CSE 592: Foundations of AI — University of Michigan

Lecture 17: RL part II



Gabriel Poesia
Assistant Professor, CSE
University of Michigan, Ann Arbor
poesia@umich.edu
https://gpoesia.com

Plan

1. Function approximation
2. Deep Q-networks
3. Policy gradients
4. Actor–critic

Where we left off

> 🧠 **Recall**
> Last time: learn to act from experience, without a model.

Q-learning: after each transition, update Q toward a target y_t:

$$y_t = R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[\, y_t - Q(S_t, A_t)\,]$$

Off-policy: we act with exploration (ϵ-greedy), but learn about the greedy policy.
SARSA: similar, but on-policy: target was $y_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$.
In this case, behavior policy and target policy are the same.
Both were for finite states and actions
E.g.: Blackjack, 200 states x 2 actions = 400 entries in a $Q(s, a)$ table).

What if the state space is huge? Infinite? Actions are continuous?

Function approximation

Too many states

Most interesting problems don't fit in a table.

Chess / Go

Moves: max a few hundred.

Positions: somewhere between 10^{50} and 10^{80}.

Atari

Actions: max 18 buttons.

States: every possible screen.

Robot control

Actions: continuous actuators.

Positions, velocities: continuous.

Problem: can't store a value for every state-action pair.

Also can't even **visit** every state, let alone many times.

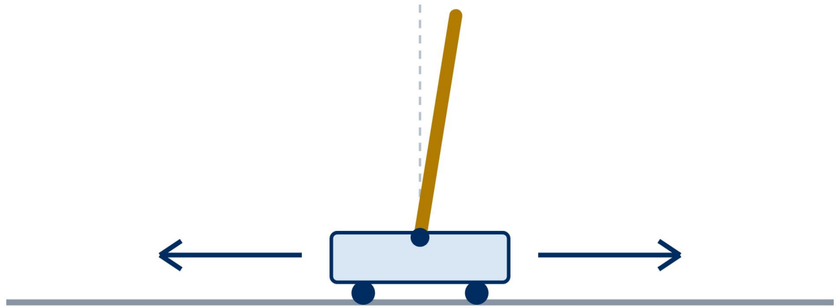
What can we do?

> 💡 **Insight**
> "Similar" states often have similar values.

"Similar" how? In representation (features)!

CartPole

Running classical example: keep a pole balanced on a moving cart.



Push the cart to keep the pole upright.

Two actions: push left, or push right.

Reward: $+1$ per step

Episode ends if the pole falls, cart leaves the track, or after 500 steps.

Gymnasium: CartPole-v1. Two actions, four state variables, up to 500 steps.

CartPole states

State: four numbers.

$$s = (x, \dot{x}, \theta, \dot{\theta})$$

Variable	Meaning
x	Cart position
$\dot{x}$	Cart velocity
θ	Pole angle
$\dot{\theta}$	Pole angular velocity

Approximating Q

Can we do Q-learning without a $Q(s, a)$ table?

Key idea: replace the table by a function with parameters w.

$$Q_w(s, a) \approx Q^*(s, a)$$

State + action	→	**Function**	→	**Predicted return**
(s, a)		Parameters w		$Q_w(s, a)$

Parameters are shared across all states, so we can generalize.

Act just like before: still ϵ-greedy wrt Q_w.

Linear action values

Simplest choice: features $\phi(s, a) \in \mathbb{R}^d$, and a linear model.

$$Q_w(s, a) = w^\top \phi(s, a) = \sum_{i=1}^{d} w_i \phi_i(s, a)$$

CartPole: position, velocity, angle, angular velocity, and a bias.

Can also add nonlinear features: θ^2, $\theta\dot{\theta}$, ...

> 🧠 **Recall**
> "Linear" on the **weights** and features, not in the state.

Sutton & Barto, Reinforcement Learning: An Introduction, 2nd ed., chs. 9–11.

Features for actions

Q: Where does the action come in?

Simple option: state features $f(s)$, separate weights per action.

$$f(s) = (1, x, \dot{x}, \theta, \dot{\theta})$$

Push left

$Q(s, \text{left}) = w_L^\top f(s)$

Push right

$Q(s, \text{right}) = w_R^\top f(s)$

Same as one weight vector $w = (w_L, w_R)$, with:

$$\phi(s, \text{left}) = (f(s), 0), \qquad \phi(s, \text{right}) = (0, f(s))$$

Learning from a target

> 🧠 **Recall**
> In linear regression, the most common is to fit predictions $f(x)$ to targets y with the squared loss.

Can do the same here at a particular state-action pair:

$$\ell(w) = \tfrac{1}{2}\left[y - Q_w(s,a)\right]^2$$

For a fixed target y, a gradient descent step is:

$$w \leftarrow w + \alpha\left[y - Q_w(s,a)\right]\nabla_w Q_w(s,a)$$

Linear model: $\nabla_w Q_w(s,a) = \phi(s,a)$.

Q: In RL, where do the targets come from?

TD targets

A: Same as in Q-learning. Bootstrap!

$$y_t = R_{t+1} + \gamma \max_{a'} Q_w(S_{t+1}, a')$$

Observed reward + our current estimate of the best future.

TD error: how far off was our prediction?

$$\delta_t = y_t - Q_w(S_t, A_t)$$

Note: y_t itself depends on w. We'll ignore that, and treat it as a fixed label.

Linear Q-learning

Putting it together: TD target + linear regression step.

$$w \leftarrow w + \alpha \delta_t \, \phi(S_t, A_t)$$

```
repeat:
   a = epsilon_greedy(Q_w(s, ·))
   s_next, r, done = env.step(a)
   y = r if done else r + γ max_a' Q_w(s_next, a')
   δ = y − Q_w(s, a)
   w = w + α δ φ(s, a)  # hold y fixed
   s = reset() if done else s_next
```

Called a "**semi-gradient**" method: we don't differentiate through y_t.

Sutton & Barto, Reinforcement Learning: An Introduction, 2nd ed., chs. 9–11.

Tables as linear models

Q: What if each state-action pair gets its own one-hot feature?

$$\phi_i(s, a) = \mathbf{1}\{i = \text{index}(s, a)\}$$

Then $Q_w(s, a)$ just picks one weight: its table entry.

And the update only changes that weight:

$$w_i \leftarrow w_i + \alpha\delta_t$$

Exactly tabular Q-learning!

Stability

Caveat: we lost the tabular convergence guarantees.

Features may not be able to express Q^* (approximation error again!).

Improving one prediction can make others worse.

Q-learning with function approximation combines three things:

Approximation	**Bootstrapping**	**Off-policy**
Shared parameters	Estimated targets	Greedy target, exploring behavior

These are called the **deadly triad** of RL.

Even for linear models this can make learning very unstable.

Sutton & Barto, Reinforcement Learning: An Introduction, 2nd ed., chs. 9–11.

Deep Q-networks

Learning the features

We can learn the representation with neural networks, like before.

Here: use a neural net for Q_θ.



In case of finite actions: one logit per action in output layer.

Neural Q-learning

Same recipe as linear Q-learning. For a transition (s, a, r, s'):

$$y = r + \gamma \max_{a'} Q_\theta(s', a')$$

Gradient step on the squared loss:

$$\ell(\theta) = \tfrac{1}{2}\left[y - Q_\theta(s, a)\right]^2$$

Backpropagate through the output for the **action we took**.

As before: treat y as fixed. If the episode ended, $y = r$.

A moving target

Q: When we update θ, what happens to the targets?

Update $Q_\theta(s, a)$ → **$Q_\theta(s', a')$ changes too** → **Target y changes**

A: They move. We're chasing our own predictions.

Problem 2: consecutive transitions are very correlated.

Problem 3: as the policy changes, so does the data we see.

Very different from supervised learning on a fixed dataset!

In practice, we need some additional tricks to work around these issues.

Mnih et al. Human-level control through deep reinforcement learning. Nature, 2015.

Experience replay

Idea 1 (experience replay): keep a buffer of past transitions.

$$D = \{(s, a, r, s', \text{done}), \ldots\}$$



Each transition gets reused many times.

Random minibatches mix experience from different times: less correlated.

Q: Old transitions came from older policies. Is that OK?

A: Yes for Q-learning: it's already off-policy!

Mnih et al. Human-level control through deep reinforcement learning. Nature, 2015.

Target network

Idea 2 (target network): keep two copies of the Q-network.

Online network Q_θ

Makes predictions. Trained every step.

Target network Q_{θ^-}

Only used to compute targets. Frozen.

$$y = r + \gamma \max_{a'} Q_{\theta^-}(s', a')$$

Every C steps, copy $\theta^- \leftarrow \theta$.

Now targets stay put for a while: more like regression.

Mnih et al. Human-level control through deep reinforcement learning. Nature, 2015.

DQN

Deep Q-Network (DQN): Q-learning + neural network + replay buffer + target network.

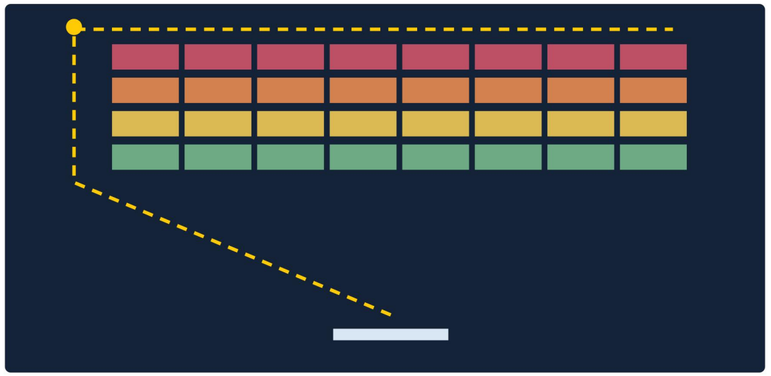
Breakthrough paper in Deep RL in 2015 (from DeepMind):

Same algorithm, architecture, hyperparameters, etc, learned to play 49 Atari games from pixels.

```
Initialize Q_θ, target Q_θ⁻ with θ⁻ = θ, replay buffer D
repeat:
   Choose a using epsilon-greedy Q_θ(s, ·)
   Observe (s, a, r, s_next, done); store it in D
   Sample a minibatch B of transitions from D
   for each transition in B:
      y = r if done else r + γ max_a' Q_θ⁻(s_next, a')
   Take a gradient step on mean_B (y − Q_θ(s, a))²
   Every C steps: θ⁻ = θ
   Continue from s_next, or reset if done
```

Mnih et al. Human-level control through deep reinforcement learning. Nature, 2015.

Learning to play Breakout



Video: DQN learning Breakout (Google DeepMind)

Another one: DQN Space Invaders

Google DeepMind. DQN Breakout (official video).

Policy gradients

Learning a policy

So far: learn Q, then act (greedily) using Q.

But what we really want is a good policy. Why not learn it directly?

$$\pi_\theta(a \mid s)$$

Model outputs a probability distribution over actions.

Small change in $\theta \rightarrow$ small change in behavior. (Not true with $\arg\max$!)

Also works for continuous actions: no $\max_a$ needed.

Policy networks

Same kind of neural net, but we interpret outputs differently:



🧠 **Recall**

Softmax turns unnormalized scores (logits) into probabilities.

$$\pi_\theta(a \mid s) = \frac{\exp z_\theta(s, a)}{\sum_{a'} \exp z_\theta(s, a')}$$

To act, sample $A_t \sim \pi_\theta(\cdot \mid S_t)$. Sampling gives us exploration.

Objective

Goal: maximize expected total reward. (To simplify assume $\gamma = 1$, finite horizon)

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T-1} R_{t+1} \right]$$

τ: a trajectory, i.e. all states, actions, and rewards of one episode.

Plan: gradient ascent on $J(\theta)$

Gradient of an expectation

> 🧠 **Recall**
> SGD estimates average gradient over the data by sampling.

$$\nabla_\theta \mathbb{E}_{x\sim D}[\ell_\theta(x)] = \mathbb{E}_{x\sim D}[\nabla_\theta \ell_\theta(x)]$$

Sample a minibatch, differentiate the loss on each example. Average.

Works because the data distribution D doesn't depend on θ.

Our objective is also an expectation:

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

But now θ is in the **distribution**: it changes which episodes we see.

Q: Can we still sample episodes and differentiate their rewards?

A one-step example

Simplest case: one state, two actions, then the episode ends. (A bandit!)

Say right always gives reward 1, and left gives 0:

$$J(\theta) = \pi_\theta(\text{right}) \cdot 1 + \pi_\theta(\text{left}) \cdot 0 = \pi_\theta(\text{right})$$

Q: We sample an action and get a reward. What's ∇_θ of that reward?

A: Zero! The reward is just a number, 1 or 0. No θ in it.

But $\nabla_\theta J(\theta) = \nabla_\theta \pi_\theta(\text{right}) \cdot 1 \neq 0$.

> 💡 **Insight**
> θ only changes how **likely** each outcome is. We need to differentiate the probabilities.

The log-derivative trick

In general, sum over all possible trajectories:

$$J(\theta) = \sum_{\tau} p_\theta(\tau)\, R(\tau) \qquad \Rightarrow \qquad \nabla_\theta J(\theta) = \sum_{\tau} \nabla_\theta p_\theta(\tau)\, R(\tau)$$

Want to turn the sum into an expectation so we can estimate by sampling.

Trick: chain rule on log.

$$\nabla_\theta \log p_\theta(\tau) = \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} \quad \Rightarrow \quad \nabla_\theta p_\theta(\tau) = p_\theta(\tau) \nabla_\theta \log p_\theta(\tau)$$

Plug it in:

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

An expectation! Play episodes, average $R(\tau) \nabla_\theta \log p_\theta(\tau)$.

Sutton & Barto, 2nd ed., ch. 13: policy gradients, REINFORCE, actor–critic.

Trajectory probabilities

Q: How do we compute $\nabla_\theta \log p_\theta(\tau)$? $p_\theta(\tau)$ involves the environment, which we don't know!

Probability of one trajectory:

$$p_\theta(\tau) = c(\tau) \prod_{t=0}^{T-1} \pi_\theta(A_t \mid S_t)$$

$c(\tau)$: probabilities of the initial state and of the environment's transitions.

Unknown, but doesn't depend on θ!

Take the log, then the gradient:

$$\nabla_\theta \log p_\theta(\tau) = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_t \mid S_t)$$

$c(\tau)$ goes away: only need gradients of the policy network.

Policy gradient

Plug in the gradient of $\log p_\theta(\tau)$:

$$\nabla_\theta J(\theta) = \mathbb{E}\left[R(\tau)\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(A_t \mid S_t)\right]$$

This lets us do stochastic gradient ascent on the policy parameters.

Intuition: make the actions we took more likely, in proportion to the episode's reward.

Need to average over many episodes to get a good estimate.

Credit assignment

Say a (very) short CartPole episode gives:

t	Action	Reward	Return from here
0	Right	1	$G_0 = 3$
1	Left	1	$G_1 = 2$
2	Right	1, then pole falls	$G_2 = 1$

$R(\tau) = 3$: all three actions get weight 3.

Q: Should the last action get credit for the first reward?

A: No: actions only affect rewards that come after them.

Not wrong on average, but this adds noise

Reward-to-go

Idea: weight actions only by the rewards after it.

$$G_t = \sum_{k=t}^{T-1} R_{k+1}$$

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{t=0}^{T-1} G_t \nabla_\theta \log \pi_\theta(A_t \mid S_t)\right]$$

Q: Does this change the gradient?

A: Not in expectation! Past rewards are already fixed when we sample A_t, and:

$$\mathbb{E}_{A_t \sim \pi_\theta}[\nabla_\theta \log \pi_\theta(A_t \mid S_t) \mid S_t] = 0$$

Same expected gradient, less noise.

Sutton & Barto, 2nd ed., ch. 13: policy gradients, REINFORCE, actor–critic.

REINFORCE

Play an episode, compute the returns, then update:

$$\theta \leftarrow \theta + \alpha \sum_{t=0}^{T-1} G_t \nabla_\theta \log \pi_\theta(A_t \mid S_t)$$

This is **REINFORCE** (Williams, 1992).

Monte Carlo: uses complete returns, like last lecture's MC.

On-policy: episodes must come from the current π_θ.

No Q-function, $\max$, etc, optimize the policy directly.

Sutton & Barto, 2nd ed., ch. 13: policy gradients, REINFORCE, actor–critic.

REINFORCE: algorithm

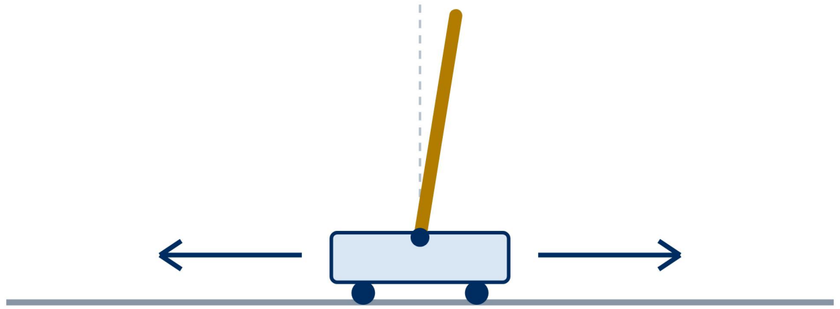
```
Initialize policy parameters θ
repeat:
    Generate an episode using a ~ π_θ(· | s)
    Compute G_t for every step, working backward
    g = Σ_t G_t ∇_θ log π_θ(A_t | S_t)
    θ = θ + α g
```

In practice: average the gradient over a batch of episodes.

Keep θ fixed while collecting the batch.

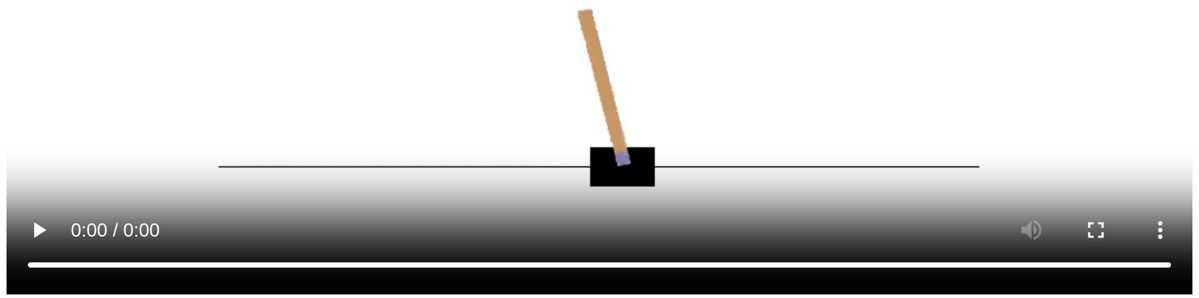
Live coding: CartPole

Let's train a CartPole neural network policy with REINFORCE!

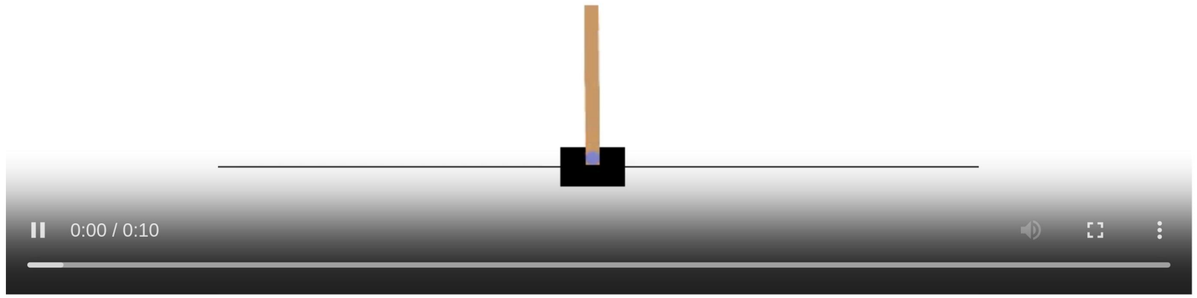


Before and after

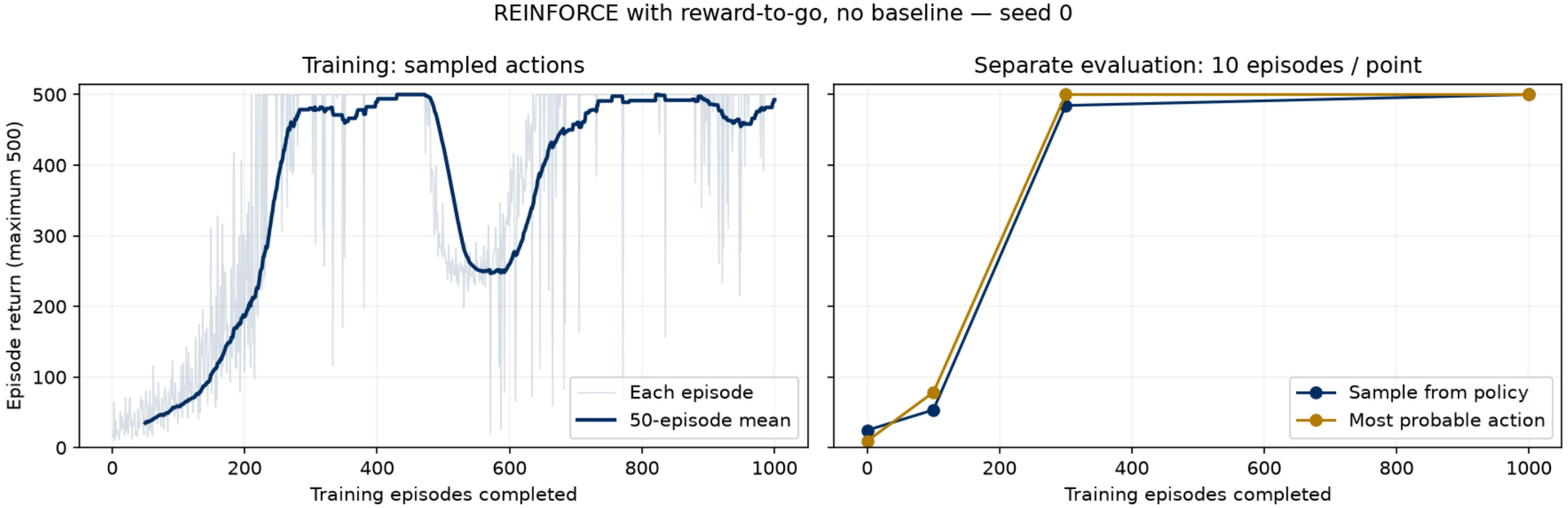
Before training



After training



A training run



Variance

One episode = one sample of the gradient.

The same policy can get very different returns.

Bigger batches help. Can we also get a better signal for each action?

Baselines

Problem: in CartPole every return is positive, so every action we took gets pushed up.

It only balances out because probabilities sum to one.

Better: incentivize actions that did "better than expected"

E.g., say from this state we usually get a return of about 80:

Return	vs. usual	Then...
100	$100 - 80 = +20$	Make this action more likely
60	$60 - 80 = -20$	Make this action less likely

Idea: subtract a **baseline** $b(s)$:

$$\hat{g} = \sum_t [\, G_t - b(S_t) \,] \nabla_\theta \log \pi_\theta(A_t \mid S_t)$$

Baselines are unbiased

Q: Can we subtract anything we want?

A: Anything that doesn't depend on the action. Expected contribution is always zero:

$$\mathbb{E}_{a\sim\pi_\theta}\left[\, b(s)\nabla_\theta \log \pi_\theta(a \mid s) \,\right] = \sum_a \pi_\theta(a \mid s)\, b(s)\, \frac{\nabla_\theta \pi_\theta(a \mid s)}{\pi_\theta(a \mid s)}$$

$$= b(s)\nabla_\theta \sum_a \pi_\theta(a \mid s) = b(s)\nabla_\theta 1 = 0$$

Good choice: $b(s) = V_w(s) \approx V^{\pi_\theta}(s)$, learned by regression on returns.

Sutton & Barto, 2nd ed., ch. 13: policy gradients, REINFORCE, actor–critic.

Actor–critic

Do we have to wait?

> 🧠 **Recall**
> Last lecture: Monte Carlo waits for the return, whereas TD methods bootstraps.

REINFORCE is Monte Carlo: it waits for G_t.

But now we have a value estimate $V_w(s)$...

$$G_t = R_{t+1} + G_{t+1}$$

Idea: same as TD. Replace G_{t+1} by $V_w(S_{t+1})$.

TD error as a signal

We get the TD error again ($\gamma = 1$):

$$\delta_t = R_{t+1} + V_w(S_{t+1}) - V_w(S_t)$$

Reward + estimated future, minus what we expected.

If V_w were exact:

$$\mathbb{E}[\delta_t \mid S_t = s, A_t = a] = Q^\pi(s, a) - V^\pi(s)$$

How much better is action a than usual? This is the **advantage** of a.

Actor and critic

Critic: V_w

Learns to predict returns.

$w \leftarrow w + \beta \delta_t \nabla_w V_w(S_t)$

Actor: π_θ

Learns to act.

$\theta \leftarrow \theta + \alpha \delta_t \nabla_\theta \log \pi_\theta(A_t \mid S_t)$

Act with π_θ → **Observe reward + state** → **Compute δ_t, update both**

Same δ_t, two uses: critic improves predictions, actor improves actions.

Can update after every step: no need to wait for the episode to end.

Sutton & Barto, 2nd ed., ch. 13: policy gradients, REINFORCE, actor–critic.

Learned rewards

So far, the environment gave us rewards.

Q: What's the reward for a language model's answer?

One option (RLHF): ask people which of two answers is better, and learn to predict that preference (reward model).

Compare responses	→	**Learn a reward model**	→	**Improve the policy**
Human feedback		Score responses		Policy gradient!

Careful: reward model $\neq$ critic. One predicts reward, the other future return.

More on this when we get to RL for language models.

Stiennon et al. Learning to summarize from human feedback. NeurIPS, 2020.

Summary

Function approximation: one transition updates predictions everywhere.

Method	Learns	Learning signal
Q-learning / DQN	Action values Q	Reward + best next-state estimate
REINFORCE	Policy π	Complete returns
Actor–critic	Policy + values	TD error (estimated advantage)

Neural networks learn the features, both for values and for policies.

All still depend on the experience the policy manages to collect.

So far: the environment is just stochastic.

Next: what if it is **adversarial**? We'll see how to play against an opponent!