1 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 reward is not necessarily a single value, but it is typically not a specific of precisely what actions should have been taken. The goal of this **reinforcement learning** is for the agent to figure out which actions to take to maximize future payoff (accumulation of rewards). In this chapter we introduce the general idea and basic formulation of such a problem domain, and will then concentrate on the case of a Markov Decision Process (MDP). Such a process is characterized by a transition process that only depends only on the last state of the agent, and we also consider that we know in which state the agent is in. In the next chapters this will be extended to a framework for partially observable situations and temporal difference (TD) learning.

1.1 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.

1.2 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 1.1. 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). In the maze as illustrated in Figure 1.1, 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.

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 1.1). 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

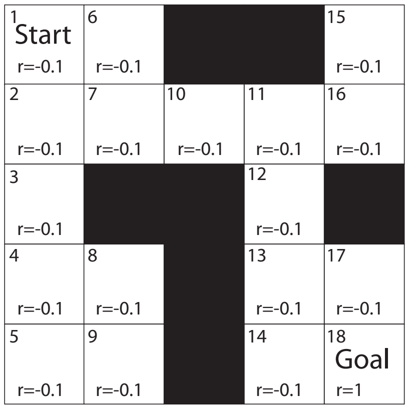

Fig. 1.1 A maze where each state is rewarded with a value r.

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.
- $R(r|s,a)$ 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.

1.3 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(rls,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

$$Q^*(s,a) = \max_\pi Q^\pi(s,a), \tag{1.1}$$

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

$$\pi^*(a|s) = \arg\max_\pi Q^\pi(s,a). \tag{1.2}$$

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

$$r_\infty(s) = r(s) + r(s_1) + r(s_2) + r(s_3) + ... \tag{1.3}$$

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

$$r_4(s) = r(s) + r(s_1) + r(s_2) + r(s_3). \tag{1.4}$$

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

$$r_\gamma(s) = r(s) + \gamma r(s_1) + \gamma^2 r(s_2) + \gamma^3 r(s_3) + ... \tag{1.5}$$

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

$$Q^\pi(s,a) = E\{r(s) + \gamma r(s_1) + \gamma^2 r(s_2) + \gamma^3 r(s_3) + ...\}_\pi. \tag{1.6}$$

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.

1.4 The Bellman equation

1.4.1 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 1.6 and separate the expected value of the immediate reward from the expected value of the reward fro visiting subsequent states,

$$Q^\pi(s,a) = E\{r(s)\}_\pi + \gamma E\{r(s_1) + \gamma r(s_2) + \gamma^2 r(s_3) + ...\}_\pi. \tag{1.7}$$

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

$$Q^\pi(s,a) = r(s) + \gamma \sum_{s'} T(s'|s,a) \sum_{a'} \pi(a'|s') E\{r(s') + \gamma R(s'_1) + \gamma^2 R(s'_2) + ...\}_\pi, \tag{1.8}$$

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 1.6 into the above formula, we get the **Bellman equation** for a specific policy, namely

$$Q^\pi(s,a) = r(s) + \gamma \sum_{s'} T(s'|s,a) \sum_{a'} \pi(a'|s') Q^\pi(s',a'). \tag{1.9}$$

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

$$V^{\pi}(s) = r(s) + \gamma \sum_{s'} T(s'|s,a) V^{\pi}(s'). \tag{1.10}$$

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,

$$\mathbf{r} = (\mathbb{1} - \gamma \mathbf{T})\mathbf{V}^{\pi}, \tag{1.11}$$

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,

$$\mathbf{V}^{\pi} = (\mathbb{1} - \gamma \mathbf{T})^{-1}\mathbf{r}^{t}, \tag{1.12}$$

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

$$\mathbf{V} \leftarrow \mathbf{r} + \gamma \mathbf{T}\mathbf{V} \tag{1.13}$$

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

1.4.2 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 1.2. 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 1.9. 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  }
              action according to current policy.                   }   $V^\pi$
            Update estimate of optimal value function.              }   equation 1.9
        } each state
    } convergence
    **2. Policy improvement**
    new policy according to equation 1.21, assuming $V^* \approx$ current $V^\pi$
} policy
```

Fig. 1.2 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,

$$T(1|1, -1) = 1 \tag{1.14}$$
$$T(N|N, +1) = 1 \tag{1.15}$$
$$T(s - a|s, a) = 1 - P \tag{1.16}$$
$$T(s + a|s, a) = P \tag{1.17}$$

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,

$$\begin{pmatrix} 1 \cdots & & \cdots 0 \\ \vdots & & \vdots \\ 0 \cdots 0 & 1-P \; 0 \; P & 0 \cdots 0 \\ \vdots & & \vdots \\ 0 \cdots & & \cdots 1 \end{pmatrix} \tag{1.18}$$

and for $a = -1$ this is

$$\begin{pmatrix} 1 \cdots & & \cdots 0 \\ \vdots & & \vdots \\ 0 \cdots 0 & P \; 0 \; 1-P & 0 \cdots 0 \\ \vdots & & \vdots \\ 0 \cdots & & \cdots 1 \end{pmatrix} \tag{1.19}$$

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 diagonal 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.

1.4.3 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

$$V^*(s) = r(s) + \max_a \gamma \sum_{s'} T(s'|s,a)V^*(s'). \tag{1.20}$$

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 1.9. As outlined in figure 1.3, 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 1.20. 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 1.20.

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,

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s'|s,a)V^*(s'), \tag{1.21}$$

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 1.20
              of optimal value function.
        } each state
} convergence
Calculate optimal value function from equation 1.21
```

Fig. 1.3 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.