Markov Decision Process

Markov Decision Processes

A Markov decision process is a tuple $(S, A, \{P_{sa}\}, \gamma, R)$ where:

- S is a set of **states**. (For example, in autonomous helicopter flight, S might be the set of all possible positions and orientations of the helicopter.)
- A is a set of **actions**. (For example, the set of all possible directions in which you can push the helicopter's control sticks.)
- P_{sa} are the state transition probabilities. For each state $s \in S$ and action $a \in A$, P_{sa} is a distribution over the state space. We'll say more about this later, but briely, P_{sa} gives the distribution over what states we will transition to if we take action a in state s.
- $\gamma \in [0, 1)$ is called the **discount factor**.
- $R : S \times A \mapsto \mathbb{R}$ is the **reward function**. (Rewards are sometimes also written as a function of a state S only, in which case we would have $R : S \mapsto \mathbb{R}$).

The dynamics of an MDP

- We start in some state s_0, and get to choose some action $a_0 \in A$
- As a result of our choice, the state of the MDP randomly transitions to some successor state s_1, drawn according to $s_1 \sim P_{s0a0}$
- Then, we get to pick another action a_1
- …

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \dots$$

The dynamics of an MDP, (Cont'd)

- Upon visiting the sequence of states s_0, s_1, …, with actions a_0, a_1, …, our total payoff is given by

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \ldots$$

- Or, when we are writing rewards as a function of the states only, this becomes

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots$$

  – For most of our development, we will use the simpler state-rewards R(s), though the generalization to state-action rewards R(s; a) offers no special diffculties.

- Our goal in reinforcement learning is to choose actions over time so as to maximize the expected value of the total payoff:

$$\mathrm{E}\left[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots\right]$$

Policy

- A policy is any function $\pi : S \mapsto A$ mapping from the states to the actions.

- We say that we are executing some policy if, whenever we are in state s, we take action $a = \pi(s)$.

- We also define the value function for a policy π according to

$$V^{\pi}(s) = \mathrm{E}\big[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots \mid s_0 = s, \pi\big]$$

  – $V^{\pi}(s)$ is simply the expected sum of discounted rewards upon starting in state s, and taking actions according to π.

Value Function

- Given a fixed policy π, its value function V^{π} satisfies the **Bellman equations**:

$$V^{\pi}(s) = \underbrace{R(s)}_{\text{Immediate reward}} + \underbrace{\gamma \sum_{s' \in S} P_{s\pi(s)}(s') V^{\pi}(s')}_{\text{expected sum of future discounted rewards}}$$

  – Bellman's equations can be used to efficiently solve for V^{π} (see later)

The Grid world

M = 0.8 in direction you want to go
0.2 in perpendicular
- 0.1 left
- 0.1 right

Policy: mapping from states to actions

An optimal policy for the stochastic environment:

3	→	→	→	+1
2	↑		↑	-1
1	↑	←	←	←
	1	2	3	4

utilities of states:

3	0.812	0.868	0.912	+1
2	0.762		0.660	-1
1	0.705	0.655	0.611	0.388
	1	2	3	4

Environment
- Observable (accessible): percept identifies the state
- Partially observable

Markov property: Transition probabilities depend on state only, not on the path to the state.

Markov decision problem (MDP).

Partially observable MDP (POMDP): percepts does not have enough info to identify transition probabilities.

Optimal value function

- We define the optimal value function according to

$$V^*(s) = \max_{\pi} V^{\pi}(s) \qquad (1)$$

  - In other words, this is the best possible expected sum of discounted rewards that can be attained using any policy

- There is a version of Bellman's equations for the optimal value function:

$$V^*(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s') \qquad (2)$$

  - Why?

Optimal policy

- We also define the optimal policy: $\pi^* \ : \ S \mapsto A$ as follows:

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in S} P_{sa}(s') V^*(s') \qquad (3)$$

- Fact:
  - $$V^*(s) = V^{\pi^*}(s) \geq V^{\pi}(s)$$
  - Policy π^* has the interesting property that it is the optimal policy for all states s.
    - It is not the case that if we were starting in some state s then there'd be some optimal policy for that state, and if we were starting in some other state s_0 then there'd be some other policy that's optimal policy for s_0.
    - The same policy π^* attains the maximum above for all states s. This means that we can use the same policy no matter what the initial state of our MDP is.

The Basic Setting for Learning

- Training data: n finite horizon trajectories, of the form

$$\{s_0, a_0, r_0, \ldots, s_T, a_T, r_T, s_{T+1}\}.$$

- Deterministic or stochastic policy: A sequence of decision rules

$$\{\pi_0, \pi_1, \ldots, \pi_T\}.$$

- Each π maps from the observable history (states and actions) to the action space at that time point.

Algorithm 1: Value iteration

- Consider only MDPs with finite state and action spaces
  $(|S| < \infty,\ |A| < \infty)$

- The value iteration algorithm:

  1. For each state s, initialize $V(s) := 0$.
  2. Repeat until convergence {
     - For every state, update
       $V(s) := R(s) + \max_{a\in A} \gamma \sum_{s'\in S} P_{sa}(s')V^*(s')$.

     }

- It can be shown that value iteration will cause V to converge to V^*. Having found V^*, we can find the optimal policy as follows:

$$\pi^*(s) = \arg\max_{a\in A} \sum_{s'\in S} P_{sa}(s')V^*(s')$$

Algorithm 2: Policy iteration

- The policy iteration algorithm:

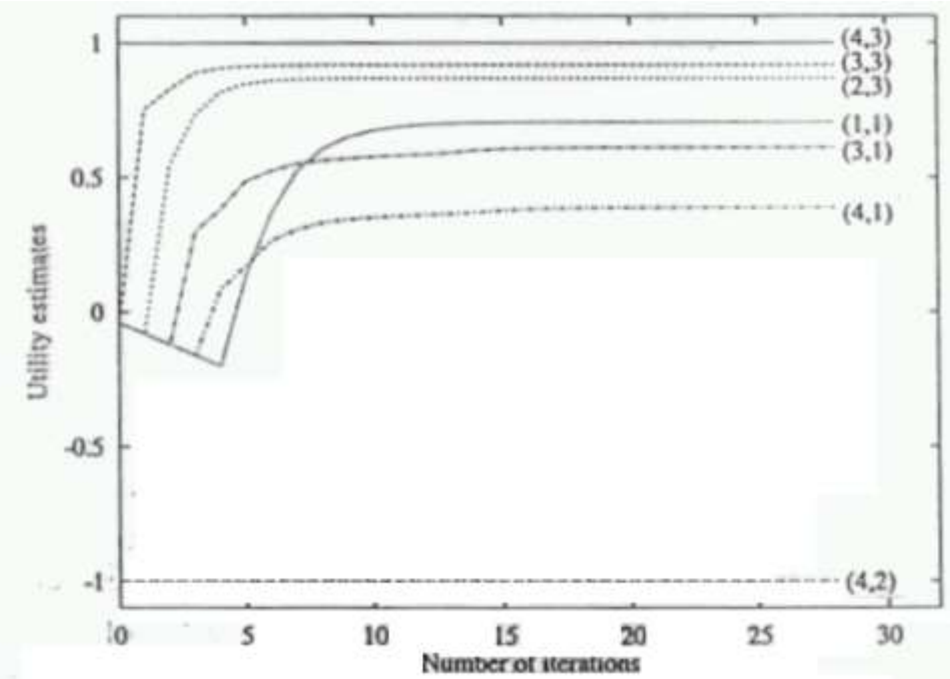
  1. Initialize π randomly.
  2. Repeat until convergence $\{$
     - Let $V := V^{\pi}$
     - For each state s, let $\pi(s) := \max_{a \in A} \sum_{s' \in S} P_{sa}(s') V^*(s')$.

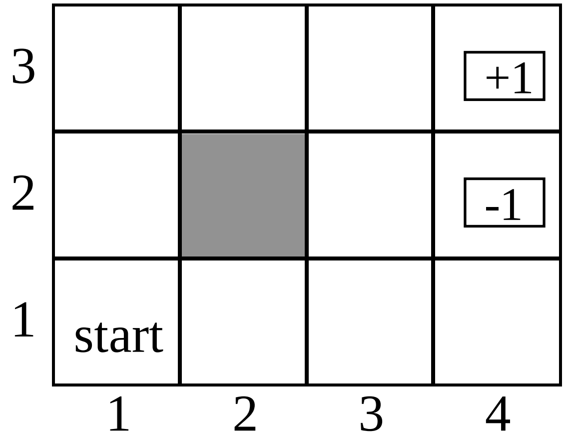
     $\}$

  – The inner-loop repeatedly computes the value function for the current policy, and then updates the policy using the current value function.
  – Greedy update
  – After at most a finite number of iterations of this algorithm, V will converge to V^*, and π will converge to π^*.

Convergence

- The utility values for selected states at each iteration step in the application of VALUE-ITERATION to the 4x3 world in our example





Thrm: As t→∞, value iteration converges to exact U even if updates are done asynchronously & *i* is picked randomly at every step.

Convergence

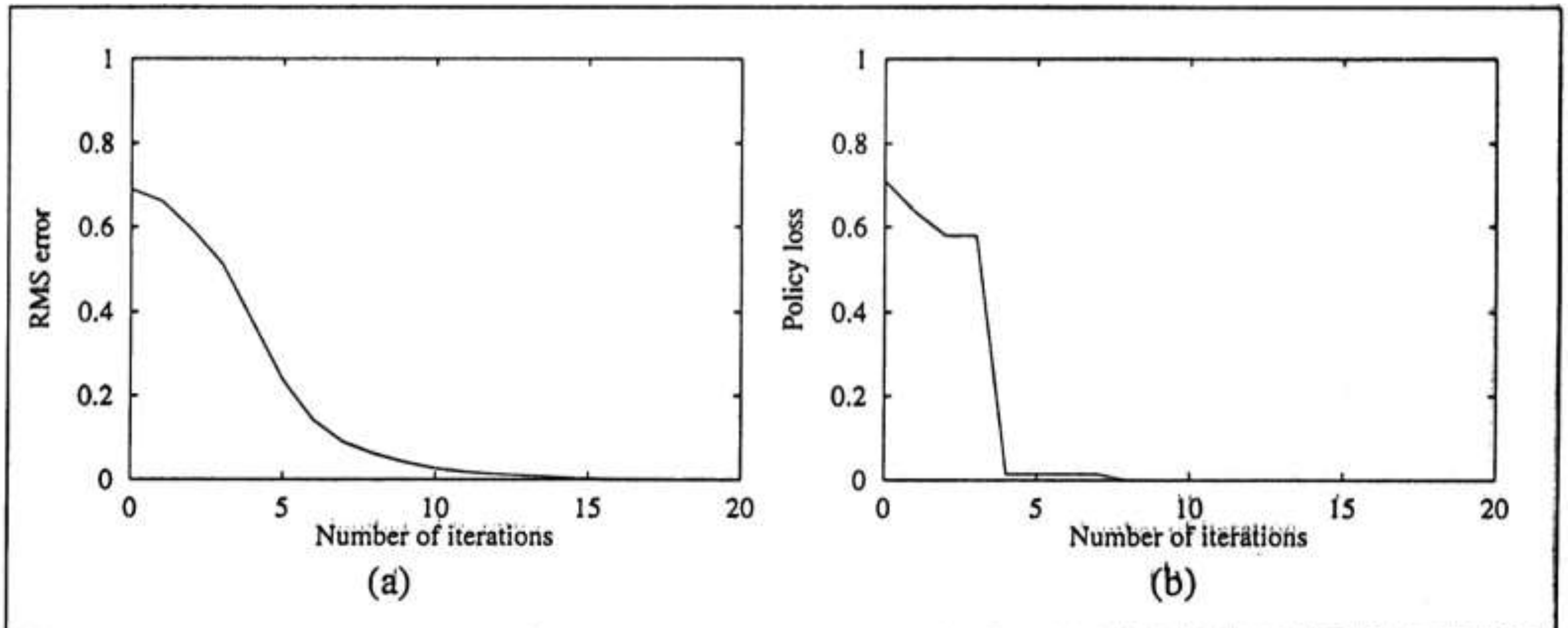


Figure 17.6 (a) The RMS (root mean square) error of the utility estimates compared to the correct values, as a function of iteration number during value iteration. (b) The expected policy loss compared to the optimal policy.

When to stop value iteration?

Q learning

- Define Q-value function

$$V(s) = \max_a Q(s, a)$$

- Rule to choose the action to take

$$a = \arg\max_a Q(s, a)$$

Algorithm 3: Q learning

For each pair (*s, a*), initialize *Q(s,a)*

Observe the current state s

Loop forever

{

Select an action ***a*** (optionally with ε---exploration) and execute it

$$a = \arg\max_a Q(s,a)$$

Receive immediate reward ***r*** and observe the new state ***s'***

Update *Q(s,a)*

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r_{t+1} + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

s=s'

}

Exploration

- Tradeoff between exploitation (control) and exploration (identification)

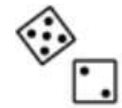
- Extremes: greedy vs. random acting

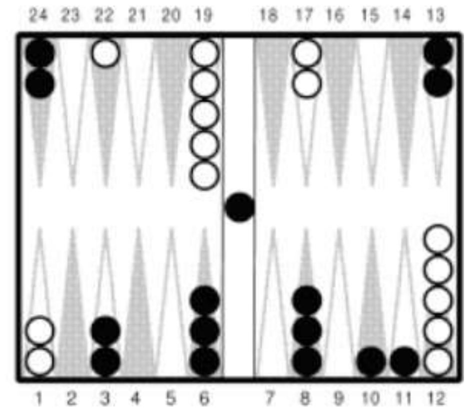
  (n-armed bandit models)

Q-learning converges to optimal Q-values if

– Every state is visited infinitely often (due to exploration),

A Success Story





- TD Gammon (Tesauro, G., 1992)

  --- A Backgammon playing program.

  --- Application of temporal difference learning.

  --- The basic learner is a neural network.

  --- It trained itself to the world class level by playing against itself and learning from the outcome. So smart!!