10 Markov Decision Process

This chapter is an introduction to a generalization of supervised learning where feedback is only given, possibly with delay, in form of reward or punishment. The goal of this **reinforcement learning** is for the agent to figure out which actions to take to maximize future payoff (accumulation of rewards). We introduce in this chapter the general idea and basic formulation of such a problem domain, and we will then then concentrate on the case of a Markov Decision Process (MDP). These processes are characterized by completely observable states and by transition processes that only depend on the last state of the agent. In the next chapters this will be extended this framework to partially observable situations and temporal difference (TD) learning.

10.1 Learning from reward and the credit assignment problem

We discussed in previous chapters supervised learning in which a teacher showed an agent the desired response $\mathbf{y}$ to a given input state $\mathbf{x}$. We are now moving to the problem when the agent must discover the right action to choose and only receives some qualitative feedback from the environment such as reward or punishment at a later time. The reward feedback does not tell the agent directly which action to take. Rather, it indicates how valuable some sequences of states and action are. The agent has to discover the right sequence of actions to optimize the reward over time. Choosing the right action of an agent is traditionally the subject of control theory, and this subject is thus often discussed in the context of optimal control.

Reward learning introduces several challenges. For example, in typical circumstances reward is only received after a long sequence of actions. The problem is then how to assign the credit for the reward to specific actions. This is the **temporal credit assignment problem**. To illustrate this, let us think about a car that crashed into a wall. It is likely that the driver used the breaks before the car crashed into the wall, though the breaks could not prevent the accident. However, from this we should not conclude that breaking is not good and lead to crashes. In some distributed systems there is, in addition, a **spatial credit assignment problem** which is the problem of how to assign the appropriate credit when different parts of a system contributed to a specific outcome or which state and action combinations should be given credit for the outcome.

Another challenge in reinforcement learning is the balance between **exploitation** and **exploration**. That is, we might find a way to receive some small food reward if we repeat certain actions, but if we only repeat these specific actions, we might never discover a bigger reward following different actions. Some escape from self-reinforcement is important.

The idea of reinforcement learning is to use the reward feedback to build up a **value function** that reflect the expected future payoff of visiting certain states and taking certain actions. We can use such a value function to make decisions of which action to take and thus which states to visit. This is called a **policy**. To formalize these ideas we start with simple processes where the transitions to new states depend only on the current state. A process which such a characteristics is called a **Markov process**. In addition to the Markov property, we also assume in this chapter that the agent has full knowledge the environment. Finally, it is again important that we acknowledge uncertainties and possible errors. For example, we can take error in motor commands into account by considering probabilistic state transitions.

10.2 The Markov Decision Process

Before formalizing the decision processes in this chapter, let us begin with an example to illustrate a common setting. In this example we consider an agent that should learn to navigate through the maze shown in Figure 10.1. The states of the maze are the possible discrete positions that are simply numbered consecutively in this example, that is, $S = \{1, 2, ..., 18\}$. The possible actions of the agent is to move one step forward, either to the north, east, south or west, that is, $A = \{N, E, S, W\}$. However, even though the agents gives these commands to its actuators, stochastic circumstances – such as faulty hardware or environmental conditions (e.g. some instructor 'kicking' the agent) – make the agent end up in different states with certain probabilities. The probabilities are specified by a transition matrix $T(s'|s, a)$. For example, the probability of following actions $a = N$ might just be 80% as it might end up in the west state (taking actions $a = W$) or the east state (taking action and $a = E$) in 10% of the cases each and never goes erroneously south. We assume for now that the transition probability is given explicitly, although in many practical circumstances we might need to estimate this from examples (e.g. supervised learning).

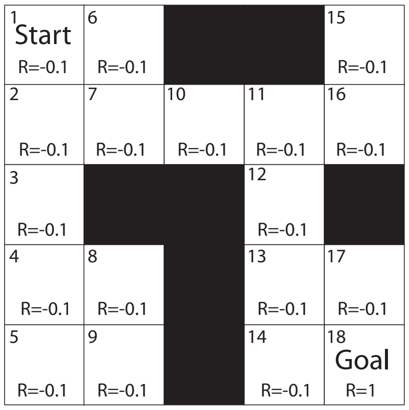


Fig. 10.1 A maze where each state is rewarded with a value r.

In the maze as illustrated in Figure 10.1, some of the states are not reachable as they represent a wall. We can take this into account by making the transition matrix state dependent. Here we simply consider that when taking an action that would drive the robot into a wall would simply though back the robot in the starting position.

Finally, the agent is given reward or punishment when the agent is moving into a new state s. For example, we can consider a deterministic reward function in which the agent is given a large reward when finding the exit to the maze ($r(18) = 1$ in the example of Figure 10.1). In practice it is also common and useful to give some small negative reward to the other states. This could, for example, represent the battery resource that the Lego robot consumes when moving to a cell in the grid, whereas it gets recharged at the exit of the maze.

A common approach to solve a deterministic maze navigation problem is path planing based on some search algorithms such as the A^* search algorithm. However, the environment here is stochastic. The probabilistic nature of the state transition is challenging for traditional search algorithms, although this can be accomplished with some dynamic extensions of the standard search algorithms. In addition the task might not be known to the agent explicitly. In other words, the agent must discover by itself the task of completing the maze. The great thing about reinforcement learning is that we can apply the such a learning system to many different situation by guiding the system with reward feedback. We can even change the task by changing the reward feedback. There should be no need to change anything in the program of the agent. Such training is typical when training animals as reward feedback is usually the main way to communicate with the animals in learning situations since we can not verbally communicate the goal of the task that we have in mind.

We now formalize such an environment as a **Markov Decision Process (MDP)**. A MDP is characterized by a set of 5 quantities, expressed as $(S, A, T(s'|s,a), R(r|s,a), \theta)$. The meaning of these quantities are as follows.

- S is a set of states.
- A is a set of actions.
- $T(s'|s,a)$ is a **transition probability**, for reaching state s' when taking action a from state s. This transition probability only depends on the previous state, which is called the Markov condition; hence the name of the process.
- $R(r|s)$ is the probability of receiving **reward** when getting to state s. This quantity provides feedback from the environment. r is a numeric value with positive values indicating reward and negative values indicating punishment.
- θ are specific parameters for some of the different kinds of RL settings. This will be the **discount factor** γ in our first examples.

An MDP is fully determined by these 5 quantities that characterize the environment completely.

10.3 Value functions and policies

To make decisions we define two quantities that will guide the behaviour of an agent. The first quantities is the **value function** $Q^\pi(s,a)$ that specifies how valuable state s is under the policy π for different actions a. This quantity is defined as the **expected future reward** as formalized below. The second quantity is the **policy** $\pi(a|s)$ which is the probability of choosing action a from state s. Note that we have kept the formulation here very general by considering probabilistic rewards and probabilistic policies, although some applications can be formulated with deterministic functions for these quantities. Since the action is uniquely specified for deterministic policies, one can use the **state value function** $V^\pi(s)$. Note that this function is still specific for an action as specified by the policy $a = \pi(s)$. The function $Q^\pi(s,a)$ is often called the **state-action value function** to distinguish it from $V^\pi(s)$. Finally, we consider here rewards that only depend on the state. In same rare cases reward might depend on the way a state is reached, in which case the reward probability can be easily extended to R(r|s,a).

Reinforcement learning algorithms are aimed at calculating or estimating value functions to determine useful actions. However, most of the time we are mostly interested in finding the best or **optimal policy**. Since choosing the right actions from states is the aim of control theory, this is sometimes called **optimal control**. The optimal policy is the policy which maximizes the value (expected reward) for each state. Thus, if we denote the maximal value as

$$Q^*(s,a) = \max_\pi Q^\pi(s,a), \tag{10.1}$$

the optimal policy is the policy that maximizes the expected reward,

$$\pi^*(a|s) = \arg\max_\pi Q^\pi(s,a). \tag{10.2}$$

While direct search in the space of all possible policies is possible in examples with small sets of states and actions, a major problem of reinforcement learning is the exploding number of policies and states with increasing dimension. This was termed the **'course of dimensionality'** by Richard Bellman. Solving the course of dimensionality problem is a major challenge for practical applications. We will get back to this point later.

We have not yet specified how we define the values. The value function is defined as the expected value of all future rewards, also called the **total payoff**. The total payoff is the sum of all future reward, that is, the immediate reward of reaching state s as well as the rewards of subsequent states by taking the specific actions under the policy. Let us consider the specific episode of consecutive states $s_1, s_2, s_3, ...$ following s. Note that the states s_n are functions of the starting state s and the actual policy. The cumulative reward for this specific episode when visiting the consecutive states $s_1, s_2, s_3, ...$ from the starting state s under policy π is thus

$$r_\infty(s) = r(s) + r(s_1) + r(s_2) + r(s_3) + ... \tag{10.3}$$

One problem with this definition is that this value could be unbounded as it runs over infinitely many states into the future. A possible solution of this problem is to restrict the sum by considering only a **finite reward horizon**, for example by only consider rewards given within a certain finite number of steps such as

$$r_4(s) = r(s) + r(s_1) + r(s_2) + r(s_3). \tag{10.4}$$

Another way to solve the infinite payoff problem is to consider reward that is discounted when it is given at later times. Considering a discount factor $0 < \gamma < 1$ for each step, we have a total payoff

$$r_\gamma(s) = r(s) + \gamma r(s_1) + \gamma^2 r(s_2) + \gamma^3 r(s_3) + ... \tag{10.5}$$

Such discounting makes sense when we value immediate reward somewhat more than reward in the future. But large rewards in the future can still have a considerable influence on values.

Since we consider probabilistic state transitions, policies and rewards, we can only estimate the expected value of the total payoff when starting at state s and taking actions according to a policy $\pi(a|s)$. We denote this expected value with the function $E\{R_\gamma(s)\}_\pi$. The expected total discounted payoff from state s when following policy π is thus

$$Q^\pi(s,a) = E\{r(s) + \gamma r(s_1) + \gamma^2 r(s_2) + \gamma^3 r(s_3) + ...\}_\pi. \tag{10.6}$$

This is called the **value-function** for policy pi. Note that this value function not only depends on a specific state but also on the action taken from state s since it is specific for a policy. We will now derive some methods to estimate the value-function for a specific policy before discussing methods of finding the optimal policy.

10.4 The Bellman equation

10.4.1 Bellman equation for a specific policy

With a complete knowledge of the system, that includes a perfect knowledge of the state the agent is in as well as the transition probability and reward function, it is possible to estimate the value function for each policy π from a self-consistent equation. This was already noted by Richard Bellman in the mid 1950s and is known as **dynamic programming**. To derive the Bellman equations we consider the value function, equation 10.6 and separate the expected value of the immediate reward from the expected value of the reward fro visiting subsequent states,

$$Q^\pi(s,a) = E\{r(s)\}_\pi + \gamma E\{r(s_1) + \gamma r(s_2) + \gamma^2 r(s_3) + ...\}_\pi. \tag{10.7}$$

The second expected value on the right hand side is that of the value function for state s_1, but state s_1 is related to state s since state s_1 is the state that can be reached with a certain probability from s when taking action a_1 according to policy π, for example like $s_1 = s + a_1$ and $s_n = s_{n-1} + a_n$. We can incorporate this into the equation by writing

$$Q^\pi(s,a) = r(s) + \gamma \sum_{s'} T(s'|s,a) \sum_{a'} \pi(a'|s') E\{r(s') + \gamma R(s'_1) + \gamma^2 R(s'_2) + ...\}_\pi, \tag{10.8}$$

where s'_1 is the next state after state s', etc. Thus, the expression on the right is the state-value-function of state s'. If we substitute the corresponding expression of equation 10.6 into the above formula, we get the **Bellman equation** for a specific policy, namely

$$Q^\pi(s,a) = r(s) + \gamma \sum_{s'} T(s'|s,a) \sum_{a'} \pi(a'|s') Q^\pi(s',a'). \tag{10.9}$$

In the case of deterministic policies, the action a is given by the policy and the value function $Q^\pi(s,a)$ reduces to $V^\pi(s)$. In this case the equation simplifies to

$$V^\pi(s) = r(s) + \gamma \sum_{s'} T(s'|s,a) V^\pi(s'). \tag{10.10}$$

Such a linear equation system can be solved with our complete knowledge of the environment. In an environment with N states, the Bellman equation is a set of N linear equations, one for each state, with N unknowns which are the expected value for each state. We can thus use well known methods from linear algebra to solve for $V^\pi(s)$. This can be formulated compactly with Matrix notation,

$$\mathbf{r} = (\mathbb{1} - \gamma \mathbf{T})\mathbf{V}^\pi, \tag{10.11}$$

where $\mathbf{r}$ is the reward vector, $\mathbb{1}$ is the unit diagonal matrix, and $\mathbf{T}$ is the transition matrix. To solve this equation we have to invert a matrix and multiply this with the reward values,

$$\mathbf{V}^\pi = (\mathbb{1} - \gamma \mathbf{T})^{-1}\mathbf{r}^t, \tag{10.12}$$

where $\mathbf{r}^t$ is the transpose of $\mathbf{r}$

Note that the analytical solution of the Bellman equation is only possible because we have complete knowledge of the system, including the reward function r, which itself requires a perfect knowledge of the state in which the agent is in. Also, while we used this solution technique from linear algebra, it is much more common to use the Bellman equation directly and calculate a state-value-function iteratively for each policy. We can start with a guess $\mathbf{V}$ for the value of each state, and calculating from this a better estimate

$$\mathbf{V} \leftarrow \mathbf{r} + \gamma \mathbf{T}\mathbf{V} \tag{10.13}$$

until this process converges. We mainly use this iterative approach, although an example of using the analytical example is given below.

10.4.2 Policy iteration

The equations above depends on a specific policy. As mentioned above, in many cases we are mainly interested in finding the policy that gives us the **optimal payoff** and we could simply search for this by considering all possible policies. But this is usually not practical in most but a small number of examples since the number of possible policies is equal to the number of actions to the power of the number of states. This explosion of the problem size with the number of states is one of the main challenges in reinforcement learning and was termed **curse of dimensionality** by Richard Bellman.

A much more efficient method is to incrementally find the value function for a specific policy and then use the policy which maximizes this value function for the next round. The **policy iteration** algorithm is outlined in Figure 10.2. In addition to an initial guess of the value function, we have now also to initialize the policy, which could be randomly chosen from the set of possible actions at each state. For this policy we can then calculate the corresponding value function according to equation 10.9. This step corresponds to an evaluation of the specific policy. The next step is to take this value function and to calculate the corresponding best set of actions for it. Of course, the best actions to take for a specific value function is to take the action from each state that maximize the corresponding future payoff. The corresponding set of actions for each state is then the next candidate policy. These two steps, the policy evaluation and the policy improvement are repeated until the policy does not change any more.

```
Choose initial policy and value function
Repeat until policy is stable {
    1. Policy evaluation
    Repeat until change in values is sufficiently small {
        For each state {
            Calculate the value of neighbouring states when taking  }   $V^\pi$
              action according to current policy.                  }  equation 10.9
            Update estimate of optimal value function.              }
        } each state
    } convergence
    2. Policy improvement
    new policy according to equation 10.21, assuming $V^* \approx$ current $V^\pi$
} policy
```

Fig. 10.2 Policy iteration with asynchronous update.

To demonstrate this scheme for solving MDPs, we will follow a simple example, that of a chain of N states. The states of the chain are labeled consecutively from left to right, $s = 1, 2, ..., N$. An agent has two possible actions, go to the left (lower state numbers; $a = -1$), or go to the right (higher state numbers; $a = +1$). However, in P cases the system responds with the opposite move from the intended. The last state in the chain, state number N, is rewarded with $r(N) = 1$, whereas going to the first state in the chain is punished with $r(1) = -1$. The reward of the intermediate states is set to a small negative value, such as $r(i) = -0.1, 1 < i < N$. We consider a discount factor γ.

The transition probabilities $T(s'|s, a)$ for the chain example are zero expect for the following elements,

$$T(1|1, -1) = 1 \tag{10.14}$$
$$T(N|N, +1) = 1 \tag{10.15}$$
$$T(s - a|s, a) = 1 - P \tag{10.16}$$
$$T(s + a|s, a) = P \tag{10.17}$$

The first two entries specify the ends of the chain as **absorbing boundaries** as the agent would stay in this state one it reaches these states. We can also write this as two **transfer matrices**, one for each possible actions. For $a = 1$ this is,

$$\begin{pmatrix} 1 \cdots & & \cdots 0 \\ \vdots & & \vdots \\ 0 \cdots 0 & 1-P \; 0 \; P & 0 \cdots 0 \\ \vdots & & \vdots \\ 0 \cdots & & \cdots 1 \end{pmatrix} \tag{10.18}$$

and for $a = -1$ this is

$$\begin{pmatrix} 1 \cdots & & \cdots 0 \\ \vdots & & \vdots \\ 0 \cdots 0 & P \; 0 \; 1-P & 0 \cdots 0 \\ \vdots & & \vdots \\ 0 \cdots & & \cdots 1 \end{pmatrix} \tag{10.19}$$

The corresponding Matlab code for setting up the chain example is

```
% Chain example:
% Policy iteration with analytical solution of Bellman equation
clear;
N=10; P=0.8; gamma=0.9; % parameters
U=diag(ones(1,N)); % unit diaogonal matrix
T=zeros(N,N,2); % transfer matrix
r=zeros(1,N)-0.1; r(1)=-1; r(N)=1; % reward function

T(1,1,:)=1; T(N,N,:)=1;
for i=2:N-1;
    T(i,i-1,1)=P;
    T(i,i+1,1)=1-P;
    T(i,i-1,2)=1-P;
    T(i,i+1,2)=P;
end
```

The policy iteration part of the program is then given as follows:

```
% random start policy
policy=floor(2*rand(1,N))+1; %random vector of 1 (going left) and 2 (going right)
Vpi=zeros(N,1); % initial arbitrary value function
iter = 0; % counting iteration
converge=0;
% Loop until convergence
    while ~converge
        % Updating the number of iterations
        iter = iter + 1;
        % Backing up the current V
        old_V = Vpi;
        %Transfer matrix of choosen action
        Tpi=zeros(N); Tpi(1,1)=1; T(N,N)=1;
        for s=2:N-1;
            Tpi(s,s-1)=T(s,s-1,policy(s));
            Tpi(s,s+1)=T(s,s+1,policy(s));
        end
        % Calculate V for this policy
        Vpi=inv(U-gamma*Tpi)*r';
        % Updating policy
        policy(1)=0; policy(N)=0; %absorbing states
        for s=2:N-1
            [tmp,policy(s)] = max([Vpi(s-1),Vpi(s+1)])
        end
        % Check for convergence
        if abs(sum(old_V - Vpi)) < 0.01
            converge = 1;
        end
    end
iter, policy
```

The whole procedure should be run until the policy does not change any more. This stable policy is then the policy we should execute in the agent.

10.4.3 Bellman equation for optimal policy and value iteration

Instead of using the above Bellman equation for an arbitrary value function and then calculating the optimal value function, we can also derive a version of **Bellman's equation for the optimal value function** itself. This second kind of a Bellman equation is given by

$$V^*(s) = r(s) + \max_a \gamma \sum_{s'} T(s'|s,a)V^*(s'). \tag{10.20}$$

The max function is a bit more difficult to implement in the analytic solution, but we can again easily use and iterative method to solve for this optimal value function. This is called **value iteration**. Note that this version includes a max function over all possible actions in contrast to the Bellman equation for a given policy, equation 10.9. As outlined in figure 10.3, we start again with a random guess for the value of each state and then iterate over all possible states using the Bellman equation for the optimal value function, equation 10.20. More specifically, This algorithm takes an initial guess of the optimal value function, typically random or all zeros. We then iterate over the main loop until the change of the value function is sufficiently small. For example, we could calculate the sum of value functions in each iteration (t) and then terminate the procedure if the absolute difference of consecutive iterations is sufficiently small, that is if $|\sum_s V^*_t(s) - \sum_s V^*_{t-1}(s)| <$threshold. In each of those iterations, we iterate over all states and update the estimated optimal value functions according to equation 10.20.

Finally, after convergence of the procedure to get a good approximation of the optimal value function, we can calculate the **optimal policy** by considering all possible actions from each state,

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s'|s,a)V^*(s'), \tag{10.21}$$

which should be used by an agent to achieve good performance.

```
Choose initial estimate of optimal value function
Repeat until change in values is sufficiently small {
        For each state {
            Calculate the maximum expected value of neigh-   \
              bouring states for each possible action.        |        $V^*$
            Use maximal value of this list to update estimate  > equation 10.20
              of optimal value function.                     /
        } each state
} convergence
Calculate optimal value function from equation 10.21
```

Fig. 10.3 Value Iteration with asynchronous update.

The state iteration can be done in various ways. For example, in the **sequential asynchronous updating schema** we update each state in sequence and repeat this procedure over several iterations. Small variations of this schema are concerned with how the algorithm iterates over states. For example, instead of iterating sequentially over the states, we could also use a random oder. We could also first calculate the maximum value of neighbours for all states before updating the value function for all states with an **synchronous updating schema**. Since it can be shown that theses procedure will converge to the optimal solution, all these schemas should work similarly well though might differ slightly for particular examples. Important is, however, that the agent goes repeatedly to every possible state in the system. This can be time consuming, but it works if we have complete knowledge of the system since we do not really perform the actions but can sit and calculate the solution for planing movements. It also works well in the examples with small state spaces but can be problematic for large state space.

The previously discussed policy iteration has some advantages over value iterations. In value iteration we have to try out all possible actions when evaluating the value function, and this can be time consuming when there are many possible actions. In policy iteration, we choose a specific policy, although we have then to iterate over consecutive policies. In practice it turns out that policy iteration often converges fairly rapidly so that it becomes a practical method. However, value iteration is a little bit easier and has more similarities to the algorithms discussed below that are also applicable to situations where we do not know the environment a priori.

Exercise:

Implement the value iteration for the chain problem and plot the learning curve (how the error changes over time), the optimal value function, and the optimal policy. Change parameters such as N, γ, and the number of iterations and discuss the results.

Exercise:

Solve the Russel&Norvig grid with the policy iteration using the basic Bellman functions iteratively, and compare this method to the value iteration.

11 Temporal Difference learning and POMDP

11.1 Temporal Difference learning

Dynamic programming can solve the basic reinforcement learning problem since we assumed a complete knowledge of the system, which includes the knowledge about the precise state of the agent, transition probabilities, the reward functions, etc. In reality, and commonly in robotics, we might not know the rewards given in different states, or the transition probabilities, etc. One approach is to estimate these quantities from interacting with the environment before using dynamic programming. However, we will see that a direct estimation of the quantities is not necessary since our main goal is to estimate the state value function that determines optimal actions. The algorithms in this chapter are all focused of solving the reinforcement problem on-line by interacting with the environment. We will first assume that we still know exactly in which state the agent is at each step, and will then discuss partially observable situations below.

Likely the most direct methods of estimating the value of states is to act in the environment and thereby to sample and memorize reward from which the expected value can be calculated by simple averaging. Such methods are generally called **Monte Carlo methods**. While general Monte Carlo methods are very universal and might work well in some applications, we will concentrate here right away on algorithms which combine ideas from Monte Carlo methods with that of dynamic programming. Such influential methods in reinforcement learning have been developed by Rick Satton and Andrew Barto, and also by Chris Watkins, although some of those methods have even been applied before by Arthur Samuel in the late 1950s to learning to play checkers. Common to these methods is that they use the difference between expected reward and actual reward. Such algorithms are therefore generally called **temporal difference (TD) learning**. We start again by estimating the value function for a specific policy before moving to schemas for the estimating the optimal policy.

Let us recall Bellman's equation for value function of a policy π (eq.10.9),

$$V^{\pi}(s) = r(s) + \gamma \sum_{s''} T(s''|s,a) V^{\pi}(s''). \tag{11.1}$$

The sum on the right-hand side is over all the states that can be reached from state s. A difficulty in practice is often that we don't know the transition probability and have to estimate this somehow. The strategy we are taking now is that we approximate the sum on the right hand side by a specific episode taken by the agent. This interaction of the interaction with the environment that makes this an on-line learning tasks as in Monte Carlo methods. But in contrast to Monte Carlo methods we do not take and memorize the following steps and associated reward but estimate the expected reward of the following step with the current estimate of the value function which is an estimate of the reward of the whole episode. Such strategies are sometime called a **bootstrap** method as if pulling oneself out of the boots by one owns strap. We will label the actual state reached by the agent as s'. Thus, the approximation can be written as

$$\sum_{s''} T(s''|s,a)V^{\pi}(s'') \approx V^{\pi}(s'). \tag{11.2}$$

While this term makes certainly an error, the idea is that this will still result in an improvement of the estimation of the value function, and that other trials have the possibility to evaluate other states that have not been reached in this trial. The value function should then be updated carefully, by considering the new estimate only incrementally,

$$V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha\{r(s) + \gamma V^{\pi}(s') - V^{\pi}(s)\}. \tag{11.3}$$

This is called **TD learning**. The constant α is called a learning rate and should be fairly small. This policy evaluation can then be combined with policy iteration as discussed already in the section on dynamic programming.

11.2 Temporal difference methods for optimal control

Most of the time we are mainly interested in optimal control that maximizes the reward receiver over time. We will now turn to this topic. In this section we will explicitly consider stochastic policies and will thus return to the notation of the state-action value function. Also, since we are always talking about the optimal value function in the following, we will drop the star in the formulas and juts use $Q(s,a) = Q^*(s,a)$ for the optimal value.

A major challenge in on-line learning of optimal control when the agent is interacting with the environment is the trade-off between maximizing the reward in each step and exploring the environment for larger future reward while excepting some smaller immediate reward. This was not a problem in dynamic programming since we would iterate over all states. However, in large state space and in situation where exploring takes time and resources, typical for robotics applications, we can not expect to iterate extensively over all states and we must thrive for a good balance between **exploration** and **exploitation**. Without exploration it can easily happen that the agent get stuck in a suboptimal solution. Indeed, we could only solve the chain problem above because it included some probabilistic transition matrices that helped us exploring the space.

Optimal control demands to maximize reward and therefore to always go to the state with the maximal expected reward at each time. But this could prevent finding even higher payoffs. A essential ingredient of the following algorithms is thus the inclusion of randomness in the policy. For example, we could follow most of the time the **greedy policy**, which chooses another possible actions in a small number ϵ of times. This probabilistic policy is called the **ϵ-greedy policy**,

$$\pi(a = \arg\max_a Q(s,a)) = \epsilon. \tag{11.4}$$

This policy is choosing the policy with the highest expected payoff most of the time while treating all other actions the same. A more graded approach is using the **softmax policy** that choses each action proportional to a Boltzmann distribution

$$\pi(a|s) = \frac{e^{Q(s,a)}}{\sum_{a'} e^{Q(s,a')}}. \tag{11.5}$$

While there are other possible choices of a probabilistic policy, the general idea of the following algorithms do not depend on this details, and we therefore use the ϵ-greedy policy for illustration purposes.

To derive the following on-line algorithms for optimal control, we now consider the Bellman equation for the optimal value function (eq 10.20) generalized for stochastic policies,

$$Q(s,a) = r(s) + \max_a \gamma \sum_{s'} T(s'|s,a) \sum_{a'} \pi(a'|s') Q(s',a'). \tag{11.6}$$

We again use an online procedure in which the agent takes specific actions. Indeed, we now always consider policies that choses actions most of the time that lead to the largest expected payoff. Thus, by taking the action according to the policy we can write a temporal difference learning rule for the optimal stochastic policy as

$$Q(s,a) \leftarrow Q(s,a) + \alpha\{r(s) + \gamma Q(s',a') - Q(s,a)\}, \tag{11.7}$$

where the actions a' is the action chosen according to the policy. This **on-policy TD algorithm** is called **Sarsa** for state-action-reward-state-action. Note that the action a' will not always be the action that maximizes the expected reward since we are using stochastic policies. Thus, slightly different approach is using only the action to the maximal expected reward for the value function update while still exploring the state space through the policy.

$$Q(s,a) \leftarrow Q(s,a) + \alpha\{r(s) + \max_{a'} \gamma Q(s',a') - Q(s,a)\}. \tag{11.8}$$

Such a **off-policy TD algorithm** is called **Q-leaning**

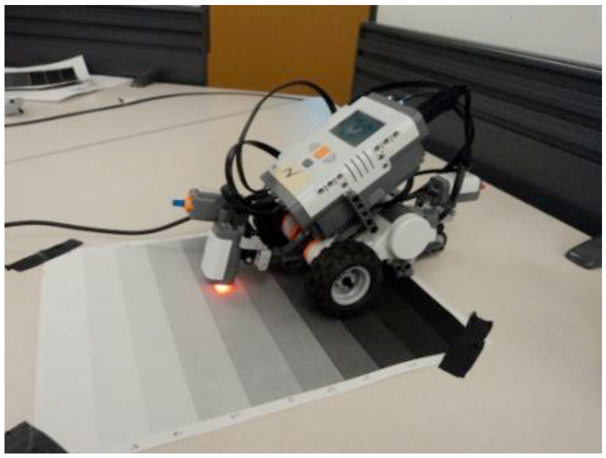
11.3 Robot exercise with reinforcement learning

11.3.1 Chain example

The first example follows closely the chain example discussed in the text. We consider thereby an environment with 8 states. An important requirement for the algorithms is that the robot must know in which state it is in. As discussed in Chapter **??**, this localization problem is a mayor challenge in robotics. We use here the example where we use a state indicator sheet as used in section **??**. You should thereby use the implemented of the calibration from the earlier exercise.

```
Choose initial policy and value function
Repeat until policy is stable {
    **1. Policy evaluation**
    Repeat until change in values is sufficiently small {
    Remembering the value function and reward of current state (eligibility trace)
    If rand>$\epsilon$
        Go to next state according to policy of equation ??
    else
        go to different state
    Update value function of previous state according to (equation 11.3)
    $V^\pi(s-1) \leftarrow V^\pi(s-1) + \alpha(R(s-1) + \gamma V^\pi(s) - V^\pi(s-1))$
        } convergence
    **2. Policy improvement**
    new policy according to equation 10.21, assuming $V^* \approx$ current $V^\pi$
} policy
```

Fig. 11.1 On-policy Temporal Difference (TD) learning



Our aim is for the robot to learn to always travel to state 8 on the state sheet from any initial position. It is easy to write a script with explicit instruction for the robot, but the main point here is that the robot must learn the appropriate action sequence from only given reward feedback. Here you should implement three RL algorithms. The first two are the basic dynamic programming algorithms of value iteration and policy iteration. Note that you can assume at this point that the robot has full knowledge of the environment so that the robot can find the solution by ‘contemplating’ about the problem. However, the robot must be able to execute the final policy.

The third algorithm that you should implement for this specific example is the temporal difference (TD) learning algorithm. This should be a full online implementation in which the robot actively explores the space.

11.3.2 Wall Avoider Robot Using Reinforcement Learning

The goal of this experiment is to teach the NXT robot to avoid walls. Use the Tribot similar with an ultrasonic sensor and a touch sensor mounted at the front. The ultrasonic sensor should be mounted to the third motor so that the robot can look around. An example is shown in Fig. 11.2. Write a program so that the robot learns to avoid bumping

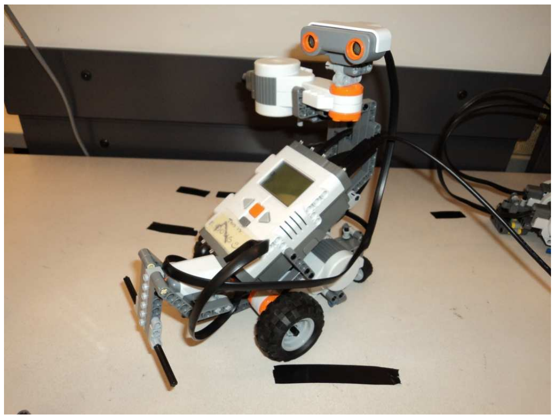


Fig. 11.2 Tribot configuration for the wall avoidance experiment.

by giving negative feedback when it hits an obstacle.

11.4 POMDP

With the introduction of a probability map, the POMDP can be mapped on a MDP

11.5 Model-based RL

11.5.1 TD-Lambda

In all of the above discussions we have assumed a discrete state space such as a chain or a grid. Of course, in practice, we might have a continuous state space, such as the position of a robot arm or a mobile robot in the environment. While discretizing the state space is a common and sometimes sufficient approach, it can also be part of the reason behind the curse of dimensionality since a increasing the resolution of the discretization will increase the number of states exponentially. We are now discussing model-based methods to overcome these problems and to make reinforcement learning applicable to a wider application area.

The idea behind this section is similar to the distinction between the histogram-based and model-based methods for approximating a pdf. The histogram method makes discrete bins and estimates the probability of each bin by averaging over examples. In contrast, a model-based approach makes a hypothesis in form of a parameterized function and estimates the parameters from examples. Thus, the later approach can be applied by making an hypothesis of the functional form of the predicted value at a specific time, $V_t(\mathbf{x}_t)$, from input $\mathbf{x}_t$ at time t,

$$V_t(\mathbf{x}_t) \approx V_t(\mathbf{x}_t; \theta). \tag{11.9}$$

Note that the function on the right is a parameterized approximation of the function on the left. We can use the same symbol as the dependence of the parameter indicates which function is meant. As before, we can use supervised learning for function approximation, and we can use the same methods for learning the parameters from data, such as maximum likelihood estimation. Also, similar to the different approaches in supervised learning, we could build very specific hypothesis for a specific problem or use hypothesis that are very general. While the later approach might suffer from a large number of parameters compared to the first method, we will follow this line here as it is more universally applicable.

A basic method of adjusting the weights is using a gradient-descent methods on an objective function. We will here consider the popular MSE[10], for which the gradient-descent rule is given by

$$\Delta\theta = \alpha \sum_{t=1}^{m} (r - V_t) \frac{\partial V_t}{\partial \theta}. \tag{11.10}$$

We considered here the total change of the weights for a whole episode of m time steps by summing the errors for each time step. One specific difference of this situation to the supervised-learning examples before is that the reward is typically only received after several time steps in the future at the end of an episode. One possible approach for this situation is to keep a history of our predictions and make the changes for the whole episode only after the reward is received at the end of the episode. Another approach is to make incremental (online) updates by following the approach of temporal difference learning and replacing the supervision signal for a particular time step by the prediction of the value of the next time step. Specifically, we can write the difference between the reward and the prediction at time step t as

$$r - V_t = \sum_{k=t}^{m} (V_{k+1} - V_k). \tag{11.11}$$

Using this in equation 11.10 gives

$$\Delta\theta = \alpha \sum_{t}^{m} \sum_{k=t}^{m} (V_{k+1} - V_k) \frac{\partial V_t}{\partial \theta} \tag{11.12}$$

$$= \alpha \sum_{t=1}^{m} (V_{t+1} - V_t) \sum_{k=1}^{t} \frac{\partial V_k}{\partial \theta}, \tag{11.13}$$

Which can be verified by writing out the sums and reordering the terms. Of course, this is just rewriting the original equation 11.10. We still have to keep a memory of all the gradients from the previous time steps, or at least a running sum of these gradients.

[10] As discussed in section **??**, this is appropriate for Gaussian data

While the rules 11.10 and 11.13 are equivalent, we also introduce here some modified rules suggested by Richard Sutton. In particular, we can weight recent gradients more than gradients in the more remote past by introducing a decay factor $0 \leq \lambda \leq 1$. The rule above correspond to $\lambda = 1$ and is thus called the **TD(1) rule**. The more general **TD(λ) rule** is given by

$$\Delta_t \theta = \alpha (V_{t+1} - V_t) \sum_{k=1}^{t} \lambda^{t-k} \frac{\partial V_k}{\partial \theta}. \tag{11.14}$$

It is interesting to look at the extreme of $\lambda = 0$. The **TD(0) rule** is given by

$$\Delta_t \theta = \alpha (V_{t+1} - V_t) \frac{\partial V_t}{\partial \theta}. \tag{11.15}$$

This rule gives different results with respect to the original supervised learning problem described by TD(1), but this rule is local in time and does not require any memory.

The TD(λ) algorithm can be implementing with a multilayer perceptron when back-propagating the error term to hidden layers.

11.5.2 TDgammon

A nice example of the success of TD(λ) was made by Gerald Tesauro from the IBM research labs and published in the *Communications of the ACM* March 1995 / Vol. 38, No. 3 with the title *Temporal Difference Learning and TD-Gammon*. His program learned to play the game at an expert level. The following is an excerpt from this article (see http://www.research.ibm.com/massive/tdl.html):

Programming a computer to play high-level backgammon has been found to be a rather difficult undertaking. In certain simplified endgame situations, it is possible to design a program that plays perfectly via table look-up. However, such an approach is not feasible for the full game, due to the enormous number of possible states (estimated at over 10 to the power of 20). Furthermore, the brute-force methodology of deep searches, which has worked so well in games such as chess, checkers and Othello, is not feasible due to the high branching ratio resulting from the probabilistic dice rolls. At each ply there are 21 dice combinations possible, with an average of about 20 legal moves per dice combination, resulting in a branching ratio of several hundred per ply. This is much larger than in checkers and chess (typical branching ratios quoted for these games are 8-10 for checkers and 30-40 for chess), and too large to reach significant depth even on the fastest available supercomputers.

11.6 Free-energy-based reinforcement learning

How about generalization to stochastic networks? This is discussed by B. Sallans G. Hinton. *Reinforcement Learning with Factored States and Actions*, Journal of Machine Learning Research, Vol 5 (Aug)] pp 1063–1088.