Machine Learning Department
School of Computer Science
Carnegie Mellon University

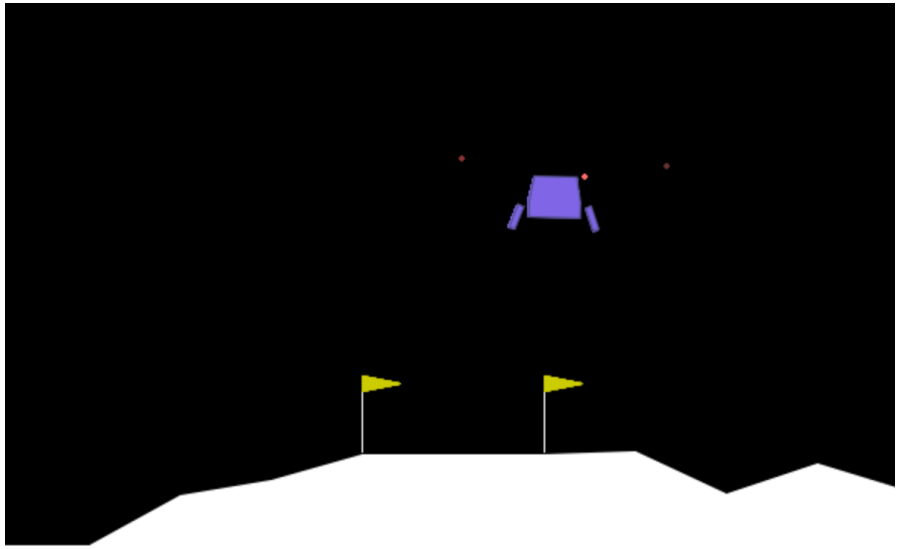
Policy Gradient + Deep Reinforcement Learning

Matt Gormley
Lecture 22
Apr. 1, 2026

The setting…

DEEP REINFORCEMENT LEARNING

Scaling up RL



Lunar lander

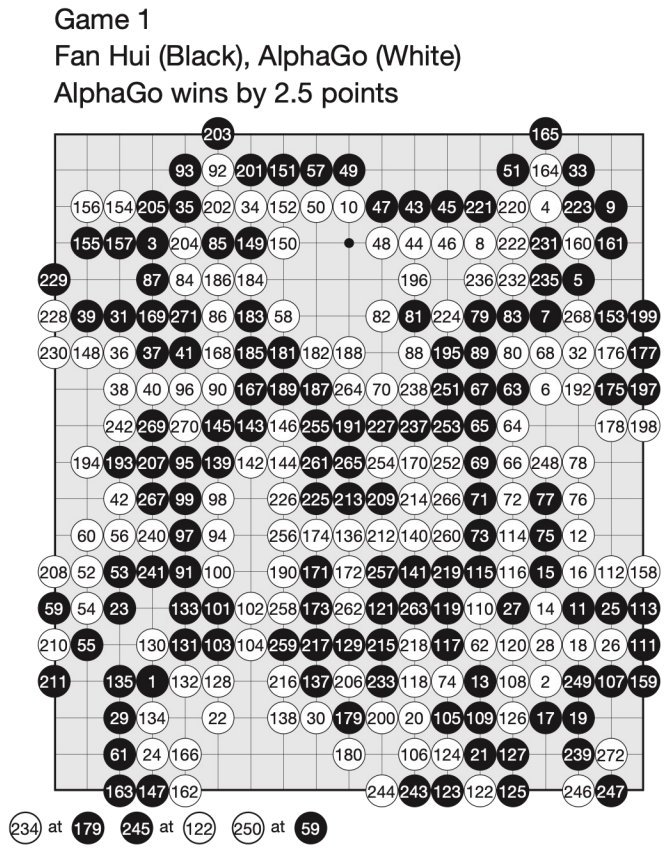
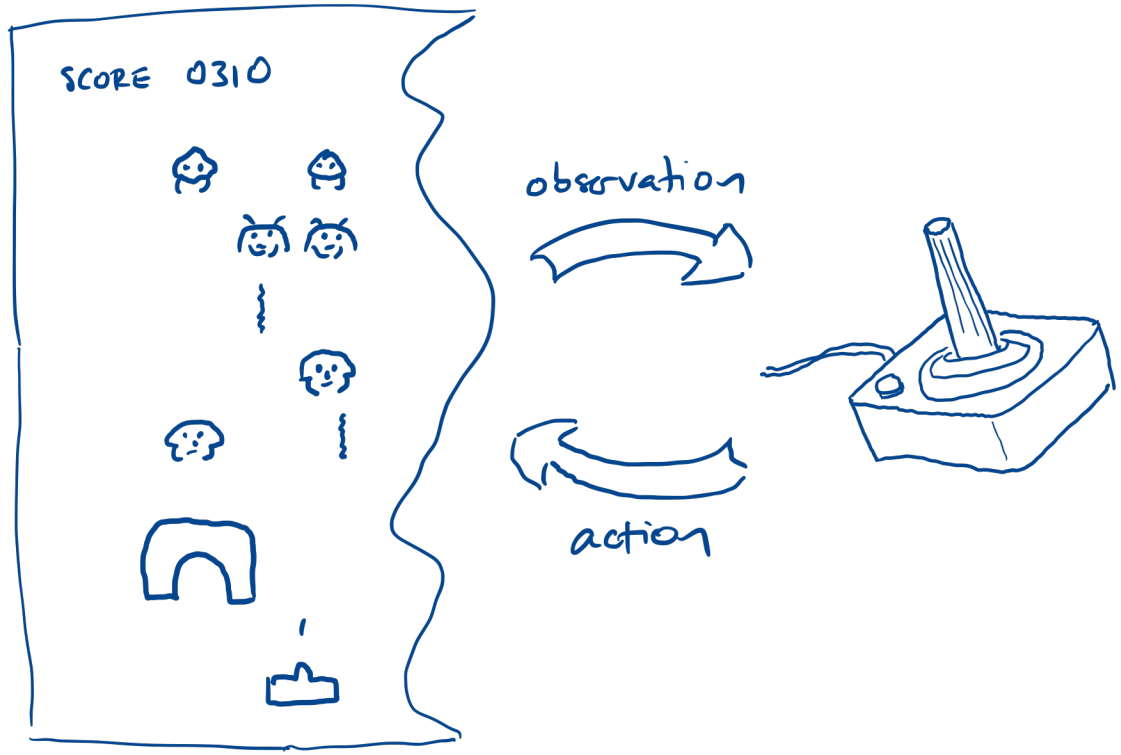


Image credit: Silver et al., Nature, 2016

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

No model

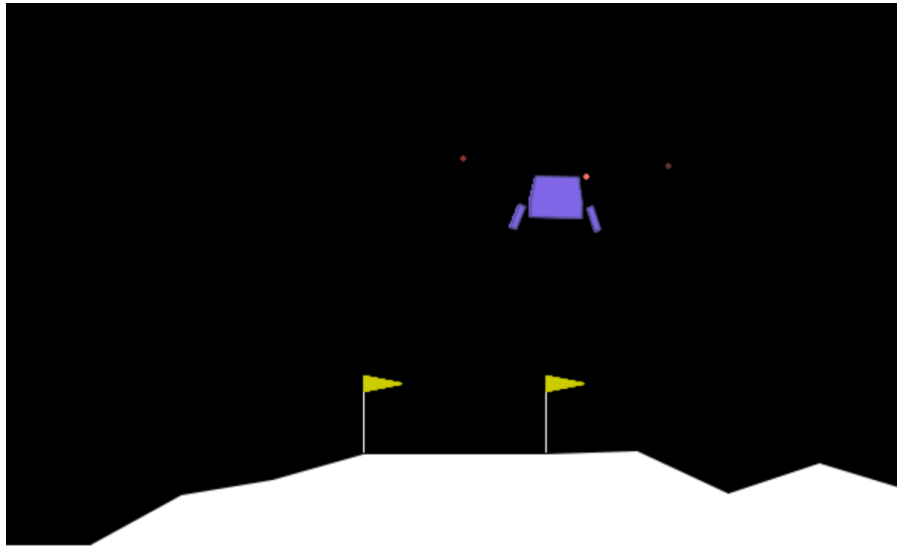
(Model-based vs. Model-free RL)



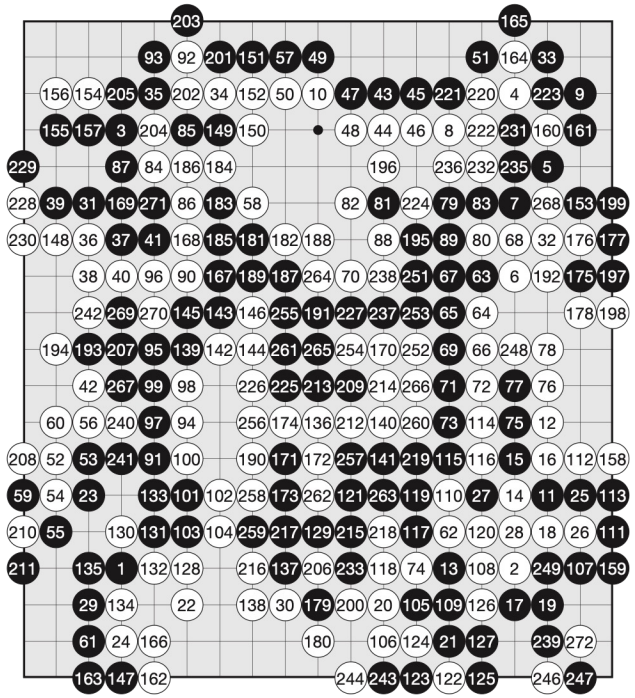
- **Model-based RL:** Previously: we knew description of the world, e.g., expressions for $R(s, a)$ or $P(s' \mid s, a)$
- **Model-free RL:** Instead: agent just interacts with environment over time — if we want $R(s, a)$ etc., have to learn it from data
- Alternating observations, actions, rewards $o_1,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

called a *trajectory*

Example environments

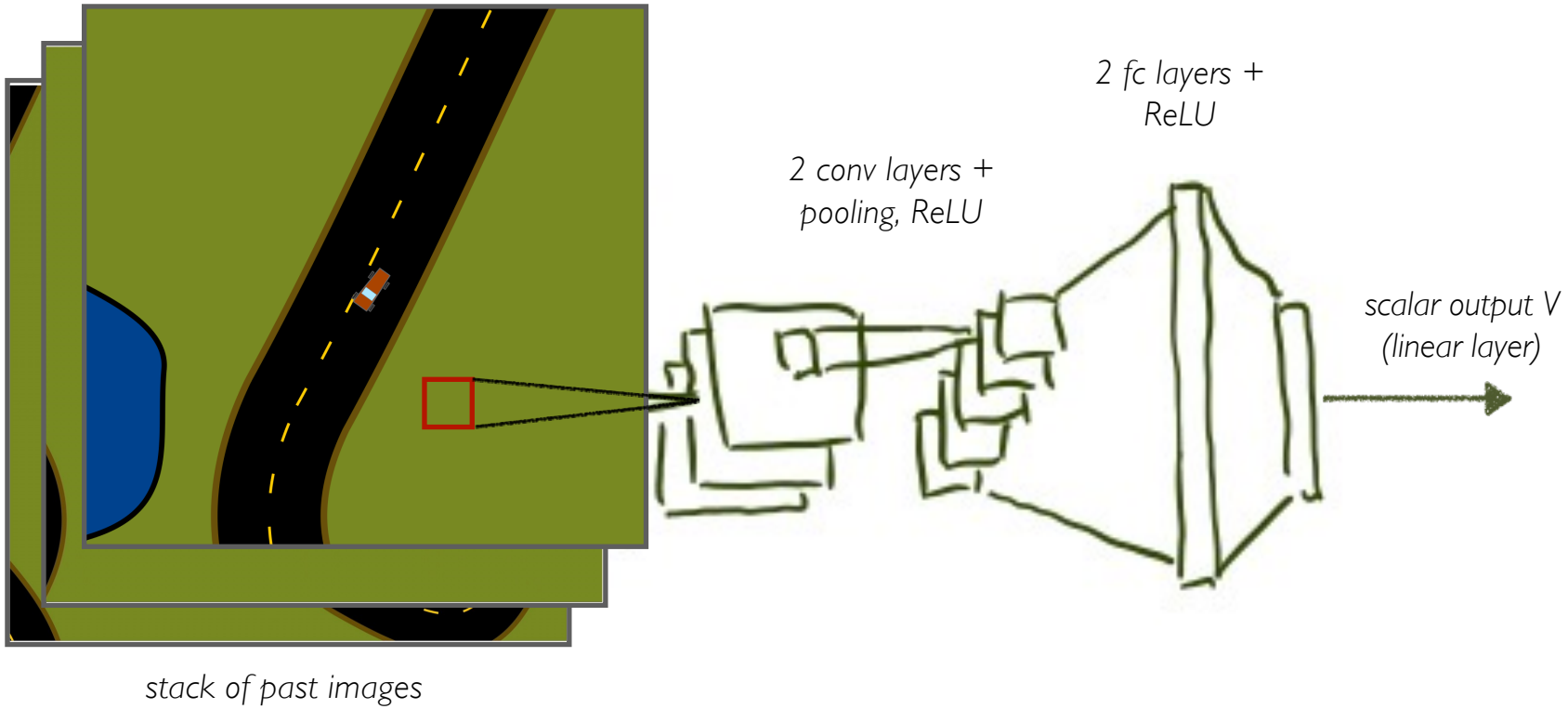


observations: screen images
actions: controller buttons, joystick position
transitions: determined by game code
reward: score increase



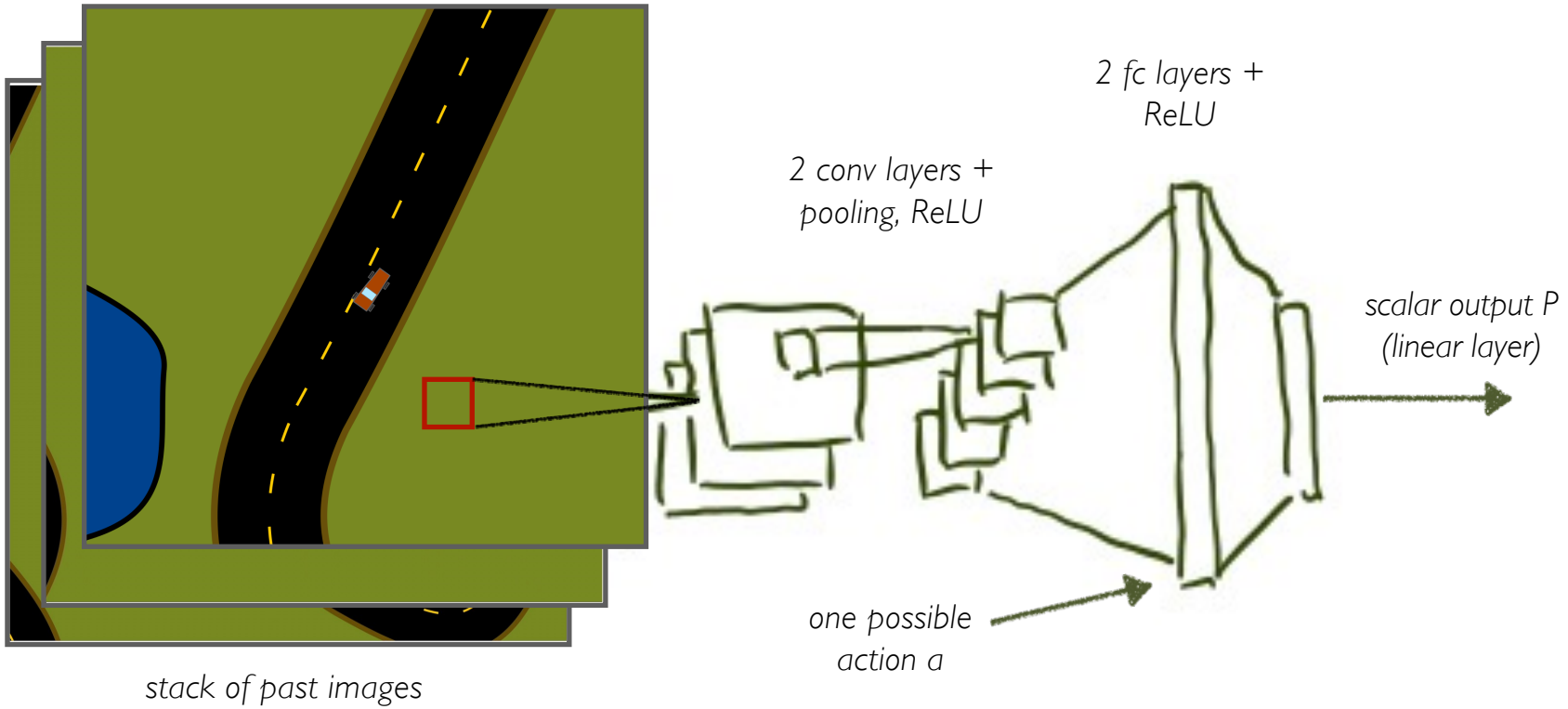
observations: board $\{B, W, \emptyset\}^{19\times19}$
actions: place a stone
transitions: rules of Go, opponent follows a previous policy (self-play)
reward: +1 for win, –1 for loss, 0 for draw, 0 if game isn't over

Learned, approximate functions



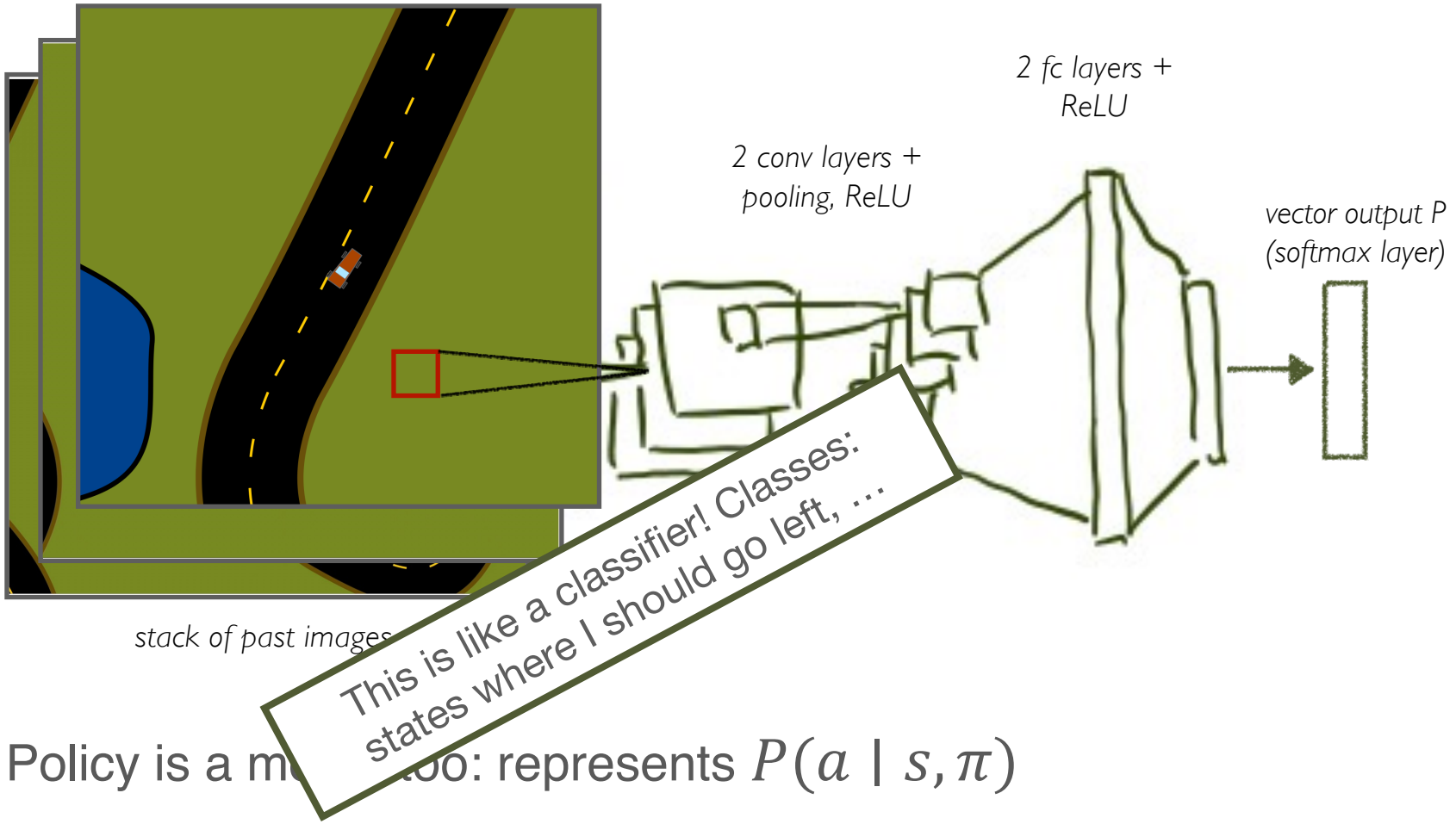
- **Tabular RL:** Previously: could list out all states, keep a table of a function like $V^{\pi}(s)$
- **RL with Function Approximation:** Now: any function we care about has to be represented as an ML model, e.g., a deep net
- One parameter vector per function we care about, each fn can have its own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

Policy



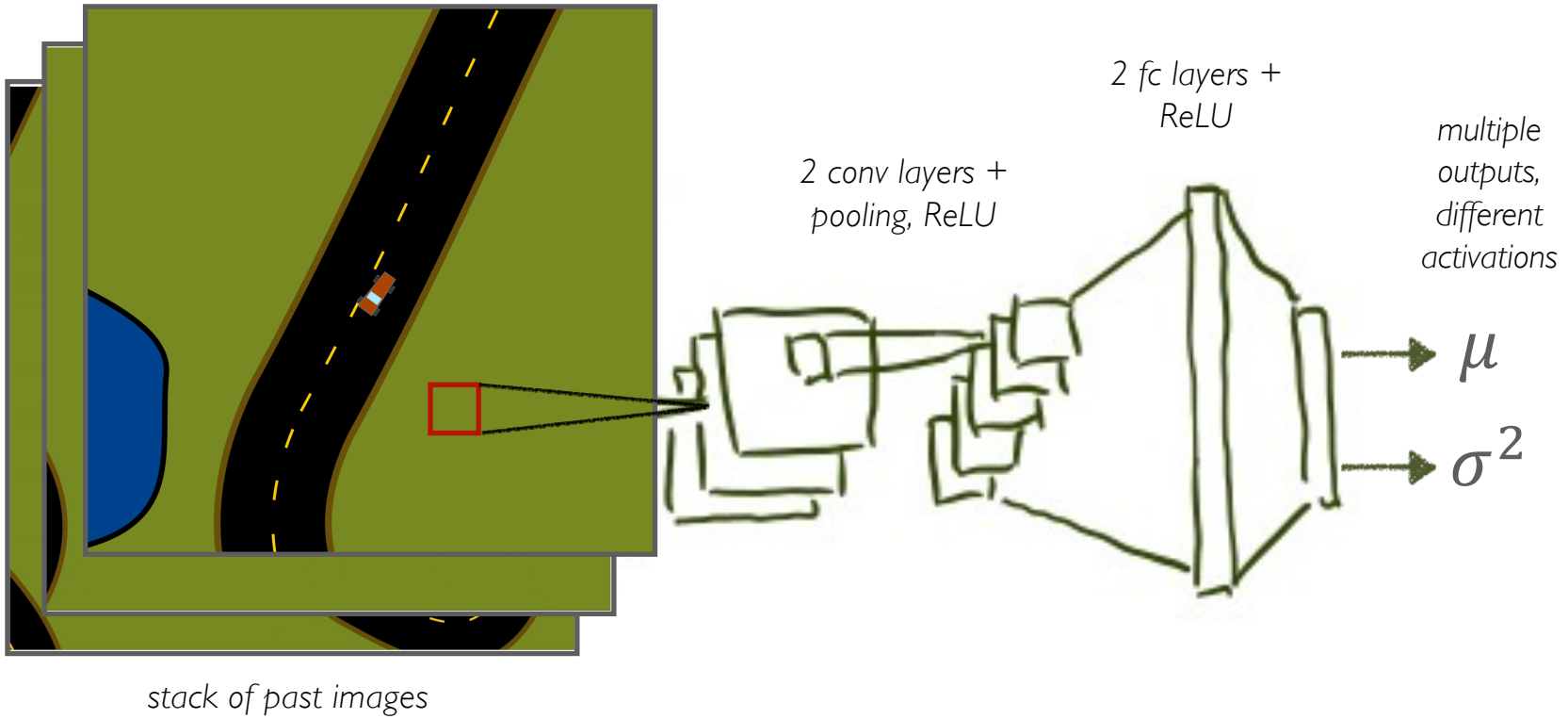
- Policy is a model too: represents $P(a \mid s, \pi)$
  - note: stochastic! (lets an optimizer make small changes)
- Several common ways to set up:
  - $s, a \mapsto P(a \mid s, \pi)$

Policy



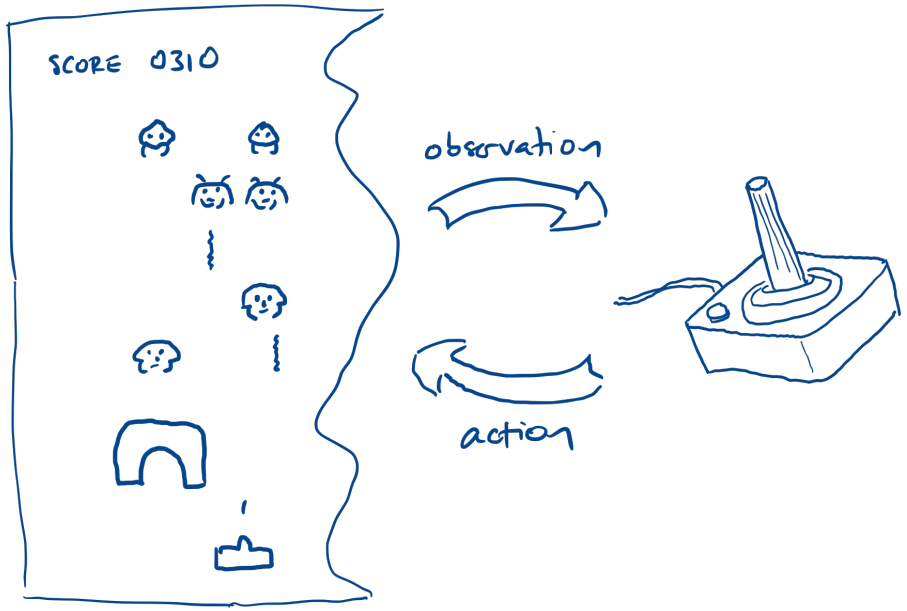
- Policy is a m[illegible]oo: represents $P(a \mid s, \pi)$
  - note: stochastic! (lets an optimizer make small changes)
- Several common ways to set up:
  - $s \mapsto [P(a_1 \mid s, \pi), P(a_2 \mid s, \pi), \ldots, P(a_k \mid s, \pi)]^\top$

Policy



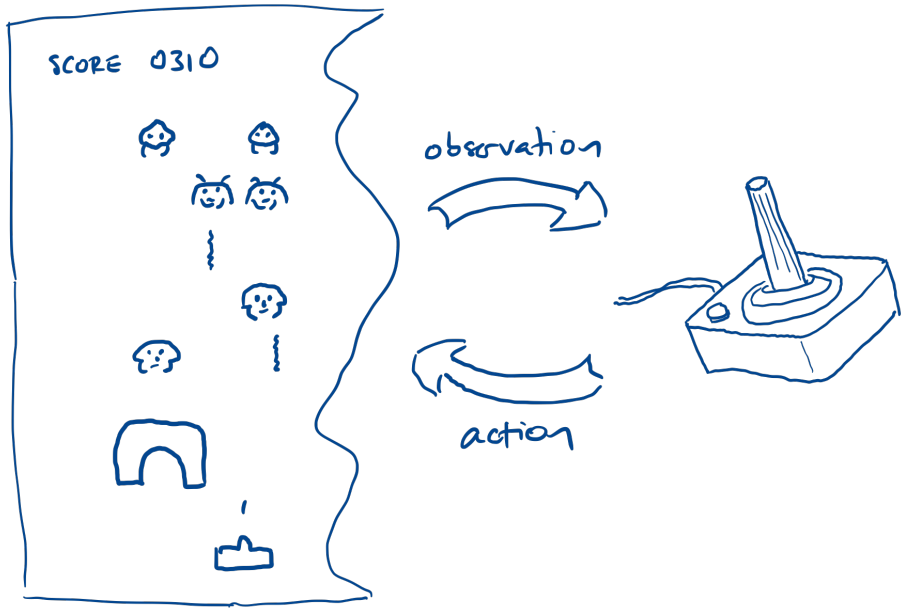
- Policy is a model too: represents $P(a \mid s, \pi)$
  - note: stochastic! (lets an optimizer make small changes)
- Several common ways to set up:
  - $s \mapsto$parameters of action distribution like mean, variance

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

BACKGROUND: MONTE CARLO ESTIMATION

Monte Carlo Estimation

Goal: Estimate the expectation of a function f of a (possibly high dimensional) random variable $\mathbf{X} \sim P(\mathbf{X})$:

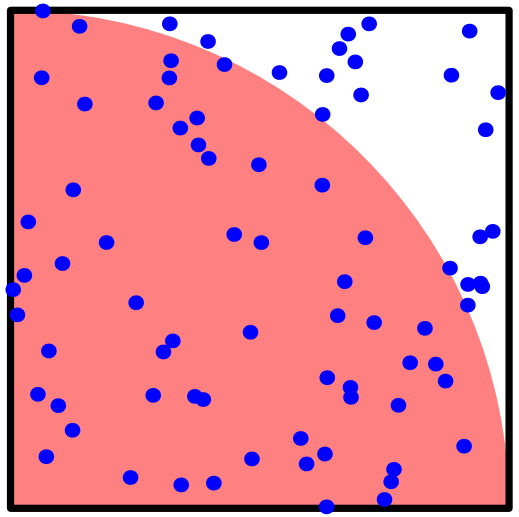
$$\phi = \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})}[f(\mathbf{x})] = \int_{\mathbf{x}} p(\mathbf{x}) f(\mathbf{x})\, d\mathbf{x}$$

Monte Carlo Estimate: Approximate the expectation by drawing S samples from p and computing the average of the function f on the samples:

$$\hat{\phi} = \frac{1}{S} \sum_{s=1}^{S} f(\mathbf{x}^{(s)}) \text{ where } \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(S)} \overset{\text{i.i.d.}}{\sim} p(\mathbf{x})$$

Monte Carlo Estimation

Example: A Dumb Approximation of π



$$P(x,y) = \begin{cases} 1 & 0<x<1 \quad \text{and} \quad 0<y<1 \\ 0 & \text{otherwise} \end{cases}$$

$$\pi = 4 \iint \mathbb{I}\left((x^2+y^2)<1\right) P(x,y)\, \mathrm{d}x\, \mathrm{d}y$$

```
$ python monte-carlo-pi.py
S=100        pi ≈ 3.080000  error=0.061593
S=1000       pi ≈ 3.228000  error=0.086407
S=10000      pi ≈ 3.134400  error=0.007193
S=100000     pi ≈ 3.151360  error=0.009767
S=1000000    pi ≈ 3.140776  error=0.000817
```

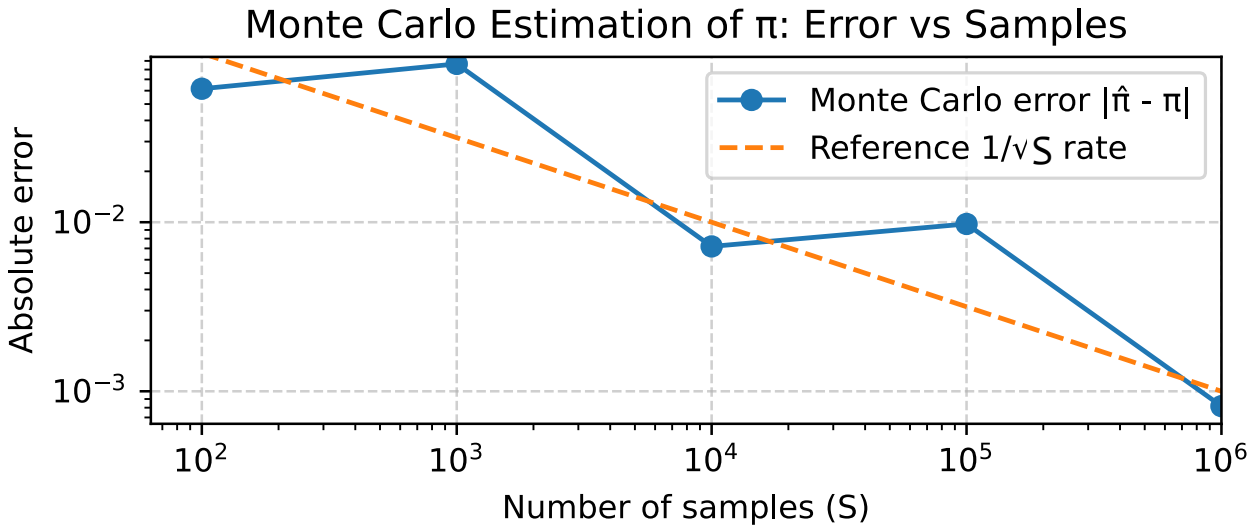


Figure / Example from Iain Murray https://homepages.inf.ed.ac.uk/imurray2/teaching/09mlss/slides.pdf

DEEP RL

Deep RL

Question: *What if our state space S is too large to represent with a table?*

Examples:

- s_t = pixels of a video game
- s_t = continuous values of a sensors in a manufacturing robot
- s_t = sensor output from a self-driving car

Answer: Use a parametric function to approximate the table entries

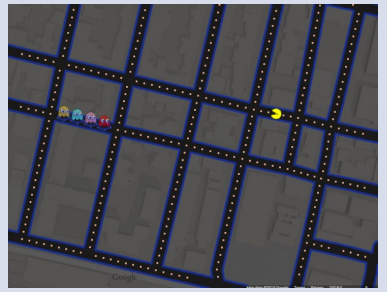
Key Idea:

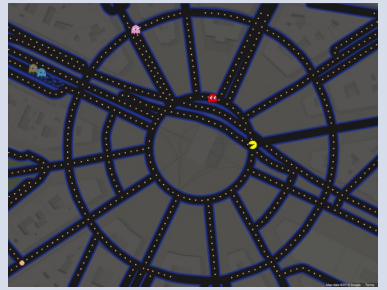
1. Use a neural network $V(s; \theta)$ to approximate $V^*(s)$
2. Learn the parameters θ via SGD with training examples $< s_t, a_t, r_t, s_{t+1} >$

Designing State Spaces

Q: Do we have to retrain our RL agent every time we change our state space?

A: Yes. But whether your state space changes from one setting to another is determined by your design of the state representation.





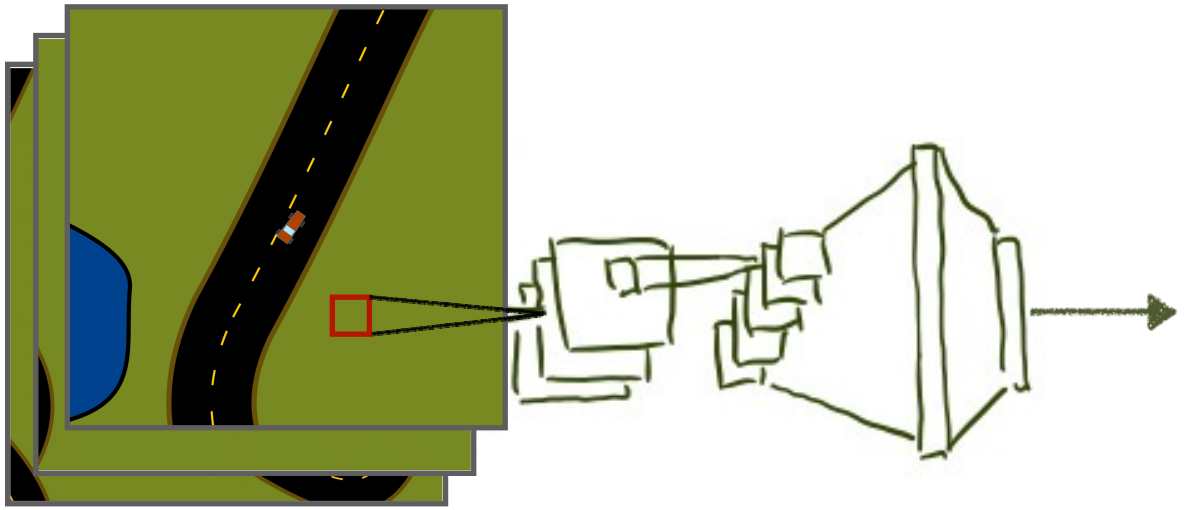
Two examples:

- State Space A: <x,y> position on map
  e.g. s_t = <74, 152>
- State Space B: window of pixel colors centered at current Pac Man location
  e.g. s_t =

0	1	0
0	0	0
1	1	1

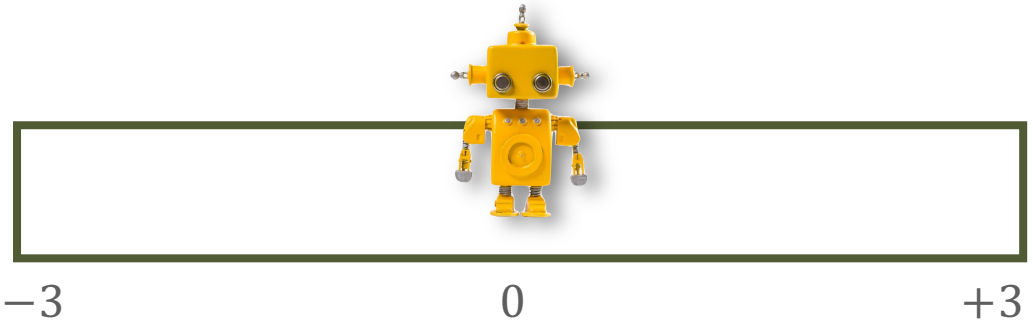
LEARNING A VALUE FUNCTION WITH FUNCTION APPROXIMATION

Learning V^{π}



- Given a fixed policy π
- Want to train a network $V_{\phi}^{\pi}(s)$ 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

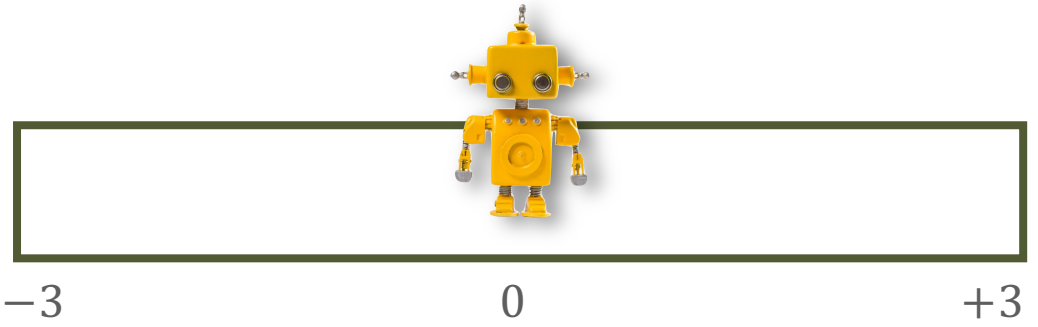
Learning V^{π}: example

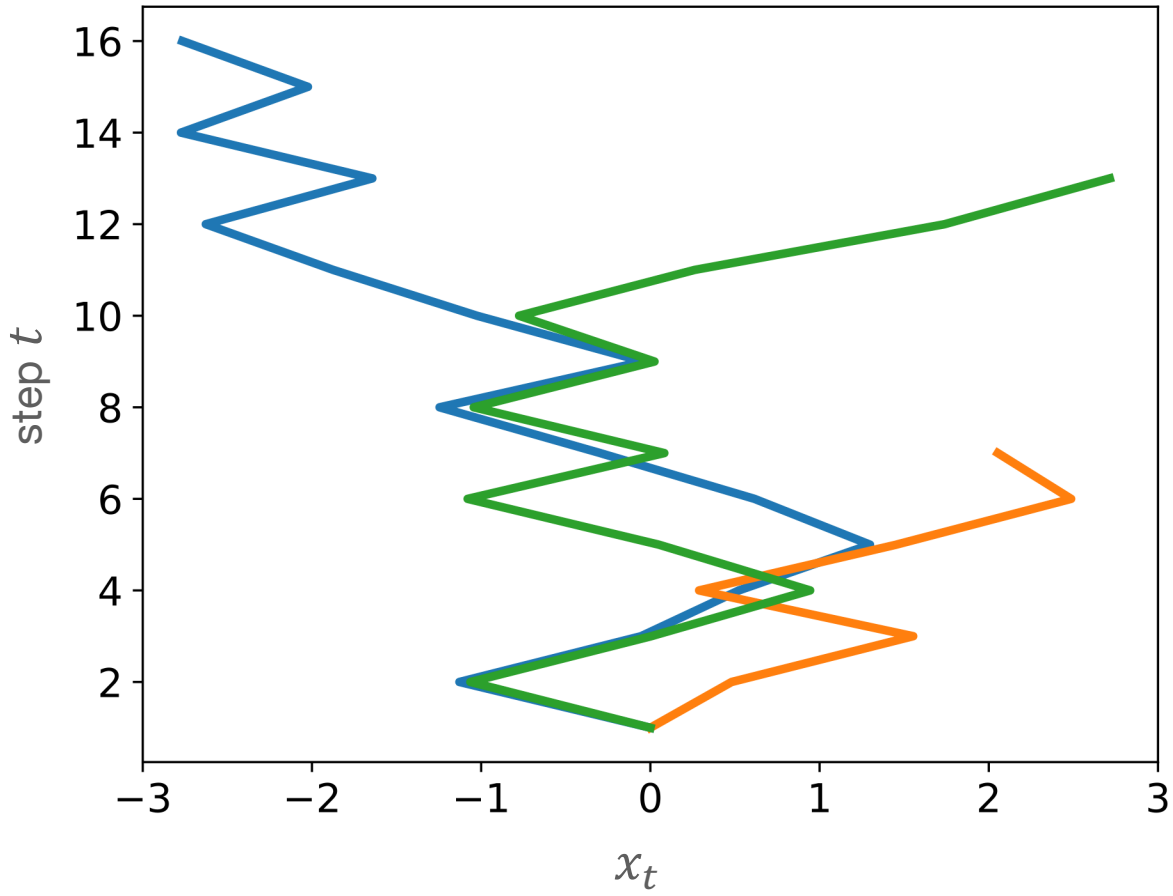


- Environment:
  - state $x \in [-3,3]$, start at $x = 0$
  - actions L: $x = x - N(1, \sigma^2)$ and R: $x = x + N(1, \sigma^2)$
  - rewards: –1 per action, terminate when $x \notin [-3,3]$
  - $\gamma = 1, \sigma = \frac{1}{4}$

Some sample trajectories

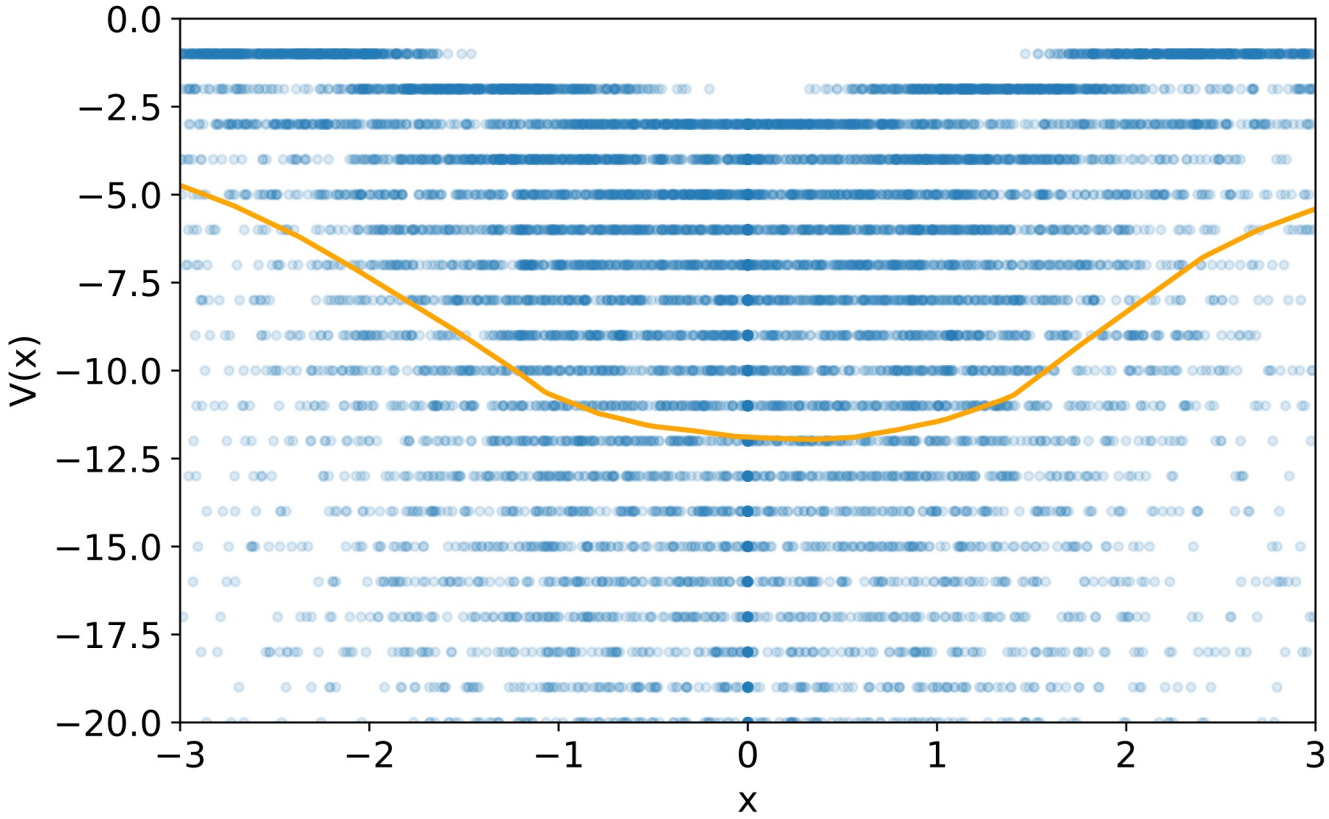
Each trajectory yields several training examples





(x, V(x))
(0, –13)
(-1, –15)
(+2, 2)
(-2, -1)
…
(-3, –20)

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!

How to Learn V^π given π ?

- **Problem:** Large state space (e.g., high-resolution images): $V(s)$ cannot be represented as a table.
- **Solution:**
  - Use a function approximator (e.g., a neural network) to represent $V(s)$ as $V_{\boldsymbol{\phi}}(s)$, where $\boldsymbol{\phi}$ are the parameters.
  - Estimate $V^\pi(s)$ via Monte Carlo: sample trajectories from π and average the returns.
  - Update $\boldsymbol{\phi}$ via gradient descent to minimize MSE between $V_{\boldsymbol{\phi}}(s)$ and the Monte Carlo estimates of $V^\pi(s)$.

Example function approximators:

- **Linear function approximation:** $V_{\boldsymbol{\phi}}(s) = \boldsymbol{\phi}^T \mathbf{x}(s)$, where $\mathbf{x}(s)$ is a feature vector representation of state s.
- **One-hidden layer neural network:** $V_{\boldsymbol{\phi}}(s) = \mathbf{w}^T \sigma(\mathbf{W}\mathbf{x}(s)+\mathbf{b})$, where $\mathbf{W}, \mathbf{b}, \mathbf{w}$ are the parameters of the neural network, $\mathbf{x}(s)$ is a feature vector representation of state s, and σ is a non-linear activation function (e.g., ReLU).

How to Learn V^{π} given π ?

Algorithm 1 On-Policy Prediction with Function Approximation

procedure MONTECARLOPREDICTIONFA(π, η, γ)
  Initialize value parameters ϕ randomly
  while not converged **do**
    Sample a trajectory $\tau = [(s_0, a_0, r_0), \ldots, (s_T, a_T, r_T)]$ using π
    $\mathbf{g} \leftarrow \sum_{t=0}^{T} \nabla_{\phi} \left(V_{\phi}(s_t) - \bar{R}_t\right)^2$
    where $\bar{R}_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$
    $\phi \leftarrow \phi - \eta \mathbf{g}$
  return V_{ϕ}

Okay, this idea of using a neural net as a value function approximator is great, but how do we **learn** a policy now?

POLICY GRADIENT ALGORITHMS

Policy Gradient Methods (generally)

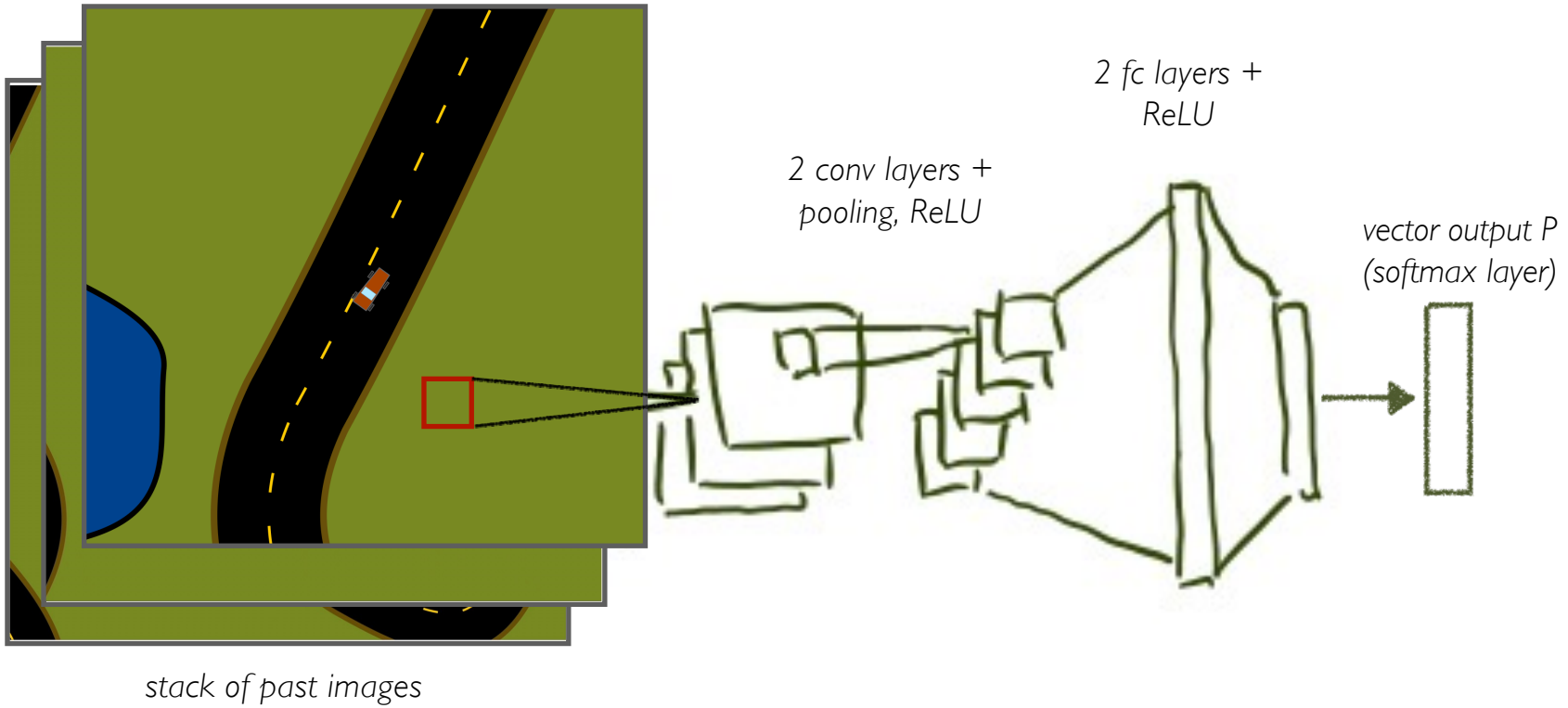
- Assume a stochastic policy $\pi_{\boldsymbol{\theta}}(a \mid s)$ parameterized by $\boldsymbol{\theta}$, that returns the probability of taking action a in state s.
- Assume we have an objective function $J(\boldsymbol{\theta})$ that we want to maximize (usually maximizing reward under the policy $\pi_{\boldsymbol{\theta}}$).
- **Definition:** A **policy gradient method** learns the parameters $\boldsymbol{\theta}$ of the policy $\pi_{\boldsymbol{\theta}}$ by performing gradient ascent:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

  where η is the learning rate.

- **Definition:** $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is called the **policy gradient**.
- (We might also use/learn a value function $V_{\boldsymbol{\phi}}(s)$ parameterized by $\boldsymbol{\phi}$, that approximates the expected discounted future reward of being in state s.)

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 Gradient Theorem

Standard Objective The goal in RL is to find a policy $\pi_{\boldsymbol{\theta}}(a|s)$ that maximizes the expected cumulative discounted reward:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}} \left[\sum_{t=0}^{T} \gamma^t r_t \right] = \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}} [R(\tau)]$$

where $\tau = [(s_0, a_0, r_0), \ldots, (s_T, a_T, r_T)]$ is a trajectory or episode, $R(\tau)$ is the total reward of the trajectory, $\gamma \in [0, 1]$ is the discount factor, and the expectation is taken over trajectories or episodes obtained by deploying the policy $\pi_{\boldsymbol{\theta}}$ on the environment.

Theorem 1

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}} [R(\tau) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau)]$$

Theorem 2

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}} \left[\sum_{t=0}^{T} \bar{R}_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \right]$$

where $\bar{R}_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$ is the discounted return from time step t.

Policy Gradient Theorem

Theorem 1 (Proof) Let $p_{\boldsymbol{\theta}}(\tau) = \prod_{t=0}^{T} \pi_{\boldsymbol{\theta}}(a_t|s_t)p(s_{t+1}|s_t, a_t)$ be the probability of a trajectory τ under the policy $\pi_{\boldsymbol{\theta}}$ and s_0 is the initial state. Then we can derive the policy gradient as follows:

$$
\begin{aligned}
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}[R(\tau)] \\
&= \nabla_{\boldsymbol{\theta}} \sum_{\tau} [R(\tau) p_{\boldsymbol{\theta}}(\tau)] && \text{expand the expectation} \\
&= \sum_{\tau} [R(\tau) \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\tau)] && \text{push in the gradient} \\
&= \sum_{\tau} \left[R(\tau) p_{\boldsymbol{\theta}}(\tau) \frac{\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\tau)}{p_{\boldsymbol{\theta}}(\tau)} \right] && \text{multiply by 1.0} \\
&= \sum_{\tau} [R(\tau) p_{\boldsymbol{\theta}}(\tau) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau))] && \nabla \log(f(x)) = (\nabla f(x))/f(x) \\
&= \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}} [R(\tau) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau)] && \text{expectation of gradient}
\end{aligned}
$$

Policy Gradient Theorem

Theorem 2 (Proof) Let $p_{\boldsymbol{\theta}}(\tau) = \prod_{t=0}^{T} \pi_{\boldsymbol{\theta}}(a_t|s_t)p(s_{t+1}|s_t, a_t)$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\tau\sim\pi_{\boldsymbol{\theta}}}[R(\tau)\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\tau)] \qquad \text{from Theorem 1}$$

$$= \mathbb{E}_{\tau\sim\pi_{\boldsymbol{\theta}}}\left[R(\tau)\nabla_{\boldsymbol{\theta}} \sum_{t=0}^{T} (\log \pi_{\boldsymbol{\theta}}(a_t|s_t) + \log p(s_{t+1}|s_t, a_t))\right] \qquad \text{expand } p_{\boldsymbol{\theta}}(\tau)$$

$$= \mathbb{E}_{\tau\sim\pi_{\boldsymbol{\theta}}}\left[R(\tau) \sum_{t=0}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)\right] \qquad \text{transition probs. do not depend on } \boldsymbol{\theta}$$

$$= \mathbb{E}_{\tau\sim\pi_{\boldsymbol{\theta}}}\left[\sum_{t=0}^{T} \bar{R}_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)\right]$$

where $\bar{R}_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$ is the discounted return from time step t.

(Last line: because the past reward part has expectation zero when multiplied by the gradient of the current action.)

Policy Gradient Theorem

Theorem 3:

We can obtain an unbiased estimate of the policy gradient by sampling a trajectory τ from the policy $\pi_{\boldsymbol{\theta}}$ and computing:

$$\mathbf{g} = \sum_{t=0}^{T} \bar{R}_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)$$

(This is a **Monte Carlo Estimate** of the gradient!)

where $\bar{R}_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$ is the discounted return from time step t.

That is, the expectation of $\mathbf{g}$ is equal to the policy gradient:

$$\mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}[\mathbf{g}] = \nabla J(\boldsymbol{\theta})$$

Policy gradient intuition

- Policy gradient: $\mathbf{g} = \sum_{t=0}^{T} \bar{R}_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)$
- Score vectors $\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)$: parameter direction that would increase (log-)probability of taking action a_t^m in state s_t^m
- Scale by $\bar{R}_t$: 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

Algorithm 1: 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:

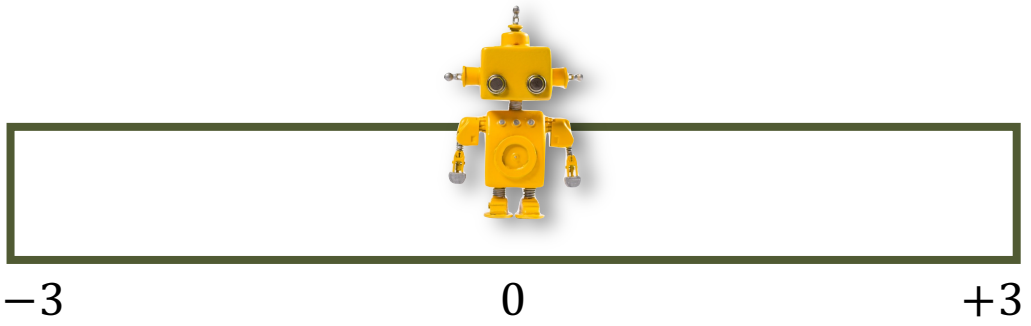
- gather some trajectories under current policy π_θ
- compute gradient estimate g by policy gradient theorem
- update θ by SGD

1: **procedure** REINFORCE(η, γ)
2: Initialize policy parameters $\boldsymbol{\theta}$ randomly
3: **while** not converged **do**
4: Sample a trajectory $\tau = [(s_0, a_0, r_0), \ldots, (s_T, a_T, r_T)]$ from $\pi_{\boldsymbol{\theta}}$
5: $\mathbf{g} \leftarrow \sum_{t=1}^{T} \bar{R}_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)$
6: where $\bar{R}_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$
7: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \mathbf{g}$
8: **return** $\pi_{\boldsymbol{\theta}}$

REINFORCE Example

Environment:

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



Policy:

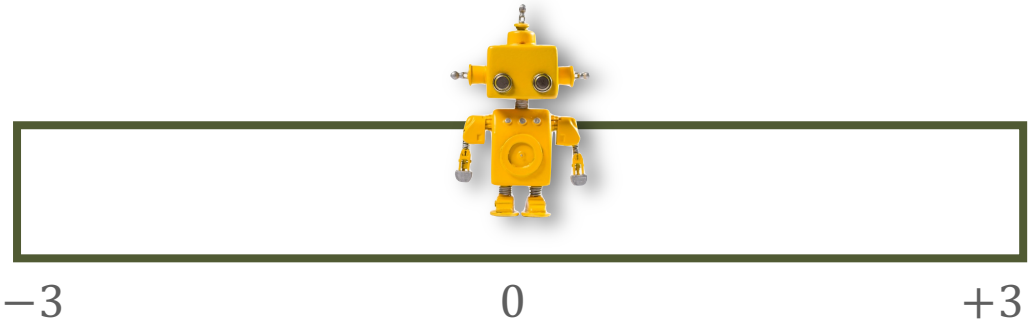
$$\pi_{w,b}(R \mid x) = \frac{1}{1 + e^{-(wx+b)}}$$

$$\pi_{w,b}(L \mid x) = \frac{1}{1 + e^{wx+b}}$$

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

Figure from Geoff Gordon

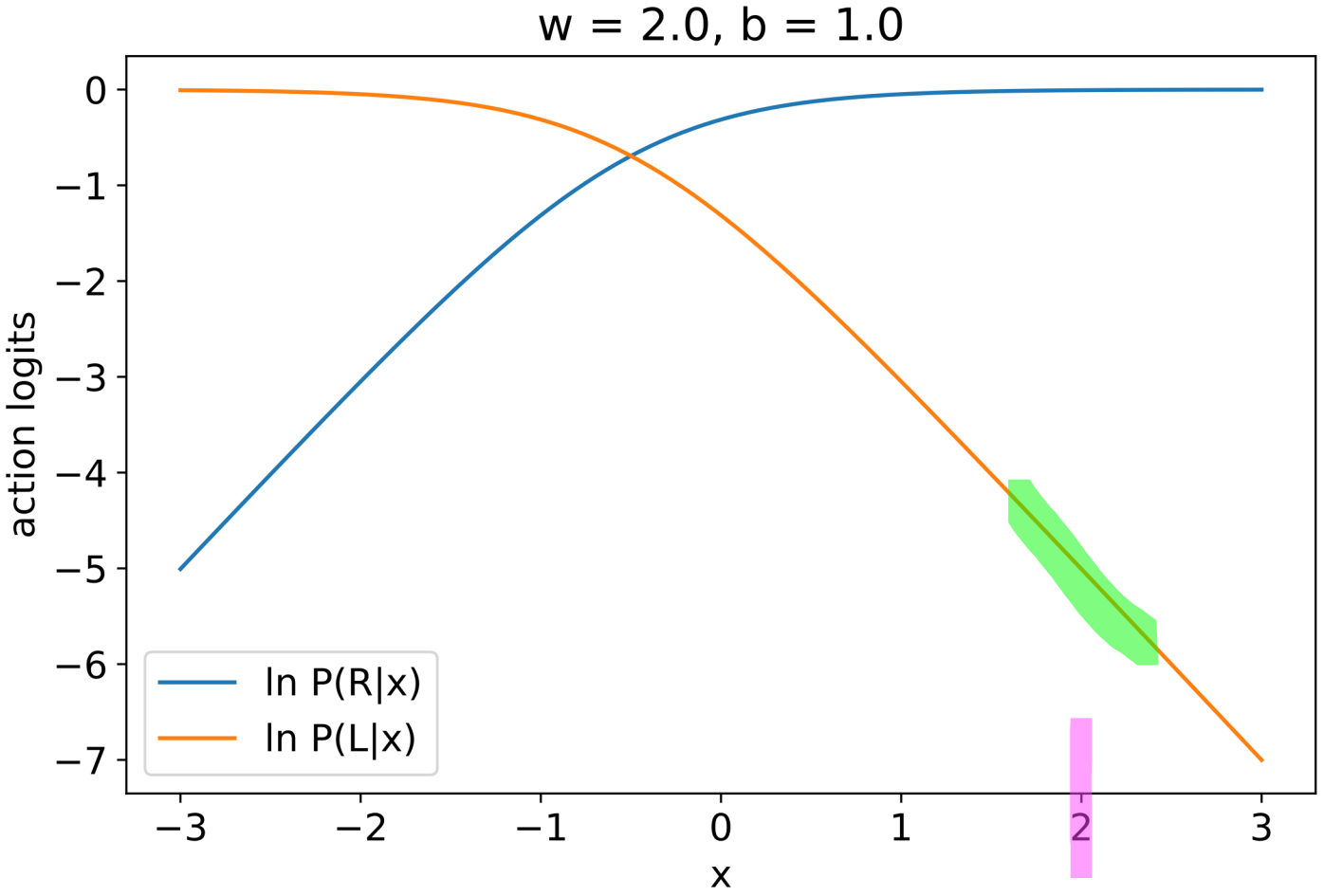
REINFORCE Example



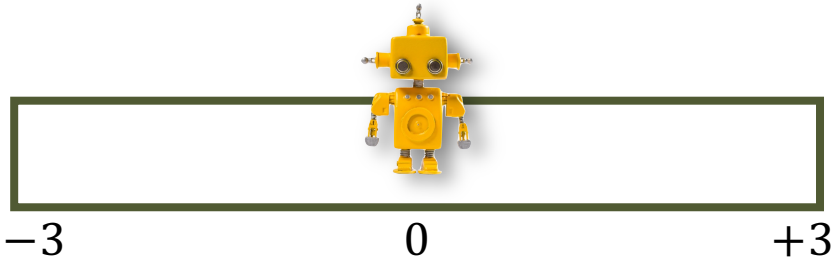
Action logits

$$\nabla_\theta \ln P(L \mid x = 2) = \begin{bmatrix} -1 \\ -2 \end{bmatrix}$$

$$\theta = \begin{bmatrix} \mathrm{w} \\ \mathrm{b} \end{bmatrix}$$



REINFORCE Example



Policy Gradient:

$$\nabla_{\boldsymbol{\theta}} \ln \pi_{w,b}(R \mid x) = -\nabla_{\boldsymbol{\theta}} \ln\left(1 + e^{-(wx+b)}\right)$$
$$= -\frac{1}{1 + e^{-(wx+b)}} e^{-(wx+b)} \left(-\nabla_{\boldsymbol{\theta}}(-wx - b)\right)$$
$$= \frac{1}{1 + e^{wx+b}} \begin{pmatrix} x \\ 1 \end{pmatrix}$$
$$= \pi_{w,b}(L \mid x) \begin{pmatrix} x \\ 1 \end{pmatrix}$$

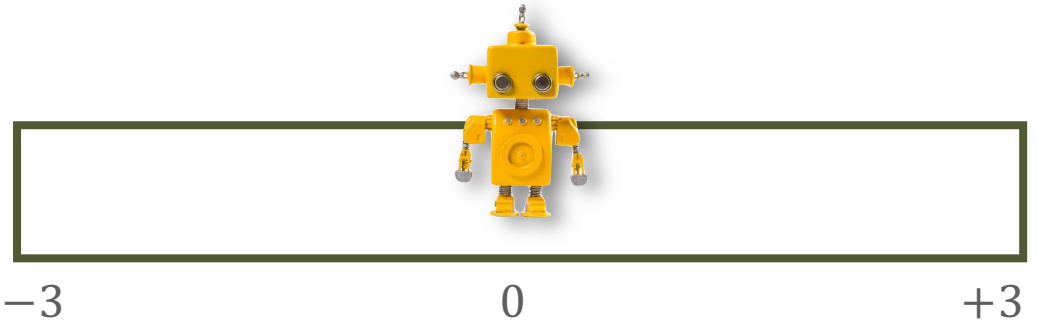
Policy:

$$\pi_{w,b}(R \mid x) = \frac{1}{1 + e^{-(wx+b)}}$$
$$\pi_{w,b}(L \mid x) = \frac{1}{1 + e^{wx+b}}$$
$$\theta = (w, b)^\top$$

$$\nabla_{\boldsymbol{\theta}} \ln \pi_{w,b}(L \mid x) = -\nabla_{\boldsymbol{\theta}} \ln\left(1 + e^{wx+b}\right)$$
$$= -\frac{1}{1 + e^{wx+b}} e^{wx+b} \left(\nabla_{\boldsymbol{\theta}}(wx + b)\right)$$
$$= -\frac{1}{1 + e^{-(wx+b)}} \begin{pmatrix} x \\ 1 \end{pmatrix}$$
$$= \pi_{w,b}(R \mid x) \begin{pmatrix} -x \\ -1 \end{pmatrix}.$$

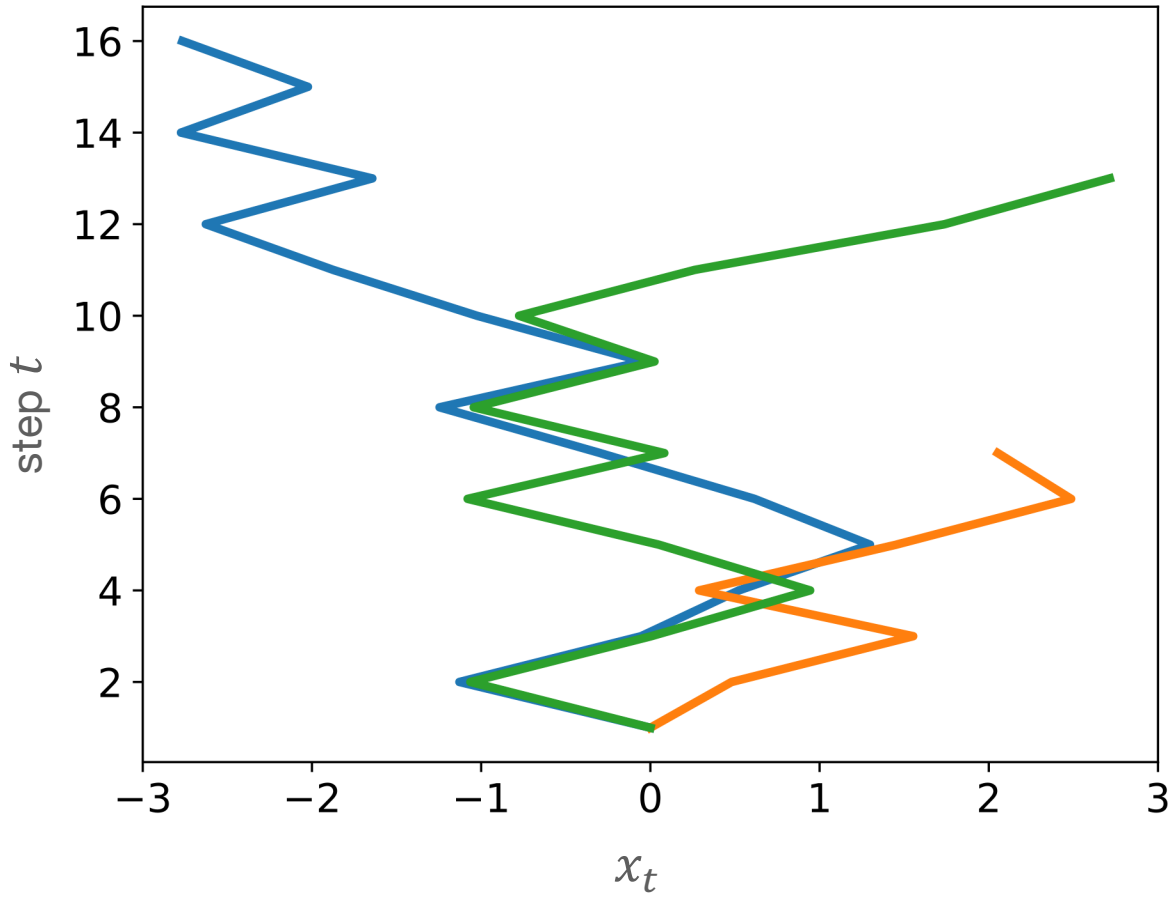
Figure from Geoff Gordon

REINFORCE Example



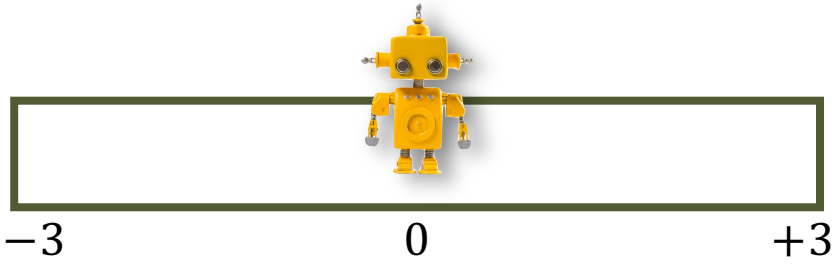
(x, a)	Q
(0, R)	–7
(.3, R)	–6
(1.6, L)	–5
(.2, R)	–4
(1.3, R)	–3
(2.4, L)	–2
(2.1, R)	–1

Calculate gradient estimate



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

REINFORCE Example



Gradient update for a trajectory:

We have one sampled trajectory of (x, a, r) tuples:

$$\tau = [(0, R, -1),\ (.3, R, -1),\ (1.6, L, -1),\ (.2, R, -1),\ (1.3, R, -1)]$$

the returns-to-go $\bar{R}_t$ are:

$$-5,\ -4,\ -3,\ -2,\ -1.$$

If $w = b = 0$, then $\pi_{w,b}(R \mid x) = \pi_{w,b}(L \mid x) = 0.5$, so

$$\begin{aligned}
\mathbf{g} &= \sum_{t=0}^{T} \bar{R}_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid x_t), \\
&= -5(0.5)\begin{pmatrix}0\\1\end{pmatrix} - 4(0.5)\begin{pmatrix}0.3\\1\end{pmatrix} - 3(0.5)\begin{pmatrix}-1.6\\-1\end{pmatrix} - 2(0.5)\begin{pmatrix}0.2\\1\end{pmatrix} - 1(0.5)\begin{pmatrix}1.3\\1\end{pmatrix} \\
&= \begin{pmatrix}0.95\\-4.5\end{pmatrix}.
\end{aligned}$$

We update the policy parameters $\boldsymbol{\theta}$ using the policy gradient $\mathbf{g}$:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \mathbf{g} = \boldsymbol{\theta} + \alpha \begin{pmatrix}0.95\\-4.5\end{pmatrix}.$$

$$\nabla_{\boldsymbol{\theta}} \log \pi_{w,b}(R \mid x) = \pi_{w,b}(L \mid x) \begin{pmatrix}x\\1\end{pmatrix},$$

$$\nabla_{\boldsymbol{\theta}} \log \pi_{w,b}(L \mid x) = \pi_{w,b}(R \mid x) \begin{pmatrix}-x\\-1\end{pmatrix}.$$

Figure from Geoff Gordon

Poll Question 1

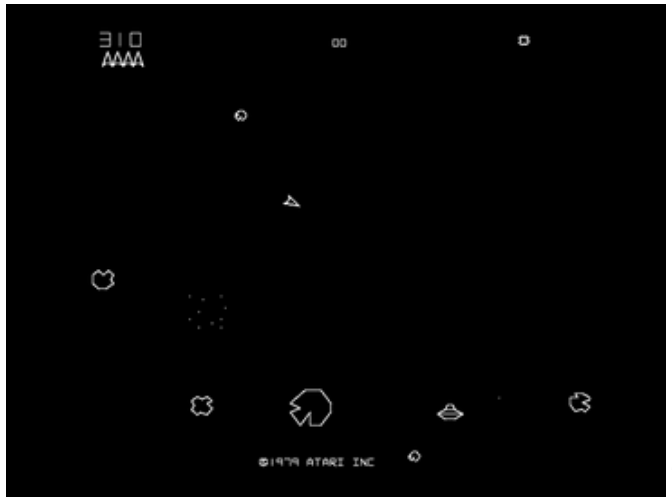


image credit: (c) Atari, via MAME and Wikipedia

We are designing an agent to play *Asteroids*. We build a policy network that takes the screen image and current controller state (buttons pressed) as input, and predicts what to do next (which buttons to press). We start at a random level, and give positive rewards for scoring points and for completing the level. We train the policy to maximize sum of rewards from the start of the level until the end or death.
Which common RL mistake are we making?

A. We are improperly handling the sum over rewards: we have to use a discount factor.
B. We are training a deterministic policy even though the environment is stochastic.
C. We are treating the immediate observation as a state.
D. We haven't yet made a blog post describing our network and training technique.

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 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

Algorithm 1: 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

procedure REINFORCE(η, γ)
    Initialize policy parameters $\boldsymbol{\theta}$ randomly
    while not converged **do**
        Sample a trajectory $\tau = [(s_0, a_0, r_0), \ldots, (s_T, a_T, r_T)]$ from $\pi_{\boldsymbol{\theta}}$
        $\mathbf{g} \leftarrow \sum_{t=1}^{T} \bar{R}_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)$
        where $\bar{R}_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \mathbf{g}$
    return $\pi_{\boldsymbol{\theta}}$

Algorithm 2: REINFORCE with Baseline

- The policy gradient estimator $\mathbf{g}$ is unbiased, but it can have high variance.
- To reduce variance, we introduce a baseline function b(s) by replacing R_t with (R_t – $b(s_t)$)
- b(s) can be any function of the state s, and $\mathbf{g}$ will still be an unbiased estimator of $\nabla J(\boldsymbol{\theta})$
- The value function $V^{\pi}(s)$ is a common choice for b(s)

procedure REINFORCEWITHBASELINE($\eta_\theta, \eta_\phi, \gamma$)

  Initialize policy parameters $\boldsymbol{\theta}$ and baseline parameters ϕ randomly

  while not converged **do**

    Sample a trajectory $\tau = [(s_0, a_0, r_0), \ldots, (s_T, a_T, r_T)]$ from $\pi_{\boldsymbol{\theta}}$

    $\mathbf{g}_\theta \leftarrow \sum_{t=1}^{T} (\bar{R}_t - V_{\boldsymbol{\phi}}(s_t)) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)$

    where $\bar{R}_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$

    $\mathbf{g}_\phi \leftarrow \sum_{t=1}^{T} \nabla_{\boldsymbol{\phi}} (V_{\boldsymbol{\phi}}(s_t) - \bar{R}_t)^2$

    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta_\theta \mathbf{g}_\theta$

    $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} - \eta_\phi \mathbf{g}_\phi$

  return $\pi_{\boldsymbol{\theta}}, V_{\boldsymbol{\phi}}$

Actor-critic

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

- Reduce variance: replace $\overline{R_t}$ by conditional expectation
  - $\mathbb{E}(\overline{R_t} \mid s_t, a_t) = Q^{\pi}(s_t, a_t)$
  - $g_{\text{exactQ}} = \sum_{t=1}^{T} Q^{\pi}(s_t, a_t) u_t$ — 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)

Algorithm 3: Actor-Critic

In practice we treat $Q_{\phi}(s_t, a_t) = r_t + \gamma V_{\phi}(s_{t+1})$

Then we learn $V_{\phi}(s)$

Here are we learning $V_{\phi}(s)$ in just the same way we learned it earlier

procedure ACTORCRITIC($\eta_\theta, \eta_\phi, \gamma$)
  Initialize policy parameters $\boldsymbol{\theta}$ and value parameters ϕ randomly
  while not converged **do**
    Collect a trajectory $\tau = [(s_0, a_0, r_0), \ldots, (s_T, a_T, r_T)]$ from $\pi_{\boldsymbol{\theta}}$
    $\mathbf{g}_\theta \leftarrow \sum_{t=1}^{T} \hat{Q}_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t)$
    where $\hat{Q}_t = r_t + \gamma V_{\boldsymbol{\phi}}(s_{t+1})$
    $\mathbf{g}_\phi \leftarrow \sum_{t=1}^{T} \nabla_{\boldsymbol{\phi}} \left(V_{\boldsymbol{\phi}}(s_t) - \hat{Q}_t \right)^2$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta_\theta \mathbf{g}_\theta$
    $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} - \eta_\phi \mathbf{g}_\phi$
  return $\pi_{\boldsymbol{\theta}}, V_{\boldsymbol{\phi}}$

- the policy is the **actor**
- the value function is the **critic**

Algorithm 4: Advantage Actor-Critic (A2C)

- In AAC we take the **baseline** and the **critic** together
- 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)

procedure A2C($\eta_\theta, \eta_\phi, \gamma$)
  Initialize policy parameters $\boldsymbol{\theta}$ and value parameters $\boldsymbol{\phi}$ randomly
  while not converged **do**
    Collect a trajectory $\tau = [(s_0, a_0, r_0), \ldots, (s_T, a_T, r_T)]$ from $\pi_{\boldsymbol{\theta}}$
    $\mathbf{g}_\theta \leftarrow \sum_{t=1}^{T} \hat{A}_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t)$
    where $\hat{A}_t = r_t + \gamma V_{\boldsymbol{\phi}}(s_{t+1}) - V_{\boldsymbol{\phi}}(s_t)$
    $\mathbf{g}_\phi \leftarrow \sum_{t=1}^{T} \nabla_{\boldsymbol{\phi}} \left(r_t + \gamma V_{\boldsymbol{\phi}}(s_{t+1}) - V_{\boldsymbol{\phi}}(s_t)\right)^2$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta_\theta \mathbf{g}_\theta$
    $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} - \eta_\phi \mathbf{g}_\phi$
  return $\pi_{\boldsymbol{\theta}}, V_{\boldsymbol{\phi}}$

Algorithm 5: A2C with n-Step Returns

- The advantage actor-critic algorithm uses a **one-step return** to estimate the advantage function, which can lead to high bias in the policy gradient estimate.
- To reduce bias, we can use an **n-step return** to estimate the advantage function
- Closely connected to temporal difference learning, specifically TD(lambda)

procedure A2C-NSTEP($\eta_\theta, \eta_\phi, \gamma, n$)
  Initialize policy parameters $\boldsymbol{\theta}$ and value parameters $\boldsymbol{\phi}$ randomly
  while not converged **do**
    Collect a trajectory $\tau = [(s_0, a_0, r_0), \ldots, (s_T, a_T, r_T)]$ from $\pi_{\boldsymbol{\theta}}$
    $\mathbf{g}_\theta \leftarrow \sum_{t=1}^{T} \hat{A}_t^{(n)} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t)$
    where $\hat{A}_t^{(n)} = \hat{R}_t^{(n)} - V_{\boldsymbol{\phi}}(s_t)$
    where $\hat{R}_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V_{\boldsymbol{\phi}}(s_{t+n})$
    $\mathbf{g}_\phi \leftarrow \sum_{t=1}^{T} \nabla_{\boldsymbol{\phi}} \left(\hat{R}_t^{(n)} - V_{\boldsymbol{\phi}}(s_t) \right)^2$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta_\theta \mathbf{g}_\theta$
    $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} - \eta_\phi \mathbf{g}_\phi$
  return $\pi_{\boldsymbol{\theta}}, V_{\boldsymbol{\phi}}$

Comparison of Policy Gradient Methods

1) All of the algorithms compute a gradient of the form:

$$\mathbf{g}_{\boldsymbol{\theta}} = \sum_{t=1}^{T} \tilde{G}_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)$$

where $\tilde{G}_t$ is an estimate of the expected discounted future reward from time step t

2) Some use a V_ϕ, and some do not

Algorithm	$\tilde{G}_t$	Estimate $V_{\boldsymbol{\phi}}$?
REINFORCE	$\bar{R}_t$	x
REINFORCE with Baseline	$(\bar{R}_t - V_{\boldsymbol{\phi}}(s_t))$	✓
Actor-Critic	$\hat{Q}_t = r_t + \gamma V_{\boldsymbol{\phi}}(s_{t+1})$	✓
Advantage Actor-Critic	$\hat{A}_t = r_t + \gamma V_{\boldsymbol{\phi}}(s_{t+1}) - V_{\boldsymbol{\phi}}(s_t)$	✓

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 $\mathrm{argmax}_a(r(s,a) + V_\phi^\pi(\delta(s,a)))$, 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)

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