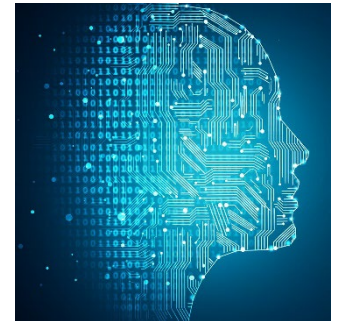


Machine Learning

Reinforcement Learning



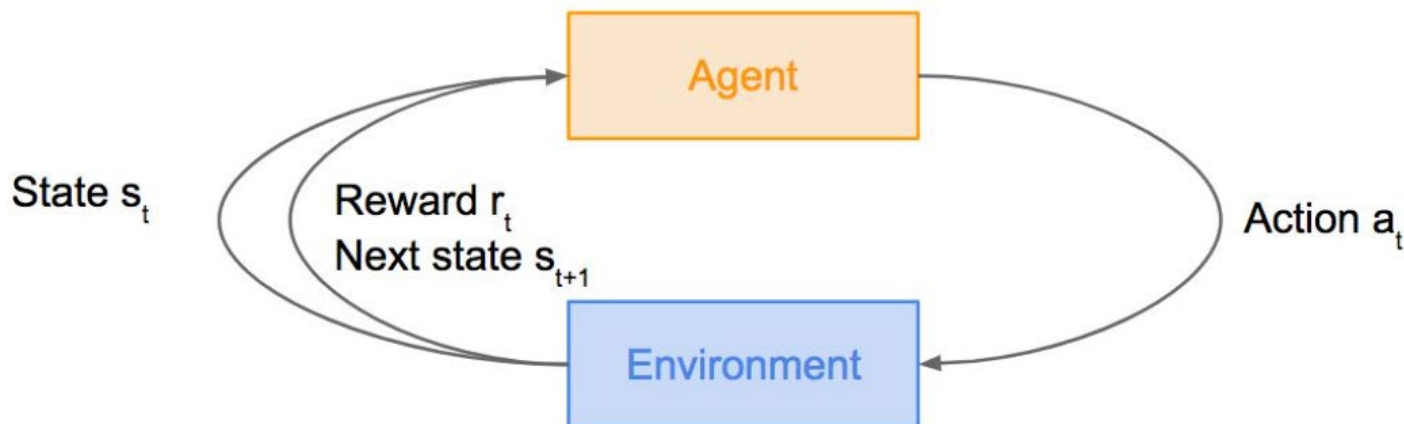
Kevin Moon (kevin.moon@usu.edu)
STAT/CS 5810/6655



Introduction



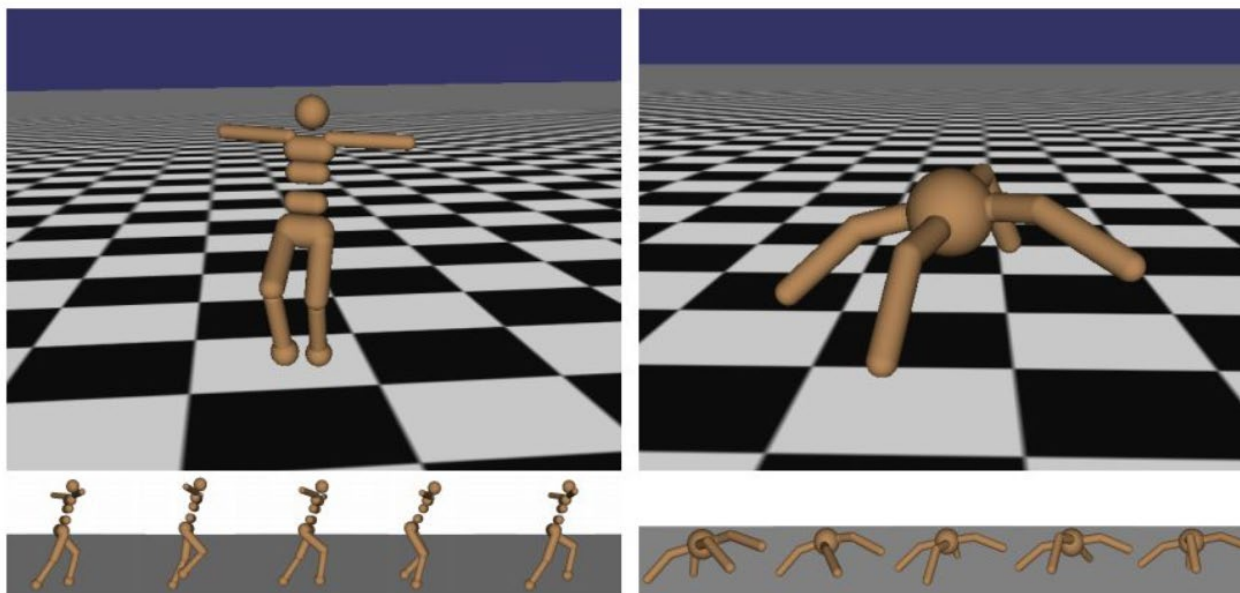
- Informally, reinforcement learning (RL) is about learning how to achieve a goal through interaction with an environment
- More formally, the goal of RL is to use an agent to learn how to map *states* to *actions* in order to maximize some *reward*
- Generally considered to be more similar to how humans learn than supervised or unsupervised learning



Example: Robot Locomotion



- **Objective:** Make the robot move forward
- **State:** Angle and position of the joints
- **Action:** Torques/movements applied by the joints
- **Reward:** +1 for each time step the robot is upright and moving forward



Example: Atari Games



- **Objective:** Get the high score
- **State:** Raw pixel inputs of the game state
- **Action:** Game controls e.g. Left, Right, Up, Down
- **Reward:** Score increases/decreases at each time step



Example: Playing Go or Chess



Alpha Go Zero



International journal of science

Access provided by Yale University

Altmetric: 2188 Citations: 1 [More detail >>](#)

Article

Mastering the game of Go without human knowledge

David Silver , Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel & Demis Hassabis

Nature **550**, 354–359 (19 October 2017)
doi:10.1038/nature24270
[Download Citation](#)

Received: 07 April 2017
Accepted: 13 September 2017
Published online: 18 October 2017

Computational science
Computer science Reward

Alpha Zero

arXiv.org > cs > arXiv:1712.01815 [Search or](#)
([Help](#) | [Advance](#))

Computer Science > Artificial Intelligence

Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dhharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, Demis Hassabis

(Submitted on 5 Dec 2017)

The game of chess is the most widely-studied domain in the history of artificial intelligence. The strongest programs are based on a combination of sophisticated search techniques, domain-specific adaptations, and handcrafted evaluation functions that have been refined by human experts over several decades. In contrast, the AlphaGo Zero program recently achieved superhuman performance in the game of Go, by tabula rasa reinforcement learning from games of self-play. In this paper, we generalise this approach into a single AlphaZero algorithm that can achieve, tabula rasa, superhuman performance in many challenging domains. Starting from random play, and given no domain knowledge except the game rules, AlphaZero achieved within 24 hours a superhuman level of play in the games of chess and shogi (Japanese chess) as well as Go, and convincingly defeated a world-champion program in each case.

Subjects: **Artificial Intelligence (cs.AI)**; Learning (cs.LG)
Cite as: [arXiv:1712.01815 \[cs.AI\]](#)
(or [arXiv:1712.01815v1 \[cs.AI\]](#) for this version)

Submission history

From: David Silver [[view email](#)]
[v1] Tue, 5 Dec 2017 18:45:38 GMT (272kb,D)

Example: Playing Chess



- **State:** s_t = current board configuration

- **Action:** a_t = your next move

- **Reward:**

$$r_t = \begin{cases} 1 & \text{if your move wins the game} \\ -1 & \text{if your move loses the game} \\ 0 & \text{if the game continues or is drawn} \end{cases}$$

- The “environment” is your opponent





- Rewards tell you what to achieve but not how
 - E.g. the chess program is not rewarded for capturing pieces although this is (probably) a good strategy. Why not?
- Output of an RL algorithm is a **policy** π
 - Or a family of policies π_t if the environment changes

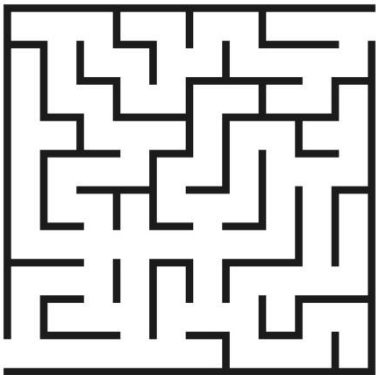
$$\pi_t(s, a) = \Pr(a_t = a | s_t = s)$$

- **Note:** policies may not be deterministic
- What makes a good policy?
 - This is what reinforcement learning is trying to solve

Episodic Tasks



- Episodic task: eventually terminates
- **Examples:** plays of a game, trips through a maze, etc.
- I.e., a task that ends when a special state (called a terminal state) is reached
 - E.g. game over, maze exited, etc.
- If a task lasts for T steps, the **return** from time t to T is



$$R_t = r_{t+1} + r_{t+2} + \cdots + r_T$$

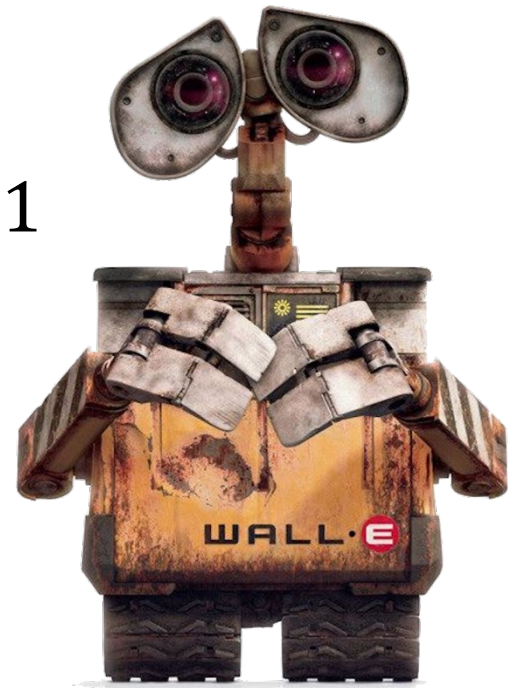
Continuing Tasks



- Continuing task: continues indefinitely
- **Examples:** elevator dispatching, traffic planning, WALL-E
- Define the discounted return:

$$R_t = \sum_{k \geq 0} \gamma^k r_{t+k+1}$$

- $0 < \gamma < 1$ is the discount rate
- An episodic task is a special case with $\gamma = 1$
 - A terminal state is an absorbing state (can't leave)
 - Reward after entering an absorbing state is 0



Value Functions



- In both kinds of tasks, the return is generally random
 - Return depends on the policy and environment
- Value functions give the **expected** return wrt a given policy π
- State value function:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[R_t | s_t = s] \\ &= \mathbb{E}_\pi \left[\sum_{k \geq 0} \gamma^k r_{t+k+1} \mid s_t = s \right] \end{aligned}$$

- Action-state value function:

$$\begin{aligned} Q^\pi(s, a) &= \mathbb{E}_\pi[R_t | s_t = s, a_t = a] \\ &= \mathbb{E}_\pi \left[\sum_{k \geq 0} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a \right] \end{aligned}$$

Value Functions



- Value and reward are different
- Reward
 - Short term desirability of a state
 - Easy to know
 - Provided by environment
- Value
 - Long-term desirability of a state
 - Usually needs to be estimated
- Value function estimation is a central part of most RL
 - Use value functions to evaluate different policies

Multi-armed bandits

A specific kind of RL problem

Multi-armed bandits

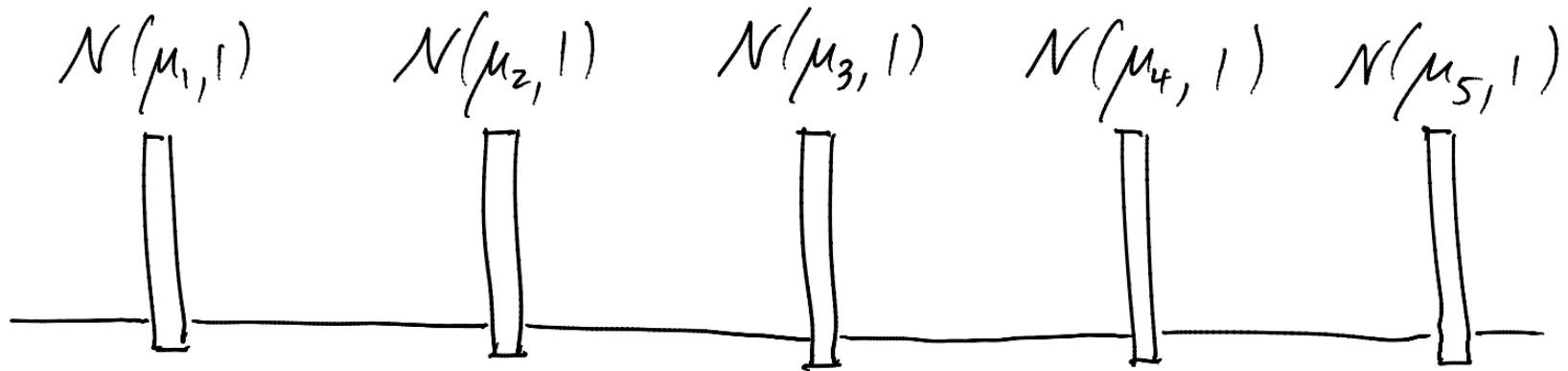


- A game with n levers
- Pull a lever, receive a reward ($n = 1 \Rightarrow$ slot machine)
- Assume the reward for a lever is a random variable
 - Each lever may have a different distribution
 - Called a stochastic multi-armed bandit
- **Example:** $n = 5$ with Gaussian rewards



- Is this an episodic or continuing task?

Multi-armed bandits



- What policy should you adopt to maximize your return over repeated plays of the game?
- If μ_i are known, choose the lever with the highest μ_i
- More interesting when the μ_i are unknown
- Need to do value function estimation

Multi-armed bandits



- $r^*(a)$ = expected reward for lever a
- $\hat{r}(a)$ = estimate of $r^*(a)$
- Denote $\hat{r}_0(a)$ an initial guess for $r^*(a)$
- After playing actions a_1, \dots, a_T we observe rewards r_1, \dots, r_T
- Sample average estimate:

$$\hat{r}(a) = \frac{1}{|I_a|} \sum_{t \in I_a} r_t$$

- $I_a = \{t: a_t = a\}$

Exploration vs. Exploitation



- Policy learning algorithms consider the exploration-exploitation tradeoff
- **Exploration**
 - Try out different actions (arms) to find the best one
 - Gather more information for long term benefit
 - I.e. try to improve the accuracy of $\hat{r}(a)$
- **Exploitation**
 - Take the best action (arm) believed to give the best reward
 - Get the maximum immediate reward given current information
- Real world examples: select a restaurant for dinner, dating, job hunting
- A key issue in RL is striking the right balance

Two policies



1. Greedy:

$$a_t = \arg \max_a \hat{r}(a)$$

2. ϵ -greedy:

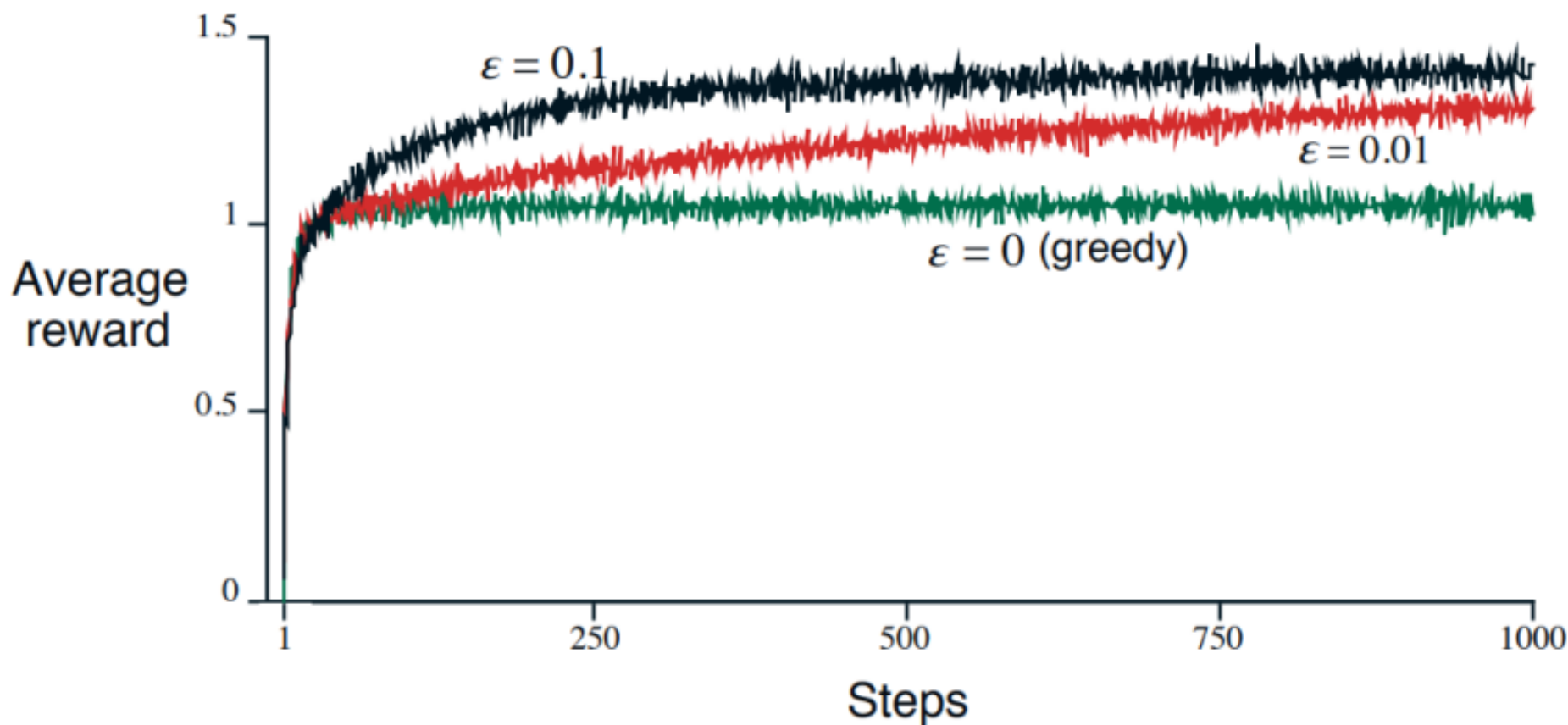
$$a_t = \begin{cases} \arg \max_a \hat{r}(a) & \text{with prob. } 1 - \epsilon \\ \text{random action} & \text{with prob. } \epsilon \end{cases}$$

- Both policies update $\hat{r}(a)$ after each observation
- The ϵ -greedy policy will try each action infinitely often
 - Guaranteed to have an accurate estimate of $r^*(a)$

Example: $n = 10$



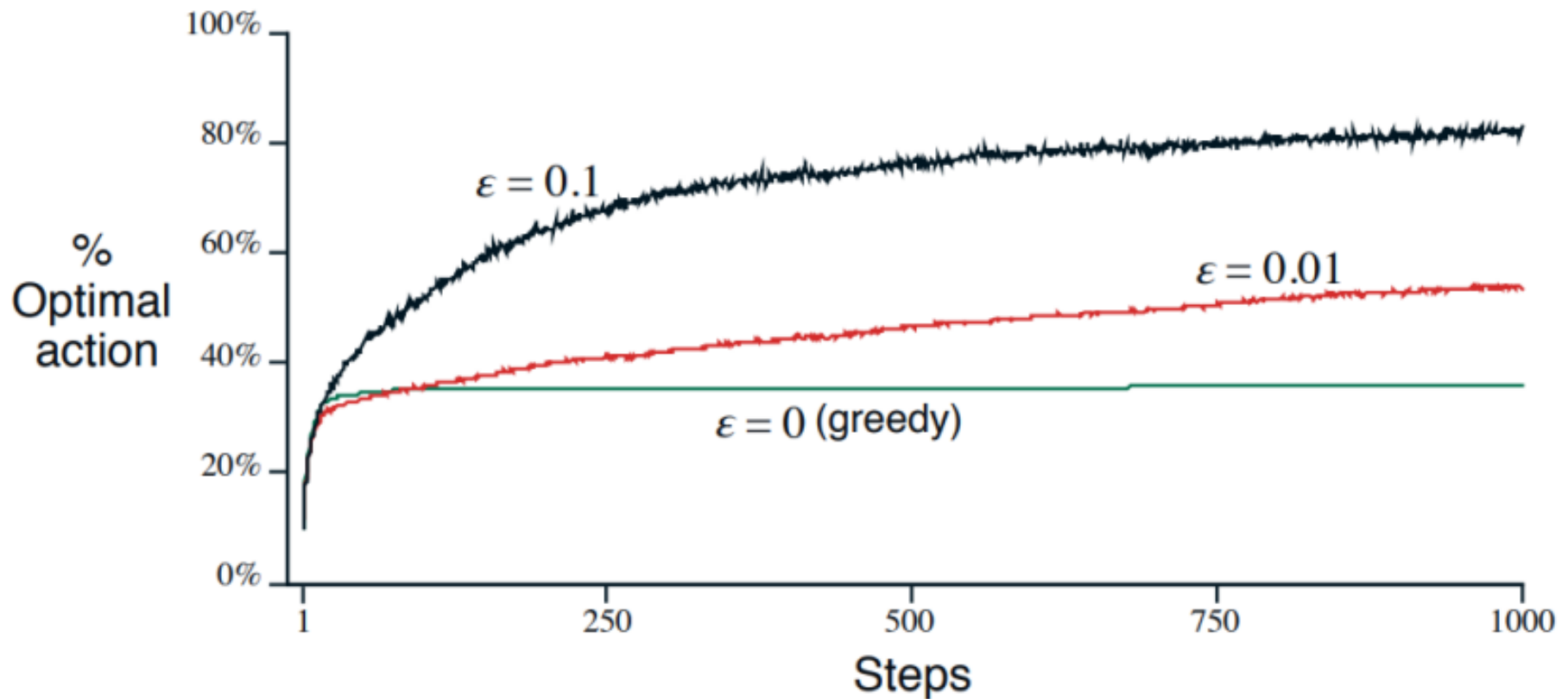
Averaged over 2000 runs/trials



Example: $n = 10$



Averaged over 2000 runs/trials



ϵ -greedy discussion



- Easy to implement and thus popular
- Often ϵ is reduced over time
- Large initial values of $\hat{r}(a)$ are often used to force earlier exploration
- **Weaknesses**
 1. What if rewards drift over time?
 2. Randomly explores and so may spend a lot of time on bad arms

Drifting rewards



- For ϵ -greedy, the reward is updated as

$$\hat{r}(a) \leftarrow \hat{r}(a) + \frac{1}{k+1} [r_{new} - \hat{r}(a)]$$

- r_{new} is the k th reward observed for action a
- For drifting rewards, add a forgetting factor:

$$\hat{r}(a) \leftarrow \hat{r}(a) + \alpha [r_{new} - \hat{r}(a)]$$

- $0 < \alpha < 1$ is fixed
- This implies

$$\hat{r}(a) = (1 - \alpha)^k \hat{r}_0(a) + \sum_{i=1}^k \alpha (1 - \alpha)^{k-i} r_{a,i}$$

- $r_{a,i}$ = reward from the i th selection of a

Softmax action selection



- Modify the probability based on reward estimate
- Select a_t with probability

$$\frac{\exp\left(\frac{\hat{r}(a)}{\tau}\right)}{\sum_b \exp\left(\frac{\hat{r}(b)}{\tau}\right)}$$

- τ = temperature parameter
 - $\tau \rightarrow \infty \Rightarrow$ random
 - $\tau \rightarrow 0 \Rightarrow$ greedy
- Explores the action space but wastes less time on really bad actions

Upper Confidence Bound (UCB) Alg.



1. Initialize reward estimates $\hat{r}(a) = 0$
2. Initialize $t_a = 0 = \#$ of times arm a is selected up to time t
3. **For** $t = 1, \dots, T$ **do**
 1. $a_t = \arg \max_a \hat{r}(a) + C \sqrt{\frac{2 \log t}{t_a}}$
 2. Obtain the reward r_t
 3. $t_a \leftarrow t_a + 1$
 4. Update reward estimate $\hat{r}(a)$
4. **End for**

Upper Confidence Bound (UCB) Alg.

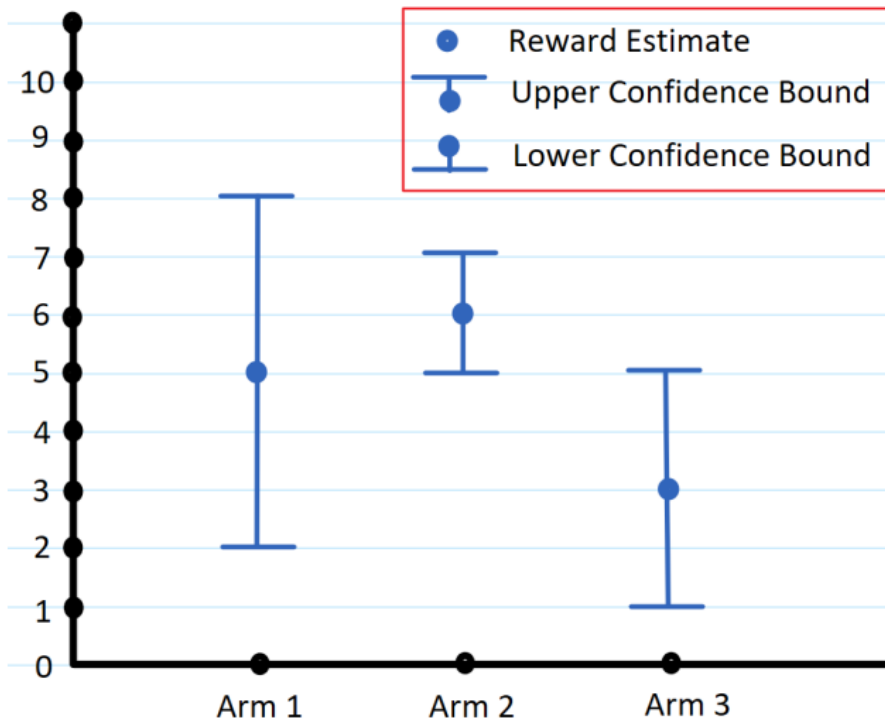


Figure: Case 1: Upper Confidence Bound

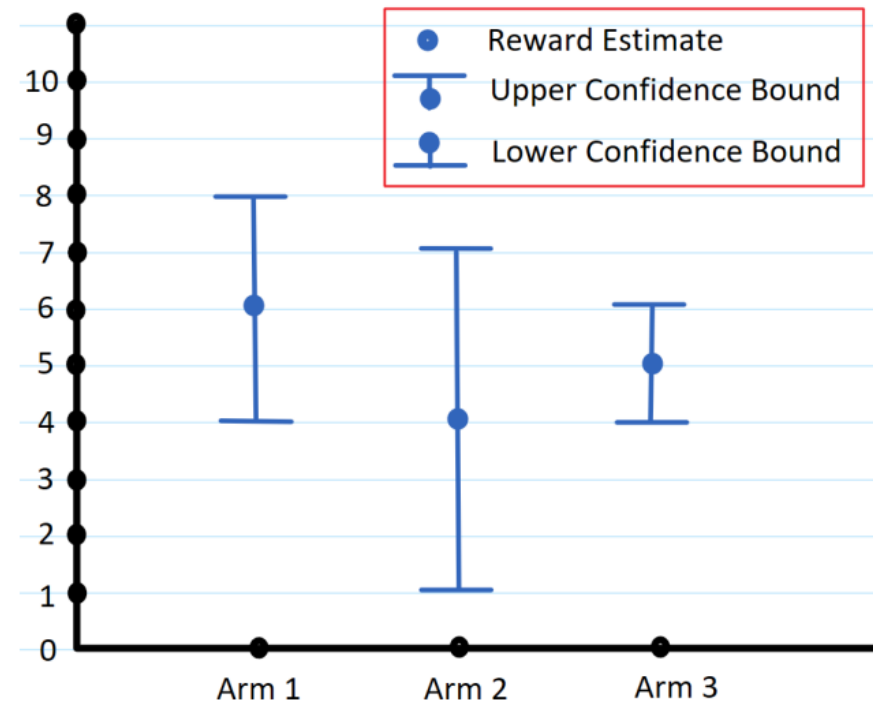


Figure: Case 2: Upper Confidence Bound

Upper Confidence Bound (UCB) Alg.

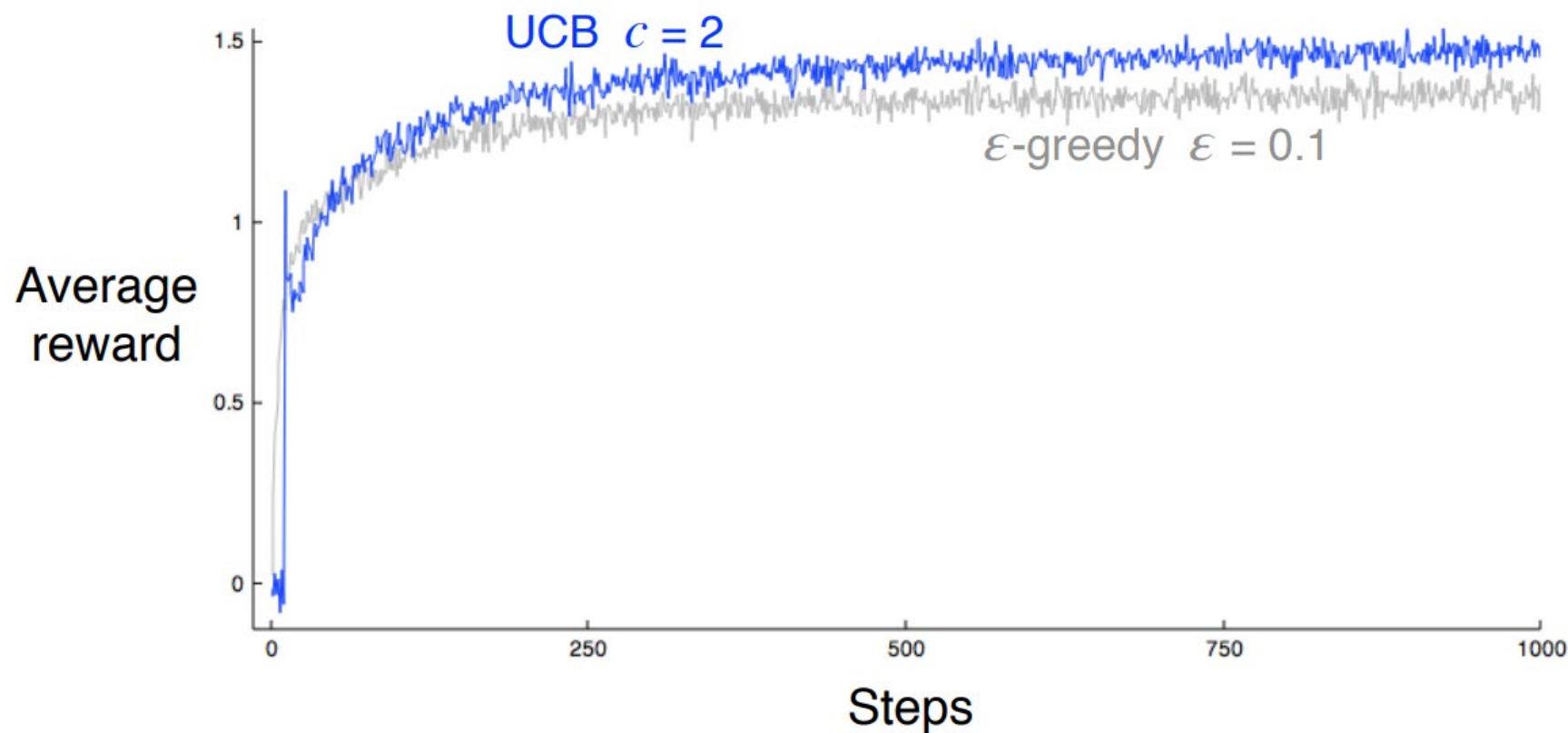


- $\sqrt{\frac{2 \log t}{t_a}}$ is a confidence interval for the estimated reward $\hat{r}(a)$
- Small t_a means a large interval
 - Uncertain about our estimate
- Select an arm that has a maximum upper confidence bound
- Explores arms that are most uncertain while exploiting arms with high average rewards

Example: $n = 10$



Averaged over 2000 runs/trials



Challenges for the UCB algorithm



- Nonstationary data
 - A more complex approach is needed if the distributions are changing over time
- Large state spaces are also hard for the UCB algorithm
 - Other approaches are needed

Other multi-arm bandit problems



- Adversarial multi-arm bandits
 - Rewards for each arm are selected independent of the past
 - Strategies try to find policies that perform well regardless of the randomness
 - Applications: rigged casino, non-stationary distributions, routing
- Contextual bandits
 - Extra (side) information is included
 - Applications: personalized news article recommendation, clinical trials
 - UCB algorithms can be adapted to include the context

Markov Decision Processes

MDP



- A Markov decision process satisfies

$$\begin{aligned} &\Pr(s_{t+1} = s', r_{t+1} = r | s_t, a_t, \dots, s_0, a_0) \\ &= \Pr(s_{t+1} = s', r_{t+1} = r | s_t, a_t) \end{aligned}$$

- I.e. the state and reward at time $t + 1$ depend only on the state and action at time t
 - Don't depend on the distant past
- Many RL problems can be (at least approximately) viewed as MDPs with an appropriate state space




A simple MDP (Gridworld)




actions = {

1. right 

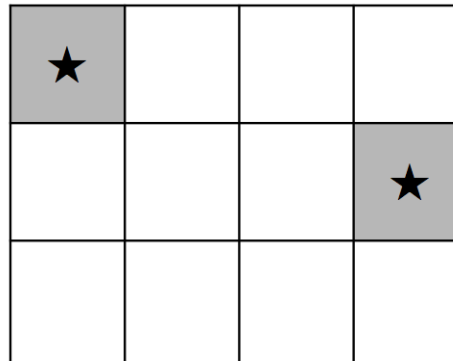
2. left 

3. up 

4. down 

}

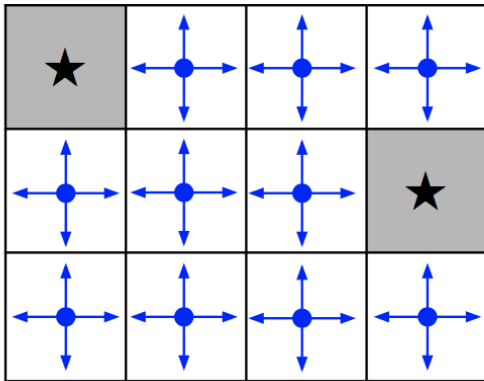
states



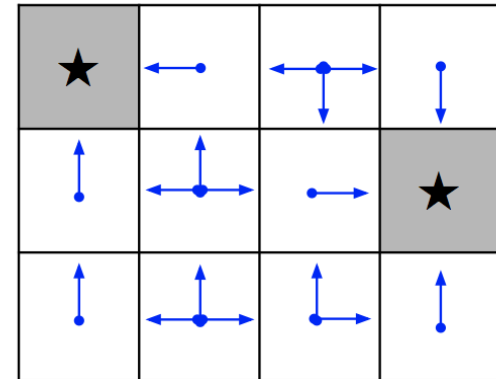
Set a negative “reward”
for each transition
(e.g. $r = -1$)

Objective: Reach one of terminal states (greyed out) in least number of actions

A simple MDP (Gridworld)



Random Policy



Optimal Policy

Mathematical Formalization



A *Markov decision process* (MDP) is a tuple $(S, A, P_{sa}, \gamma, r)$ where

- S is a set of **states**. (For example, in autonomous helicopter flight, S might be the set of all possible positions and orientations of the helicopter)
- A is a set of **actions**. (For example, the set of all possible directions in which you can push the helicopter's control sticks.)
- P_{sa} are the **state transition probabilities**. For each state $s \in S$ and action $a \in A$, P_{sa} is a probability distribution over the state space and gives the distribution over what states we will transition to if we take action a in state s .
- $\gamma \in [0,1)$ is the **discount factor**.
- $r: S \times A \rightarrow \mathbb{R}$ is the **reward function**. Rewards are sometimes also written as a function of a state s only.

MDP Dynamics



- Given initial state $s_0 \in S$, choose an action $a_0 \in A$
- Given the action a_0 , the next state s_1 is drawn randomly according to $s_1 \sim P_{s_0 a_0}$
- Given s_1 , choose an action a_1 . The next state s_2 is drawn randomly according to $s_2 \sim P_{s_1 a_1}$
- And so on...
- Can represent this pictorially as:

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \dots$$

- Total payoff is

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \dots$$

- When writing reward as a function of the states only,

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots$$

Maximizing expected reward



- Goal in RL: choose actions to maximize the expected total reward

$$\mathbb{E}\left[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots\right]$$

- Note that the reward at timestep t is discounted by a factor of γ^t
- A *policy* is any function $\pi: S \rightarrow A$ mapping from states to actions
- We are *executing* some policy π if whenever we are in state s , we take action $a = \pi(s)$

The value function



- Assume from now on a finite MDP: $|S| < \infty, |A| < \infty$
- The *value function* for policy π :

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi \left[\sum_{k \geq 0} \gamma^k r_{t+k+1} \middle| s_t = s \right] \\ &= \mathbb{E}_\pi [R_t | s_t = s] \end{aligned}$$

The value function



- $V^\pi(s)$ is the expected cumulative reward from following policy π starting at state s
- The *Q-value function* at state s and action a is the expected cumulative reward from taking initial action a in initial state s and then following policy π :

$$Q^\pi(s, a) = \mathbb{E}_\pi[R_t | s_t = s, a_t = a]$$

The Bellman equations



$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[R_t | s_t = s] \\ &= \sum_a \pi(s, a) \mathbb{E}_\pi[R_t | s_t = s, a_t = a] \\ &= \sum_a \pi(s, a) Q^\pi(s, a) \\ &= \sum_a \pi(s, a) \sum_{s'} P_{sa}(s') \mathbb{E}_\pi[R_t | s_t = s, a_t = a, s_{t+1} = s'] \\ &= \sum_a \pi(s, a) \sum_{s'} P_{sa}(s') \mathbb{E}_\pi[r_{t+1} + \gamma R_{t+1} | s_t = s, a_t = a, s_{t+1} = s'] \\ &= \sum_{a, s'} \pi(s, a) P_{sa}(s') [R_{sa}(s') + \gamma V^\pi(s')] \end{aligned}$$

Where $R_{sa}(s') = \mathbb{E}[r_{t+1} | s_t = s, a_t = a, s_{t+1} = s']$

The Bellman equations



$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[R_t | s_t = s] \\ &= \sum_{a,s'} \pi(s, a) P_{sa}(s') [R_{sa}(s') + \gamma V^\pi(s')] \end{aligned}$$

- This is a linear system of equations for V^π
- Given any policy π and knowledge of the dynamics (P_{sa} and R_{sa}), we can determine V^π by solving a linear system of equations
 - The solution is also unique

The Bellman equations



A similar result holds for $Q^\pi(s, a)$:

$$\begin{aligned} Q^\pi(s, a) &= \mathbb{E}_\pi[R_t | s_t = s, a_t = a] \\ &= \sum_{s'} P_{sa}(s') \mathbb{E}_\pi[r_{t+1} + \gamma R_{t+1} | s_t = s, a_t = a, s_{t+1} = s'] \\ &= \sum_{s'} P_{sa}(s') [R_{sa}(s') + \gamma V^\pi(s')] \\ &= \sum_{s'} P_{sa}(s') \left[R_{sa}(s') + \gamma \sum_a \pi(s', a) Q(s', a) \right] \end{aligned}$$

- **Problem:** the state space S is too large for many problems to solve the Bellman equations directly
- But these equations still form the basis for efficient algorithms

Bellman Optimality Equations



- We say $\pi \geq \pi'$ iff $V^\pi(s) \geq V^{\pi'}(s) \forall s$
- We say π^* is optimal iff $\pi^* \geq \pi \forall \pi$
- It can be shown that there is always an optimal policy
 - May not be unique
- All optimal policies have the same optimal state value function and optimal state-action value function

$$V^*(s) = \max_{\pi} V^\pi(s)$$
$$Q^*(s, a) = \max_{\pi} Q^\pi(s, a)$$

Bellman Optimality Equations



- These optimal functions satisfy the Bellman optimality equations:

$$\begin{aligned} V^*(s) &= V^{\pi^*}(s) \\ &= \max_a Q^{\pi^*}(s, a) \\ &= \max_a \sum_{s'} P_{sa}(s') [R_{sa}(s') + \gamma V^{\pi^*}(s')] \\ &= \max_a \sum_{s'} P_{sa}(s') [R_{sa}(s') + \gamma V^*(s')] \end{aligned}$$

Similarly,

$$\begin{aligned} Q^*(s, a) &= \sum_{s'} P_{sa}(s') [R_{sa}(s') + \gamma V^*(s')] \\ &= \sum_{s'} P_{sa}(s') \left[R_{sa}(s') + \gamma \max_{a'} Q^*(s', a') \right] \end{aligned}$$

Bellman Optimality Equations



$$V^*(s) = \max_a \sum_{s'} P_{sa}(s') [R_{sa}(s') + \gamma V^*(s')]$$
$$Q^*(s, a) = \sum_{s'} P_{sa}(s') \left[R_{sa}(s') + \gamma \max_{a'} Q^*(s', a') \right]$$

- These equations are nonlinear and do not involve π^*
- If V^* or Q^* are known, an optimal policy can be found:
$$\pi^*(s) = \arg \max_a Q^*(s, a)$$
$$= \arg \max_a \sum_{s'} P_{sa}(s') [R_{sa}(s') + \gamma V^*(s')]$$
- Many RL algorithms can be understood as (approximately) solving the Bellman optimality equations

Optimal Planning for MDPs

Overview



- Assume the MDP dynamics (P_{sa}, R_{sa}) are known
- Can apply the Bellman optimality equations iteratively to compute optimal value functions
 - Use these to obtain an optimal policy
 - These iterative algorithms are examples of **dynamic programming**
- We'll look at ways for determining the value of a given policy and for improving it

Policy Evaluation



- (Relatively) fast way of determining the value of a policy
- Let π be an arbitrary policy
- Define a sequence of functions:

$$V_{k+1}(s) = \sum_a \pi(s, a) \sum_{s'} P_{sa}(s') [R_{sa}(s') + \gamma V_k(s')]$$

- Can show that $\{V_k\}$ converges to V^π
- Much more efficient than solving the Bellman equations directly when $|S|$ is large
- V_0 can be arbitrary except for episodic tasks we require $V_0(\text{terminal state}) = 0$

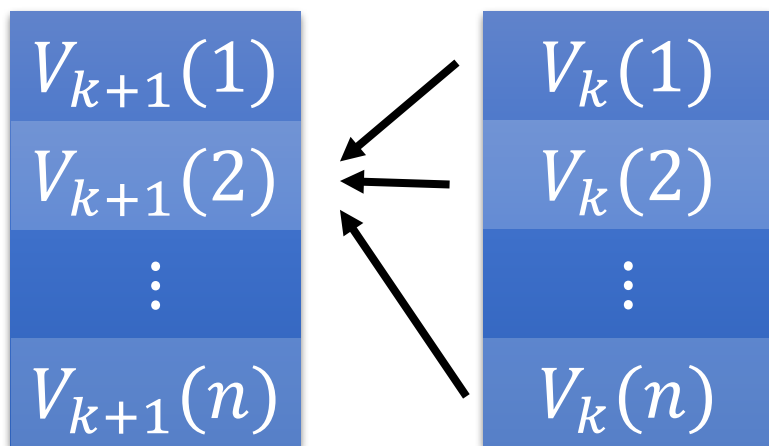
Policy Evaluation



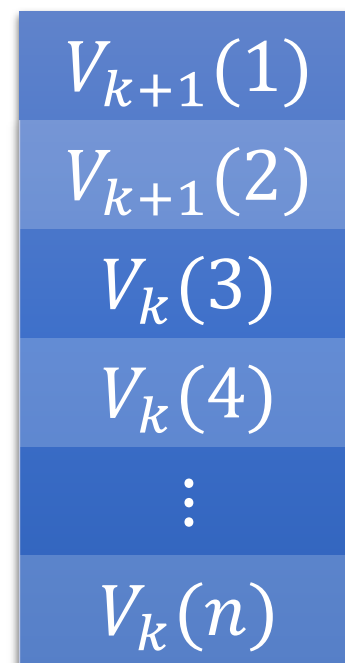
$$V_{k+1}(s) = \sum_a \pi(s, a) \sum_{s'} P_{sa}(s') [R_{sa}(s') + \gamma V_k(s')]$$

- Can use either one or two arrays for implementation
 - Convergence guaranteed either way

Sweep through the states



In place updates



Policy Improvement



- If we can compute V^π , we can compute

$$Q^\pi(s, a) = \sum_{s'} P_{sa}(s') [R_{sa}(s') + \gamma V^\pi(s')]$$

- If $Q^\pi(s, a) > V^\pi(s)$ for some a , then π is suboptimal
 - Improve it by selecting action a when in state s

- I.e.:

$$\pi' = \arg \max_a Q^\pi(s, a)$$

- **Policy Improvement Theorem:** $\pi' \geq \pi$. I.e., $V^{\pi'}(s) \geq V^\pi(s) \forall s$. Furthermore, if π is not optimal, then $\exists s$ s.t. $V^{\pi'}(s) > V^\pi(s)$.

Policy Improvement Theorem



Proof:

$$\begin{aligned} V^\pi(s) &\leq Q^\pi(s, \pi'(s)) \\ &= \mathbb{E}_{\pi'}[r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s] \\ &\leq \mathbb{E}_{\pi'}[r_{t+1} + \gamma Q^\pi(s_{t+1}, \pi'(s_{t+1})) | s_t = s] \\ &= \mathbb{E}_{\pi'}[r_{t+1} + \gamma \mathbb{E}_{\pi'}[r_{t+2} + \gamma V^\pi(s_{t+2})] | s_t = s] \\ &= \mathbb{E}_{\pi'}[r_{t+1} + \gamma r_{t+2} + \gamma^2 V^\pi(s_{t+2}) | s_t = s] \\ &\leq \mathbb{E}_{\pi'}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots | s_t = s] \\ &= V^{\pi'}(s) \end{aligned}$$

Policy Improvement Theorem



Proof (continued): Prove the contrapositive. Suppose $V^\pi(s) = V^{\pi'}(s) \forall s$. This implies $Q^\pi(s, a) = Q^{\pi'}(s, a) \forall s, a$. Then $\forall s$

$$\begin{aligned} V^\pi(s) &= V^{\pi'}(s) \\ &= Q^{\pi'}(s, \pi'(s)) \\ &= Q^\pi(s, \pi(s)) \\ &= \max_a Q^\pi(s, a) \\ &= \max_a \sum_{s'} P_{sa}(s') [R_{sa}(s') + \gamma V^\pi(s')] \end{aligned}$$

Thus V^π satisfies the Bellman optimality equations, implying π is optimal.

Policy Iteration



- Apply policy evaluation and improvement iteratively

$$\pi_0 \xrightarrow{E} V^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V^{\pi_1} \xrightarrow{I} \dots$$

- Remember that each evaluation step is also iterative
- For finite MDPs, policy iteration will converge to π^* in a finite, and often small, number of steps

Example: Car rental service

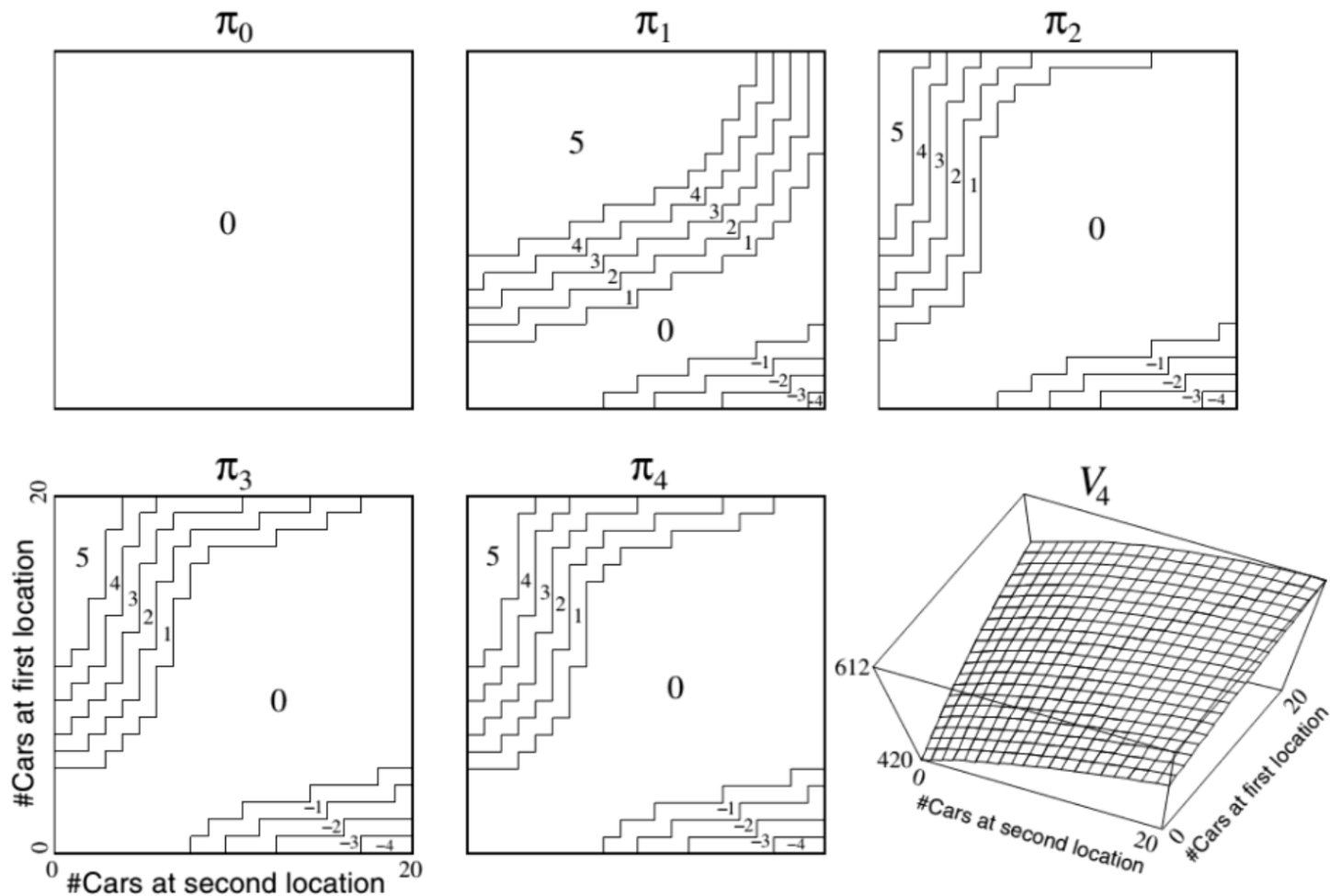


- Each day customers arrive at two locations to rent cars
- If a car is available, we rent it out and make \$10
 - Cars become available the day after they are returned
- If no cars available, the business is lost
- We can move cars between locations overnight for \$2 per car moved
- Assume the # of cars requested and returned are Poisson
- Assume $\lambda = 3,4$ for rental requests and $\lambda = 3,2$ for returns at the two locations, respectively
- Other assumptions: no more than 20 cars per location, maximum 5 cars can be moved per night
- Continuing finite MDP where state is # of cars at each location and actions are the # of cars moved between locations

Example: Car rental service



- Last policy is optimal



Value Iteration



- Policy iteration except with only a single policy evaluation iteration at each E step
- Obtain this sequence:

$$V_{k+1}(s) = \max_a \sum_{s'} P_{sa}(s') [R_{sa}(s') + \gamma V_k(s')]$$

- Basically, we use the Bellman optimality equation as an update rule
- Convergence to the optimal value function and policy is guaranteed
 - More generally, also guaranteed for any finite # of iterations at the policy evaluation step
- Somewhat analogous to stochastic gradient descent

Asynchronous Updates



- So far, we've assumed that every iteration of policy evaluation or improvement does a full sweep through the state space
- This is infeasible for large state space
- In asynchronous dynamic programming, some states are updated more often than others
- Convergence theorems can still be established if each state is updated infinitely often
- Can still improve our policy by focusing more on the "important states"

Learning Policies from Experience

Generalized policy iteration



- Now assume the MDP parameters are unknown
- Need to learn about the MDP through experience
- General framework: generalized policy iteration (GPI)

Monte Carlo (MC) Methods



- Let π be a policy for an episodic task
 - Suppose we can easily perform the task repeatedly
- MC estimate of π 's Q function is $\hat{Q}(s, a)$ = average of all returns following the first occurrence of (s, a) in an episode
- Suppose an episode can be initialized to an arbitrary (s, a)
- We can use the following GPI strategy (next slide)

Monte Carlo (MC) Methods



- Initialize $\forall s, a$ values for $Q(s, a)$, $\pi(s)$, (arbitrary initialization) and $Returns(s, a)$ (empty list)
- Repeat:
 1. Generate an episode at (s, a) chosen randomly s.t. every pair occurs with nonzero probability
 2. For each pair (s, a) appearing in the episode
 1. $R \leftarrow$ return following the first occurrence of (s, a)
 2. Append R to $Returns(s, a)$
 3. $Q(s, a) \leftarrow avg(Returns(s, a))$
 3. For each s in the episode
 1. $\pi(s) \leftarrow \arg \max_a Q(s, a)$

Example: Blackjack



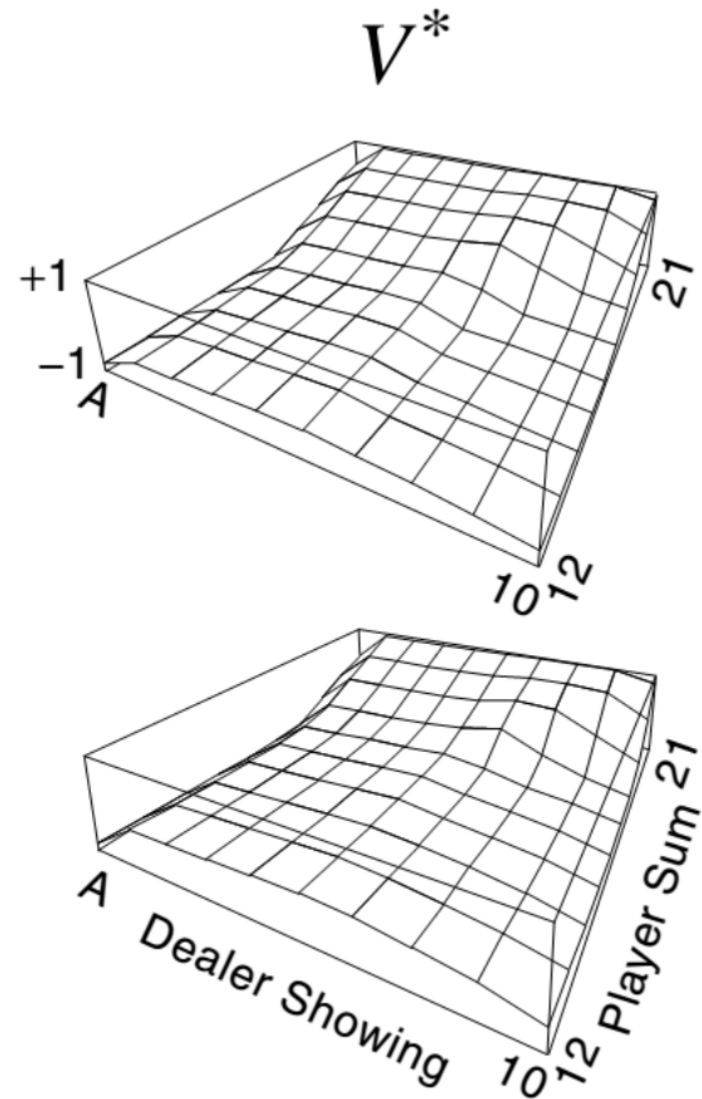
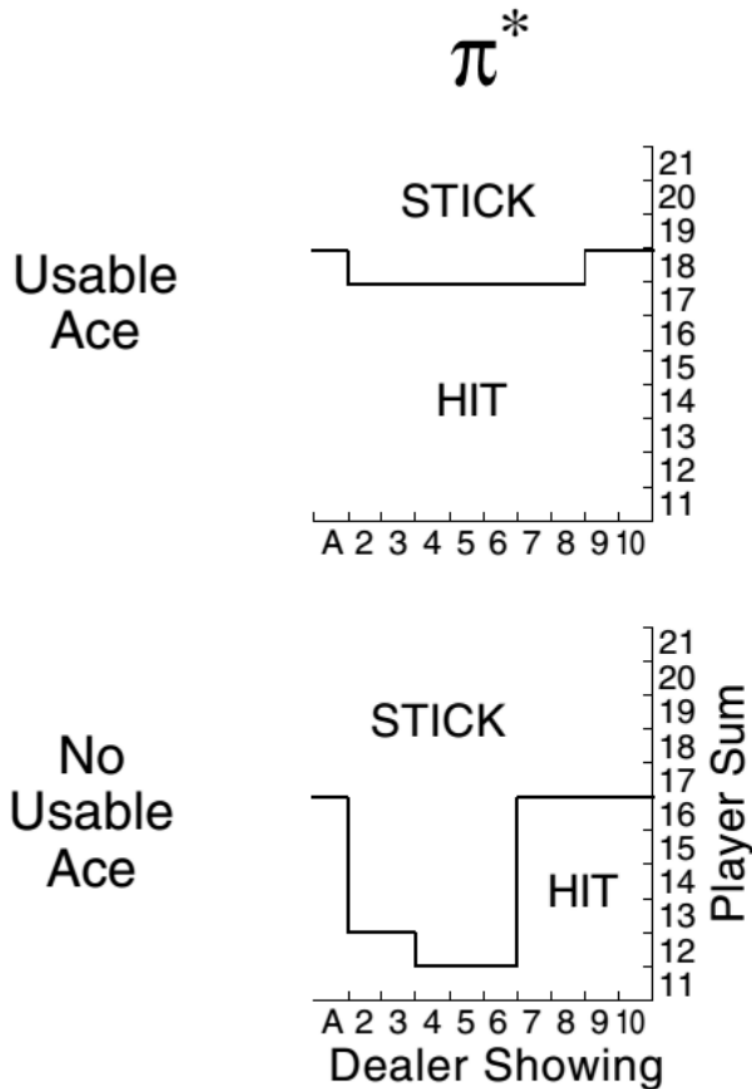
- Game played against the dealer with face cards
- **Goal:** obtain cards with largest sum without exceeding 21
- Face cards (jack, queen, king) count as 10 and aces are 1 or 11
- 2 cards dealt to each player and dealer
- Dealer has one card faceup and one card facedown
- Player can request additional cards one by one (hits) until he stops (sticks) or exceeds 21 (goes bust).
- If he sticks, dealer can hit or stick with a fixed strategy
 - Sticks on any sum of 17 or greater, hits otherwise
- Player wins if he is closest to 21 and didn't go bust or if the dealer goes bust

Example: Blackjack



- **State:** player's cards, dealer's visible card, and whether the player has a "usable ace"
- **Action:** hit or stick
- **Reward:**
$$\begin{cases} 1 & \text{win} \\ -1 & \text{lose} \\ 0 & \text{draw} \end{cases}$$
- Apply the MC strategy described before:

Example: Blackjack





- This MC strategy is sometimes called MC with exploring starts
- May not be possible to initialize a task arbitrarily
 - May need to incorporate exploration, e.g. via ϵ -greedy policy
- MC methods were at the heart of state of the art programs for playing Go (Alpha Go)

Temporal Difference (TD) Methods



- Update before the end of a task
 - Suitable for continuing tasks

- Update form:

$$V(s_t) \leftarrow V(s_t) + \alpha[TARGET - V(s_t)]$$

- Monte Carlo methods have this form where $\alpha = \frac{1}{k+1}$ and TARGET is the $(k + 1)$ st return observed after a first occurrence of state s_t

Temporal Difference (TD) Methods



- Consider:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[R_t | s_t = s] \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s] \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s] \end{aligned}$$

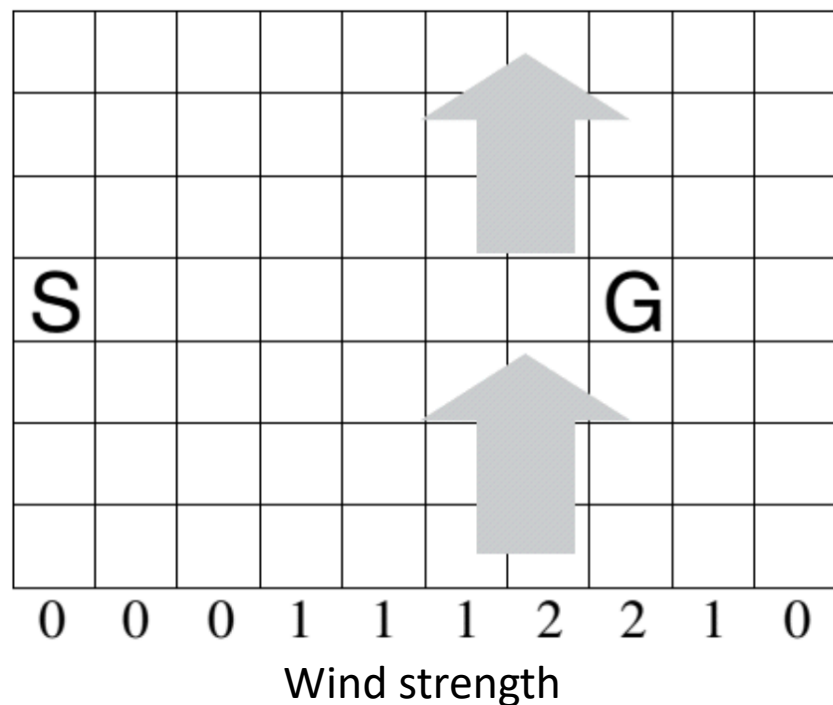
- TD(0) update uses TARGET = $r_{t+1} + \gamma V(s_{t+1})$
- Value function update is:
$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$
- This converges to V^π when π is fixed
- Can be combined with policy improvement to learn a policy



Involves s, a, r, s', a'

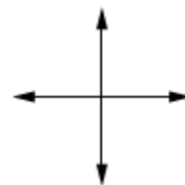
1. Initialize $Q(s, a)$ arbitrarily
2. Repeat (for each episode)
 1. Initialize s
 2. Choose a from s using policy derived from Q (e.g. ϵ -greedy)
 3. Repeat (for each step of episode)
 1. Take action a , observe r, s'
 2. Choose a' from s' using policy derived from Q
 3. $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
 4. $s \leftarrow s'; a \leftarrow a'$
 4. Until s is terminal

Example: Windy Gridworld



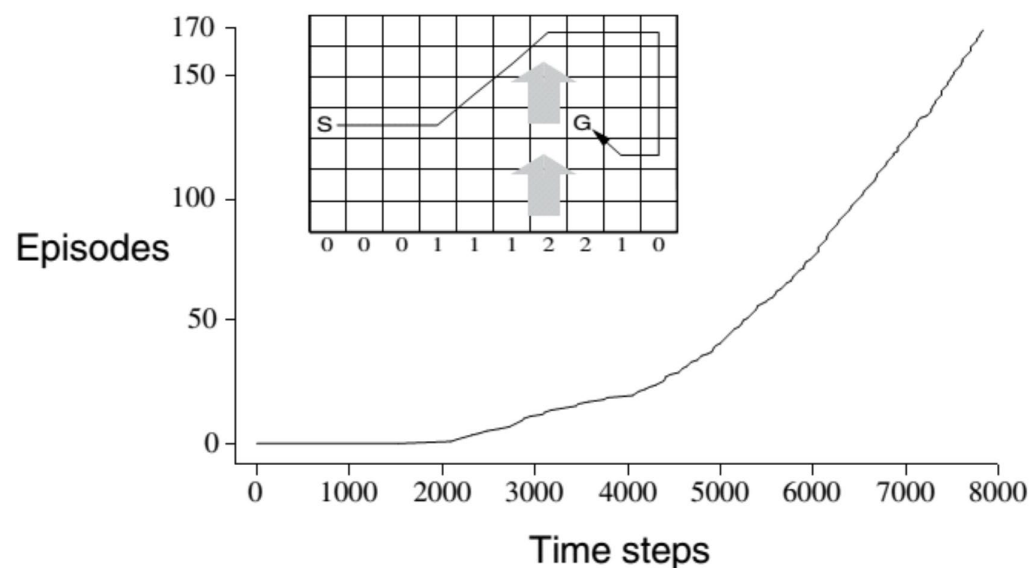
- Step-by-step methods (e.g. Sarsa) learn during the episode
 - Don't get stuck

- MC methods may fail



standard
moves

- Can get stuck in same state and not terminate



Q-learning



- Sarsa is an **on-policy** algorithm
 - The policy that generates the behavior is also evaluated and improved
- If not the case, it is an **off-policy** algorithm
- Q-learning is the most well-known off-policy algorithm

Q-learning



1. Initialize $Q(s, a)$ arbitrarily
2. Repeat (for each episode)
 1. Initialize s
 2. Repeat (for each step of the episode)
 1. Choose a from s using policy derived from Q (e.g. ϵ -greedy)
 2. Take action a , observe r, s'
 3. $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
 4. $s \leftarrow s'$
 3. Until s is terminal



Directly approximates the optimal Q^*

Q-learning



- Q-learning explores with one policy (e.g. ϵ -greedy) using current estimate of Q
- Converges to the optimal Q^* under certain assumptions
 - Each state-action pair visited infinitely often
 - $\alpha = \alpha_t \rightarrow 0$ at a certain rate
- Q-learning is computationally cheap but only propagates experience one step
 - Replay can help with this (store all data discovered and use all data for learning Q)
- Can try to parametrize Q :
$$Q(s, a; \theta) \approx Q^*(s, a)$$
 - θ contains the function parameters (e.g. weights)
- If the function approximator is a deep neural network, this gives **deep Q-learning**

Other TD variations



- A generalization of $TD(0)$ is $TD(\lambda)$
 - Sort of a compromise between $TD(0)$ and MC that typically outperforms both
- There are on-policy and off-policy learning algorithms for $TD(\lambda)$, similar to $TD(0)$

Further reading



- Deep RL course at Berkeley:
<http://rail.eecs.berkeley.edu/deeprlcourse/>
- Reinforcement Learning by Rich Sutton
 - Solutions to problems:
<https://github.com/dennybritz/reinforcement-learning>