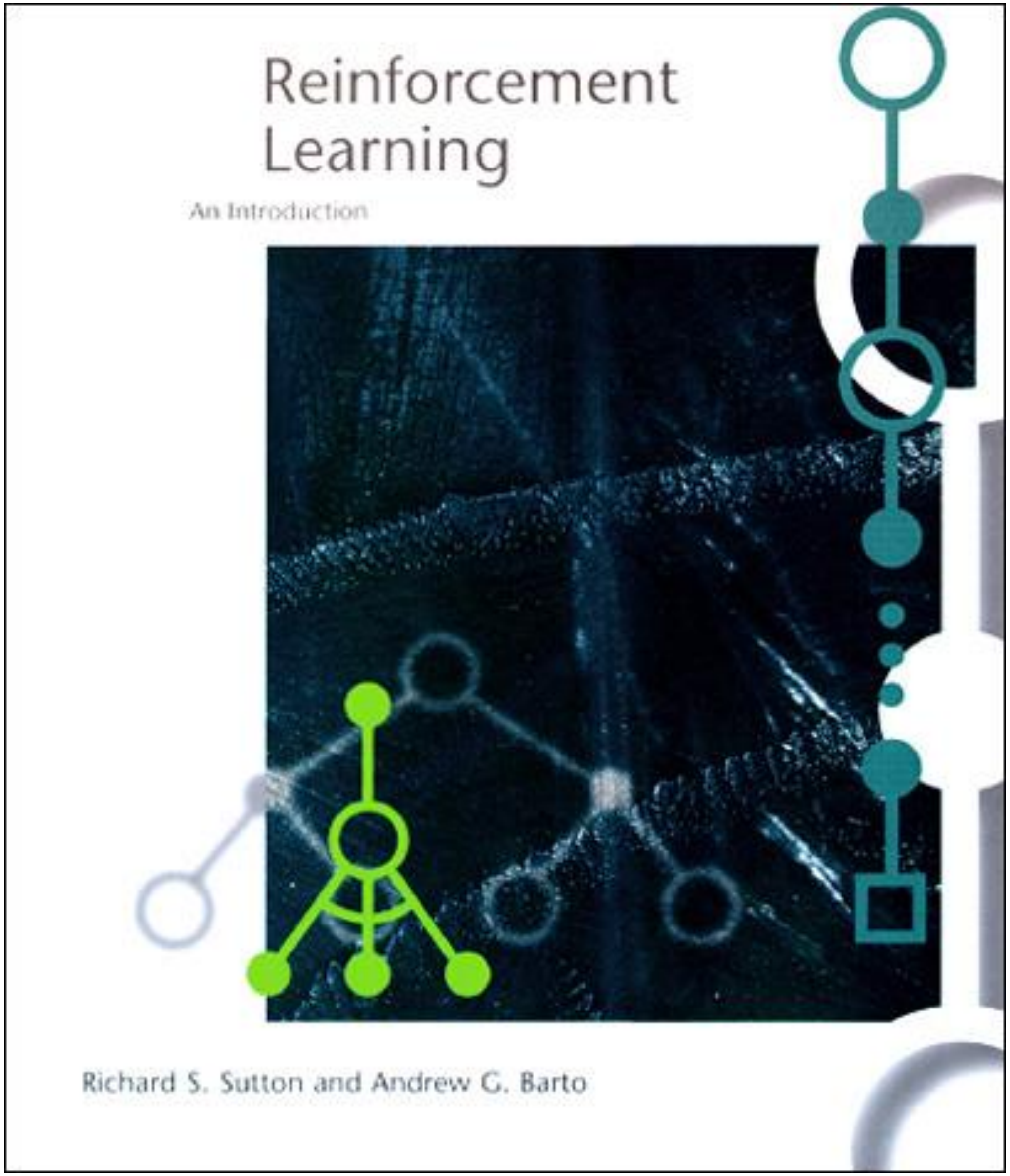
CMSC 471: Reinforcement Learning

Spring 2021 (Sections 01 & 03)

Some slides courtesy Cynthia Matuszek, with some material from Marie desJardin, Lise Getoor, Jean-Claude Latombe, and Daphne Koller

There's an entire book!

http://incompleteideas.net/book/the-book-2nd.html



The Big Idea

- "Planning": Find a sequence of steps to accomplish a goal.
  - Given start state, transition model, goal functions…
- This is a kind of **sequential decision making**.
  - Transitions are deterministic.
- What if they are stochastic (probabilistic)?
  - One time in ten, you drop your sock
- Probabilistic Planning: Make a plan that accounts for probability by **carrying it through the plan**.

Way back when…

Okay, but really? What is AI?

"Artificial intelligence, or AI, is the field that studies the synthesis and analysis of computational agents that act intelligently." --Poole & Mackworth

- Makes appropriate actions for circumstances & goals
- Balances short & long-term appropriately
- Flexible & reactive
- Learns/recognizes patterns
- Aware of computational/task budgets & limitations

Use "computation" to explain and traceback the actions

something that acts in an environment; it does something.

Way back when…

(0) Table-driven agents

simple

Use percept sequence/action table to find next action. Implemented by a **lookup table**

(1) Simple reflex agents

Based on **condition-action rules**, stateless devices with no memory of past world states

(2) Agents with memory

have **represent states** and keep track of past world states

(3) Agents with goals

Have a state and **goal information** describing desirable situations; can take future events into consideration

(4) Utility-based agents

base decisions on **utility theory** in order to act rationally

complex

Adapted from Tim Finin

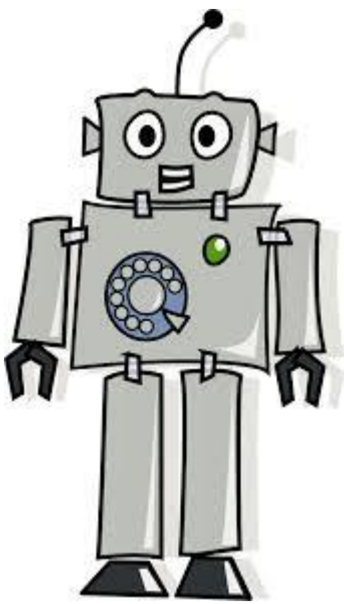
Review: Formalizing Agents

- Given:
  - A state space S
  - A set of actions $a_1, \ldots, a_k$ including their results
  - Reward value at the end of each trial (series of action) (may be positive or negative)
- Output:
  - A **mapping from states to actions**
  - Which is a **policy**, π

Reinforcement Learning

- We often have an agent which has a **task** to perform
  - It takes some actions in the world
  - At some later point, gets feedback on how well it did
  - The agent performs the same task repeatedly
- This problem is called **reinforcement learning**:
  - The agent gets positive reinforcement for tasks done well
  - And gets negative reinforcement for tasks done poorly
  - Must somehow figure out which actions to take next time

Reinforcement Learning



agent

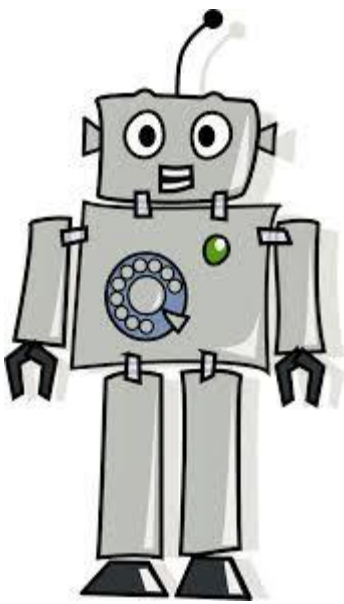
environment

Robot image: openclipart.org

https://static.vecteezy.com/system/resources/previews/000/090/451/original/four-seasons-landscape-illustrations-vector.jpg

Reinforcement Learning

take action



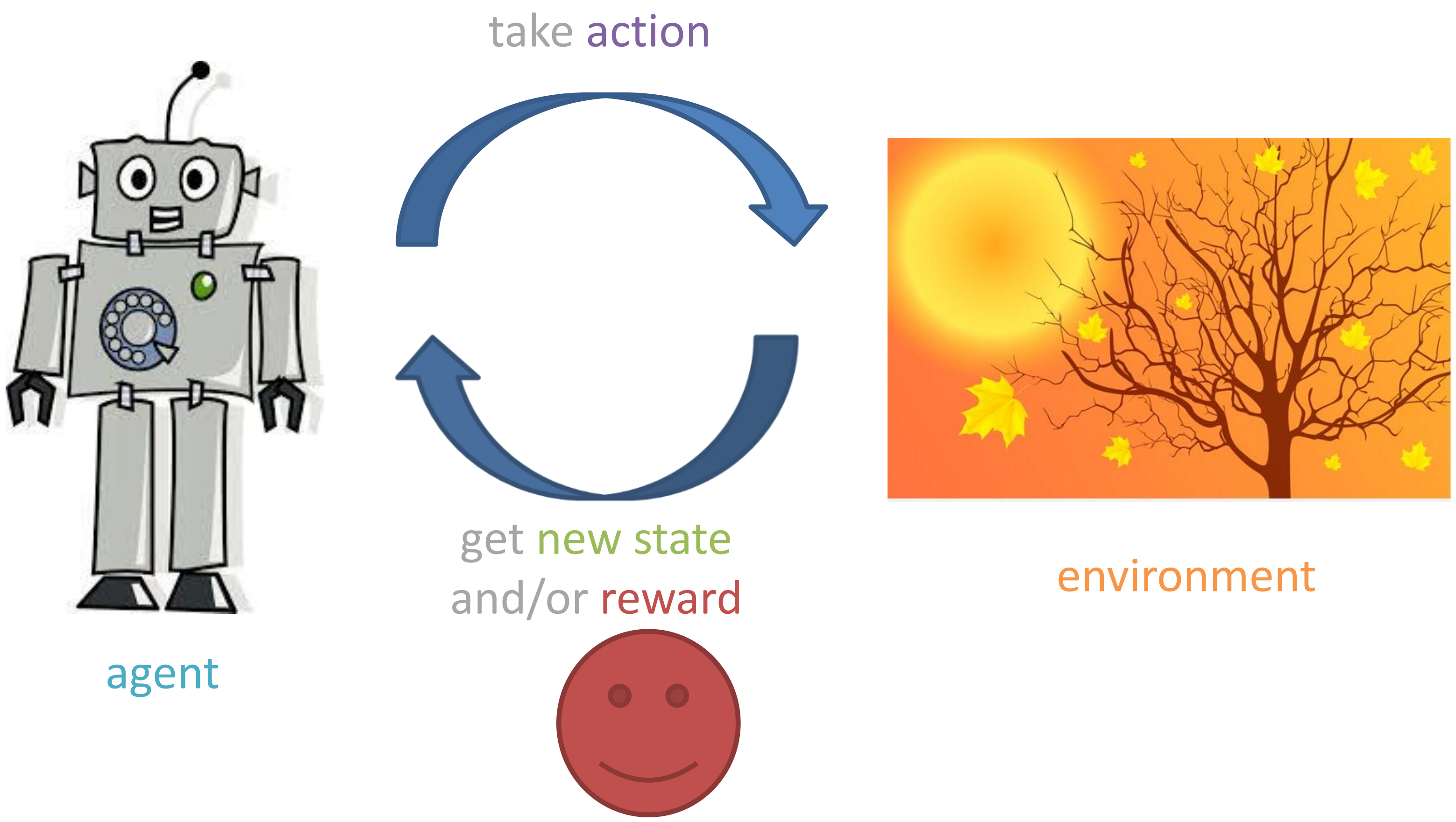
agent

environment

Robot image: openclipart.org

https://static.vecteezy.com/system/resources/previews/000/090/451/original/four-seasons-landscape-illustrations-vector.jpg

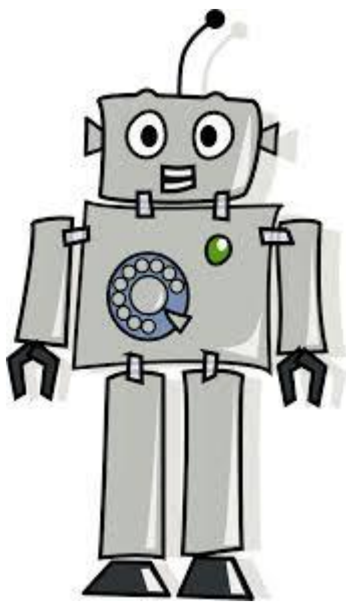
Reinforcement Learning



Robot image: openclipart.org

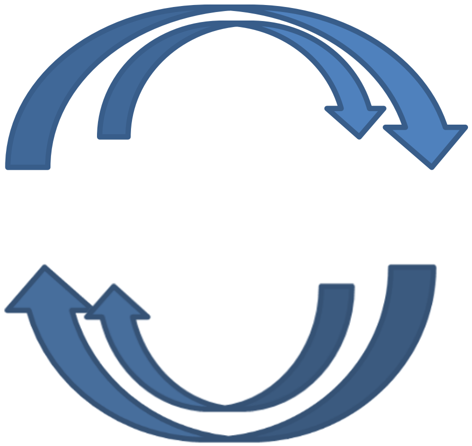
https://static.vecteezy.com/system/resources/previews/000/090/451/original/four-seasons-landscape-illustrations-vector.jpg

Reinforcement Learning



agent

take action



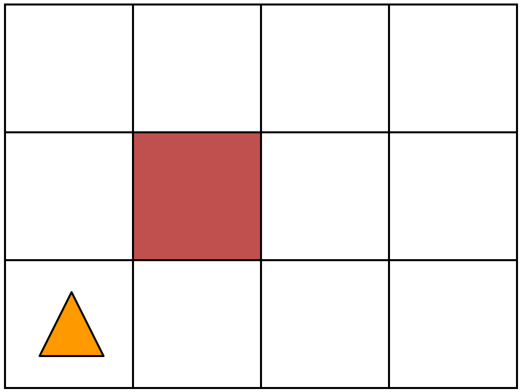
get new state
and/or reward

environment

Robot image: openclipart.org

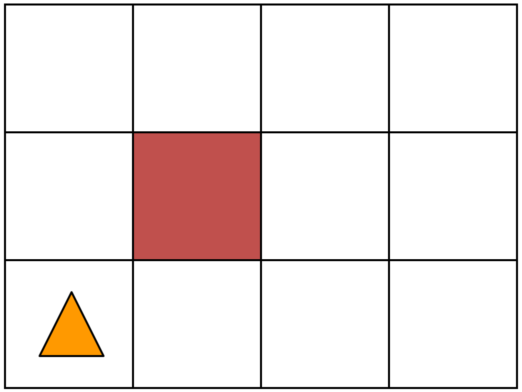
https://static.vecteezy.com/system/resources/previews/000/090/451/original/four-seasons-landscape-illustrations-vector.jpg

Simple Robot Navigation Problem



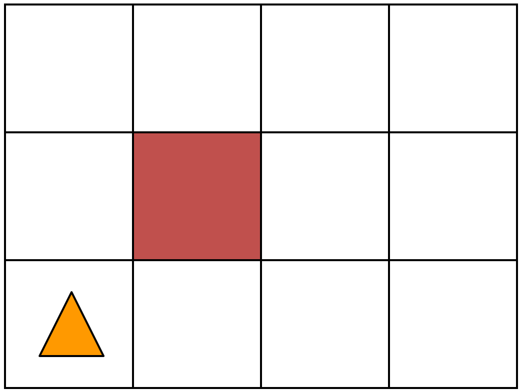
- In each state, the possible actions are U, D, R, and L

Probabilistic Transition Model



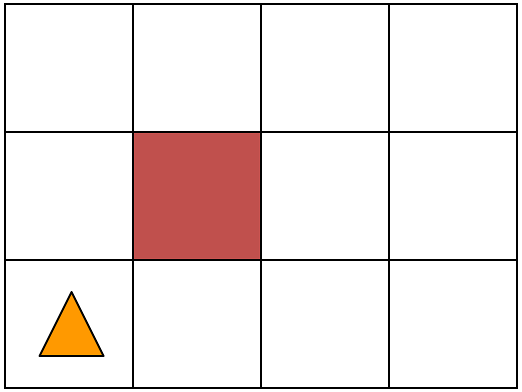
- In each state, the possible actions are U, D, R, and L
- The effect of U is as follows (transition model):
    - With probability 0.8, the robot moves up one square (if the robot is already in the top row, then it does not move)

Probabilistic Transition Model



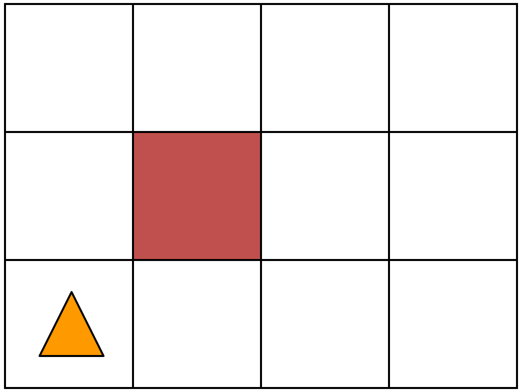
- In each state, the possible actions are U, D, R, and L
- The effect of U is as follows (transition model):
    - With probability 0.8, the robot moves up one square (if the robot is already in the top row, then it does not move)
    - With probability 0.1, the robot moves right one square (if the robot is already in the rightmost row, then it does not move)

Probabilistic Transition Model



- In each state, the possible actions are U, D, R, and L
- The effect of U is as follows (transition model):
  - With probability 0.8, the robot moves up one square (if the robot is already in the top row, then it does not move)
  - With probability 0.1, the robot moves right one square (if the robot is already in the rightmost row, then it does not move)
  - With probability 0.1, the robot moves left one square (if the robot is already in the leftmost row, then it does not move)

Probabilistic Transition Model



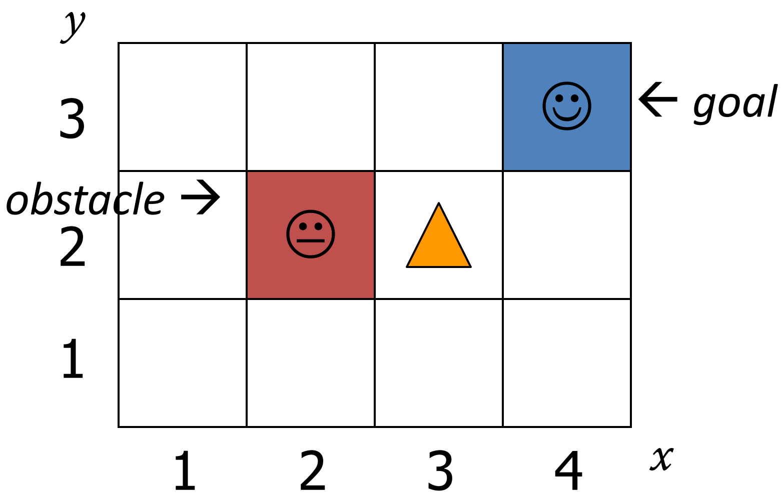
- In each state, the possible actions are U, D, R, and L
- The effect of U is as follows (transition model):
  - With probability 0.8, the robot moves up one square (if the robot is already in the top row, then it does not move)
  - With probability 0.1, the robot moves right one square (if the robot is already in the rightmost row, then it does not move)
  - With probability 0.1, the robot moves left one square (if the robot is already in the leftmost row, then it does not move)
- D, R, and L have similar probabilistic effects

Markov Property

The transition properties depend only on the current state, not on the previous history (how that state was reached)

Markov assumption generally: current state only ever depends on previous state (or finite set of previous states).

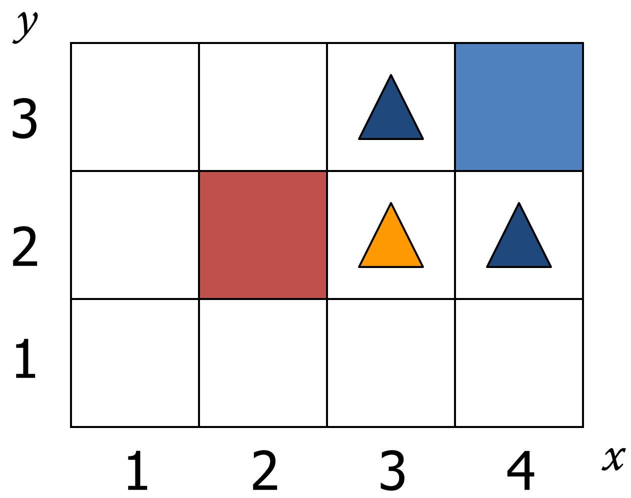
Sequence of Actions

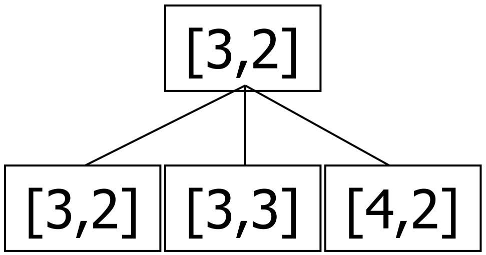


[3,2] ← *start state*

- Planned sequence of actions: (U, R)

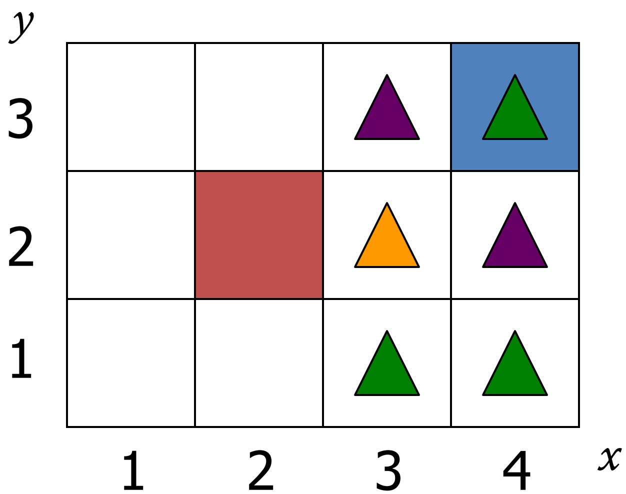
Sequence of Actions

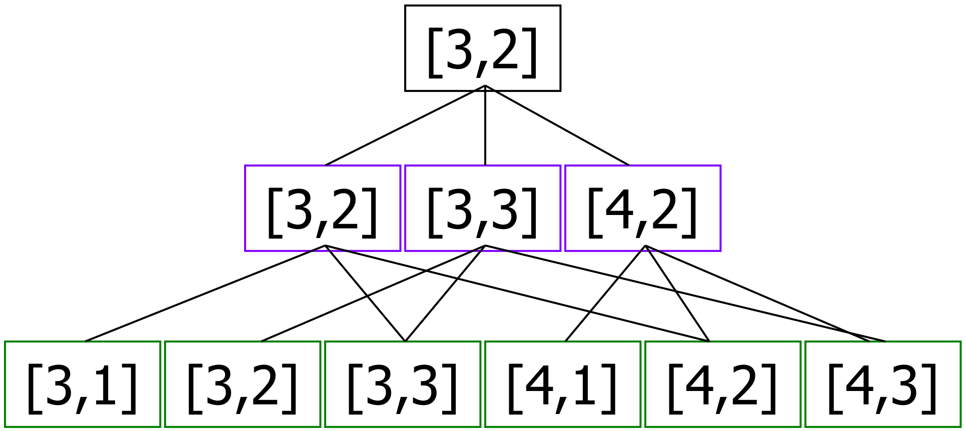




- Planned sequence of actions: (U, R)
- U is executed

Histories

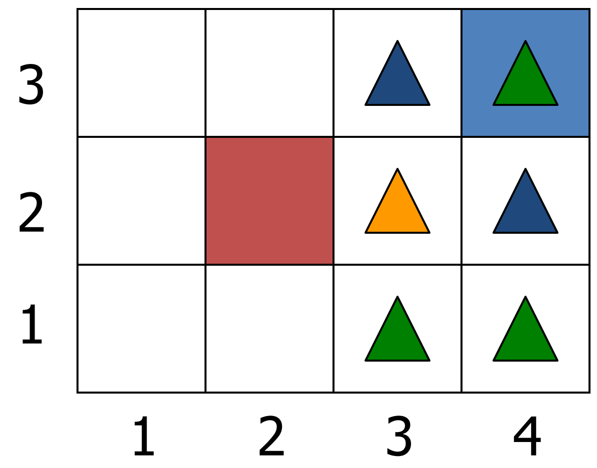




- Planned sequence of actions: (U, R)
- U has been executed
- R is executed

- 9 possible sequences of states – called histories
- 6 possible final states for the robot!

Probability of Reaching the Goal

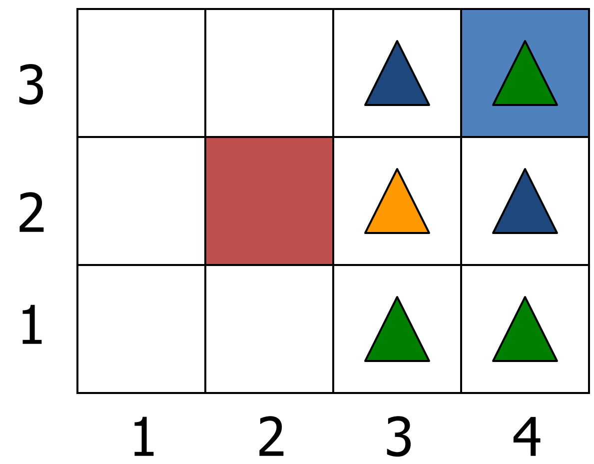


- **P**([4,3] | (U,R).[3,2]) =

  P([4,3] | R.[3,3]) x **P**([3,3] | U.[3,2])

  + **P**([4,3] | R.[4,2]) x **P**([4,2] | U.[3,2])

Probability of Reaching the Goal



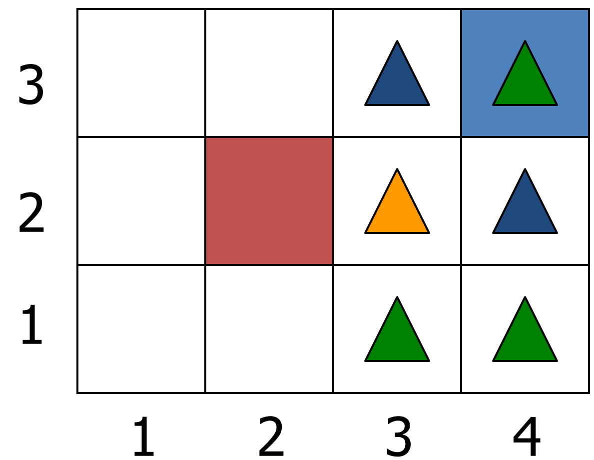
- **P**([4,3] | (U,R).[3,2]) =

  P([4,3] | R.[3,3]) x **P**([3,3] | U.[3,2])

  \+ **P**([4,3] | R.[4,2]) x **P**([4,2] | U.[3,2])

  - **P**([3,3] | U.[3,2]) = 0.8
  - **P**([4,2] | U.[3,2]) = 0.1

Probability of Reaching the Goal



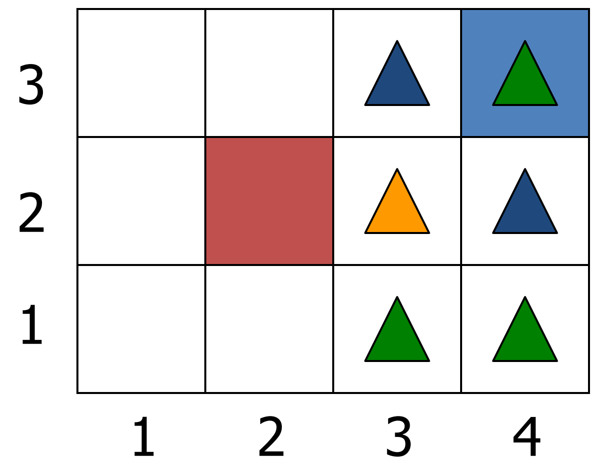
- **P**([4,3] | (U,R).[3,2]) =

  P([4,3] | R.[3,3]) x **P**([3,3] | U.[3,2])

  \+ **P**([4,3] | R.[4,2]) x **P**([4,2] | U.[3,2])

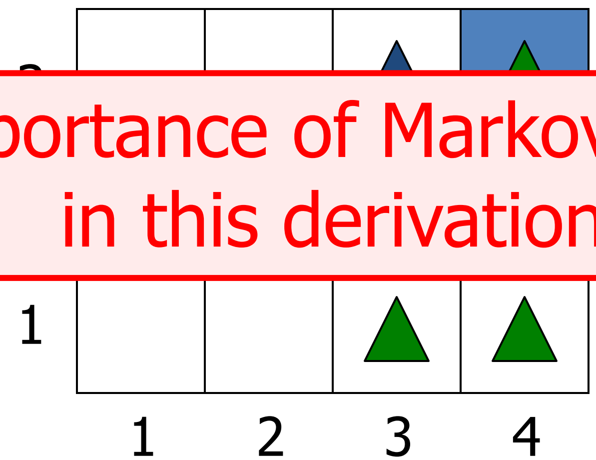
- **P**([4,3] | R.[3,3]) = 0.8
- **P**([4,3] | R.[4,2]) = 0.1
- **P**([3,3] | U.[3,2]) = 0.8
- **P**([4,2] | U.[3,2]) = 0.1

Probability of Reaching the Goal



- **P**([4,3] | (U,R).[3,2]) =
  P([4,3] | R.[3,3]) x **P**([3,3] | U.[3,2])
  \+ **P**([4,3] | R.[4,2]) x **P**([4,2] | U.[3,2])
- **P**([4,3] | R.[3,3]) = 0.8
- **P**([4,3] | R.[4,2]) = 0.1
- **P**([3,3] | U.[3,2]) = 0.8
- **P**([4,2] | U.[3,2]) = 0.1
- **P**([4,3] | (U,R).[3,2]) = 0.65

Probability of Reaching the Goal



Note importance of Markov property in this derivation

- **P**([4,3] | (U,R).[3,2]) =

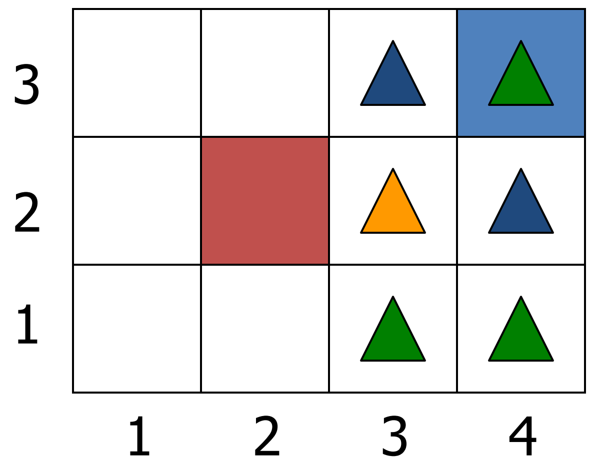
  P([4,3] | R.[3,3]) x **P**([3,3] | U.[3,2])

  \+ **P**([4,3] | R.[4,2]) x **P**([4,2] | U.[3,2])

- **P**([4,3] | R.[3,3]) = 0.8
- **P**([4,3] | R.[4,2]) = 0.1
- **P**([3,3] | U.[3,2]) = 0.8
- **P**([4,2] | U.[3,2]) = 0.1

- **P**([4,3] | (U,R).[3,2]) = 0.65

Probability of Reaching the Goal



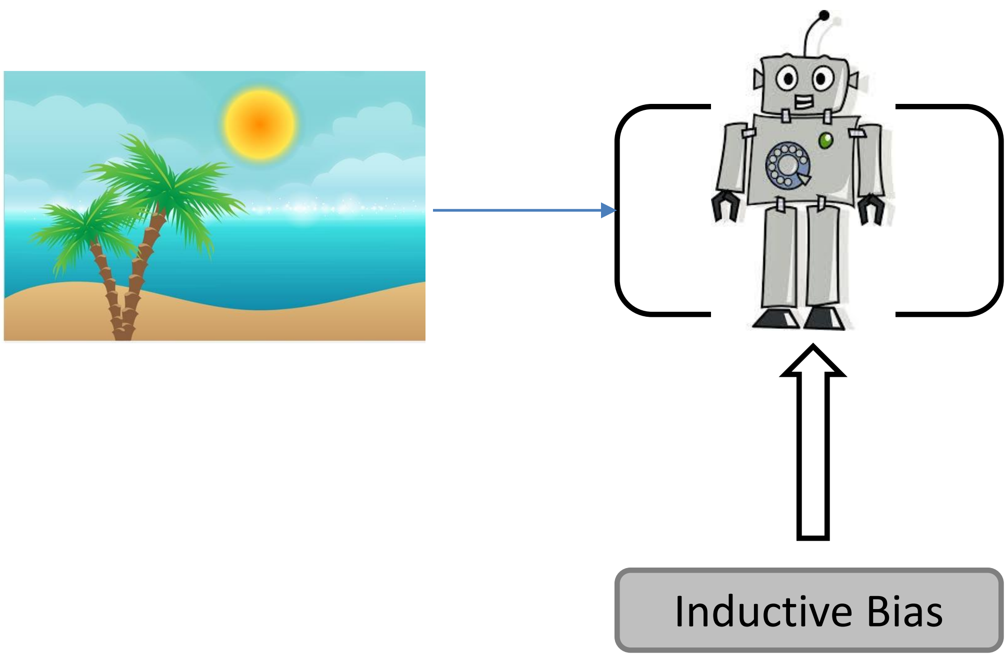
- Main idea: **multiply backward** probabilities of each step taken from end state reached (*because* of our Markov/independence assumptions)
- But we still need to consider different ways of reaching a state
  - Going all the way around the obstacle would be “worse”

But what about the **learning** part of reinforcement learning?

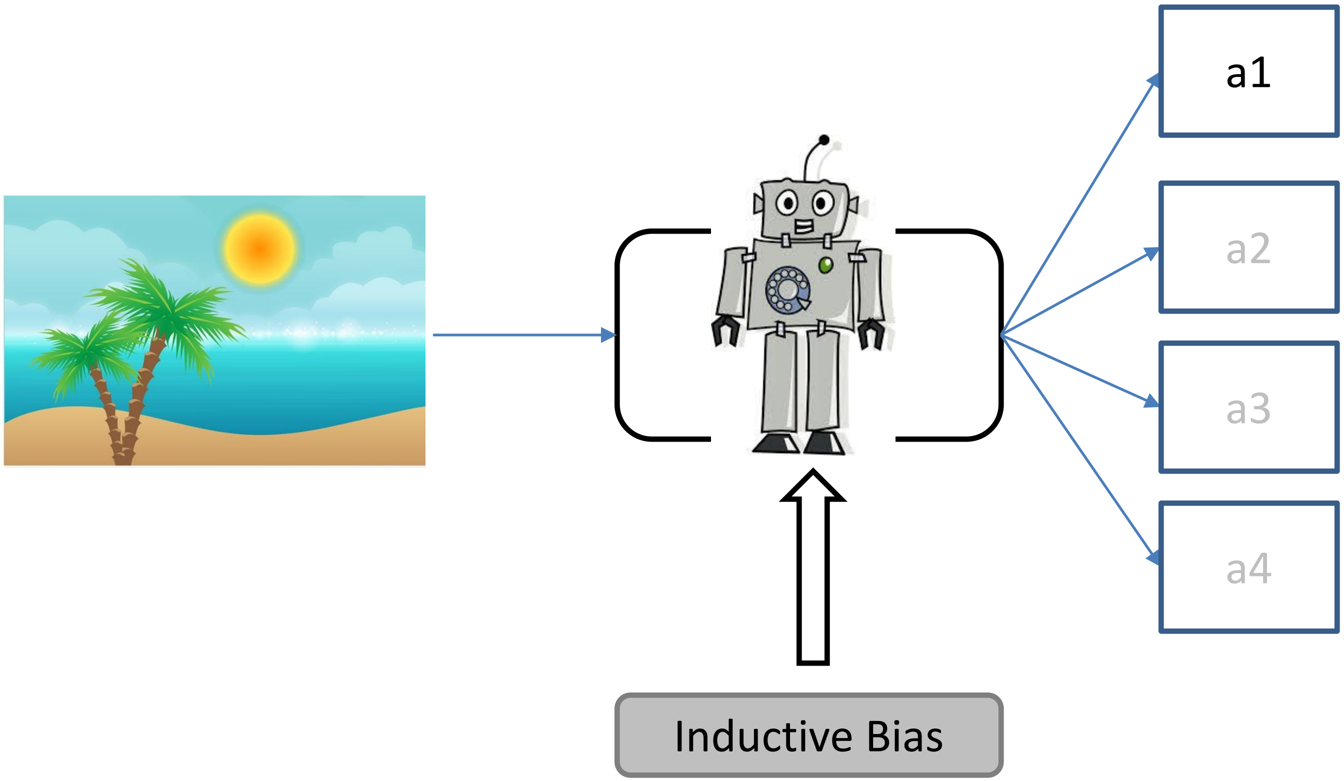
Review: What is ML?

- ML is a way to get a computer (in our parlance, a **system**) to do things without having to explicitly describe what steps to take.
- By giving it **examples** (training data)
- Or by giving it **feedback**
- It can then look for patterns which explain or predict what happens.
- The learned system of beliefs is called a **model**.

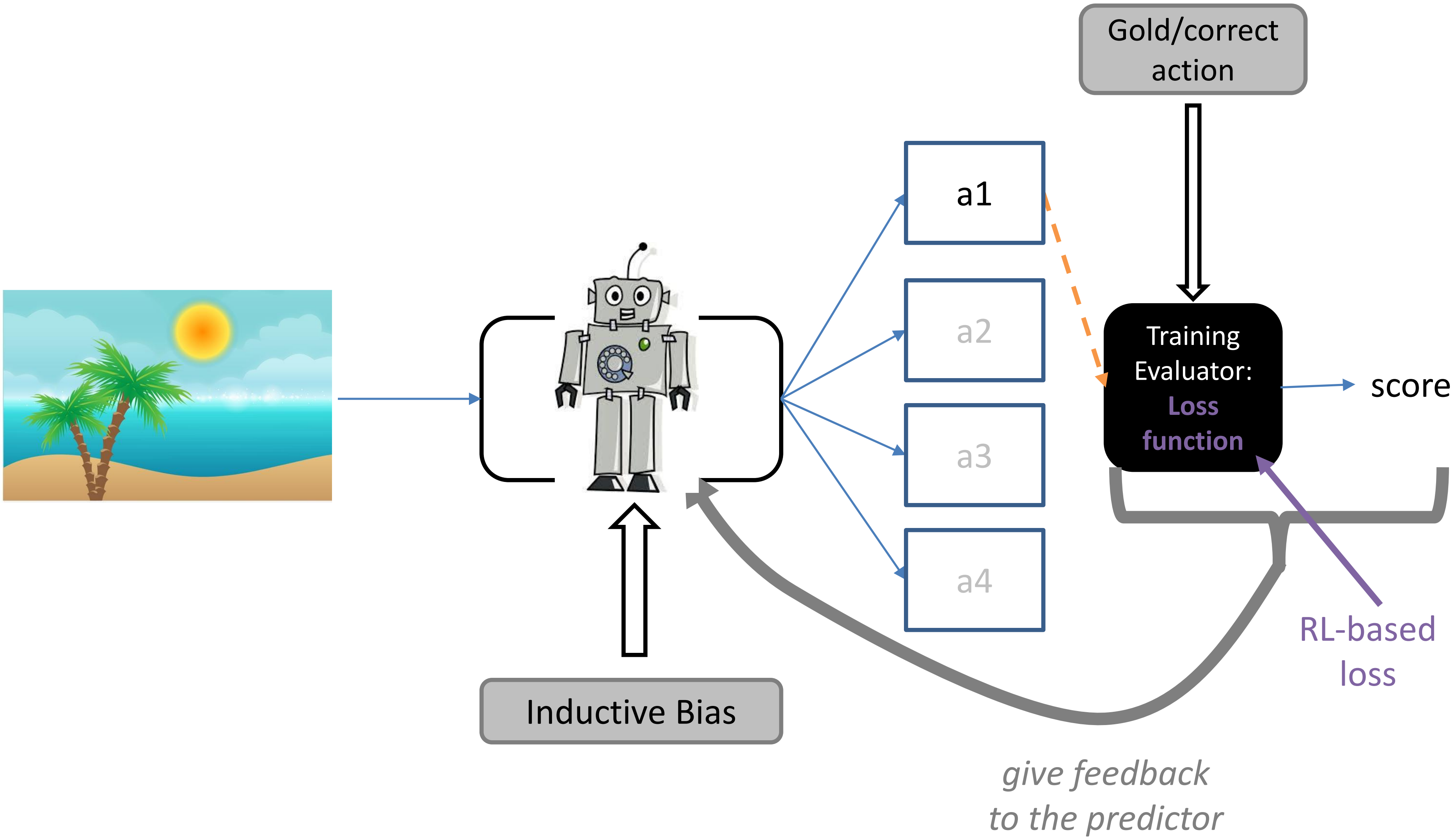
RL, in our ML framework



RL, in our ML framework

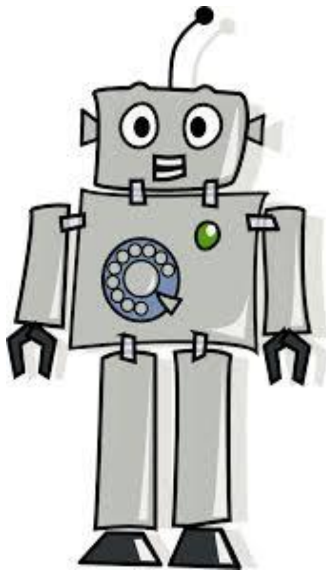


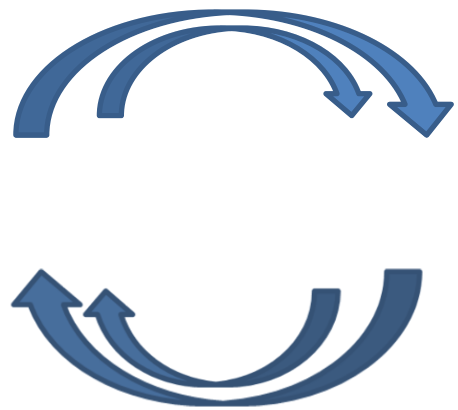
RL, in our ML framework

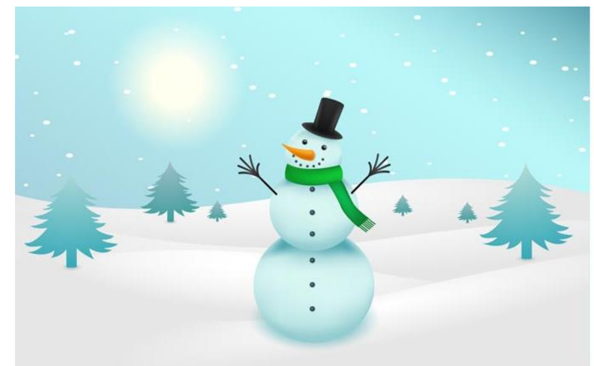


Markov Decision Process: Formalizing Reinforcement Learning

take action







get new state
and/or reward

environment

agent

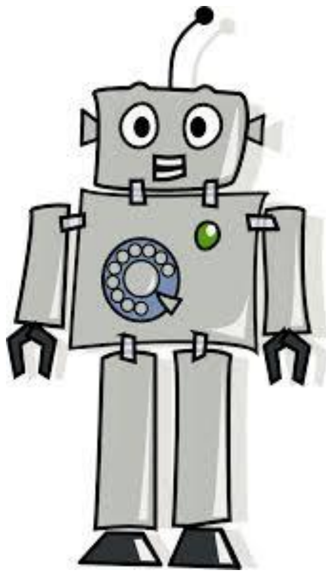
Markov Decision Process: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p, \gamma)$

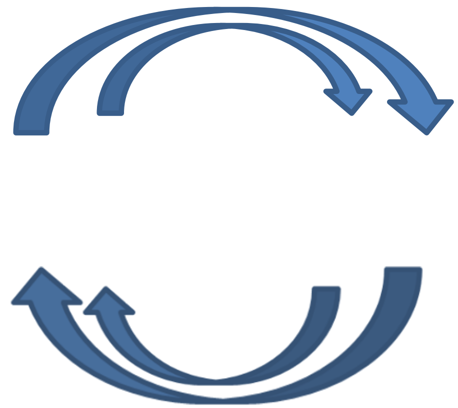
Robot image: openclipart.org

https://static.vecteezy.com/system/resources/previews/000/090/451/original/four-seasons-landscape-illustrations-vector.jpg

Markov Decision Process: Formalizing Reinforcement Learning

take action





get new state
and/or reward

environment

agent

set of
possible
actions

Markov Decision Process: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p, \gamma)$

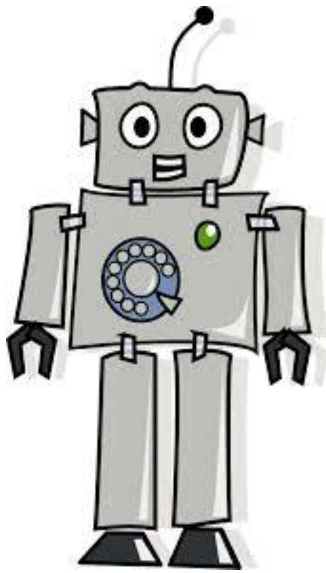
set of
possible
states

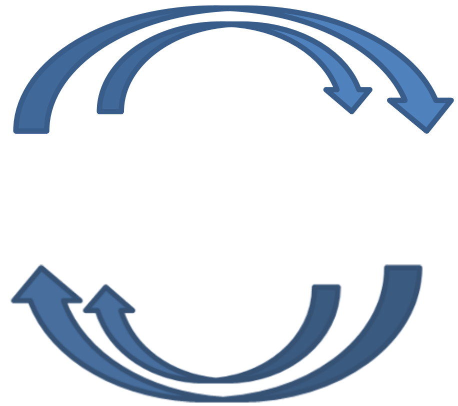
Robot image: openclipart.org

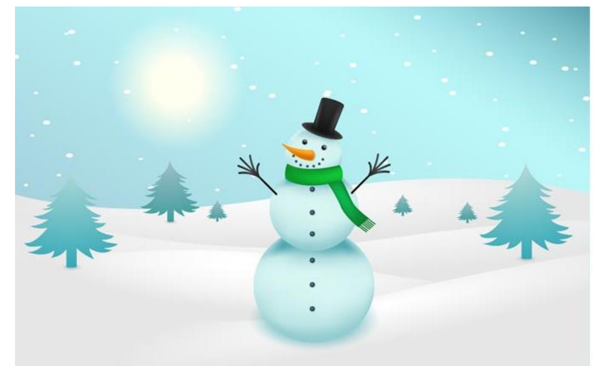
https://static.vecteezy.com/system/resources/previews/000/090/451/original/four-seasons-landscape-illustrations-vector.jpg

Markov Decision Process: Formalizing Reinforcement Learning

take action







get new state
and/or reward

environment

agent

Markov Decision Process:

$$(\mathcal{S}, \mathcal{A}, \mathcal{R}, p, \gamma)$$

set of possible states

set of possible actions

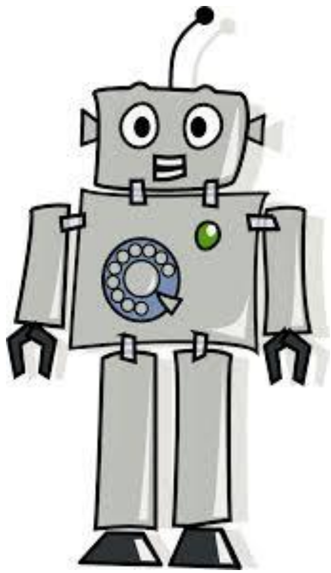
reward of (state, action) pairs

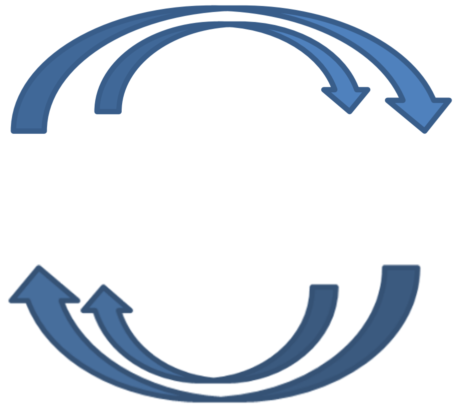
Robot image: openclipart.org

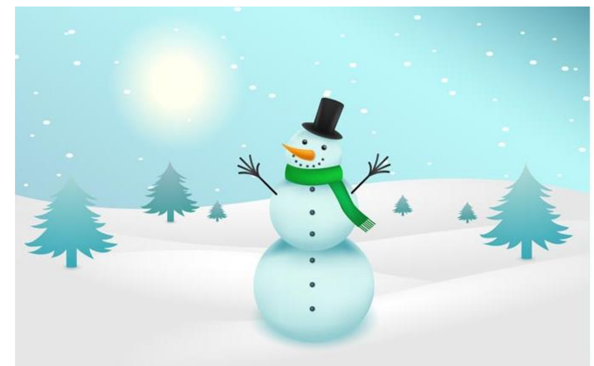
https://static.vecteezy.com/system/resources/previews/000/090/451/original/four-seasons-landscape-illustrations-vector.jpg

Markov Decision Process: Formalizing Reinforcement Learning

take action







get new state
and/or reward

environment

agent

Markov Decision Process:

$$(\mathcal{S}, \mathcal{A}, \mathcal{R}, p, \gamma)$$

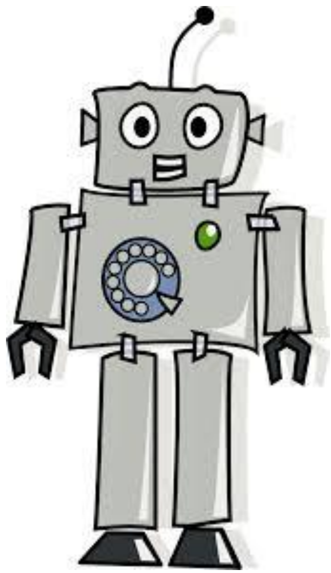
- $\mathcal{S}$: set of possible states
- $\mathcal{A}$: set of possible actions
- $\mathcal{R}$: reward of (state, action) pairs
- p: state-action transition distribution

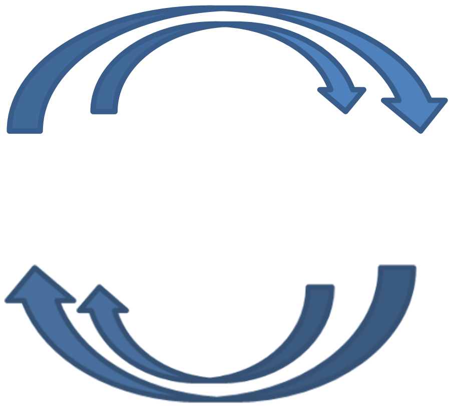
Robot image: openclipart.org

https://static.vecteezy.com/system/resources/previews/000/090/451/original/four-seasons-landscape-illustrations-vector.jpg

Markov Decision Process: Formalizing Reinforcement Learning

take action





get new state and/or reward

environment

agent

Markov Decision Process:

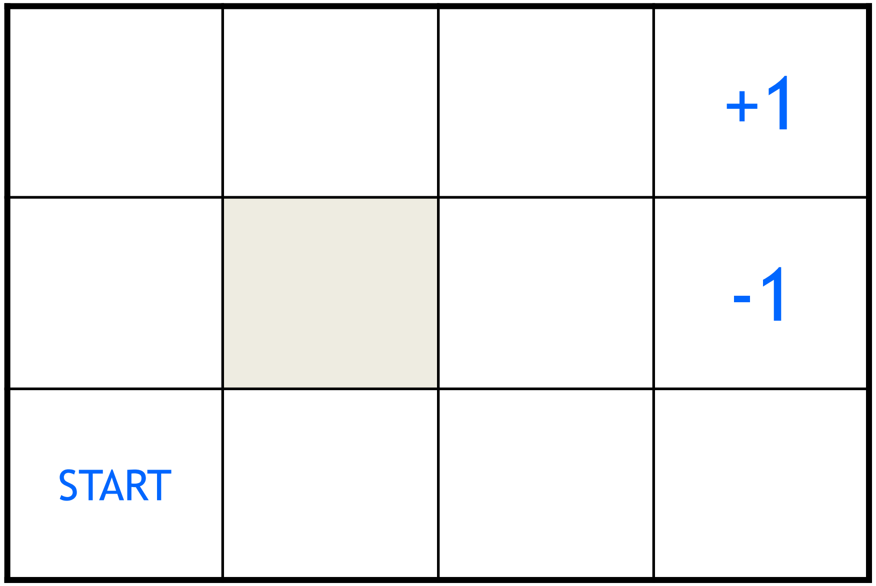
$$(\mathcal{S}, \mathcal{A}, \mathcal{R}, p, \gamma)$$

- $\mathcal{S}$: set of possible states
- $\mathcal{A}$: set of possible actions
- $\mathcal{R}$: reward of (state, action) pairs
- p: state-action transition distribution
- γ: discount factor

Robot image: openclipart.org

https://static.vecteezy.com/system/resources/previews/000/090/451/original/four-seasons-landscape-illustrations-vector.jpg

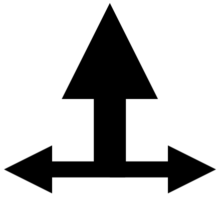
Robot in a room



actions: UP, DOWN, LEFT, RIGHT

UP

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

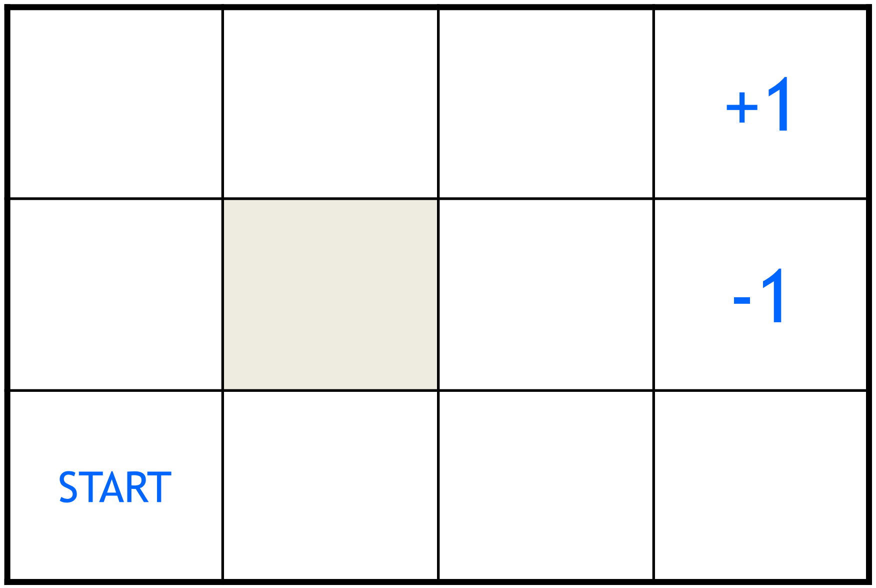


reward +1 at [4,3], -1 at [4,2]
reward -0.04 for each step

Goal: what's the strategy to achieve the maximum reward?

Slide courtesy Peter Bodík

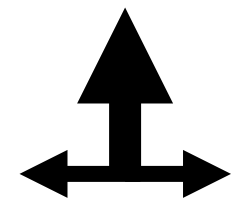
Robot in a room



actions: UP, DOWN, LEFT, RIGHT

UP

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



reward +1 at [4,3], -1 at [4,2]
reward -0.04 for each step

states: current location
actions: where to go next
rewards

what is the solution? Learn a mapping from (state, action) pairs to new states

Slide courtesy Peter Bodík

Markov Decision Process: Formalizing Reinforcement Learning

Markov Decision Process: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \pi, \gamma)$

set of possible actions

state-action transition distribution

set of possible states

reward of (state, action) pairs

discount factor

Start in initial state s_0

Markov Decision Process: Formalizing Reinforcement Learning

set of possible actions

state-action transition distribution

Markov Decision Process: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \pi, \gamma)$

set of possible states

reward of (state, action) pairs

discount factor

Start in initial state s_0
for t = 1 to …:
  choose action a_t

Markov Decision Process: Formalizing Reinforcement Learning

Markov Decision Process:

set of possible actions

state-action transition distribution

$$(\mathcal{S}, \mathcal{A}, \mathcal{R}, \pi, \gamma)$$

set of possible states

reward of (state, action) pairs

discount factor

Start in initial state s_0
for t = 1 to …:
  choose action a_t
  "move" to next state $s_t \sim \pi(\cdot | s_{t-1}, a_t)$

Markov Decision Process: Formalizing Reinforcement Learning

Markov Decision Process: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \pi, \gamma)$

- $\mathcal{S}$: set of possible states
- $\mathcal{A}$: set of possible actions
- $\mathcal{R}$: reward of (state, action) pairs
- π: state-action transition distribution
- γ: discount factor

Start in initial state s_0
for t = 1 to …:
  choose action a_t
  "move" to next state $s_t \sim \pi(\cdot|s_{t-1}, a_t)$
  get reward $r_t = \mathcal{R}(s_t, a_t)$

Markov Decision Process: Formalizing Reinforcement Learning

set of possible actions

state-action transition distribution

Markov Decision Process: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \pi, \gamma)$

set of possible states

reward of (state, action) pairs

discount factor

Start in initial state s_0
for t = 1 to …:
  choose action a_t
  “move” to next state $s_t \sim \pi(\cdot | s_{t-1}, a_t)$
  get reward $r_t = \mathcal{R}(s_t, a_t)$

objective: **maximize time-discounted reward**

Markov Decision Process: Formalizing Reinforcement Learning

set of possible actions

state-action transition distribution

Markov Decision Process: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \pi, \gamma)$

set of possible states

reward of (state, action) pairs

discount factor

Start in initial state s_0
for t = 1 to …:
  choose action a_t
  “move” to next state $s_t \sim \pi(\cdot | s_{t-1}, a_t)$
  get reward $r_t = \mathcal{R}(s_t, a_t)$

objective: maximize discounted reward

Consider all possible future times t

Reward at time t

Markov Decision Process: Formalizing Reinforcement Learning

Markov Decision Process: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \pi, \gamma)$

set of possible states

set of possible actions

reward of (state, action) pairs

state-action transition distribution

discount factor

Start in initial state s_0
for t = 1 to …:
  choose action a_t
  "move" to next state $s_t \sim \pi(\cdot | s_{t-1}, a_t)$
  get reward $r_t = \mathcal{R}(s_t, a_t)$

objective: maximize discounted reward

Consider all possible future times t

Discount at time t

Reward at time t

Markov Decision Process: Formalizing Reinforcement Learning

Markov Decision Process:

$$\left(\underbrace{\mathcal{S}}_{\text{set of possible states}}, \overbrace{\mathcal{A}}^{\text{set of possible actions}}, \underbrace{\mathcal{R}}_{\text{reward of (state, action) pairs}}, \overbrace{\pi}^{\text{state-action transition distribution}}, \underbrace{\gamma}_{\text{discount factor}}\right)$$

Start in initial state s_0
for t = 1 to ...:
  choose action a_t
  "move" to next state $s_t \sim \pi(\cdot | s_{t-1}, a_t)$
  get reward $r_t = \mathcal{R}(s_t, a_t)$

objective: maximize discounted reward

$$\max_{\pi} \sum_{t>0} \gamma^t r_t$$

Consider all possible future times t

Discount at time t

Reward at time t

Example of Discounted Reward

objective: maximize discounted reward

$$\max_{\pi} \sum_{t>0} \gamma^t r_t$$

Consider all possible future times t

Discount at time t

Reward at time t

- If the discount factor $\gamma = 0.8$ then reward

$$0.8^0 r_0 + 0.8^1 r_1 + 0.8^2 r_2 + 0.8^3 r_3 + \cdots + 0.8^n r_n + \ \ldots$$

- Allows you to consider all possible rewards in the future but preferring current vs. future self

Markov Decision Process: Formalizing Reinforcement Learning

Markov Decision Process: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \pi, \gamma)$

- $\mathcal{S}$: set of possible states
- $\mathcal{A}$: set of possible actions
- $\mathcal{R}$: reward of (state, action) pairs
- π: state-action transition distribution
- γ: discount factor

Start in initial state s_0
for t = 1 to …:
    choose action a_t
    "move" to next state $s_t \sim \pi(\cdot | s_{t-1}, a_t)$
    get reward $r_t = \mathcal{R}(s_t, a_t)$

objective: maximize discounted reward

$$\max_{\pi} \sum_{t>0} \gamma^t r_t$$

"solution": the policy π^* that maximizes the expected (average) time-discounted reward

Markov Decision Process: Formalizing Reinforcement Learning

Markov Decision Process: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \pi, \gamma)$

- $\mathcal{S}$: set of possible states
- $\mathcal{A}$: set of possible actions
- $\mathcal{R}$: reward of (state, action) pairs
- π: state-action transition distribution
- γ: discount factor

Start in initial state s_0
for t = 1 to …:
  choose action a_t
  "move" to next state $s_t \sim \pi(\cdot | s_{t-1}, a_t)$
  get reward $r_t = \mathcal{R}(s_t, a_t)$

objective: maximize discounted reward

$$\max_{\pi} \sum_{t>0} \gamma^t r_t$$

"solution" $$\pi^* = \operatorname*{argmax}_{\pi} \mathbb{E}\left[\sum_{t>0} \gamma^t r_t ; \pi\right]$$

Expected Value of a Random Variable

random variable

$$X \sim p(\cdot)$$

Expected Value of a Random Variable

random variable

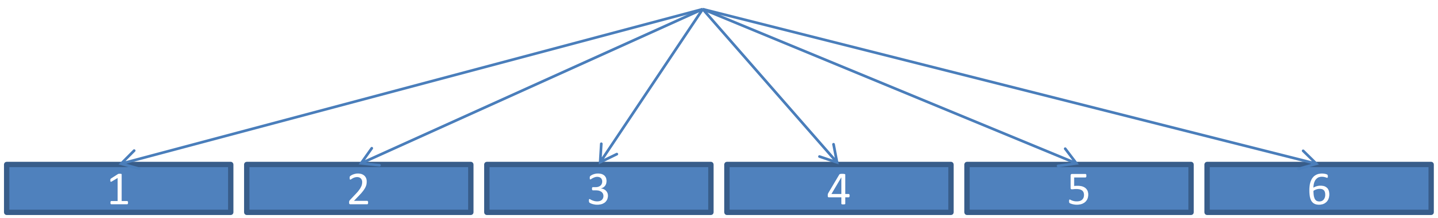
$$X \sim p(\cdot)$$

$$\mathbb{E}[X] = \sum_{x} x\, p(x)$$

expected value (distribution p is implicit)

Expected Value: Example

uniform distribution of number of cats I have



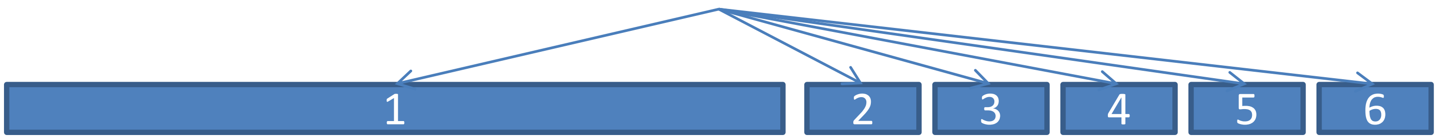
$$\mathbb{E}[X] = \sum_x x\, p(x)$$

1/6 * 1 +
1/6 * 2 +
1/6 * 3 +
1/6 * 4 +
1/6 * 5 +
1/6 * 6

= 3.5

Expected Value: Example 2

non-uniform distribution of number of cats a normal cat person has



$$\mathbb{E}[X] = \sum_x x\, p(x)$$

1/2 * 1 +
1/10 * 2 +
1/10 * 3 +
1/10 * 4 +
1/10 * 5 +
1/10 * 6

= 2.5

Expected Value of a Function of a Random Variable

$$X \sim p(\cdot)$$

$$\mathbb{E}[X] = \sum_{x} x\, p(x)$$

$$\mathbb{E}[f(X)] = ???$$

Expected Value of a Function of a Random Variable

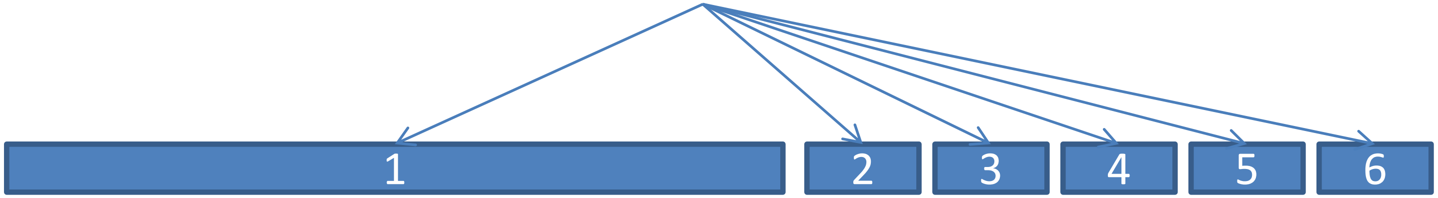
$$X \sim p(\cdot)$$

$$\mathbb{E}[X] = \sum_{x} x\, p(x)$$

$$\mathbb{E}[f(X)] = \sum_{x} f(x)\, p(x)$$

Expected Value of Function: Example

non-uniform distribution of number of cats I start with



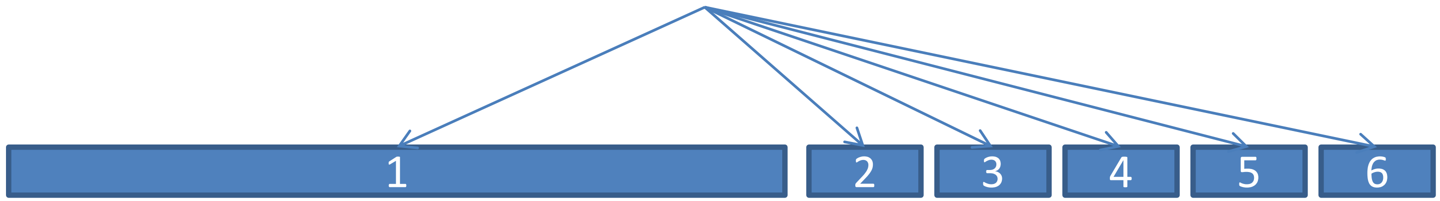
What if each cat magically becomes two?

$$f(k) = 2^k$$

$$\mathbb{E}[f(X)] = \sum_x f(x)\, p(x)$$

Expected Value of Function: Example

non-uniform distribution of number of cats I start with



What if each cat magically becomes two?

$$f(k) = 2^k$$

$$\mathbb{E}[f(X)] = \sum_x f(x)\, p(x) = \sum_x 2^x p(x)$$

$1/2 * 2^1 +$
$1/10 * 2^2 +$
$1/10 * 2^3 +$
$1/10 * 2^4 +$
$1/10 * 2^5 +$
$1/10 * 2^6$

$= 13.4$

Markov Decision Process: Formalizing Reinforcement Learning

Mar

Here, r_t is a function of random variable s_t.

Start in initia

for t = 1 to ..

choose act

"move" to next state $s_t \sim \pi(\cdot|s_{t-1}, a_t)$

get reward $r_t = \mathcal{R}(s_t, a_t)$

$$\max_{\pi} \sum_{t>0} \gamma^t r_t$$

"solution" $\pi^* = \operatorname{argmax}_{\pi} \mathbb{E}\left[\sum_{t>0} \gamma^t r_t ; \pi\right]$

Markov Decision Process: Formalizing Reinforcement Learning

Mar

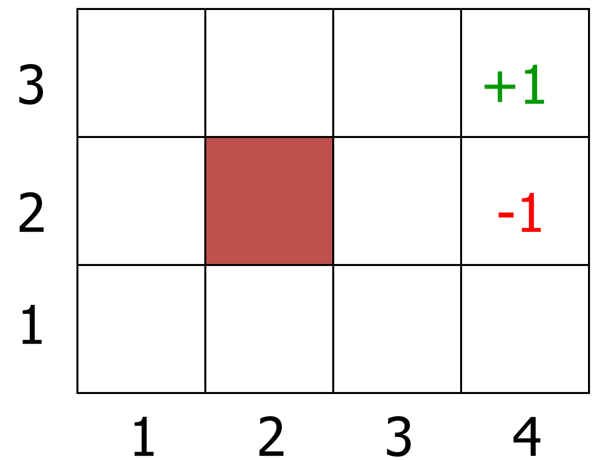
Here, r_t is a function of random variable s_t. ➔
The expectation is over the different states s_t the agent could be in at time t (equiv. actions the agent could take).

Start in initia
for t = 1 to ..
choose act
"move" to next state $s_t \sim \pi(\cdot|s_{t-1}, a_t)$
get reward $r_t = \mathcal{R}(s_t, a_t)$

$$\max_{\pi} \sum_{t>0} \gamma^t r_t$$

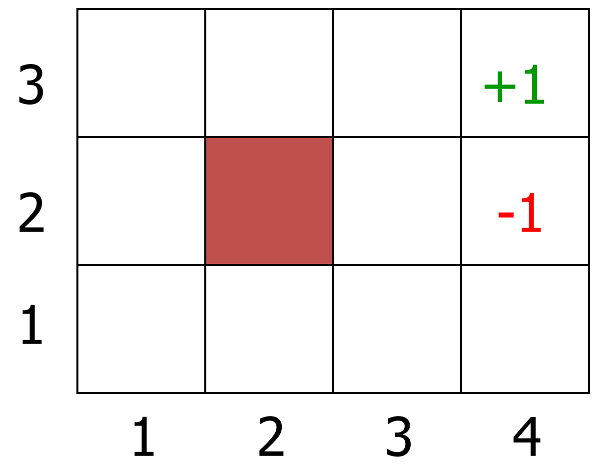
"solution" $\pi^* = \operatorname*{argmax}_{\pi} \mathbb{E}\left[\sum_{t>0} \gamma^t r_t \,; \pi\right]$

Utility Function



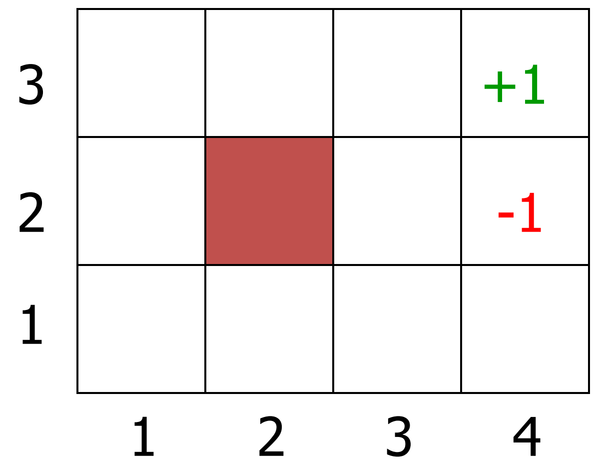
- [4,3] provides power supply
- [4,2] is a sand area from which the robot cannot escape

Utility Function



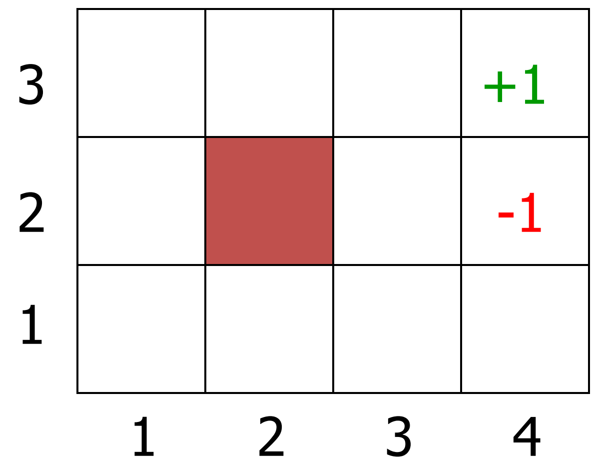
- [4,3] provides power supply
- [4,2] is a sand area from which the robot cannot escape
- The robot needs to recharge its batteries

Utility Function



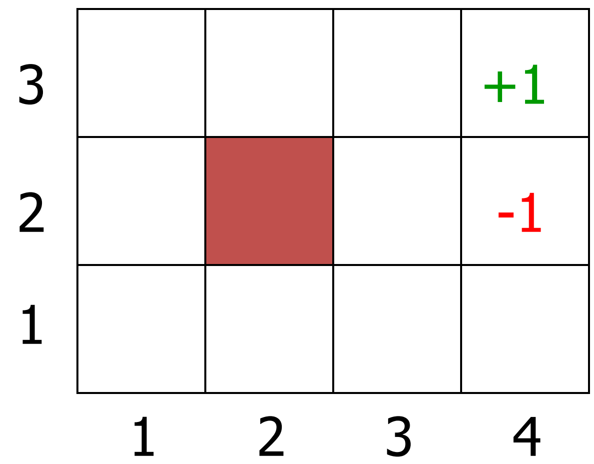
- [4,3] provides power supply
- [4,2] is a sand area from which the robot cannot escape
- The robot needs to recharge its batteries
- [4,3] and [4,2] are terminal states

Utility Function



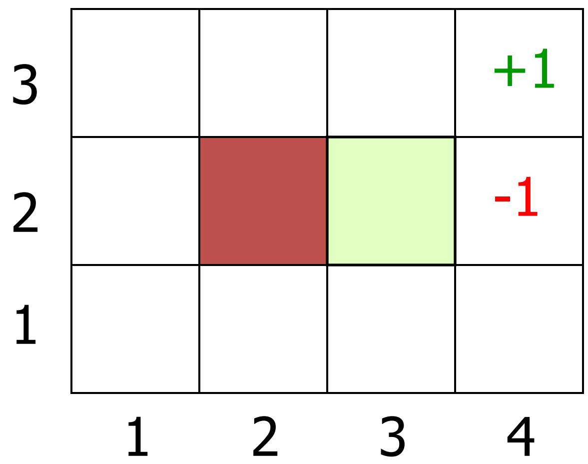
- [4,3] provides power supply
- [4,2] is a sand area from which the robot cannot escape
- The robot needs to recharge its batteries
- [4,3] and [4,2] are terminal states
- Histories have utility!

Utility of a History



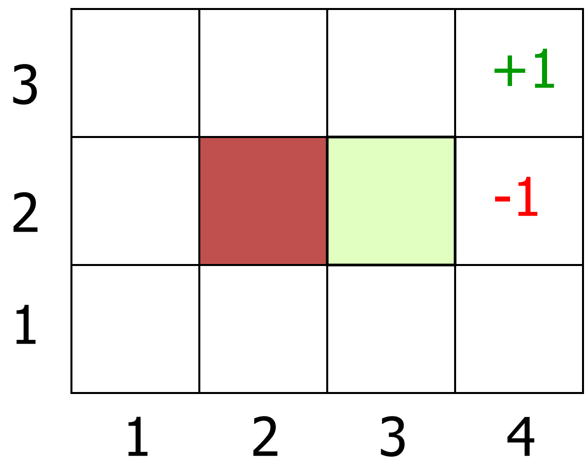
- [4,3] provides power supply
- [4,2] is a sand area from which the robot cannot escape
- The robot needs to recharge its batteries
- [4,3] or [4,2] are terminal states
- Histories have utility!
- The utility of a history is defined by the utility of the last state (+1 or –1) minus n/25, where n is the number of moves
  - Many utility functions possible, for many kinds of problems.

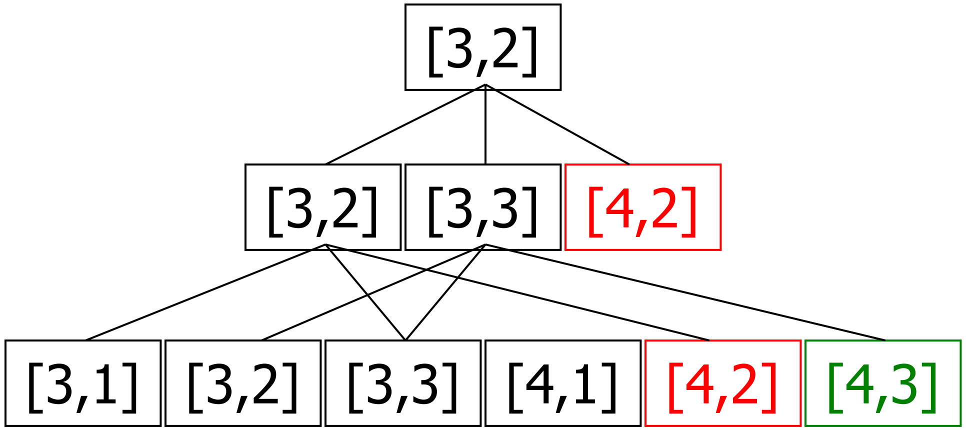
Utility of an Action Sequence



- Consider the action sequence (U,R) from [3,2]

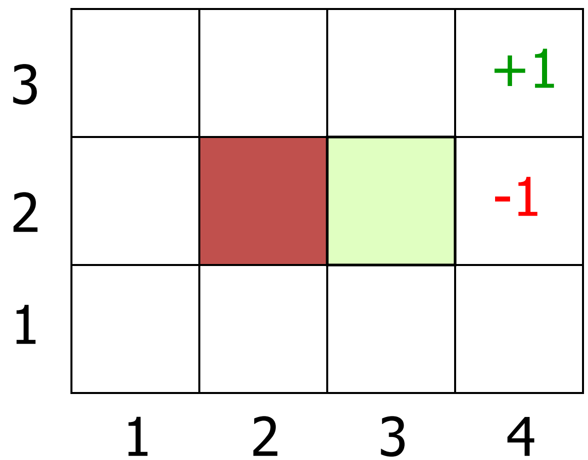
Utility of an Action Sequence

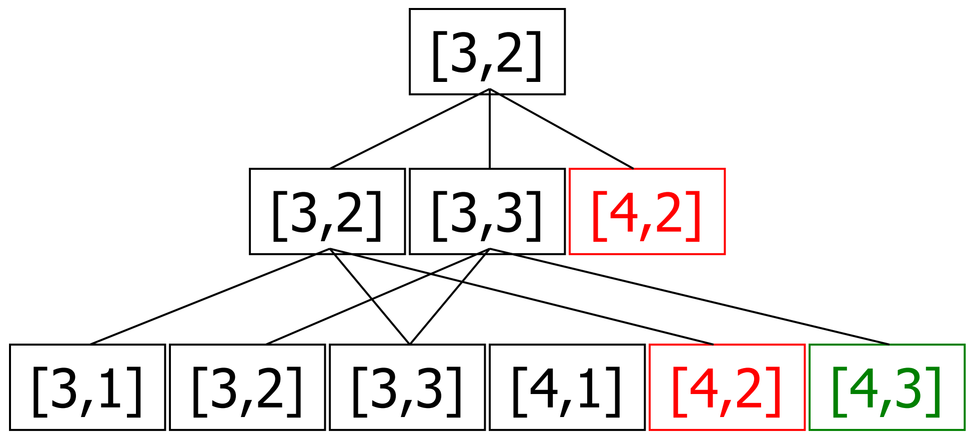




- Consider the action sequence (U,R) from [3,2]
- A run produces one of 7 possible histories, each with some probability

Utility of an Action Sequence

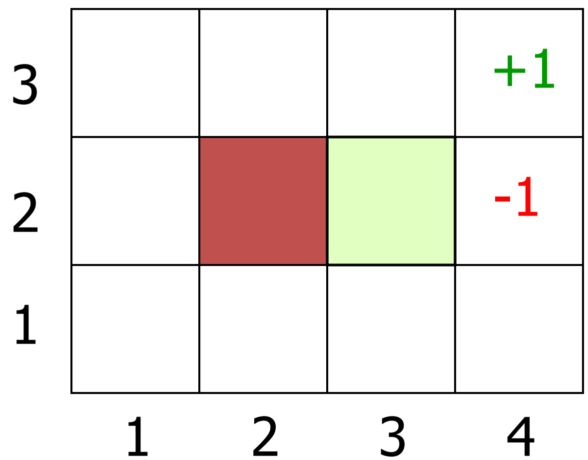


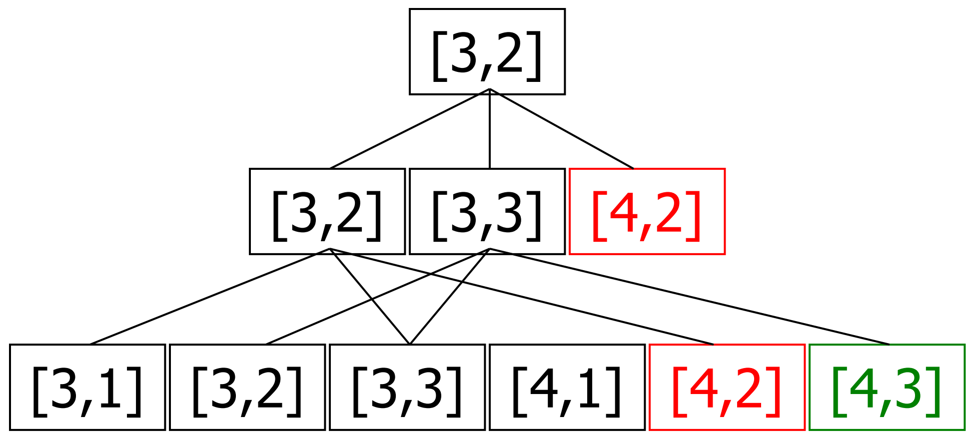


- Consider the action sequence (U,R) from [3,2]
- A run produces one of 7 possible histories, each with some probability
- The utility of the sequence is the expected utility of the histories:

$$\mathcal{U} = \Sigma_h \mathcal{U}_h \mathbf{P}(h)$$

Optimal Action Sequence



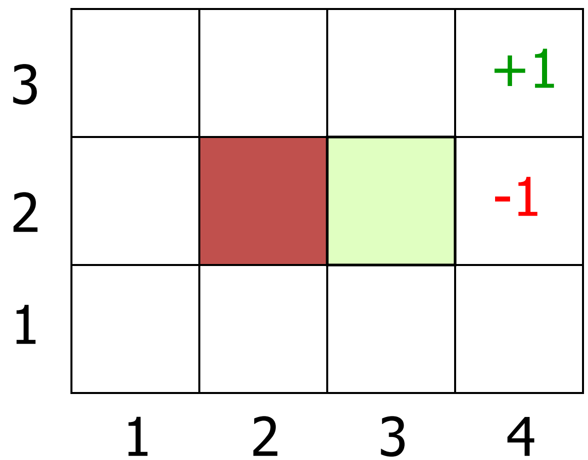


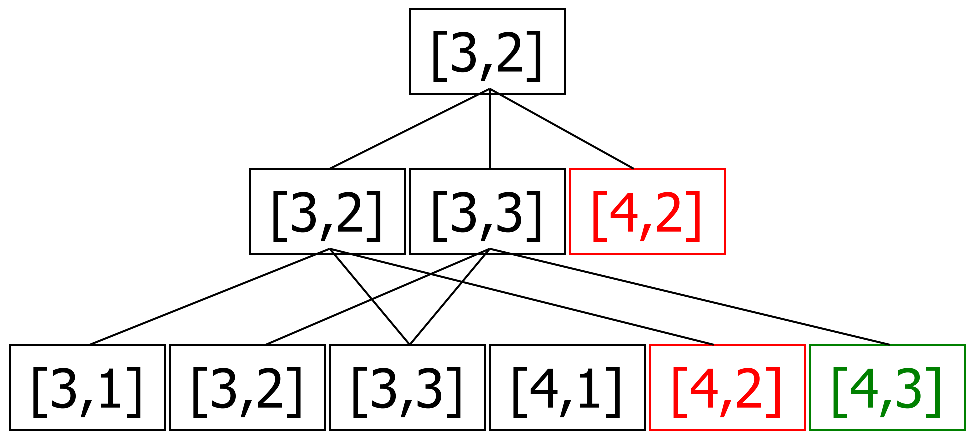
- Consider the action sequence (U,R) from [3,2]
- A run produces one of 7 possible histories, each with some probability
- The utility of the sequence is the expected utility of the histories:

$$\mathcal{U} = \Sigma_h \mathcal{U}_h \mathbf{P}(h)$$

- The optimal sequence is the one with maximal utility

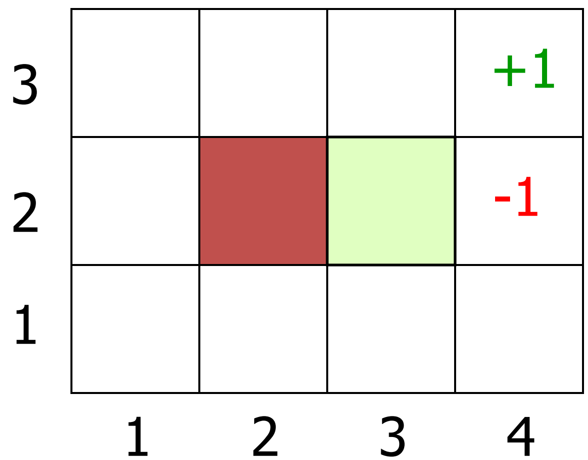
Optimal Action Sequence

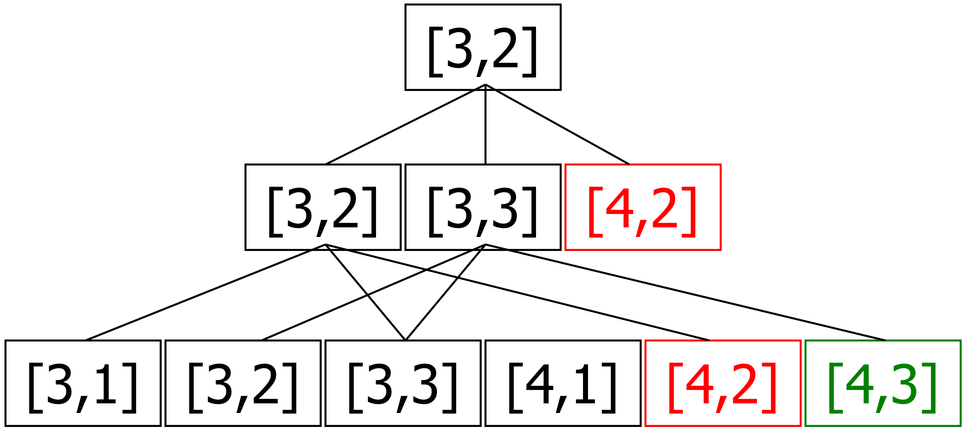




- Consider the action sequence (U,R) from [3,2]
- A run produces one of 7 possible histories, each with some probability
- The utility of the sequence is the expected utility of the histories
- The optimal sequence is the one with maximal utility
- **But is the optimal action sequence what we want to compute?**

Optimal Action Sequence





- Consider the action sequence (U,R) from [3,2]
- A run produc … ability
- The utility of the sequence is the expected utility of the histories
- The optimal sequence is the one with maximal utility
- **But is the optimal action sequence what we want to compute?**

only if the sequence is executed blindly!

Some Challenges

1. Representing states (and actions)

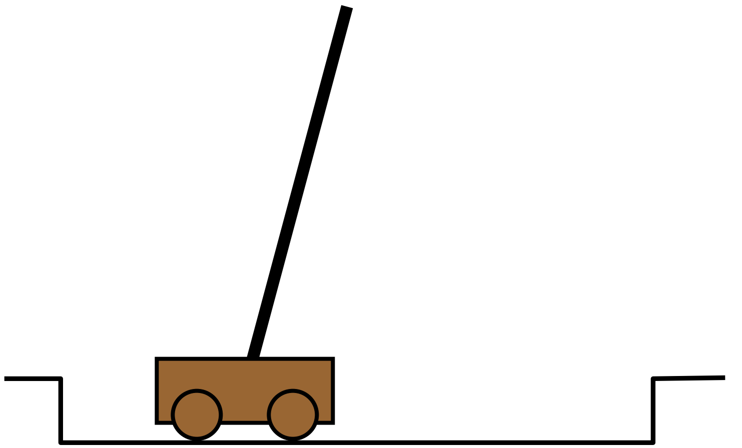
2. Defining our reward

3. Learning our policy

State Representation

Task: pole-balancing

state representation?



move car left/right to
keep the pole balanced

State Representation

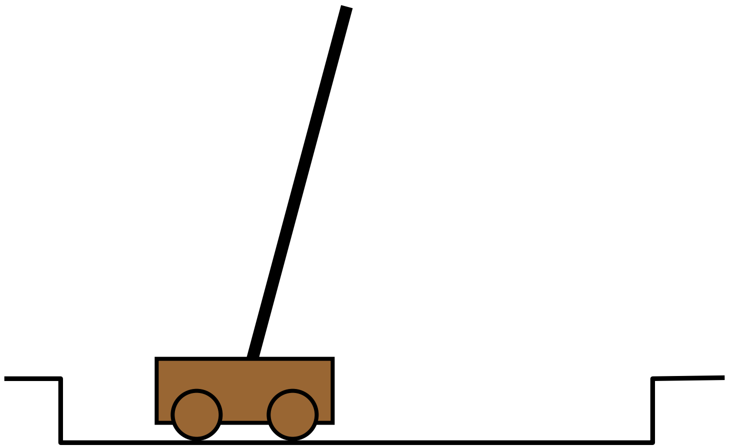
Task: pole-balancing

state representation

position and velocity of car

angle and angular velocity of pole

what about *Markov property*?



move car left/right to
keep the pole balanced

State Representation

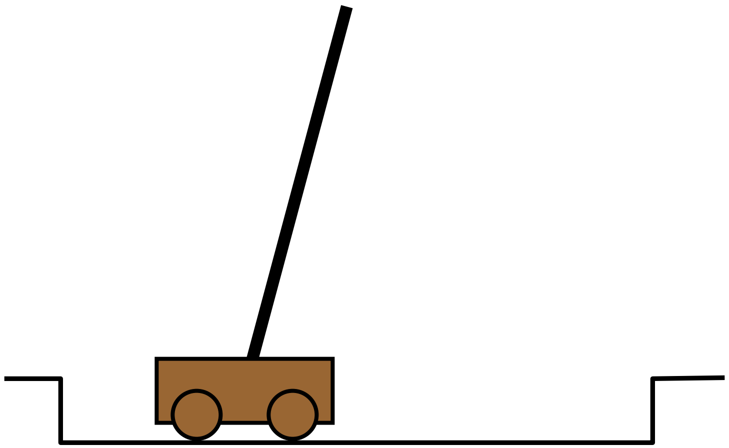
Task: pole-balancing

state representation

- position and velocity of car
- angle and angular velocity of pole

what about *Markov property*?

- would need more info
- noise in sensors, temperature, bending of pole



move car left/right to keep the pole balanced

Some Challenges

1. Representing states (and actions)

2. Defining our reward

3. Learning our policy

Designing Rewards

robot in a maze

- episodic task, not discounted, +1 when out, 0 for each step

chess

- GOOD: +1 for winning, -1 losing
- BAD: +0.25 for taking opponent's pieces
  - high reward even when lose

Slide courtesy/adapted: Peter Bodík

Designing Rewards

robot in a maze

- episodic task, not discounted, +1 when out, 0 for each step

chess

- GOOD: +1 for winning, -1 losing
- BAD: +0.25 for taking opponent's pieces
  - high reward even when lose

rewards

- rewards indicate what we want to accomplish
- NOT how we want to accomplish it

Designing Rewards

robot in a maze

episodic task, not discounted, +1 when out, 0 for each step

chess

GOOD: +1 for winning, -1 losing

BAD: +0.25 for taking opponent's pieces

high reward even when lose

rewards

rewards indicate what we want to accomplish

NOT how we want to accomplish it

shaping

positive reward often very "far away"

rewards for achieving subgoals (domain knowledge)

also: adjust initial policy or initial value function

Slide courtesy/adapted: Peter Bodík

Simple Reinforcement Learning

- Feedback is at the end, on a **series** of actions.
- Very early concept in Artificial Intelligence!
- Arthur Samuels' checker program was a simple reinforcement based learner, initially developed in 1956.
- In 1962 it beat a human checkers master.

www-03.ibm.com/ibm/history/ibm100/us/en/icons/ibm700series/impacts/

Example: Animals Game

- Human: I’m thinking of an animal.
- Computer: Is it a bird?
- Human: No.
- Computer: Is it a frog?
- Human: No.
- Computer: What is it?
- Human: A mouse.

Example: Animals Guessing Game Architecture

- All of the parts of ML Architecture:
  - The Representation is a sequence of questions and pairs of yes/no answers (decision tree).
  - The Actor "walks" the tree, interacting with a human; at each question it chooses whether to follow the "yes" branch or the "no" branch.
  - The Evaluator/loss function is the human player telling the game whether it has guessed correctly.
  - The Learner elicits new questions and adds questions, guesses and branches to the tree.

Reinforcement Learning (cont.)

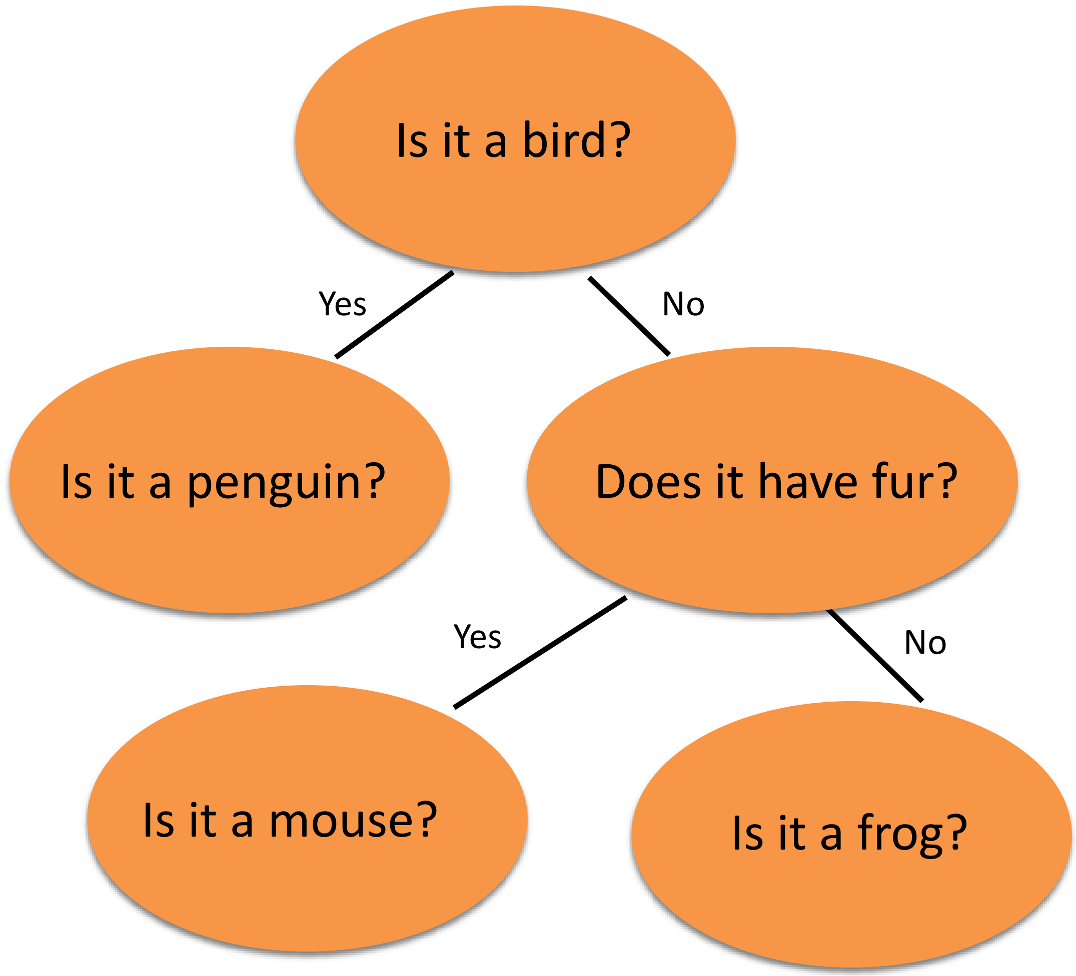
- Goal: agent acts in the world to maximize its rewards
- Agent has to figure out what it did that made it get that reward/punishment
  - This is known as the credit assignment problem

Example: Animals Game

- Human: I’m thinking of an animal.
- Computer: Is it a bird?
- Human: No.
- Computer: Is it a frog?
- Human: No.
- Computer: What is it?
- Human: A mouse.
- Computer: **Tell me a question that distinguishes a frog from a mouse.**
- Human: Does it have fur?
- Computer: What is the answer for a frog?
- Human: No.

Example: Animals Behind the Scene

Computer: Is it a bird?
Human: No
Computer: Is it a frog?
Human: No
Computer: What is it?
Human: A mouse
Computer: Tell me a question that distinguishes a frog from a mouse.
Human: Does it have fur?
Computer: What is the answer for a frog?
Human: no



After several rounds...

Reinforcement Learning (cont.)

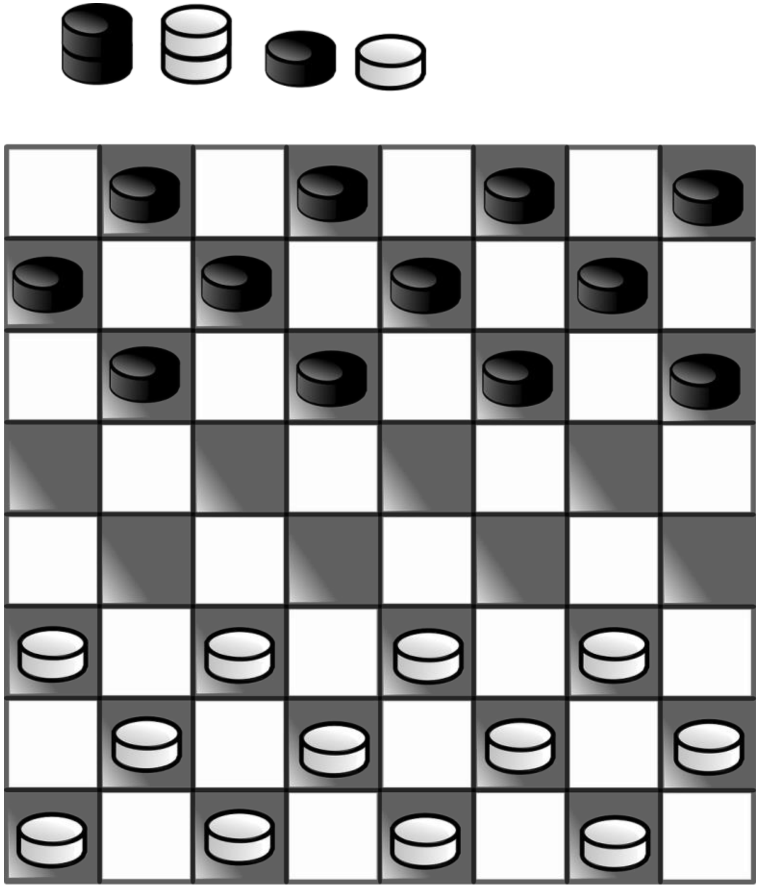
- Goal: agent acts in the world to maximize its rewards
- Agent has to figure out what it did that made it get that reward/punishment
  - This is known as the credit assignment problem
- RL can be used to train computers to do many tasks
  - Backgammon and chess playing
  - Job shop scheduling
  - Controlling robot limbs

Reactive Agent

- This kind of agent is a reactive agent
- The general algorithm for a reactive agent is:
  - Observe some state
  - If it is a terminal state, stop
  - Otherwise choose an action from the actions possible in that state
  - Perform the action
  - Recur.

Simple Example

- Learn to play checkers
  - Two-person game
  - 8x8 boards, 12 checkers/side
  - relatively simple set of rules: http://www.darkfish.com/checkers/rules.html
  - Goal is to eliminate all your opponent's pieces



Representing Checkers

- First we need to represent the game
- To completely describe one step in the game you need
  - A representation of the game board.
  - A representation of the current pieces
  - A variable which indicates whose turn it is
  - A variable which tells you which side is “black”
- There is no history needed
- A look at the current board setup gives you a complete picture of the state of the game

Representing Checkers

- Second, we need to represent the rules
- Represented as a **set of allowable moves** given board state
  - If a checker is at row x, column y, and row x+1 column y±1 is empty, it can move there.
  - If a checker is at (x,y), a checker of the opposite color is at (x+1, y+1), and (x+2,y+2) is empty, the checker must move there, and remove the "jumped" checker from play.
- There are additional rules, but all can be expressed in terms of the state of the board and the checkers.
- Each rule includes the outcome of the relevant action in terms of the state.
- What's a good reward?

A More Complex Example

- Consider an agent which must learn to drive a car
  - State?
  - Possible actions?
  - Rewards?

Some Challenges

1. Representing states (and actions)
2. Defining our reward
3. Learning our policy

What Do We Want to Learn

- Given
  - A description of some state of the game
  - A list of the moves allowed by the rules
  - **What move should we make?**
- Typically more than one move is possible
  - Need strategies, heuristics, or hints about what move to make
  - **This is what we are learning**
- We learn **from** whether the game was won or lost
  - Information to learn from is sometimes called "training signal"

Simple Checkers Learning

- Can represent some heuristics in the same formalism as the board and rules
  - If there is a legal move that will create a king, take it.
    - If checkers at (7,y) and (8,y-1) or (8,y+1) is free, move there.
  - If there are two legal moves, choose the one that moves a checker farther toward the top row
    - If checker(x,y) and checker(p,q) can both move, and x>p, move checker(x,y).
  - But then each of these heuristics needs some kind of priority or **weight.**

Formalization for RL Agent

- Given:
  - A state space S
  - A set of actions $a_1, \ldots, a_k$ including their results
  - A set of heuristics for resolving conflict among actions
  - Reward value at the end of each trial (series of action) (may be positive or negative)
- Output:
  - A policy (a mapping from states to preferred actions)

Learning Agent

- The general algorithm for this learning agent is:
  - Observe some state
  - If it is a terminal state
    - Stop
    - If won, **increase** the weight on **all** heuristics used
    - If lost, **decrease** the weight on **all** heuristics used
  - Otherwise choose an action from those possible in that state, using heuristics to select the preferred action
  - Perform the action

Policy

- A complete mapping from states to actions
  - There must be an action for each state
  - There may be more than one action
  - Not necessarily optimal
- The goal of a learning agent is to **tune** the policy so that the preferred action is optimal, or at least good.
  - analogous to training a classifier
- Checkers
  - Trained policy includes all legal actions, with **weights**
  - "Preferred" actions are **weighted up**

Approaches

- Learn policy directly: Discover function mapping from states to actions
  - Could be directly learned values
    - Ex: Value of state which removes last opponent checker is +1.
  - Or a heuristic function which has itself been trained
- Learn utility values for states (value function)
  - Estimate the value for each state
  - Checkers:
    - How happy am I with this state that turns a man into a king?

Value Function

- The agent knows what state it is in
- It has actions it can perform in each state
- Initially, don't know the value of any of the states
- If the outcome of performing an action at a state is deterministic, then the agent can update the utility value U() of states:
  - U(oldstate) = reward + U(newstate)
- The agent learns the utility values of states as it works its way through the state space

Learning States and Actions

- A typical approach is:
- At state S choose, some action A
- Taking us to new State S_1
  - If S_1 has a positive value: increase value of A at S.
  - If S_1has a negative value: decrease value of A at S.
  - If S_1 is new, initial value is unknown: value of A unchanged.
- One complete learning pass or **trial** eventually gets to a terminal, deterministic state. (E.g., “win” or “lose”)
- Repeat until? Convergence? Some performance level?

Selecting an Action

- Simply choose action with highest (current) expected utility?
- Problem: each action has two effects
  - Yields a **reward** on current sequence
  - Gives **information** for learning future sequences
- Trade-off: immediate good for long-term well-being
  - Like trying a shortcut: might get lost, might find quicker path

Exploration vs. Exploitation

- Problem with naïve reinforcement learning:
  - What action to take?
  - **Best apparent action, based on learning to date** } Exploitation
    - Greedy strategy
    - Often prematurely converges to a suboptimal policy!
  - **Random (or unknown) action** } Exploration
    - Will cover entire state space
    - Very expensive and slow to learn!
    - When to stop being random?
  - Balance exploration (try random actions) with exploitation (use best action so far)

More on Exploration

- Agent may sometimes choose to explore suboptimal moves in hopes of finding better outcomes
  - Only by visiting all states frequently enough can we guarantee learning the true values of all the states
- When the agent is **learning**, ideal would be to get accurate values for all states
  - Even though that may mean getting a negative outcome
- When agent is **performing**, ideal would be to get optimal outcome
- A learning agent should have an **exploration policy**

Exploration Policy

- Wacky approach (exploration): act randomly in hopes of eventually exploring entire environment
  - Choose any legal checkers move
- Greedy approach (exploitation): act to maximize utility using current estimate
  - Choose moves that have in the past led to wins
- Reasonable balance: act more wacky (exploratory) when agent has little idea of environment; more greedy when the model is close to correct
  - Suppose you know no checkers strategy?
  - What's the best way to get better?

Example: N-Armed Bandits

- A row of slot machines
- Which to play and how often?
- State Space is a set of machines
  - Each has cost, payout, and percentage values
- Action is pull a lever.
- Each action has a positive or negative result
  - …which then adjusts the utility of that action (pulling that lever)

¢25	¢95	$10
$100	$200	$900
0.1%	0.6%	10%

N-Armed Bandits Example

- Each action initialized to a standard payout
- Result is either some cash (a win) or none (a lose)
- **Exploration:** Try things until we have estimates for payouts
- **Exploitation:** When we have some idea of the value of each action, choose the best.
- Clearly this is a heuristic.
- No proof we ever find the best lever to pull!
  - The more exploration we can do the better our model
  - But the higher the cost over multiple trials

Overview: Learning Strategies

Dynamic Programming

Q-learning

Monte Carlo approaches

Dynamic programming

use value functions to structure the search for good policies

Dynamic programming

use value functions to structure the search for good policies

policy evaluation: compute V^π from π

policy improvement: improve π based on V^π

Dynamic programming

use value functions to structure the search for good policies

policy evaluation: compute V^{π} from π

policy improvement: improve π based on V^{π}

start with an arbitrary policy

repeat evaluation/improvement until convergence

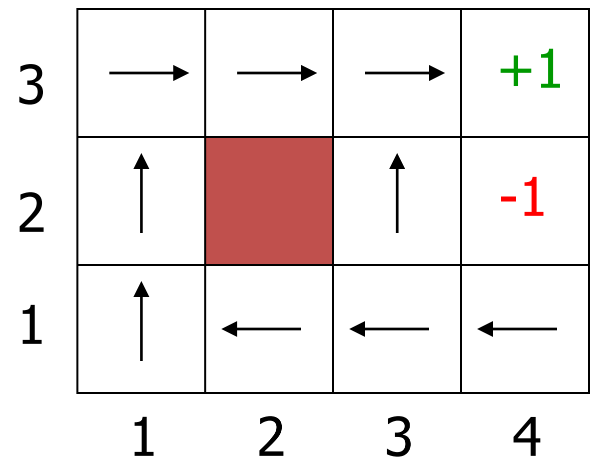
Reactive Agent Algorithm

Repeat:

Accessible or observable state

- s ← sensed state
- If s is a terminal state then exit
- a ← choose action (given s)
- Perform a

Policy (Reactive/Closed-Loop Strategy)



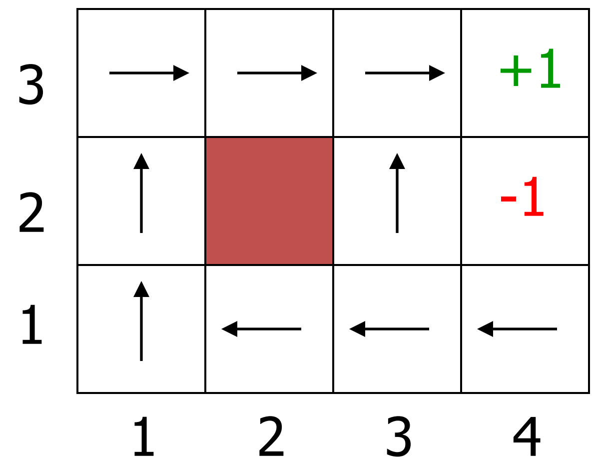
- In every state, we need to know what to do
- The **goal** doesn't change
- A policy (Π) is a complete mapping from *states* to *actions*
  - "If in [3,2], go up; if in [3,1], go left; if in…"

Reactive Agent Algorithm

Repeat:

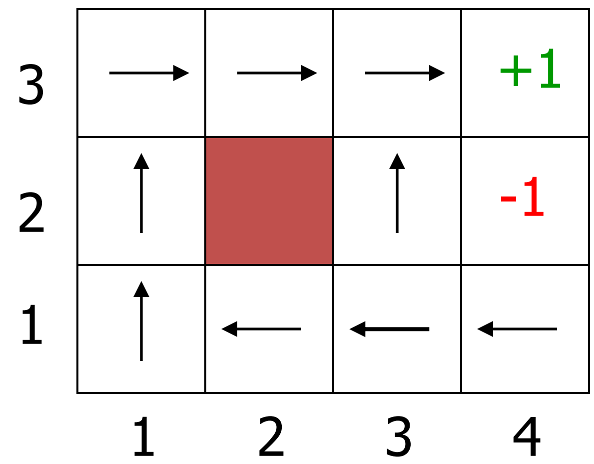
- $s \leftarrow$ sensed state
- If s is terminal then exit
- $a \leftarrow \Pi(s)$
- Perform a

Optimal Policy



- A policy Π is a complete mapping from states to actions
- The optimal policy Π* is the one that always yields a history (sequence of steps ending at a terminal state) with maximal ***expected*** utility

Optimal Policy



- A policy Π is a comp ns
- The optimal policy Π history with maximal expected utility

This problem is called a Markov Decision Problem (MDP)

How to compute Π*?

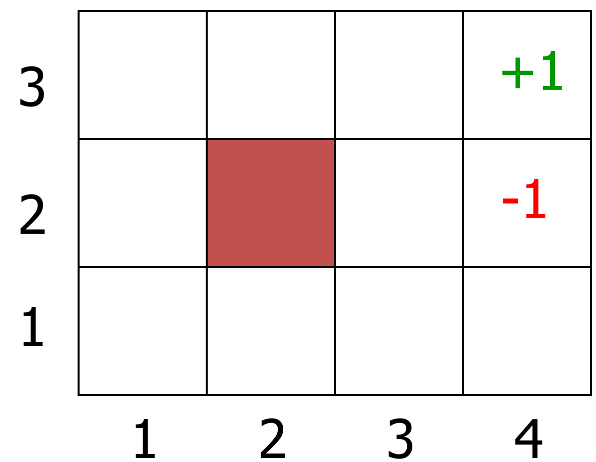
Defining State Utility

- Problem:
  - When making a decision, we only know the reward so far, and the possible actions
  - We've defined utility retroactively (i.e., the utility of a history is known *once we finish it*)
  - What is the utility of a particular ***state*** in the middle of decision making?
  - Need to compute ***expected utility*** of possible future histories

Value Iteration

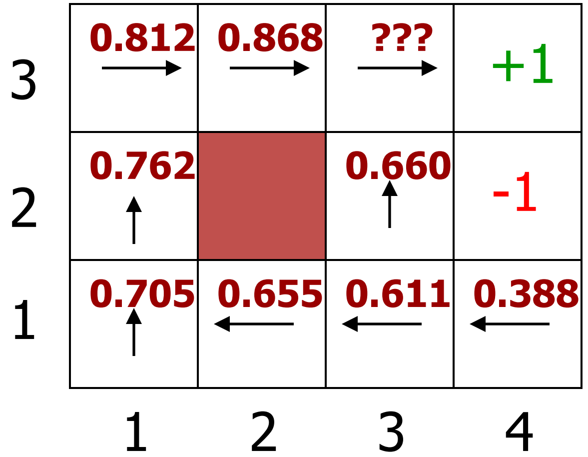
- Initialize the utility of each non-terminal state
  s_i to $\mathcal{U}_0(i) = 0$ } *or some uniform or uniformly distributed value*
- For t = 0, 1, 2, …, do:

$$\mathcal{U}_{t+1}(i) \leftarrow \mathcal{R}(i) + \max_a \textstyle\sum_k \mathbf{P}(k \mid a.i)\, \mathcal{U}_t(k)$$



Value Iteration

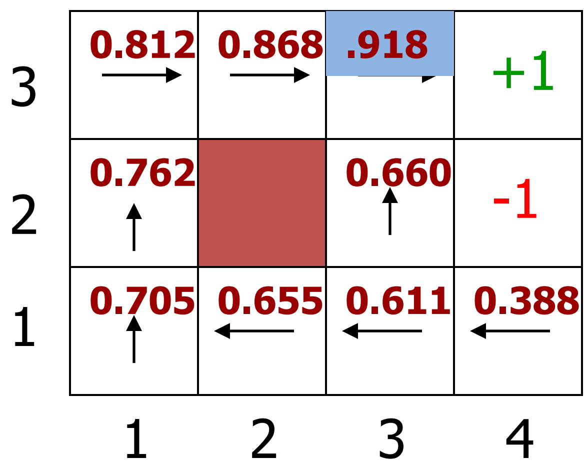
- Initialize the utility of each non-terminal state
  s_i to $\mathcal{U}_0(i) = 0$
- For t = 0, 1, 2, …, do:
  $$\mathcal{U}_{t+1}(i) \leftarrow \mathcal{R}(i) + \max_a \sum_k \mathbf{P}(k \mid a.i)\, \mathcal{U}_t(k)$$



EXERCISE: What is U*([3,3]) (assuming that the other U* are as shown)?

Value Iteration

- Initialize the utility of each non-terminal state
  s_i to $\mathcal{U}_0(i) = 0$
- For t = 0, 1, 2, …, do:
  $$\mathcal{U}_{t+1}(i) \leftarrow \mathcal{R}(i) + \max_a \sum_k \mathbf{P}(k \mid a.i)\, \mathcal{U}_t(k)$$



$U^*_{3,3} =$
$R_{3,3} +$
$[P_{3,2}\, U^*_{3,2} + P_{3,3}\, U^*_{3,3} + P_{4,3}\, U^*_{4,3}]$

Policy Iteration

- Pick a policy Π at random

Policy Iteration

- Pick a policy Π at random
- Repeat:
  - Compute the utility of each state for Π

    $\mathcal{U}_{\mathbf{t+1}}(i) \leftarrow \mathcal{R}(i) + \sum_k \mathbf{P}(k \mid \Pi(i).i)\ \mathcal{U}_{\mathbf{t}}(k)$

Policy Iteration

- Pick a policy Π at random
- Repeat:
  - Compute the utility of each state for Π

    $\mathcal{U}_{\mathbf{t+1}}(i) \leftarrow \mathcal{R}(i) + \sum_k \mathbf{P}(k \mid \Pi(i).i)\ \mathcal{U}_{\mathbf{t}}(k)$
  - Compute the policy Π' given these utilities

    $\Pi'(i) = \arg\max_a \sum_k \mathbf{P}(k \mid a.i)\ \mathcal{U}(k)$

Policy Iteration

- Pick a policy Π at random
- Repeat:
  - Compute the utility of each state for Π

    $\mathcal{U}_{\mathbf{t+1}}(\text{i}) \leftarrow \mathcal{R}(\text{i}) + \sum_{\text{k}} \mathbf{P}(\text{k} \mid \Pi(\text{i}).\text{i})\ \mathcal{U}_{\mathbf{t}}(\text{k})$
  - Compute the policy Π' given these utilities

    $\Pi'(\text{i}) = \arg\max_{\text{a}} \sum_{\text{k}} \mathbf{P}(\text{k} \mid \text{a.i})\ \mathcal{U}(\text{k})$
  - If $\Pi' = \Pi$ then return Π

Policy Iteration

- Pick a policy Π at random
- Repeat:
  - Compute the utility of each state for Π

    $\mathcal{U}_{\mathbf{t+1}}(i) \leftarrow \mathcal{R}(i) + \sum_k \mathbf{P}(k \mid \Pi(i).i)\, \mathcal{U}_{\mathbf{t}}(k)$
  - Compute the policy Π' given these utilities

    $\Pi'(i) = \arg\max_a \sum_k \mathbf{P}(k \mid a.i)\, \mathcal{U}(k)$
  - If $\Pi' = \Pi$ then return Π

Or solve the set of linear equations:

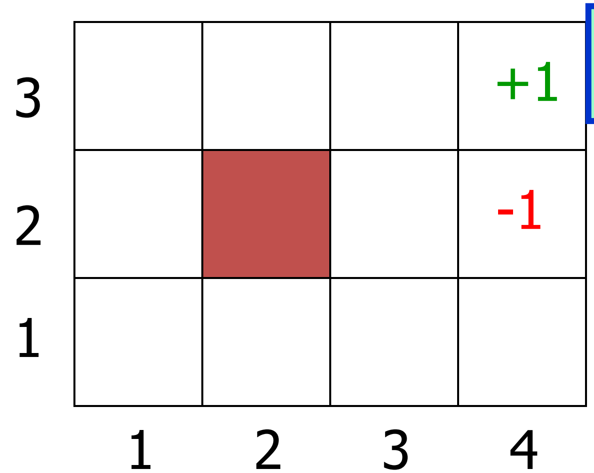
$\mathcal{U}(i) = \mathcal{R}(i) + \sum_k \mathbf{P}(k \mid \Pi(i).i)\, \mathcal{U}(k)$

(often a sparse system)

Advanced topic

Infinite Horizon

In many problems, e.g., the robot navigation example, histories are potentially unbounded and the same state can be reached many times



What if the robot lives forever?

One trick:
Use discounting to make an infinite horizon problem mathematically tractable

Value Iteration: Summary

- Initialize state values (expected utilities) randomly
- Repeatedly update state values using best action, according to current approximation of state values
- Terminate when state values stabilize
- Resulting policy will be the best policy because it's based on accurate state value estimation

Policy Iteration: Summary

– Initialize policy randomly
– Repeatedly update state values using best action, according to current approximation of state values
– Then update policy based on new state values
– Terminate when policy stabilizes
– Resulting policy is the best policy, but state values may not be accurate (may not have converged yet)
– Policy iteration is often faster (because we don't have to get the state values right)

- **Both methods have a major weakness: They require us to know the transition function exactly in advance!**

Overview: Learning Strategies

Dynamic Programming

Q-learning

Monte Carlo approaches

Q-learning

$$Q: (s, a) \rightarrow \mathbb{R}$$

Goal: learn a function that computes a "goodness" score for taking a particular action a in state s

Q-learning

previous algorithms: on-policy algorithms

- start with a random policy, iteratively improve
- converge to optimal

Q-learning: off-policy

- use any policy to estimate Q

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

- Q directly approximates Q* (Bellman optimality equation)
- independent of the policy being followed
- only requirement: keep updating each (s,a) pair

Slide courtesy/adapted: Peter Bodík

Q-learning

previous algorithms: on-policy algorithms

- start with a random policy, iteratively improve
- converge to optimal

Q-learning: off-policy

- use any policy to estimate Q

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

- Q directly approximates Q* (Bellman optimality equation)
- independent of the policy being followed
- only requirement: keep updating each (s,a) pair

Deep/Neural Q-learning

$$Q(s, a; \theta) \approx Q^*(s, a)$$

neural network desired optimal solution

Deep/Neural Q-learning

$$Q(s, a; \theta) \approx Q^*(s, a)$$

neural network desired optimal solution

Approach: Form (and learn) a neural network to model our optimal Q function

Deep/Neural Q-learning

Learn weights (parameters) θ of our neural network

$$\underbrace{Q(s,a;\theta)}_{\text{neural network}} \approx \underbrace{Q^*(s,a)}_{\text{desired optimal solution}}$$

Approach: Form (and learn) a neural network to model our optimal Q function

Overview: Learning Strategies

Dynamic Programming

Q-learning

Monte Carlo approaches

Monte Carlo policy evaluation

don't need full
knowledge of
environment (just
(simulated) experience)

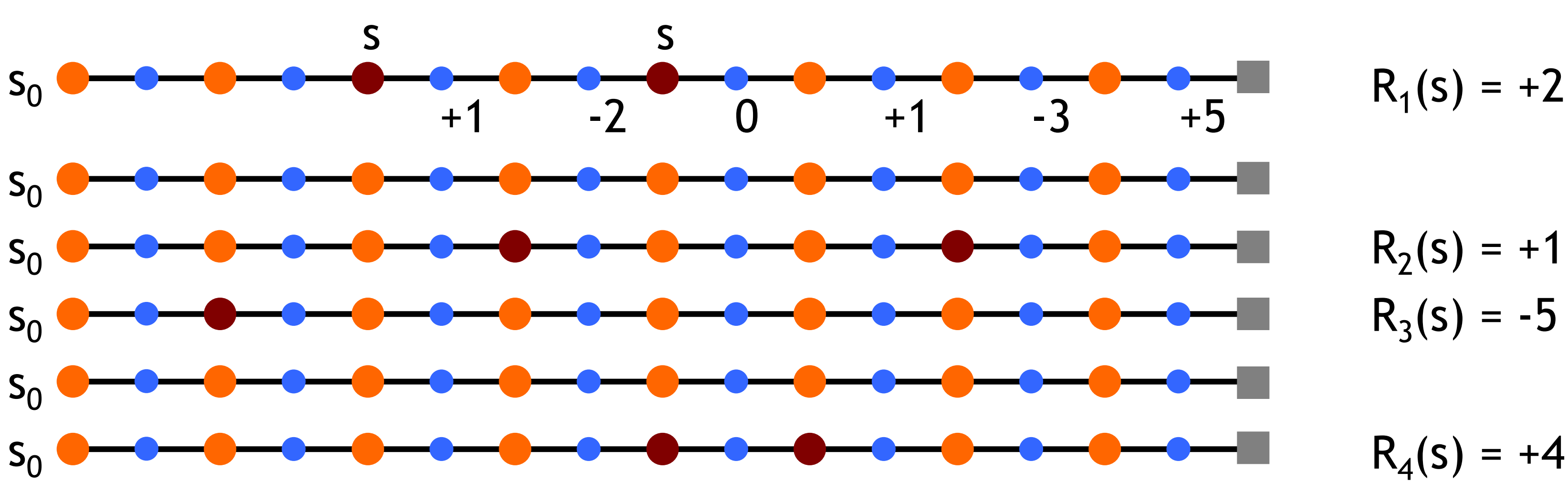
want to estimate $V^{\pi}(s)$

Monte Carlo policy evaluation

don't need full knowledge of environment (just (simulated) experience)

want to estimate $V^{\pi}(s)$

- expected return starting from s and following π
- estimate as average of observed returns in state s



$$V^{\pi}(s) \approx (2 + 1 - 5 + 4)/4 = 0.5$$

Slide courtesy/adapted: Peter Bodík

Maintaining exploration

key ingredient of RL

deterministic/greedy policy won't explore all actions
- don't know anything about the environment at the beginning
- need to try all actions to find the optimal one

maintain exploration
- use *soft* policies instead: $\pi(s,a)>0$ (for all s,a)

ε-greedy policy
- with probability 1-ε perform the optimal/greedy action
- with probability ε perform a random action

- will keep exploring the environment
- slowly move it towards greedy policy: ε -> 0

Slide courtesy/adapted: Peter Bodík

RL Summary 1:

- **Reinforcement learning systems**
  - Learn **series** of actions or decisions, rather than a single decision
  - Based on feedback given at the end of the series
- A reinforcement learner has
  - A goal
  - Carries out trial-and-error search
  - Finds the best paths toward that goal

RL Summary 2:

- A typical reinforcement learning system is an active agent, interacting with its environment.
- It must balance:
  - Exploration: trying different actions and sequences of actions to discover which ones work best
  - Exploitation (achievement): using sequences which have worked well so far
- Must learn **successful sequences of actions** in an uncertain environment

RL Summary 3

- Very hot area of research at the moment
- There are **many** more sophisticated RL algorithms
  - Most notably: probabilistic approaches
- Applicable to game-playing, search, finance, robot control, driving, scheduling, diagnosis, …