

Scaling up RL



*10-701 Introduction to Machine Learning
Geoff Gordon and Pradeep Ravikumar*

Greedy policy

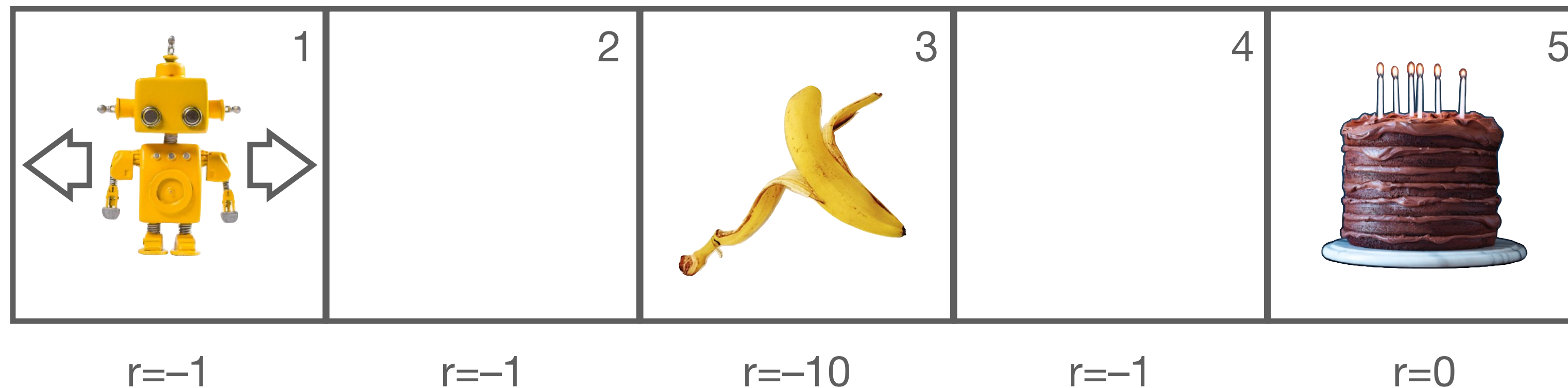
- If we have V^π or V^* , helps us design an agent
- Look at RHS of Bellman: $r(s, a) + \sum_{s'} T(s' | s, a) V(s')$
 - ▶ if agent is at s , this is predicted total future reward for choosing action a
- **Greedy** policy: choose a to maximize this predicted reward (separately at each state)
 - ▶ a one-step lookahead search to choose best action
 - ▶ also makes sense to look farther ahead: k -step greedy
- **Theorem:** the greedy policy for V^π is always at least as good as π — and strictly better unless π is already optimal (i.e., unless $V^\pi = V^*$)
- Motivates algorithms that estimate V^π , improve π , repeat

simplest version: **policy iteration**

Q function

Q^π or Q^ makes it trivial to compute the greedy policy — don't need to know $r(s, a)$ or $T(s' | s, a)$*

Policy π : always move right



Q^π

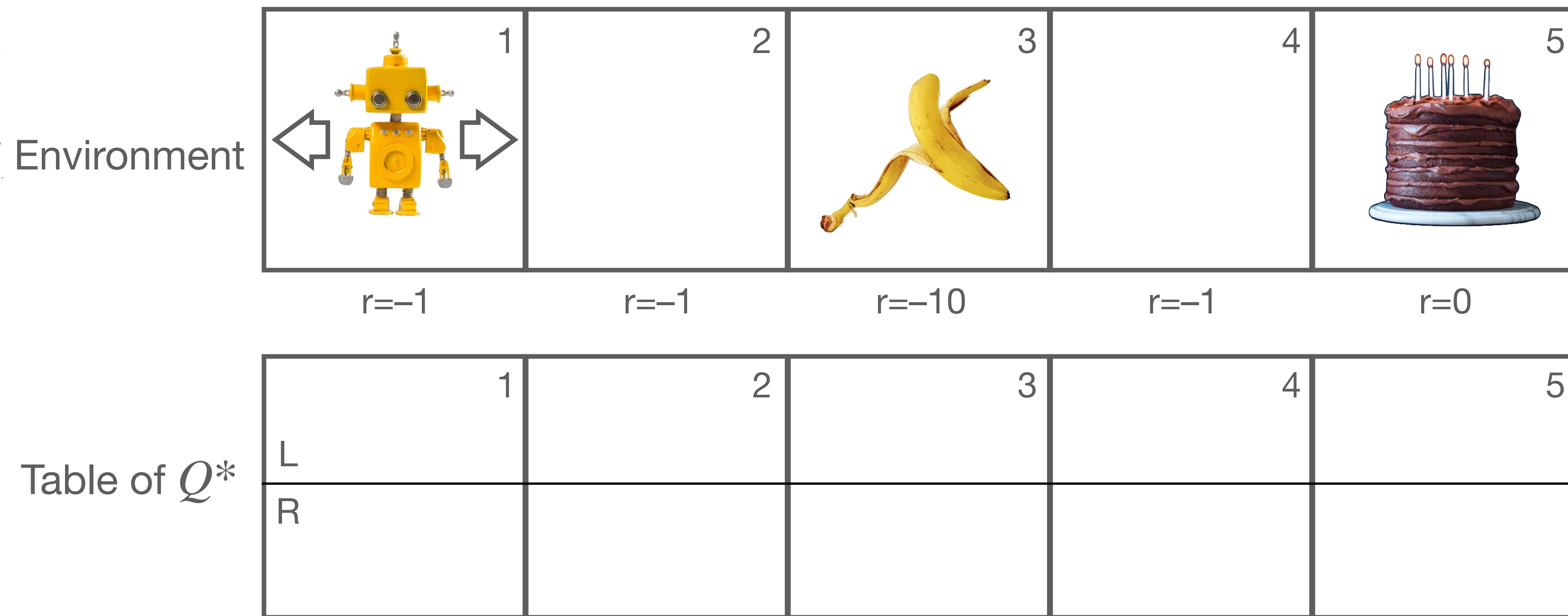
	1	2	3	4	5
L	-14	-14	-13	-12	0
R	-13	-12	-11	-1	0

- **Def'n:** $Q(s, a) = r(s, a) + \sum_{s'} T(s' | s, a) V(s')$
 - ▶ $Q^\pi(s, a)$ or $Q^*(s, a) =$ how much total reward if we start at $s_1 = s$, choose $a_1 = a$, continue with π or π^*
 - ▶ precalculated one-step lookahead on V
- Called the Q function or action-value function

Dynamic programming for Q-function

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

Q-iteration



- ▶ Given: $r(s, a), T(s' | s, a)$
- ▶ 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)

note base case:
 $Q(\text{next}, \cdot) = 0$ if done

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

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

Q-learning

same idea, but from data instead of from environment model

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

note base case: important to train $Q^*(5, L/R)$ toward 0

Table of Q^*

	1	2	3	4	5
L	0	0	0	0	0
R	0 -0.75 0	0 -0.5	0 -5	0	0

- ▶ 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

```
loss += 0.5 * (Q[s, a] - target)**2;
loss.backward(); optim.step()
```

note base case: important to
train $Q^\pi(5, L/R)$ toward 0

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

Table of Q^π

	1	2	3	4	5
L					
R					

SARSA

*uses actual next
action instead of
sum or max over
next action*

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

note base case: important to
train $Q^\pi(5, L/R)$ toward 0

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

Table of Q^π

	1	2	3	4	5
L					
R					

SARSA

*uses actual next
action instead of
sum or max over
next action*

- ▶ Given: observed transitions from π , $\{s_t, a_t, r_t, s_{t+1}, a_{t+1}\}$
- ▶ Initialize $Q^\pi(s, a)$ arbitrarily (e.g., to 0)
- ▶ Repeat until converged
 - ▶ sample a transition s, a, r, s', a' (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 $r_1 + \gamma r_2 + \gamma^2 r_3 + \dots$
 - ▶ “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

Exploration

- Dynamic programming does temporal credit assignment
- For explore-exploit: need to intentionally collect data in data-poor areas of environment
- Example strategy: exploration bonus (*R-max* algorithm)
 - ▶ if enough data that we're ϵ -confident in rewards and transitions for s, a , learn model as usual
 - ▶ else, pretend s, a is utopia: big reward, trajectory ends
 - ▶ run value iteration (or ...) and follow greedy policy
 - ▶ if s, a is reachable w/ reasonable cost and time, we'll go there (and therefore collect more data)
 - ▶ else s, a didn't matter for optimal policy

[Brafman & Tenenbholz]

The end of the world

- Sometimes task ends: win or lose the game, reach the goal
 - ▶ learn over a sequence of trials

- Several possible setups for the end of the world:

- ▶ Always same number of steps: *fixed-horizon*

$$\min_{\text{policy } \pi \in \Pi} \mathbb{E}_{\text{trajectory } \tau | \pi} [c_1 + c_2 + \dots + c_T]$$

- ▶ Task ends when we reach *terminal state*, all intermediate costs positive (rewards negative): *stochastic shortest paths (SSP)*

$$\min_{\text{policy } \pi \in \Pi} \mathbb{E}_{\text{trajectory } \tau | \pi} [c_1 + c_2 + \dots + c_{T(\tau)}]$$

- ▶ Task continues w.p. $\gamma \in [0, 1)$ after each time step: *discounted*

$$\min_{\text{policy } \pi \in \Pi} \mathbb{E}_{\text{trajectory } \tau | \pi} [c_1 + \gamma c_2 + \gamma^2 c_3 + \dots]$$

- World never ends: *infinite-horizon*

- ▶ Must do something to prevent infinite costs/rewards: *discounted or average*

The end of the world

- Sometimes task ends: win or lose the game, reach the goal
 - ▶ learn over a sequence of trials

- Several possible setups for the end of the world

- ▶ Always same number of steps: *fixed-horizon*

$$\min_{\text{policy } \pi \in \Pi} \mathbb{E}_{\text{trajectory } \tau \sim \pi} [c_1 + c_2 + \dots + c_T]$$

- ▶ Task ends when we reach a goal state, all intermediate costs positive (rewards negative): *stochastic shortest paths (SSP)*

$$\min_{\text{policy } \pi \in \Pi} \mathbb{E}_{\text{trajectory } \tau \sim \pi} [c_1 + c_2 + \dots + c_{T(\tau)}]$$

- ▶ Task continues forever, discount factor $\gamma \in [0, 1)$ after each time step: *discounted*

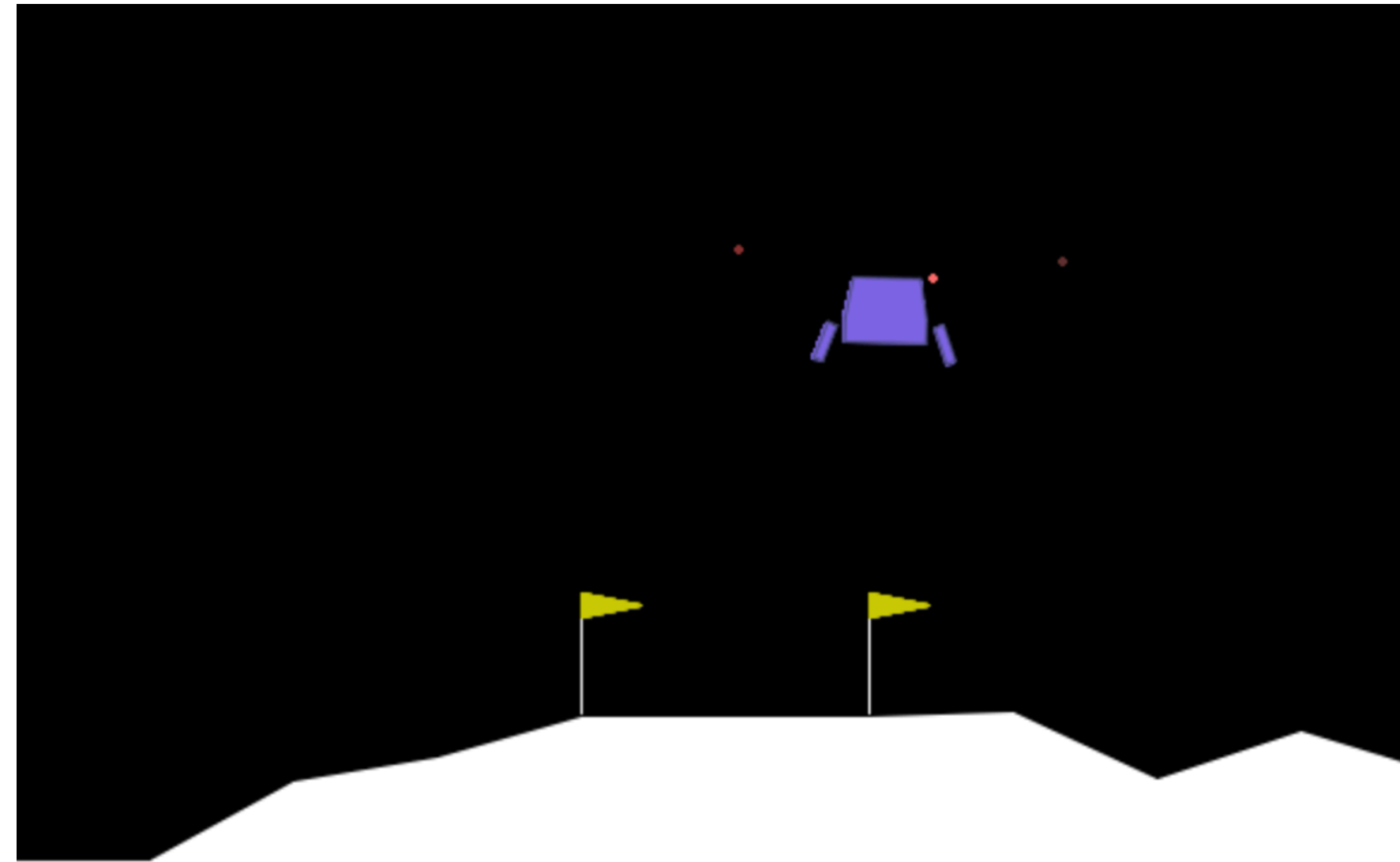
$$\min_{\text{policy } \pi \in \Pi} \mathbb{E}_{\text{trajectory } \tau \sim \pi} [c_1 + \gamma c_2 + \gamma^2 c_3 + \dots]$$

- World never ends: *infinite-horizon*

- ▶ Must do something to prevent infinite costs/rewards: *discounted or average*

All setups are fine — but important to pick one! (common source of bugs)

Scaling up RL



Lunar lander

Game 1
Fan Hui (Black), AlphaGo (White)
AlphaGo wins by 2.5 points

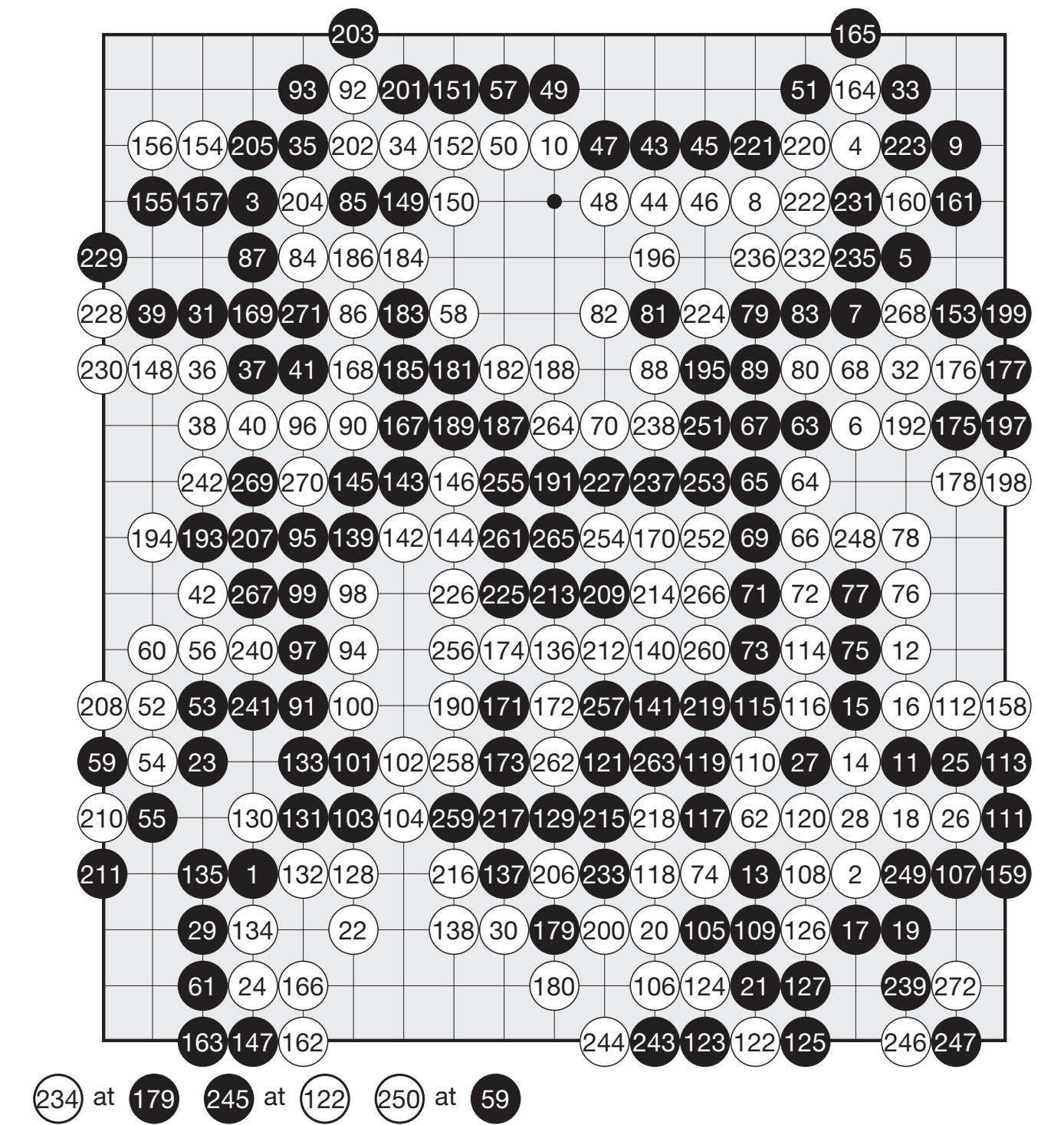
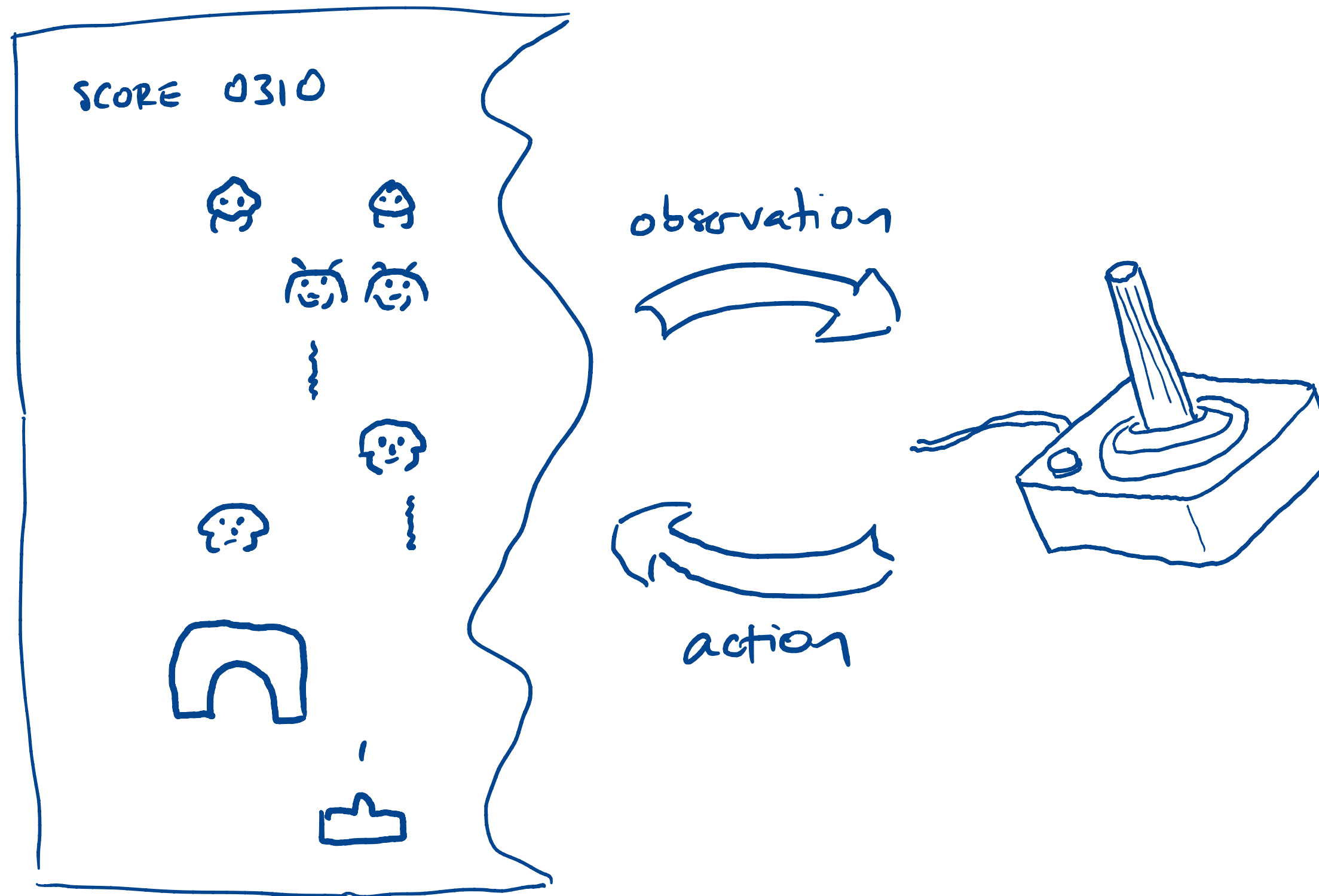


Image credit: Silver et al., Nature, 2016

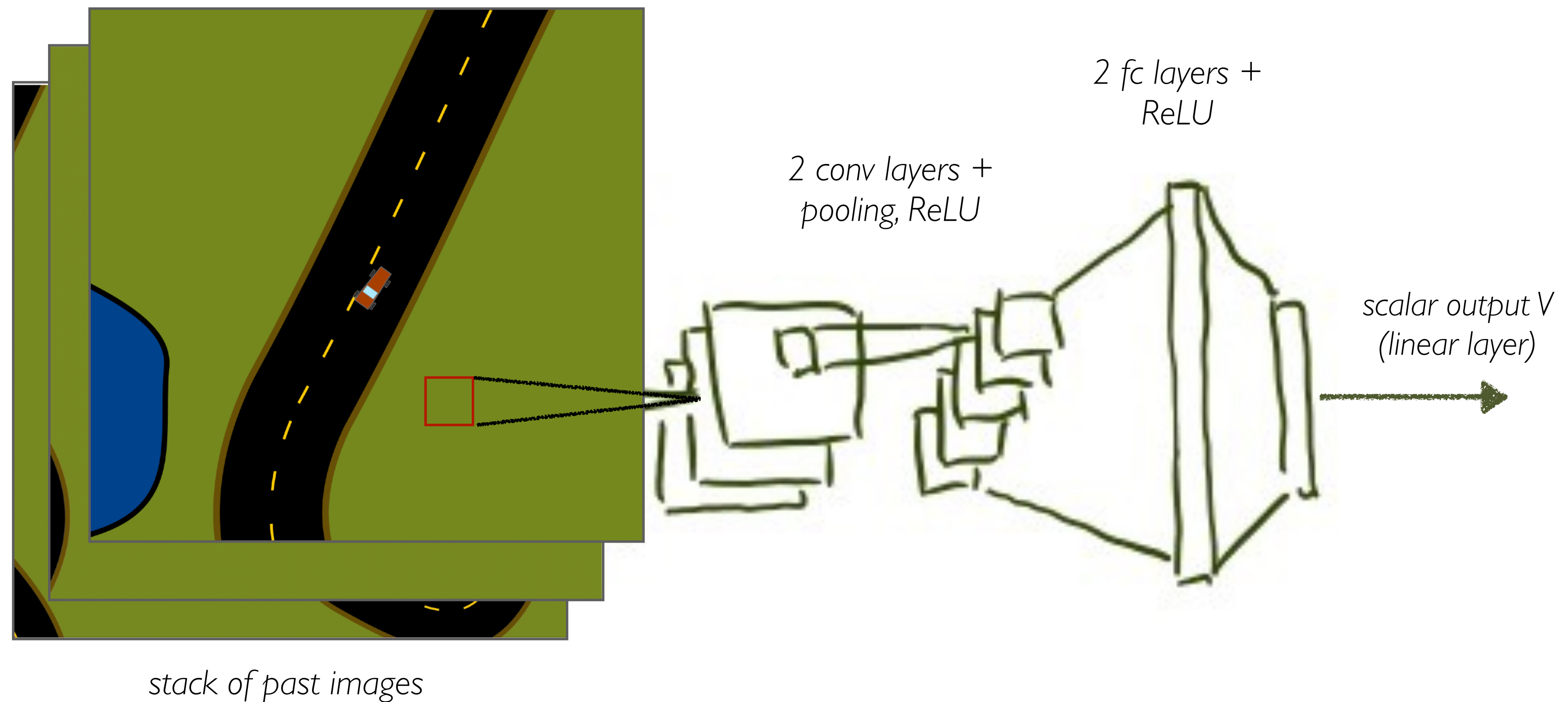
- To go beyond tabular, need some changes in our setup

No exact model



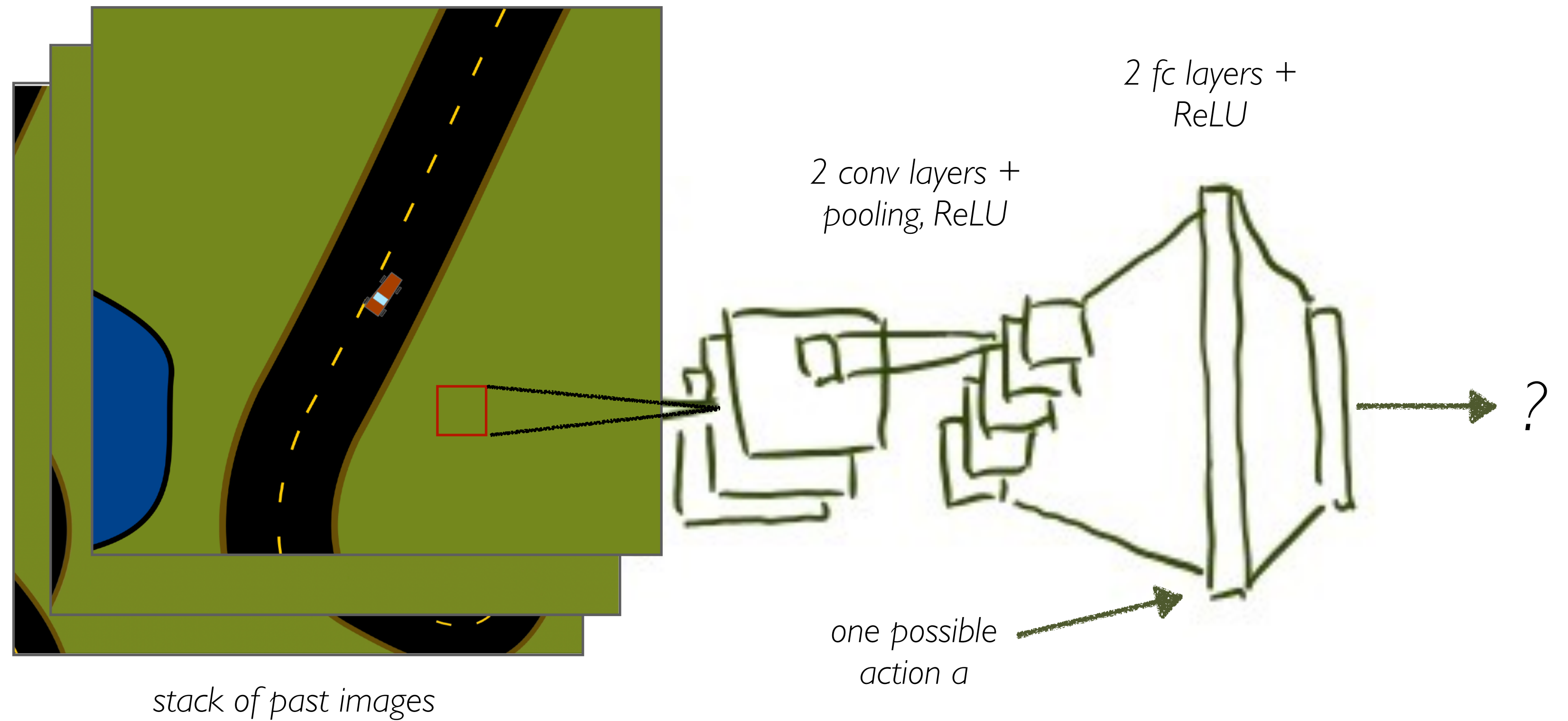
- Previously: we could write down a description of the world, e.g., expressions for $r(s, a)$ or $T(s' | s, a)$
- Instead: agent just interacts with environment over time — if we want $r(s, a)$ etc., have to learn it from data
- Learn from trajectories $o_1, a_1, r_1, o_2, a_2, r_2, \dots$

Learned, approximate functions



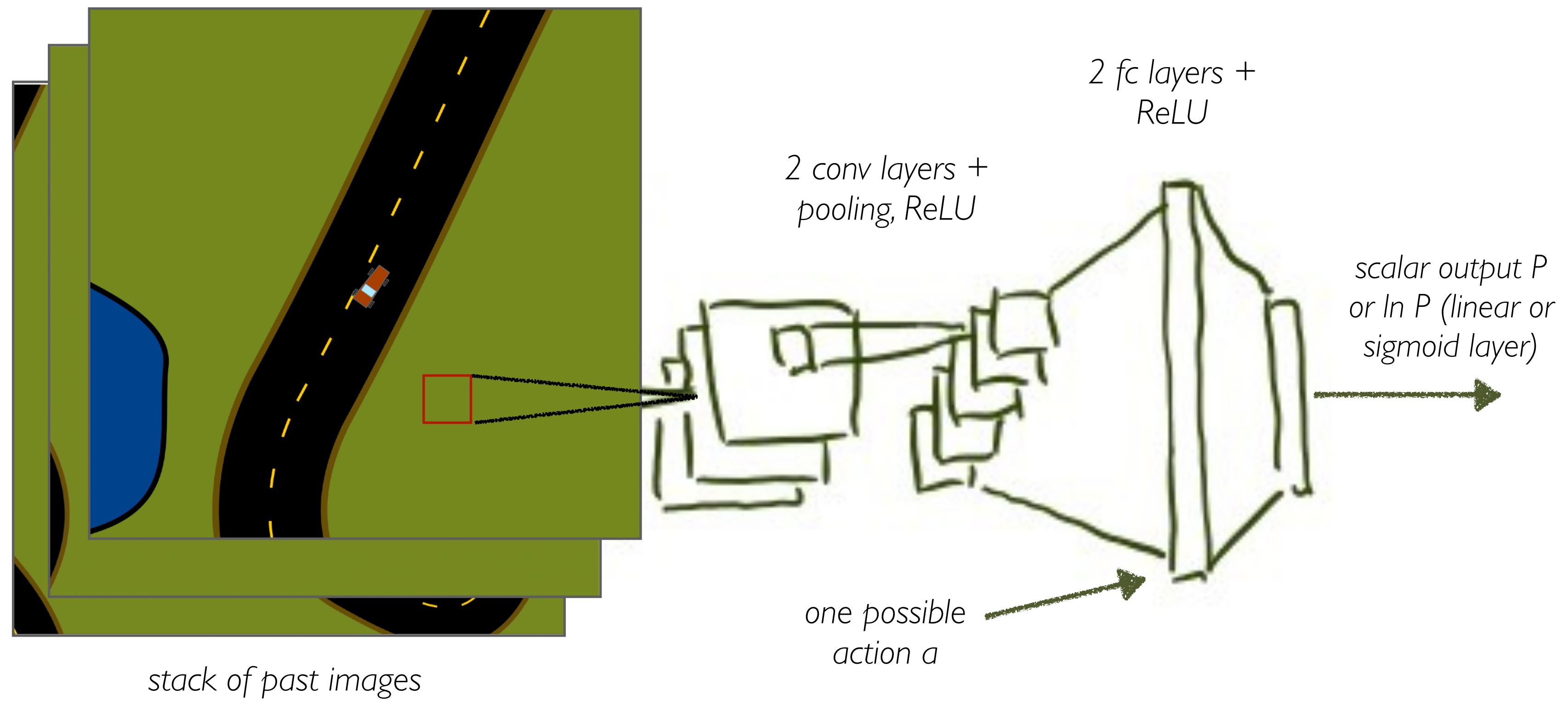
- Previously: could list out all states, keep a table of a function like $V^\pi(s)$
- Now: any function we care about has to be represented as an ML model, e.g., a deep net
- One parameter vector per function, each can have its own network architecture

Policy



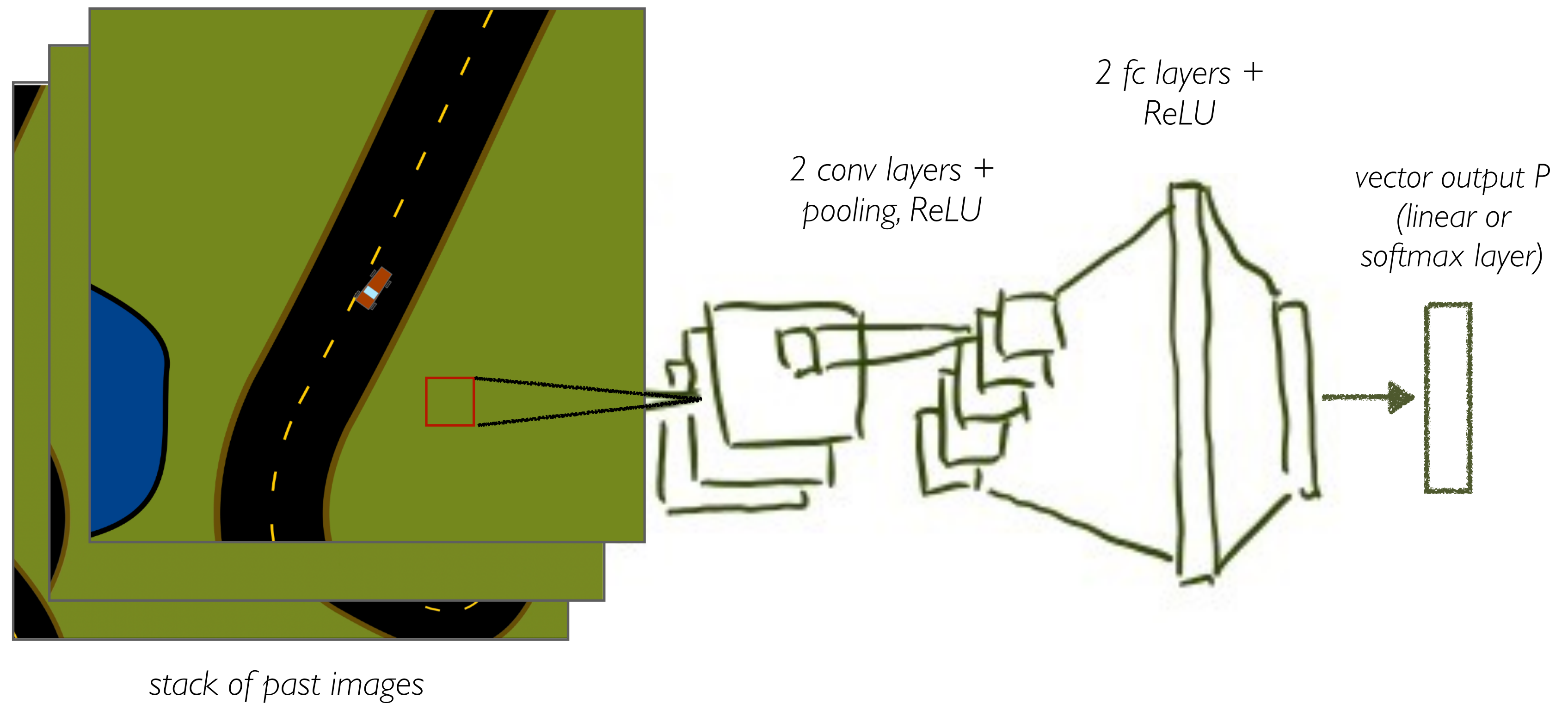
- V and Q are straightforward to represent as models: regression $\mathcal{S} \rightarrow \mathbb{R}$ or $\mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$
- Policy is a model too: represents $P(a \mid s, \pi)$
 - ▶ recall: stochastic! (lets optimizer make small changes)
- Several common ways to set up

Policy



- Policy network could be:
 - ▶ $s, a \mapsto P(a \mid s, \pi)$ or $\ln P(a \mid s, \pi)$

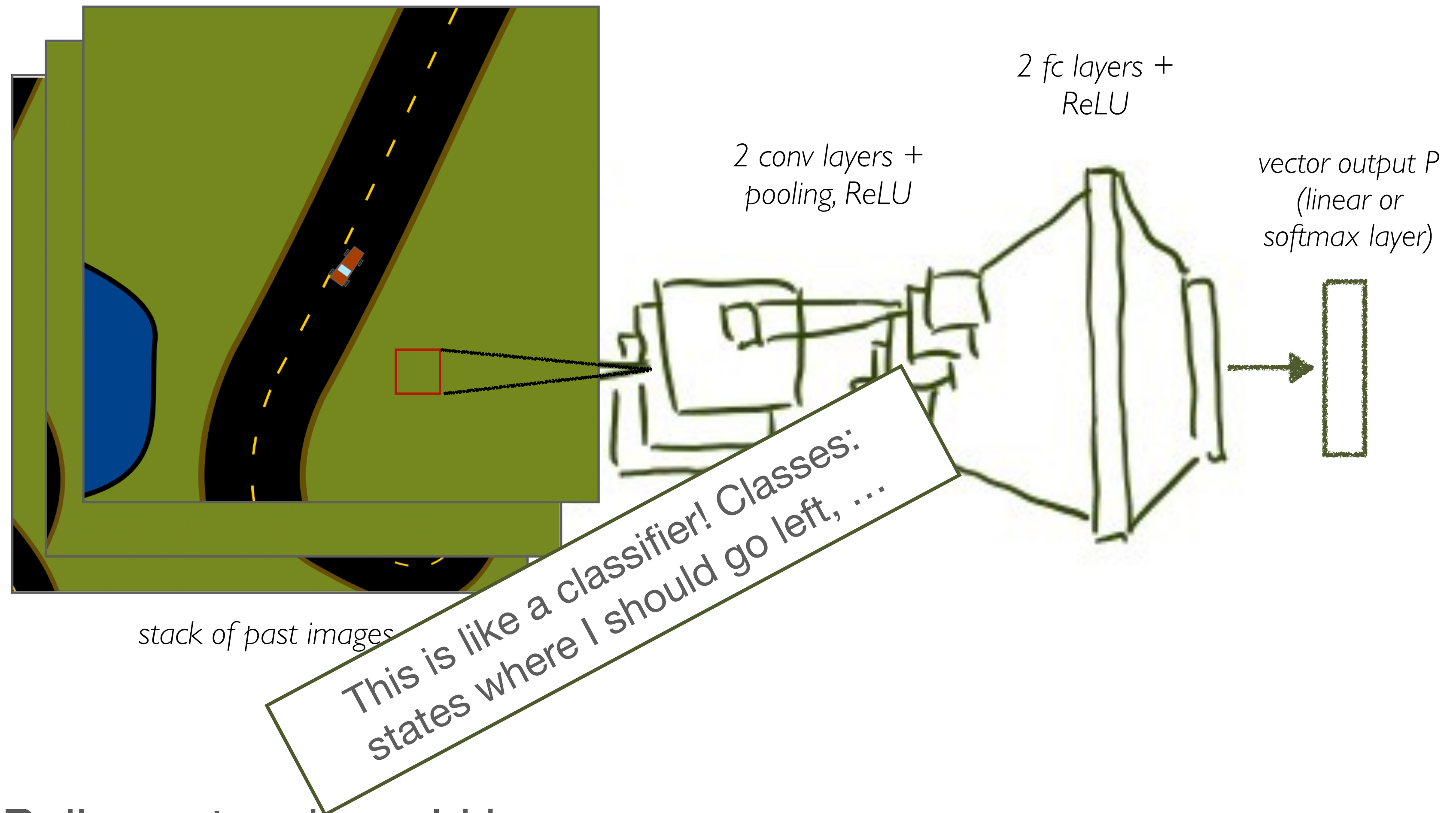
Policy



- Policy network could be:

- ▶ $s \mapsto [P(a_1 | s, \pi), P(a_2 | s, \pi), \dots, P(a_k | s, \pi)]^T$ (or logs)

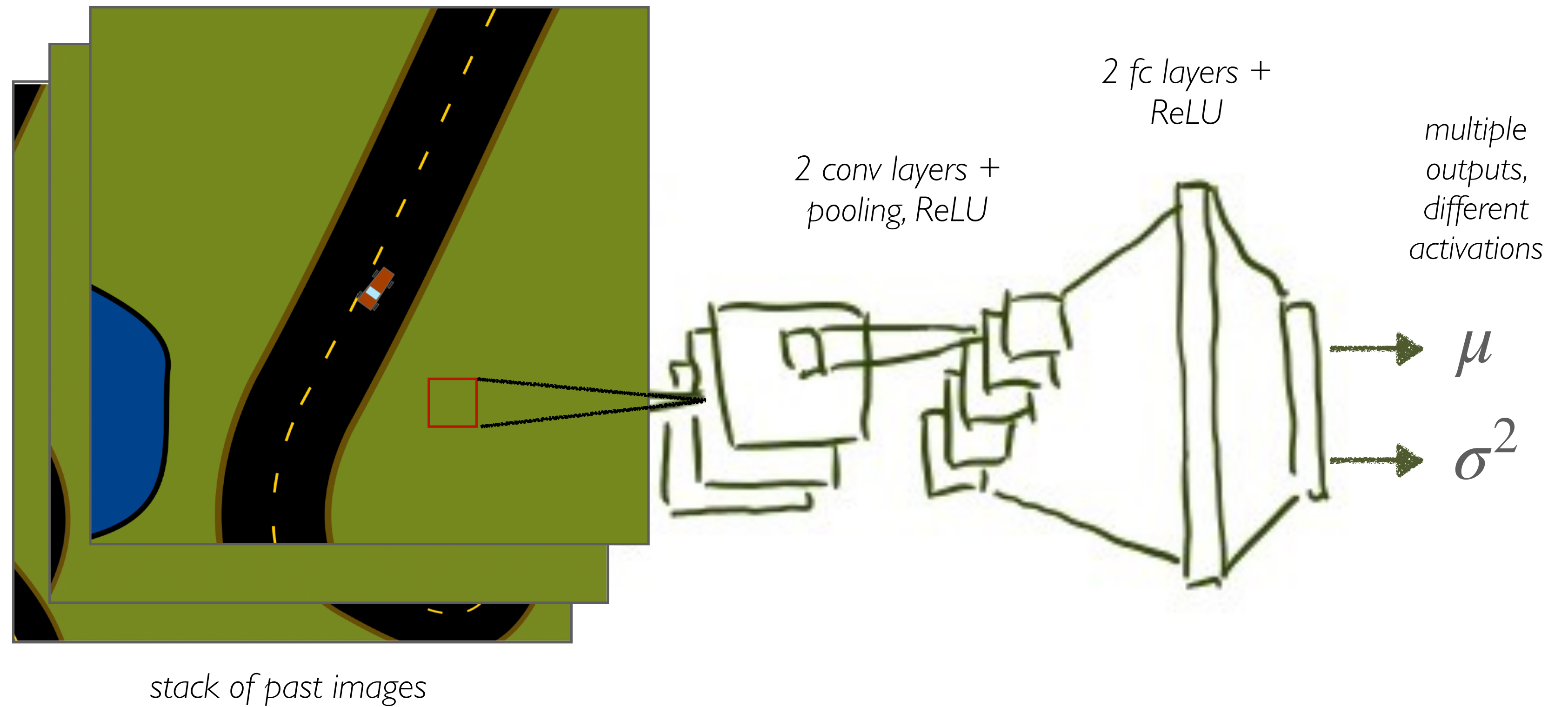
Policy



- Policy network could be:

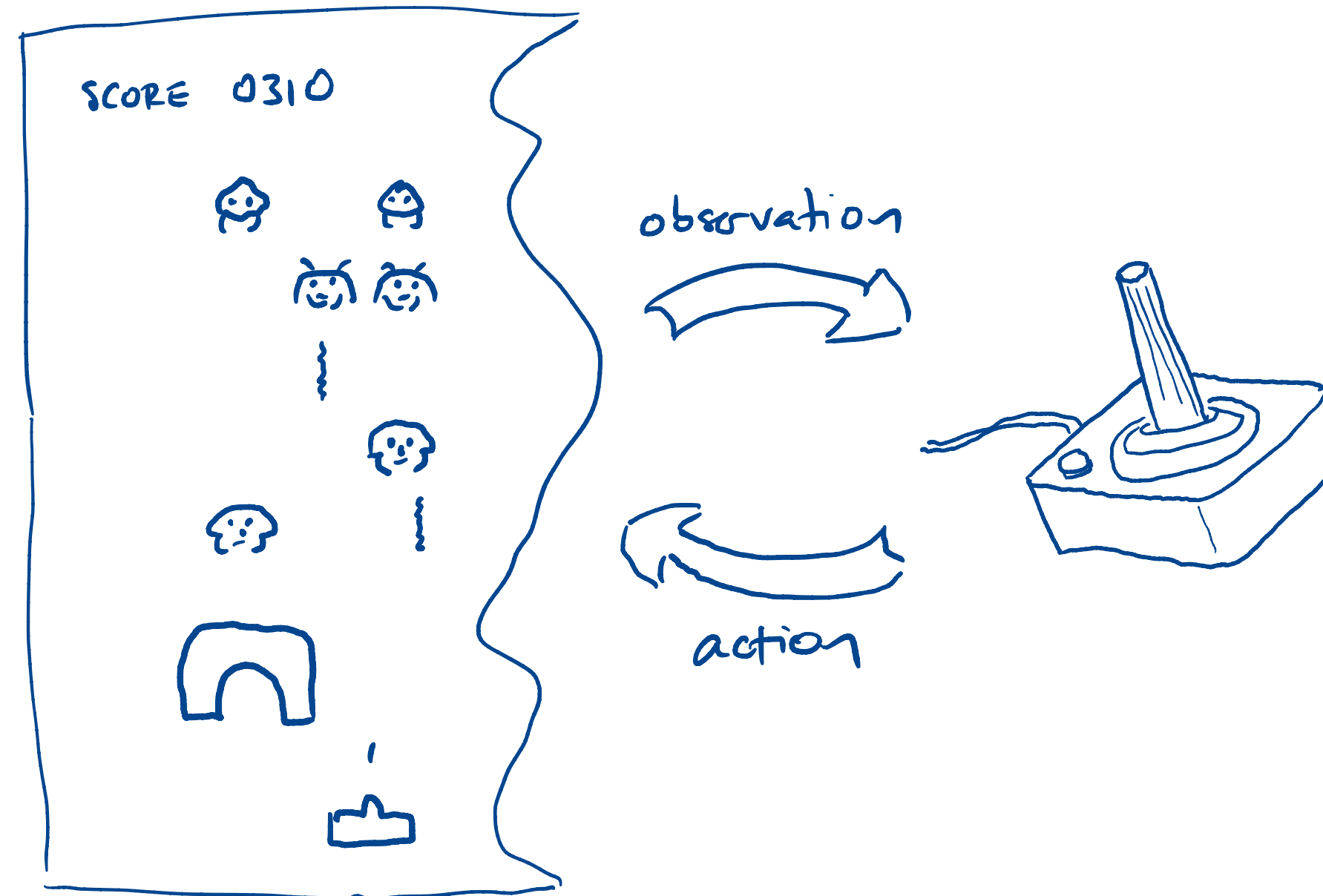
- ▶ $s \mapsto [P(a_1 | s, \pi), P(a_2 | s, \pi), \dots, P(a_k | s, \pi)]^T$ (or logs)

Policy



- Policy network could be:
 - ▶ $s \mapsto$ parameters of action distribution like mean, variance

State vs. observation

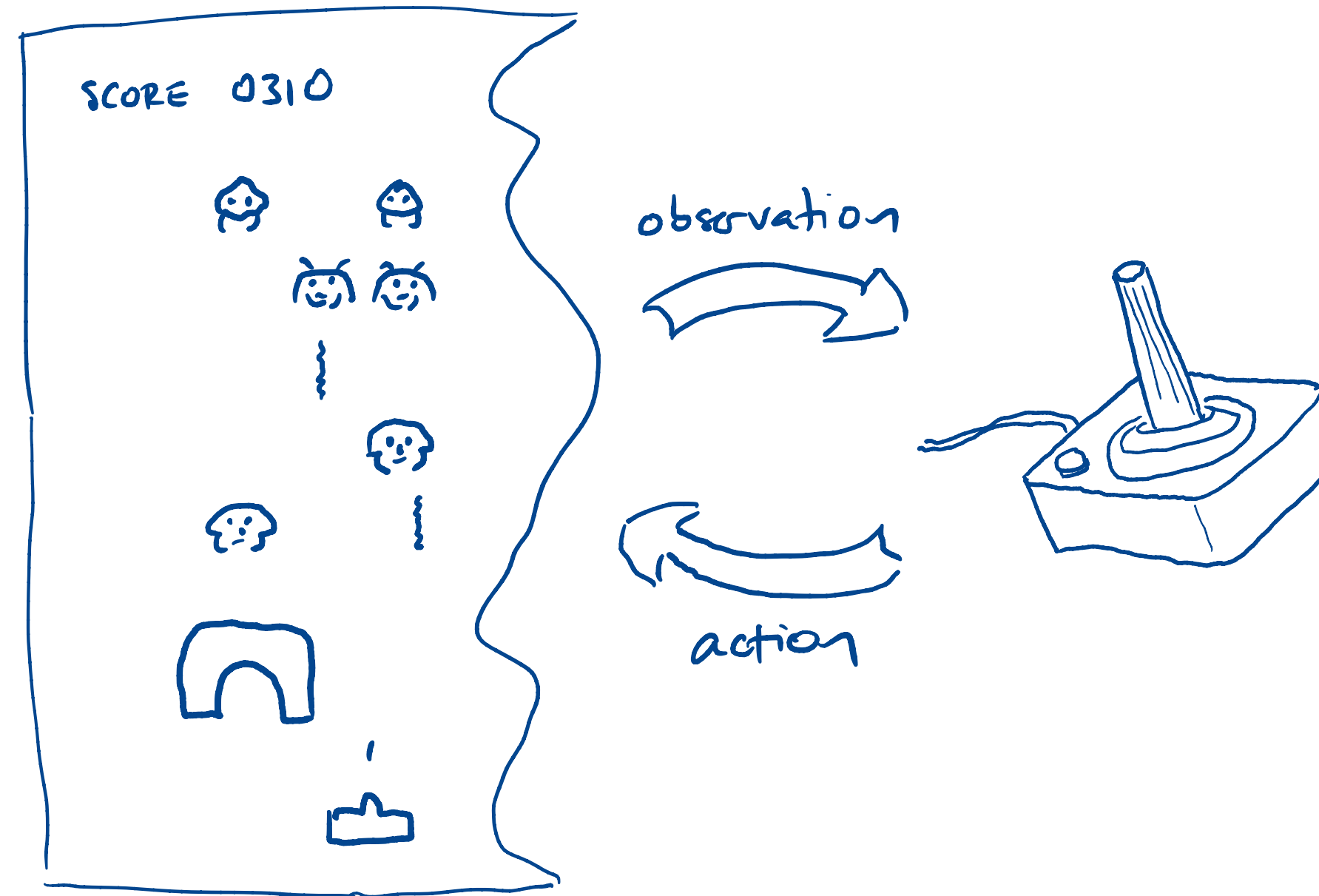


- As we scale up, more likely that agent doesn't see state completely and directly: $s_t \neq o_t$, common source of bugs!
 - ▶ instead, observation informs about state: e.g., screen images help determine position and velocity
 - ▶ often need to fuse information from several o_t : e.g., velocity
- Terminology: *fully/partially observable*

Can't simply copy tabular algorithms

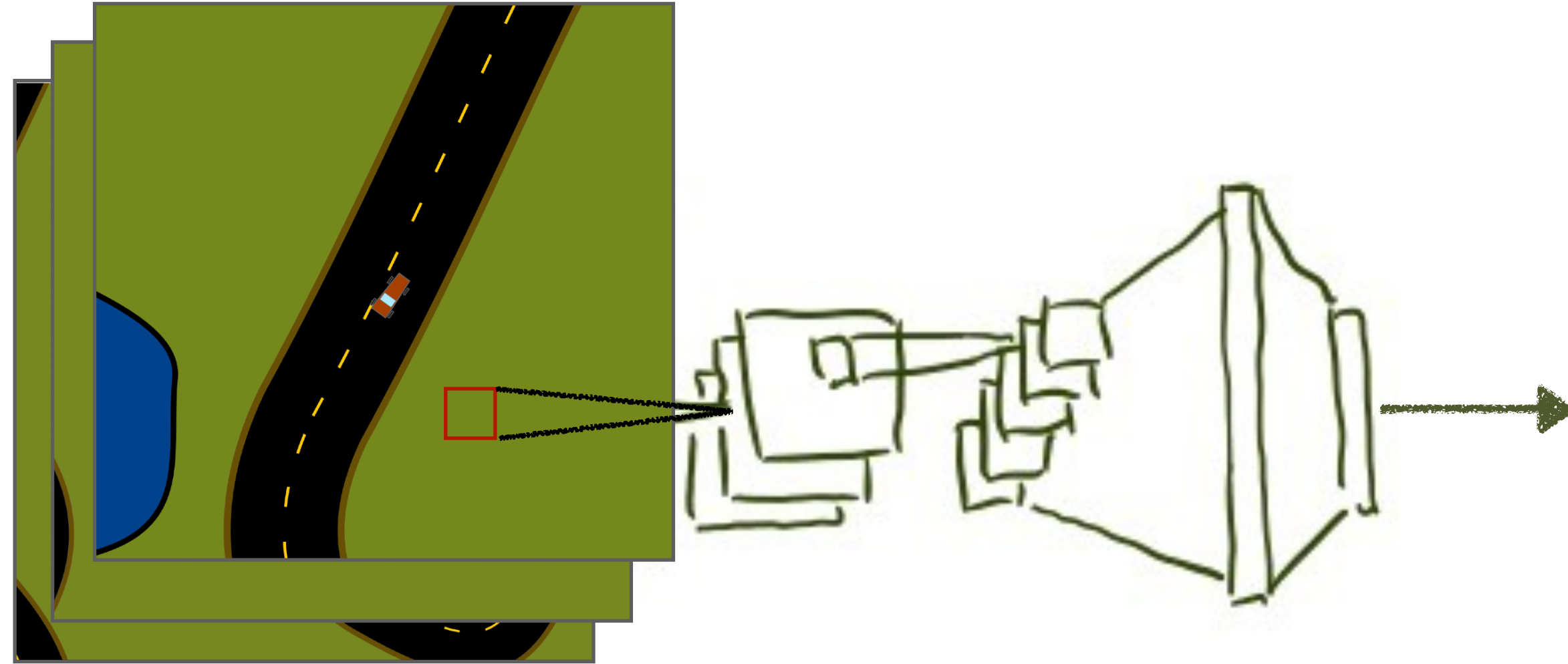
- We showed lots of dynamic programming algorithms for the tabular case — many **do not scale**
 - ▶ fail in weird ways when we use function approximation
- In general,
 - ▶ SGD-like policy evaluation algorithms (TD for V^π , SARSA for Q^π) still work if value = output of deep net
 - ▶ algorithms w/ max (Q-learning for Q^* , value iteration for V^*) no longer work well
 - ▶ can make them to do something by force of engineering, e.g., DQN — but other methods work better
- So what do we do instead?

State vs. observation



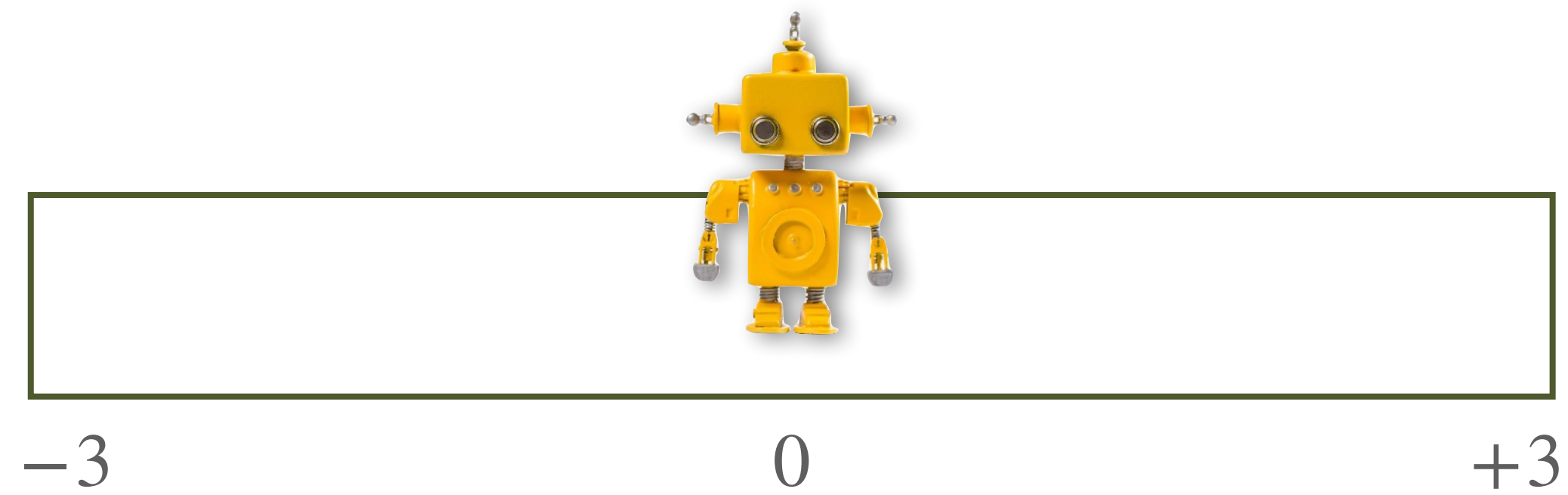
- What if we don't know s_t ?
- Simplest approach: network implicitly figures out state
 - ▶ suppose s_t is **sufficient info to reconstruct state**
 - ▶ e.g., stack of past observations & actions (images & controls)
 - ▶ in order to predict V^π , some hidden activations somewhere must be a state representation
- Doesn't always work, many fancier approaches — but not in 10-701

Learning V^π



- Want to train a network $V_\phi^\pi(s)$ w/ parameters ϕ
 - ▶ inputs: state info, e.g., stack of images
 - ▶ output: value estimate
- Simplest approach: regression on whole-trajectory returns
- Follow π , observe trajectories $s_1, a_1, r_1, s_2, a_2, r_2, \dots$
- Each trajectory yields several training examples

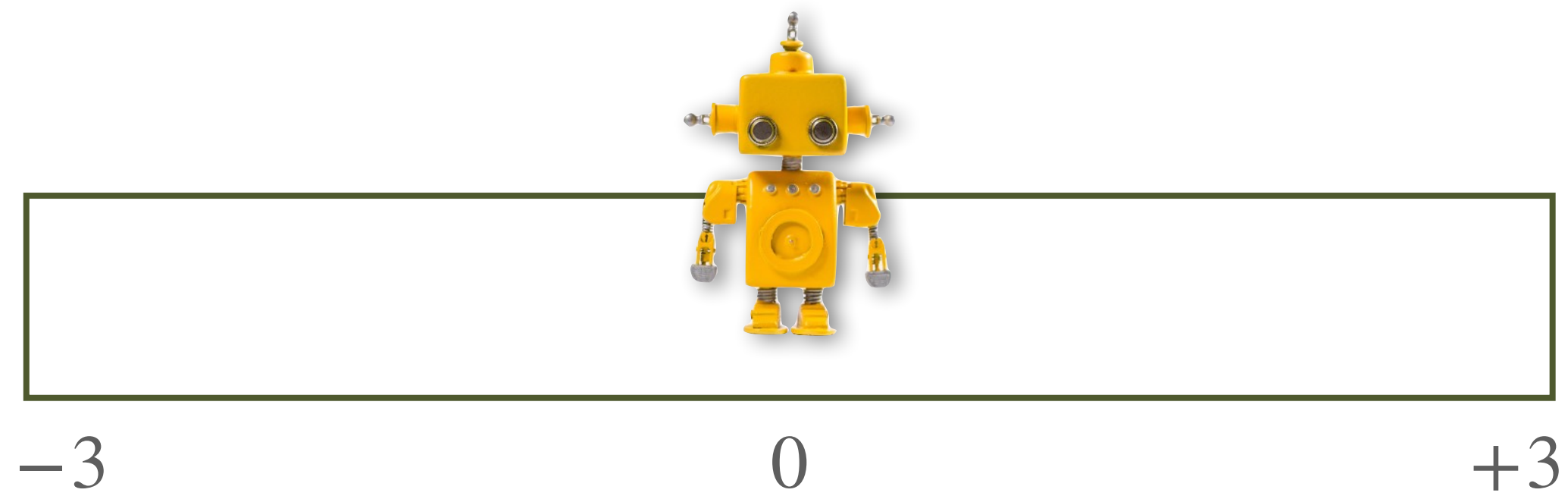
Learning V^π : example



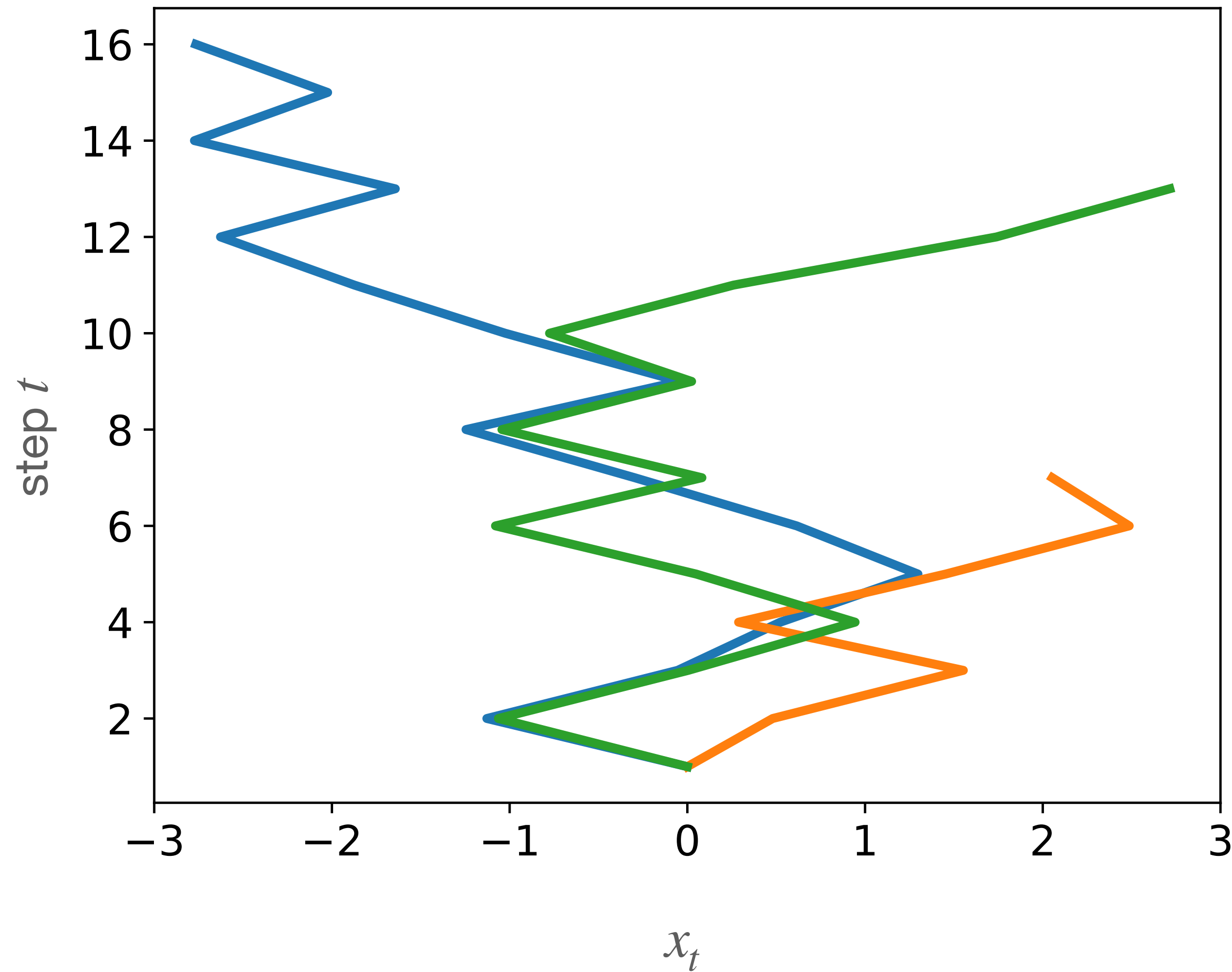
- Environment (continuous version of earlier example):
 - ▶ state $x \in [-3, 3]$, start at $x = 0$
 - ▶ actions L: $x = x - N(1, \sigma^2)$ and R: $x = x + N(1, \sigma^2)$
 - ▶ rewards: -1 per action, terminate when $x \notin [-3, 3]$
 - ▶ $\gamma = 1, \sigma = \frac{1}{4}$

Some sample trajectories

Each trajectory yields several training examples

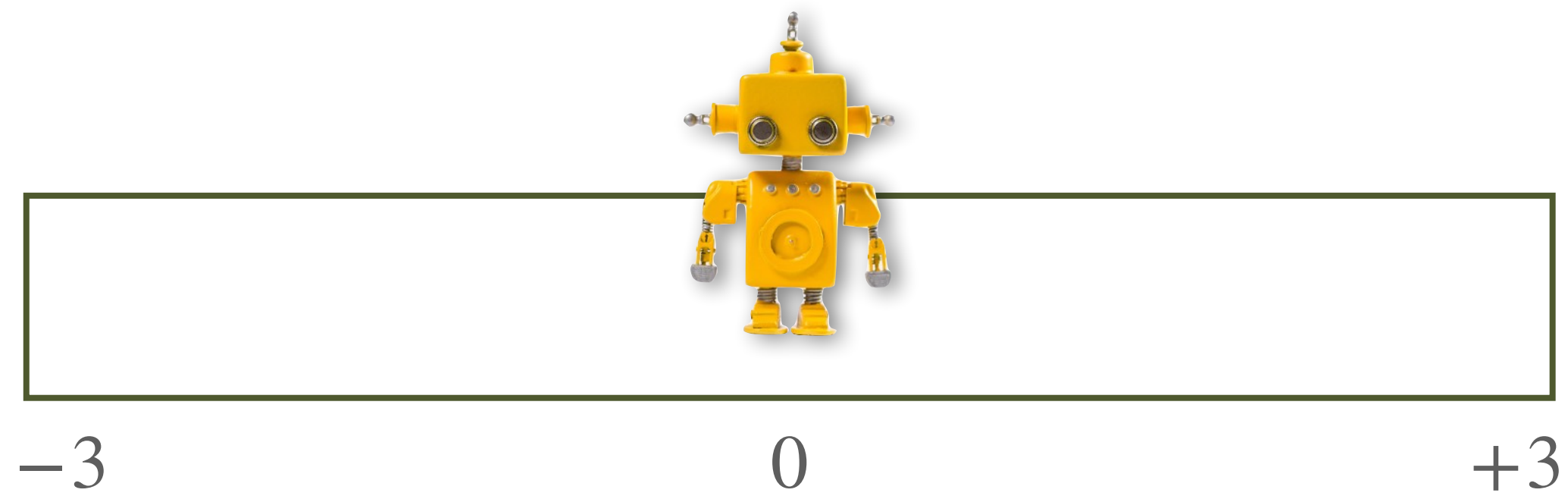


$(x, V(x))$
 $(0, -16)$

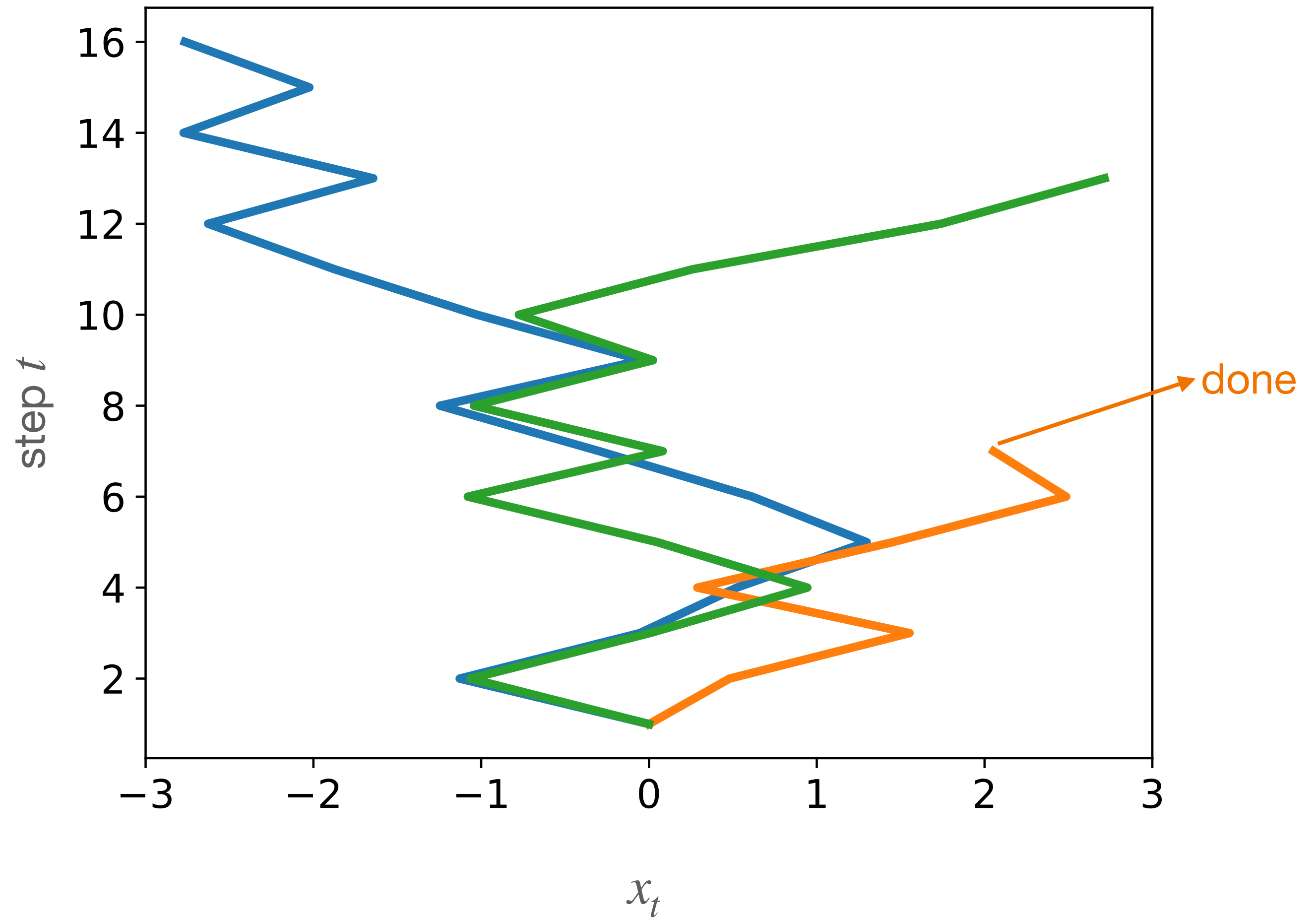


Some sample trajectories

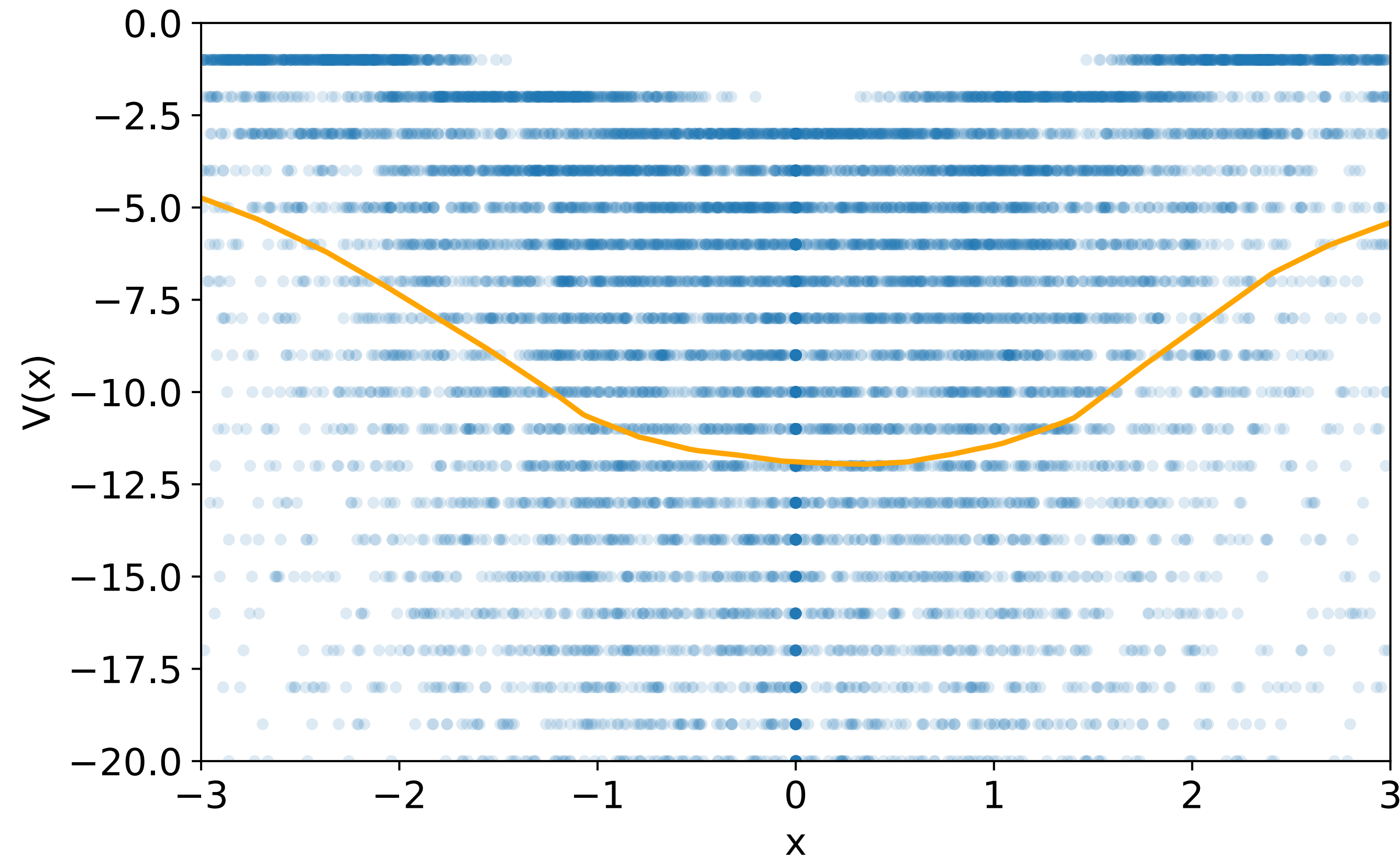
Each trajectory yields several training examples



$(x, V(x))$
 $(0, -16)$
 $(-1.15, -15)$
 $(-3, -14)$



Learned V^π



- Train as supervised regression (minimize MSE)
- Blue dots: training points (1k trajectories, \sim 12k samples)
- Orange line: fitted V^π (2-layer ReLU net, width 64)
- Note: extremely noisy!

Dynamic programming

- 2nd try: learn V^π by dynamic programming
 - ▶ generalize value iteration and TD-learning from above
- Idea: suppose, after t iterations, we've learned an estimate V_t^π with parameters ϕ_t
- Value iteration would *loop through* states s , calculate lookahead values $\mathbb{E}[r(s, a) + \gamma V_t^\pi(s') \mid s, \pi]$, update by *assigning* these values to a fresh function $V_{t+1}^\pi(s)$
- Instead, *sample* states s , *train* a new approximate value function V_{t+1}^π with parameters ϕ_{t+1}
 - ▶ want to use training data ~~$(s, \mathbb{E}[r(s, a) + \gamma V_t^\pi(s') \mid s, \pi])$~~
 - ▶ but we don't know $r(s, a)$ and $T(s' \mid s, a)$
 - ▶ so, use sampled outcome: data $(s, r + \gamma V_t^\pi(s'))$

Dynamic programming (code)

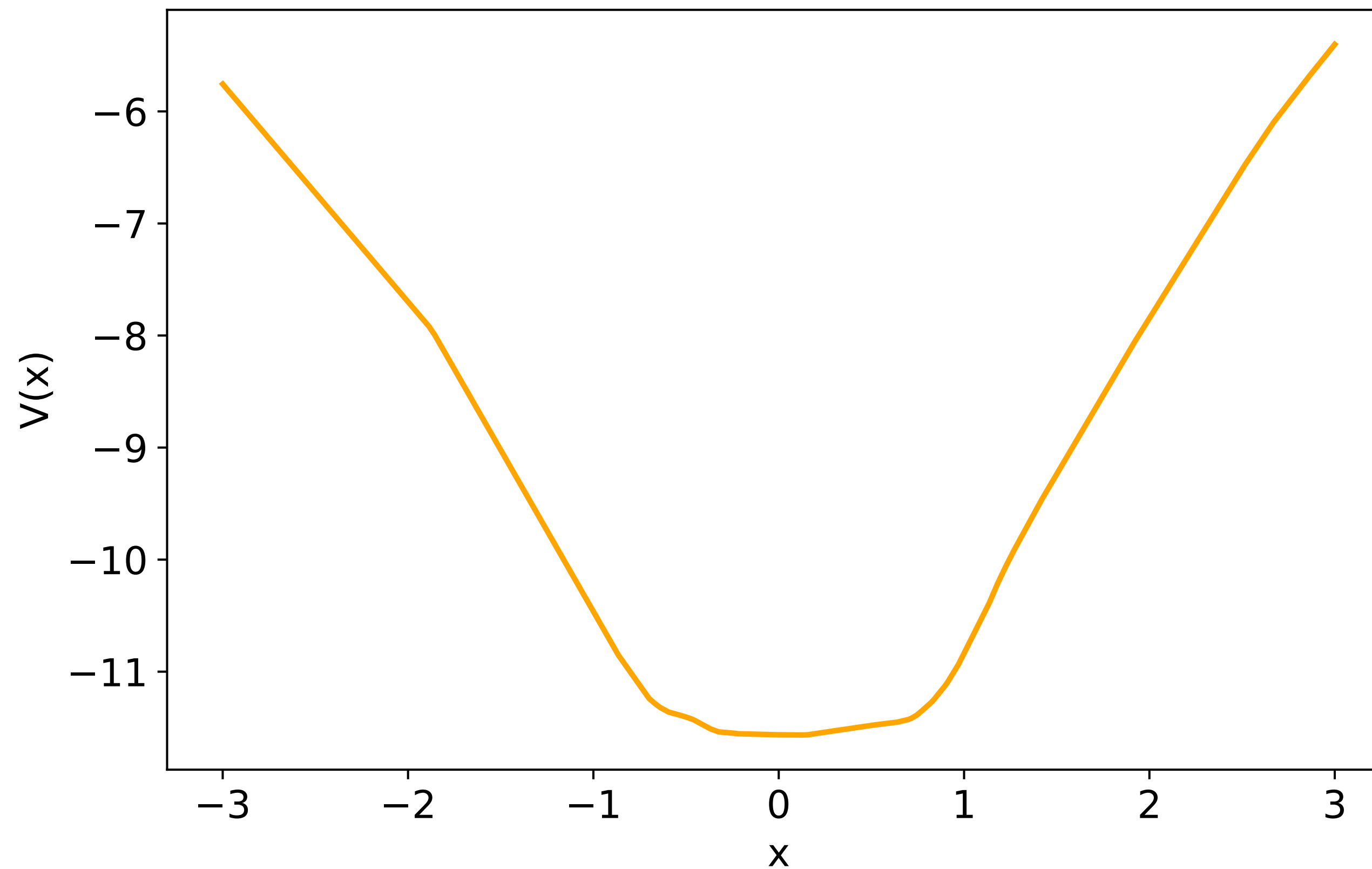
- Iteration t : estimated parameters ϕ_t
- Initialize new value network, parameters ϕ_{t+1}
- Repeat:
 - ▶ observe s, r, s'
 - ▶ calculate target $\hat{V}(s) = r + \gamma V_t^\pi(s')$
 - ▶ train V_{t+1}^π by SGD: loss $\frac{1}{2}(V_{t+1}^\pi(s) - \hat{V}(s))^2$
 - ▶ $V_{t+1}^\pi(s)$ is new (trainable) value function
 - ▶ $V_t^\pi(s')$ is old (fixed, no gradient) value function
- After a while, increment t :
 - ▶ fix ϕ_{t+1} , start training V_{t+2}^π with parameters ϕ_{t+2}
- Repeat
- Fixed ϕ_t is called a *target network*

How many samples per update?

- Hyperparameter: how often to update target network?
 - ▶ every 10 trajectories? every 100? every SGD step?
 - ▶ less often: closer to value iteration
 - ▶ more often: closer to temporal difference learning
- If we update after every SGD step, get $TD(0)$ algorithm
 - ▶ no need to store ϕ_t separately, just update in place
 - ▶ probably the best choice: in this context the only effect of waiting to update is to slow down learning
- By contrast, supervised regression method from a few slides ago is called $TD(1)$

There's a family of algorithms $TD(\lambda)$ for $\lambda \in [0,1]$ interpolating between $TD(0)$ and $TD(1)$

TD(0) example



```
for _ in range(3000):
    xs, rs = trajectory()
    T = len(xs)
    with torch.no_grad():
        tgt = [rs[t] + model(xs[t+1]) for t in range(T-1)] + [rs[T-1]]
    err = 0.0
    for t in range(T):
        err += criterion(model(xs[t]), tgt[t])
    optimizer.zero_grad()
    err.backward()
    optimizer.step()
```

Learn Q^π the same ways as V^π

- Set up supervised regression problem
 - ▶ training examples map $s_t, a_t \mapsto$ sum of discounted rewards after step t (cf. learning V^π)
- Or use fixed point iteration:
 - ▶ Q^π satisfies a Bellman equation just like V^π :
$$Q^\pi(s_t, a_t) \approx r_t + \gamma Q^\pi(s_{t+1}, a_{t+1})$$
 - ▶ evaluate RHS with (frozen) target network
 - ▶ train LHS by SGD
- These are called *SARSA(1)* and *SARSA(0)*

Policy improvement

- In tabular case, can just update π to be greedy for Q^π
$$\pi^{\text{gr}}(s) = \arg \max_a Q^\pi(s, a)$$
 - ▶ could try the same here, **but** there will be errors in our estimate of Q^π , so switching to π^{gr} is too aggressive
- In tabular case, can use max versions (Q iteration, Q-learning) to directly learn Q^*
 - ▶ but doesn't work as well w/ approximate Q (DQN kind of works but is messy and not SOTA)
 - ▶ same reason: errors in current Q estimates mean it's too aggressive to commit to greedy action
- Instead, scalable RL methods take a small step based on Q^π to improve π

Poll



- We are in a large, complex environment (say, a self-driving car) and have no accurate environment model
- We want to improve our policy π — previous slide suggests learning Q^π and taking a step towards being greedy for Q^π
- Would it work instead to learn V^π , compute the greedy policy for V^π , and use this policy as input to our update?
 - A. Yes
 - B. No
 - C. Need more information to determine
 - D. Don't pick this answer
 - E. Don't pick this answer

How to improve our policy?

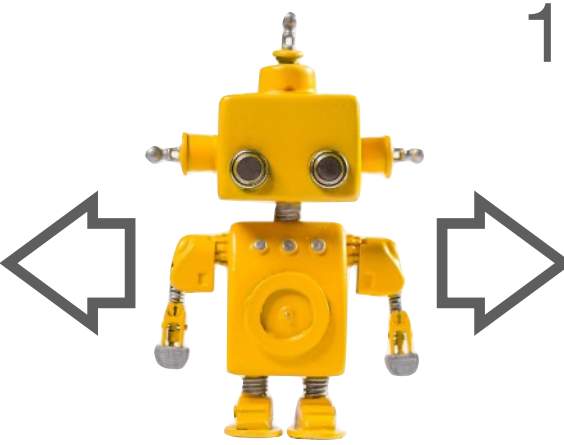


- Want to maximize $J(\theta) = \mathbb{E}_{\pi_{\theta}} [r_1 + r_2 + \dots + r_T]$
- Maybe the simplest idea: analogous to SGD
 - ▶ initialize policy parameters $\theta^1 \in \mathbb{R}^d$
 - ▶ on training iteration $m = 1, 2, \dots$:
 - ▶ compute stochastic estimate $g^m \approx \left. \frac{d}{d\theta} J(\theta) \right|_{\theta^m}$
 - ▶ update $\theta^{m+1} \leftarrow \theta^m + \eta g^m$ (learning rate η , could be η^m)
- Called the **policy gradient** method
- But how do we get g^m ?
 - ▶ not obvious how to differentiate expected cost J wrt θ : depends on (unknown) properties of environment

note: we're using undiscounted finite horizon here, but other setups are analogous

Advantage

- Def'n: **advantage function** is $A(s, a) = Q(s, a) - V(s)$
 - ▶ with A^π and A^* versions using Q^π/V^π and Q^*/V^*
 - ▶ “how does action a compare to the optimal action in state s (A^*) or to the distribution $\pi(a | s)$ (A^π)?”
 - ▶ if $A(s, a) \geq 0$ then $Q(s, a) \geq V(s)$: a is at least as good
 - ▶ else $Q(s, a) < V(s)$: a is worse
- Intuitively, $A^\pi(s, a)$ seems related to improving π
- Note: still call it advantage even if we're using costs
 - ▶ “disadvantage” would be more accurate

Advantage example

Environment						
	1	2	3	4	5	
	$r=-1$	$r=-1$	$r=-10$	$r=-1$	$r=0$	
V^*	1	2	3	4	5	
	-13	-12	-11	-1	0	
Q^*	1	2	3	4	5	
	L	-14	-14	-13	-12	0
	R	-13	-12	-11	-1	0
A^*	1	2	3	4	5	
	L	-1	-2	-2	-11	0
	R	0	0	0	0	0

Policy gradient

- **Thm:** the gradient of total reward J wrt policy $\pi(a | s)$ is

$$g^\pi(s, a) = d^\pi(s) A^\pi(s, a)$$

- ▶ here $d^\pi(s)$ is how often we visit state s under π

$$d^\pi(s) = \mathbb{1}(s_1 = s) + \gamma \mathbb{1}(s_2 = s) + \gamma^2 \mathbb{1}(s_3 = s) + \dots$$

- To improve policy quickly, increase $\pi(a | s)$ if
 - ▶ action a is better than alternatives (large $A^\pi(s, a)$)
 - ▶ state s is visited often (large $d^\pi(s)$)
- Expression above is the gradient wrt π itself
 - ▶ if $\pi_\theta(a | s)$ is a learned function with parameters θ then

$$\frac{dJ}{d\theta} = \sum_{s,a} g(s, a) \frac{d}{d\theta} \pi_\theta(a | s) \text{ (chain rule)}$$

Stochastic gradient estimates

- Previous slide gave the exact gradient $\frac{dJ}{d\pi}$ or $\frac{dJ}{d\theta}$
 - ▶ but it's too expensive to use (have to iterate over all s, a)
- Instead, need a stochastic estimate that we can use with SGD-like updates in policy gradient
 - ▶ correct expected value, but with sampling noise
- Three common ways to get such estimates
 - ▶ actor-critic
 - ▶ REINFORCE
 - ▶ reparameterization

Sampling states

- From above: with parameterized policy $\pi_\theta(a | s)$,

$$\frac{dJ}{d\theta} = \sum_{s,a} d^\pi(s) A^\pi(s, a) \frac{d}{d\theta} \pi_\theta(a | s)$$

- RHS looks like an expectation: that is,

- ▶ sample a state s according to (normalized) $d^\pi(s)$

- ▶ compute $\hat{g}(\theta) = \sum_a A^\pi(s, a) \frac{d}{d\theta} \pi_\theta(a | s)$, so that

$$\mathbb{E}(\hat{g}(\theta)) = \sum_s \frac{1}{Z} d^\pi(s) \sum_a A^\pi(s, a) \frac{d}{d\theta} \pi_\theta(a | s) = \frac{1}{Z} \frac{dJ}{d\theta}$$

where Z is normalizer for d^π

- ▶ as desired!

Actor-critic

- Don't know A^π ?
 - ▶ learn Q^π, V^π with methods above!
 - ▶ called a ***critic*** (in contrast with the ***actor***, π)
- How do we sample from d^π / Z ?
 - ▶ run a trajectory according to π
 - ▶ keep state from step t with probability proportional to γ^t
 - ▶ or just keep all states and weight by γ^t
 - ▶ or don't even bother with weighting
- Worry: what if our estimate of A^π is bad?

Sampling states and actions

- From above, $\frac{dJ}{d\theta} = \sum_{s,a} d^\pi(s) A^\pi(s, a) \frac{d}{d\theta} \pi_\theta(a | s)$
- Useful fact:
 - ▶ $\frac{d}{d\theta} \ln \pi_\theta(\overset{als}{s, a}) = \frac{1}{\pi_\theta(a | s)} \frac{d}{d\theta} \pi_\theta(a | s)$ (chain rule)
 - ▶ so $\pi_\theta(a | s) \frac{d}{d\theta} \ln \pi_\theta(a | s) = \frac{d}{d\theta} \pi_\theta(a | s)$
- Substituting:
$$\frac{dJ}{d\theta} = \sum_{s,a} d^\pi(s) \pi_\theta(a | s) A^\pi(s, a) \frac{d}{d\theta} \ln \pi_\theta(a | s)$$
- This looks like an expectation where we sample s, a proportional to $d^\pi(s) \pi_\theta(a | s)$
 - i.e., just sample a step s, a from a trajectory following π_θ

Advantage estimates

- Means we only need advantage estimates for s, a we actually executed during a trajectory
 - ▶ i.e., we need estimates $Q^\pi(s, a)$ and $V^\pi(s)$
- $Q^\pi(s, a)$ is easy: sum of rewards starting from s, a
 - ▶ unbiased! (but higher variance)
- Turns out we can use *any* estimate of $V^\pi(s)$
 - ▶ $\sum_a \pi_\theta(a | s) = 1$
 - ▶ differentiate both sides: $\sum_a \frac{d}{d\theta} \pi_\theta(a | s) = 0$
 - ▶ use log identity: $\mathbb{E}_{a \sim \pi} \frac{d}{d\theta} \ln \pi_\theta(a | s) = 0$
 - ▶ so any per-state offset to A^π cancels out on average
 - ▶ called a **baseline** in this context

Policy gradient theorem

d = number of parameters in policy, so $\theta \in \mathbb{R}^d$

$$J(\theta) = \mathbb{E}_{\pi_\theta} [r_1 + r_2 + \dots + r_T]$$

- On iteration m , get trajectory $\tau^m = (s_1^m, a_1^m, r_1^m, \dots, s_T^m, a_T^m, r_T^m)$ by following policy $\pi_\theta(a_t^m | s_t^m)$

- Define

$$A_t^m = \sum_{i=t}^T r_i^m \in \mathbb{R} \text{ (empirical total reward starting from step } t\text{)}$$

— optionally subtract baseline estimate of $V^\pi(s)$

$$u_t^m = \frac{d}{d\theta} \ln \pi_\theta(a_t^m | s_t^m) \in \mathbb{R}^d \text{ (action score vector, from autodiff)}$$

$$g^m = \sum_{t=1}^T A_t^m u_t^m \in \mathbb{R}^d \text{ (the gradient estimate)}$$

Theorem:

g^m is an unbiased estimate of $\frac{d}{d\theta} J(\theta)$

Policy gradient theorem

d = number of parameters in policy, so $\theta \in \mathbb{R}^d$

$$J(\theta) = \mathbb{E}_{\pi_{\theta}} [r_1 + r_2 + \dots + r_T]$$

- On iteration m , get trajectory $\tau^m = (s_1^m, a_1^m, r_1^m, \dots, s_T^m, a_T^m, r_T^m)$ by following policy $\pi_{\theta}(a_t^m | s_t^m)$

- Define

$$A_t^m = \sum_{i=t}^T r_i^m \in \mathbb{R} \text{ (empirical total reward starting from step } t\text{)}$$

— optionally subtract baseline estimate of $V^{\pi}(s)$

$$u_t^m = \frac{d}{d\theta} \ln \pi_{\theta}(a_t^m | s_t^m) \in \mathbb{R}^d \text{ (action score vector, from autodiff)}$$

$$g^m = \sum_{t=1}^T A_t^m u_t^m \in \mathbb{R}^d \text{ (the gradient estimate)}$$

Theorem:

$$g^m \text{ is an unbiased estimate of } \frac{d}{d\theta} J(\theta)$$

even if we don't know anything about the environment

Policy gradient theorem

d = number of parameters in policy, so $\theta \in \mathbb{R}^d$

$$J(\theta) = \mathbb{E}_{\pi_{\theta}} [r_1 + r_2 + \dots + r_T]$$

- On iteration m , get trajectory $\tau^m = (s_1^m, a_1^m, r_1^m, \dots, s_T^m, a_T^m, r_T^m)$ by following policy $\pi_{\theta}(a_t^m | s_t^m)$

- Define

$$A_t^m = \sum_{i=t}^T r_i^m \in \mathbb{R} \text{ (empirical total reward starting from step } t\text{)}$$

— optionally subtract baseline estimate of $V^{\pi}(s)$

$$u_t^m = \frac{d}{d\theta} \ln \pi_{\theta}(a_t^m | s_t^m) \in \mathbb{R}^d \text{ (action score vector, from autodiff)}$$

$$g^m = \sum_{t=1}^T A_t^m u_t^m \in \mathbb{R}^d \text{ (the gradient estimate)}$$

Theorem:

g^m is an unbiased estimate of $\frac{d}{d\theta} J(\theta)$

even if we don't know anything about the environment

and **even if** environment is PO

REINFORCE ***intuition***

- From above, gradient estimate $g^m = \sum_{t=1}^T A_t^m u_t^m$
- Score vectors u_t^m : parameter direction that would increase (log-)probability of taking action a_t^m in state s_t^m
- Scale by A_t^m : upweight score vector when advantage is large, flip score vector if advantage is negative
- On average: a direction that changes policy by taking actions more often if they were associated with high (total future) rewards
 - ▶ step along this direction: change policy *multiplicatively* in favor of high-reward actions
- To minimize costs, step along *negative* gradient: take actions *less* often if associated with high costs

REINFORCE

- If we plug the estimate from the policy gradient theorem into the policy gradient method, we get *REINFORCE* [Williams, 1992], one of the oldest RL algorithms
- Repeat:
 - ▶ gather some trajectories under current policy π_θ
 - ▶ compute gradient estimate g by formula above
 - ▶ update θ by SGD
- Showed version for undiscounted fixed horizon; results and algorithms are almost the same for discounted or stochastic shortest paths, or for costs instead of rewards

Continuous actions

- For continuous actions, policy is often given as a network whose outputs are parameters of conditional distribution over actions
 - ▶ $\pi_{\theta}(a | s) = \mathcal{N}(a; \mu_{\theta}(s), \sigma^2)$
- We can *reparameterize* to isolate randomness
 - ▶ $a = \mu_{\theta}(s) + z$ where $z \sim \mathcal{N}(0, \sigma^2)$
- Fixing $A_s(a)$ to be a learned advantage estimate for each state s (considered as a function over a , $A^{\pi}(s, \cdot)$) we can rearrange REINFORCE gradient formula:

$$\begin{aligned}\frac{dJ}{d\theta} &= \mathbb{E}_{s, a \sim \pi_{\theta}} \left[A_s(a) \frac{d}{d\theta} \ln \pi_{\theta}(a | s) \right] \\ &= \mathbb{E}_{s \sim \pi_{\theta}} \left[\sum_a A_s(a) \frac{d}{d\theta} \pi_{\theta}(a | s) \right] \\ &= \mathbb{E}_{s \sim \pi_{\theta}} \left[\frac{d}{d\theta} \sum_a A_s(a) \pi_{\theta}(a | s) \right] \\ &= \mathbb{E}_{s \sim \pi_{\theta}} \left[\frac{d}{d\theta} \mathbb{E}_{a \sim \pi_{\theta}} A_s(a) \right]\end{aligned}$$



Reparameterization gradient

- From above: $\frac{dJ}{d\theta} = \mathbb{E}_{s \sim \pi_\theta} \left[\frac{d}{d\theta} \mathbb{E}_{a \sim \pi_\theta} A_s(a) \right]$

- Substituting $a = \mu_\theta(s) + z$

$$\frac{dJ}{d\theta} = \mathbb{E}_{s \sim \pi_\theta} \left[\frac{d}{d\theta} \mathbb{E}_{z \sim \mathcal{N}(0, \sigma^2)} A_s(\mu_\theta(s) + z) \right]$$

$$= \mathbb{E}_{s \sim \pi_\theta, z \sim \mathcal{N}(0, \sigma^2)} \left[A'_s(\mu_\theta(s) + z) \frac{d}{d\theta} \mu_\theta(s) \right]$$

- That is, we can get gradient samples by taking s, z from a trajectory (note: randomness z instead of action a)

- ▶ we use $A'_s(a) = \frac{d}{da} A^\pi(s, a)$ — not used in other methods we showed above

- ▶ have to hope our learned advantage estimate has accurate derivatives wrt actions