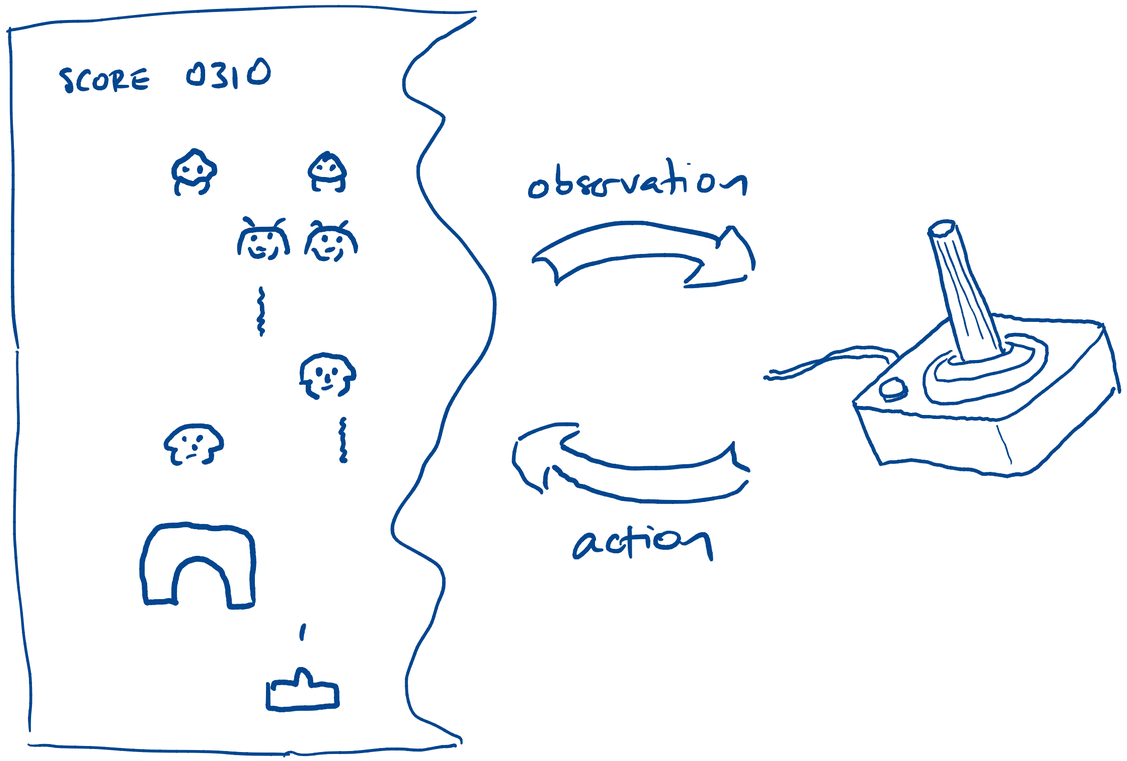
Reinforcement learning

10-701 Introduction to Machine Learning

Geoff Gordon and Pradeep Ravikumar

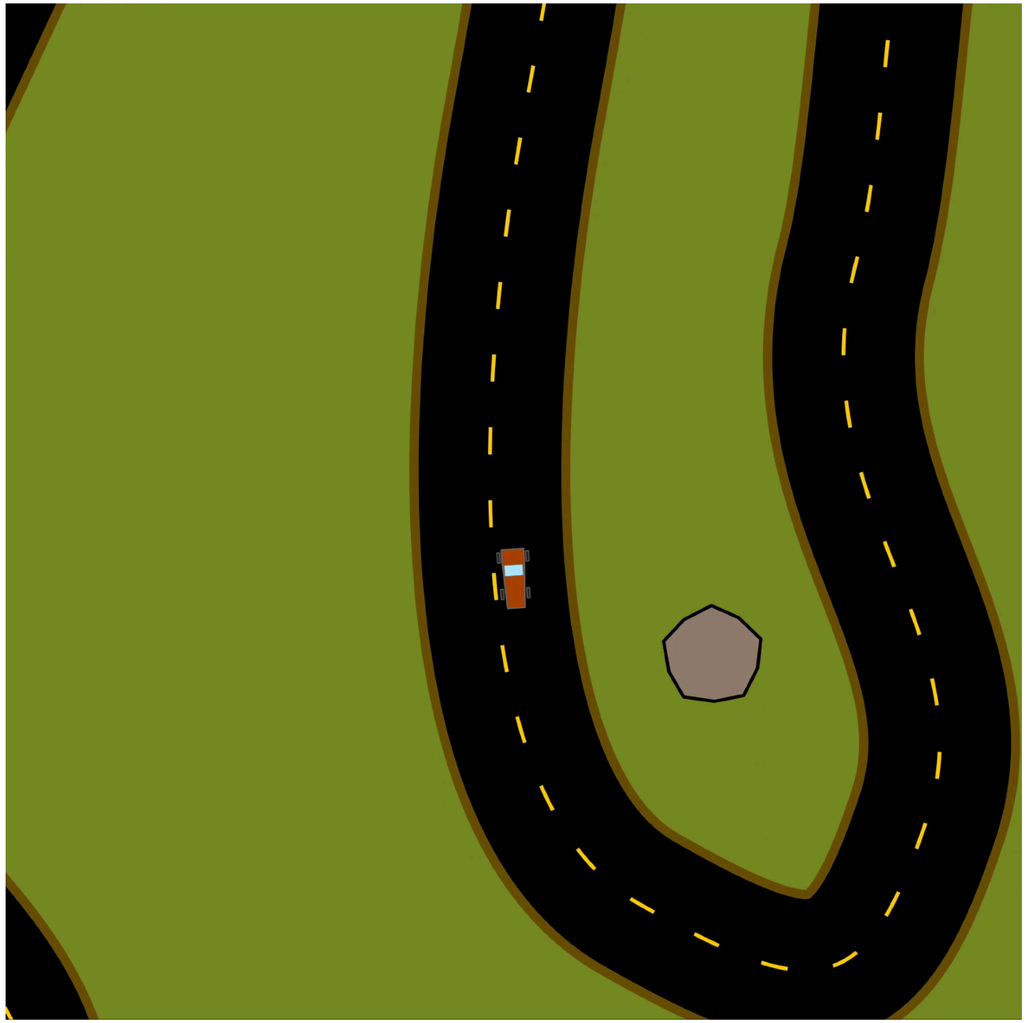
Sequential decision problem



- 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

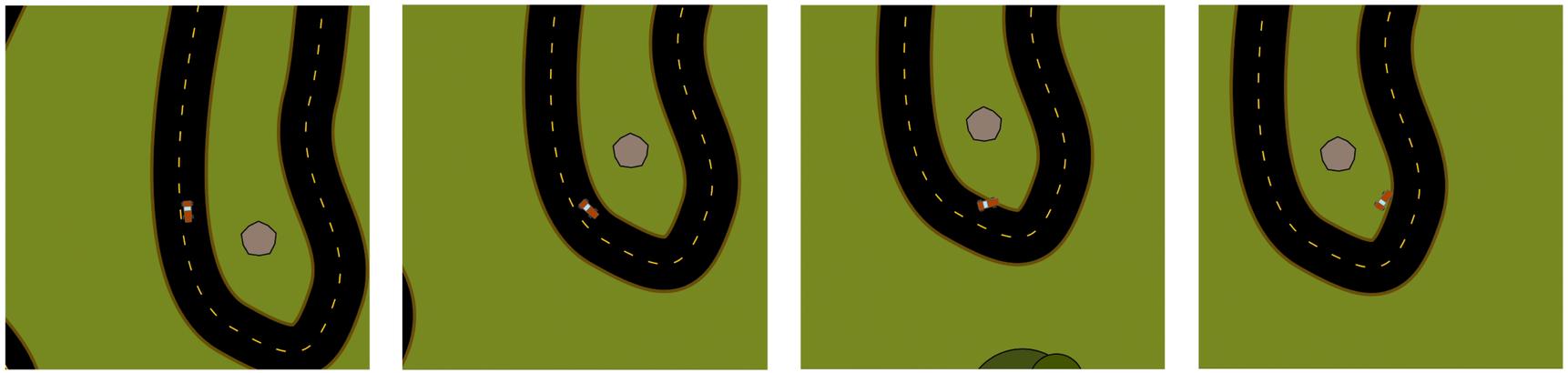


- observations: 60 fps images, downscale to 32×32
- actions: 2D real = offset from car center to steering target
- trajectory: images and steering over time

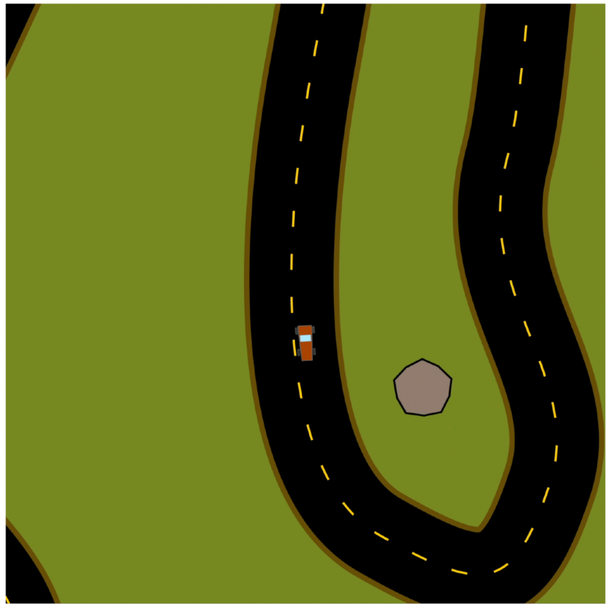
Reinforcement learning

- 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)

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



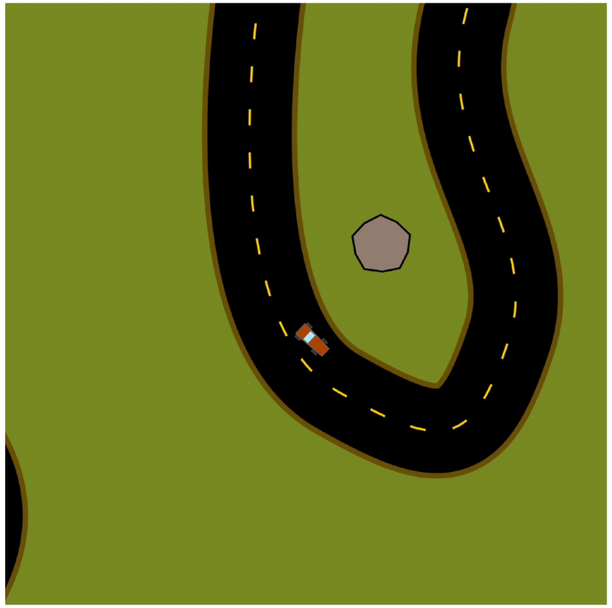
cost=0 cost=0.1 cost=0.5 cost=1.1



Stay on track: cost=0

Go straight: cost=0

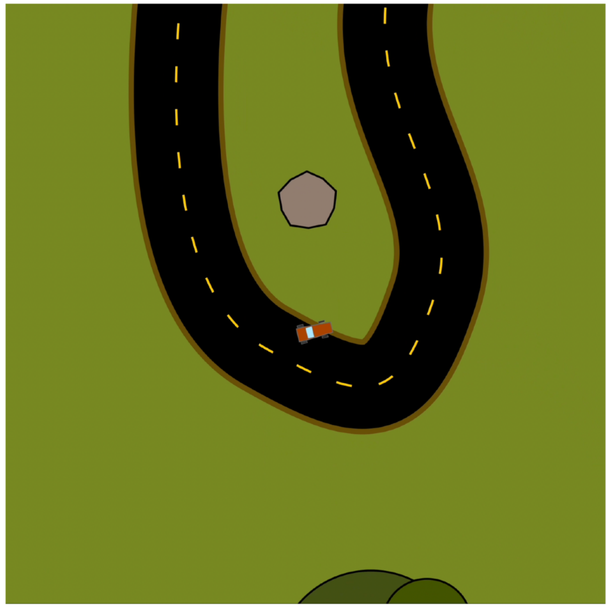
Go fast: cost=0



Stay on track: cost=0

Go straight: cost=0.7

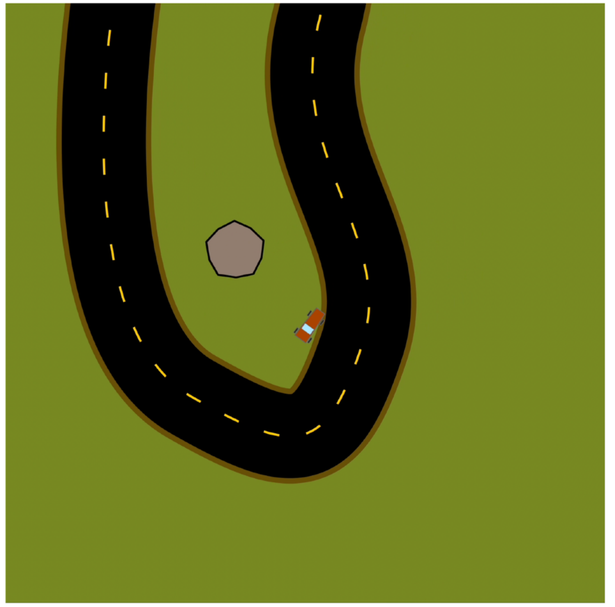
Go fast: cost=0.3



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

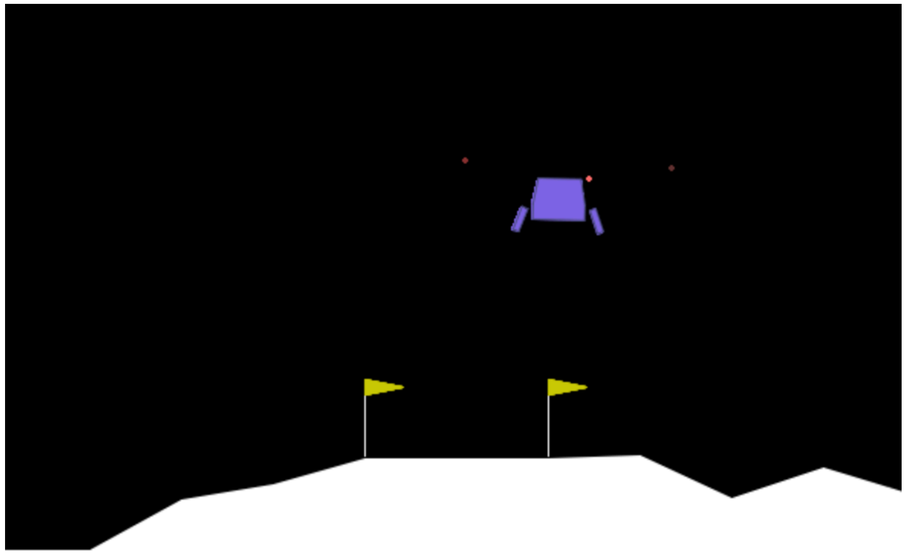
Go fast: cost=1

One-step cost

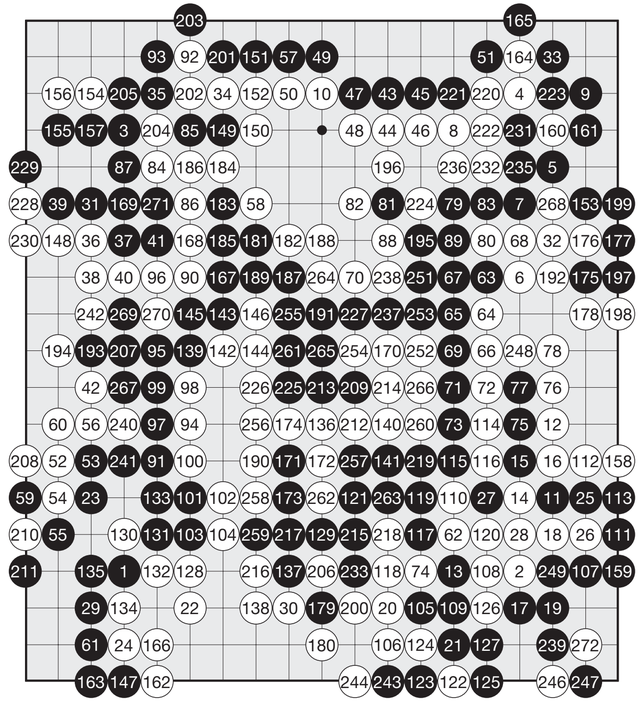
- Many possible cost/reward functions: stay on track, go fast, avoid sharp turns, …
- Choosing the right one is a hard problem in itself!
  - common source of bugs: reasonable-seeming cost/reward leads to unreasonable behavior

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***
    -

Warmup: explore-exploit

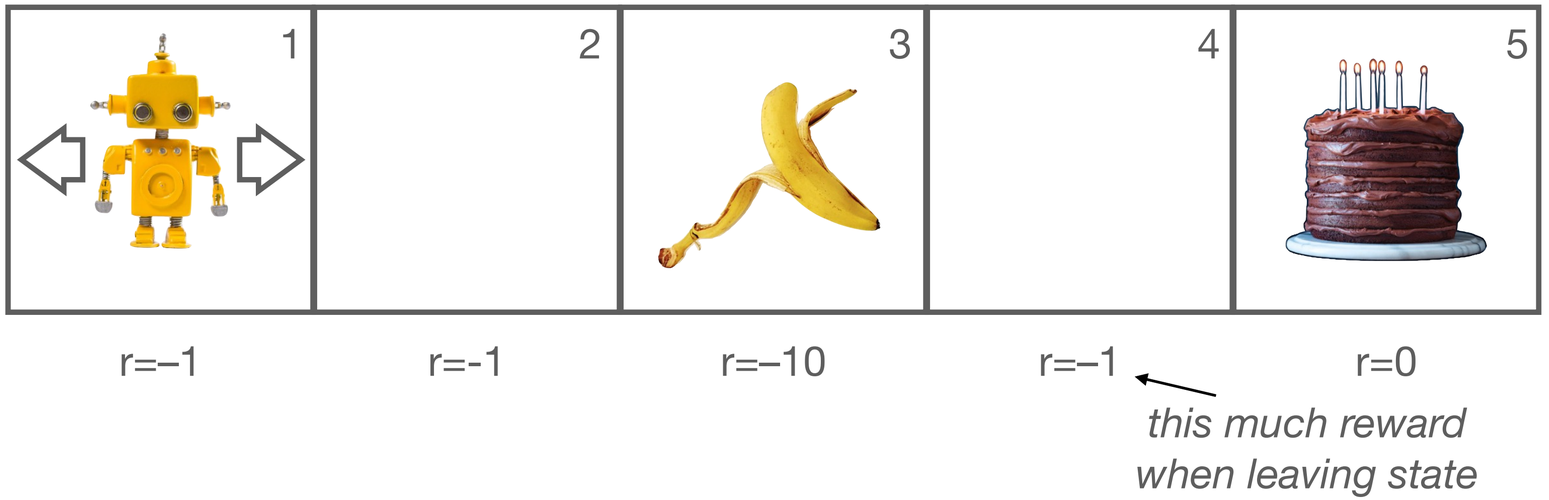
- If we fix the horizon (max episode length) to 1, RL becomes the ***contextual bandits*** problem
  - agent sees context o (independent from all previous contexts and actions)
  - chooses action a
  - gets reward r
  - repeat

Results

Context	*Action*	*Reward*

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

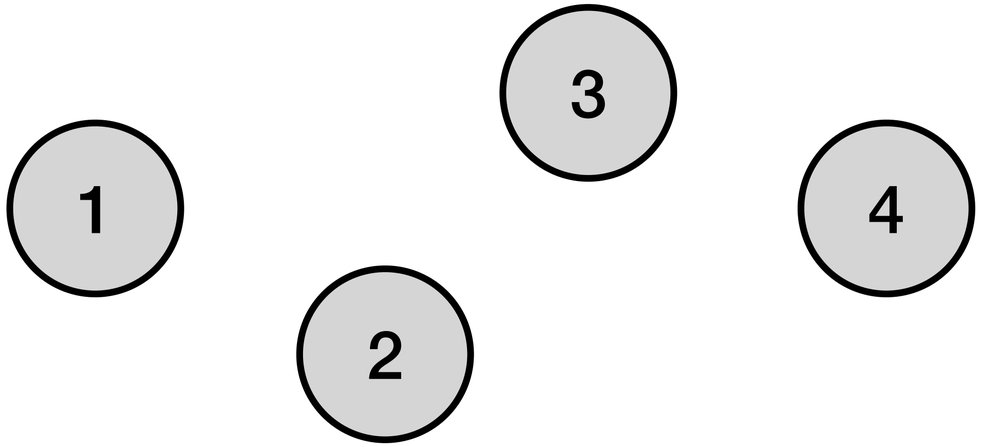
Collect data, learn $P(s_1)$***,*** $r(s, a)$***, and*** $T(s' \mid s, a)$

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

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

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

note: ***substochastic***



start state

1	2	3	4

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

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

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 tabular 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 reward

Policy

State	***P(left)***	***P(right)***
1		
2		
3		
4		
5		

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

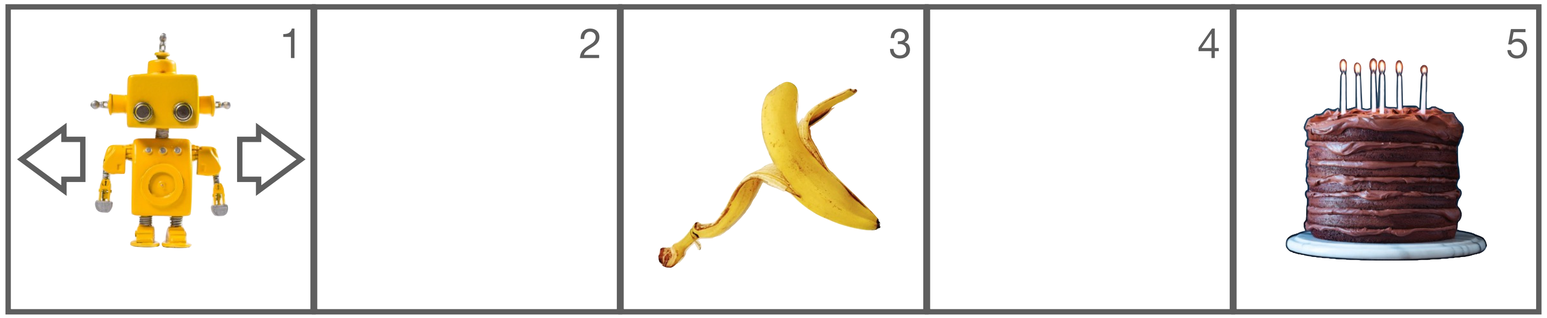
$$V^\pi(s) = \mathbb{E}\left[\sum_{i=t,\,t+1,\,\ldots} r_i \,\middle|\, s_t = s, \text{following } \pi\right]$$

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

Value function example

For simplicity, suppose we know the exact MDP

Environment



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
- 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

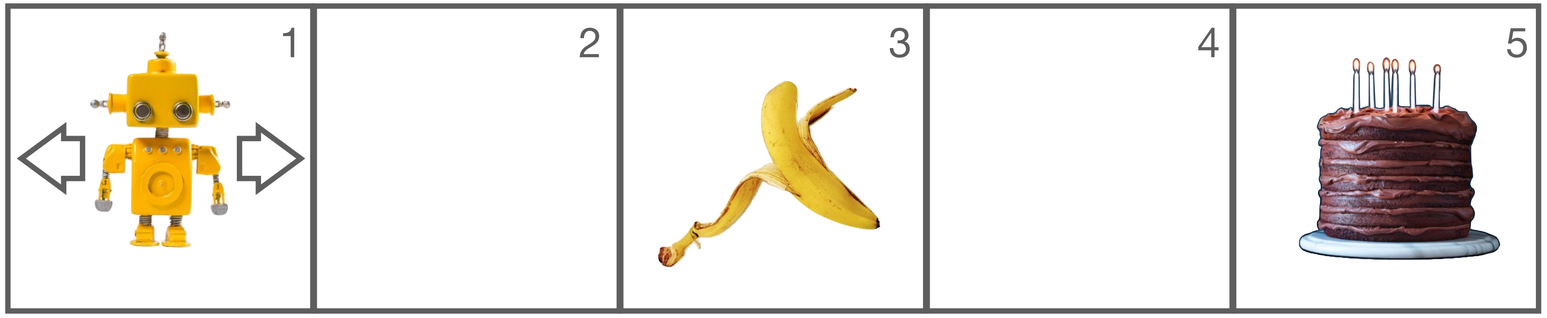


Table of V^π

1	2	3	4	5

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

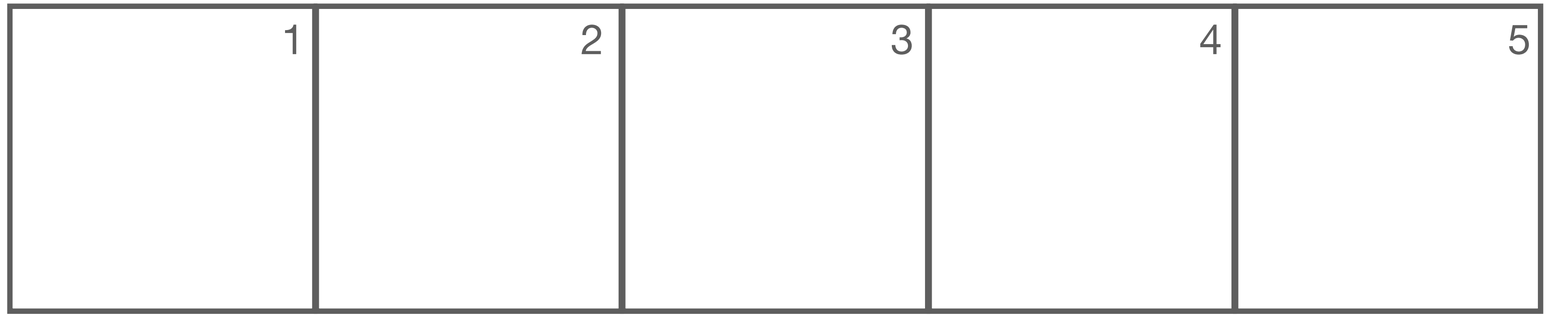
$$V^\pi(s) \leftarrow \sum_a \pi(a \mid s)\left[r(s,a) + \sum_{s'} T(s' \mid s,a) V^\pi(s')\right]$$

Temporal difference (TD) learning

same idea, but from data instead of from environment model

Data $1 \to^{1} 2 \to^{1} 3 \to^{10} 4 \to^{1} 5 \to^{0}$ done (policy: 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

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

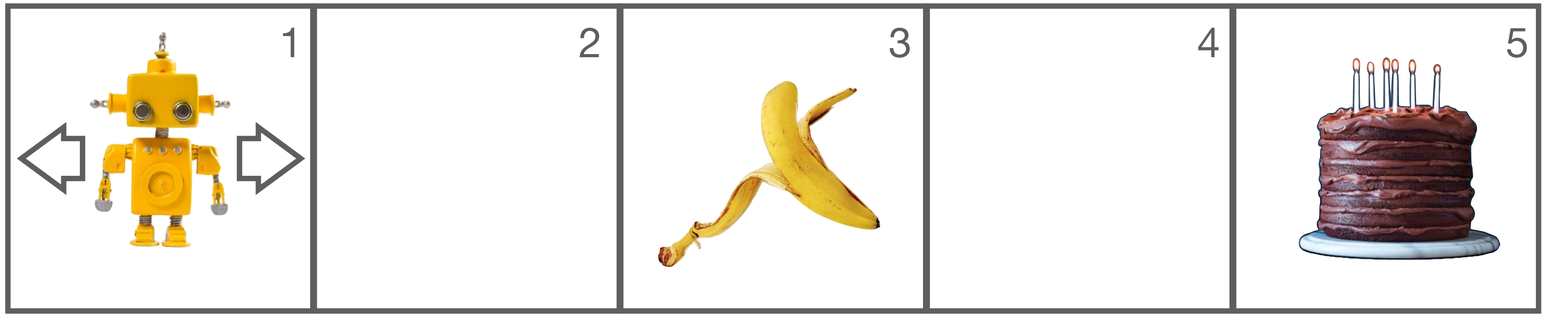


Table of V^π

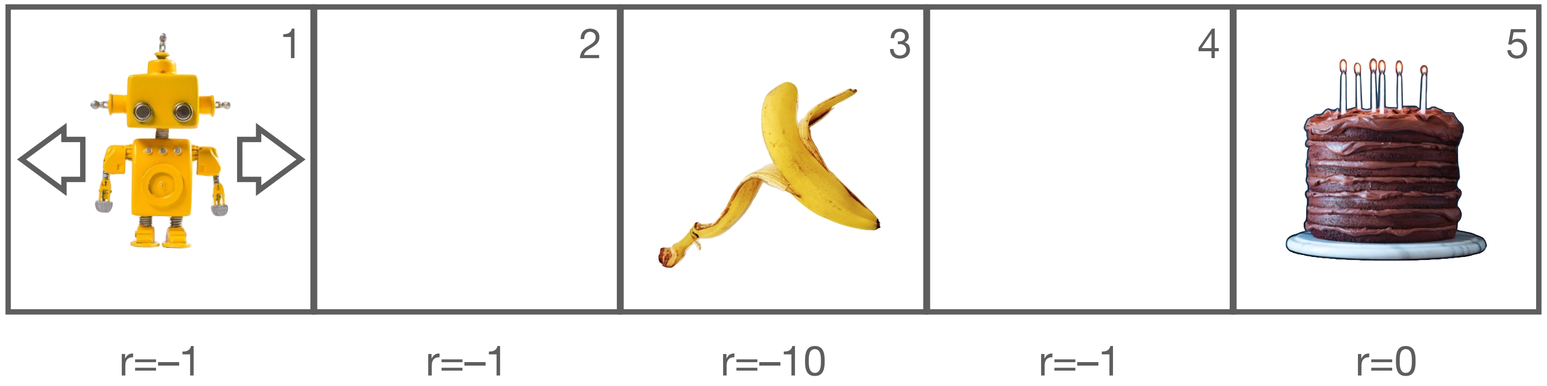
1	2	3	4	5

- 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)

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

Q function

Policy π: always move right



Q^π

$Q(1,R) = -13$	$Q(2,R) = -12$	$Q(3,R) = -11$	$Q(4,R) = -1$	$Q(5,R) = 0$
$Q(1,L) = -14$	$Q(2,L) = -14$	$Q(3,L) = -22$	$Q(4,L) = -12$	$Q(5,L) = -1$

- RHS of Bellman: $r(s,a) + \sum_{s'} T(s' \mid s,a) V(s')$
- Def'n: this is the Q function or action-value function
  - $Q^\pi(s,a)$ or $Q^*(s,a) =$ how much total reward if we start at $s_1 = s$, choose $a_1 = a$, continue with π or π^*

Dynamic programming for Q-function

- Just like the value function, dynamic programming algorithms for Q^π or Q^*
- From (given or learned) environment model:
  - for Q^π or Q^*: ***Q-iteration***
- From trajectories:
  - for Q^*: ***Q-learning***
  - for Q^π: ***SARSA***

Q-iteration

Environment

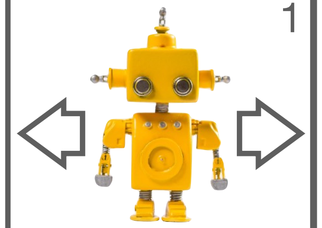
1	2	3	4	5
				
r=–1	r=–1	r=–10	r=–1	r=0

Table of Q^*

1	2	3	4	5

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

$$Q^*(s,a) \leftarrow r(s,a) + \sum_{s'} T(s' \mid s,a) \max_{a'} Q^*(s',a')$$

$$Q^\pi(s,a) \leftarrow r(s,a) + \sum_{s',a'} T(s' \mid s,a)\pi(a' \mid s')Q^\pi(s',a')$$

Q-learning

same idea, but from data instead of from environment model

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

Table of Q^*

1	2	3	4	5

- Given: observed transitions from π, $\{s_t, a_t, r_t, s_{t+1}\}$
- Initialize $Q^*(s, a)$ arbitrarily (e.g., to 0)
- Repeat until converged
  - sample a transition s, a, r, s' (or minibatch)
  - compute target(s) $r + \max_{a'} Q^*(s', a')$ [note: stop-grad]
  - update each $Q^*(s, a)$ by SGD towards its target

SARSA

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 Q^π

1	2	3	4	5

- Given: observed transitions from π, $\{s_t, a_t, r_t, s_{t+1}\}$
- Initialize $Q^\pi(s, a)$ arbitrarily (e.g., to 0)
- Repeat until converged
  - sample a transition s, a, r, s' (or minibatch)
  - compute target(s) $r + Q^\pi(s', a')$ [note: stop-grad]
  - update each $Q^\pi(s, a)$ by SGD towards its target

Discounting

- Would you rather have $10 today or $11 a year from today?

Discounting

- Would you rather have $10 today or $11 a year from today?
- The future is uncertain: “better an egg today than a hen tomorrow”
- Math: suppose our model is right w.p. $\gamma \in (0,1)$ independently on each step
  - if right, business as usual
  - if wrong, all future costs or rewards are ***independent*** of my actions and observations so far

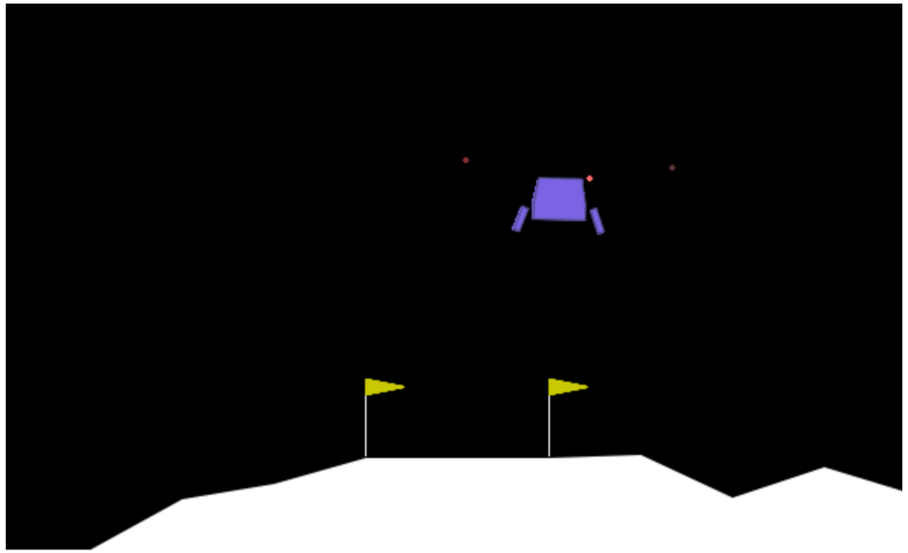
    $$V^\pi(s) = \mathbb{E}(r_1 + \gamma V^\pi(s_2) + (1-\gamma)\,\text{const} \mid s_1 = s, \pi)$$

    and similarly for V^*, Q^π, Q^*
  - unrolling, expectation of $r_1 + \gamma r_2 + \gamma^2 r_3 + \ldots$
    - “expected discounted total return”

Discounts everywhere

- Big practical advantages of discounting:
  - sum over time always converges (resolves worry from earlier about existence of expectation)
  - dynamic programming algorithms become more stable
- So it's common to use a discount factor even if it's a less-accurate model of the world
  - or to use a too-strong (too-small) discount factor
- Luckily, optimal policy is often not too sensitive to γ
  - but if we see weird behavior, remember to check whether a too-small γ is the cause
  - failure mode: excessive impatience
  - chooses a smaller reward soon, instead of waiting for a larger reward

Scaling up RL



Lunar lander

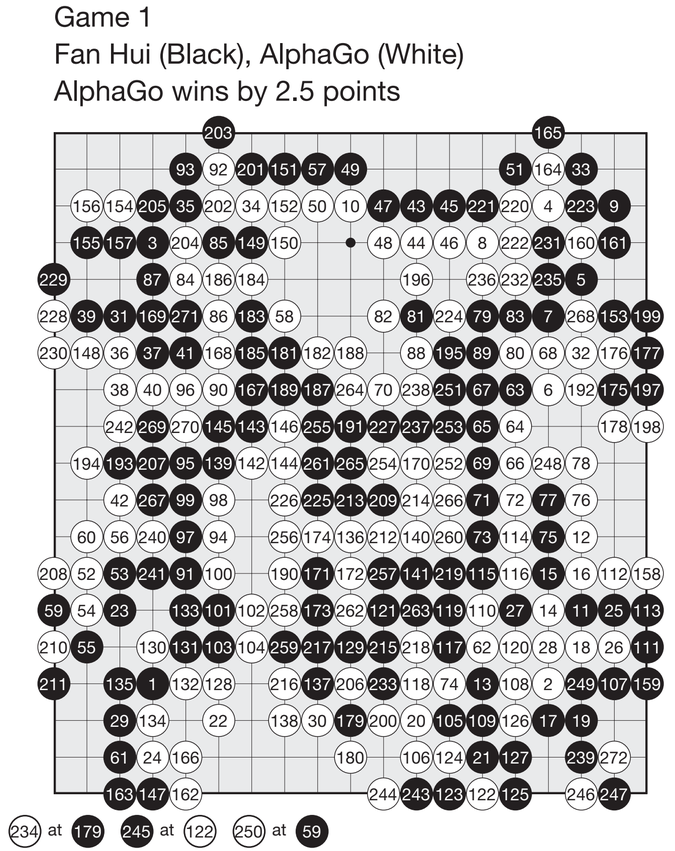
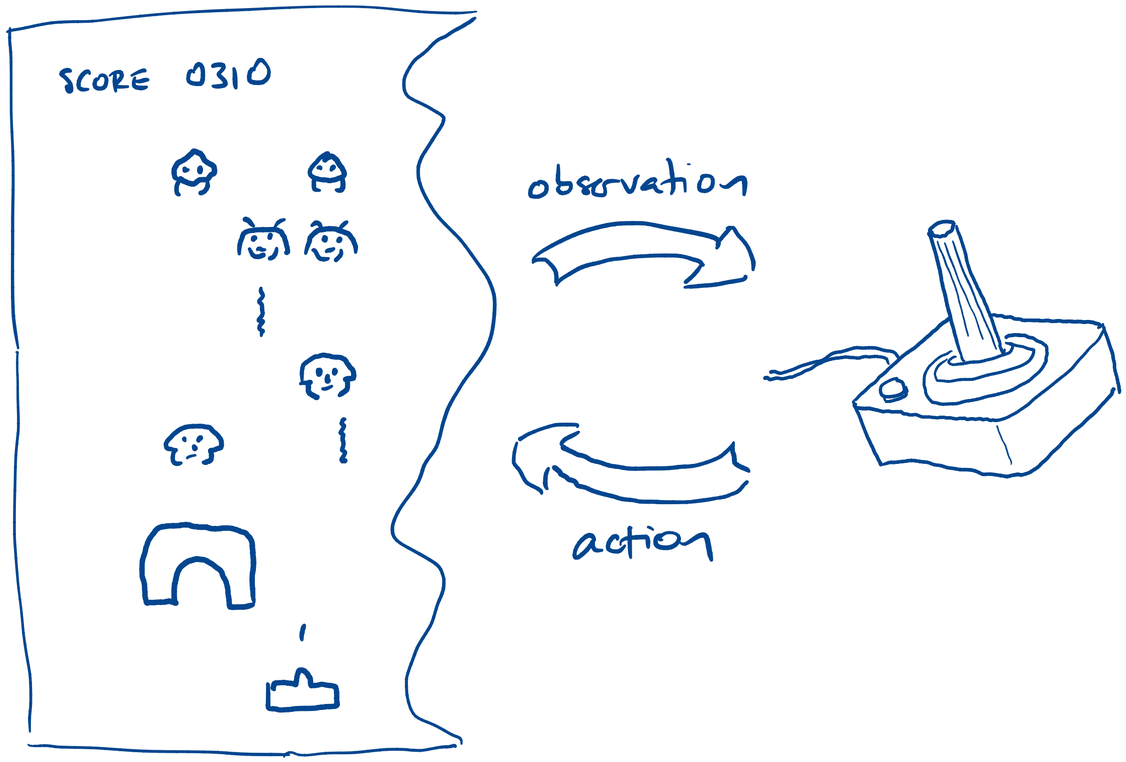


Image credit: Silver et al., Nature, 2016

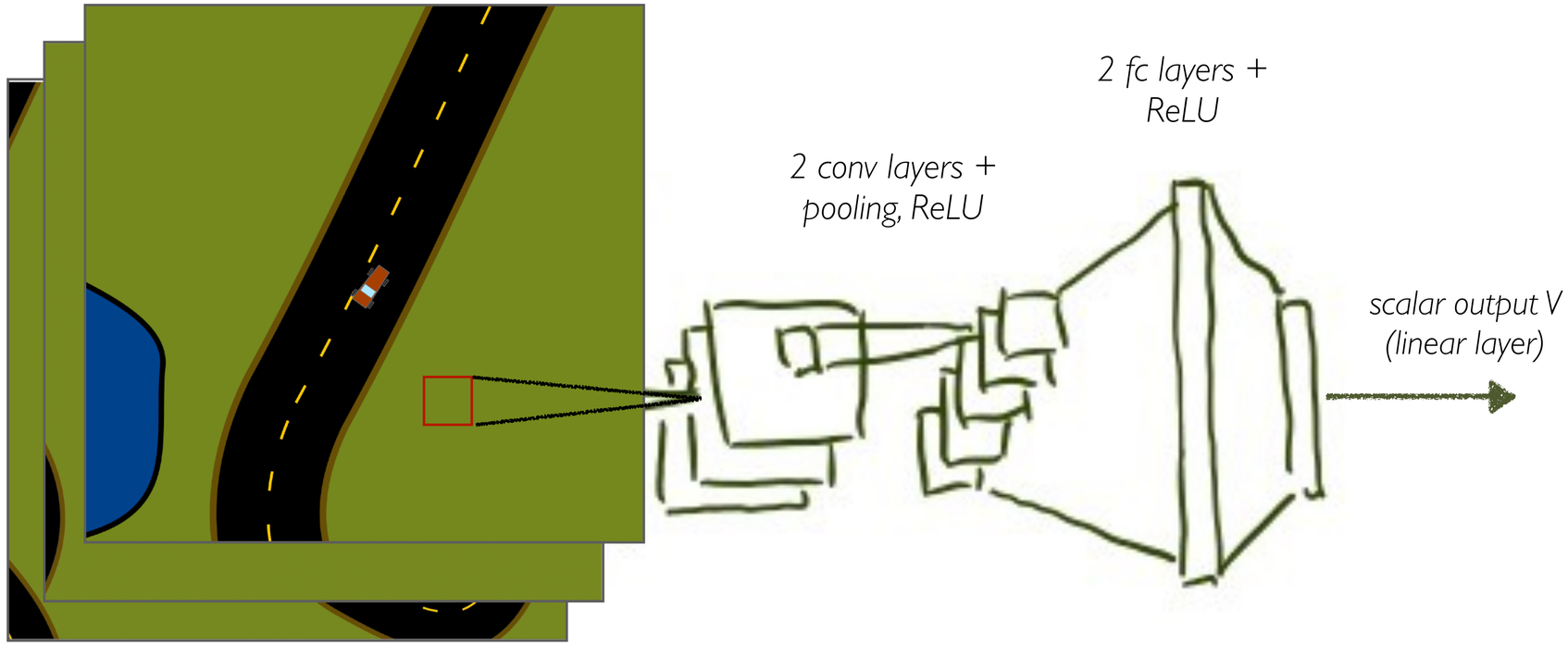
- To go beyond tabular, need some changes in our setup

No exact model



- Previously: we could write down a description of the world, e.g., expressions for $r(s, a)$ or $T(s' \mid s, a)$
- Instead: agent just interacts with environment over time — if we want $r(s, a)$ etc., have to learn it from data
- Just learn from trajectories $o_1, a_1, r_1, o_2, a_2, r_2, \ldots$

Learned, approximate functions

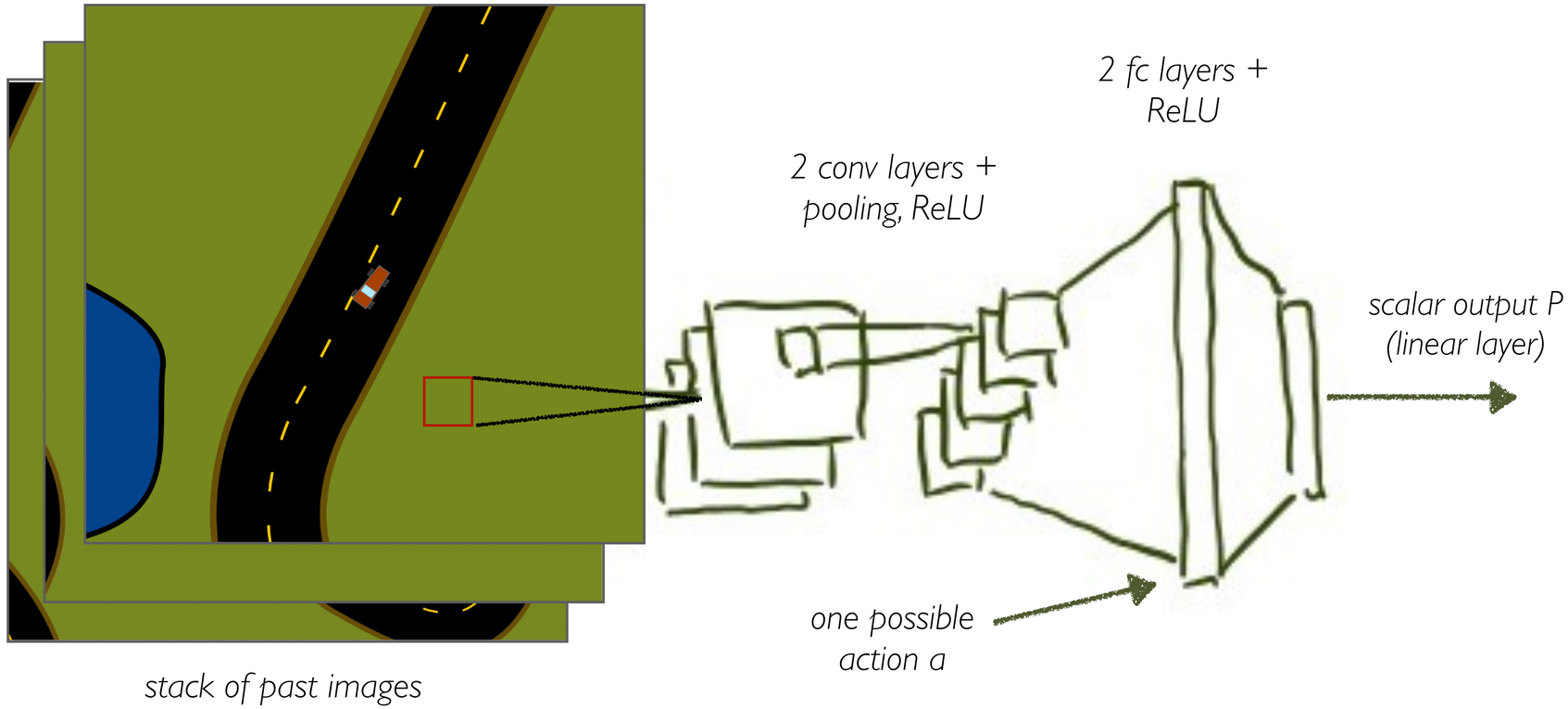


- Previously: could list out all states, keep a table of a function like $V^{\pi}(s)$
- Now: any function we care about has to be represented as an ML model, e.g., a deep net
- One parameter vector per function, each can have its own network architecture

Can't simply copy tabular algorithms

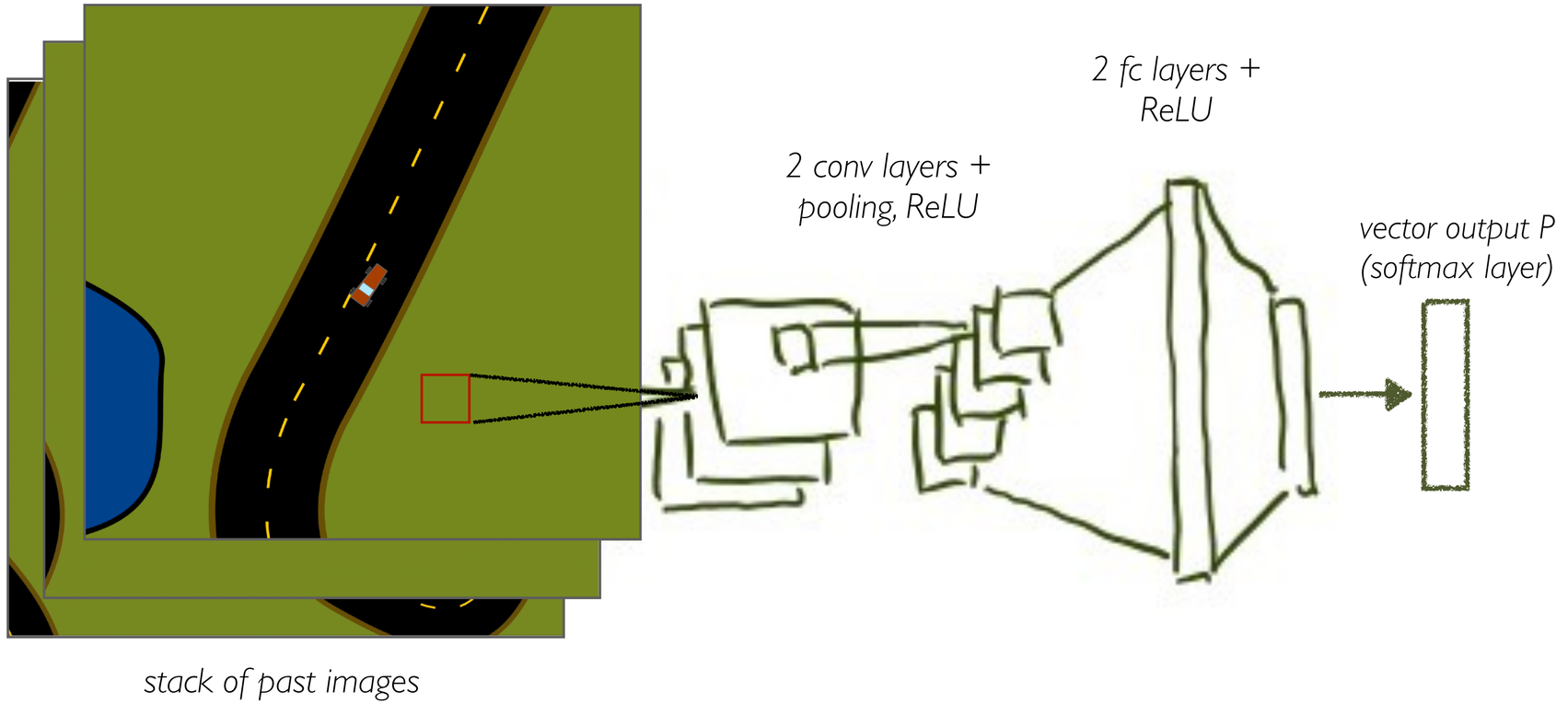
- Lots of dynamic programming algorithms for tabular case — many ***do not scale***
  - fail in weird ways when we use function approximation
- In general,
  - SGD-like policy evaluation algorithms (TD for V^π, SARSA for Q^π) still work
  - algorithms w/ max don't (Q-learning for Q^*, value iteration for V^*)
  - can make them to do something by force of engineering, e.g., DQN — but other methods work better
- But without the max, how do we improve policy?

Policy



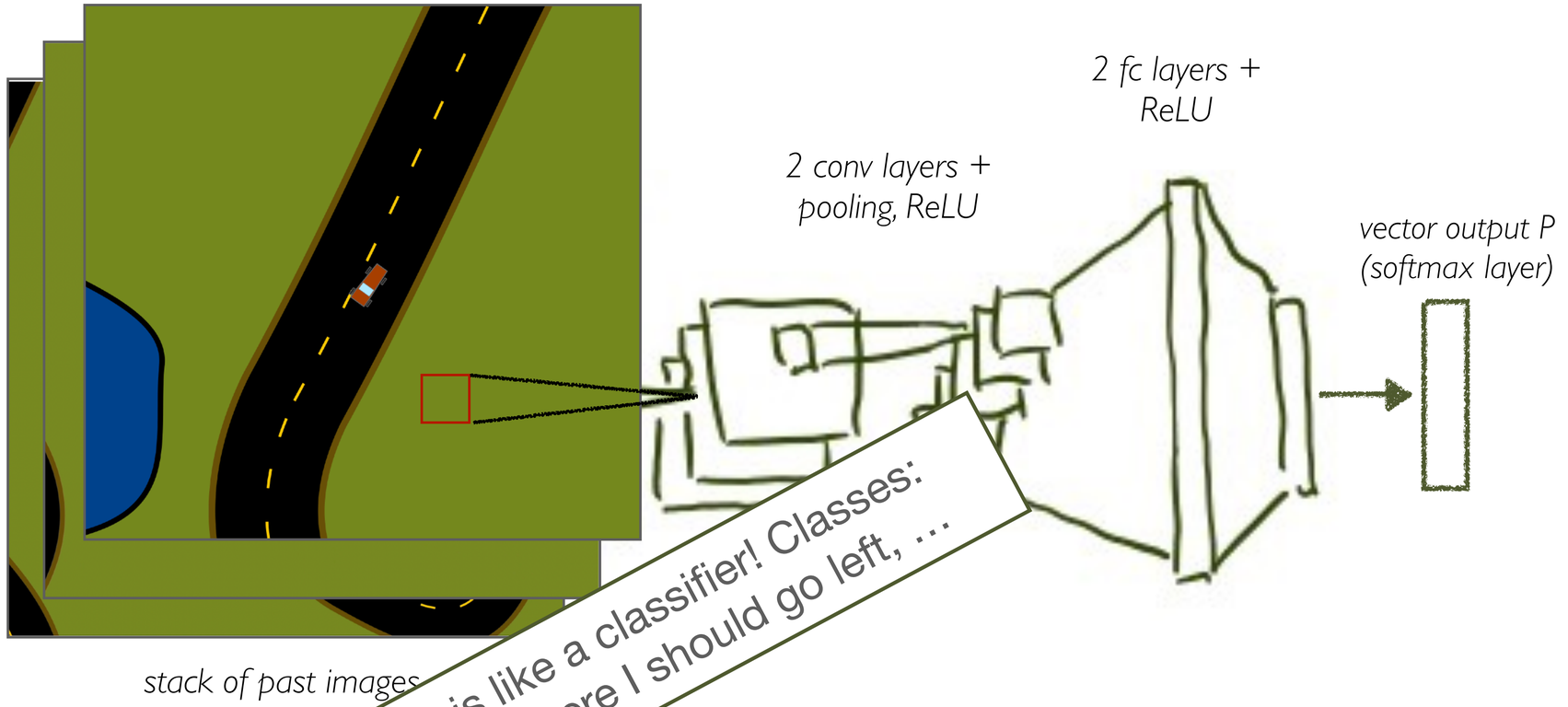
- Policy is a model too: represents $P(a \mid s, \pi)$
  - note: stochastic! (lets an optimizer make small changes)
- Several common ways to set up:
  - $s, a \mapsto P(a \mid s, \pi)$

Policy



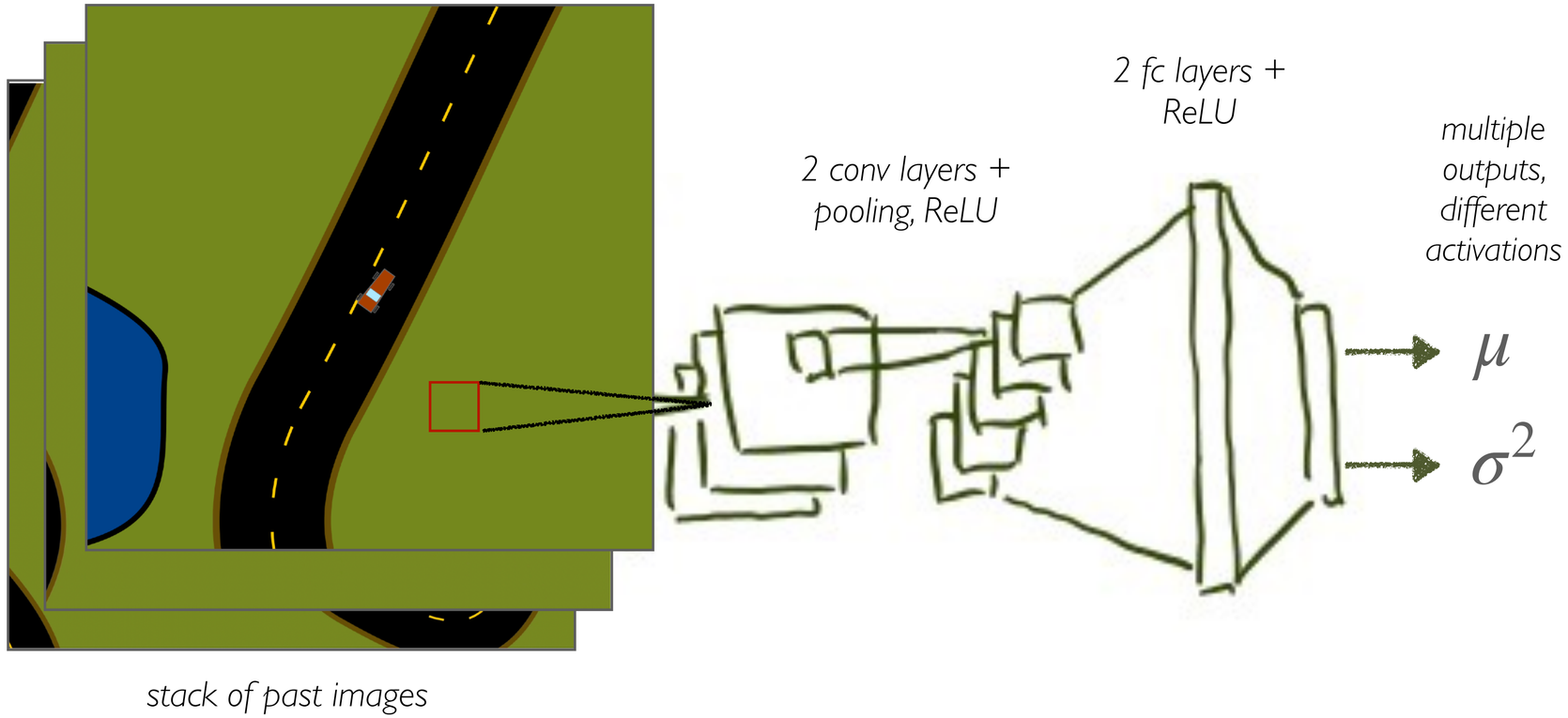
- Policy is a model too: represents $P(a \mid s, \pi)$
  - note: stochastic! (lets an optimizer make small changes)
- Several common ways to set up:
  - $s \mapsto [P(a_1 \mid s, \pi), P(a_2 \mid s, \pi), \ldots, P(a_k \mid s, \pi)]^\top$

Policy



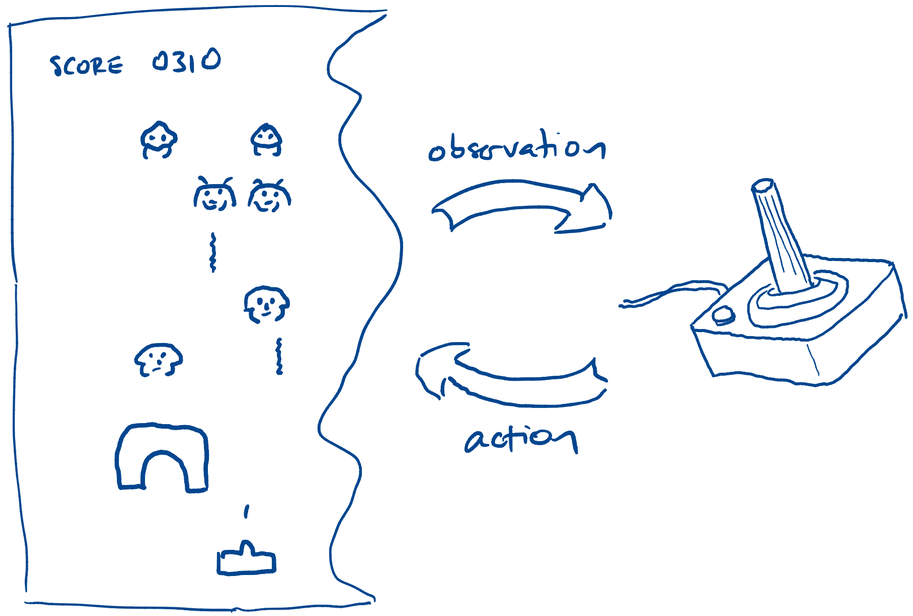
- Policy is a m[illegible]: represents $P(a \mid s, \pi)$
  - note: stochastic! (lets an optimizer make small changes)
- Several common ways to set up:
  - $s \mapsto [P(a_1 \mid s, \pi), P(a_2 \mid s, \pi), \ldots, P(a_k \mid s, \pi)]^\top$

Policy



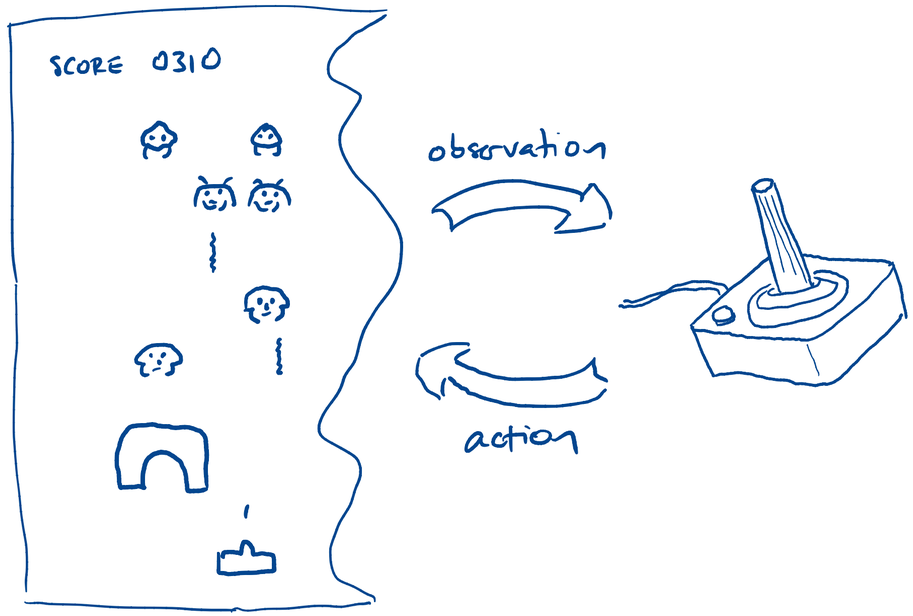
- Policy is a model too: represents $P(a \mid s, \pi)$
  - note: stochastic! (lets an optimizer make small changes)
- Several common ways to set up:
  - $s \mapsto$ parameters of action distribution like mean, variance

State vs. observation



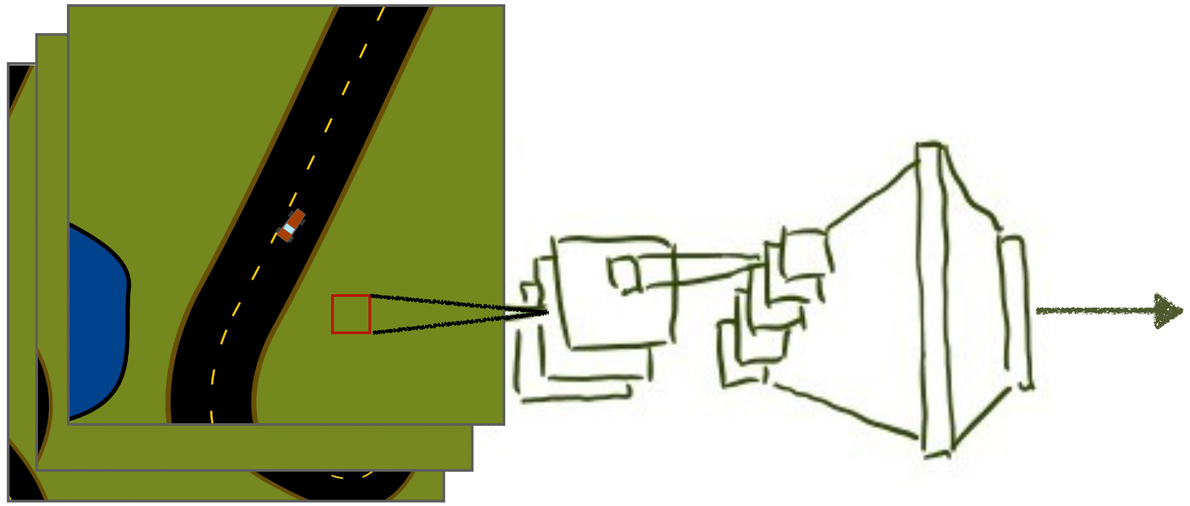
- Agent doesn't see state directly: $s_t \neq o_t$, common mistake!
  - observation informs about state: e.g., screen image → position
  - but often need to fuse information from several o_t: e.g., velocities
- Terminology: *fully/partially observable*

State vs. observation



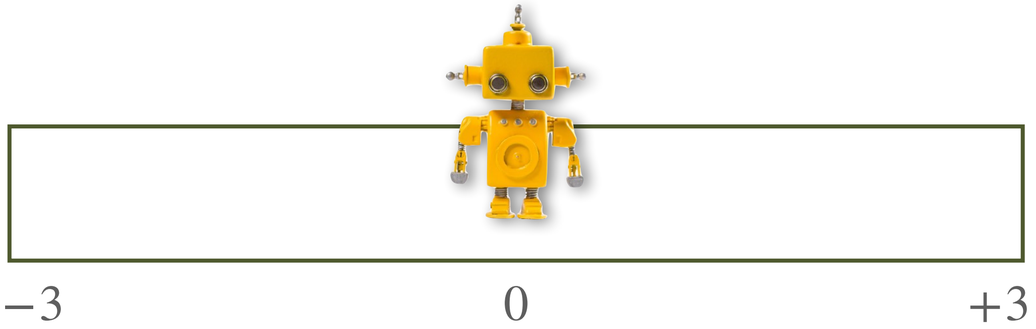
- What do we do if we don't know s_t?
- Simplest approach: network implicitly figures out the state from its input (e.g., stack of images) — not always possible
  - lots of more complicated approaches, but not in this class
- Assume this approach: a trajectory is now $s_1, a_1, r_1, s_2, \ldots$
  - each s_t is ***sufficient info for network to reconstruct state***
  - e.g., stack of past observations and actions

Learning V^π



- Want to train a network $V^\pi_\phi(s)$ w/ parameters ϕ
  - inputs: state info, e.g., stack of images
  - output: value estimate
- Data: follow π, observe one or more trajectories $s_1, a_1, r_1, s_2, a_2, r_2, \ldots$
- Each trajectory yields several training examples

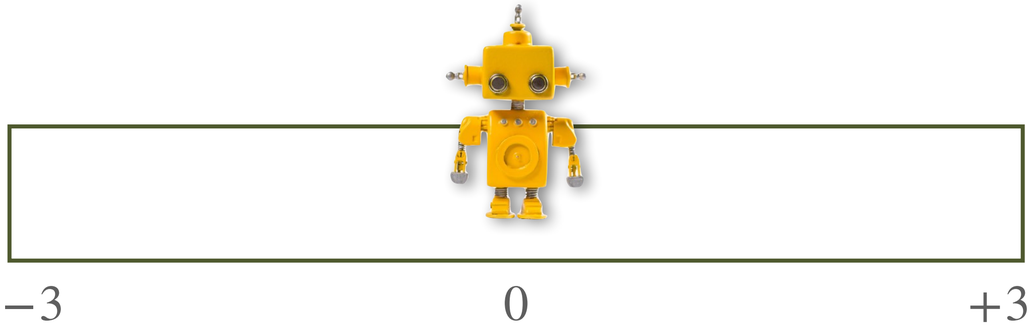
Learning V^π: example



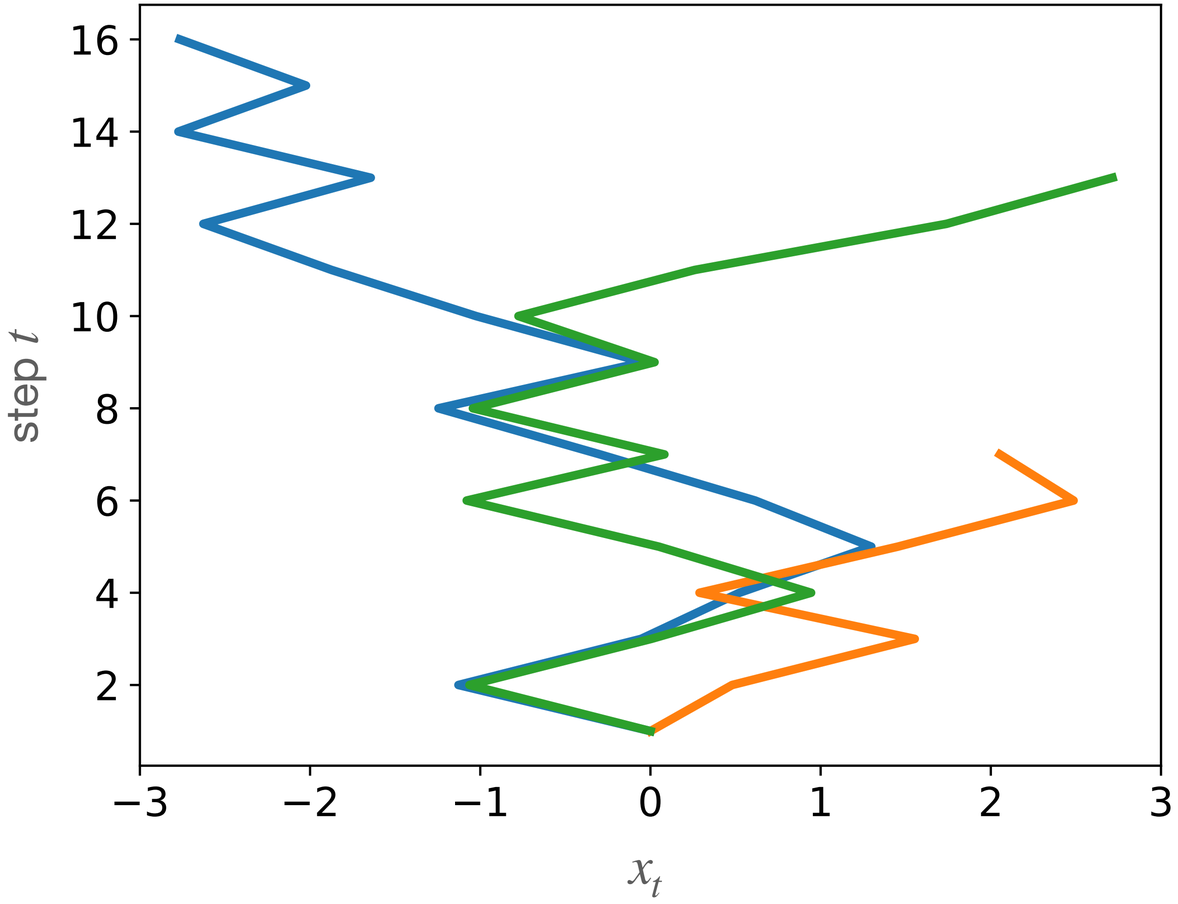
- Environment:
  - state $x \in [-3,3]$, start at $x = 0$
  - actions L: $x = x - N(1, \sigma^2)$ and R: $x = x + N(1, \sigma^2)$
  - rewards: –1 per action, terminate when $x \notin [-3, 3]$
  - $\gamma = 1, \sigma = \frac{1}{4}$

Some sample trajectories

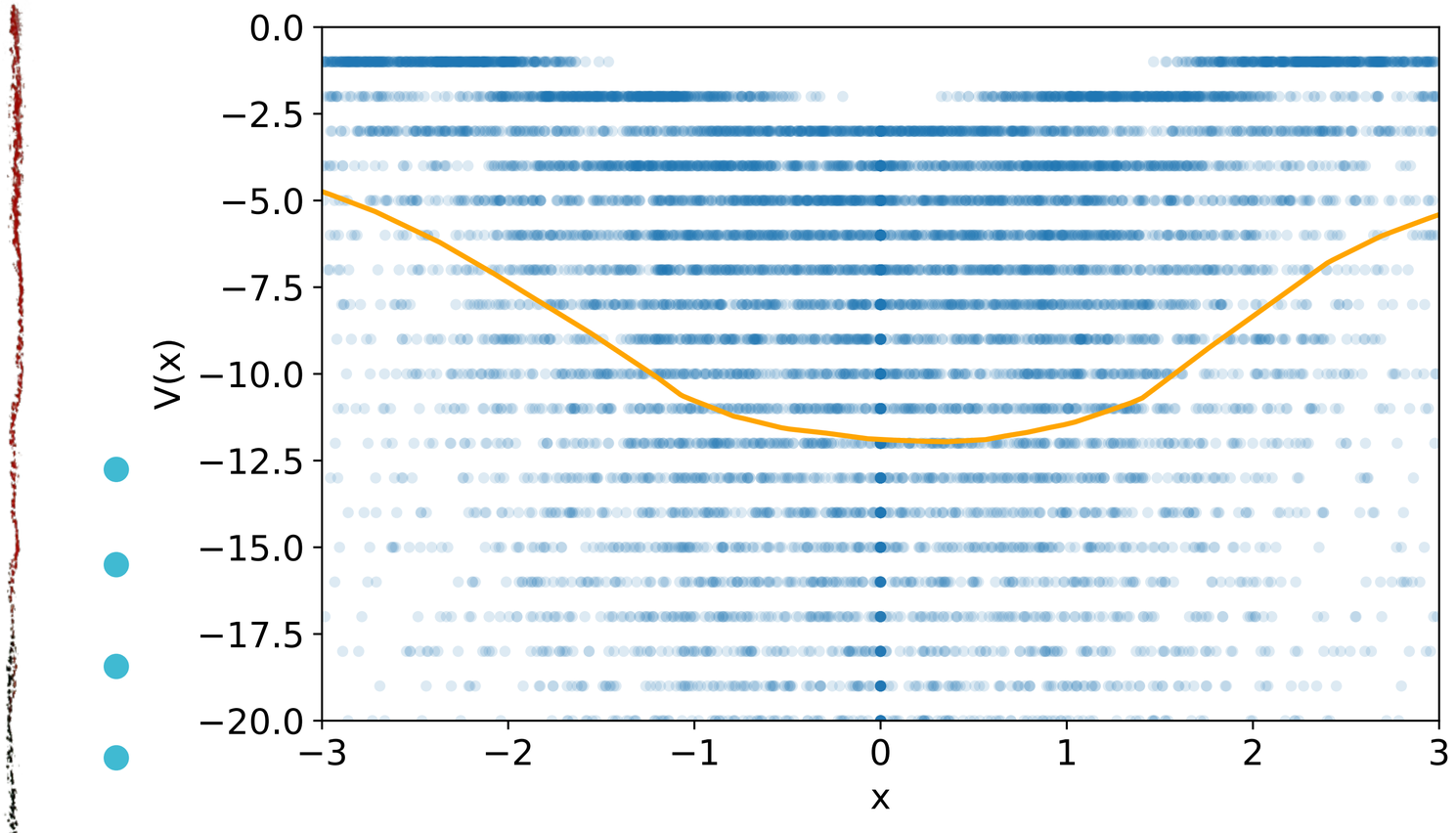
Each trajectory yields several training examples



(x, V(x))
(0, −16)



Learned V^π



;)