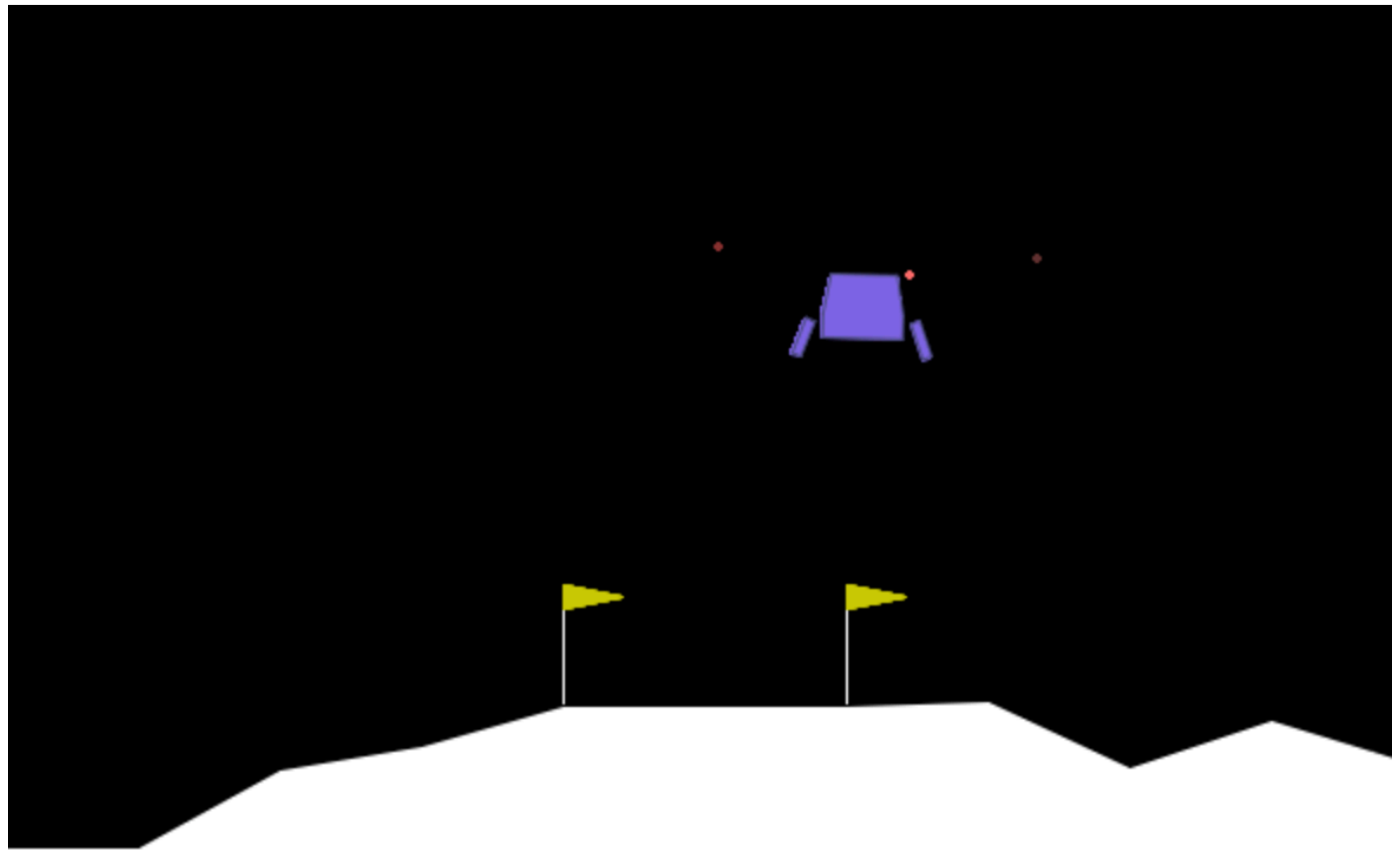
Scaling up RL



Lunar lander

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

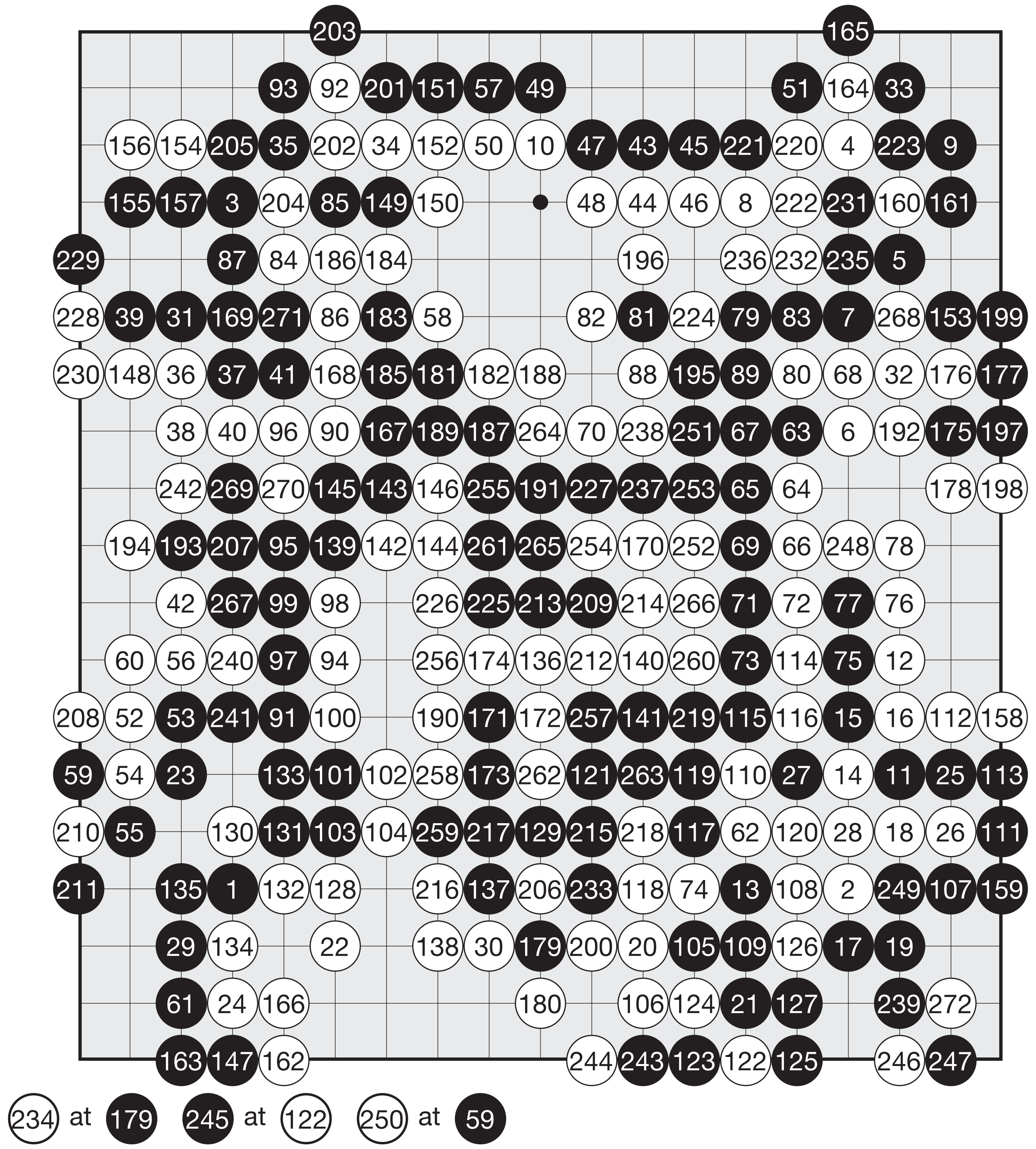
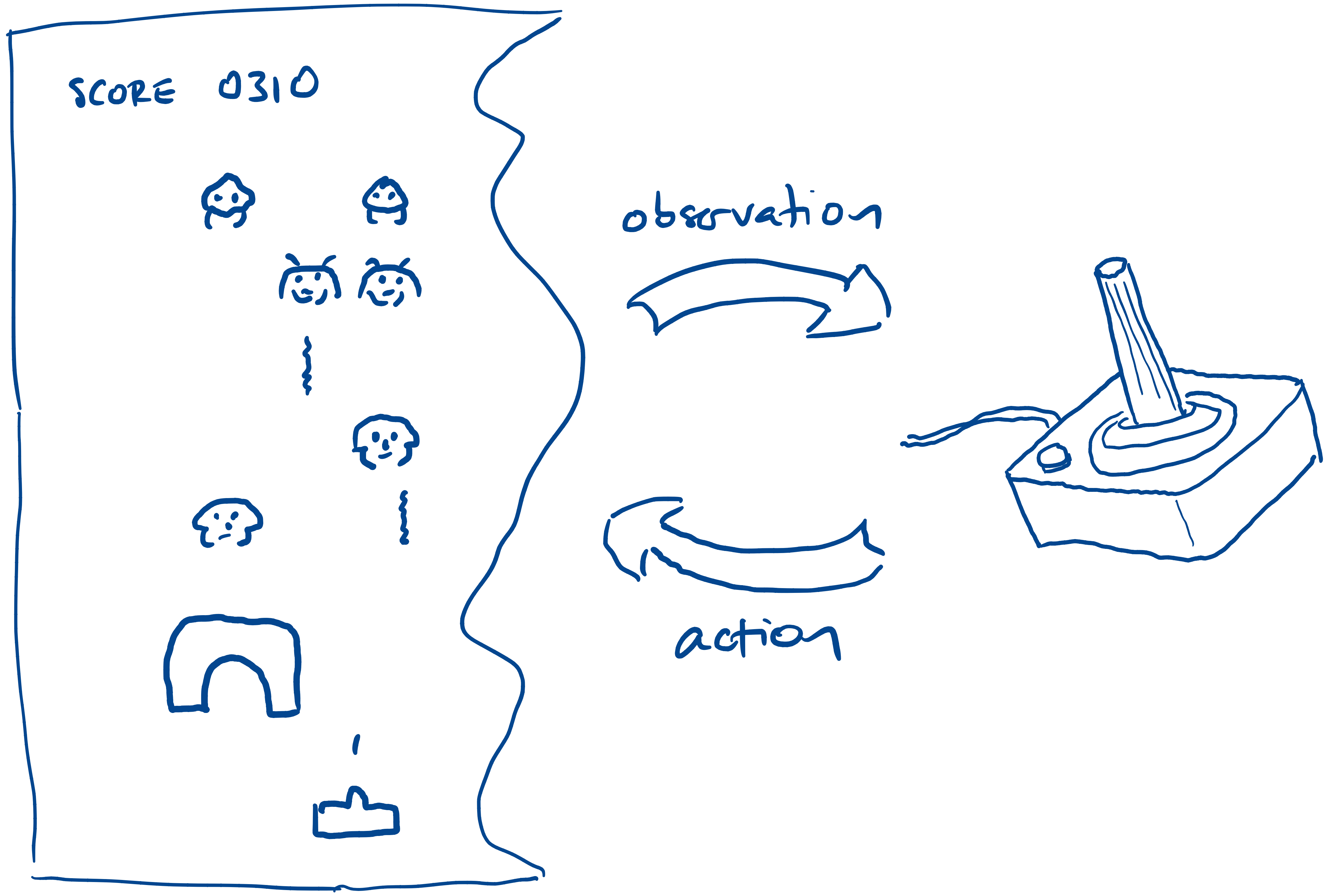


Image credit: Silver et al., Nature, 2016

- Today's lecture: going beyond the simple RL problems we've done so far
- We'll need some changes in our setup

No model

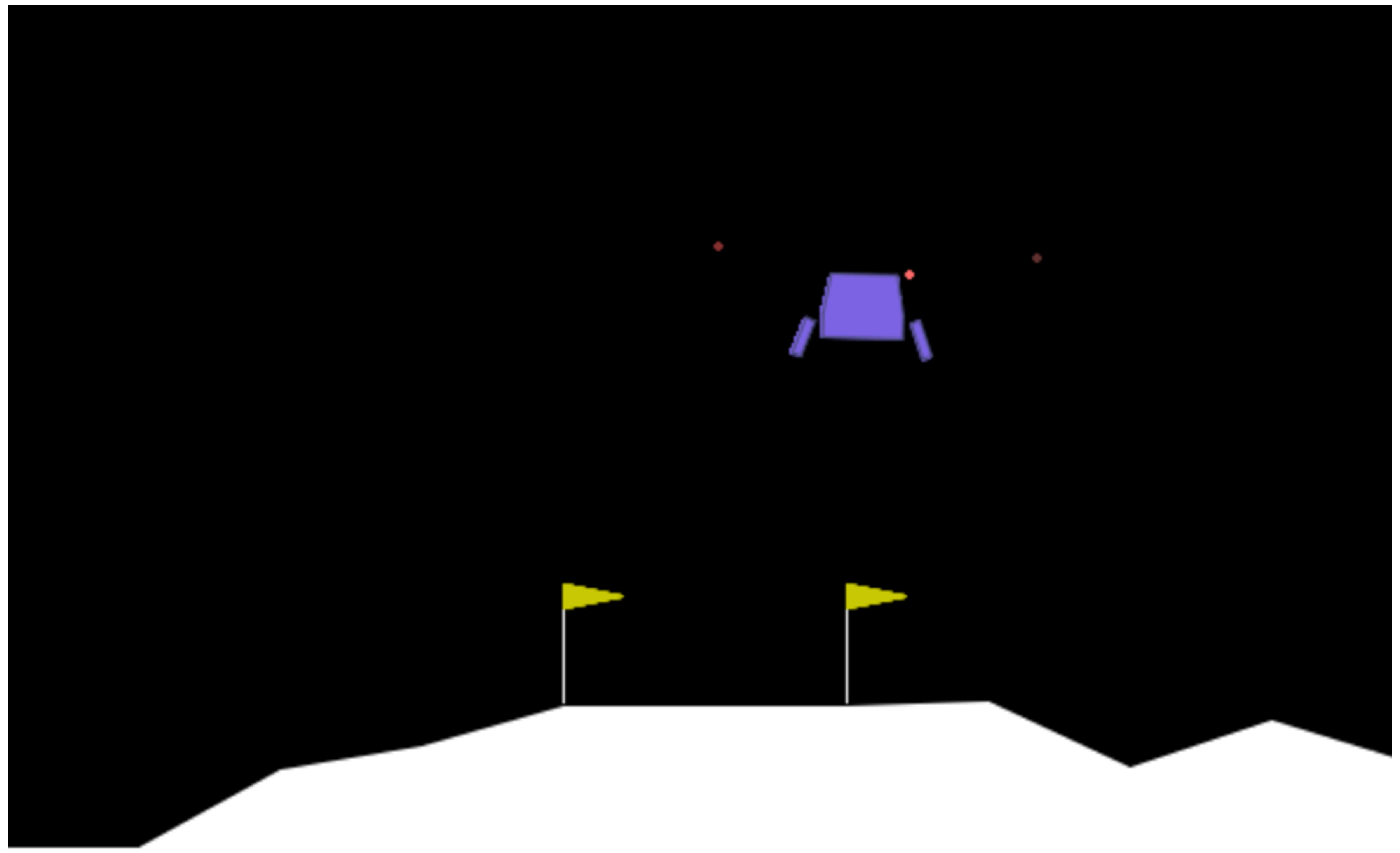


- Previously: we knew description of the world, e.g., expressions for $R(s, a)$ or $P(s' \mid s, a)$
- Instead: agent just interacts with environment over time — if we want $R(s, a)$ etc., have to learn it from data
- Alternating observations, actions, rewards $o_1, a_1, r_1, o_2, a_2, r_2, \ldots$

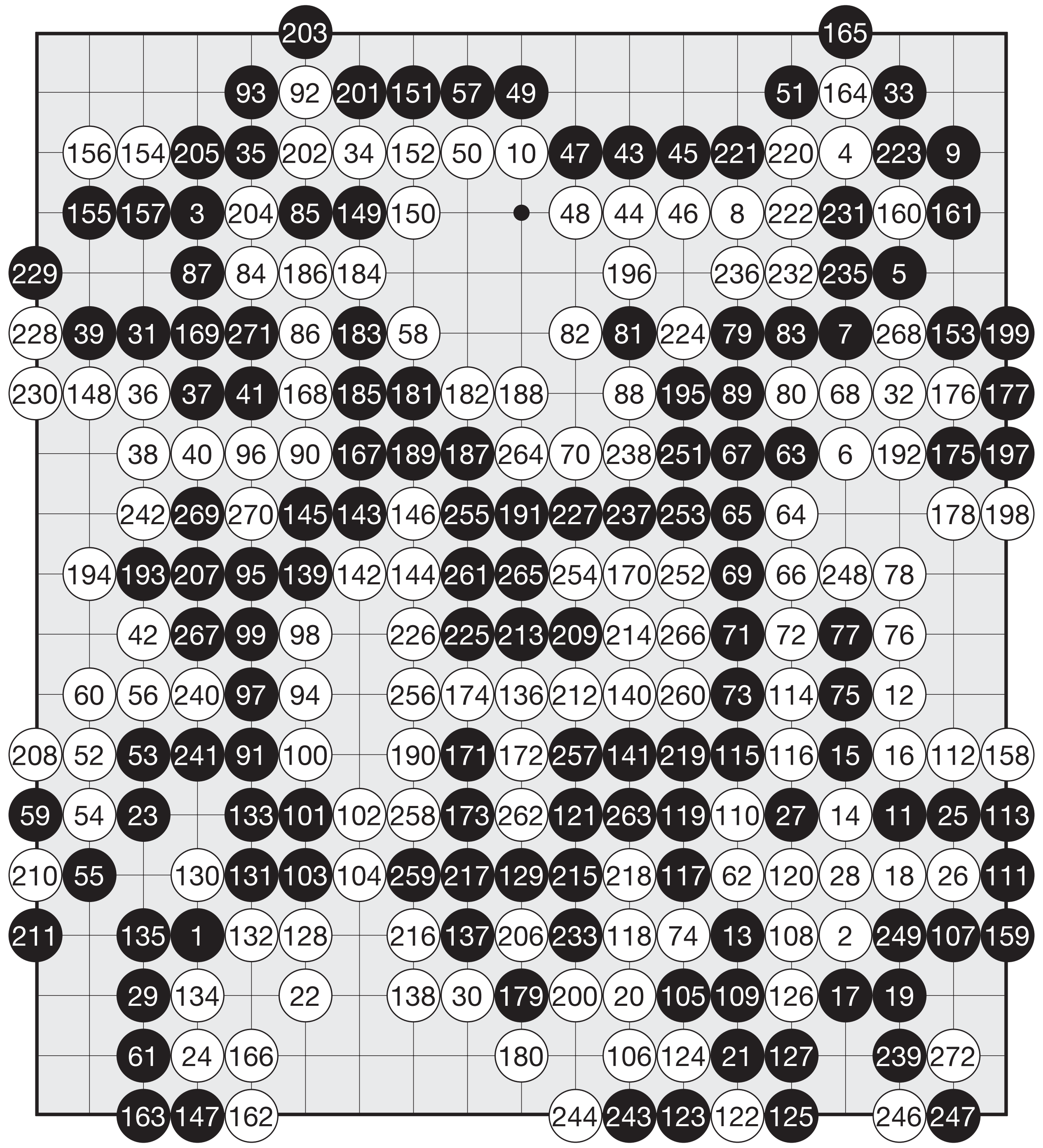
called a *trajectory*

Geoff Gordon

Example environments

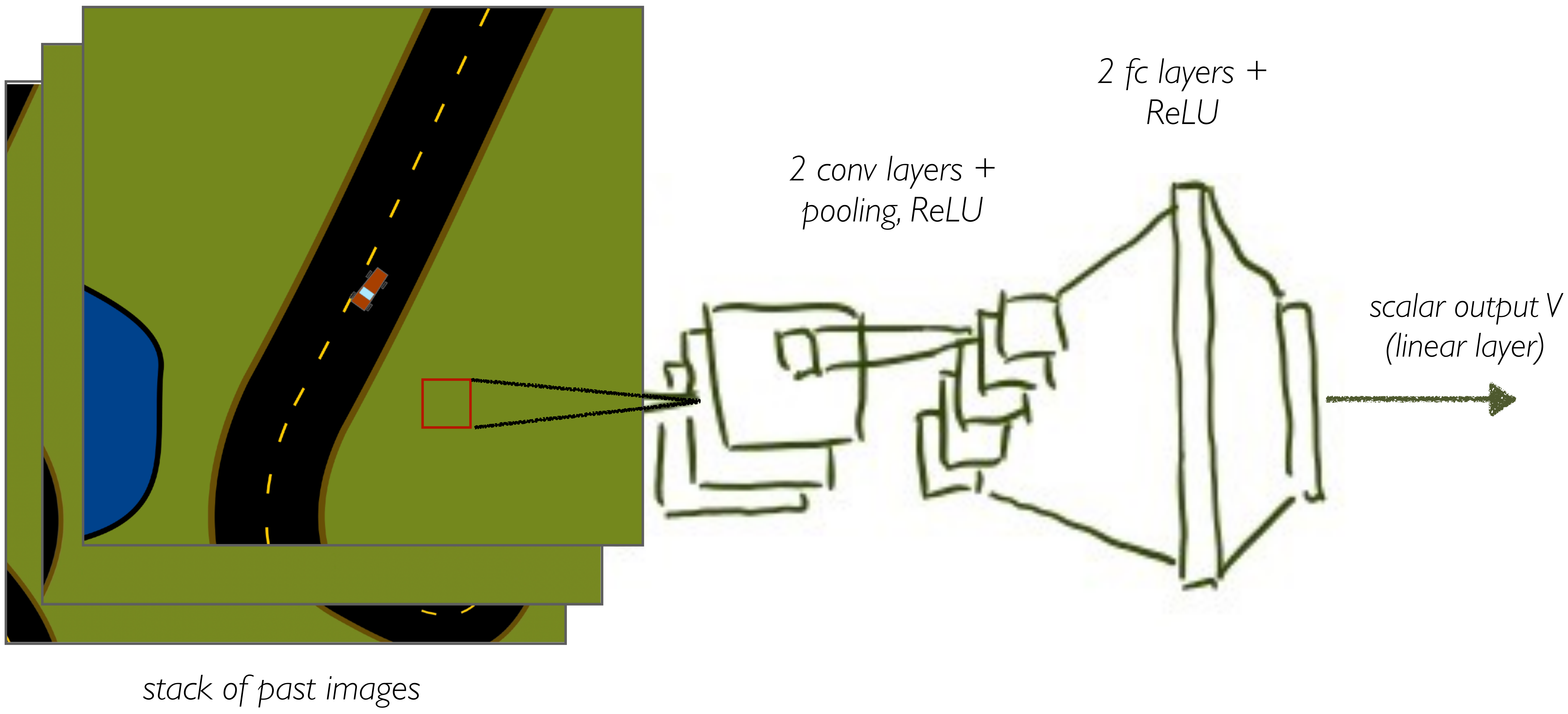


observations: screen images
actions: controller buttons, joystick position
transitions: determined by game code
reward: score increase



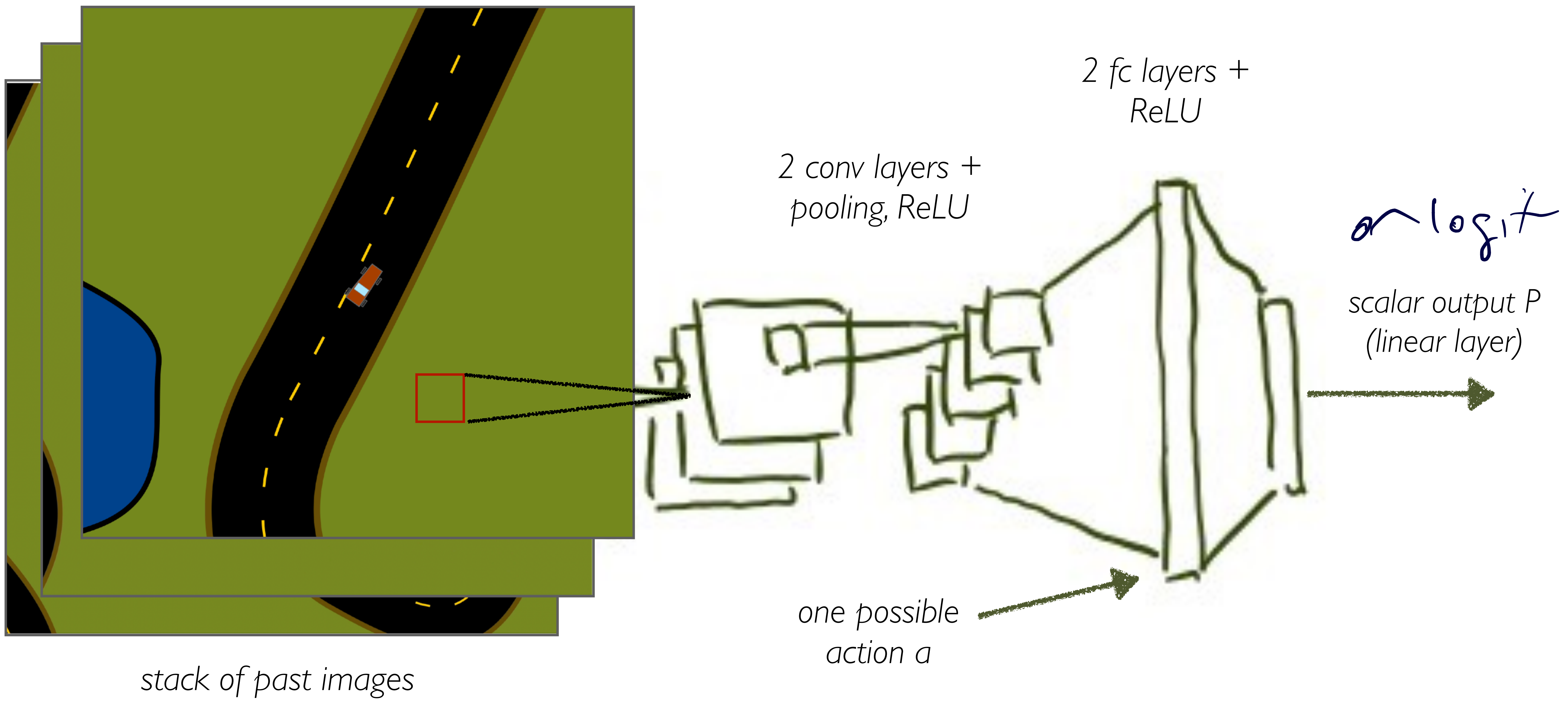
observations: board $\{B, W, \varnothing\}^{19\times19}$
actions: place a stone
transitions: rules of Go, opponent follows a previous policy (self-play)
reward: +1 for win, –1 for loss, 0 for draw, 0 if game isn't over

Learned, approximate functions



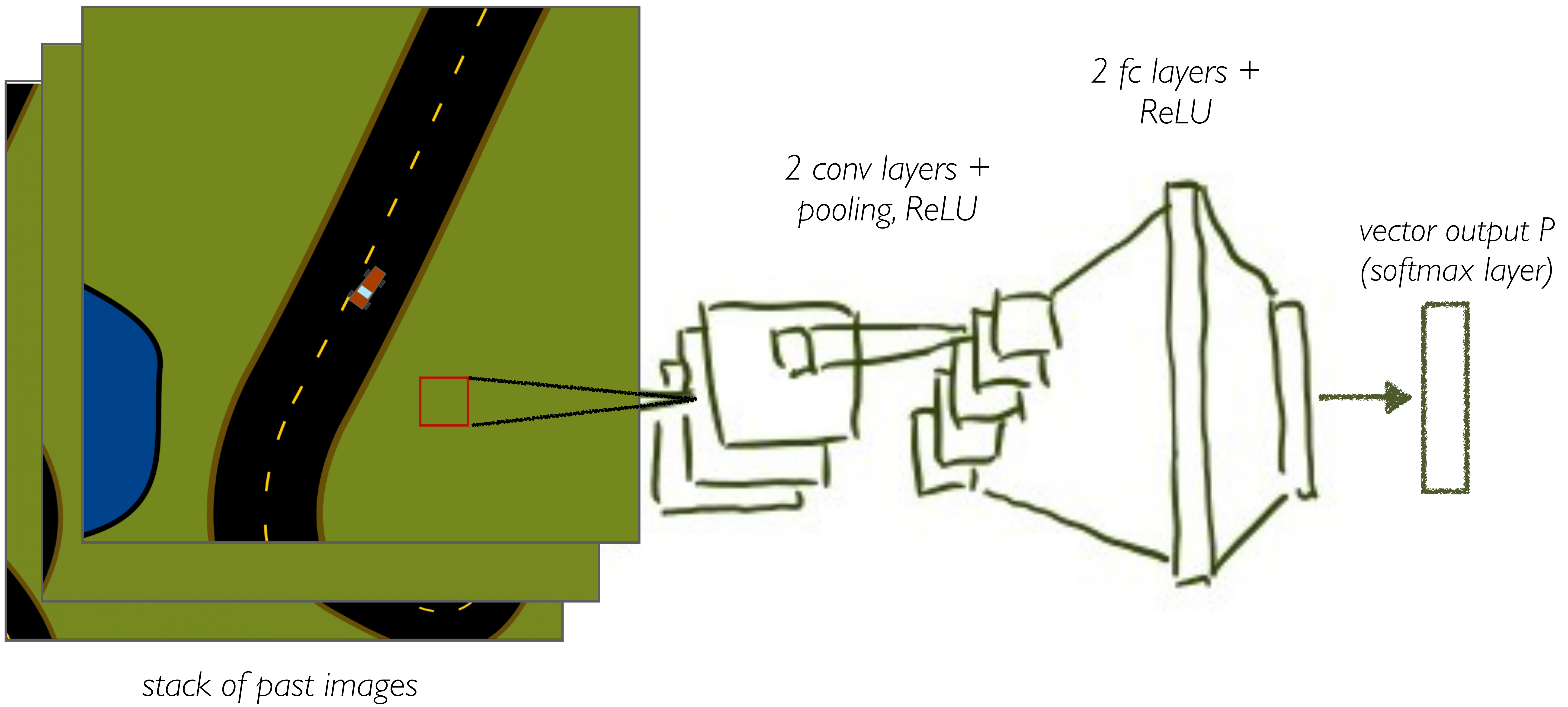
- Previously: could list out all states, keep a table of a function like $V^{\pi}(s)$
- Now: any function we care about has to be represented as an ML model, e.g., a deep net
- One parameter vector per function we care about, each fn can have its own network architecture

Policy



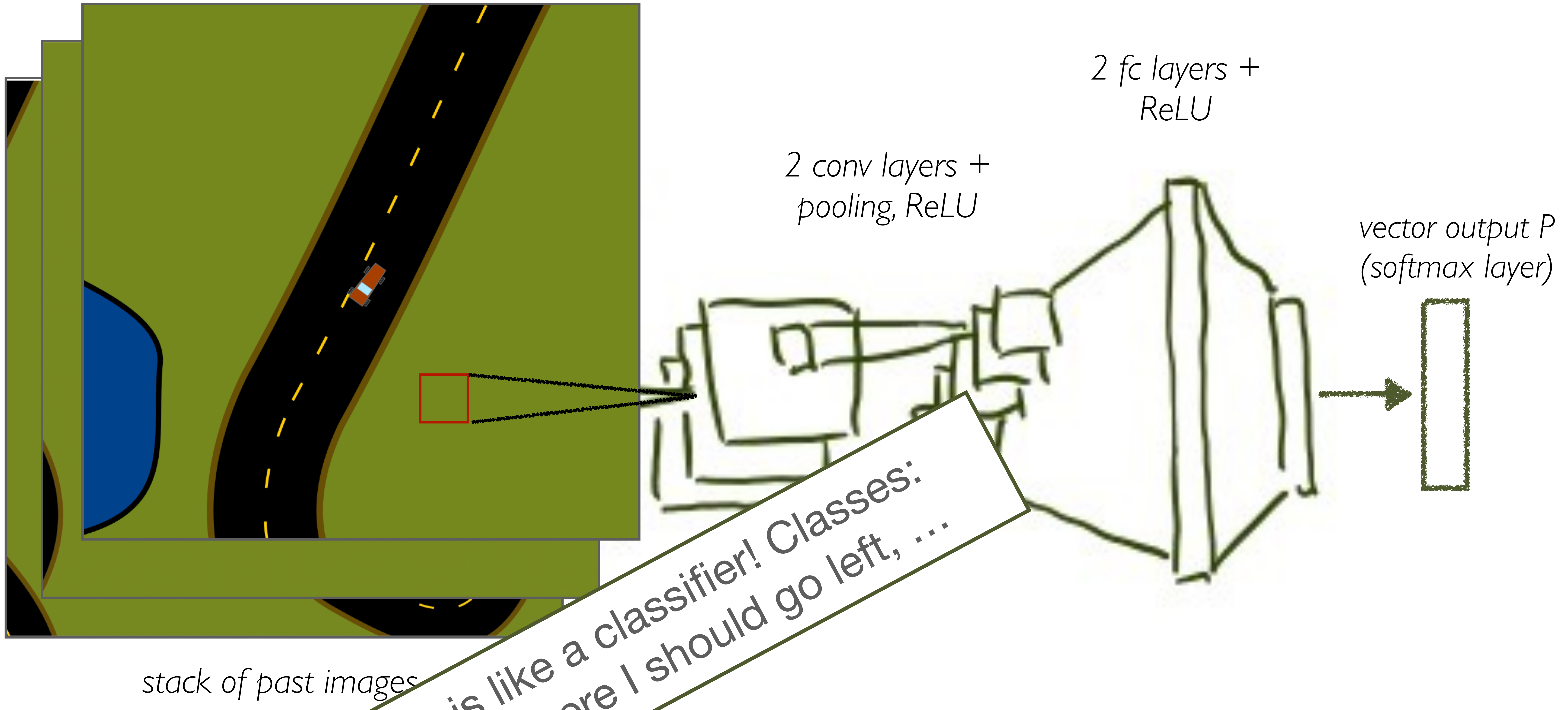
- Policy is a model too: represents $P(a \mid s, \pi)$
  - note: stochastic! (lets an optimizer make small changes)
- Several common ways to set up:
  - $s, a \mapsto P(a \mid s, \pi)$

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:
  - $s \mapsto [P(a_1 \mid s, \pi), P(a_2 \mid s, \pi), \ldots, P(a_k \mid s, \pi)]^\top$

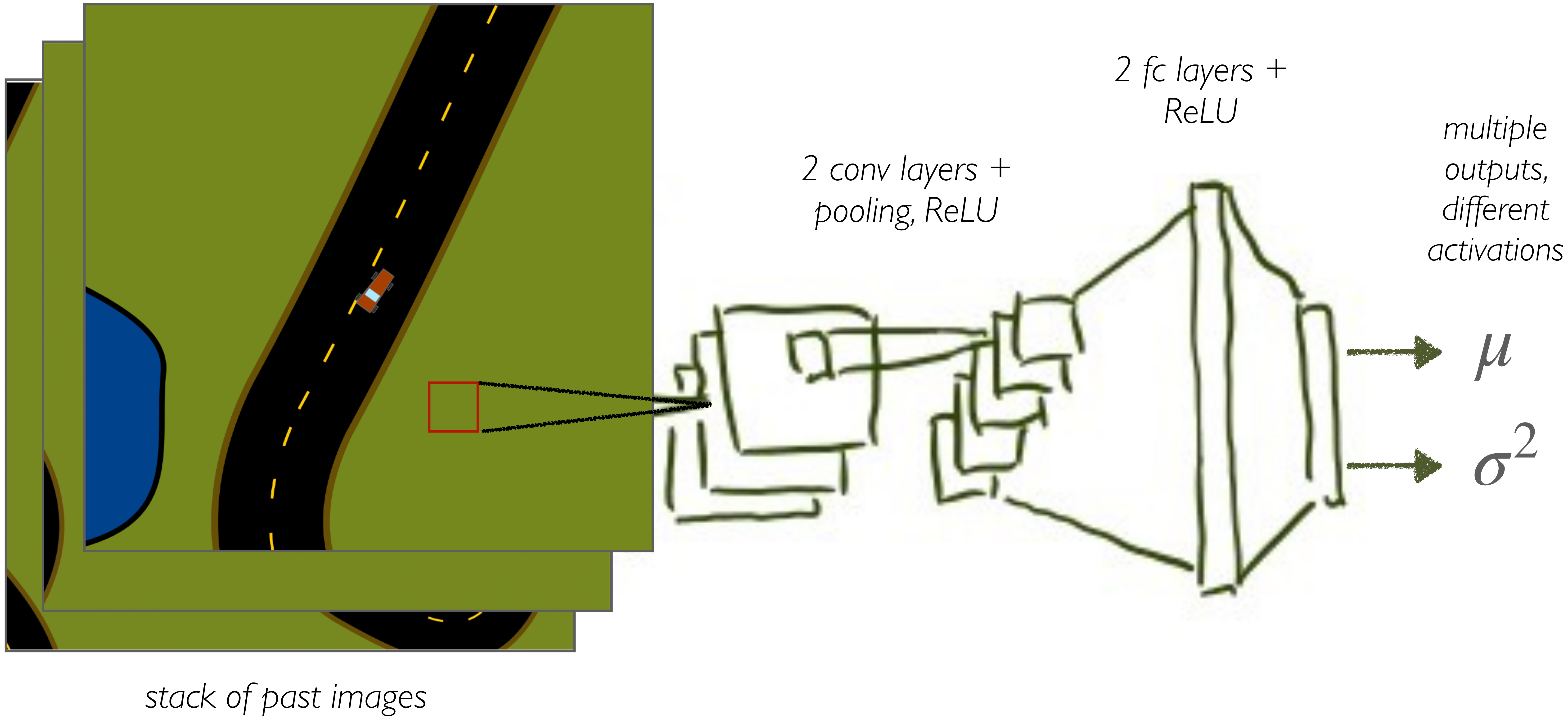
Policy



This is like a classifier! Classes:
states where I should go left, …

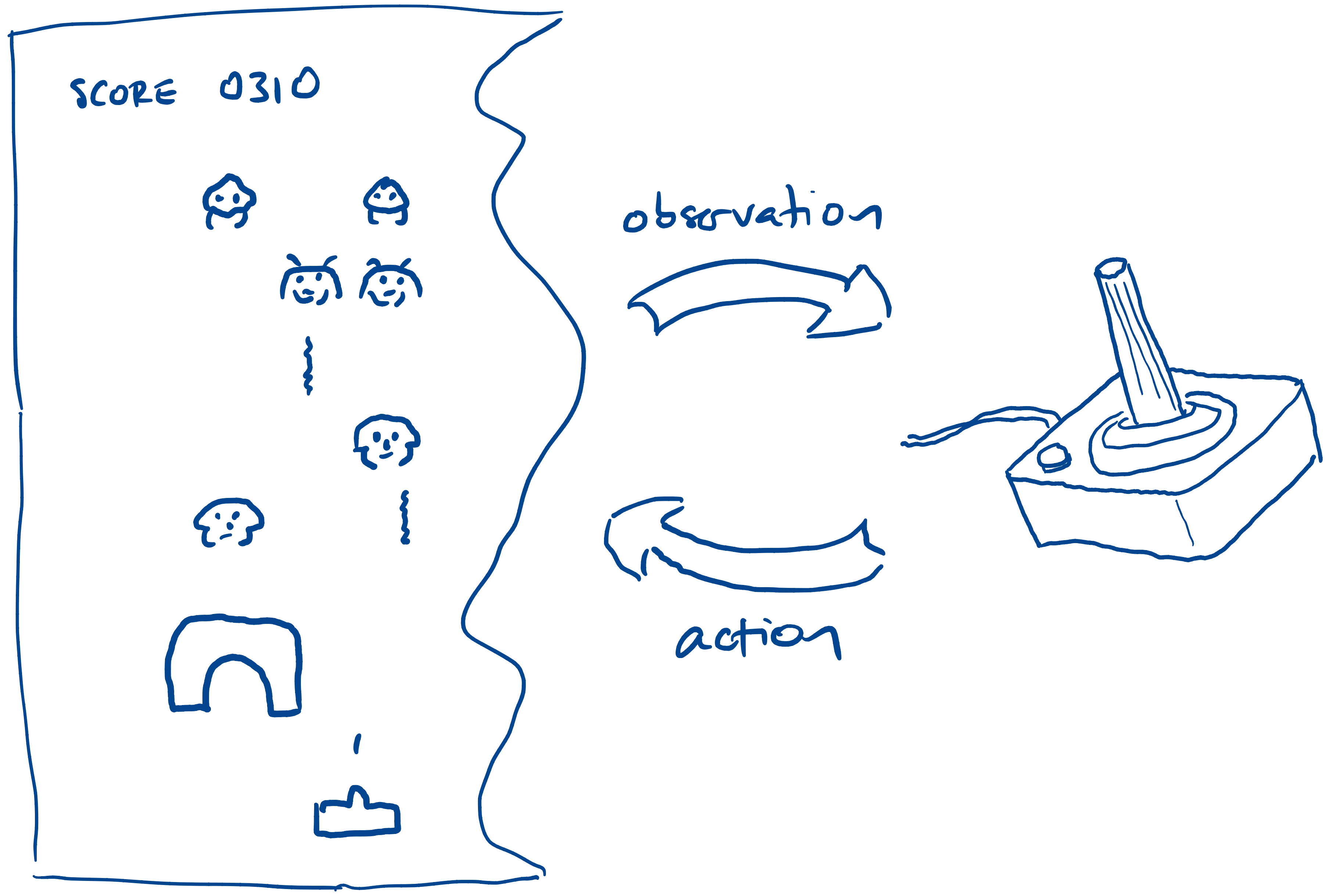
- Policy is a mo[…]oo: represents $P(a \mid s, \pi)$
  - note: stochastic! (lets an optimizer make small changes)
- Several common ways to set up:
  - $s \mapsto [P(a_1 \mid s, \pi), P(a_2 \mid s, \pi), \ldots, P(a_k \mid s, \pi)]^\top$

Policy



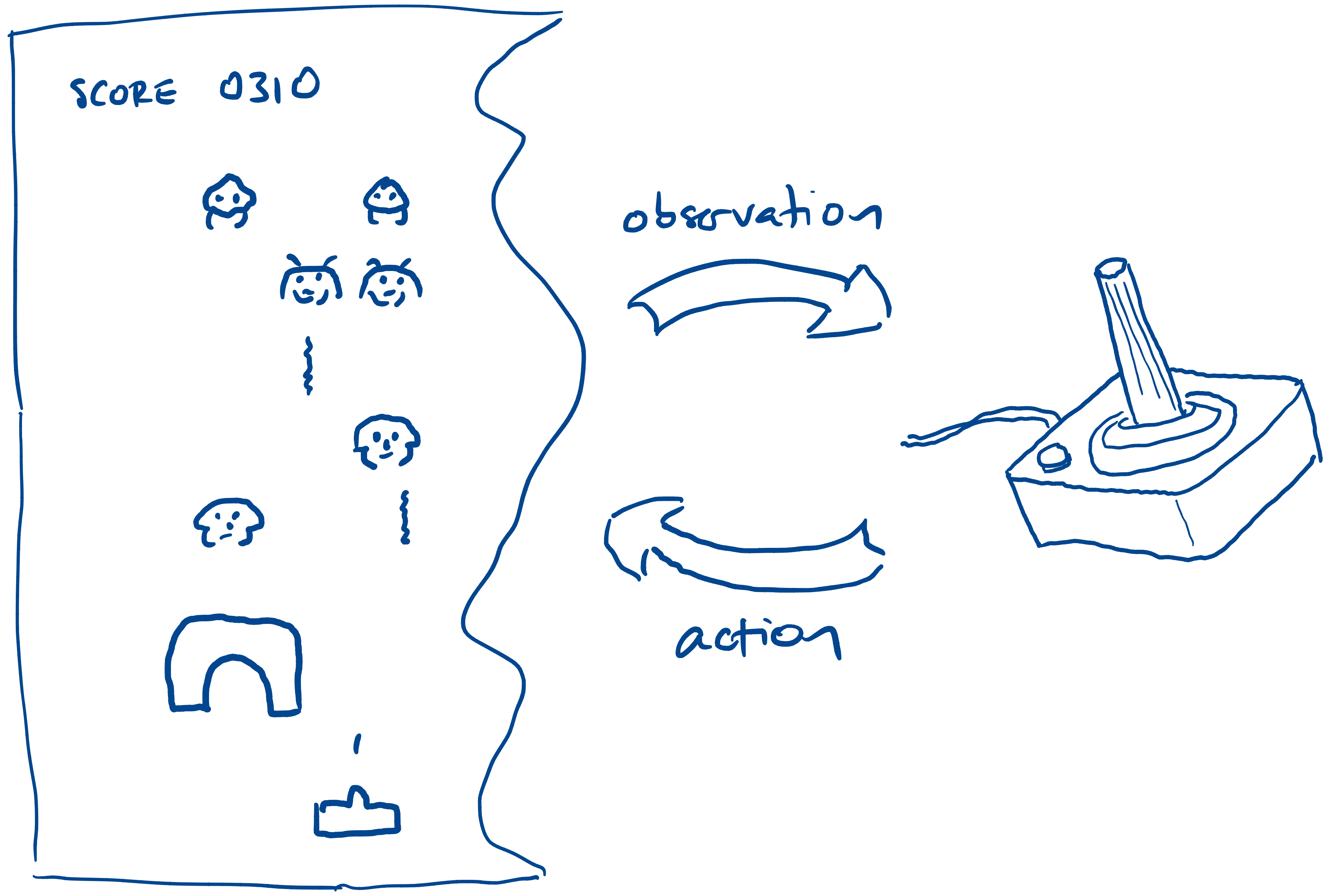
- Policy is a model too: represents $P(a \mid s, \pi)$
  - note: stochastic! (lets an optimizer make small changes)
- Several common ways to set up:
  - $s \mapsto$ parameters of action distribution like mean, variance

State vs. observation



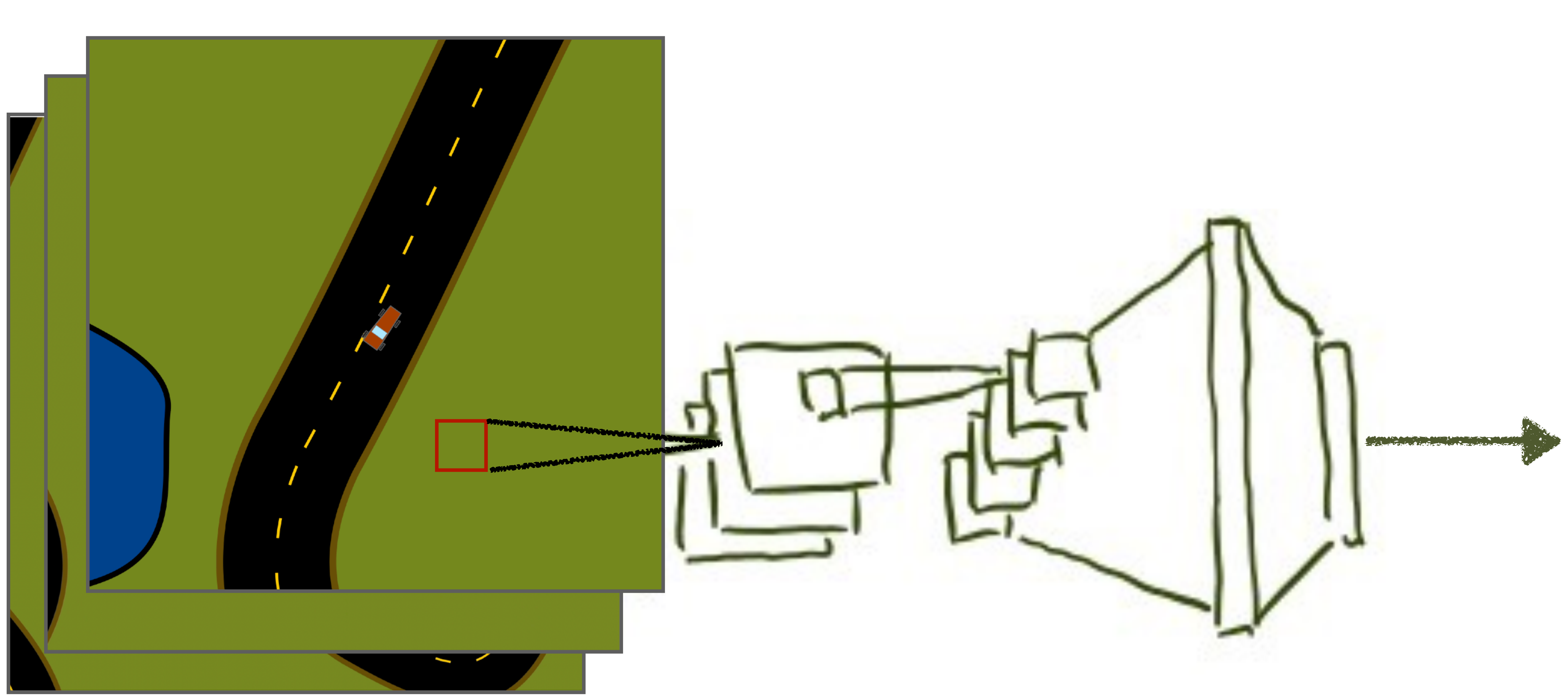
- Agent doesn't see state directly: $s_t \neq o_t$, common mistake!
  - observation informs about state: e.g., screen image → position
  - but often need to fuse information from several o_t: e.g., velocities
- Terminology: *fully/partially observable*

State vs. observation



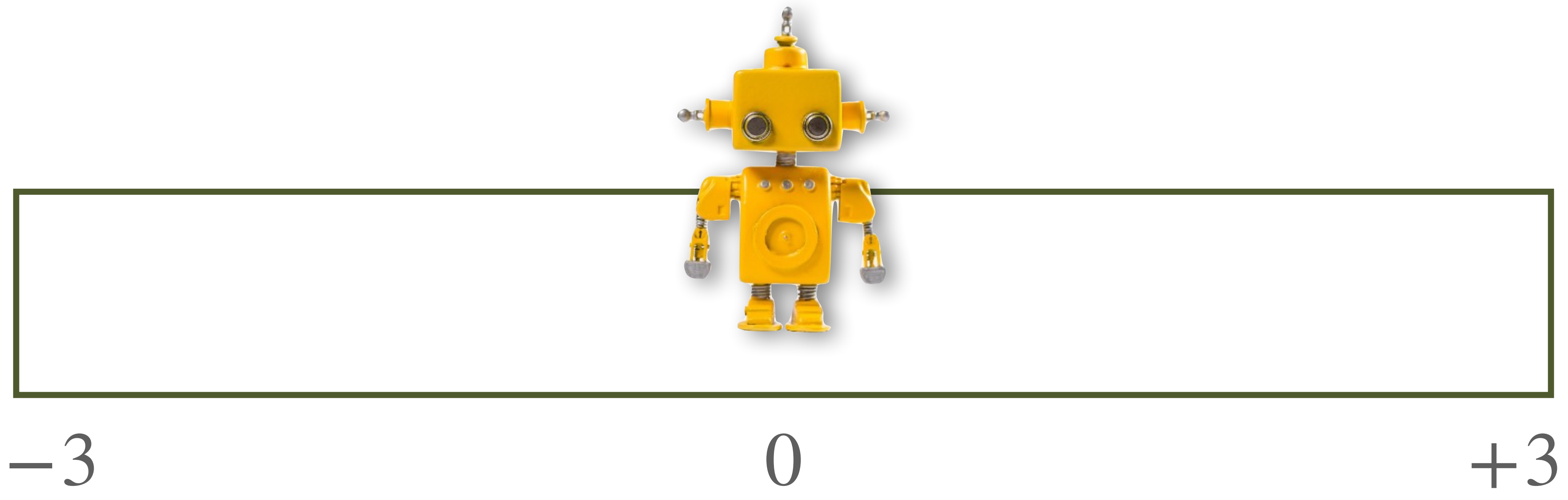
- What do we do if we don't know s_t?
- Simplest approach: network implicitly figures out the state from its input (such as a stack of images in slides above)
  - lots of more complicated approaches, but not in 301/601
- Assume this approach: a trajectory is now $s_1, a_1, r_1, s_2, \ldots$
  - each s_t is ***sufficient info for network to reconstruct state***
  - e.g., stack of past observations and actions

Learning V^π



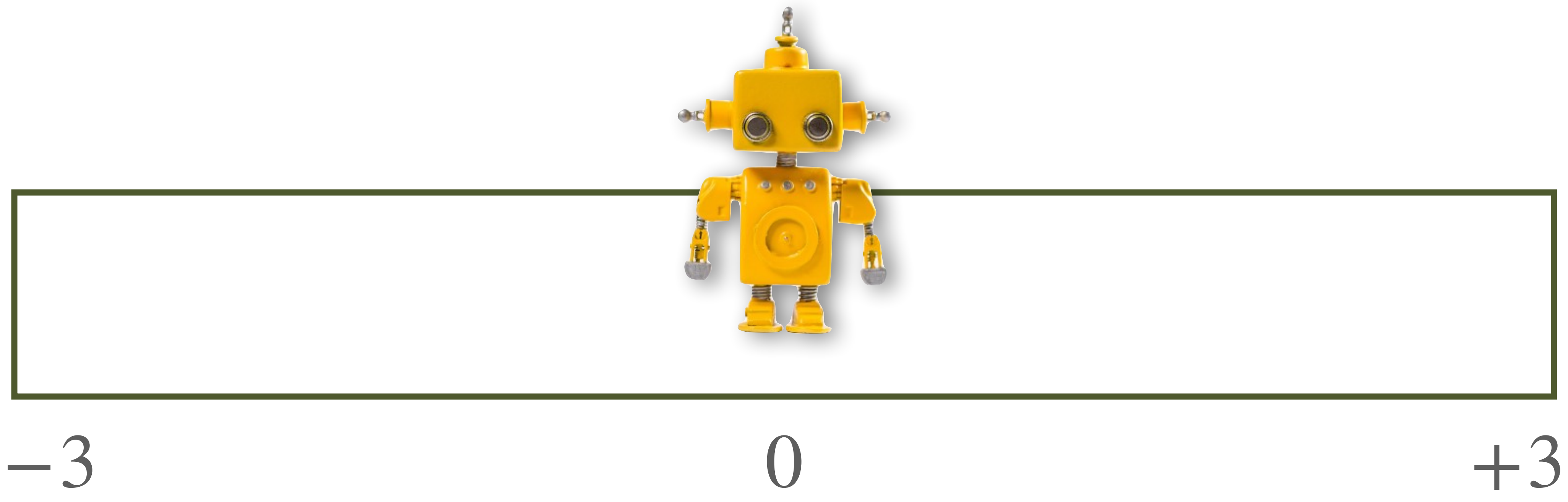
- Want to train a network $V^\pi_\phi(s)$ w/ parameters ϕ
  - inputs: state info, e.g., stack of images
  - output: value estimate
- Data: follow π, observe one or more trajectories $s_1, a_1, r_1, s_2, a_2, r_2, \ldots$
- Each trajectory yields several training examples

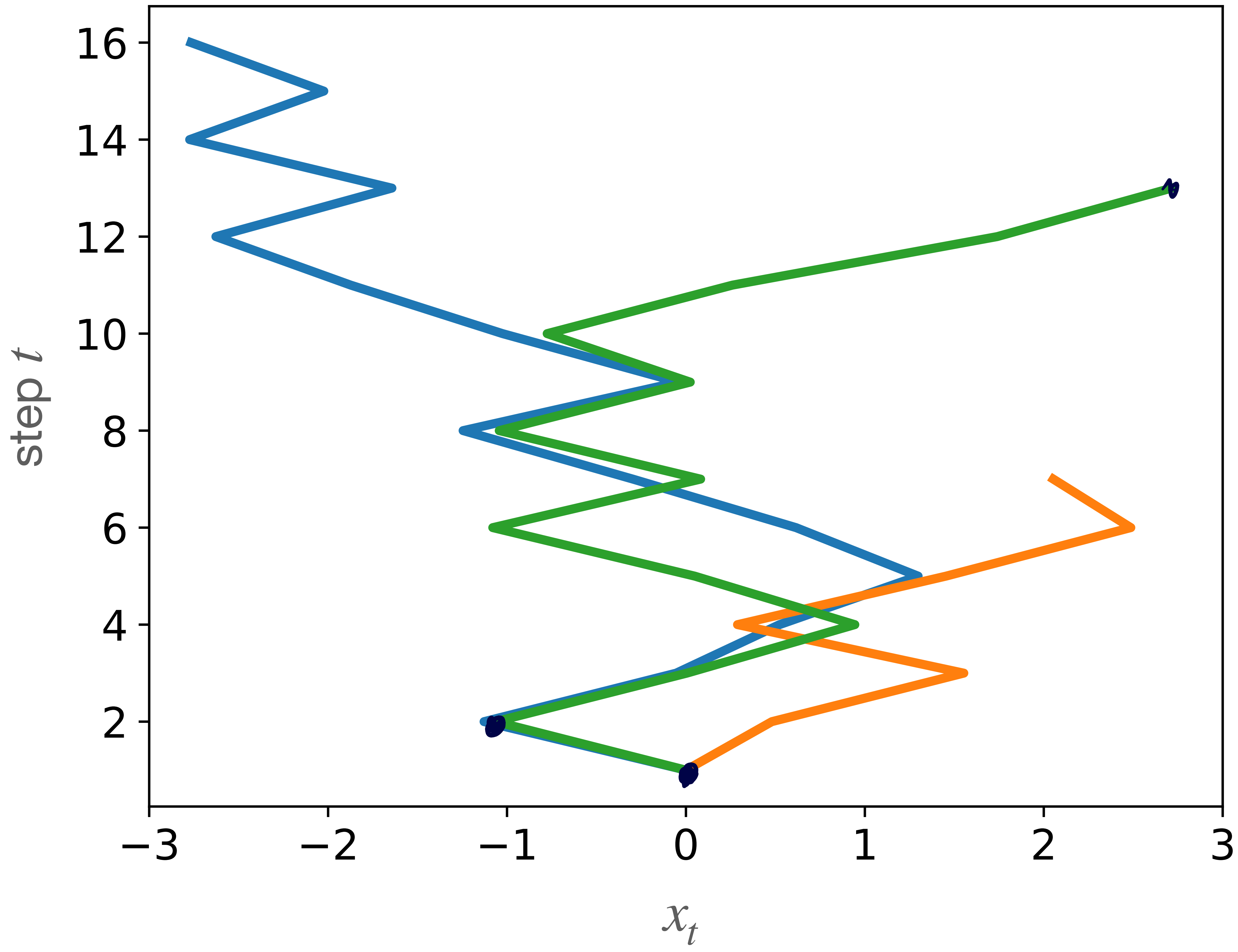
Learning V^{π}: example



- 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

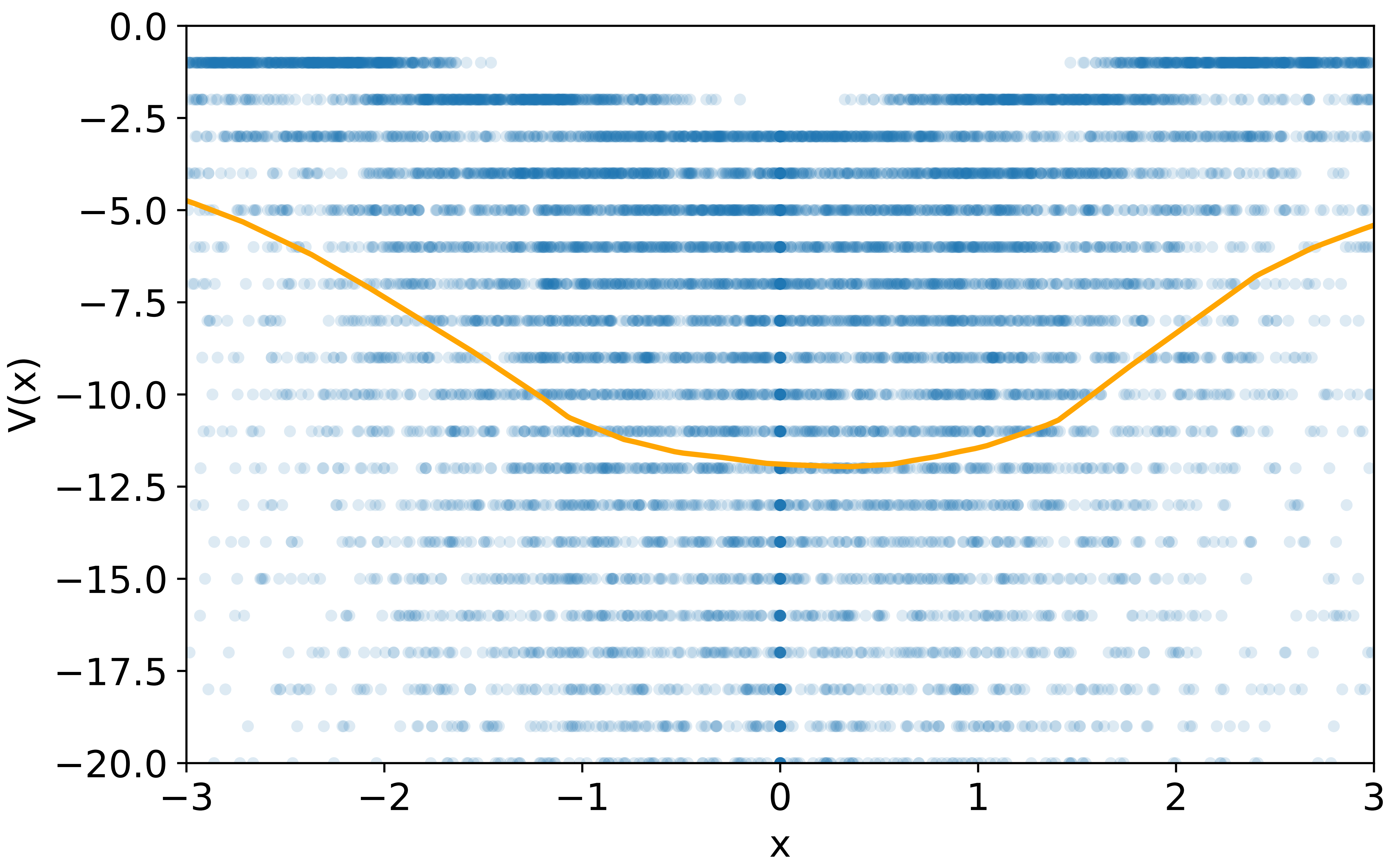




(x, V(x))
(0, −13)
(−1.1, −12)
(2.8, −1)

Each trajectory yields several training examples

Learned V^{π}



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

Fixed point iteration

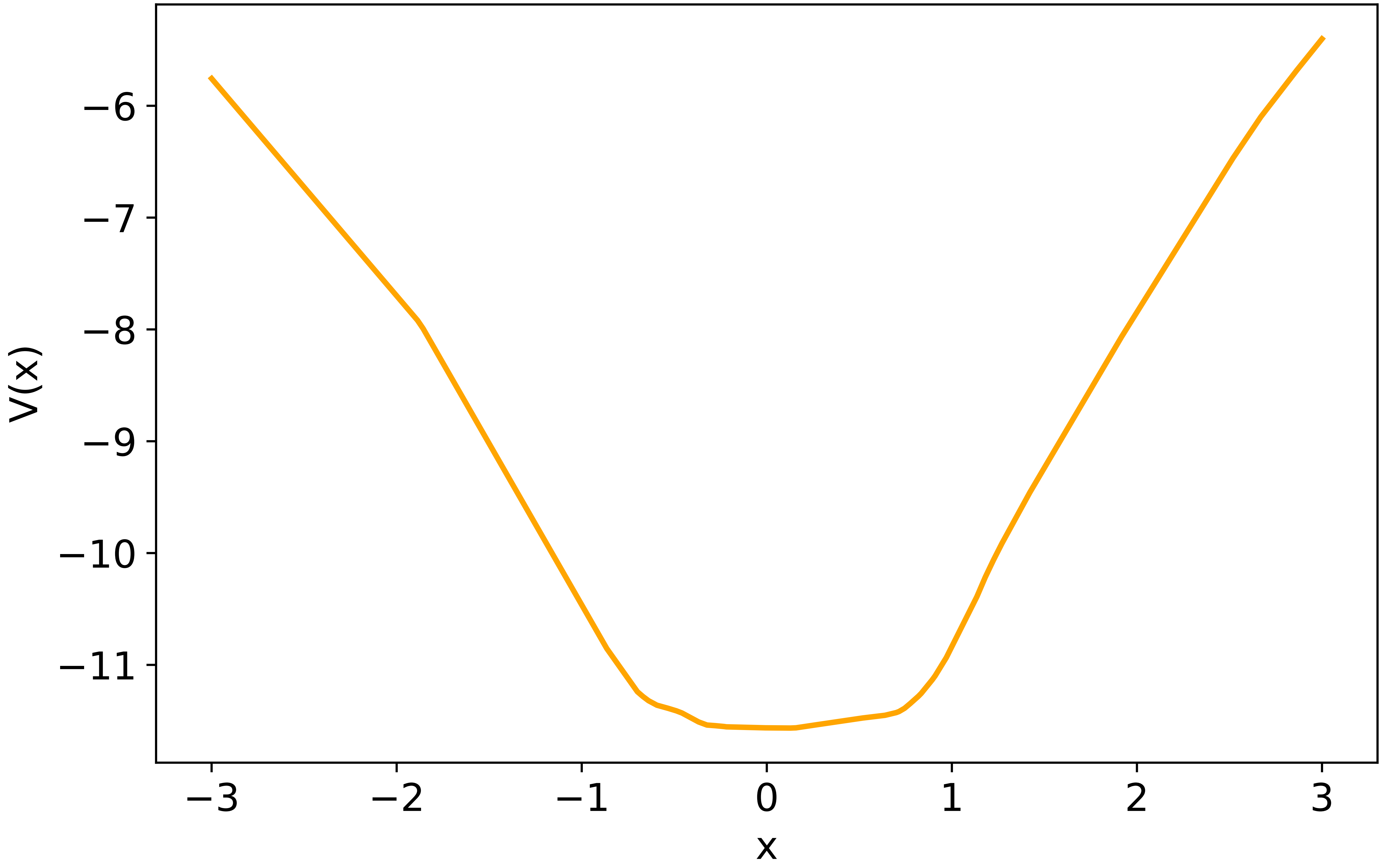
- Suppose we've already learned estimated parameters ϕ_1
- Observe s, r, s'
  - Bellman equation: $V^\pi(s) = \mathbb{E}[r + \gamma V^\pi(s')]$
  - Fixed point iteration: train $V^\pi_\phi(s) \approx r + \gamma V^\pi_{\phi_1}(s')$
  - i.e., SGD on ϕ to minimize MSE [note: ϕ_1 fixed]
- After a while, set $\phi_2 = \phi$
  - reinitialize ϕ and train $V^\pi_\phi(s) \approx r + \gamma V^\pi_{\phi_2}(s')$
- Repeat
- Fixed ϕ_i called *target network*

Temporal difference learning

- Hyperparameter: how often do we update target network?
  - every 10 trajectories? every 100?
- If we update after every SGD step, get *TD(0)* algorithm
  - in this case, no need to store ϕ_i separately
  - probably the best choice: in this context the only effect of waiting to update is to slow down learning
  - in other RL methods, slower target network updates can help convergence and performance
- By contrast, supervised regression method is called *TD(1)*

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

TD(0) example



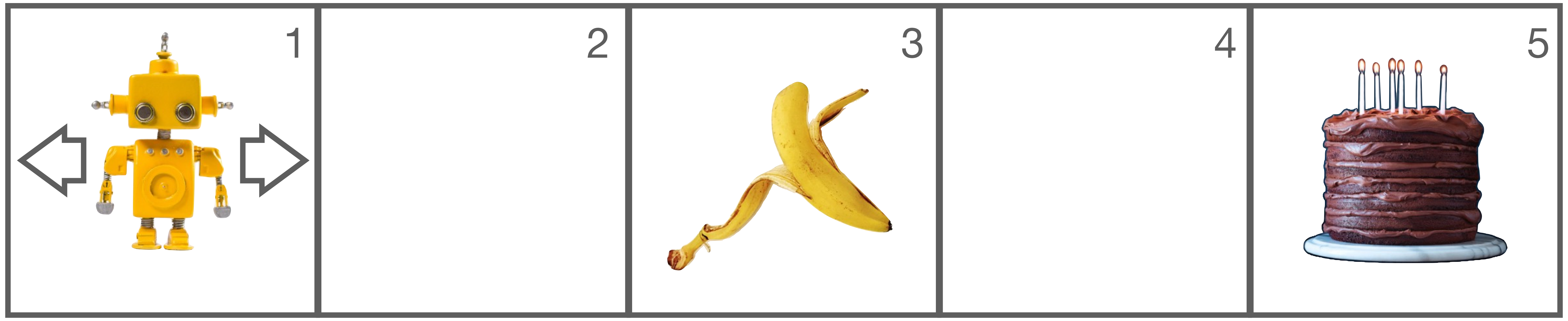

```
for _ in range(3000):
    xs, rs = trajectory()
    T = len(xs)
    with torch.no_grad():
        tgt = [rs[t] + model(xs[t+1]) for t in range(T−1)] + [rs[T−1]]
    err = 0.0
    for t in range(T):
       err += criterion(model(xs[t]), tgt[t])
    optimizer.zero_grad()
    err.backward()
    optimizer.step()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

State values vs. action values

- V^π tells us what states are good to be in
- What if we want to know what actions are good to take?
- Definition: the ***action-value*** function is
  - $Q^\pi(s,a) = \mathbb{E}_\pi(r_1 + \gamma r_2 + \gamma^2 r_3 + \ldots \mid s_1 = s, a_1 = a)$
- Cf. definition of ***(state-)value*** function V^π:
  - $V^\pi(s) = \mathbb{E}_\pi(r_1 + \gamma r_2 + \gamma^2 r_3 + \ldots \mid s_1 = s)$

Value function example

Environment



r=–1	r=–1	r=–10	r=–1	r=0

pay this much cost when leaving state

Policy π: always move right

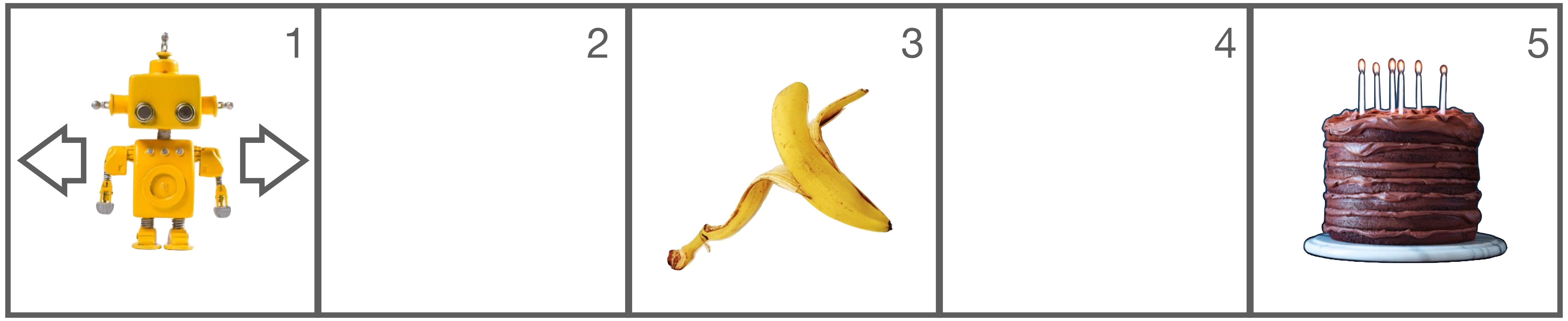
Table of V^π

$V(1) = -13$	$V(2) = -12$	$V(3) = -11$	$V(4) = -1$	$V(5) = 0$

Geoff Gordon

Q function example

Environment



r=–1	r=–1	r=–10	r=–1	r=0

Policy π: always move right

Table of Q^{π}

$Q(1,r) = -13$	$Q(2,r) = -12$	$Q(3,r) = -11$	$Q(4,r) = -1$	$Q(5,r) = 0$
$Q(1,\ell) = -14$	$Q(2,\ell) = -14$	$Q(3,\ell) = -22$	$Q(4,\ell) = -12$	$Q(5,\ell) = -1$

Geoff Gordon

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 ~~costs~~ reward 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 target network, train LHS by SGD step
- These are called *SARSA(1)* and *SARSA(0)*
  - SARSA = s, a, r, s', a'

Policy improvement

- In tabular case (previous lecture), we can just update π to be the greedy policy for Q^π

$$\pi^{\text{gr}}(s) = \arg\max_a Q^\pi(s, a)$$

- Could do the same here, ***but***
  - there will be errors in our estimate of Q^π
  - so switching to π^{gr} is too aggressive
- Instead we want to take a small step to improve π
- Q: could we switch to (or take a step toward) the greedy policy for V^π instead?
  - A: want $\arg\max_a r(s,a) + \gamma \mathbb{E}(V(s') \mid s,a)$

How to improve our policy?

- Want to maximize $J(\theta) = \mathbb{E}_{\pi_\theta}[r_1 + r_2 + \ldots + r_T]$
- Maybe the simplest idea: analogous to SGD
  - initialize policy parameters $\theta^1 \in \mathbb{R}^d$
  - on training iteration $m = 1,2,\ldots$:
    - compute stochastic estimate $g^m \approx \frac{d}{d\theta} J(\theta) \Big|_{\theta^m}$
    - 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, but other setups are analogous

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

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

Policy gradient theorem

- Observe trajectory $\tau^m = (s_1^m, a_1^m, r_1^m \ldots, s_T^m, a_T^m, r_T^m)$ by following policy $\pi_\theta(a_t^m \mid s_t^m)$
- Define

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

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

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

Policy gradient theorem:

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

Policy gradient theorem

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

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

- Observe trajectory $\tau^m = (s_1^m, a_1^m, r_1^m \ldots, s_T^m, a_T^m, r_T^m)$ by following policy $\pi_\theta(a_t^m \mid s_t^m)$
- Define

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

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

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

Policy gradient theorem:

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

even if we don't know anything about the environment or state

Policy gradient theorem

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

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

- Observe trajectory $\tau^m = (s_1^m, a_1^m, r_1^m \ldots, s_T^m, a_T^m, r_T^m)$ by following policy $\pi_\theta(a_t^m \mid s_t^m)$
- Define

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

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

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

Policy gradient theorem:

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

even if we don't know anything about the environment or state

and ***even if*** environment is PO

Geoff Gordon

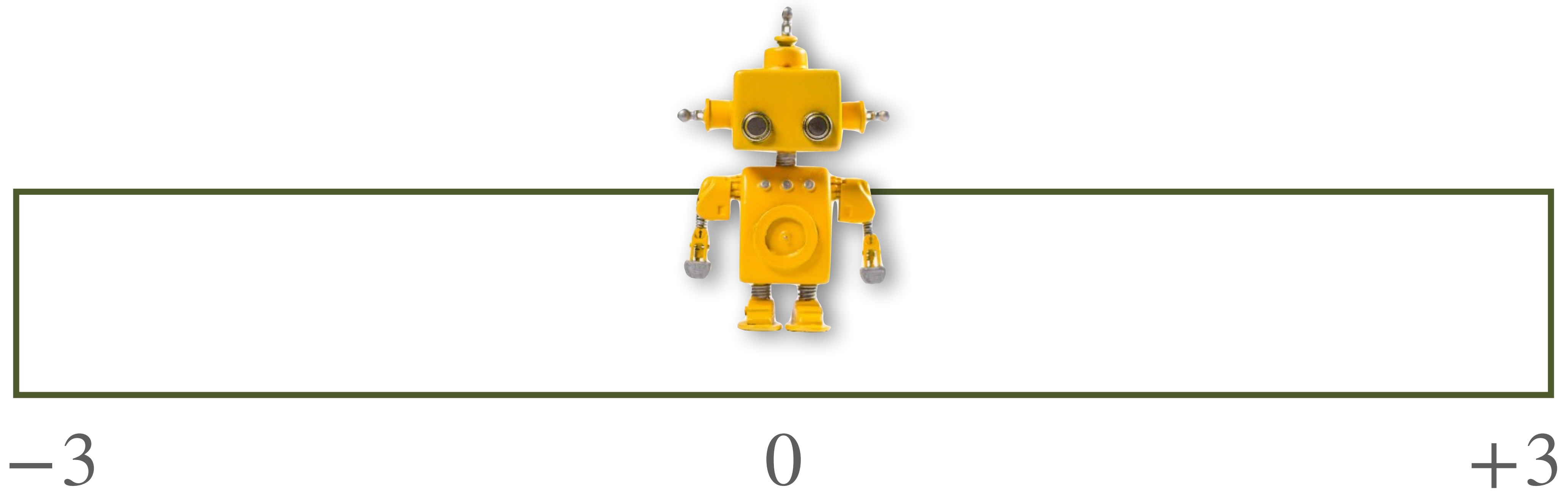
Policy gradient intuition

- Policy gradient: $g^m = \sum_{t=1}^{T} Q_t^m u_t^m$
- 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 Q_t^m: upweight score vector when reward is large, flip score vector if reward 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 one of the oldest RL algorithms: *REINFORCE* [Williams, 1992]
- Repeat:
  - gather some trajectories under current policy π_θ
  - compute gradient estimate g by policy gradient theorem
  - 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

REINFORCE example



- Policy:

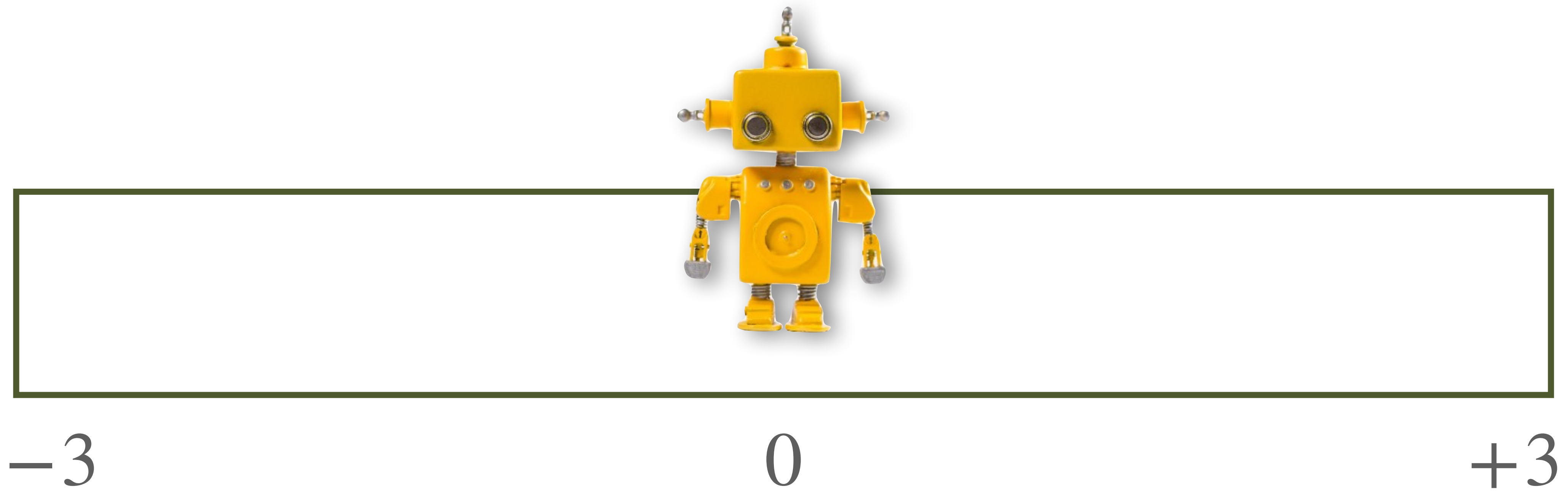
$$P(\mathsf{R} \mid x) = \frac{1}{1 + e^{-(wx+b)}}$$

$$P(\mathsf{L} \mid x) = \frac{1}{1 + e^{wx+b}}$$

$$\theta = (w, b)^\top$$

Geoff Gordon

Action logits

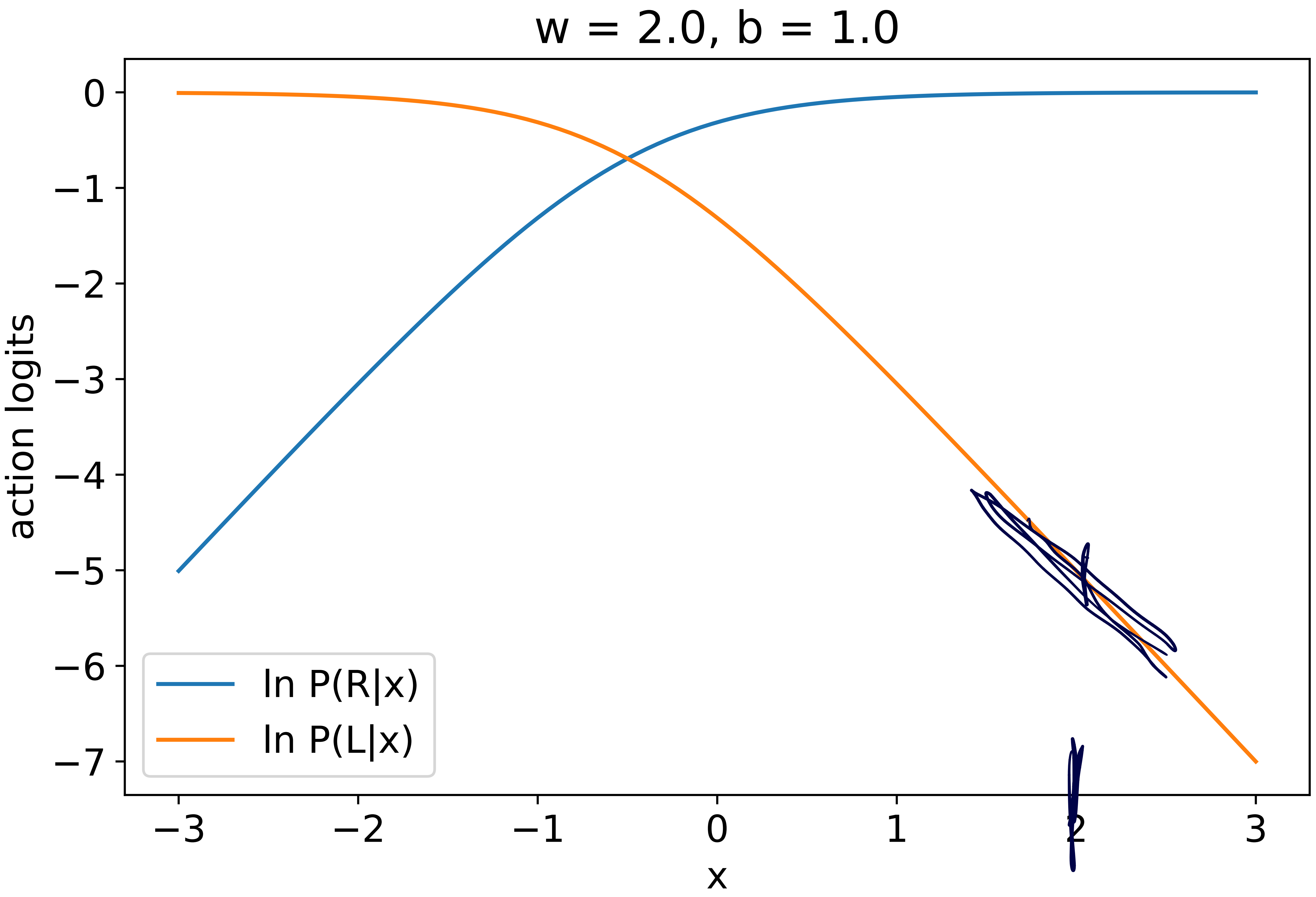


$\nabla_\theta \ln P(\mathsf{L} \mid x = 2) =$

$\nabla_b \uparrow = -1$

$\nabla_w \uparrow = -1 \cdot x$

$= -2$



Action scores

updated

$$\frac{1}{1+e^{-(w\cdot x+b)}}$$

- $\nabla \ln P(\mathsf{R} \mid x) = -\nabla \ln\left(1+e^{-(w\cdot x+b)}\right)$

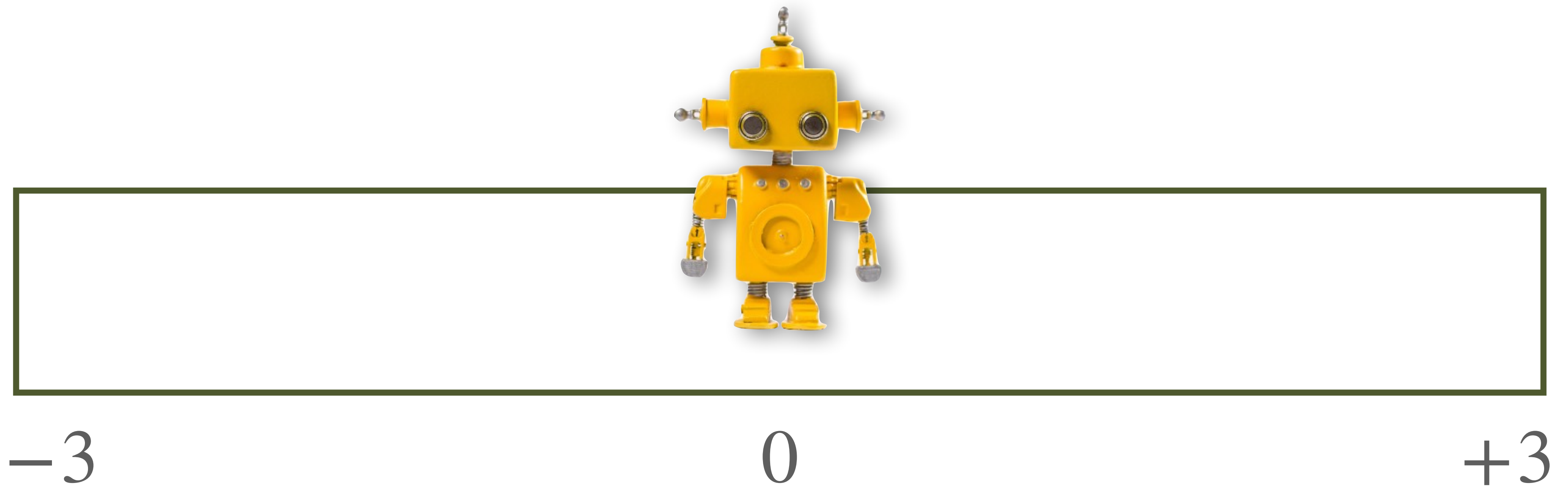
$$= -\frac{1}{1+e^{-w\cdot x+b}}\, e^{-(w\cdot x+b)}\left(-\nabla(w\cdot x+b)\right) = \underbrace{\frac{1}{1+e^{w\cdot x+b}}}_{P(L|x)}\begin{pmatrix} x \\ 1 \end{pmatrix}$$

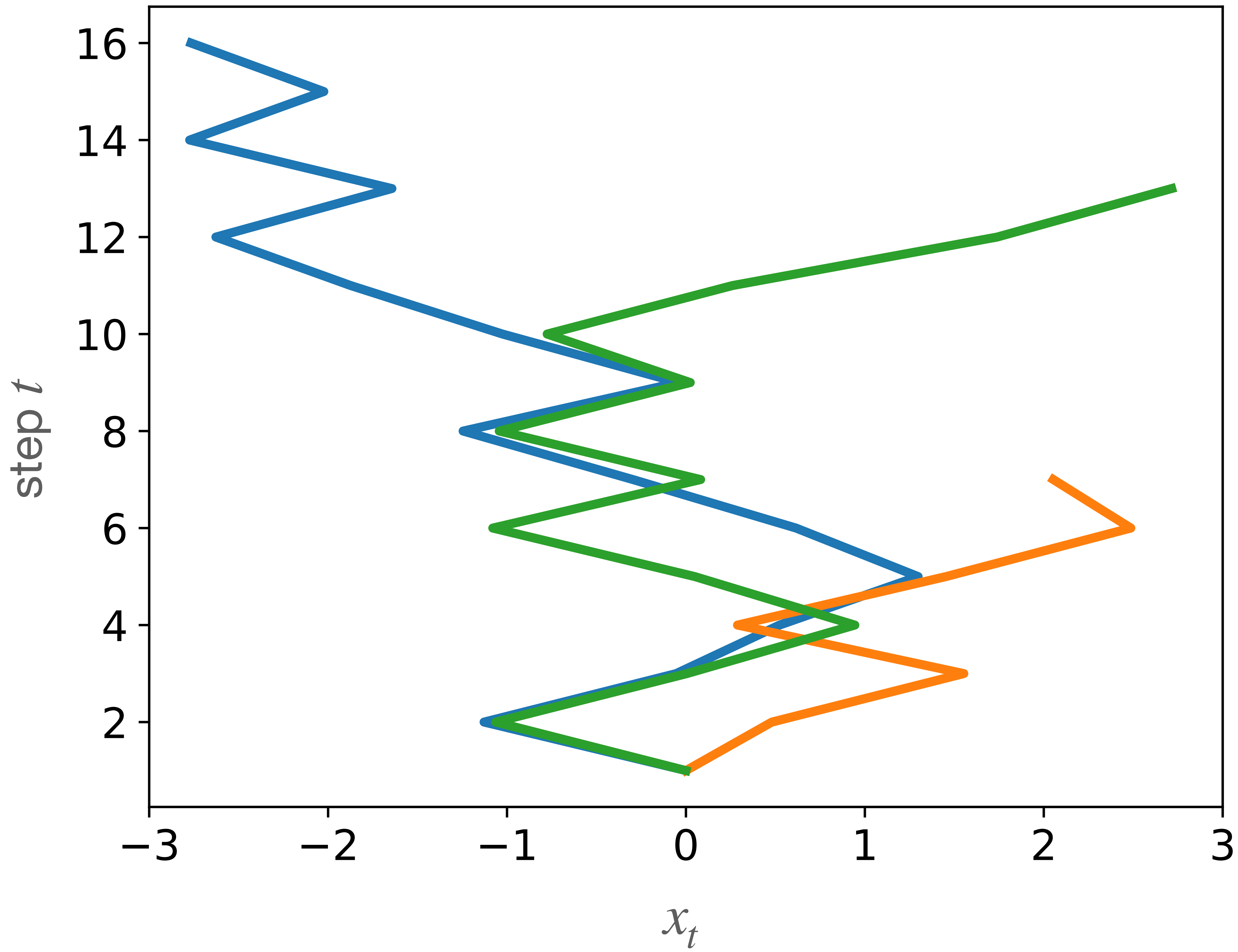
$$\frac{1}{1+e^{w\cdot x+b}}$$

- $\nabla \ln P(\mathsf{L} \mid x) = -\nabla \ln\left(1+e^{w\cdot x+b}\right)$

$$= -\frac{1}{1+e^{w\cdot x+b}}\, e^{w\cdot x+b}\left(\nabla(w\cdot x+b)\right) = -\underbrace{\frac{1}{1+e^{-(w\cdot x+b)}}}_{P(R|x)}\begin{pmatrix} x \\ 1 \end{pmatrix}$$

Calculate gradient estimate



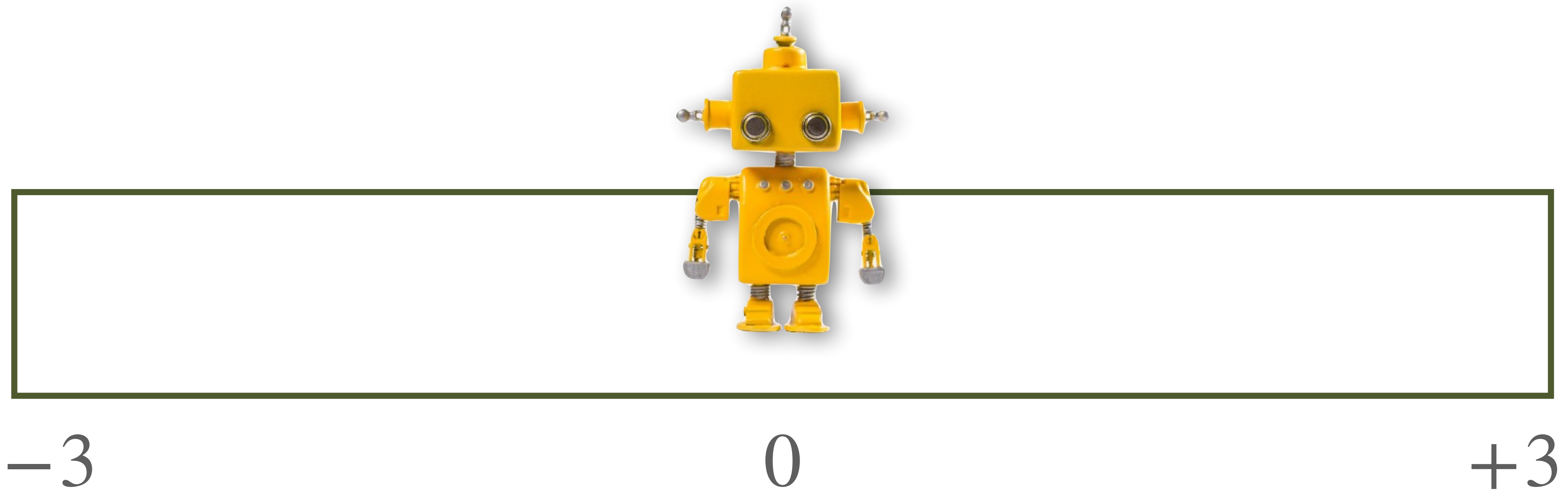


(x, a)	Q
(0, R)	−7
(.3, R)	−6
(1.6, L)	−5
(.2, R)	−4
(1.3, R)	−3
(2.4, L)	−2
(2.1, R)	−1

- Start at $w = b = 0$ (so π is uniform random at all x)

Calculate gradient estimate

$\theta \in \mathbb{R}^2$



(x, a)	Q
(0, R)	−7
(.3, R)	−6
(1.6, L)	−5
(.2, R)	−4
(1.3, R)	−3
(2.4, L)	−2
(2.1, R)	−1

$\in \mathbb{R}^2$

$$g = -7 \cdot 0.5 \begin{pmatrix} 0 \\ 1 \end{pmatrix} - 6 \cdot 0.5 \begin{pmatrix} .3 \\ 1 \end{pmatrix} - 5 \cdot 0.5 \begin{pmatrix} -1.6 \\ -1 \end{pmatrix} \cdots$$

updated

$$\nabla \ln P(\mathsf{R} \mid x) = P(\mathsf{L} \mid x) \begin{pmatrix} x \\ 1 \end{pmatrix}$$

$$\nabla \ln P(\mathsf{L} \mid x) = P(\mathsf{R} \mid x) \begin{pmatrix} -x \\ -1 \end{pmatrix}$$

Behavior of REINFORCE and TD learning

- REINFORCE and TD learning are simple to implement
- Can work well both in practice and in theory:
  - In practice, handle moderate-horizon tasks w/ dense reward
  - Led to models of animal behavior that were used to explain operant conditioning experiments
  - Theorem: w/ sufficiently small and decreasing learning rate,
    - REINFORCE will converge to a local optimum of $J(\theta)$
    - TD(1)/SARSA(1) will converge to locally min-MSE estimate of V^π or Q^π
    - Slightly more complicated results for TD(0)/SARSA(0)
- ***But…***

Failure modes of REINFORCE

- Exploration
  - We get a nonzero gradient only if cost/reward is nonzero
  - If feedback is sparse and we start with a random policy, might wander forever without learning anything
  - Imagine: learning to cross a tightrope
- Cancellation (low SNR)
  - Total return Q can scale with horizon, and can vary a lot due to randomness in policy, environment
  - Overall gradient can be much smaller (terms w/ opposite signs)
- Getting stuck
  - When policy gets close to border of simplex, score vectors for unlikely actions get large, probability of seeing them gets small
  - In the limit, $\infty \cdot 0$ (have to sample a really long time to average!)

Failure modes of REINFORCE

- Exploration
  - We get a nonzero gradient only if cost/reward is nonz
  - If feedback is sparse and we start with a ran ... t wander forever without learning anythi
  - Imagine: learning to cross a ti
- Cancellation (low SN
  - Total retur ... can vary a lot due to rand
  - Ove ... h smaller (terms w/ opposite signs)
- Getting
  - When policy gets close to border of simplex, score vectors for unlikely actions get large, probability of seeing them gets small
  - In the limit, $\infty \cdot 0$ (have to sample a really long time to average!)

Failure modes multiply: might have to explore a long time to find feedback, then do it over to average out variance from cancellation, then over again to compensate for being stuck

Upshot: if we try to scale, can only use tiny learning rate

Reducing variance

- Policy gradient: $g = \sum_{t=1}^{T} Q_t u_t$ (suppressing trajectory index m)
  - stochastic gradient $g \in \mathbb{R}^d$ for this trajectory
  - at each step t, total future reward or cost $Q_t \in \mathbb{R}$
  - score vector $u_t = \frac{d}{d\theta} \ln \pi_\theta(a_t \mid s_t)$
- Problem is high variance in g due to randomness in policy and environment
  - Q_t depends on future trajectory, u_t depends on action
  - both can be large compared to g (cancellation)
- To reduce variance, replace random quantities with their expectations where possible

REINFORCE: $g = \sum_{t=1}^{T} Q_t u_t$

Actor-critic

- Reduce variance: replace Q_t by conditional expectation
  - $\mathbb{E}(Q_t \mid s_t, a_t) = Q^\pi(s_t, a_t)$
  - $g_{\text{exactQ}} = \sum_{t=1}^{T} Q^\pi(s_t, a_t) u_t$ — usually not implementable
- Or use learned approximation
  - $g_{\text{AC}} = \sum_{t=1}^{T} Q_\phi^\pi(s_t, a_t) u_t$
  - unsafe: introduces bias to gradient estimate due to Q_ϕ^π
  - but still can be highly successful
- Bias means we may make policy worse instead of better
  - if bias is enough to alter gradient more than $90°$
  - happens if we already have a good policy, ***or*** if we have a very small gradient signal (e.g., sparse costs/rewards)

Actor-critic: $g = \sum_{t=1}^{T} Q_\phi^\pi(s_t, a_t) u_t$

Actor-critic

- Can train π and Q_ϕ^π simultaneously
  - π is called the *actor,* and Q_ϕ^π is called the *critic*
  - typically, want critic to learn faster — big policy changes can cause instability in learning
- Qualitatively:
  - critic always trying to accurately evaluate state, action value
  - actor: a step in direction u_t will increase probability of a_t, so
  - any a associated w/ higher $Q(s_t, a)$ will increase in probability
- Gradually tries to make policy greedier (more likely to take action $\arg\max_a Q(s_t, a)$)

$u_t = \frac{d}{d\theta} \ln \pi_\theta(a_t \mid s_t)$

Reduce variance using $\mathbb{E}$[score]

- Next idea: action score vector has expectation 0
- Derivation:

$$\begin{aligned}
\mathbb{E}(u_t \mid s_t) &= \sum_a \pi_\theta(a \mid s_t) \frac{d}{d\theta} \ln \pi_\theta(a \mid s_t) \\
&= \sum_a \pi_\theta(a \mid s_t) \frac{1}{\pi_\theta(a \mid s_t)} \frac{d}{d\theta} \pi_\theta(a \mid s_t) \\
&= \sum_a \frac{d}{d\theta} \pi_\theta(a \mid s_t) \\
&= 0 \qquad \leftarrow \text{differentiate both sides of } 1 = \textstyle\sum_a \pi_\theta(a \mid s_t)
\end{aligned}$$

Baseline

- Let $B(s)$ be any function of state

$$g = \sum_t Q_t u_t = \sum_t (Q_t - B(s_t) + B(s_t))\, u_t$$

$$\begin{aligned}\mathbb{E}[g] &= \mathbb{E}\left[(Q_t - B(s_t) + B(s_t))\, u_t \mid s_t\right] \\ &= \mathbb{E}\left[(Q_t - B(s_t))\, u_t \mid s_t\right] + B(s_t)\, \mathbb{E}\left[u_t \mid s_t\right] \\ &= \mathbb{E}\left[(Q_t - B(s_t))\, u_t \mid s_t\right]\end{aligned}$$

  - s_t is constant in $\mathbb{E}(\,\cdot \mid s_t)$, lets us pull out $B(s_t)$
  - Used known expectation of u_t to eliminate last term
- Replace $Q_t u_t \to (Q_t - B(s_t))u_t$
  - same conditional expectation
  - could be lower or higher variance, depending on B

Baseline

- New gradient estimate:

  $$g = \sum_t (Q_t - B(s_t))\, u_t$$

- B is called a *baseline*: we are comparing total cost to $B(s_t)$ instead of to 0
  - recover old method by setting $B(s) = 0 \quad \forall s$
- Good baseline: $B(s_t)$ should be close to $\mathbb{E}(Q_t \mid s_t)$ so that term in parentheses is small
  - can't use $Q(s_t, a_t)$ since $B(s_t)$ doesn't depend on a_t
  - if we've learned a value function estimate, $V^{\pi}_{\phi}(s_t)$ fits the bill

Baseline vs. actor-critic, and AAC

- REINFORCE w/ baseline vs. actor-critic:
  - baseline: no bias (above derivation is exact)
  - but higher variance: more randomness remains in $Q_t - V^\pi_\phi(s_t)$ than in $Q^\pi_\phi(s_t, a_t)$
- What if both?
  - $g = \sum_t (Q^\pi_\phi(s_t, a_t) - V^\pi_\phi(s_t))\, u_t$
  - even less variance than actor-critic, but bias remains
  - $A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$ is called the *advantage* of a in s
    - large $A^\pi(s, a)$ means it's advantageous to take a in s (vs. following π)
  - so, using this g is called *advantage* actor-critic (AAC)

Scaling up RL

- Any function we learn (policy, value, environment model): scale up w/ standard techniques (model parallelism, data parallelism, parameter server, …)
- RL-specific scaling:
  - Even with variance-reduction techniques, variance of a policy gradient estimate is high (much higher than typical for SGD)
  - Means we can take advantage of bigger batch sizes → better ratio of computation to communication
  - E.g., useful for RL to run 300 trajectories of length 300 on each worker and reduce (about 10^5 points, vs. our earlier example of diminishing returns after ~10 points)
- For parallelism, need to generate trajectories in silico
  - a simulator
  - or a purely computational task like Go or Minecraft
  - or a giant farm of robots surrounded by safety cones

Example: AlphaGo

- One component of AlphaGo is a policy trained by REINFORCE with baseline
- Go is fully observable, $s_t =$ the current Go board
- $V^\pi(s) =$ win probability for black, given board s with black to move, averaged across our pool of opponents
- Baseline = deep net trained to approximate V by TD(1) regression on a large dataset of positions from games

- One key component we didn't cover: during play, instead of learned π_θ or greedy $\arg\max_a(r(s,a) + V^\pi_\phi(\delta(s,a)))$, we look ahead several moves by randomized tree search (MCTS) — transforms from a Go player that beats most amateurs to one that beats Lee Sedol

Example: AlphaGo

- Training and play ran on a cluster of 50 GPUs
- Training:
  - supervised policy: minibatches of 16 positions, 340,000,000 iterations of async SGD via parameter server, ~1k GPU-days
  - REINFORCE: minibatches of 128 games, embarrassingly parallel; 10,000 iterations of synchronous policy gradient, 50 GPU-days
  - self-play data: generated 30,000,000 positions, each from a separate game, embarrassingly parallel
  - Value network: minibatches of 32 positions, 50,000,000 iterations of async SGD, parameter server, 350 GPU-days
- Play: custom parallel variant of MCTS

Example: generative language model

Consider for example the factorial function, which might be defined recursively as:

```
int factorial(int n) { if (n == 0) then return 1; else
return n * factorial(n-1); }
```

To provide a meaning for this recursive definition, the denotation is built up as the limit of approximations, where

credit: Wikipedia

- Generative language model: produce text by repeatedly choosing next word to fill in *(actually subword token)*
- Sequential decision problem: state = words so far, action = next word
- Train base model as a classifier on a giant corpus: e.g., internet crawl, Wikipedia, public Github repos
- The result is *not* what we want: it predicts what a random internet user would say next (bigoted, NSFW, cruel)

Example: generative language model

Consider for example the factorial function, which might be defined recursively as:

```
int factorial(int n) { if (n == 0) then return 1; else
return n * factorial(n-1); }
```

To provide a meaning for this recursive definition, the denotation is built up as the limit of approximations, where each

credit: Wikipedia

- Generative language model: produce text by repeatedly choosing next word to fill in *(actually subword token)*
- Sequential decision problem: state = words so far, action = next word
- Train base model as a classifier on a giant corpus: e.g., internet crawl, Wikipedia, public Github repos
- The result is *not* what we want: it predicts what a random internet user would say next (bigoted, NSFW, cruel)

Example: generative language model

Consider for example the factorial function, which might be defined recursively as:

```
int factorial(int n) { if (n == 0) then return 1; else
return n * factorial(n-1); }
```

To provide a meaning for this recursive definition, the denotation is built up as the limit of approximations, where each approximation

credit: Wikipedia

- Generative language model: produce text by repeatedly choosing next word to fill in *(actually subword token)*
- Sequential decision problem: state = words so far, action = next word
- Train base model as a classifier on a giant corpus: e.g., internet crawl, Wikipedia, public Github repos
- The result is *not* what we want: it predicts what a random internet user would say next (bigoted, NSFW, cruel)

Reinforcement learning from human feedback (RLHF)

- Make it better: train as an RL problem, where humans provide feedback signal
  - learn to generate what we actually want, instead of the worst of the internet
- Many ways to set up feedback (human's rating problem)
  - e.g., give two complete generations, ask which is preferred
  - train regression to predict score that determines P(preferred)
  - use learned score as delayed reward for RL, improve generation policy (next-word picker)
- Can use a *much* smaller dataset to train reward model (feasible to create with paid human raters)
- Often RL method is a variant of policy gradient, scaled by running simulations in parallel across many workers (share reward model (once), policy parameters (once each batch))

Another fun example