Reinforcement Learning (RL)

CE-717: Machine Learning

Sharif University of Technology

M. Soleymani

Fall 2016

Reinforcement Learning (RL)

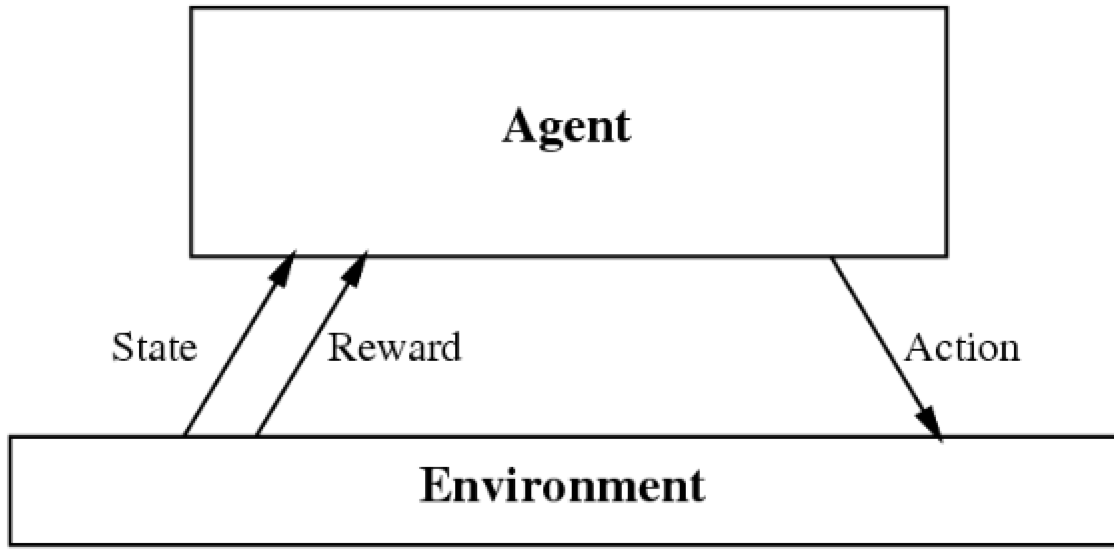
- Learning as a result of interaction with an environment
  - to improve the agent's ability to behave optimally in the future to achieve the goal.
- The first idea when we think about the nature of learning
- Examples:
  - Baby movements
  - Learning to drive car
    - Environment's response affects our subsequent actions
    - We find out the effects of our actions later

Paradigms of learning

- Supervised learning
  - Training data: features and labels for N samples $\left\{\left(\boldsymbol{x}^{(n)}, y^{(n)}\right)\right\}_{n=1}^{N}$
- Unsupervised learning
  - Training data: only features for N samples $\left\{\boldsymbol{x}^{(n)}\right\}_{n=1}^{N}$
- Reinforcement learning
  - Training data: a sequence of (s, a, r)
    - (state, action, reward)
  - Agent acts on its environment, it receives some evaluation of its action via reinforcement signal
    - it is not told of which action is the correct one to achieve its goal

Reinforcement Learning (RL)

- S: Set of states
- A: Set of actions



- Goal: Learning an optimal policy (mapping from states to actions) in order to maximize its long-term reward
  - The agent's objective is to maximize amount of reward it receives over time.

Environment properties

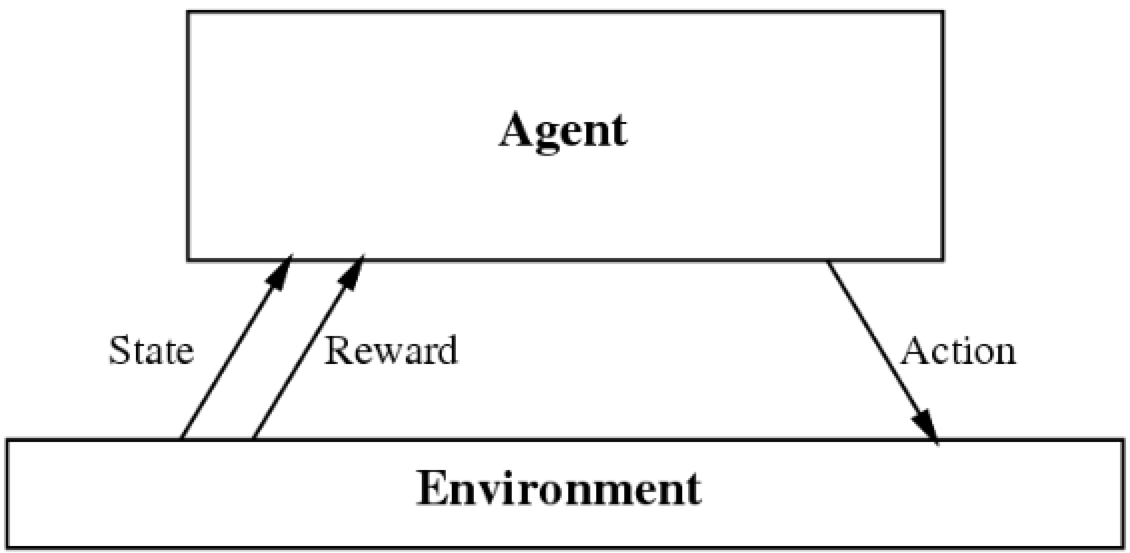
- Deterministic vs. stochastic
  - Stochastic: stochastic reward & transition

- Known vs. unknown
  - Unknown: Agent doesn't know the precise results of its actions before doing them

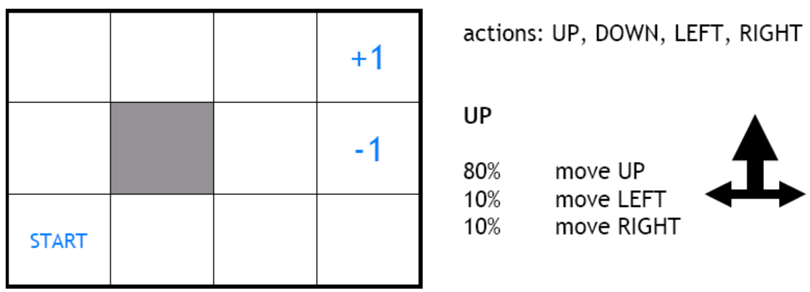
- Fully observable vs. partially observable
  - Observable (accessible): percept identifies the state
  - Partially observable: Agent doesn't necessarily know all about the current state
    - [We discuss about only fully observable environments.]

Reinforcement Learning: Example

- Chess game (deterministic game)
  - Learning task: chose move at arbitrary board states
  - Training signal: final win or loss
  - Training: e.g., played n games against itself



Non-deterministic world: Example



Other action
reward: -0.04

[Russel, AIMA, 2010]

- What is the policy to achieve max reward?

Main characteristics and applications of RL

- Main characteristics of RL
  - Learning is a multistage decision making process
    - Actions influence later perceptions (inputs)
    - Delayed reward: actions may affect not only the immediate reward but also subsequent rewards
  - agent must learn from interactions with environment
    - Agent must be able to learn from its own experience
    - Not entirely supervised, but interactive
      - by trial-and-error
    - Opportunity for active exploration
      - Needs trade-off between exploration and exploitation

Popular applications

- Robotics and control
- Game playing

Main elements of RL

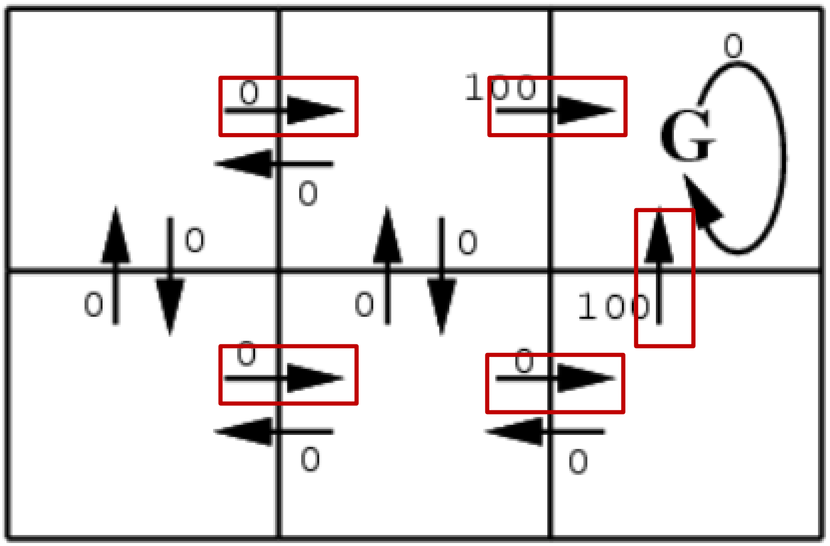
- A policy
  - A map from state space to action space.
  - May be stochastic.

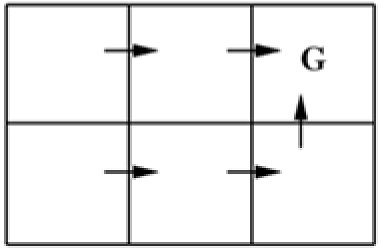
- A reward function
  - It maps each state (or, state-action pair) to a real number, called reward.

- A value function
  - Value of a state (or state-action) is the total expected reward, starting from that state (or state-action).

RL deterministic world: Example

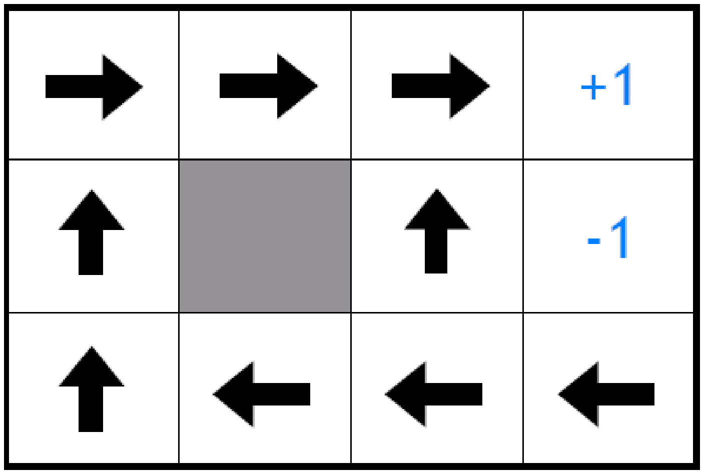
- Example: Robot grid world
  - Deterministic and known reward and transitions





One optimal policy

Optimal policy

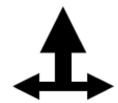


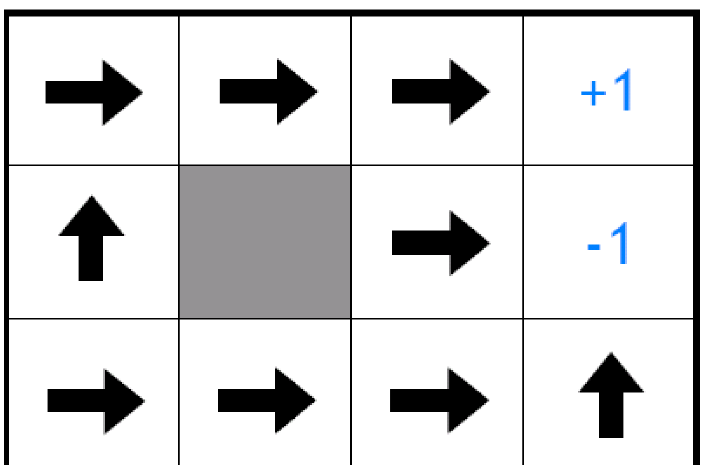
$r = -0.04$ for other actions

actions: UP, DOWN, LEFT, RIGHT

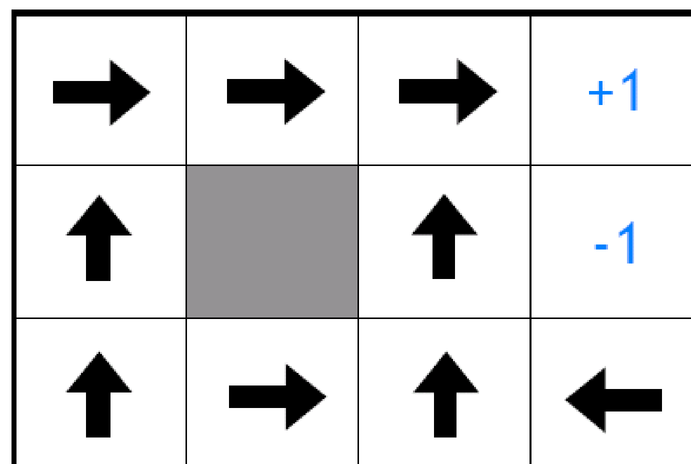
UP

80% move UP
10% move LEFT
10% move RIGHT



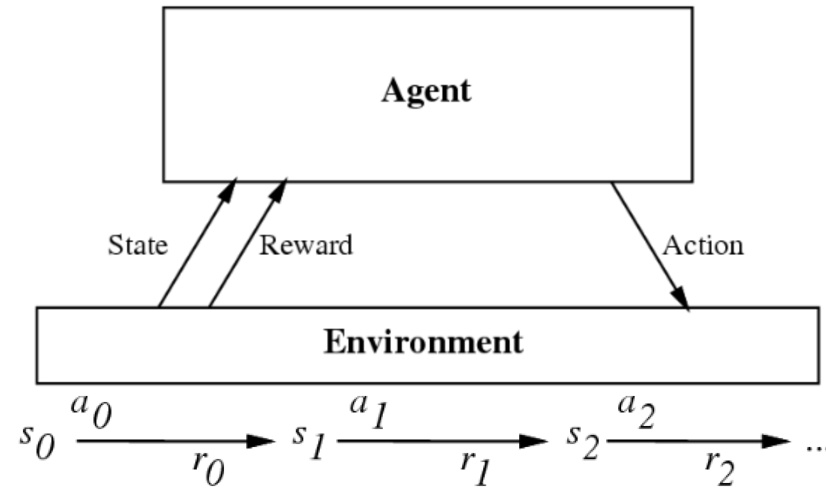


$r = -4$ for other actions



$r = -0.4$ for other actions

RL problem: deterministic environment



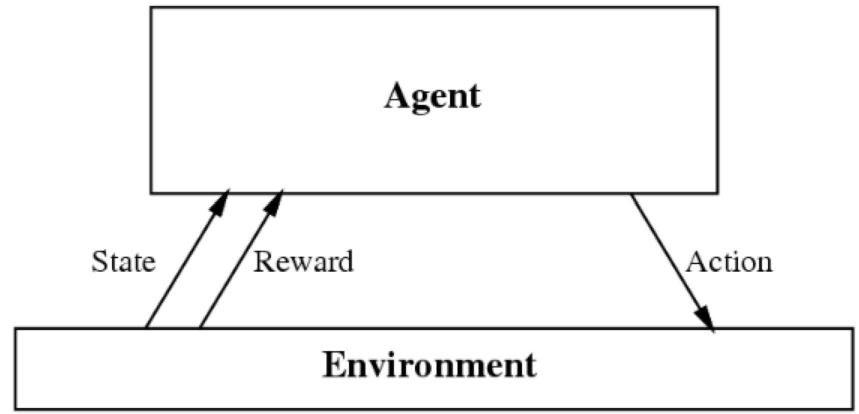
- Deterministic
  - Transition and reward functions
- At time t:
  - Agent observes state $s_t \in S$
  - Then chooses action $a_t \in A$
  - Then receives reward r_t, and state changes to s_{t+1}
- Learn to choose action a_t in state s_t that maximizes the discounted return:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k}\,, \qquad 0 < \gamma \leq 1$$

Upon visiting the sequence of states st, st+1, … with actions at, at+1, … it shows the total payoff

RL problem: stochastic environment

- Stochastic environment
  - Stochastic transition and/or reward



- Learn to choose a policy that maximizes the expected discounted **return**:

$$E[R_t] = E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots]$$

starting from state s_t

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

Markov Decision Process (RL Setting)

- We encounter a multistage decision making process.

- Markov assumption:

$$P(s_{t+1}, r_t | s_t, a_t, r_{t-1}, s_{t-1}, a_{t-1}, , r_{t-2}, \dots) = P(s_{t+1}, r_t | s_t, a_t)$$

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

- Goal: for every possible state $s \in S$ learn a policy π for choosing actions that maximizes

$$E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots | s_t = s, \pi], \qquad 0 < \gamma \leq 1$$

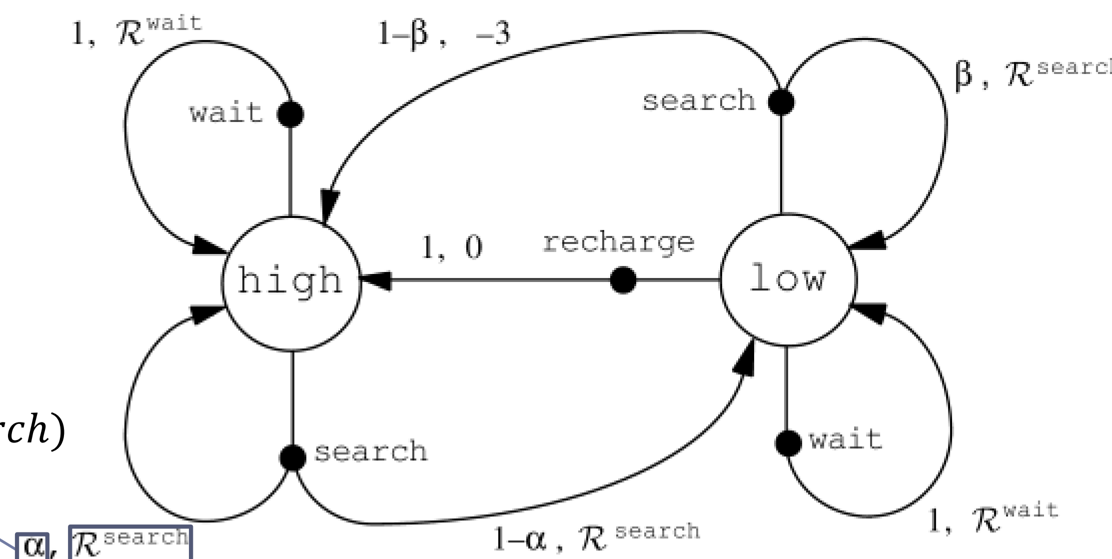
MDP: Definition

- A Markov decision process is composed:
  - $\mathcal{S}$: a finite set of states
  - $\mathcal{A}$: a finite set of actions
  - Transition probabilities
    - $\mathcal{P}_{ss'}^{a} = P(s_{t+1} = s' | s_t = s, a_t = a)$ as the probability that action a in state s at time t will lead to state s' at time $t+1$
  - Immediate rewards:
    - $\mathcal{R}_{ss'}^{a} = E\{r_t | s_t = s, a_t = a, s_{t+1} = s'\}$ as the immediate reward received after transition to state s' from state s with action a
  - $\gamma \in [0,1]$: discount factor
    - represents the difference in importance between future rewards and present rewards.

MDP: Recycling Robot example

- $S = \{high, low\}$
- $A = \{search, wait, recharge\}$
  - $\mathcal{A}(high) = \{search, wait\}$ → Available actions in the 'high' state
  - $\mathcal{A}(low) = \{search, wait, recharge\}$
- $\mathcal{R}_{search} > \mathcal{R}_{wait}$

$P(s_{t+1} = high | s_t = high, a_t = search)$



reward

RL: Autonomous Agent

- Execute actions in environment, observe results, and learn
  - Learn (perhaps stochastic) policy that maximizes $E[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \,|s_t = s, \pi]$ for every state $s \in S$

- Function to be learned is the policy $\pi: S \times A \rightarrow [0,1]$ (when the policy is deterministic $\pi: S \rightarrow A$)
  - Training examples in supervised learning: $\langle s, a \rangle$ pairs
  - RL training data shows the amount of reward for a pair $\langle s, a \rangle$.
    - training data are of the form $\langle \langle s, a \rangle, r \rangle$

State-value function for policy π

- Given a policy π, define value function

$$V^{\pi}(s) = E\{\sum_{k=0}^{\infty} \gamma^k r_{t+k} \,|\, s_t = s, \pi\}$$

- $V^{\pi}(s)$: How good for the agent to be in the state s when its policy is π
  - It is simply the expected sum of discounted rewards upon starting in state s and taking actions according to π

Approaches to solve RL problems

- Three fundamental classes of methods for solving the RL problems:
  - Dynamic programming
  - Monte Carlo methods
  - Temporal-difference learning

- Main approaches
  - Value iteration and Policy iteration are two more classic approaches to this problem.
    - They are dynamic programming approaches
  - Q-learning is a more recent approaches to this problem.
    - It is a temporal-difference method.

Recursive definition for $V^{\pi}(S)$

$$V^{\pi}(s) = E\{\sum_{k=0}^{\infty} \gamma^k r_{t+k} \,|s_t = s, \pi\}$$

$$= E\left\{r_t + \gamma \sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k} \,|s_t = s, \pi\right\}$$

$$= E\left\{r_t + \gamma \sum_{k=0}^{\infty} \gamma^{k} r_{t+k+1} \,|s_t = s, \pi\right\}$$

$$= \sum_{a} \pi(s,a) \sum_{s'} \mathcal{P}_{ss'}^{a} \left(\mathcal{R}_{ss'}^{a} + \underbrace{\gamma E\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,|s_{t+1} = s', \pi\}}_{V^{\pi}(s')}\right)$$

$$\mathcal{P}_{ss'}^{a} = P(s_{t+1} = s' | s_t = s, a_t = a)$$

$$\mathcal{R}_{ss'}^{a} = E\{r_t | s_t = s, a_t = a, s_{t+1} = s'\}$$

Bellman Equations

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} \mathcal{P}_{ss'}^{a} \left(\mathcal{R}_{ss'}^{a} + \gamma V^{\pi}(s')\right)$$

Base equation for dynamic programming approaches

State-action value function for policy π

$$Q^{\pi}(s,a) = E\left\{\sum_{k=0}^{\infty} \gamma^{k} r_{t+k} \,\middle|\, s_t = s, a_t = a, \pi\right\}$$

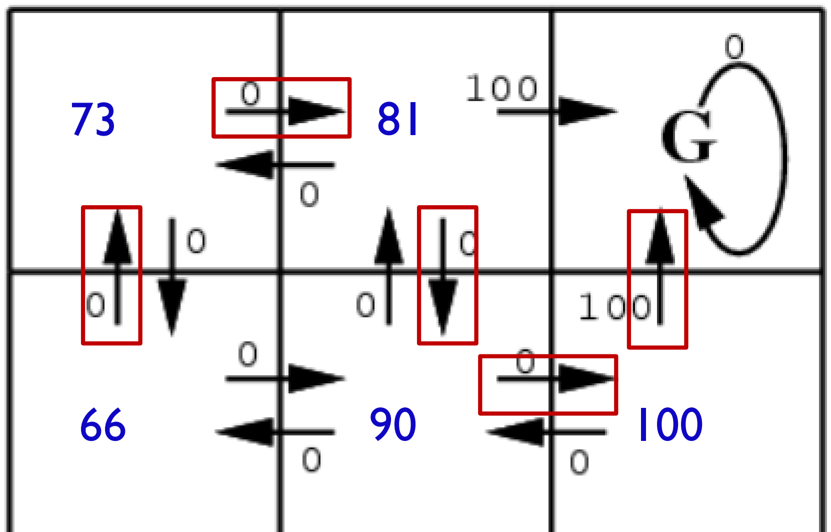
$$= \sum_{s'} \mathcal{P}_{ss'}^{a} \left(\mathcal{R}_{ss'}^{a} + \gamma \underbrace{E\{\textstyle\sum_{k=0}^{\infty} \gamma^{k} r_{t+k+1} \,|s_{t+1} = s', \pi\}}_{V^{\pi}(s')}\right)$$

$$\boxed{Q^{\pi}(s,a) = \sum_{s'} \mathcal{P}_{ss'}^{a} \left(\mathcal{R}_{ss'}^{a} + \gamma V^{\pi}(s')\right)}$$

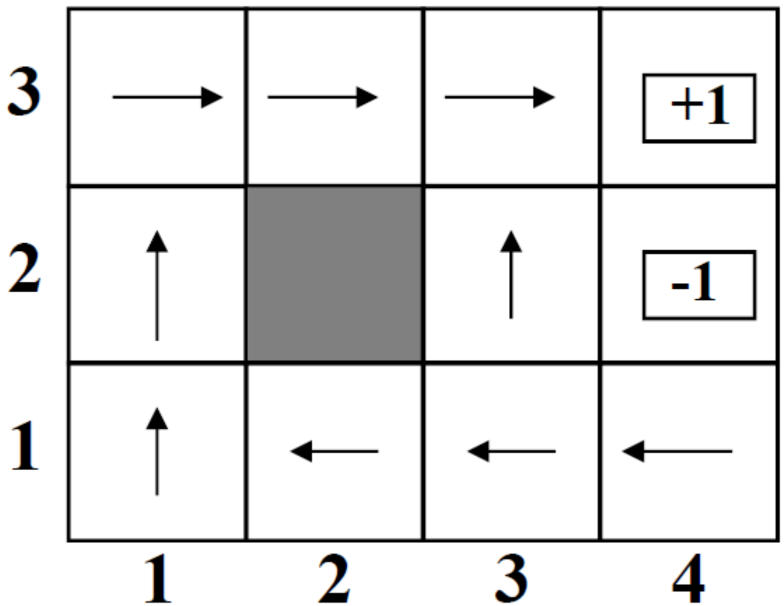
State-value function for policy π: example

- Deterministic example

$$V^{\pi}(s) = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \qquad s_t = s$$



Grid-world: value function example



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

Optimal policy

- Policy π is better than (or equal to) π' (i.e. $\pi \geq \pi'$) iff
$$V^{\pi}(s) \geq V^{\pi'}(s), \qquad \forall s \in S$$

- Optimal policy π^* is better than (or equal to) all other policies ($\forall \pi, \pi^* \geq \pi$)

- **Optimal policy π^*:**
$$\pi^*(s) = \underset{\pi}{\operatorname{argmax}}\, V^{\pi}(s), \qquad \forall s \in S$$

MDP: Optimal policy state-value and action-value functions

- Optimal policies share the same optimal state-value function ($V^{\pi^*}(s)$ will be abbreviated as $V^*(s)$):

$$V^*(s) = \max_\pi V^\pi(s)\,, \qquad \forall s \in S$$

- And the same optimal action-value function:

$$Q^*(s,a) = \max_\pi Q^\pi(s,a)\,, \qquad \forall s \in S, a \in \mathcal{A}(s)$$

- For any MDP, a deterministic optimal policy exists!

Optimal policy

- If we have $V^*(s)$ and $P(s_{t+1}|s_t, a_t)$ we can compute $\pi^*(s)$

$$\pi^*(s) = \underset{a}{\operatorname{argmax}} \left\{ \sum_{s'} \mathcal{P}_{ss'}^{a} \left(\mathcal{R}_{ss'}^{a} + \gamma V^*(s') \right) \right\}$$

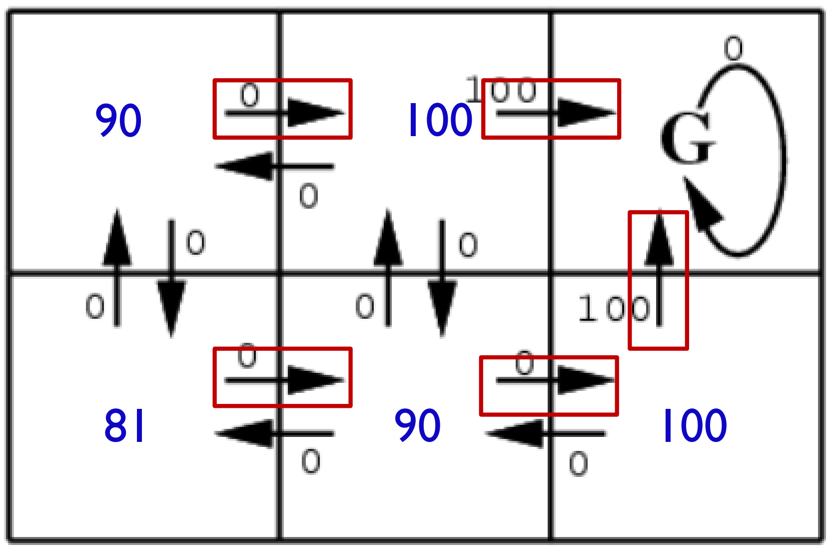
- It can also be computed as:

$$\pi^*(s) = \underset{a \in \mathcal{A}(s)}{\operatorname{argmax}} \, Q^*(s, a)$$

- Optimal policy has the interesting property that it is the optimal policy for all states.
  - Share the same optimal state-value function
  - It is not dependent on the initial state.
    - use the same policy no matter what the initial state of MDP is

State-value function for policy π^*: example

- Deterministic example



Bellman optimality equation

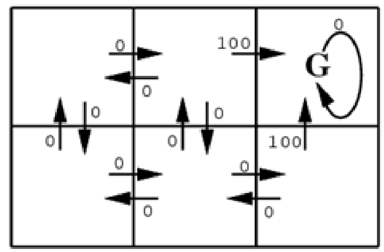
$$V^*(s) = \max_{a \in \mathcal{A}(s)} \sum_{s'} \mathcal{P}_{ss'}^{a} \left(\mathcal{R}_{ss'}^{a} + \gamma V^*(s') \right)$$

$$Q^*(s,a) = \sum_{s'} \mathcal{P}_{ss'}^{a} \left(\mathcal{R}_{ss'}^{a} + \gamma \max_{a'} Q^*(s',a') \right)$$

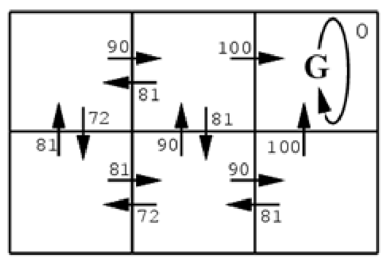
$$V^*(s) = \max_{a \in \mathcal{A}(s)} Q^*(s,a)$$

$$Q^*(s,a) = \sum_{s'} \mathcal{P}_{ss'}^{a} \left(\mathcal{R}_{ss'}^{a} + \gamma V^*(s') \right)$$

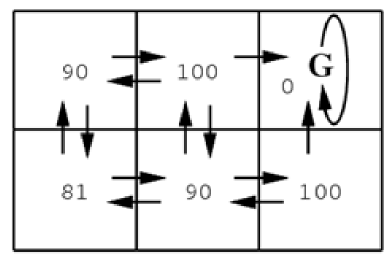
Optimal policy: example 1 (deterministic env.)



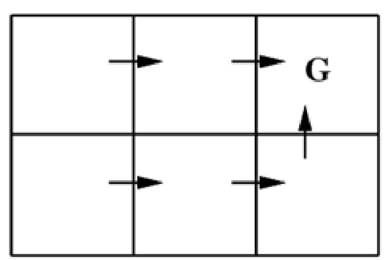
$r(s,a)$ (immediate reward) values



$Q(s,a)$ values



$V^*(s)$ values



One optimal policy

RL algorithms

- Model-based (passive)
  - Known environment model (transition and reward probabilities)
    - Value iteration and policy iteration algorithms

- Model-free (active)
  - Unknown environment model

First, we introduce the model-based algorithms

Value Iteration algorithm

Consider only MDPs with finite state and action spaces:

1) Initialize $V(s)$ arbitrarily
2) Repeat until convergence

   for $s \in S$

$$V(s) \leftarrow \max_a \sum_{s'} \mathcal{P}_{ss'}^a \left(\mathcal{R}_{ss'}^a + \gamma V(s')\right)$$

$V(s)$ converges to $V^*(s)$

Asynchronous: Instead of updating values for all states at once in each iteration, it can update them state by state, or more often to some states than others.

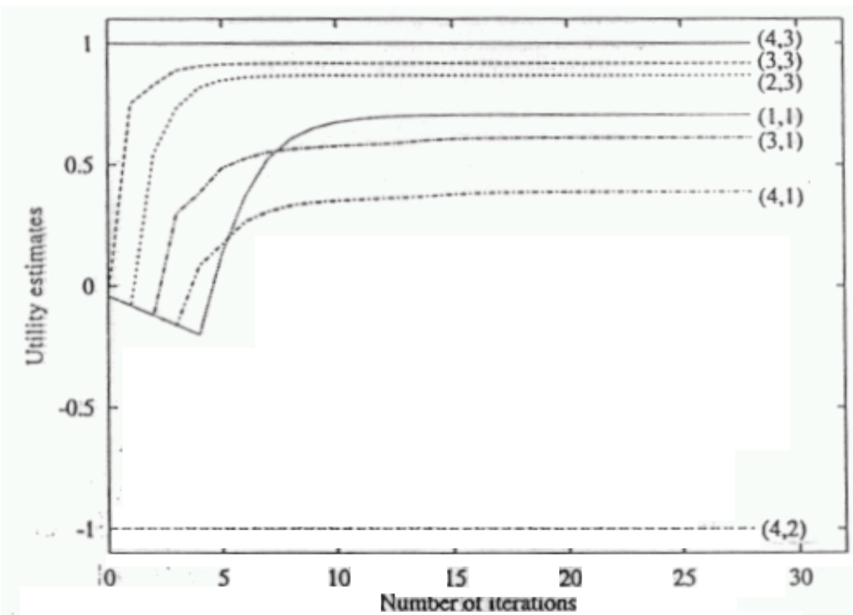
Value Iteration

- Value iteration works even if we randomly traverse the environment instead of looping through each state and action (update asynchronously)
  - but we must still visit each state infinitely often

- If $\max_{s \in S} \left| V^{old}(s) - V(s) \right| < \epsilon$, then the value of the greedy policy differs from the optimal policy by no more than $\frac{2\epsilon\gamma}{1-\gamma}$

- Value Iteration
  - It is time and memory expensive

Convergence



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

[Russel, AIMA, 2010]

Main steps in solving Bellman optimality equations

- Two kinds of steps, which are repeated in some order for all the states until no further changes take place

$$\pi(s) = \underset{a}{\operatorname{argmax}} \left\{ \sum_{s'} \mathcal{P}_{ss'}^{a} \left(\mathcal{R}_{ss'}^{a} + \gamma V^{\pi}(s') \right) \right\}$$

$$V^{\pi}(s) = \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)} \left(\mathcal{R}_{ss'}^{\pi(s)} + \gamma V^{\pi}(s') \right)$$

Policy Iteration algorithm

1) Initialize $\pi(s)$ arbitrarily
2) Repeat until convergence

   Compute the value function for the current policy π (V^{π})

   $V \leftarrow V^{\pi}$

   for $s \in S$

   $$\pi(s) \leftarrow \underset{a}{\operatorname{argmax}} \sum_{s'} \mathcal{P}_{ss'}^{a}\left(\mathcal{R}_{ss'}^{a} + \gamma V(s')\right)$$

updates the policy (greedily) using the current value function.

$\pi(s)$ converges to $\pi^*(s)$

When to stop iterations:

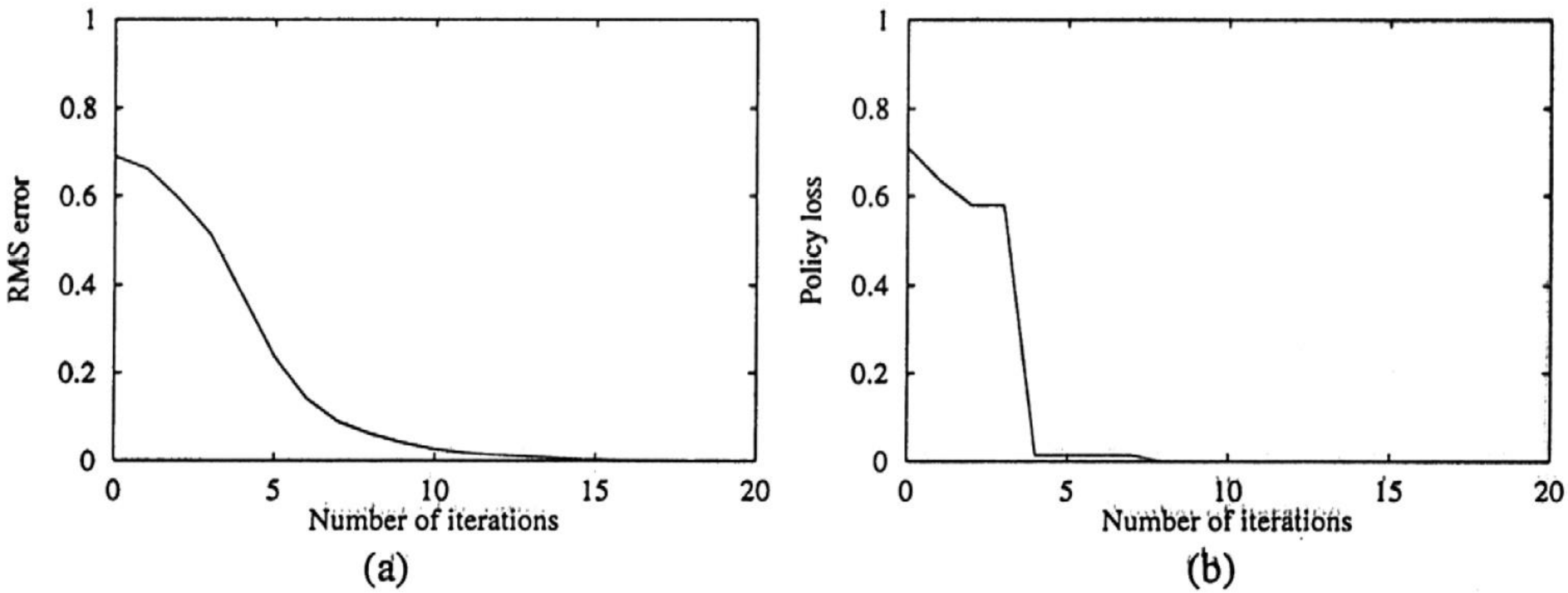


Figure 17.6 (a) The RMS (root mean square) error of the utility estimates compared to the correct values, as a function of iteration number during value iteration. (b) The expected policy loss compared to the optimal policy.

[Russel, AIMA 2010]

Unknown transition model

- So far: learning optimal policy when we know $\mathcal{P}_{ss'}^{a}$ and $\mathcal{R}_{ss'}^{a}$
  - it requires prior knowledge of the environment's dynamics

- If a model is not available, then it is particularly useful to estimate *action* values rather than *state* values

Unknown transition model: action value

- With a model, state values alone are sufficient to determine a policy
  - simply look ahead one step and chooses whichever action leads to the best combination of reward and next state

$$\pi^*(s) = \operatorname*{argmax}_{a \in \mathcal{A}(s)} \sum_{s'} \mathcal{P}_{ss'}^{a} \left(\mathcal{R}_{ss'}^{a} + \gamma V^*(s') \right)$$

  - Without a model, state values alone are not sufficient.

- However, if agent knows $Q(s, a)$, it can choose optimal action without knowing $\mathcal{P}_{ss'}^{a}$ and $\mathcal{R}_{ss'}^{a}$:

$$\pi^*(s) = \operatorname*{argmax}_{a} Q(s, a)$$

Monte Carlo methods

- do not assume complete knowledge of the environment
- require only *experience*
  - sample sequences of states, actions, and rewards from on-line or simulated interaction with an environment
- are based on averaging sample returns
  - are defined for episodic tasks

A Monte Carlo control algorithm using exploring starts

1) Initialize Q and π arbitrarily and $Returns$ to empty lists
2) Repeat

   Generate an episode using π and exploring starts

   for each pair of s and a appearing in the episode

   $R \leftarrow$return following the first occurrence of s, a

   Append R to $Returns(s, a)$

   $Q(s, a) \leftarrow average(Returns(s, a))$

   for each s in the episode

   $\pi(s) \leftarrow \underset{a}{\operatorname{argmax}}\, Q(s, a)$

A Monte Carlo control algorithm

1) Initialize Q and π arbitrarily and $Returns$ to empty lists
2) Repeat

Generate an episode using π

for each pair of s and a appearing in the episode

$R \leftarrow$return following the first occurrence of s, a

Append R to $Returns(s, a)$

$Q(s, a) \leftarrow average(Returns(s, a))$

for each s in the episode

$a^* \leftarrow \operatorname{argmax}_a Q(s, a)$

$$\pi(s, a) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|} & a = a^* \\ \frac{\epsilon}{|\mathcal{A}(s)|} & a \neq a^* \end{cases}$$

Temporal difference methods

- TD learning is a combination of MC and DP ideas.
  - Like MC methods, can learn directly from raw experience without a model of the environment's dynamics.
  - Like DP, update estimates based in part on other learned estimates, without waiting for a final outcome.

Temporal difference on value function

- $V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$

π: the policy to be evaluated

1) Initialize $V(s)$ arbitrarily
2) Repeat (for each episode)

    Initialize s

    a ←action given by policy π for s

    Take action a; observe reward r, and next state s'

    $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$

   until s is terminal

fully incremental fashion

Q-learning

- Update rule for doing action a in state s and achieving reward r:

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

$\hat{Q}(s,a)$ after n-th visit of s,a

- We can prove convergence of $\hat{Q}$ to Q (under certain assumptions)

$$\lim_{n\to\infty} \hat{Q}_n(s,a) = Q^*(s,a), \quad \forall s \in S, a \in A$$

Q-learning algorithm: Non-deterministic environments

Initialize $\hat{Q}(s,a)$ arbitrarily

Repeat (for each episode):

Initialize s

Repeat (for each step of episode):

e.g., greedy, ε-greedy

Choose a from s using a policy derived from $\hat{Q}$

Take action a, receive reward r, observe new state s'

$$\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \alpha \left[r + \gamma \max_{a'} \hat{Q}(s',a') - \hat{Q}(s,a) \right]$$

$s \leftarrow s'$

until s is terminal

Exploration/exploitation tradeoff

- Exploitation: High rewards from trying previously-well-rewarded actions

- Exploration: Which actions are best?
  - Must try ones not tried before

Q-learning: Policy

- Greedy action selection:

$$\pi(s) = \underset{a}{\operatorname{argmax}}\, \hat{Q}(s,a)$$

- ϵ-greedy: greedy most of the times, occasionally take a random action

- Softmax policy: Give a higher probability to the actions that currently have better utility, e.g,

$$\pi(s,a) = \frac{b^{\hat{Q}(s,a)}}{\sum_{a'} b^{\hat{Q}(s,a')}}$$

- After learning Q^*, the policy is greedy?

Q-learning convergence

- Q-learning converges to optimal Q-values if
  - Every state is visited infinitely often
  - The policy for action selection becomes greedy as time approaches infinity
  - The step size parameter is chosen appropriately

Step size parameter

- Stochastic approximation conditions
  - The learning rate is decreased fast enough but not too fast

- One of choices for α_n

$$\alpha_n = \frac{1}{visits_n(s,a)}$$

Tabular methods: Problem

- All of the introduced methods maintain a table
- Table size can be very large for complex environments
- We may not even visit some states
- But computation and memory problem

Function Approximation

- Use an approximate functional representation to generalize over states.
  - Instead of huge tables for $V(s)$ and $Q(s,a)$, we approximate $V(s)$ and $Q(s,a)$ with any supervised learning methods

$$V_{\boldsymbol{\theta}}(s) = \theta_1 f_1(s) + \cdots + \theta_m f_m(s)$$

or

$$Q_{\boldsymbol{\theta}}(s,a) = \theta_1 f_1(s,a) + \cdots + \theta_m f_m(s,a)$$

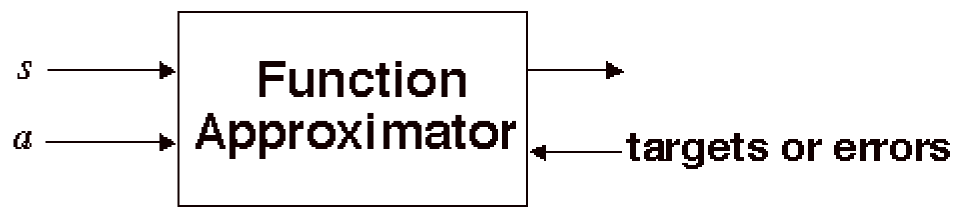
- We can generalize from visited states to unvisited ones.
  - In addition to the less space requirement

Adjusting function weights

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\left[r + \gamma\,\hat{V}_{\boldsymbol{\theta}}(s') - \hat{V}_{\boldsymbol{\theta}}(s)\right]\nabla_{\boldsymbol{\theta}}\hat{V}_{\boldsymbol{\theta}}(s)$$

or

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\left[r + \gamma\max_{a'}\hat{Q}_{\boldsymbol{\theta}}(s',a') - \hat{Q}_{\boldsymbol{\theta}}(s,a)\right]\nabla_{\boldsymbol{\theta}}\hat{Q}_{\boldsymbol{\theta}}(s,a)$$



Tesauro used function approximation in his Backgammon playing temporal difference learning research.
TD-Gammon plays at level of best human players (learn through self play)

Applications

- Control & robotics
  - Autonomous helicopter
    - self-reliant agent must do to learn from its own experiences.
    - eliminating hand coding of control strategies
- Board games
- Resource (time, memory, channel, …) allocation

References

- T. Mitchell, Machine Learning, MIT Press, 1998. [Chapter 13]

- R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction, MIT Press, 1999 [Chapters 1,3,4,6].