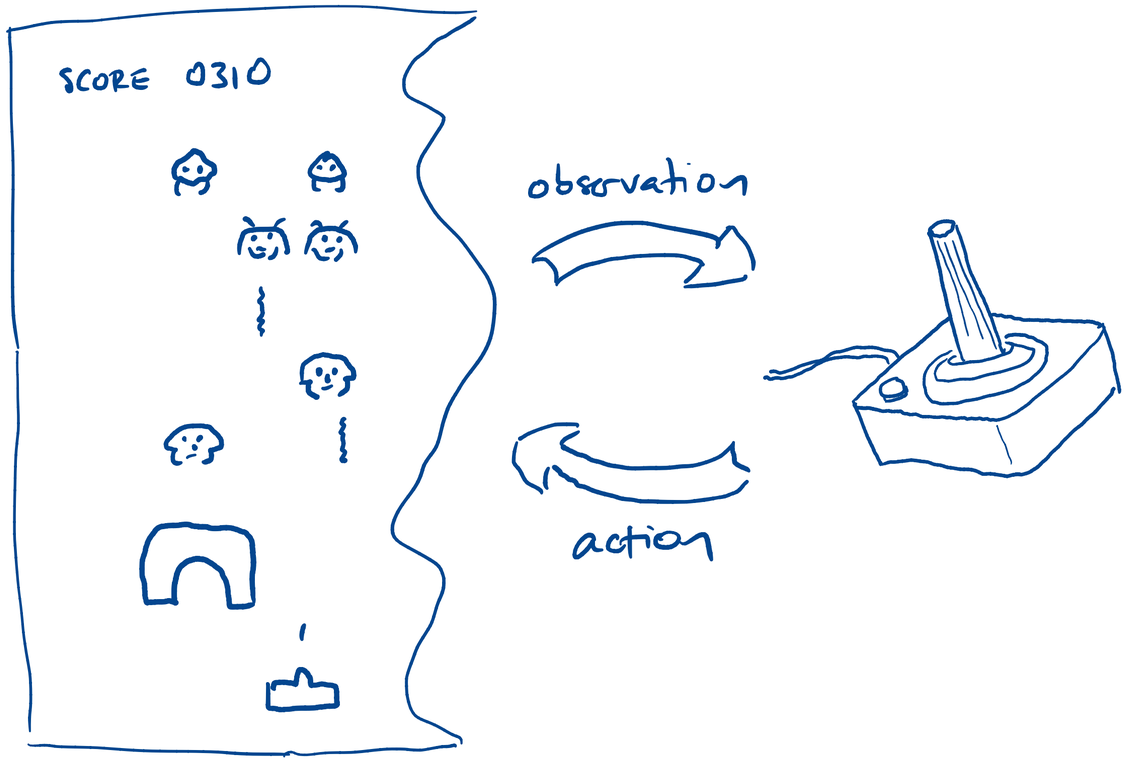
Reinforcement learning

10-701 Introduction to Machine Learning

Geoff Gordon and Pradeep Ravikumar

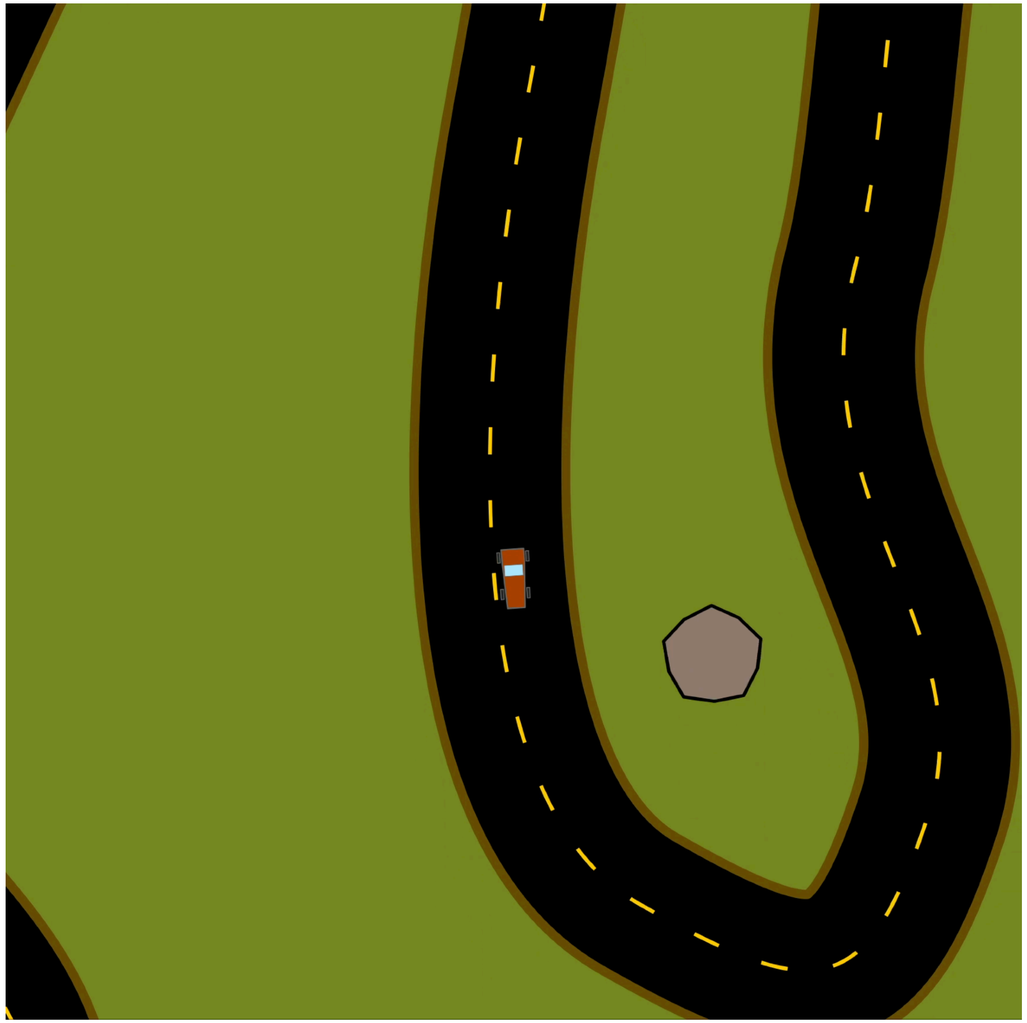
Sequential decision problem



- Agent interacts w/ environment over time
- Alternating observations and actions $o_1, a_1, o_2, a_2, \ldots$

called a *trajectory*

Example: simple racing game

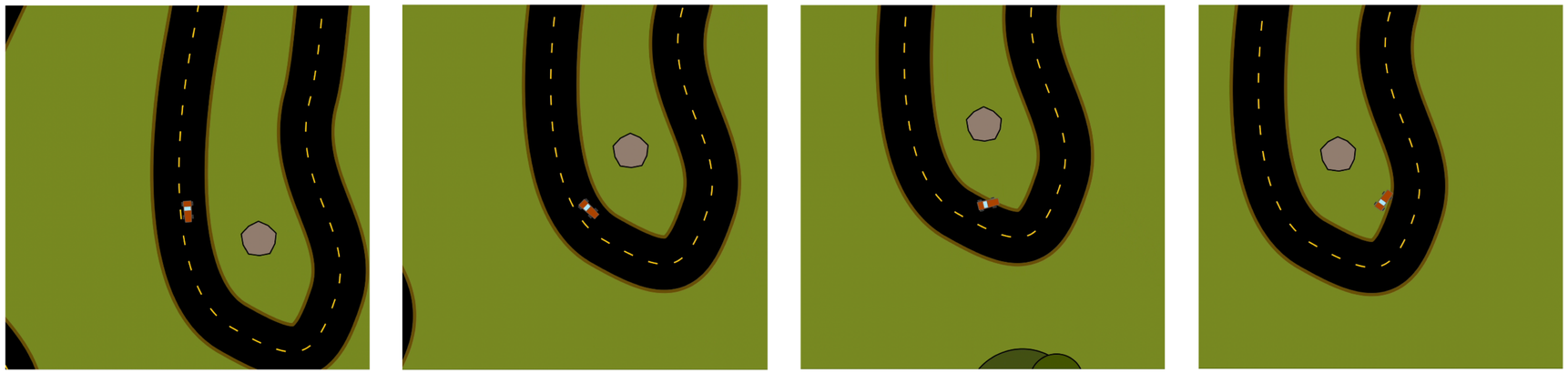


- observations: 60 fps images, downscale to 32×32
- actions: 2D real = offset from car center to steering target
- trajectory: images and steering over time

Reinforcement learning

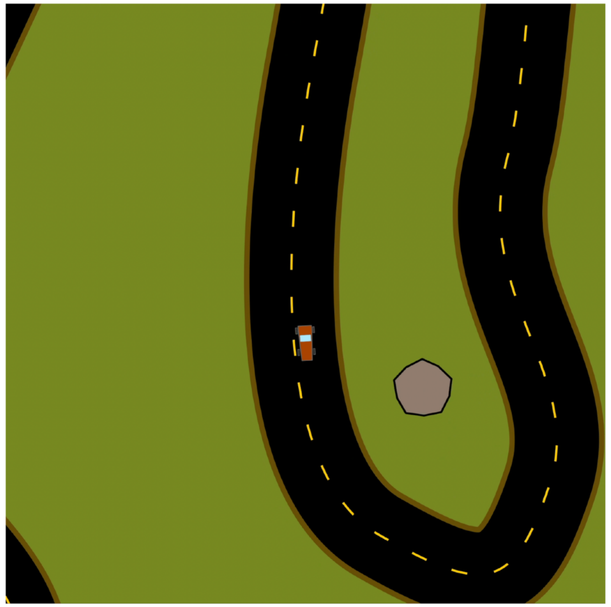
- Many possible goals in a sequential decision problem: environment modeling, reinforcement learning, imitation, apprenticeship, behavior design
- Warning: we'll cover one goal here (RL); popular but often not the right way to approach a problem
- Assume: given *one-step* costs or rewards: $c_t = c(o_t, a_t)$
- RL: act to maximize total reward or minimize total cost
  - say, sum to end of level (cross finish or pass time limit)

$\min_{\text{agent code A}} \mathbb{E}_{\text{trajectory } \tau \mid \text{A}}$ total cost of τ



cost=0 cost=0.1 cost=0.5 cost=1.1

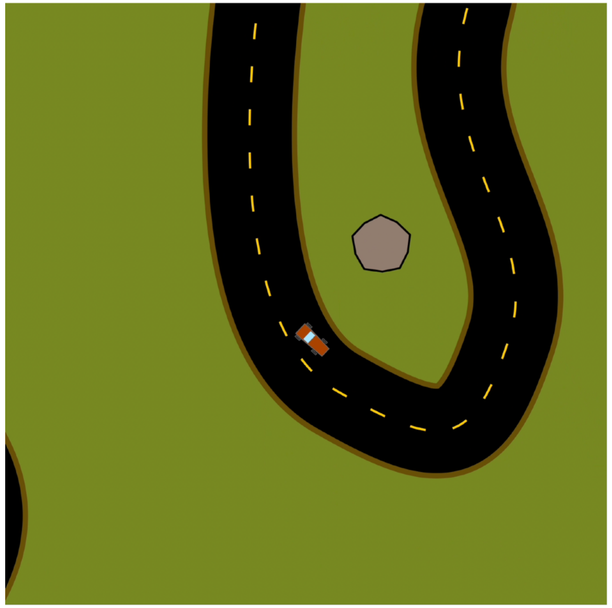
costs determined by *current* observation and action — if not, update observation features



Stay on track: cost=0

Go straight: cost=0

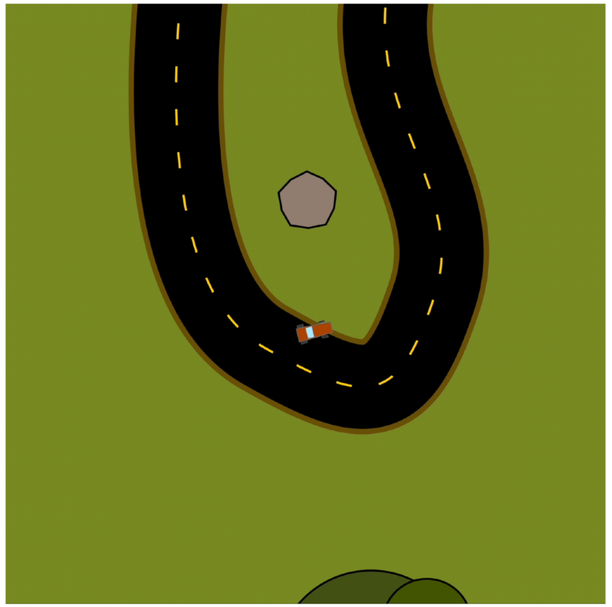
Go fast: cost=0



Stay on track: cost=0

Go straight: cost=0.7

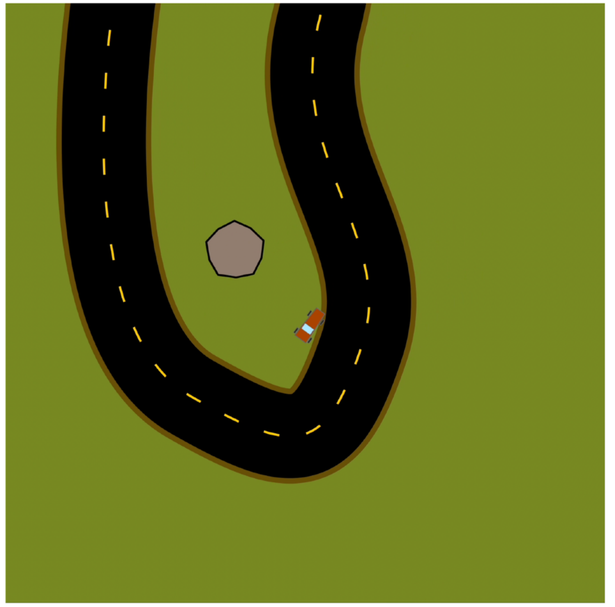
Go fast: cost=0.3



Stay on track: cost=0.5

Go straight: cost=1

Go fast: cost=0.7



Stay on track: cost=1

Go straight: cost=0.7

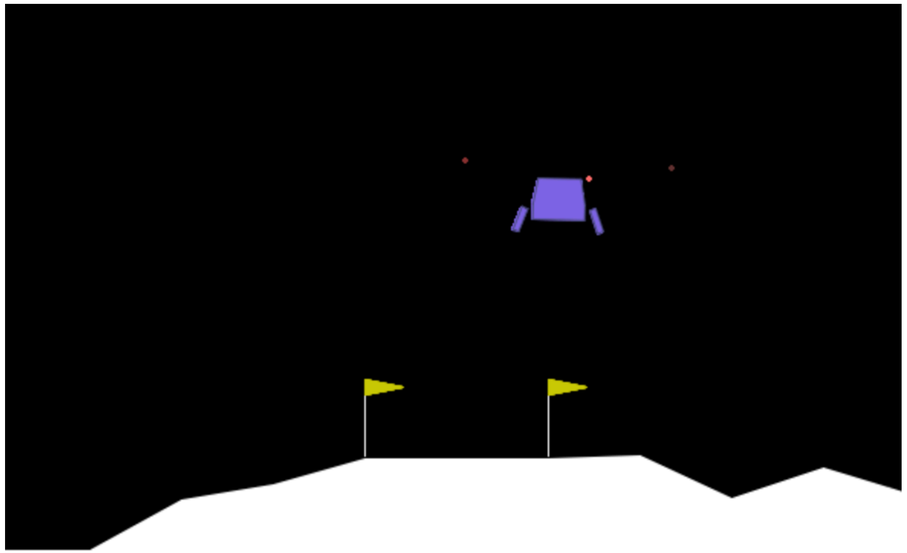
Go fast: cost=1

One-step cost

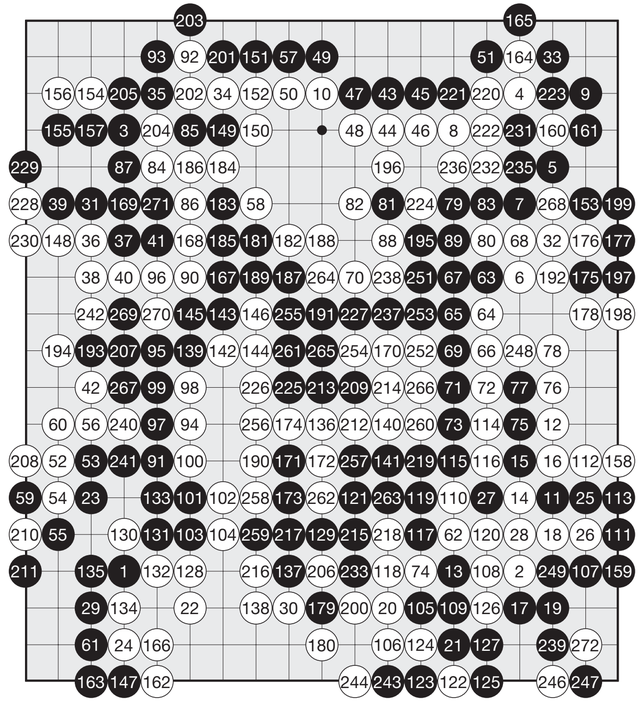
- Many possible cost/reward functions: stay on track, go fast, avoid sharp turns, …
- Choosing the right one is a hard problem in itself!
  - common source of bugs: reasonable-seeming cost/reward leads to unreasonable behavior

Example environments

Sometimes we consider the goal (cost/reward) as part of the environment, but sometimes the same environment could support more than one RL problem



observations: screen images
actions: controller buttons, joystick position
transitions: determined by game code
reward: score increase



observations: board $\{B, W, \varnothing\}^{19\times19}$
actions: place a stone
transitions: rules of Go, opponent follows a previous policy (self-play)
reward: +1 for win, –1 for loss, 0 for draw, 0 if game isn't over

Why is RL hard?

- Two key difficulties make RL harder than supervised learning
  - the ***explore-exploit dilemma***
    -
  - the ***temporal credit assignment problem***
    -

function approximation interaction

Warmup: explore-exploit

- If we fix the horizon (max episode length) to 1, RL becomes the ***contextual bandits*** problem
  - agent sees context o (independent from all previous contexts and actions)
  - chooses action a
  - gets reward r
  - repeat
- E.g.:
  - recommend a news article
  - pick ads to serve on a search page

Context

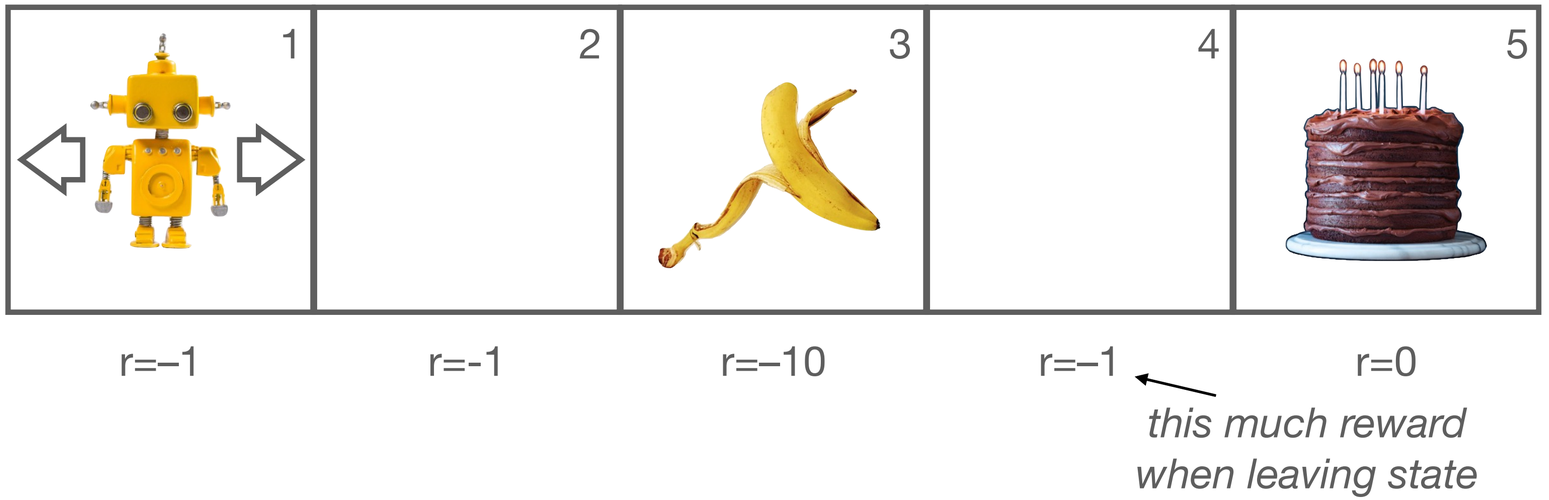
Choose an action: ***greet*** or ***run***

Results

Context	Action	Reward
B	G	+1
B	G	−1
G	G	−1
G	G	+2
B	R	0
B	G	−1

Warmup 2: tabular setting

Both explore-exploit and temporal credit assignment, but no worry about how to approximate high-d functions



- Tractable set of observations {1..5} × actions {L, R}
- The ***Markov property***
  - we only need to remember the most recent observation
  - $P(o_{t+1} \mid o_1, a_1, \ldots, o_t, a_t) = P(o_{t+1} \mid o_t, a_t)$
  - in this case, o_t is called a ***state***, often write s_t instead
- Why “tabular”? OK to store everything as tables: e.g., table of costs or table of best actions to take

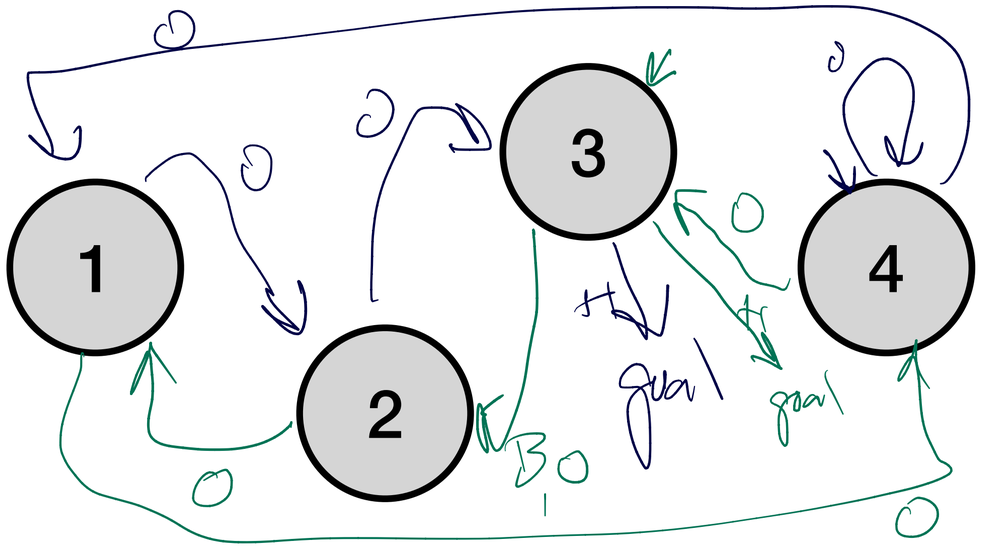
Warmup 2: tabular setting

Both explore-exploit and temporal credit assignment, but no worry about how to approximate high-d functions

Don't know full environment to start:
only $\mathcal{S} = \{1 \ldots 5\}$ and $\mathcal{A} = \{L, R\}$

- Tractable set of observations {1..5} × actions {L, R}
- The ***Markov property***
  - we only need to remember the most recent observation
  - $P(o_{t+1} \mid o_1, a_1, \ldots, o_t, a_t) = P(o_{t+1} \mid o_t, a_t)$
  - in this case, o_t is called a ***state***, often write s_t instead
- Why "tabular"? OK to store everything as tables: e.g., table of costs or table of best actions to take

Collect data, learn $P(s_1)$***,*** $r(s, a)$***, and*** $T(s' \mid s, a)$



start state

1	2	3	4
		1	1

4 states {1, 2, 3, 4}
2 actions {A, B}

note: I should have kept separate counts for actions A & B here

from state	count	reward (A)	reward (B)	to state (A) 1	2	3	4	to state (B) 1	2	3	4
1	1	0	0		1						1
2	1	0	0			1		1			
3	1	1	1						1		
4	1	0	0	1			1			1	

$P(s_1)$: initial state dist'n

$r(s, a)$: one-step reward

$T(s' \mid s, a)$: transition prob

note: ***substochastic***

Aside: time as state

- In our tabular examples so far, environment dynamics don't change with time
  - the environments are ***stationary***
- If we want a nonstationary environment (e.g., a deadline by which something has to happen, or a bonus for doing something early), just include the time index as part of the state
  - $T(s' \mid s, a)$ will increment it
  - finite $\mathcal{S}$ means there's a hard time limit — the horizon
  - we'll effectively keep one set of tables per time step
- Or we can have an approximate time index: e.g., a counter that increments w.p. $\frac{1}{2}$ and maxes at 20 means we know approximate time for ~40 steps, then "it's been a while"

Agent behavior = policy

- Policy = function π
  - input: history (trajectory so far) $o_1, a_1, \ldots, o_t$
  - output: $P(\text{action} \mid \text{history})$
- With Markov assumption:
  - enough to depend just on state (last observation)
  - write $\pi(a_t \mid s_t)$
  - optimal policy can be deterministic, but we typically optimize over stochastic policies (continuous set)
- Optimal answer: policy with best expected (total) reward

Policy

State	*P(left)*	*P(right)*
1	0.3	0.7
2	0	1
3	0.5	0.5
4	0.4	0.6
5	0.8	0.2

Estimating the total reward of a policy

- Naive method: run some trajectories with agent acting according to policy π, average their total rewards
- Insight: the actual sequence of states visited is informative — use it to reduce variance
- We'll estimate rewards for each state, use relationships among states to improve accuracy
- ***Defn**:* the ***value function*** of π maps state $\rightarrow$ total expected future reward if we are at that state following π

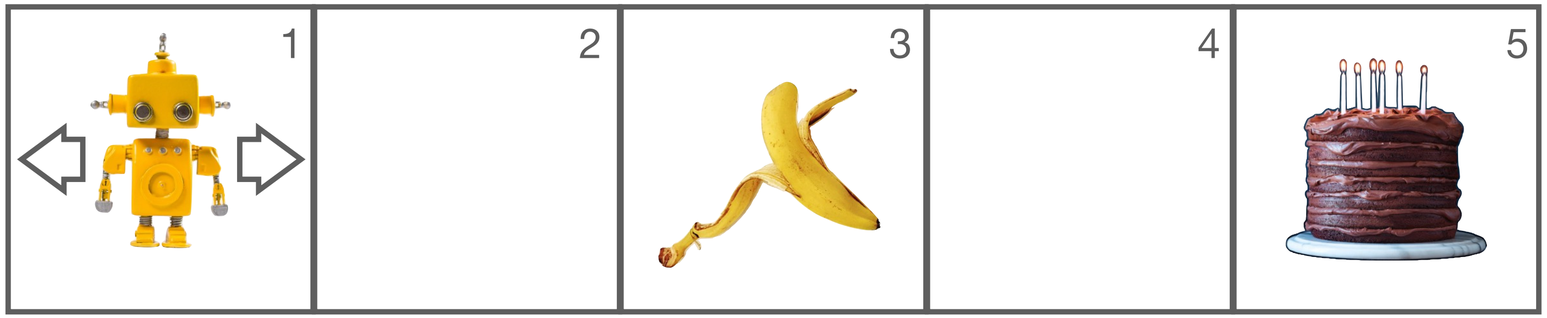
$$V^\pi(s) = \mathbb{E}\left[\sum_{i=t,\,t+1,\,\ldots} r_i \,\middle|\, s_t = s, \text{following } \pi\right]$$

  - for now, assume expectation exists; we'll return to this

Value function example

For simplicity, suppose we know the exact ~~MDP~~ environment

Environment



Policy π: always move right

Trajectory ends with any action from 5

Table of V^π

1	2	3	4	5
–13	–12	–11	–1	0

Computing the value function

$$
\begin{aligned}
V^\pi(s) &= \mathbb{E}(r_1 + r_2 + r_3 + \ldots \mid s_1 = s, \text{following } \pi) \\
&= \mathbb{E}(r_1 + \mathbb{E}(r_2 + r_3 + \ldots \mid s_2 = \text{observed } s_2, \pi) \mid s_1 = s, \pi) \\
&= \mathbb{E}(r_1 + V^\pi(s_2) \mid s_1 = s, \pi) \\
&= \sum_a \pi(a \mid s) \left[r(s,a) + \sum_{s'} T(s' \mid s, a) V^\pi(s') \right]
\end{aligned}
$$

note: T is substochastic — handles end of trajectory automatically

- Split $V^\pi(s)$ into first step and subsequent steps
- Use law of iterated expectations + Markov property
- Result: V^π is expressed recursively (in terms of itself)
  - called the ***Bellman equation***
  - will allow a ***dynamic programming*** algorithm

Value iteration for V^π

Dynamic programming:
replace = by ← in Bellman equation

Environment

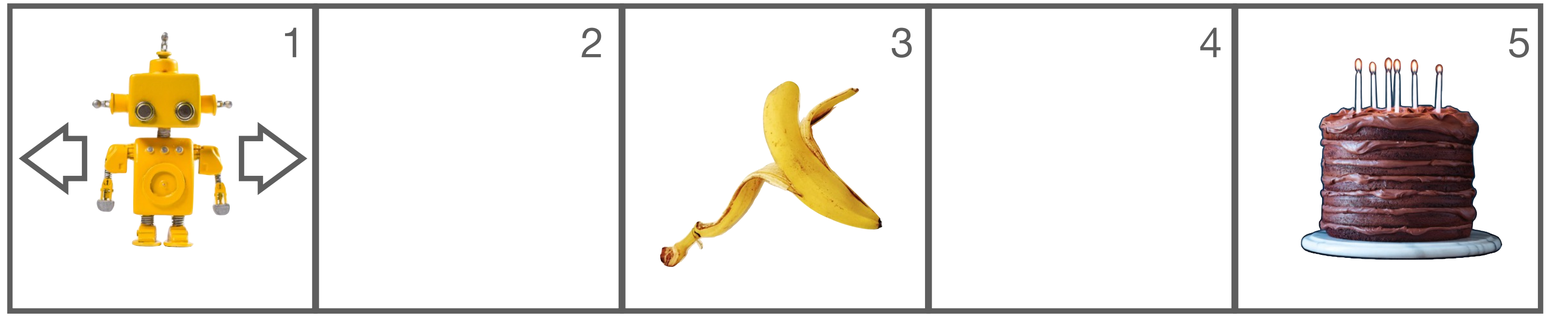
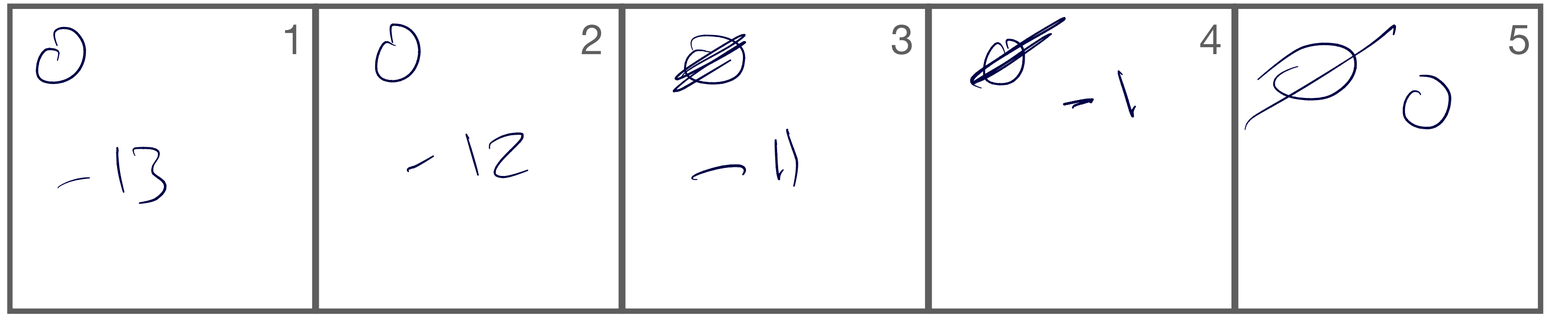


Table of V^π



- Given: $r(s,a)$, $T(s' \mid s,a)$ (exact or learned)
- Initialize $V^\pi(s)$ arbitrarily (e.g., to 0 for all s)

or warm start if we know one

- Repeat until converged
  - for each state s (in parallel or in an arbitrary order)

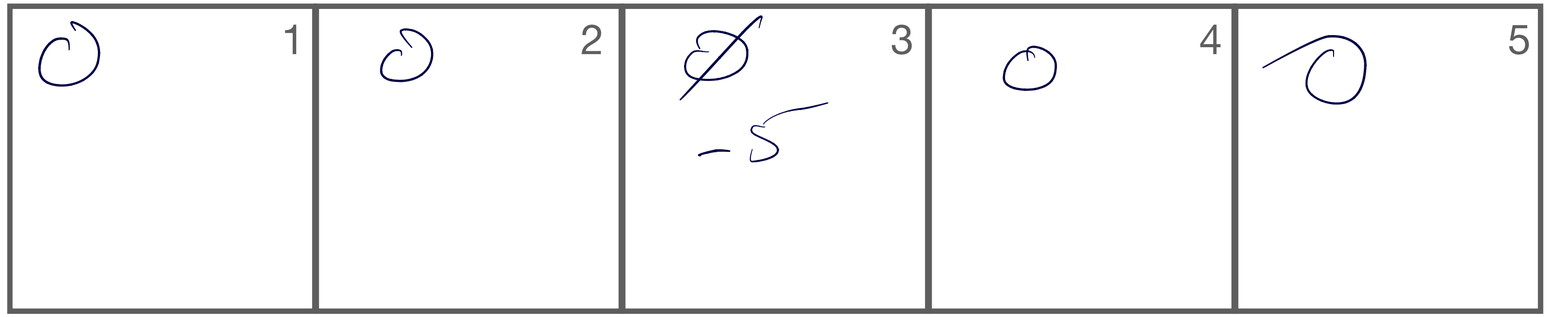
$$V^\pi(s) \leftarrow \sum_a \pi(a \mid s)\left[r(s,a) + \sum_{s'} T(s' \mid s,a) V^\pi(s')\right]$$

Temporal difference (TD) learning

same idea, but from data instead of from environment model

Data $1 \xrightarrow{-1} 2 \xrightarrow{-1} 3 \xrightarrow{-10} 4 \xrightarrow{-1} 5 \xrightarrow{0}$ done (π: always R)

Table of V^π



- Given: observed transitions from π, $\{s_t, a_t, r_t, s_{t+1}\}$
- Initialize $V^\pi(s)$ arbitrarily (e.g., to 0 for all s)
- Repeat until converged
  - sample a transition s, a, r, s' (or minibatch)
  - compute target(s) $r + V^\pi(s')$
    - notes: a not used; frozen copy of V^π (stop-grad)
  - update each $V^\pi(s)$ by SGD towards its target

Poll

- Which of the following statements are true about the Markov property?
  A. The Markov property holds when the most recent observation is enough to let us predict the future (as accurately as knowing the entire history)
  B. The Markov property holds in all the Atari games, if we use a window of the last 5 video frames and controller inputs as our state representation
  C. The value iteration algorithm requires the Markov property in order to have guaranteed convergence to the optimal value function
  D. All the above
  E. A & C only

Optimal value function

Def'n:

$$V^*(s) = \max_\pi \mathbb{E}(r_1 + r_2 + r_3 + \ldots \mid s_1 = s, \text{following } \pi)$$

$$V^*(s) = \max_a \mathbb{E}\left[r_1 + \max_\pi \mathbb{E}(r_2 + r_3 + \ldots \mid s_2 = \text{observed } s_2, \pi) \mid s_1 = s, a_1 = a)\right]$$

$$= \max_a \mathbb{E}\left[r_1 + V^*(s_2) \mid s_1 = s, a_1 = a\right]$$

$$= \max_a \left[r(s,a) + \sum_{s'} T(s' \mid s, a) V^*(s')\right]$$

- Decompose $V^*(s)$ into first step and subsequent steps, using law of iterated expectations as above
- Split $\max_\pi$ into separate max over a_1 and over π for $t = 2$ onward
- Result: V^* expressed recursively (another Bellman eq)

Value iteration for V^*

Replace = by ← in Bellman equation

Environment

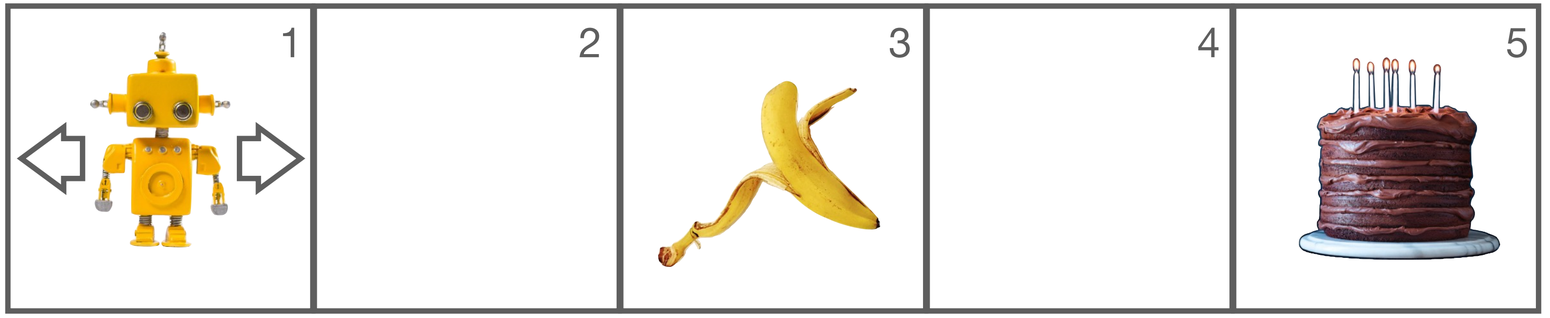
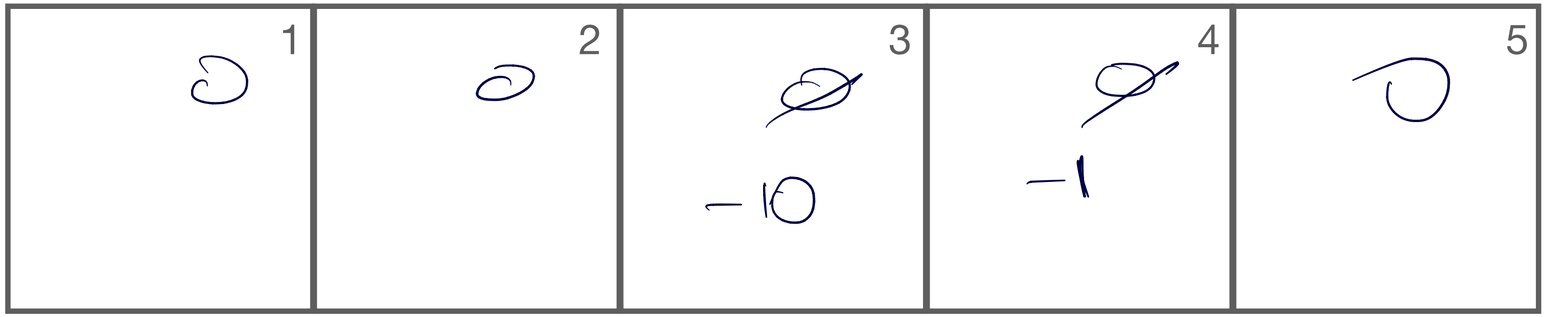


Table of V^π



- Given: $r(s,a)$, $T(s' \mid s,a)$
- Initialize $V^*(s)$ arbitrarily (e.g., to 0 for all s)
- Repeat until converged
  - for each state s (in parallel or in an arbitrary order)

$$V^*(s) \leftarrow \max_a \left[r(s,a) + \sum_{s'} T(s' \mid s,a) V^*(s') \right]$$