well-functioning and established learning algorithms for training our machines. The task for the human trainer or developer, however, is still demanding for complex applications. There are namely many possibilities for how to structure the training of a robot and it will not be successful without experimentation. This experimentation can be very tedious in practice because each new learning project must be designed and programmed. Tools are needed here which, besides the various learning algorithms, also offer the trainer the ability to combine these with traditional programming and demonstration learning. One of the first of this kind of tool is the Teaching-Box [ESCT09], which in addition to an extensive program library also offers templates for the configuration of learning projects and for communication between the robot and the environment. For example, the human teacher can give the robot further feedback from the keyboard or through a speech interface in addition to feedback from the environment.

Reinforcement learning is a fascinating and active area of research that will be increasingly used in the future. More and more robot control systems, but also other programs, will learn through feedback from the environment. Today there exist a multitude of variations of the presented algorithms and also completely different algorithms. The scaling problem remains unsolved. For small action and state spaces with few degrees of freedom, impressive results can be achieved. If the number of degrees of freedom in the state space grows to 18, for example for a simple humanoid robot, then learning becomes very expensive.

For further foundational lectures, we recommend the compact introduction into reinforcement learning in Tom Mitchell's book [Mit97]. The standard work by Sutton and Barto [SB98] is thorough and comprehensive, as is the survey article by Kaelbling, Littman and Moore [KLM96].

10.13 Exercises

Exercise 10.1

- (a) Calculate the number of different policies for n states and n actions. Thus transitions from each state to each state are possible.
- (b) How does the number of policies change in subproblem (a) if empty actions, i.e., actions from one state to itself, are not allowed.
- (c) Using arrow diagrams like those in Fig. 10.3 on page 291, give all policies for two states.
- (d) Using arrow diagrams, give all policies without empty actions for three states.

Exercise 10.2 Use value iteration manually on Example 10.1 on page 289 with $n_x = n_y = 2$.

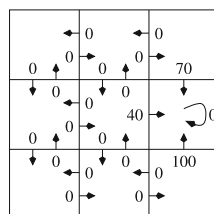
Exercise 10.3 Carry out various experiments using a value iteration simulator.

- Install the value iteration simulator from [Tok06].
- Reproduce the results from Exercise 10.2 on page 310 by first putting in the feedback with the feedback editor and then carrying out value iteration.
- Model surfaces of differing smoothness and observe how the policy changes.
- With a similar feedback matrix, enlarge the state space incrementally up to about 100×100 and fit the discount factor γ such that a sensible policy results.

* **Exercise 10.4** Show that for the example calculation in Fig. 10.8 on page 297 the exact value is $V^*(3, 3) = 1.9/(1 - 0.9^6) \approx 4.05499$.

Exercise 10.5

Carry out Q-learning on the 3×3 grid on the right. The state in the middle right is an absorbing goal state.



Exercise 10.6 A robot arm with n joints (dimensions) and ℓ discrete states per joint is given. Actions from each state to each state are possible (if the robot does nothing, this is evaluated as an (empty) action).

- Give a formula for the number of states and the number of actions in each state and for the number of policies for the robot.
- Create a table with the number of strategies for $n = 1, 2, 3, 4, 8$ and $\ell = 1, 2, 3, 4, 10$.
- To reduce the number of possible strategies, assume that the number of possible actions per joint is always equal to 2 and that the robot can only move one joint at a time. Give a new formula for the number of strategies and create the associated table.
- With the calculated result, justify that an agent which operates autonomously and adaptively with $n = 8$ and $\ell = 10$ can certainly be called intelligent.