CS 5/7320
Artificial Intelligence

Reinforcement Learning
AIMA Chapter 17+22

Slides by Michael Hahsler
with figures from the AIMA textbook.

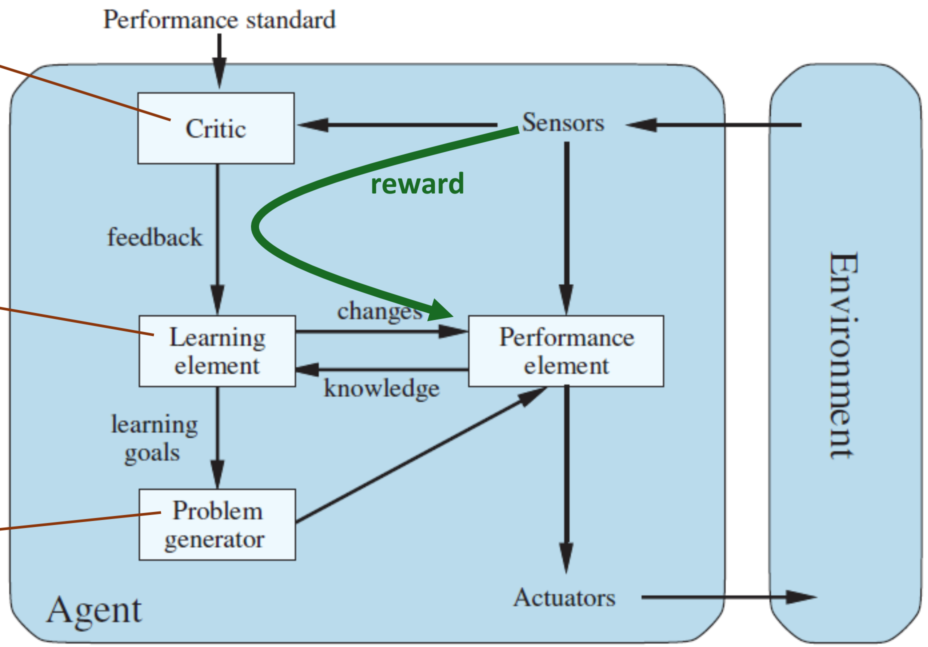
Online Material

From Chapter 2:
Agents That Learn

Critic: How is the agent currently performing?

Learning Element: Improves the performance element and changes how it selects actions.
E.g., adding rules, changing weights

Problem generators: Explore new actions.



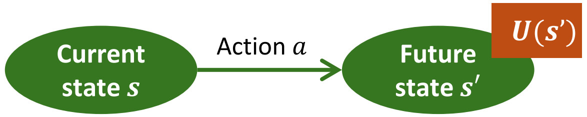
Positive feedback from the critic, called "reward," reinforces the performance element.

Reinforcement Learning: How do we learn a good performance element from rewards using trial-and-error?

Making Complex Decisions: Sequential Decision Making

AIMA Chapter 17

Remember Chapter 16:
Making Simple Decisions



For a decision that we make frequently and making it once does not affect the future decisions (**episodic environment**), we can use the **Principle of Maximum Expected Utility (MEU).**

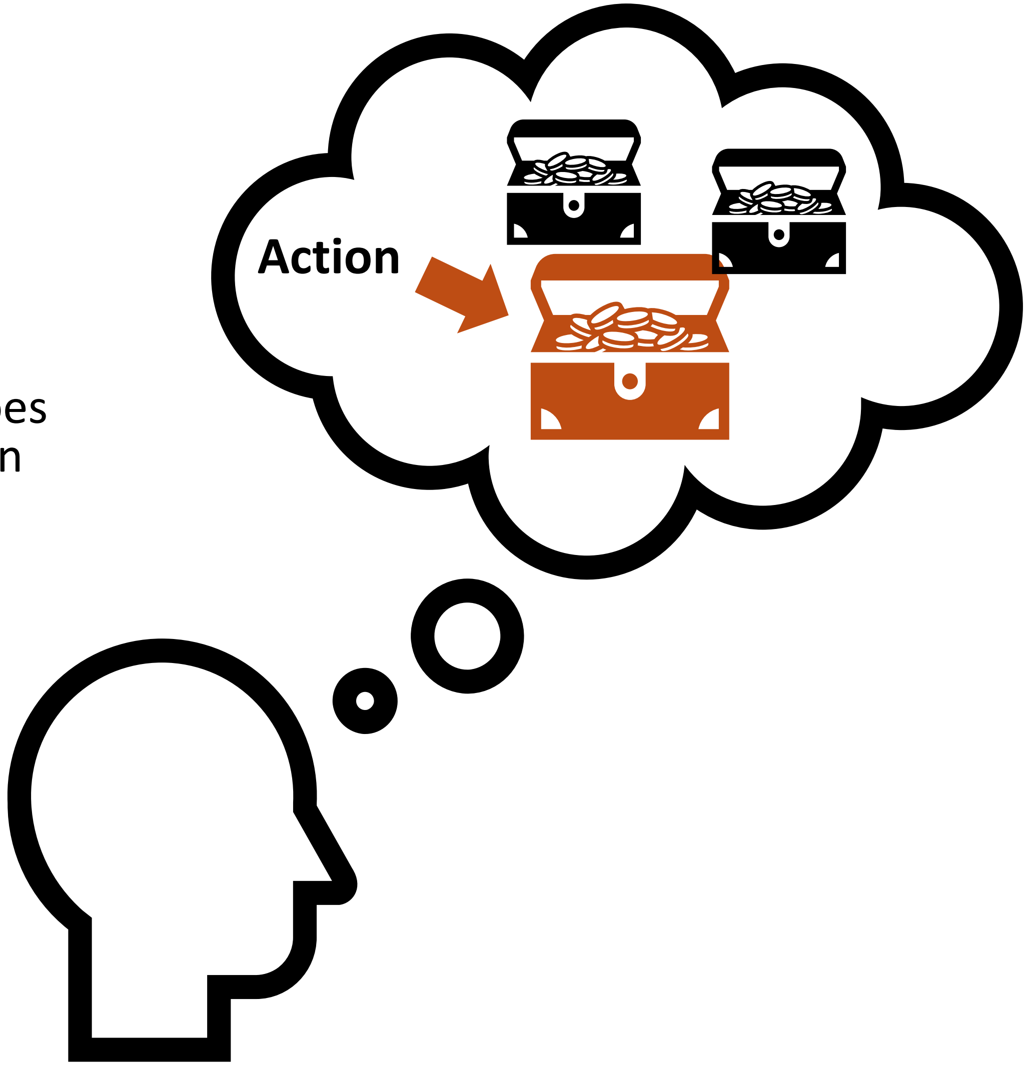
Given the expected utility of an action

$$EU(a) = \sum_{s'} \sum_{s} P(s)\, P(s'|s,a)\, U(s')$$

choose action that maximizes the expected utility:

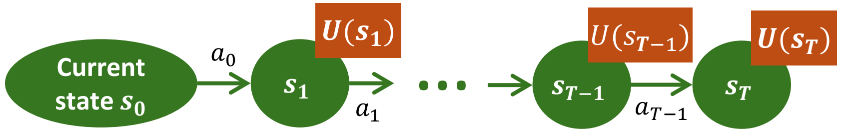
$$a^* = \text{argmax}_a\, EU(a)$$

Now we will talk about **sequential decision making**.

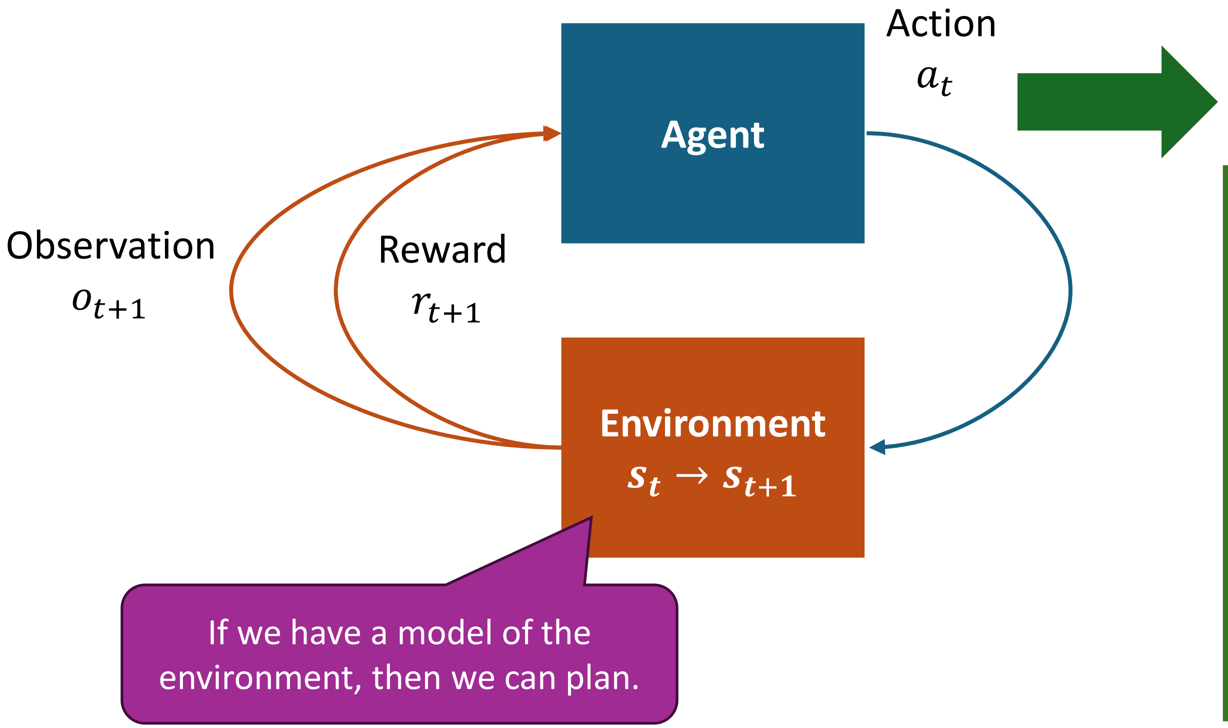


$P(s)$ … Uncertainty about current state (= partial observability)
$P(s'|s,a)$ … Stochastic transition function (= non deterministic actions).
$U(s')$ … cardinal utility function.

Sequential Decision Problems



- **Utility-based agent**: The agent's utility depends on a sequence of decisions that depend on each other.
- Sequential decision problems incorporate utility (called immediate and long-term reward), uncertainty, and sensing.



Sequence: $\underbrace{(o_0, r_0)}_{s_0}, a_0, \underbrace{(o_1, r_1)}_{s_1}, a_1, \underbrace{(o_2, r_2)}_{s_2}, a_2, \ldots$

Goal: Observations and rewards depend on the state of the system, and the agent wants to maximize the expected discounted reward:

$$U = \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t R_t\right]$$

R_t… reward at time t as a random variable
γ … discounting factor
T … time horizon may be infinity

An Environment Model: Markov Decision Process (MDP)

MDPs are discrete-time stochastic control processes defined by:

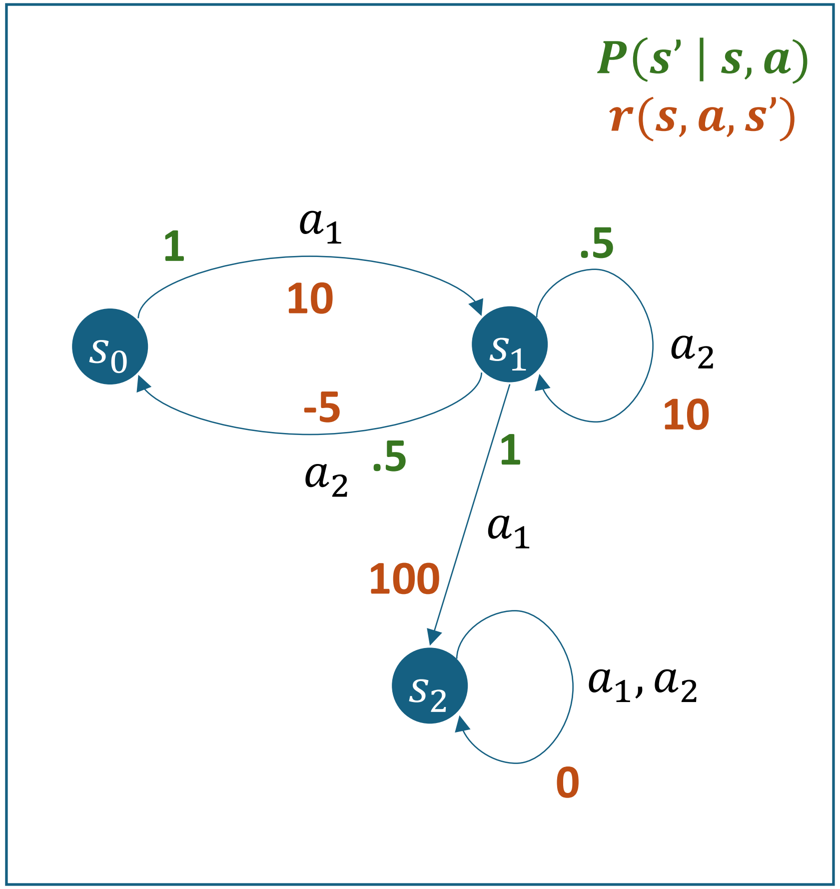
- a finite set of **states** $\mathcal{S} = \{s_0, s_1, s_2, \dots\}$ (initial state s_0)
- a set of available **actions** $ACTIONS(s)$ in each state s
- a **transition model** $P(s' \mid s, a)$ where $a \in ACTIONS(s)$
- a **reward function** $r(s)$ where the immediate reward depends on the current state (often $r(s, a, s')$ is used to make modelling easier)

MDPs model sequential decision problems with

- a fully observable, stochastic, and known environment;
- a Markovian transition model (i.e., future states do not depend on past states given the current state);
- additive immediate rewards.

Time horizon

- **Infinite horizon**: non-episodic (continuous) tasks with no terminal state.
- **Finite horizon**: episodic tasks. Episode ends after a number of periods or when a terminal state is reached. Episodes contain a sequence of several actions that affect each other.

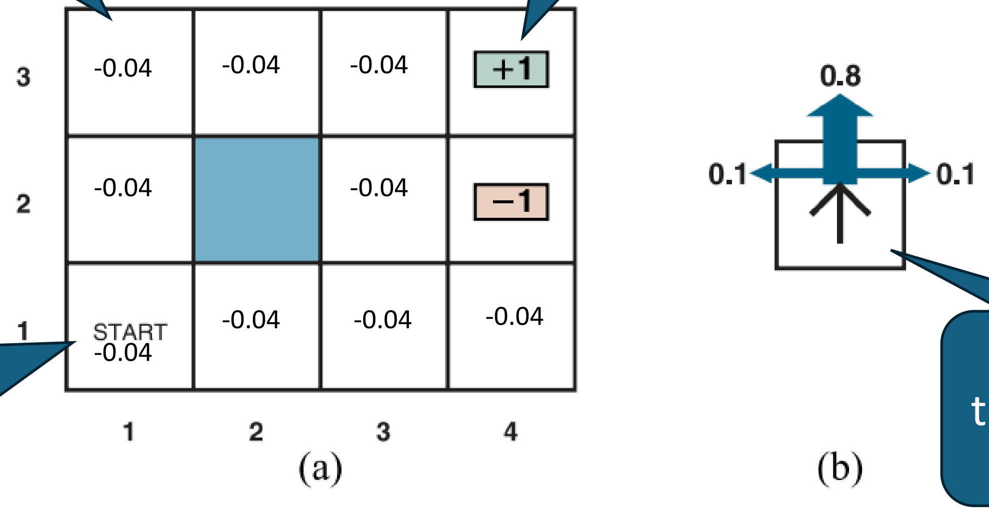


Example: 4x3 Grid World

$ACTIONS(s)$

Rewards $r(s)$

States S are squares. START is the initial state



Stochastic transition model $P(s' \mid s, a)$

Figure 17.1 (a) A simple, stochastic 4×3 environment that presents the agent with a sequential decision problem. (b) Illustration of the transition model of the environment: the "intended" outcome occurs with probability 0.8, but with probability 0.2 the agent moves at right angles to the intended direction. A collision with a wall results in no movement. Transitions into the two terminal states have reward +1 and –1, respectively, and all other transitions have a reward of –0.04.

Since we know the complete MDP model, we can solve this as a **planning problem**.

For each square: specify what direction should we try to go to maximize the expected total utility.

This is called a **policy** written as the function

$$\pi: S \rightarrow ACTIONS(S)$$

Policy as a Table

s	**Action $\pi(s)$**
(1,1)	Up
…	…
…	…

Value Function

- A policy $\boldsymbol{\pi} = \{\pi(s_0), \pi(s_1), \dots\}$ defines for each state which action to take.
- The expected utility of being in state s under policy π (i.e., following the policy starting from s) can be calculated as the sum of the immediate rewards over the visited sequence of states:

$$U^\pi(s) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t r(s_t) \,|\, s_0 = s\right]$$

- $U^\pi(s)$ (also often written as $V(s)$) is called **the value function**. It is stored as a table.

γ .. Discounting factor to give more weight to immediate rewards.

$\mathbb{E}_\pi$ ... Expectation over sequences that can be created by following π.

$r(s)$ .. Reward function.

Value Function

	1	2	3	4
3	0.8516	0.9078	0.9578	+1
2	0.8016		0.7003	−1
1	0.7453	0.6953	0.6514	0.4279

Value Function

s	State Value $V(s)$
(1,1)	0.7453
(1,2)	0.8016
…	…

Planning: Finding the Optimal Policy

- The goal of solving an MDP is to find an optimal policy $\boldsymbol{\pi}$ that maximizes the expected future utility for each state

$$\pi^*(s) = \underset{\pi}{\operatorname{argmax}}\, U^\pi(s) \quad \text{for all } s \in \mathcal{S}$$

- **Issue**: π^* depends on U^π and vice versa!

- The problem can be formulated recursively using the **Bellman equation** which holds for the optimal value function U (“Bellman optimality condition”):

$$U^{\pi^*}(s) = r(s) + \gamma \max_{a \in A} \sum_{s'} P(s'|s,a)\; U^{\pi^*}(s')$$

Immediate Reward

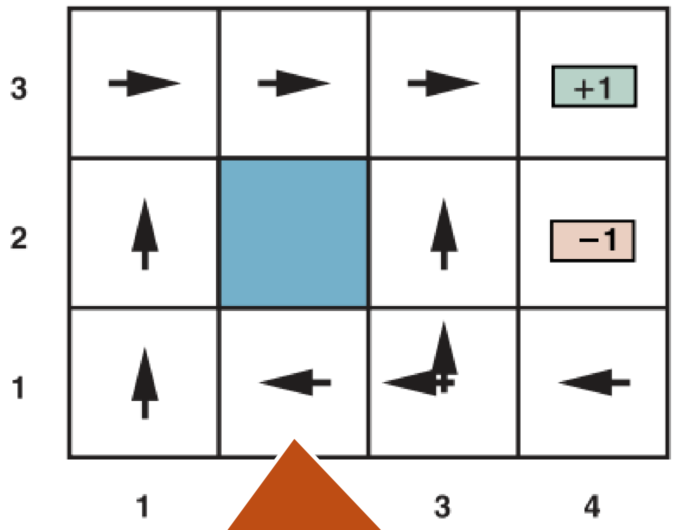
π^* uses the best action

Expectation

Utility of the next state

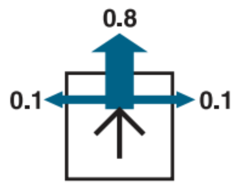
Example Solution: 4x3 Grid World

Optimal action in each state (policy π^*)



It is optimal to walk away from the +1 square to avoid the -1 square!

Greedy policy:
Always pick the action leading to the state with the highest expected utility.



Value of being in a state $U^{\pi^*}(s)$ (given that we will follow π^*)

	1	2	3	4
3	0.8516	0.9078	0.9578	+1
2	0.8016		0.7003	−1
1	0.7453	0.6953	0.6514	0.4279

$\gamma = 1$

How do we find the optimal value function/optimal policy?

Policy Iteration

Value Iteration

Q-Function

- $Q(s,a)$ is called the state-action value function. It gives the expected utility of taking action a in state s and then following the policy.

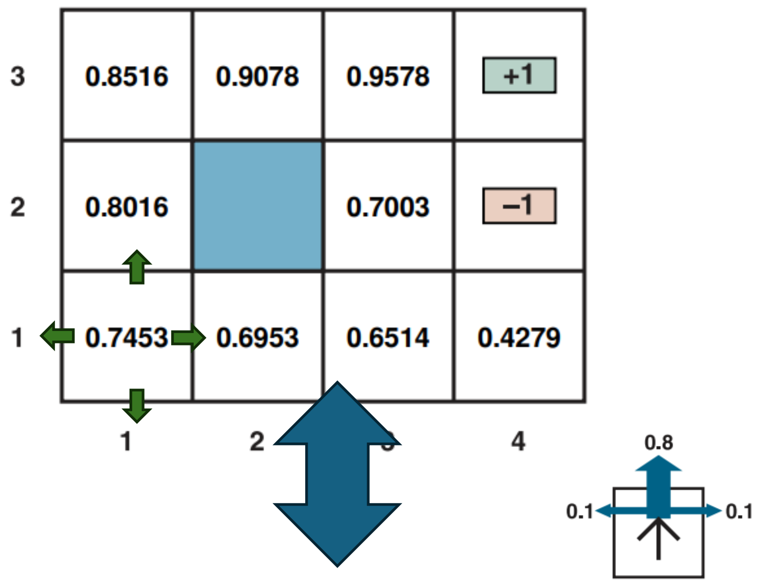
$$Q(s,a) = r(s) + \gamma \sum_{s'} P(s'|s,a)U(s')$$

Immediate Reward

Expected utility of the next state

- The Relationship with the state value function: $U(s) = \max_{a \in A(s)} Q(s,a)$
- The Q-function lets us compare the value of taking different action is a given state. It is used in algorithms to determine what action is the best.

Value Function



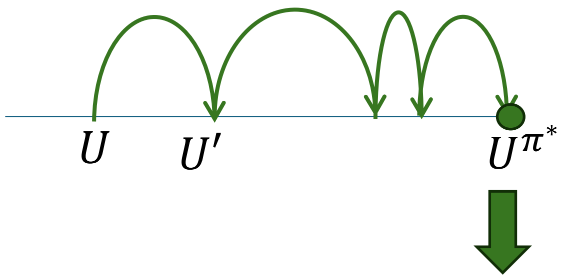
Q-Table

s	a	$Q(s,a)$
(1,1)	Up	0.7453
(1,1)	Right	0.6709
(1,1)	Down	0.7003
(1,1)	Left	0.7109
…	…	…

Value Iteration: Estimate the Optimal Value Function U^{π^*}

Algorithm: Start with a U table (vector) of 0 for all states and then apply the Bellman update over the entries of the table until it converges to the unique optimal solution U^{π^*}.

Bellman updates



Extract greedy π^*

Guarantee: It will converge because the optimal solution is a fixed point of the Bellam operator.

function VALUE-ITERATION(mdp, ϵ) **returns** a utility function
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$,
      rewards $R(s, a, s')$, discount γ
    ϵ, the maximum error allowed in the utility of any state
  local variables: U, U', vectors of utilities for states in S, initially zero
      δ, the maximum relative change in the utility of any state

  repeat
    $U \leftarrow U'$; $\delta \leftarrow 0$
    for each state s **in** S **do**
      $U'[s] \leftarrow \max_{a \in A(s)}$ Q-VALUE(mdp, s, a, U)
      if $|U'[s] - U[s]| > \delta$ **then** $\delta \leftarrow |U'[s] - U[s]|$
  until $\delta \leq \epsilon(1-\gamma)/\gamma$
  return U

Sweep over the U table

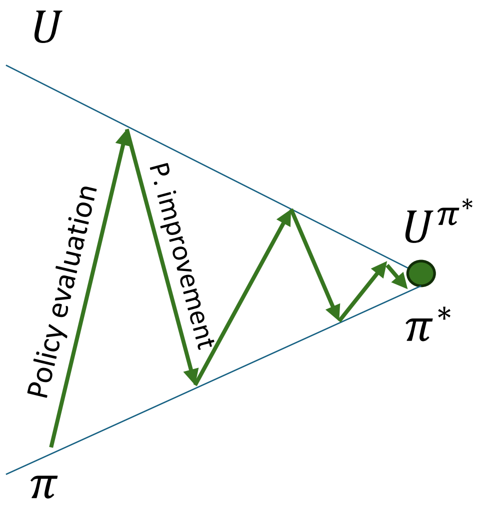
Bellman update: Value of the best action in state s.

Convergence? Uses a proxy for policy loss $||U^\pi - U||_\infty$ as the stopping criterion

U converges to U^{π^*} and we can extract π^*

Policy Iteration: Find the Optimal Policy π^*

Policy iteration tries to directly find the optimal policy by iterating policy evaluation and improvement.




```
function POLICY-ITERATION(mdp) returns a policy
  inputs: mdp, an MDP with states S, actions A(s), transition model P(s' | s, a)
  local variables: U, a vector of utilities for states in S, initially zero
                   π, a policy vector indexed by state, initially random

  repeat
     U ← POLICY-EVALUATION(π, U, mdp)
     unchanged? ← true
     for each state s in S do
        a* ← argmax_{a ∈ A(s)} Q-VALUE(mdp, s, a, U)
        if Q-VALUE(mdp, s, a*, U) > Q-VALUE(mdp, s, π[s], U) then
           π[s] ← a*; unchanged? ← false
  until unchanged?
  return π
```

Estimate U given the current policy (either solve an LP or value iteration with fixed policy)

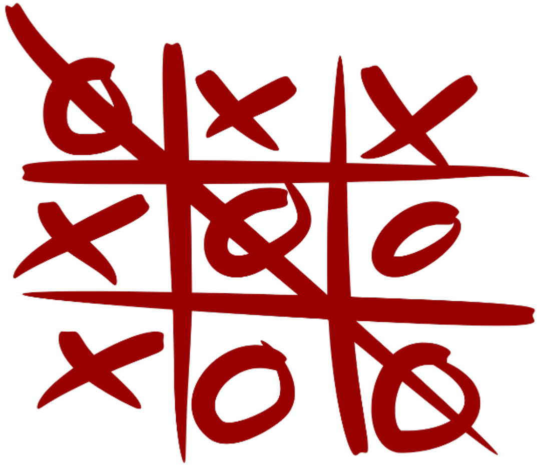
Greedy policy Improvement.

Convergence test

π converges to π^*
(and U converges to U^{π^*})

Guarantee: It will converge because each step improves the utility/policy and there is a finite number of steps.

Find the Optimal Policy for Tic-Tac-Toe



Definitions from Chapter 5 on Games for a goal-based agent:

s_0	Empty board.
$Actions(s)$	Play empty squares.
$Result(s, a)$	Symbol (x/o) is placed on empty square. → Stochastic transition model $P(s'\|s, a)$
$Terminal(s)$	Did a player win or is the game a draw?
$Utility(s)$	+1 if x wins, -1 if o wins and 0 for a draw. Utility is only defined for terminal states. → Reward function $r(s)$

Implementation as a planning agent:

1. Model the environment as an MDP. It is completely described by the rules of the game.
2. Plan the optimal policy $\pi^*(s)$ for each state (e.g., using value iteration).
3. Executed the policy.

Potential issues:

- There are many states, so the state value table $U(s)$ has many entries.
- The stochastic transition model $P(s'|s, a)$ needs to be known. We need to assume the other player's policy.
- The tables (value function, policy) are very large. This does not scale for more complicated games (e.g., Connect-4).
- For games, all the rewards are delayed. Immediate rewards are always 0 until the end of the game.

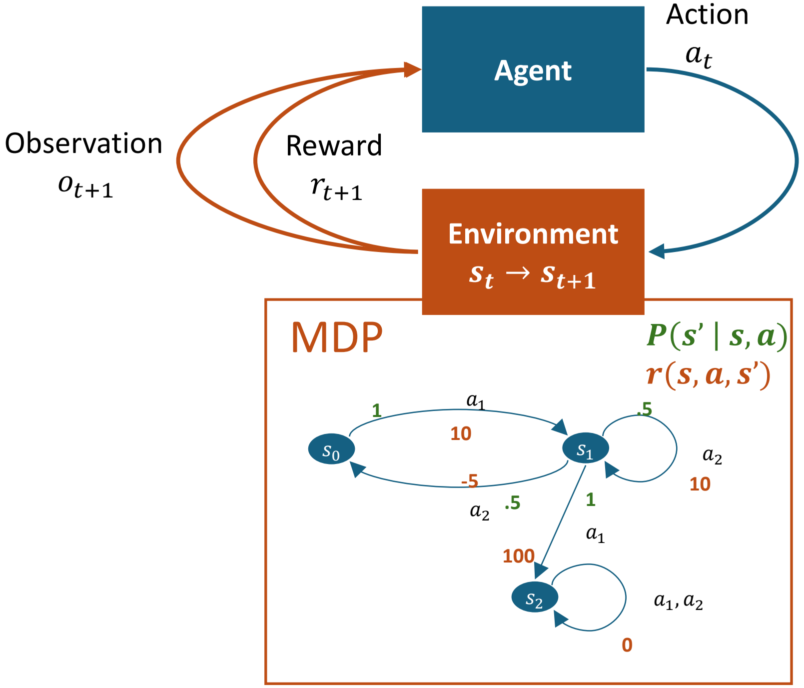
This makes planning hard! An alternative solution is to use online learning with model-free reinforcement learning methods.

(Model-Free) Reinforcement Learning

AIMA Chapter 22

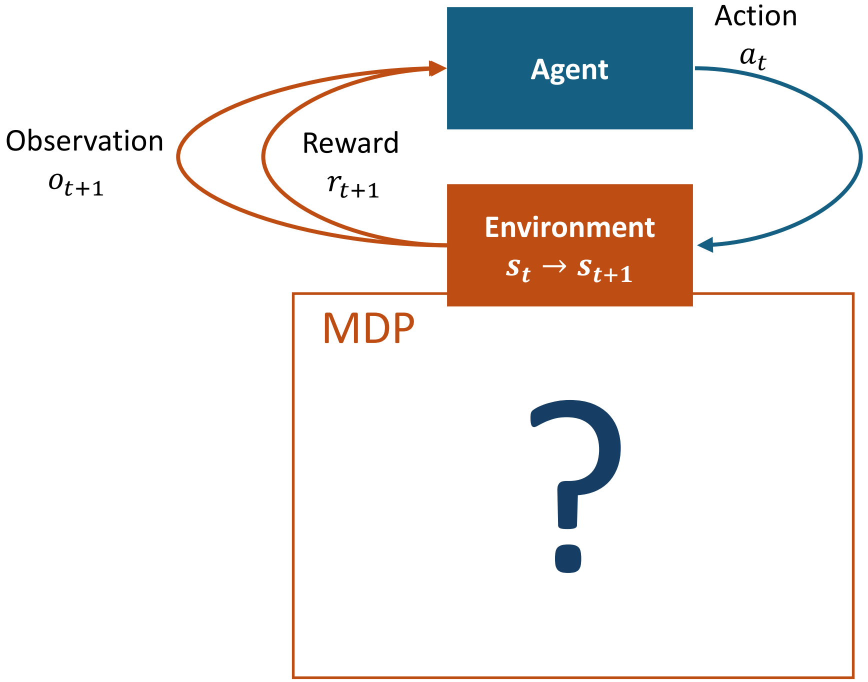
Model-based vs. Model-free Reinforcement Learning (RL)

Model-based RL



Use the MDP model for **planning**
(e.g., value iteration, policy iteration)

Model-free RL



An unknown MDP model means we have to try actions and use **online learning**.

Reinforcement Learning (RL)

- RL assumes that the problem can be modeled as a **Markov Decision Process (MDP)**.

- However, we do not know the transition or the reward model. This means we have an **unknown environment,** and we need “model-free” methods.

- We cannot use offline planning in unknown environments. The agent needs to explore the environment (try actions) and **use the reward signal to update its estimate of the utility of states and actions**. This is an online learning process that provides positive reinforcement through rewards.

- A popular algorithm is Q-Learning, which tries to learn the state-action value function of important states.

Q-Learning

Q-Table ➡ $\pi(s) = \underset{a \in A(s)}{\operatorname{argmax}}\, Q(s,a)$

s	a	$Q(s,a)$

Q-Learning learns the state-action value function as a table from interactions with the environment.

function Q-LEARNING-AGENT(*percept*) **returns** an action
  inputs: *percept*, a percept indicating the current state s' and reward signal r
  persistent: Q, a table of action values indexed by state and action, initially zero
    N_{sa}, a table of frequencies for state–action pairs, initially zero
    s, a, the previous state and action, initially null

  if s is not null **then**
    increment $N_{sa}[s,a]$
    $Q[s,a] \leftarrow Q[s,a] + \alpha(N_{sa}[s,a])(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$
  $s, a \leftarrow s', \operatorname{argmax}_{a'} f(Q[s',a'], N_{sa}[s',a'])$
  return a

Encodes learned policy

A new episode starts with no previous state.

Learning rate

Make $Q[s,a]$ a little more similar to the received reward + the best Q-value of the successor state.

TD-error

Behavior policy: $f(\cdot)$ is the exploration function and decides on the next action. As N increases, it can exploit good actions more.

Tabular Methods vs. Value Function Approximation

- U (or Q) tables needs to store and estimate one entry for each state (state/action combination).
- Issues and possible solutions
  - Too many entries to store → lossy compression
  - Many combinations are rarely seen → generalize to unseen entries
- **Idea**: Estimate the state value by learning an approximation function $\hat{U}(s) = h_{\boldsymbol{\theta}}(s)$ based on features of s (ML).
- **Example**: 4x3 Grid World with a linear combination of state features (x, y) and learn $\boldsymbol{\theta}$ from observed data.

Value function $U(s)$
Stored as a table

y \ x	1	2	3	4
3	0.8516	0.9078	0.9578	+1
2	0.8016		0.7003	–1
1	0.7453	0.6953	0.6514	0.4279

Table vs. approximate $U(s)$

Example: Linear approximation using state features (x, y)

$$\hat{U}_{\boldsymbol{\theta}}(x, y) = \theta_0 + \theta_1 x + \theta_2 y$$

$\boldsymbol{\theta} = (\theta_0, \theta_1, \theta_2)$ can be updated iteratively after each new observed reward using gradient descent.

Traditional Tabular Q-Learning

s → Q-Table → $Q(s,a)$

s	a	$Q(s,a)$
1	1	0.7
1	2	0.3
…	…	…

Q-Table

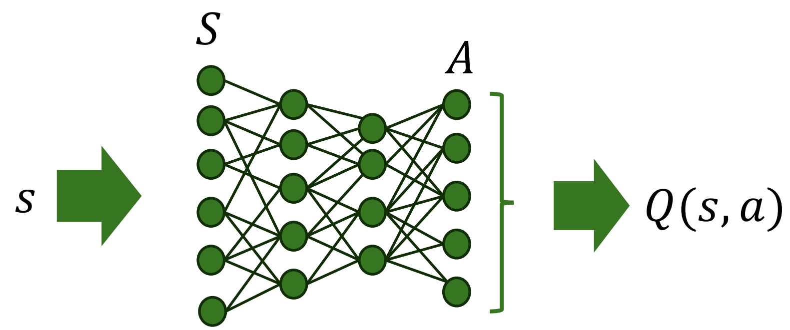
function Q-LEARNING-AGENT(*percept*) **returns** an action
  inputs: *percept*, a percept indicating the current state s' and reward signal r
  persistent: Q, a table of action values indexed by state and action, initially zero
    N_{sa}, a table of frequencies for state–action pairs, initially zero
    s, a, the previous state and action, initially null

  if s is not null **then**
    increment $N_{sa}[s,a]$
    $Q[s,a] \leftarrow Q[s,a] + \alpha(N_{sa}[s,a])(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$
  $s, a \leftarrow s', \operatorname{argmax}_{a'} f(Q[s',a'], N_{sa}[s',a'])$
  return a

target: $r + \gamma \max_{a'} Q[s',a']$

prediction: $Q[s,a]$

Deep Q-Learning



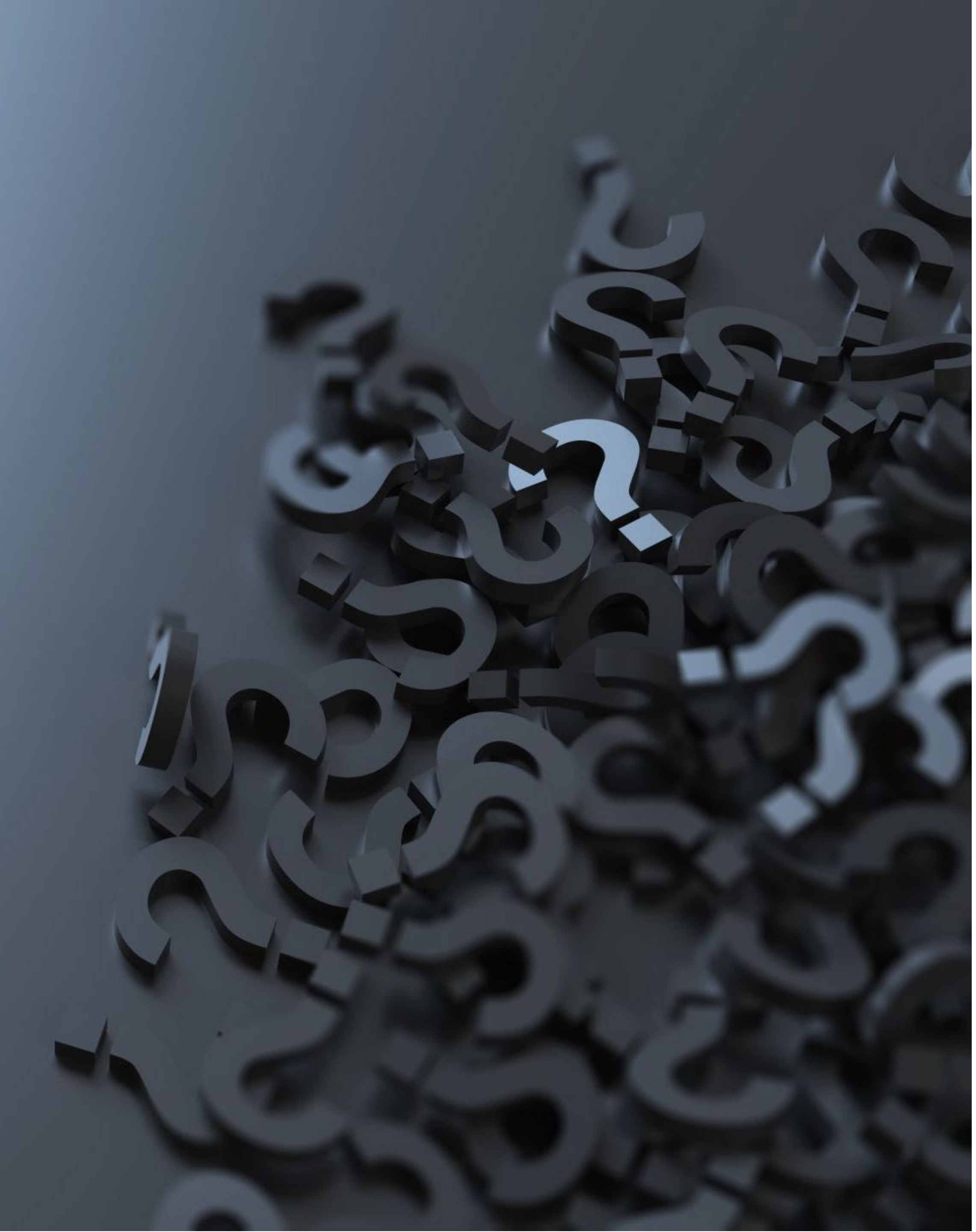
Q-Table as a Deep Q-Network (DQN)

Target networks: It turns out that the Q-Network is unstable if the same network is used to estimate $Q(s,a)$ and also $Q(s',a')$. Deep Q-Learning uses a second target network for $Q(s',a')$ that is updated with the prediction network every C steps.

Experience replay: To reduce instability more, generate actions using the current network and store the experience $\langle s,a,r,s' \rangle$ in a table. Regularly use samples from the table to update the networks..

Loss function: squared difference between prediction and target.

Volodymyr Mni et al., Playing Atari with Deep Reinforcement Learning, NIPS Deep Learning Workshop 2013.



Summary

- Agents can learn the value of being in a state from **reward signals**.
- Rewards can be delayed (e.g., at the end of a game).
- **Unknown transition models** lead to the need for exploration by trying actions (model-free methods such as Q-Learning).
- All RL problems are computationally very expensive and often can only be solved by **approximation**. The state-of-the-art approach is to use deep artificial neural networks for function approximation.
- Not covered here: Not being able to **fully observe the state** makes the problem more difficult and leads to Partially Observable MDPs.