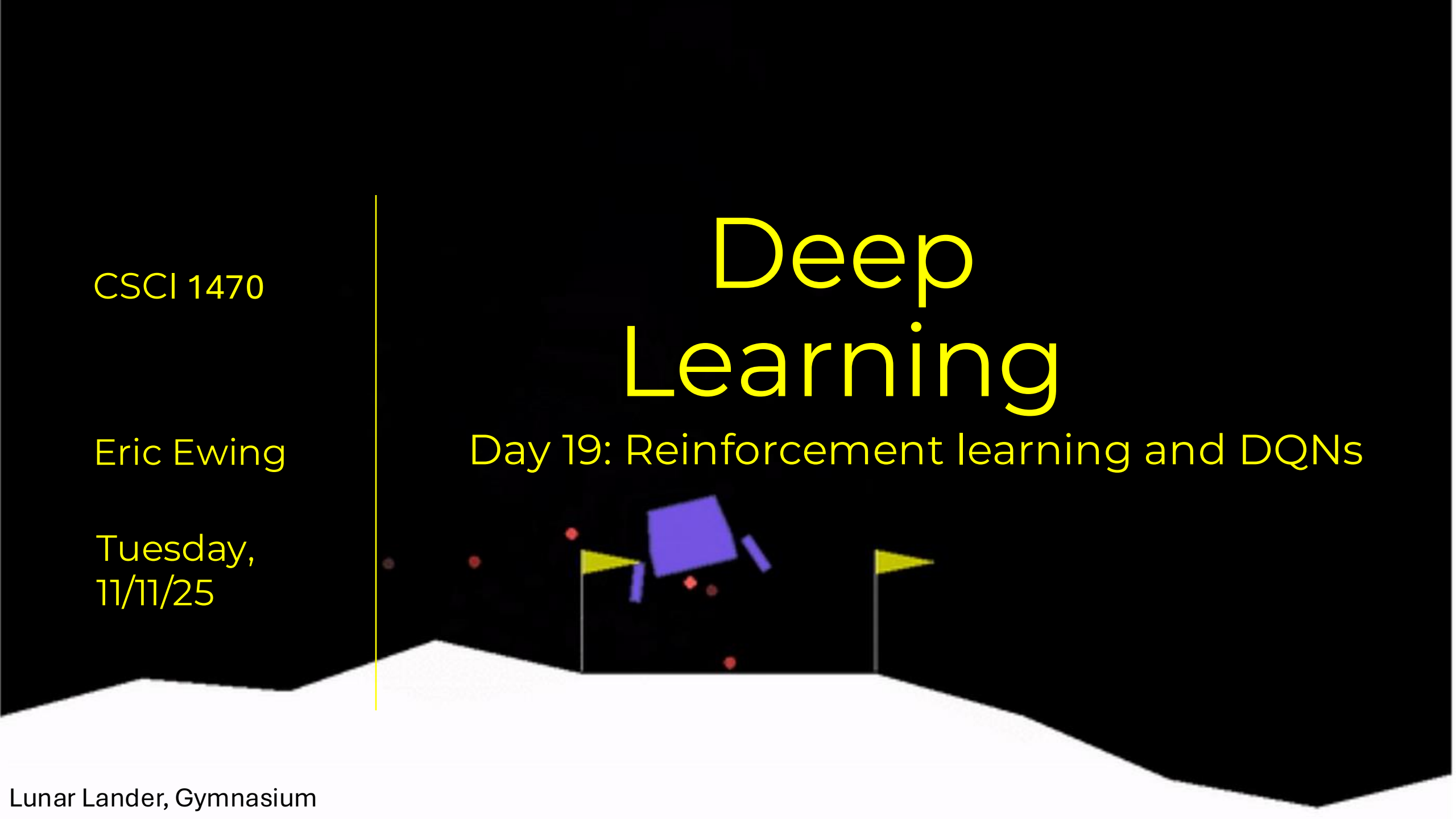
CSCI 1470

Deep Learning

Day 19: Reinforcement learning and DQNs

Eric Ewing

Tuesday, 11/11/25

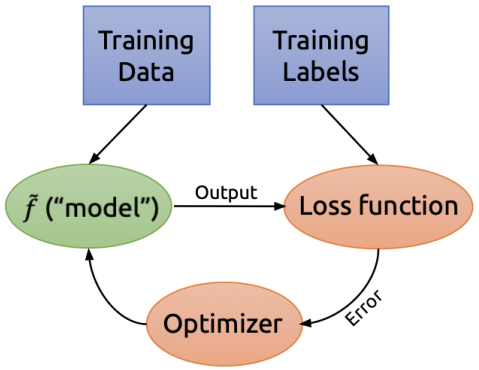


Lunar Lander, Gymnasium

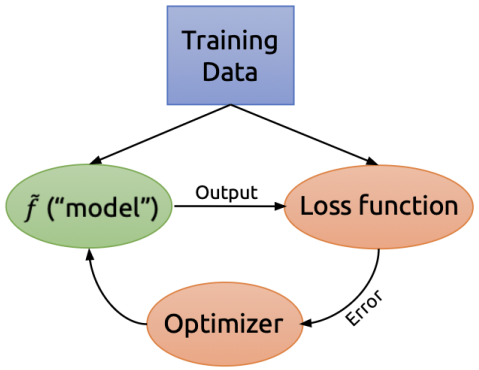
What we've done so far

Different Learning Paradigms

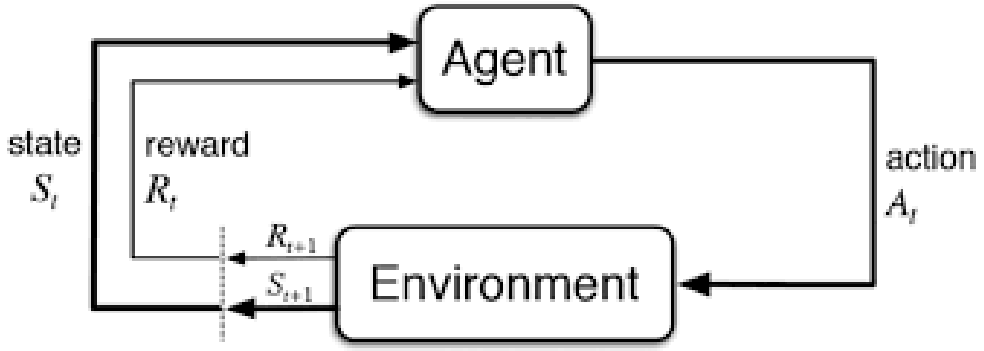
Supervised Learning



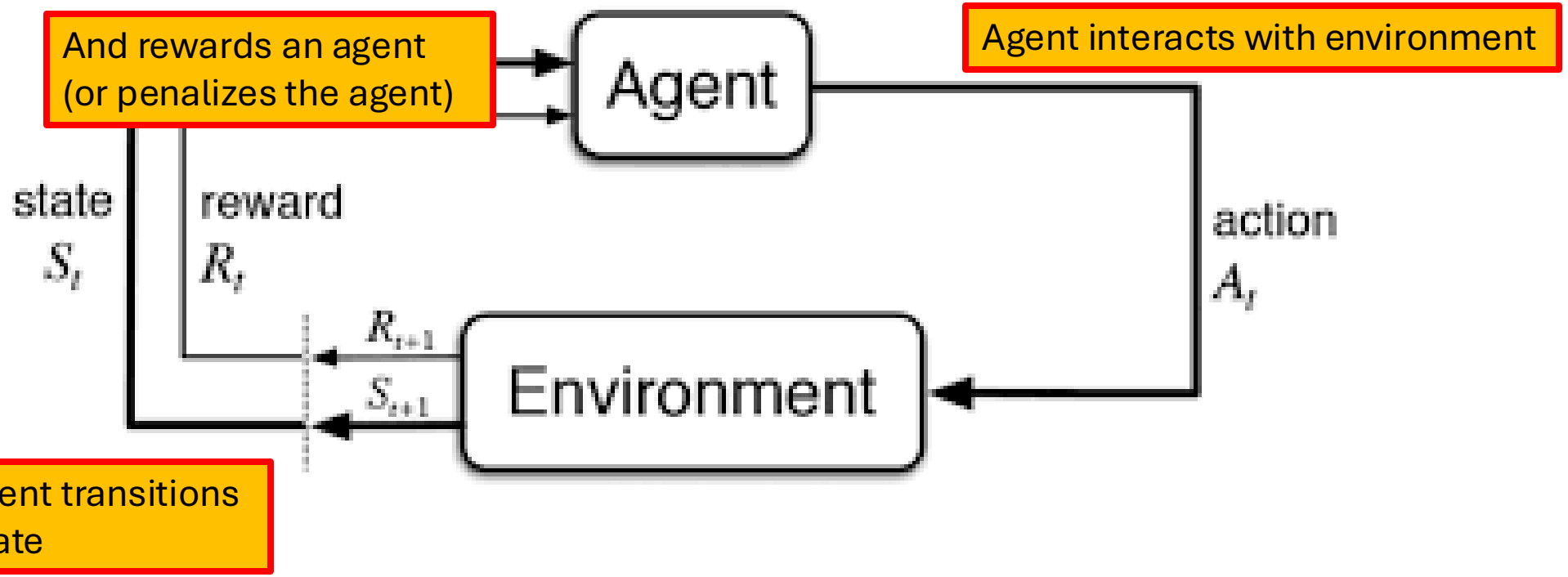
Unsupervised Learning



Reinforcement Learning



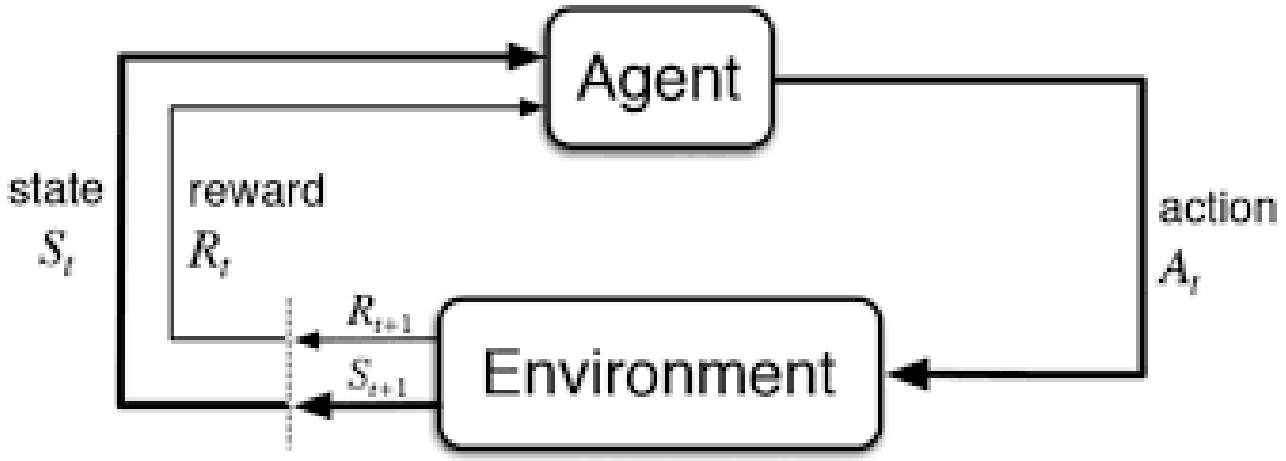
We've focused on this thus far...


And rewards an agent
(or penalizes the agent)
Agent
Agent interacts with environment
state
S_t
reward
R_t
action
A_t
R_{t+1}
S_{t+1}
Environment
Environment transitions
to new state

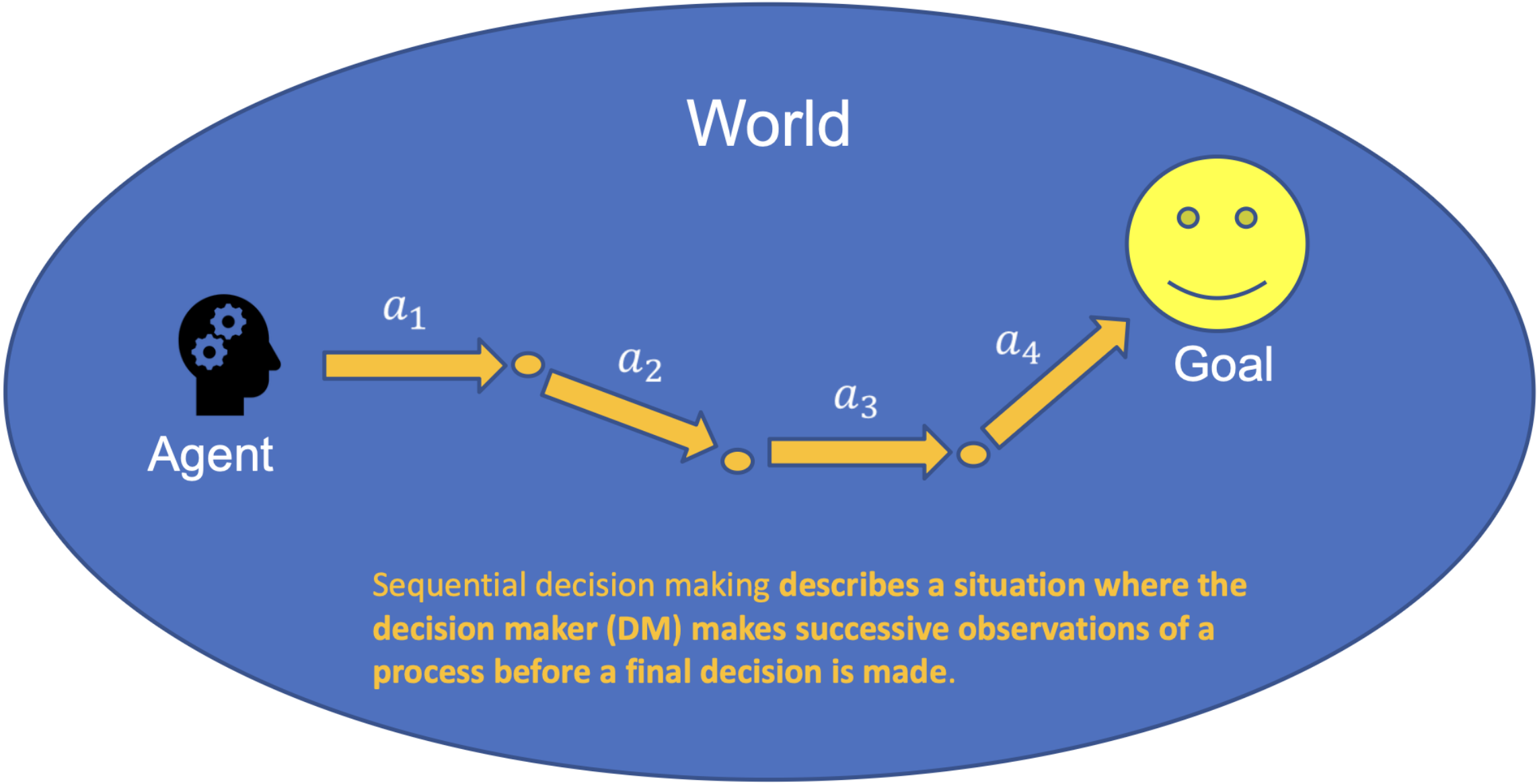
Why Reinforcement Learning?

- Reinforcement learning doesn't require data in the same way that supervised and unsupervised learning do
- There is no dataset *X* required, just a model of the environment
- Agents learn from interacting with the environment

This is how you got so smart…



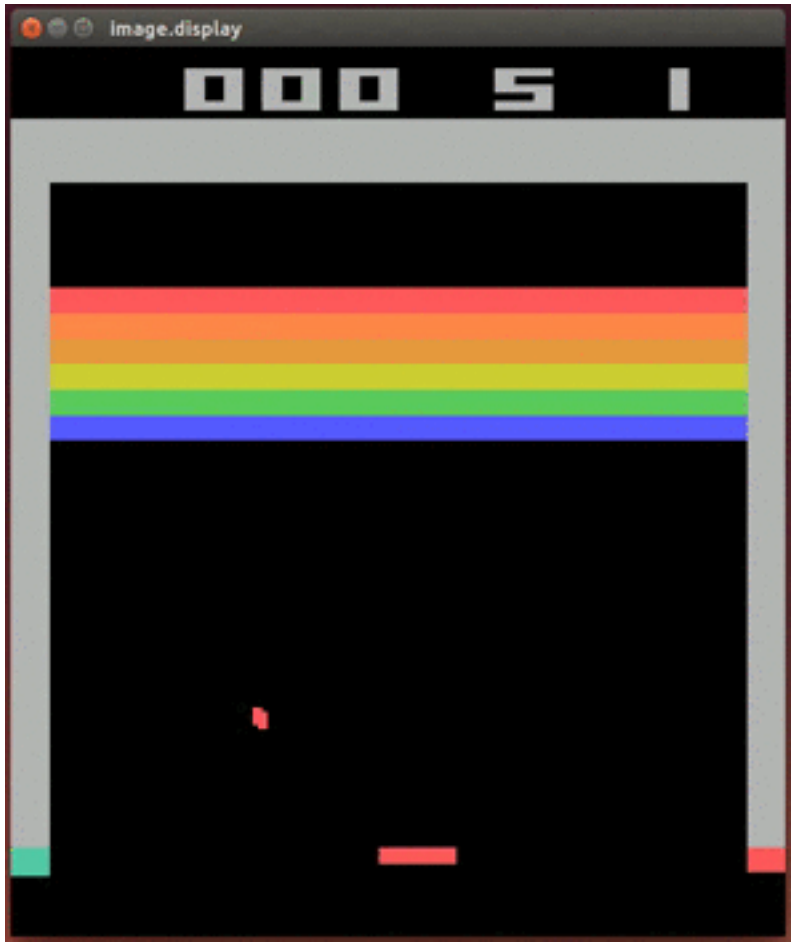
RL: Sequential Decision Making



What's a common example of a sequential decision making process?

- Playing games!
- Let’s look at a specific example…

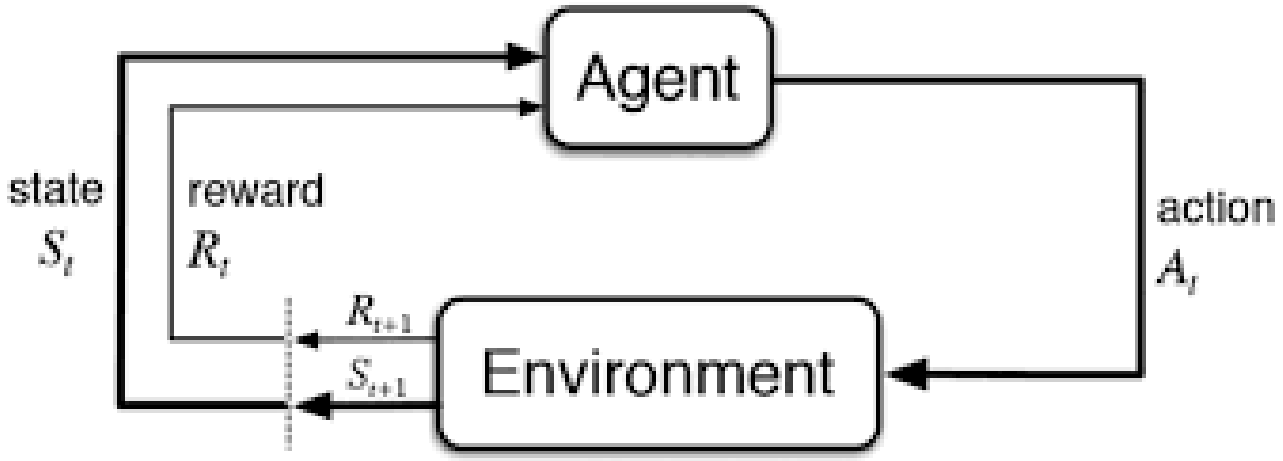
Atari!



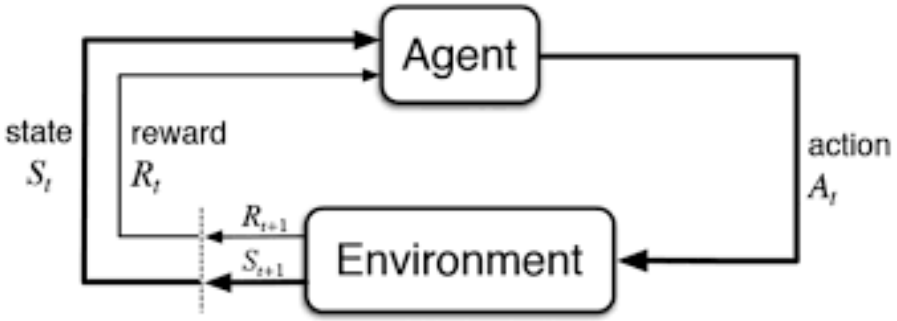
Mnih et al. Playing Atari with Deep Reinforcement Learning, 2013

Markov Decision Processes (MDPs)

How can we formalize the problem we are trying to solve? What components does it have?



Markov Decision Processes (MDPs)



- Set of States: S
  - All possible configurations the world can be in
- Set of Actions: A
  - All possible actions the agent is able to take
- Reward Function: R: $S \rightarrow \mathbb{R}$
  - Reward function takes in a state and returns a number
- Transition Function: T: $S \times A \times S \rightarrow \mathbb{R}$
  - If you take an action in a specific state, what's the probability you transition to any other state?

States

What would the state for breakout be?

Option: Location of paddle, ball, and all breakable blocks

Option: The image of the game…

Actions

What actions can the agent take?

A = Left or Right

Reward Function

What is the reward function?

There is no predefined reward function necessarily

We can use:

1. The score (get reward when a block is broken)
2. Large penalty for losing, Large reward for winning
3. And many other combinations of things

Transition Function

In general, MDPs describe stochastic processes. There can be randomness in what happens.

Breakout is deterministic, the physics of the ball is known and when you tell your paddle to go left it goes left.

Solving MDPs

What would it mean to solve an MDP, like breakout?

Policy: A function $\pi: S \rightarrow A$, that takes in a state and returns an action

We seek the best possible policy π^*, that could tell us the best action to take in any state.

But how do we know one policy is better than another?
If we try to learn a policy, what would our loss function be?
And many many more remaining questions… for next time

Gamma

One more term to add to MDPs: a discount factor $\gamma \in [0, 1]$

Some MDPs have no terminal state (or otherwise can have an agent take infinitely many actions)

We care about the total reward the agent gets, how do we reason about that when we need to sum infinitely many things?

The discount factor is a helpful mathematical trick: Each step into the future, we care about reward a little less

$$r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$

This sum is never infinite if *r* is bounded

Key Terms

Episode: (For *Episodic* MDPs with defined start and terminal states) a single run through of the MDP from a start state to a terminal state (or until a cutoff time *T*)

Trajectory: state, action, reward for every timestep in an episode

$\tau = (s_0, a_0, r_0, \ldots, s_T, a_T, r_T)$

Return: Cumulative discounted rewards from timestep *t* for a single episode

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$

$$G_t = \sum_{i=0}^{T-t} \gamma^i r^{i+t}$$

The return G_0 (sometimes just denoted G) is total discounted reward of the entire episode

Breakout Example

- Episode: From start of game until player loses (or wins)
- Trajectory: list of all states, actions, and rewards from that episode
- Return: Cumulative discounted reward of that trajectory (if $\gamma = 1$, then it is the sum of all rewards)

Key Terms

Value of a state: the **expected** returns from a state

$$V(s_t) = \mathbb{E}[G_t]$$

Q-Values: The expected returns of being in a state and **taking an action**

$$Q(s_t, a_t) = \mathbb{E}_{s' \sim T(s_t, a_t)}[V(s')]$$

Value Function

A value function is defined for a specific policy π!

(If you have a bad policy, you expect your values to be smaller)

$$V^{\pi}(s_t) = \mathbb{E}[G_t]$$

$$V^{\pi}(s_t) = \mathbb{E}\left[\sum_{i=0}^{T-t} \gamma^i r^{i+t}\right]$$

$$V^{\pi}(s_t) = r_t + \gamma \cdot \mathbb{E}\left[\sum_{i=1}^{T-t} \gamma^i r^{i+t}\right]$$

$$V^{\pi}(s_t) = r_t + \gamma \sum_{s' \in S} \Pr(s'|s_t, a_t)\, V^{\pi}(s')$$

This is called policy evaluation

We can the value function as a recursive formula:
How good it is to be in a state is the immediate reward for being in that state + the expected returns for future states

Value Function → Policy

What if we don't have a policy already and want to find one?

If we already have a value function:

For every state

iterate over all possible actions

calculate the expected value if the agent takes that action

$Q(s,a) = \sum_{s'} \Pr(s'|s,a)\,[R(s') + \gamma V(s')]$

set $\pi(s)$ to be the action with highest expected Q-value

This is called *policy improvement*

Value Iteration

1. Start with a random Value function V
2. Run Policy Improvement to determine best actions at each state
3. Run Policy Estimation to determine the new values with the updated policy
4. Repeat

Value Iteration

Repeatedly apply Policy Estimation and Policy improvement steps

Run until convergence (i.e., estimates of V no longer changes)

Algorithm 1 Value Iteration

Initialize $V(s) = 0$ for all states $s \in \mathcal{S}$
Initialize threshold $\theta > 0$ (convergence criterion)
repeat
  $\Delta \leftarrow 0$
  for each state $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s'|s,a)[R(s') + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end for
until $\Delta < \theta$
Return V

Tabular Value Iteration

Value iteration is typically a **dynamic programming** algorithms

A table of values is constructed (one row for each state) and then updated according to the Bellman Equation:

$$V(s) = r + \gamma \max_{a} \sum_{s'} \Pr(s'|s,a)\, V(s')$$

Q-Learning

Q-Learning is our first actual RL algorithm

- Reinforcement Learning algorithms actually simulate episodes, gather trajectories, and learn from experiences.

- Collect experiences (i.e.: (s, a, r, s') tuples)

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

Q-Learning

Q-Learning is our first actual RL algorithm

- Reinforcement Learning algorithms actually simulate episodes, gather trajectories, and learn from experiences.

- Collect experiences (i.e.: (s, a, r, s') tuples)

$$Q(s,a) = r + \gamma \max_{a'} Q(s', a')$$

Important:
How do we collect experiences (i.e., how do we select what action to take)?

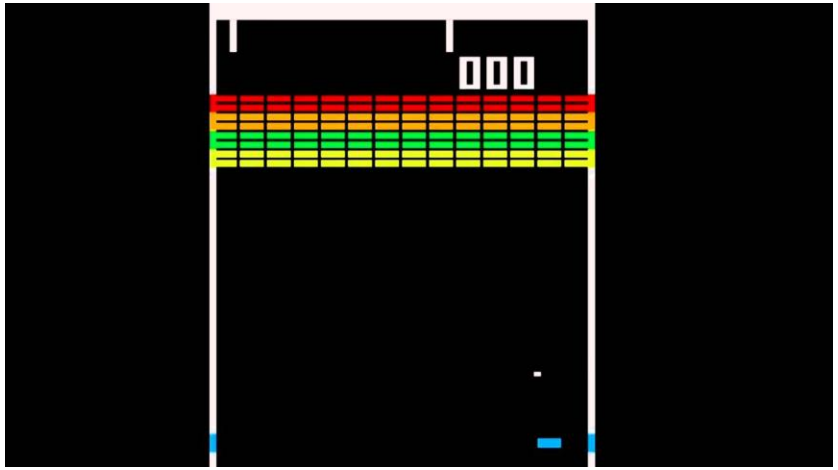
How do we update our estimates of Q?

Collecting Experiences

$$Q(s,a) = r + \gamma \max_{a'} Q(s',a')$$

What if we always took the action go-right?

- We'd update our estimates for go-right, but never go-left



What if we take uniform random actions?

- We'd update estimates for both right and left, but we'd be unlikely to get too far into the game

What if we find a happy middle ground between fully deterministic and fully random?

- With probability ϵ take a random action
- With probability $1 - \epsilon$ take the best action (action with highest Q-value)

ϵ-greedy Algorithm for balancing exploration in RL

$$Q(s,a) = r + \gamma \max_{a'} Q(s',a')$$
$$0 = [r + \gamma \max_{a'} Q(s',a')] - Q(s,a)$$

Updating estimates of Q-values

Q-learning:

Maintain estimates of Q(s, a) for all (s, a) pairs

Collect experiences, update Q estimates with:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

Learning rate

Current estimate

Error in estimate (Temporal Difference Error)

Tabular Q-Learning

Algorithm 2 Q-Learning

Initialize $Q(s, a) = 0$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}$
Initialize learning rate $\alpha \in (0, 1]$ and discount factor $\gamma \in [0, 1)$
Initialize exploration parameter $\epsilon \in (0, 1)$
for each episode **do**
    Initialize state s
    repeat
        With probability ϵ: choose a random action $a \in \mathcal{A}$
        Otherwise: choose $a = \arg\max_{a'} Q(s, a')$
        Take action a, observe reward r and next state s'
        $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
        (Or $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a'))$
        $s \leftarrow s'$
    until s is terminal
end for
Return Q

Where's the Deep Learning part of this?

- Neural Networks are Function approximators and we have some functions…
  - $V: S \rightarrow \mathbb{R}$
  - $Q: S \times A \rightarrow \mathbb{R}$
  - $\pi: S \rightarrow A$
- Deep Reinforcement Learning seeks to approximate these functions with neural networks

Deep Q-Learning

- Approximate Q-values with a neural network
- Always needed a loss function with neural networks before…
- Can we come up with a loss function here?
- We want this equality to hold: $0 = [r + \gamma \max_{a'} Q(s', a')] - Q(s, a)$
- If we can force $[r + \gamma \max_{a'} Q(s', a')] - Q(s, a)$ to be close to 0, we will have good approximations of Q-values

$$L = \left(\left[r + \gamma \max_{a'} Q(s', a') \right] - Q(s, a) \right)^2$$

Q-Learning

How to update tabular Q-learning to be deep Q-learning

$$L = \left(\left[r + \gamma \max_{a'} Q(s', a') \right] - Q(s, a) \right)^2$$

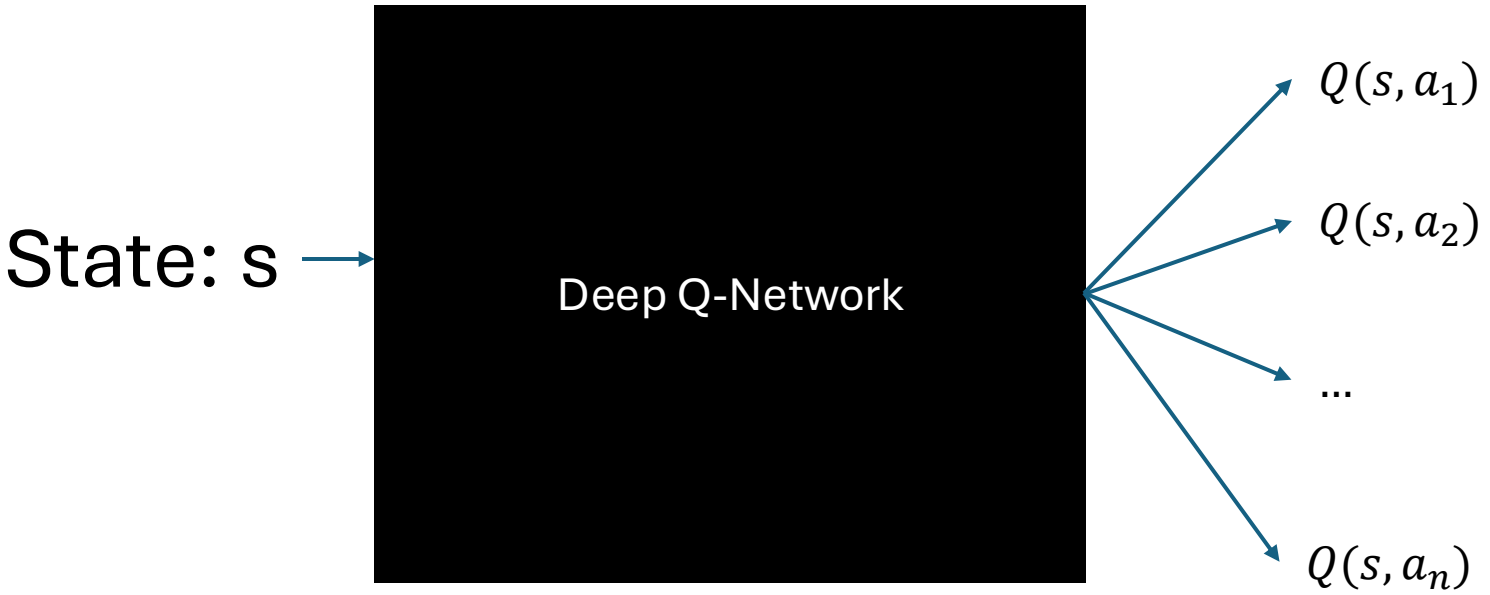
Algorithm 2 Q-Learning

Initialize $Q(s, a) = 0$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}$
Initialize learning rate $\alpha \in (0, 1]$ and discount factor $\gamma \in [0, 1)$
Initialize exploration parameter $\epsilon \in (0, 1)$
for each episode **do**
  Initialize state s
  repeat
    With probability ϵ: choose a random action $a \in \mathcal{A}$
    Otherwise: choose $a = \arg\max_{a'} Q(s, a')$
    Take action a, observe reward r and next state s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    (Or $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a'))$
    $s \leftarrow s'$
  until s is terminal
end for
Return Q

Can't just update outputs of a NN directly…
Instead, compute loss and run a step of SGD

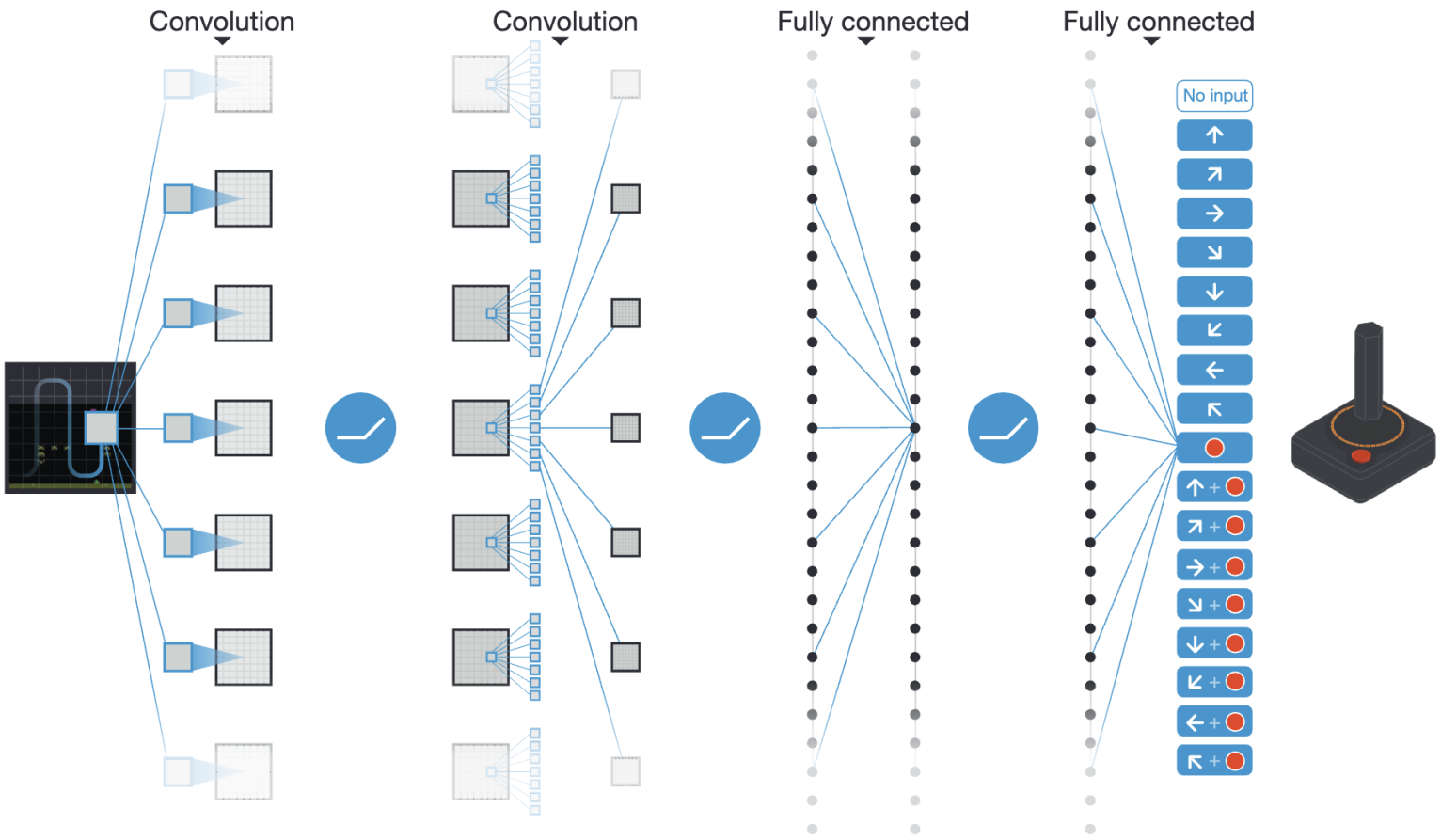
Deep-Q Network

Deep Q-Networks (DQNs):
1. Take in a state
2. Return Q-values for each action



What activation function should the final layer use?

DQN for Atari



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

Deep-Q Learning

Initialize DQN to approximate Q

Maintain estimates of Q(s, a) for all (s, a) pairs

Collect experiences, update Q estimates with:

Compute $L_\theta = \left[r + \gamma \max_{a'} Q_\theta(s', a') - Q_\theta(s, a)\right]^2$

update θ with SGD on Loss function

(non-)Stationarity in RL

$$L_\theta = \Big[\overbrace{r + \gamma \max_{a'} Q_\theta(s', a')}^{\text{Target}} - \overbrace{Q_\theta(s, a)}^{\text{Estimate}}\Big]^2$$

We'd like our current estimate $Q_\theta(s, a)$ to be like our estimate for the next timestep $r + \gamma \max_{a'} Q_\theta(s', a')$.

(non-)Stationarity in RL

$$L_\theta = \Big[\overbrace{r + \gamma \max_{a'} Q_\theta(s', a')}^{\text{Target}} - \overbrace{Q_\theta(s, a)}^{\text{Estimate}}\Big]^2$$

We'd like our current estimate $Q_\theta(s, a)$ to be like our estimate for the next timestep $r + \gamma \max_{a'} Q_\theta(s', a')$.

We do not include $\nabla Q_\theta(s', a')$ when calculating $\nabla_\theta L$, we treat $\gamma \max_{a'} Q_\theta(s', a')$ as a constant:

(non-)Stationarity in RL

$$L_\theta = \Big[\overbrace{r + \gamma \max_{a'} Q_\theta(s', a')}^{\text{Target}} - \overbrace{Q_\theta(s, a)}^{\text{Estimate}}\Big]^2$$

We'd like our current estimate $Q_\theta(s, a)$ to be like our estimate for the next timestep $r + \gamma \max_{a'} Q_\theta(s', a')$.

We do not include $\nabla Q_\theta(s', a')$ when calculating $\nabla_\theta L$, we treat $\gamma \max_{a'} Q_\theta(s', a')$ as a constant:

1. $\max_{a'} Q_\theta(s', a')$ is not differentiable

(non-)Stationarity in RL

$$L_\theta = \Big[\overbrace{r + \gamma \max_{a'} Q_\theta(s', a')}^{\text{Target}} - \overbrace{Q_\theta(s, a)}^{\text{Estimate}}\Big]^2$$

We'd like our current estimate $Q_\theta(s, a)$ to be like our estimate for the next timestep $r + \gamma \max_{a'} Q_\theta(s', a')$.

We do not include $\nabla Q_\theta(s', a')$ when calculating $\nabla_\theta L$, we treat $\gamma \max_{a'} Q_\theta(s', a')$ as a constant:

1. $\max_{a'} Q_\theta(s', a')$ is not differentiable
2. $\nabla Q_\theta(s', a')$ would tell us how to update the target to match our current estimate (that's backwards)

(non-)Stationarity in RL

$$L_\theta = \Big[\overbrace{r + \gamma \max_{a'} Q_\theta(s', a')}^{\text{Target}} - \overbrace{Q_\theta(s, a)}^{\text{Estimate}}\Big]^2$$

We'd like our current estimate $Q_\theta(s, a)$ to be like our estimate for the next timestep $r + \gamma \max_{a'} Q_\theta(s', a')$.

If we included the target gradient, it would be like trying to update our estimate to fit our target AND update our target to fit our estimate at the same time

(non-)Stationarity in RL

$$L_\theta = \Big[\overbrace{r + \gamma \max_{a'} Q_\theta(s', a')}^{\text{Target}} - \overbrace{Q_\theta(s, a)}^{\text{Estimate}}\Big]^2$$

We'd like our current estimate $Q_\theta(s, a)$ to be like our estimate for the next timestep $r + \gamma \max_{a'} Q_\theta(s', a')$.

If we included the target gradient, it would be like trying to update our estimate to fit our target AND update our target to fit our estimate at the same time

Using only the gradient of the estimate helps with *stationarity*

Double DQN

Use a separate target and prediction network for stability. Every so often, update the target network to be the Q-network.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

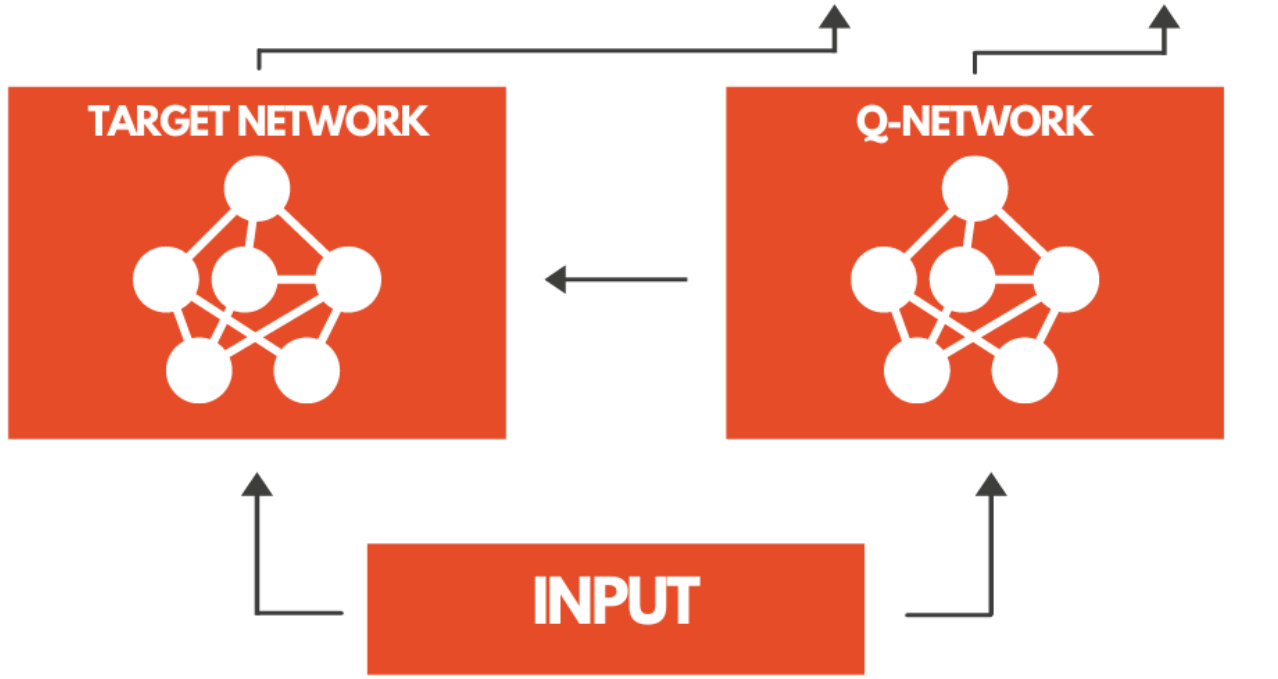
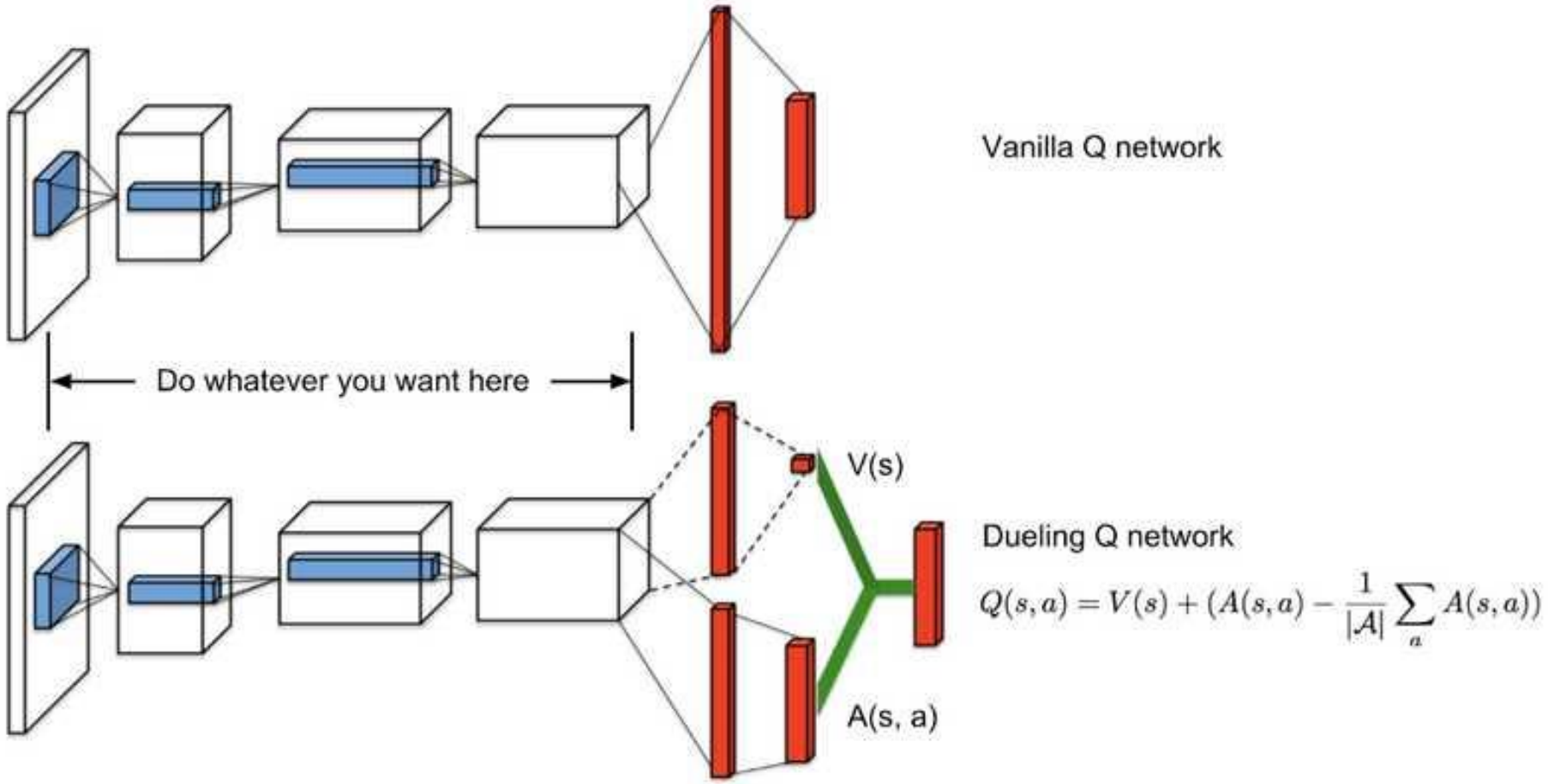


Image source: https://rubikscode.net/2021/07/20/introduction-to-double-q-learning/

Dueling Q Networks

Advantage function A(s, a):
Q(s, a) = V(s) + A(s, a)

Estimate V(s) and the advantage A(s, a) separately to create Q-value estimates.



Q-Values to Policy

What do we do after we learn Q? We need to turn them into a policy.

For a given state, take the action associated with the best Q-value.

$$\pi(s) = argmax_a \, Q(s, a)$$

Policies

Why learn Q-values first and turn them into a policy? Why not just learn a policy?

Policies

Why learn Q-values first and turn them into a policy? Why not just learn a policy?

Policies

Why learn Q-values first and turn them into a policy? Why not just learn a policy?

What should the activation function be for the final layer?

How do we train a policy network?

How do we train a policy network?

Need to find an appropriate loss function.

How do we train a policy network?

Need to find an appropriate loss function.

What’s our objective?

How do we train a policy network?

Need to find an appropriate loss function.

What's our objective?

Find a policy π such that the value of the start state is maximized:

How do we train a policy network?

Need to find an appropriate loss function.

What's our objective?

Find a policy π such that the value of the start state is maximized:

$$\pi = \text{argmax}_{\pi} \left(V(s_0)\right)$$

How do we train a policy network?

Need to find an appropriate loss function.

What's our objective?

Find a policy π such that the value of the start state is maximized:

$$\pi = \text{argmax}_{\pi} \left(V(s_0)\right)$$

How can we maximize $V(s_0)$?

Let $J(\theta)$ be our objective function:

$$J(\theta) = V(s_0)$$

Let $J(\theta)$ be our objective function:

$$J(\theta) = V(s_0)$$

$$J(\theta) = \mathbb{E}[G_0]$$

Let $J(\theta)$ be our objective function:

$$J(\theta) = V(s_0)$$

$$J(\theta) = \mathbb{E}[G_0]$$

$$J(\theta) = \sum_{\tau} \Pr(\tau|\theta)\, G(\tau)$$

Let $J(\theta)$ be our objective function:

$$J(\theta) = V(s_0)$$

$$J(\theta) = \mathbb{E}[G_0]$$

$$J(\theta) = \sum_{\tau} \Pr(\tau|\theta)\, G(\tau)$$

Probability of a trajectory occurring

Returns of a specific trajectory

Let $J(\theta)$ be our objective function:

$$J(\theta) = V(s_0)$$

$$J(\theta) = \mathbb{E}[G_0]$$

$$J(\theta) = \sum_{\tau} \Pr(\tau|\theta)\, G(\tau)$$

Returns of a specific trajectory

Probability of a trajectory occurring

$$\Pr(\tau|\theta) = \Pi_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$$

Let $J(\theta)$ be our objective function:

$$J(\theta) = V(s_0)$$

$$J(\theta) = \mathbb{E}[G_0]$$

$$J(\theta) = \sum_{\tau} \Pr(\tau|\theta)\, G(\tau)$$

Returns of a specific trajectory

Probability of a trajectory occurring

$$\Pr(\tau|\theta) = \Pi_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$$

State transition Probability

Probability of taking an action for a given state

Let $J(\theta)$ be our objective function:

$$J(\theta) = V(s_0)$$

$$J(\theta) = \mathbb{E}[G_0]$$

$$J(\theta) = \sum_{\tau} \Pr(\tau|\theta)\, G(\tau)$$

Sum over all possible trajectories

Probability of a trajectory occurring

Returns of a specific trajectory

$$\Pr(\tau|\theta) = \Pi_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$$

State transition Probability

Probability of taking an action for a given state

Log-Derivative Trick

We can rewrite the derivative of a function using the derivative of the natural log function:

$$\nabla \ln f(x) = \frac{\nabla \mathrm{f}(x)}{f(x)}$$

$$\nabla f(x) = f(x) \nabla \ln \mathrm{f}(\mathrm{x})$$

When applied to $\Pr(\tau|\theta)$:

$$\nabla_\theta \Pr(\tau|\theta) = \Pr(\tau|\theta)\, \nabla_\theta \ln \Pr(\tau|\theta)$$

Log Probability Trick

$$\Pr(\tau|\theta) = \Pi_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$$

Log Probability Trick

$$\Pr(\tau|\theta) = \Pi_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$$

$$\nabla_\theta \Pr(\tau|\theta) = \Pr(\tau|\theta)\, \nabla_\theta \ln \Pr(\tau|\theta)$$

This gradient term is what we want to calculate

Log Probability Trick

$$\Pr(\tau|\theta) = \Pi_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$$

$$\nabla_\theta \Pr(\tau|\theta) = \Pr(\tau|\theta)\, \nabla_\theta \ln \Pr(\tau|\theta)$$

This gradient term is what we want to calculate

$$\nabla_\theta \ln \Pr(\tau|\theta) = \nabla_\theta \sum_{t=0}^{T} \ln P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$$

Log Probability Trick

$$\Pr(\tau|\theta) = \Pi_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$$

$$\nabla_\theta \Pr(\tau|\theta) = \Pr(\tau|\theta)\, \nabla_\theta \ln \Pr(\tau|\theta)$$

This gradient term is what we want to calculate

$$\nabla_\theta \ln \Pr(\tau|\theta) = \nabla_\theta \sum_{t=0}^{T} \ln P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$$

$$\nabla_\theta \ln \Pr(\tau|\theta) = \nabla_\theta \sum_{t=0}^{T} \ln P(s_{t+1}|s_t, a_t) + \ln \pi_\theta(a_t|s_t)$$

Log Probability Trick

$$\Pr(\tau|\theta) = \Pi_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$$

$$\nabla_\theta \Pr(\tau|\theta) = \Pr(\tau|\theta)\, \nabla_\theta \ln \Pr(\tau|\theta)$$

This gradient term is what we want to calculate

$$\nabla_\theta \ln \Pr(\tau|\theta) = \nabla_\theta \sum_{t=0}^{T} \ln P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$$

Log of product -> sum of logs

$$\nabla_\theta \ln \Pr(\tau|\theta) = \nabla_\theta \sum_{t=0}^{T} \ln P(s_{t+1}|s_t, a_t) + \ln \pi_\theta(a_t|s_t)$$

Log of product -> sum of logs

$$\nabla_\theta \ln \Pr(\tau|\theta) = \sum_{t=0}^{T} \nabla_\theta \ln P(s_{t+1}|s_t, a_t) + \nabla_\theta \ln \pi_\theta(a_t|s_t)$$

Derivative of sum -> sum of derivative

Gradient of a trajectory

$$\nabla_\theta \ln \Pr(\tau|\theta) = \sum_{t=0}^{T} \nabla_\theta \ln P(s_{t+1}|s_t, a_t) + \nabla_\theta \ln \pi_\theta(a_t|s_t)$$

State transition function does not depend on θ!

$$\nabla_\theta \ln \Pr(\tau|\theta) = \sum_{t=0}^{T} \nabla_\theta \ln \pi_\theta(a_t|s_t)$$

Policy Gradient Derivation

Putting it all back together:

$$J(\theta) = \sum_{\tau} \Pr(\tau|\theta)\, G(\tau)$$

Our Objective

Policy Gradient Derivation

Putting it all back together:

$$J(\theta) = \sum_{\tau} \Pr(\tau|\theta)\, G(\tau)$$ Our Objective

$$\nabla_\theta J(\theta) = \sum_{\tau} \nabla_\theta \Pr(\tau|\theta)\, G(\tau)$$ Take the gradient

Policy Gradient Derivation

Putting it all back together:

$$J(\theta) = \sum_{\tau} \Pr(\tau|\theta)\, G(\tau)$$ Our Objective

$$\nabla_\theta J(\theta) = \sum_{\tau} \nabla_\theta \Pr(\tau|\theta)\, G(\tau)$$ Take the gradient

$$\nabla_\theta J(\theta) = \sum_{\tau} \Pr(\tau|\theta) G(\tau) \nabla_\theta \ln \Pr(\tau|\theta)$$ Log-Derivative Trick

Policy Gradient Derivation

Putting it all back together:

$$J(\theta) = \sum_{\tau} \Pr(\tau|\theta)\, G(\tau)$$ Our Objective

$$\nabla_\theta J(\theta) = \sum_{\tau} \nabla_\theta \Pr(\tau|\theta)\, G(\tau)$$ Take the gradient

$$\nabla_\theta J(\theta) = \sum_{\tau} \Pr(\tau|\theta) G(\tau) \nabla_\theta \ln \Pr(\tau|\theta)$$ Log-Derivative Trick

$$\nabla_\theta J(\theta) = \sum_{\tau} [\Pr(\tau|\theta) G(\tau) \sum_{t=0}^{T} \nabla_\theta \ln \pi_\theta(a_t|s_t)\ ]$$ Gradient of a Trajectory

Policy Gradient Derivation

Putting it all back together:

$$J(\theta) = \sum_{\tau} \Pr(\tau|\theta)\, G(\tau)$$ Our Objective

$$\nabla_\theta J(\theta) = \sum_{\tau} \nabla_\theta \Pr(\tau|\theta)\, G(\tau)$$ Take the gradient

$$\nabla_\theta J(\theta) = \sum_{\tau} \Pr(\tau|\theta) G(\tau) \nabla_\theta \ln \Pr(\tau|\theta)$$ Log-Derivative Trick

$$\nabla_\theta J(\theta) = \sum_{\tau} [\Pr(\tau|\theta) G(\tau) \sum_{t=0}^{T} \nabla_\theta \ln \pi_\theta(a_t|s_t)\ ]$$ Gradient of a Trajectory

$$\nabla_\theta J(\theta) = \mathbb{E}[G_0 \sum_{t=0}^{T} \nabla_\theta \ln \pi_\theta(a_t|s_t)]$$ Convert back to Expectation

Policy Gradient

Bigger step if better returns

Direction to move in to increase probability of trajectory

$$\nabla_\theta J(\theta) = \mathbb{E}[G_0 \sum_{t=0}^{T} \nabla_\theta \ln \pi_\theta(a_t|s_t)]$$

We will never be able to sum over all possible trajectories…

How do we get around this?

Policy Gradient

Bigger step if better returns

Direction to move in to increase probability of trajectory

$$\nabla_\theta J(\theta) = \mathbb{E}[G_0 \sum_{t=0}^{T} \nabla_\theta \ln \pi_\theta(a_t|s_t)]$$

We will never be able to sum over all possible trajectories…

How do we get around this?

Sampling!

1. Collect n trajectories following policy π_θ
2. $\Pr(\tau|\theta) = 1/n$ for each trajectory
3. Calculate the total return for each trajectory $G(\tau)$

Reward-To-Go Policy Gradient

You can also do the policy gradient derivation such that the gradient does not depend on G_0, but on G_t

$$\nabla_\theta J(\theta) = \mathbb{E}[\sum_{t=0}^{T} G_t \nabla_\theta \ln \pi_\theta(a_t|s_t)]$$

Or

$$\nabla_\theta J(\theta) = \mathbb{E}[\sum_{t=0}^{T} Q(s_t, a_t) \nabla_\theta \ln \pi_\theta(a_t|s_t)]$$

REINFORCE (Policy Gradient Learning)

REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  For each step of the episode $t = 0, \ldots, T-1$:
    $G \leftarrow$ return from step t
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Source: Sutton and Barto, Reinforcement Learning: An Introduction

REINFORCE (Policy Gradient Learning)

REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  For each step of the episode $t = 0, \ldots, T-1$:
    $G \leftarrow$ return from step t
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Why is the update adding the gradient instead of subtracting?

Source: Sutton and Barto, Reinforcement Learning: An Introduction

REINFORCE (Policy Gradient Learning)

REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  For each step of the episode $t = 0, \ldots, T-1$:
    $G \leftarrow$ return from step t
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Why is the update adding the gradient instead of subtracting?

When π is based on a softmax, $\nabla_\theta \ln \pi_\theta(a|s)$ is actually easy to compute by hand using log rules and the fact that $\ln e^x = x$

Source: Sutton and Barto, Reinforcement Learning: An Introduction

Variance of REINFORCE

REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  For each step of the episode $t = 0, \ldots, T-1$:
    $G \leftarrow$ return from step t
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Variance of REINFORCE

REINFORCE has **high** variance

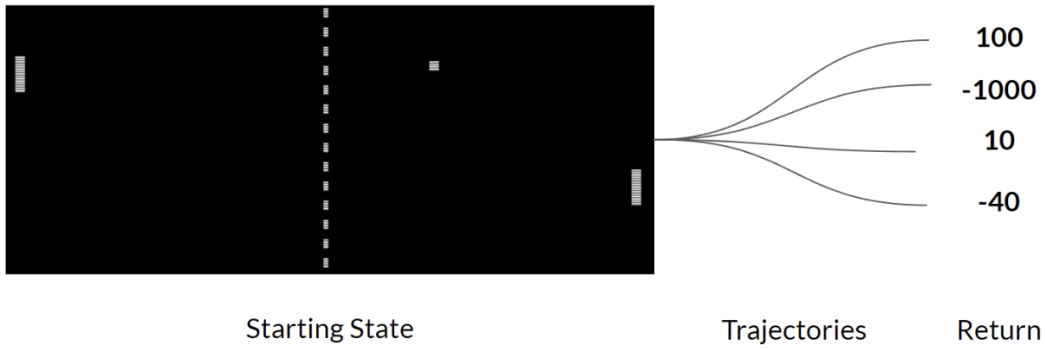
REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  For each step of the episode $t = 0, \ldots, T-1$:
    $G \leftarrow$ return from step t
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Variance of REINFORCE

REINFORCE has **high** variance

It depends heavily on the returns of a single episode



REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

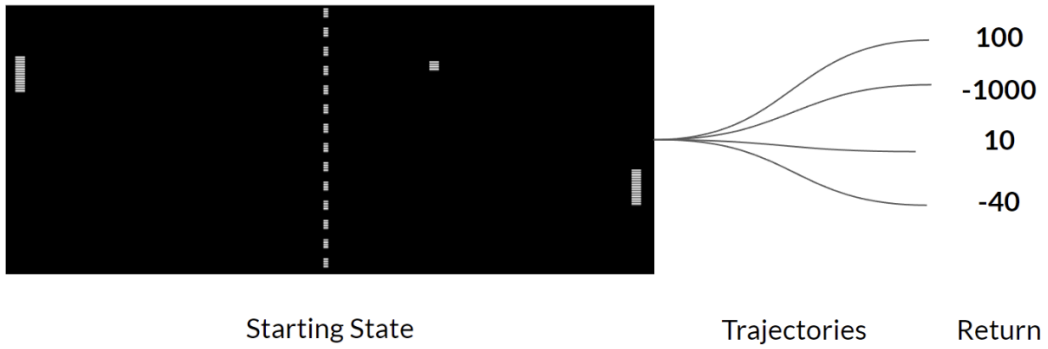
Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  For each step of the episode $t = 0, \ldots, T-1$:
    $G \leftarrow$ return from step t
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Variance of REINFORCE

REINFORCE has **high** variance

It depends heavily on the returns of a single episode

We can reduce variance by collecting more than one trajectory



REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  For each step of the episode $t = 0, \ldots, T-1$:
    $G \leftarrow$ return from step t
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta})$

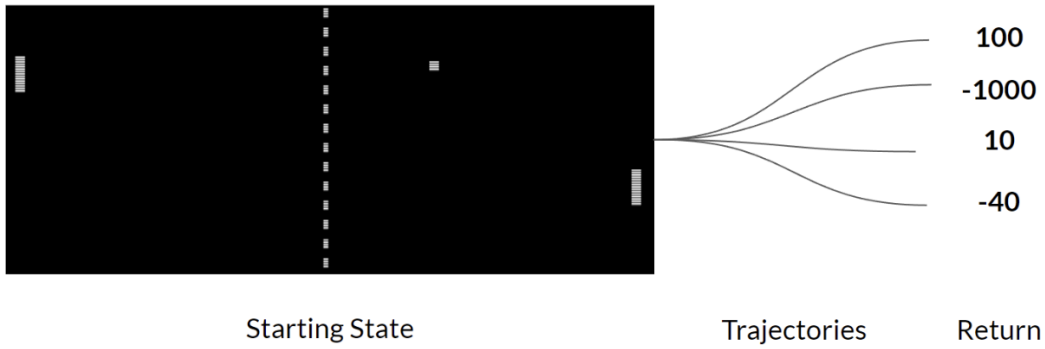
Variance of REINFORCE

REINFORCE has **high** variance

It depends heavily on the returns of a single episode

We can reduce variance by collecting more than one trajectory

Or…



REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  For each step of the episode $t = 0, \ldots, T-1$:
    $G \leftarrow$ return from step t
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Baseline Functions

Baseline Functions

Subtracting a baseline function from G_t does not change the expected gradient

Baseline Functions

Subtracting a *baseline* function from G_t does not change the expected gradient

A baseline function $b(s)$ is any function that depends only on the state (not on actions)

Baseline Functions

Subtracting a baseline function from G_t does not change the expected gradient

A baseline function $b(s)$ is any function that depends only on the state (not on actions)

$$\nabla_\theta J(\theta) = \mathbb{E}[\sum_{t=0}^{T} (G_t - b(s)) \nabla_\theta \ln \pi_\theta(a_t|s_t)]$$

Baseline Functions

Subtracting a baseline function from G_t does not change the expected gradient

A baseline function $b(s)$ is any function that depends only on the state (not on actions)

$$\nabla_\theta J(\theta) = \mathbb{E}[\sum_{t=0}^{T} (G_t - b(s)) \nabla_\theta \ln \pi_\theta(a_t|s_t)]$$

Baseline functions can reduce the variance of the gradient estimate

Baseline Functions

Subtracting a baseline function from G_t does not change the expected gradient

A baseline function $b(s)$ is any function that depends only on the state (not on actions)

$$\nabla_\theta J(\theta) = \mathbb{E}[\sum_{t=0}^{T} (G_t - b(s)) \nabla_\theta \ln \pi_\theta(a_t|s_t)]$$

Baseline functions can reduce the variance of the gradient estimate

The value function V(s) is the ideal baseline function

REINFORCE with Baseline

REINFORCE with Baseline (episodic), for estimating $\pi_\theta \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Pseudocode uses SGD, but you can just as easily use any other optimizer (e.g., Adam)

Source: Sutton and Barto Chapter 13

REINFORCE with Baseline

REINFORCE with Baseline (episodic), for estimating $\pi_\theta \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\qquad (G_t)$
    $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \boxed{\delta \nabla \hat{v}(S_t,\mathbf{w})}$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

Gradient of $\mathrm{L} = \frac{1}{2}\delta\text{^}2$

Pseudocode uses SGD, but you can just as easily use any other optimizer (e.g., Adam)

Source: Sutton and Barto Chapter 13

Extra Material

Sutton and Barto: Policy Gradient methods chapter 13
http://www.incompleteideas.net/book/RLbook2020.pdf

Spinning up policy gradient:
https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html

Derivation of REINFORCE w/ Baseline Function

First, let's show that the gradient estimate is unbiased. We see that with the baseline, we can distribute and rearrange and get:

$$\nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t)\left(\sum_{t'=t}^{T-1} r_{t'}\right) - \sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t)b(s_t)\right]$$

Due to linearity of expectation, all we need to show is that for any single time t, the gradient of $\log \pi_\theta(a_t|s_t)$ multiplied with $b(s_t)$ is zero. This is true because

$$\begin{aligned}
\mathbb{E}_{\tau\sim\pi_\theta}\Big[\nabla_\theta \log \pi_\theta(a_t|s_t)b(s_t)\Big] &= \mathbb{E}_{s_{0:t},a_{0:t-1}}\Big[\mathbb{E}_{s_{t+1:T},a_{t:T-1}}[\nabla_\theta \log \pi_\theta(a_t|s_t)b(s_t)]\Big] \\
&= \mathbb{E}_{s_{0:t},a_{0:t-1}}\Big[b(s_t)\cdot \underbrace{\mathbb{E}_{s_{t+1:T},a_{t:T-1}}[\nabla_\theta \log \pi_\theta(a_t|s_t)]}_{E}\Big] \\
&= \mathbb{E}_{s_{0:t},a_{0:t-1}}\Big[b(s_t)\cdot \mathbb{E}_{a_t}[\nabla_\theta \log \pi_\theta(a_t|s_t)]\Big] \\
&= \mathbb{E}_{s_{0:t},a_{0:t-1}}\Big[b(s_t)\cdot 0\Big] = 0
\end{aligned}$$

Derivation: https://danieltakeshi.github.io/2017/03/28/going-deeper-into-reinforcement-learning-fundamentals-of-policy-gradients/