LECTURE 1

Reinforcement Learning

Introduction

How do we learn?

- Connection to the environment
- Consequences of actions
- What to do in order to achieve goals
- How our environment responds to what we do
- Influence what happens through our behavior

What we are going to learn!

- Reinforcement Learning: goal-directed learning from interaction
  - Computational approach to learning from interaction
  - Designs for machines that are effective in solving learning problems of scientific or economic interest
  - Evaluate the designs through mathematical analysis or computational experiments

Reinforcement Learning

- Learning what to do
- How to map situations to actions → maximize a numerical reward signal
- Learner discovers which actions yield the most reward by trying them
- Actions may affect not only the intermediate reward but also the next situation, i.e., all subsequent rewards
- RL unique characteristics:
  - Trial and error search
  - Delayed reward

Components of Reinforcement Learning

1. Problem
2. A class of solution methods that work well on the problem
3. The field that studies this problem and its solution

Formalize the problem of RL

- Use of Dynamical Systems Theory
- Optimal control of incompletely-known Markov decision processes
- Learning Agent:
  - Sense the state of its environment
  - Take actions that affect the state
  - Have goal(s) relating to the state of the environment

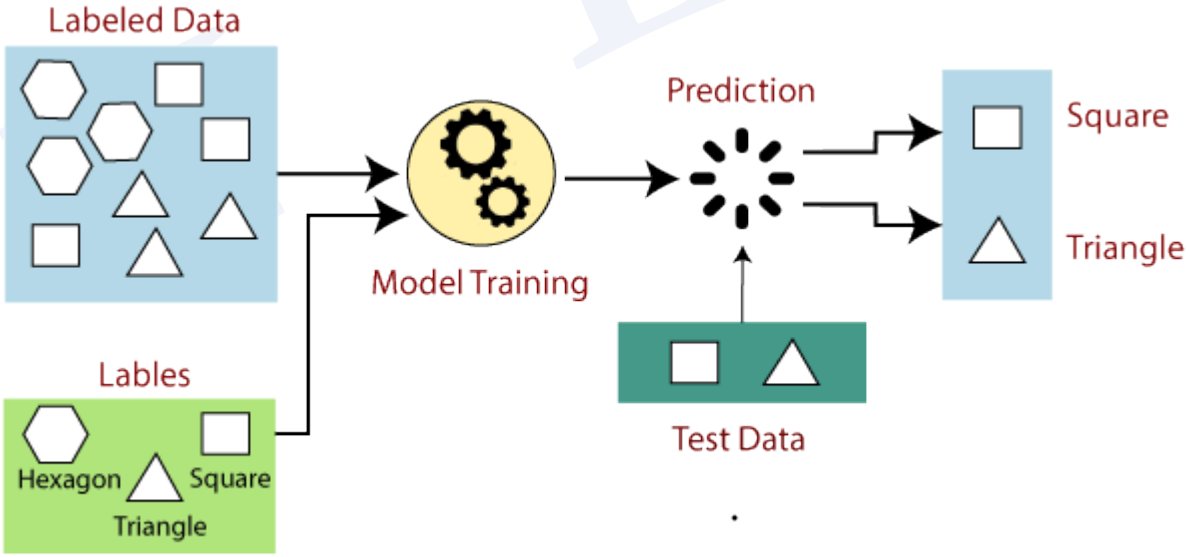
- Markov decision processes
  - Sensation
  - Action
  - Goal

Categories of Machine Learning

- Supervised Machine Learning
- Unsupervised Machine Learning
- Reinforcement Learning

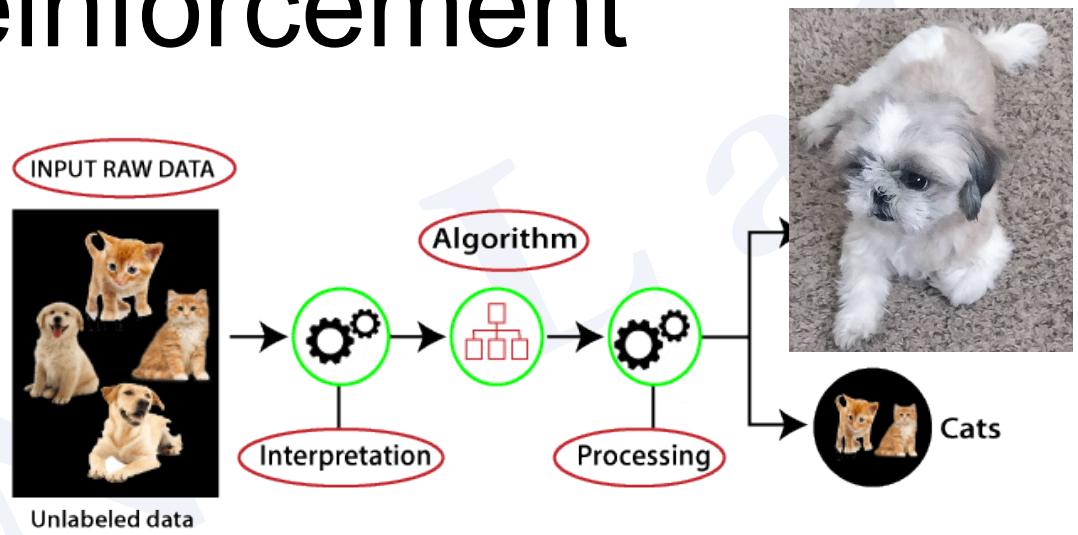
Supervised ML vs Reinforcement Learning

- Supervised ML
  - Learn from a training set of labeled examples from an external supervisor
  - Each example is a description of a situation together with a specification (label)
  - Identify a category to which the situation belongs to
  - Generalize the responses so that it acts correctly in situations not present in the training set
- Drawbacks:
  - Not adequate for learning from interaction
  - Impractical to obtain examples of all the situations
  - In uncharted territory, an agent must learn from its **own** experience



Unsupervised ML vs Reinforcement Learning

- Unsupervised ML
  - Find structure hidden in collections of unlabeled data
- Differences to RL
  - RL is **not** an unsupervised ML model
  - RL aims at maximizing a reward signal instead of trying to find hidden structure



Exploration & Exploitation

- Trade-off between exploration and exploitation – *top 10 problem!*
- RL agent prefers actions that has tried and resulted in high reward → ***Exploits*** what it has already experienced
- RL agent tries actions that has not selected before to discover better rewarding actions → ***Explores*** to make better action selections in the future
- Trial and error search: progressively favor actions that appear to be the best
- Stochastic task/environment: action must be tried many times to gain a reliable estimate of its expected reward

Features of Reinforcement Learning

- Considers the whole problem and **not** isolated subproblems
- Goal-directed agent interacting with an uncertain environment
- Real-time decision-making
- Complete, interactive, goal-seeking agent
  - Explicit goals
  - Senses aspects of the environment
  - Chooses actions to influence the environment
  - Can be part of a larger system and indirectly interacting with its environment

RL in Engineering and Scientific Disciplines

- Core component of Artificial Intelligence (AI)
  - Learning, search, and decision-making
- Statistics and Optimization
- Operations Research (OR)
- Control Theory
- Psychology
- Neuroscience

…originally inspired by biological learning systems

Real-life Examples

Master chess player:

- Makes a move
- Anticipates possible replies
- Immediate, intuitive judgement of the desirability of particular positions and moves

Petroleum Refineries Operation

- Adaptive controller adjusts parameters of a petroleum refinery's operation in real time
- Controller (RL agent) optimizes the cost and quality trade-off accounting for the specified marginal costs

Gazelle Calf

- Struggles to its feet minutes after being born
- Half an hour later it is running at 20 miles per hour!

Mobile Robot

- Decides to enter a new room in search of more trash to collect
- OR Try to find its way back to the recharging station
- Real-time decision-making based on current charge level && experience to find the recharger in the past

Lessons *learned*…

- Interaction between an active decision-making agent and its environment
- Agent seeks to achieve a goal despite the environment's uncertainty
- Agent's actions may affect the future state of the environment, and actions and opportunities available to the agent in the future
- Planning may be needed as correct choices should account for indirect, delayed consequences of actions
- Effects of the actions cannot be fully predicted
- Agent must monitor the environment frequently and react appropriately
- Agent uses its experience to improve its performance over time

Elements of Reinforcement Learning

- Main elements:
  - Agent
  - Environment
- Subelements:
  - Policy
  - Reward signal
  - Value function
  - Model of the environment

Policy

- Defines the learning agent's way of behaving at a given time
- Mapping from perceived states of the environment to actions to be taken when in those states
- Simple function, lookup table, or involve extensive computation, e.g., search process
- Alone is sufficient to determine the RL agent's behavior
- Stochastic: specifying probabilities for each action

Reward Signal

- Defines the goal of an RL problem
- The environment sends a single number, i.e., reward, to the RL agent
- RL agent's objective: max(long run reward)
- Defines what is good in an **immediate sense**
  - If an action selected by the policy provides low reward
  - Select some other action in the future

Value Function

- Specifies what is good in the **long run**
- Total amount of reward an agent can expect to accumulate over the future starting from a state
- Values indicate long-term desirability of states after taking into account the states that are likely to follow and the rewards available to those states
- Action choices are made based on the value judgements **not** the reward
- Values are (re-)estimated from the sequences of observations an agent makes over its entire lifetime
- RL algorithms design: method for efficiently estimating the values – *top 10 problem!*

LECTURE 2

Reinforcement Learning

Introduction

Model of the Environment

- Allows inferences to be made about how the environment will behave
- The model predicts the resultant next state and next reward, **given** a current state and action
- Models are used for **planning** →deciding on a course of action by considering possible future situations before they are actually experienced
- **Model-based RL:** Uses models for planning
- **Model-free RL:** Explicitly trial-and-error

RL vs Evolutionary Methods

- Evolutionary methods
  - The policies that obtain the most reward are carried to the next generation of policies in an iterative manner
  - They do not learn while interacting with the environment
  - Examples: genetic algorithms, simulated annealing, etc.

Tic-Tac-Toe Example

- Solved based on classical techniques:
  - Game-theoretic Minimax solution: assumes a particular playing by the opponent
  - Classical Optimization, e.g., dynamic programming: requires a complete specification of the opponent (i.e., probabilities of the opponent's moves) to compute the optimal solution
- Evolutionary method
  - Policy identifies the player's move for every state of the game
  - For each **fixed** policy: play multiple games with the opponent in order to estimate the winning probability

Reinforcement Learning approach to solve Tic-Tac-Toe Problem

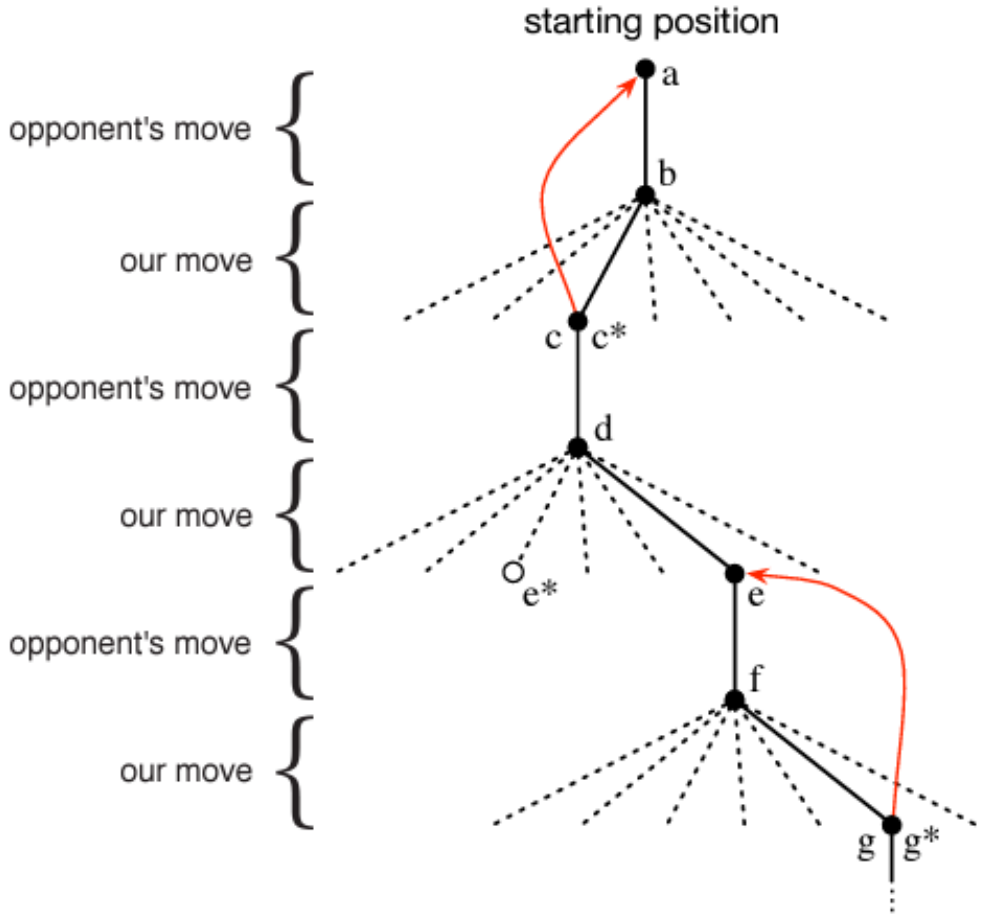
- Set up a **table of numbers (value function)**, one for each possible state of the game
- Each number represents the latest estimate of the probability of winning from that state
- Example: (I play Xs)
  - State with 3X's in the row → probability=1
  - State with 3O's in the row → probability=0
  - All other states → probability=0.5 (50% to win)
- Play multiple games with the opponent to build the table
  - Tend to play greedily (select the move that leads to the state with greatest values in the table)
  - BUT, exploratory moves (randomly selecting other moves) as well! Explore potentially better moves!

Temporal-difference Learning Method solving Tic-Tac-Toe Problem

- Goal: accurately estimate the probability of winning
- How? Back up the value of the state after each greedy move to the state before the move
- S_t: state before the greedy move
- S_{t+1}: state after the greedy move
- Estimated value of S_t: $V(S_t) \leftarrow V(S_t) + \alpha[V(S_{t+1}) - V(S_t)]$
  - α : Step-size parameter influencing the learning rate
  - $[V(S_{t+1}) - V(S_t)]$: temporal difference

Sequence of Tic-Tac-Toe Moves

- Solid black: moves taken
- Dashed lines: available moves (not selected)
- *: moves currently estimated to be the best
- Second move: exploratory move (e* was ranked with higher reward)
- Exploratory moves do not result in learning (e.g., f)
- Other moves result in updates (red arrows)



Evolutionary method vs RL in Tic-Tac-Toe Problem

- Holds a policy fixed
- Plays many games against the opponent
- Determines the frequency of wins
- Gives an unbiased estimate of the probability of winning with that policy
- Directs the next policy selection and repeat…
- What happens during the game is ignored

- Reinforcement learning
  - RL agent learns during the game
  - Individual states are evaluated

Key features of Reinforcement Learning

- Learning while interacting with the environment
  - RL agent has a clear goal
  - Correct behavior requires planning or foresight that considers delayed effects of one's choices
- RL applies in the case that there is no external adversary (like in the tic-tac-toe example)
- RL is not restricted to problems which behavior breaks down into separate episodes
- RL applies to problems that do not break down into discrete time steps, i.e., continuous time problems
- RL applies in problems where the state set is very large or even infinite (e.g., Gerry Tesauro: play backgammon with 10^{20} states)
- RL can generalize RL agent's decision based on past experience: combine supervised learning (artificial neural networks) and deep learning within reinforcement learning

More Key Features…

- RL can incorporate prior information for efficient learning
- RL can be applied when part of the state is hidden from the RL agent or the RL agent perceives different states as the same
- RL methods
  - Model-based
  - Model-free: convenient when constructing a sufficiently accurate environment model is not easy

What did we learn?

- RL is a computational approach to understanding and automating goal-directed learning and decision making
- RL agent learns by interacting with the environment
- No need for exemplary supervision or complete models of the environment
- RL uses the formal framework of Markov decision processes to define the interaction between the RL agent and its environment in terms of *states*, *actions*, and *rewards*
- *Value function*: key component in the design of an RL method
- Value function characterizes the RL methods vs the evolutionary methods which search directly in policy space guided by evaluation of entire **fixed** policies

RL has a long history

- It started as…*optimal control*
- And then…*trial and error*
- Modernized with…*temporal-difference methods*

Optimal Control

- Design a controller to minimize/maximize a measure of a dynamical system's behavior over time (1950's Richard Bellman)
- A functional equation (Bellman equation) is defined based on the dynamical system's state and a value function
  - Methods that solve the optimal control problems: **Dynamic Programming (DP)**
- Discrete stochastic version of optimal control problem: **Markov Decision Process (MDP)**
- DP suffers from the "curse of dimensionality": computational requirements grow exponentially with the number of state variables
- Extensions of DP
  - Partially observable MDPs
  - Approximation methods
  - Asynchronous methods
- DP is an offline computation depending on accurate system models and analytic solutions to the Bellman equation
- DP proceeds backward in time vs RL proceeds in a forward direction
- Chris Watkins (1989): treats RL with MDP formalism
- Dimitri Bertsekas and John Tsitsiklis (1996): Neurodynamic programming: combination of DP and ANN (artificial neural networks)

Trial and Error

- “Law of Effect” (Thorndike, 1911): describes the effect of reinforcing events on the tendency to select actions
- “Pleasure-pain system” (Alan Turing, 1948)
- Electro-mechanical machines were designed based on the trial and error learning
- Confusion between Trial and Error and Artificial Neural Networks (ANNs)
  - ANNs learn from training examples
  - ANNs use error information to update connection weights
  - Trial and Error selects an action based on the evaluative feedback without knowing what the correct action should be
- “Selective bootstrap adaptation” or “Learning with a critic” (Widrow, Gupta, Maitra, 1973) modified the Least Mean Square (LMS) algorithm to produce an RL rule that could learn from success and failure signals instead of from training examples
- “Stochastic Learning Automata”: methods to solve purely selectional learning problems (k-armed bandit)
  - Low memory machines for improving the probability of reward in these problems
  - Update action probabilities on the basis of reward signals
- RL in Economics and RL in AI

Temporal-difference Learning

- TD learning methods are distinctive in being driven by the difference between temporally successive estimates of the same quantity
- TD learning origins: animal learning psychology based on the notion of *secondary reinforcers*.
  - A secondary reinforcer is a stimulus that has been paired with a primary reinforcer such as food or pain and has come to take on similar reinforcing properties
- Actor-Critic Architecture: combination of TD learning and trial and error
- Tabular TD(0): adaptive controller for solving MDPs
- TD(λ) algorithm: general prediction method

Reinforcement Learning

Tabular Solution Methods

Multi-armed Bandits

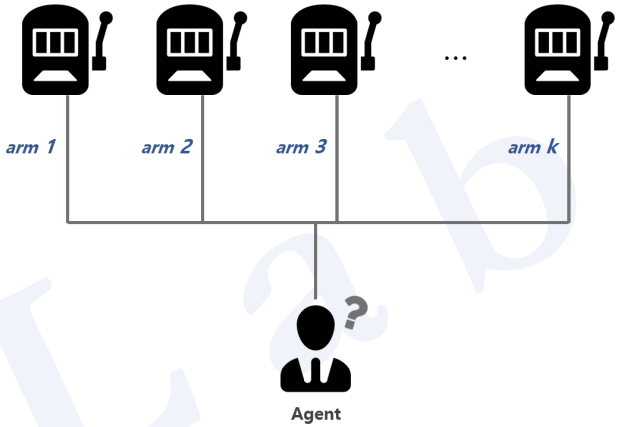
Tabular Solution Methods

- Special case of RL: only a single state → bandit problems
- Finite Markov decision processes, Bellman equation, and value functions
- Dynamic Programming: well-developed mathematically, require a complete and accurate model of the environment
- Monte Carlo methods: don't require a model, conceptually simple, not well-suited for step-by-step incremental computation
- Temporal-difference learning: require no model, fully incremental, but more complex to analyze
- Combinations of the above

Evaluative vs Purely Instructive Feedback

- *Evaluative feedback*: indicates how good the action taken was, but **not** whether it was the best or the worst action possible
  - Depends on the action taken
  - Non-associative setting: does not involve learning in more than one situation
  - Example: k-armed bandit problem
- *Purely instructive feedback*: indicates the correct action to take, independently of the action actually taken (supervised learning)
  - Independent on the action taken
  - Associative setting: the best action depends on the situation

k-armed Bandits



- Choose among *k* different actions
- After each choice, a numerical reward is received chosen from a stationary probability distribution that depends on the selected action
- Goal of RL agent: maximize the expected total reward over some time period
- Action selected in time step t: A_t providing a reward R_t (value of the action)
- The value of an arbitrary action a is $\mathrm{q}^*(\mathrm{a})$ and defined as the expected reward given that a is selected:

$$\mathrm{q}^*(\mathrm{a}) = \mathbb{E}[\mathrm{R_t}|\mathrm{A_t} = \mathrm{a}]$$

- If the value of the action is known, then the RL agent would always select the action with the highest value (complete information scenario)
- Incomplete information scenario: Estimated value of action a at time step t: $Q_t(a)$
- Goal: $Q_t(a) \rightarrow \mathrm{q}^*(\mathrm{a})$
- Exploitation: select the action with the highest estimated value (greedy action)
- Exploration: select a non-greedy action that will offer a lower reward in the short run, but potentially a higher reward in the long run
- Balancing exploration and exploitation

Action-value Methods

- Methods to *estimate the values of actions*, so values estimates can be used for action selection decisions
- Sample-average method:

$$Q_t(a) = \frac{\sum_{i=1}^{t-1} R_i \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}}$$

where $\mathbb{1}_{A_i=a} = 1$ if the statement is true, or 0 otherwise

- If the denominator is 0, the $Q_t(a)$ is assigned a default value, e.g., 0
- As the denominator goes to infinity (law of large numbers), $Q_t(a)$ converges to $\mathrm{q}^*(\mathrm{a})$
- The sample-average method is just a way to estimate action values, but not necessarily the best one
- Simplest way to select an action: always select the one with the highest value (*greedy action*) – if there is a tie select randomly among the greedy actions → performs only exploitation, no exploration

$$A_t = \mathrm{argmax}_a \, Q_t(a)$$

- Combination of exploration and exploitation: ε-greedy method → select a non-greedy action with a small probability ε and with probability 1-ε a greedy action. The selection among the non-greedy actions is performed with equal probability.
- Following the law of large numbers, as the number of steps increases, every action is sampled an infinite number of times, thus $Q_t(a)$ converges to $\mathrm{q}^*(\mathrm{a})$ for all actions

LECTURE 3

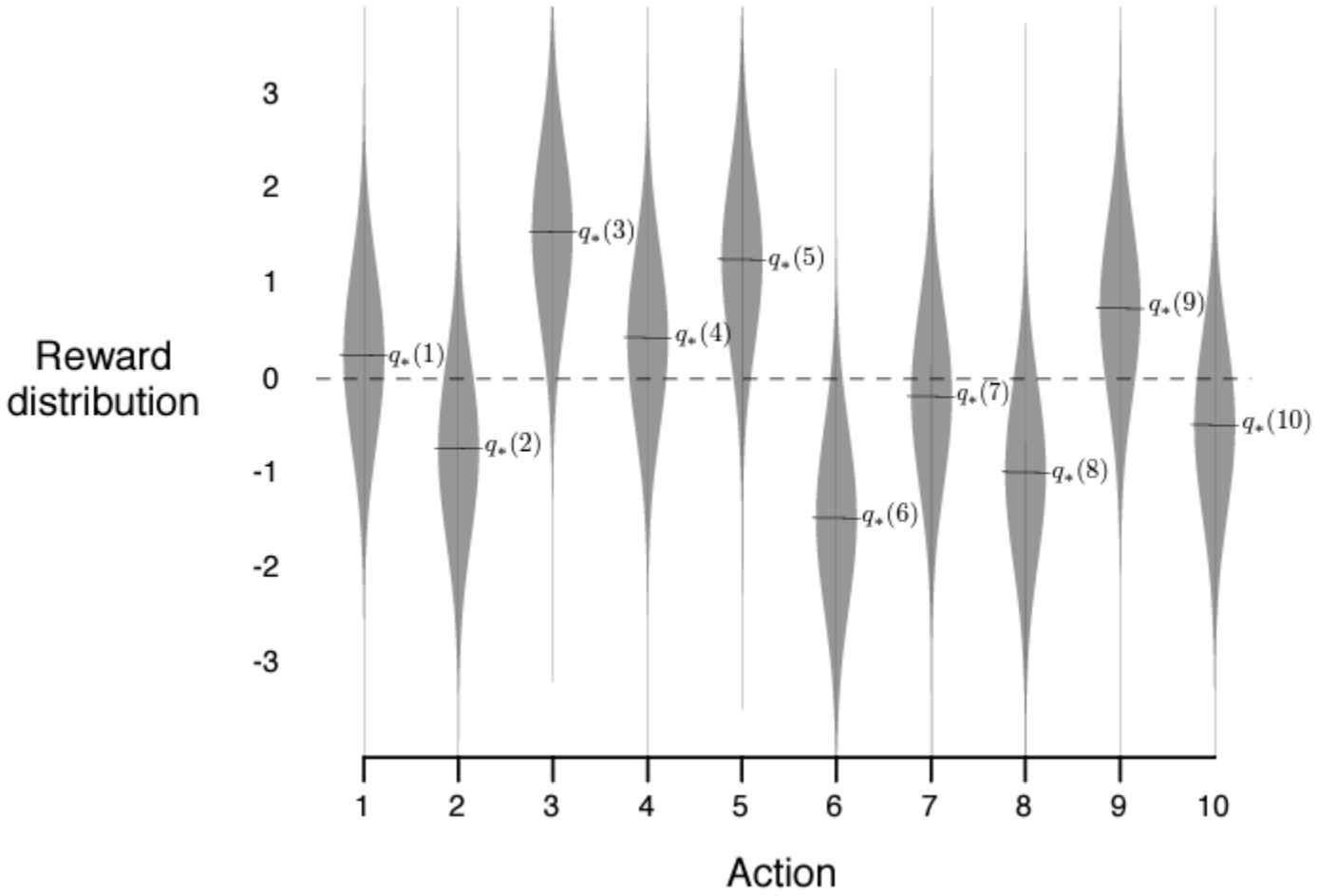
Reinforcement Learning

Tabular Solution Methods

Multi-armed Bandits

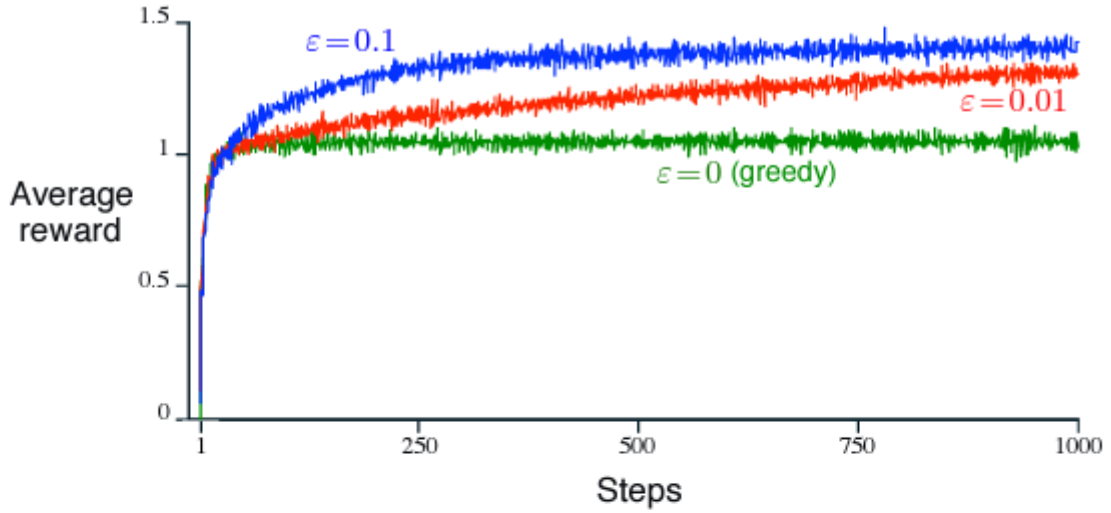
Example: 10-armed Testbed

- 2,000 randomly generated k-armed bandit problems ($k = 10$)
- The true value $q^*(a)$ of each of the ten actions was selected according to a normal distribution with mean zero and unit variance
- The actual rewards were selected according to a mean $q^*(a)$, unit variance normal distribution (gray distributions)



Example: 10-armed Testbed

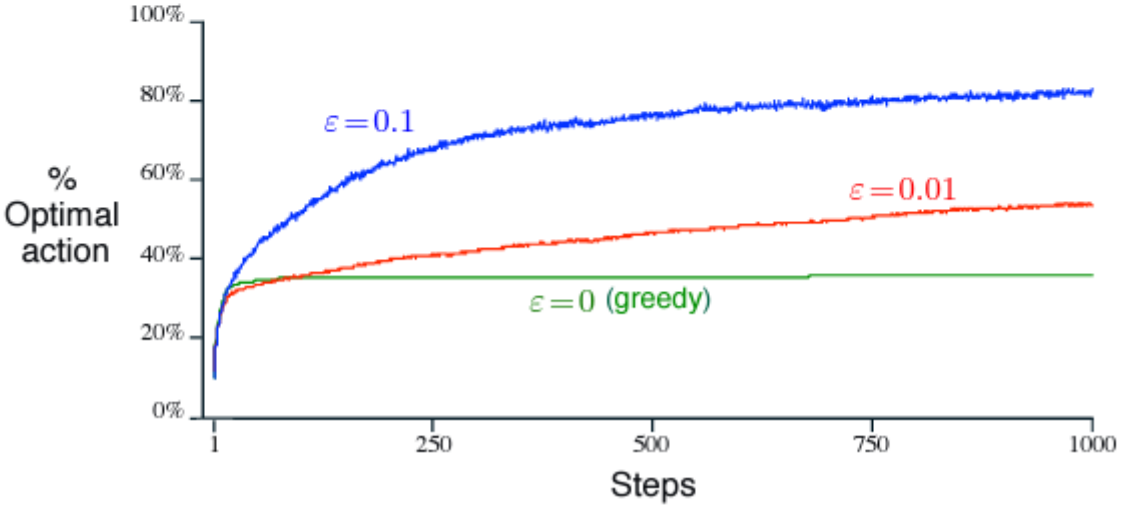
- The greedy method improved slightly faster than the other methods at the very beginning, but then leveled off at a lower level.
- The greedy method performed significantly worse in the long run because it often got stuck performing suboptimal actions.



ε-greedy methods performed better (exploration and exploitation)

ε=0.1 method explored more, found the optimal action earlier, but it never selected that action more than 91% of the time.

ε=0.01 method improved more slowly, but eventually would perform better than the ε=0.1 method.



Incremental Implementation

- How can we compute the averages in a computationally efficient manner?
- Focus on one single action
- R_i: reward received after the *i*th selection of this action
- $\boldsymbol{Q_n}$: **estimate of the action** value after it has been selected $n-1$ times:

$$Q_n = \frac{R_1 + R_2 + \cdots + R_{n-1}}{n-1}$$

- Computationally inefficient: maintain record of all the rewards → computational requirements grow over the time
- Computationally efficient approach:

$$Q_{n+1} = \frac{1}{n}\sum_{i=1}^{n} R_i = \frac{1}{n}(R_n + \sum_{i=1}^{n-1} R_i) = \frac{1}{n}(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i) = \frac{1}{n}(R_n + (n-1)Q_n) = \frac{1}{n}(R_n + nQ_n - Q_n) = Q_n + \frac{1}{n}(R_n - Q_n)$$

- Holds even for $n = 1$, where $Q_2 = R_1$ for arbitrary Q_1
- Memory only for Q_n and n
- General Form:
- $NewEstimate \leftarrow OldEstimate + StepSize\,[Target - OldEstimate]$
- $[Target - OldEstimate]$: error in estimate
- $StepSize$: step-size parameter, changes over the time, general notation: α or $\alpha_t(a)$

A simple bandit algorithm

A simple bandit algorithm

Initialize, for $a = 1$ to k:

$Q(a) \leftarrow 0$

$N(a) \leftarrow 0$

Loop forever:

$$A \leftarrow \begin{cases} \text{argmax}_a\, Q(a) & \text{with probability } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$$

$R \leftarrow bandit(A)$

$N(A) \leftarrow N(A) + 1$

$Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\left[R - Q(A)\right]$

Tracking a Nonstationary Problem

- Averaging methods: appropriate for stationary bandit problems, i.e., the reward probabilities do not change over time
- How can we address reinforcement learning **nonstationary bandit problems**?
- Give more weight to recent rewards than to long-past rewards by using a constant step-size parameter $a \in (0,1]$

$$
\begin{aligned}
Q_{n+1} &= Q_n + \alpha(R_n - Q_n) \\
&= \alpha R_n + (1-\alpha) Q_n \\
&= \alpha R_n + (1-\alpha)[\alpha \mathrm{R}_{\mathrm{n}-1} + (1-\alpha)\mathrm{Q}_{\mathrm{n}-1}] \\
&= \alpha R_n + (1-\alpha)\alpha \mathrm{R}_{\mathrm{n}-1} + (1-\alpha)^2 Q_{n-1} \\
&= \alpha R_n + (1-\alpha)\alpha \mathrm{R}_{\mathrm{n}-1} + (1-\alpha)^2 \alpha R_{n-2} + \cdots + (1-\alpha)^{n-1} \alpha R_1 + (1-\alpha)^n Q_1 \\
&= (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha (1-\alpha)^{n-i} R_i
\end{aligned}
$$

- Weighted average, as $(1-\alpha)^n + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} = 1$
- The weight $\alpha(1-\alpha)^{n-i}$ given to the reward R_i depends on how many rewards ago, $n-i$, it was observed
- $1-\alpha < 1$: the weight given to R_i as the number of intervening rewards increases
- **Exponential recency-weighted average**

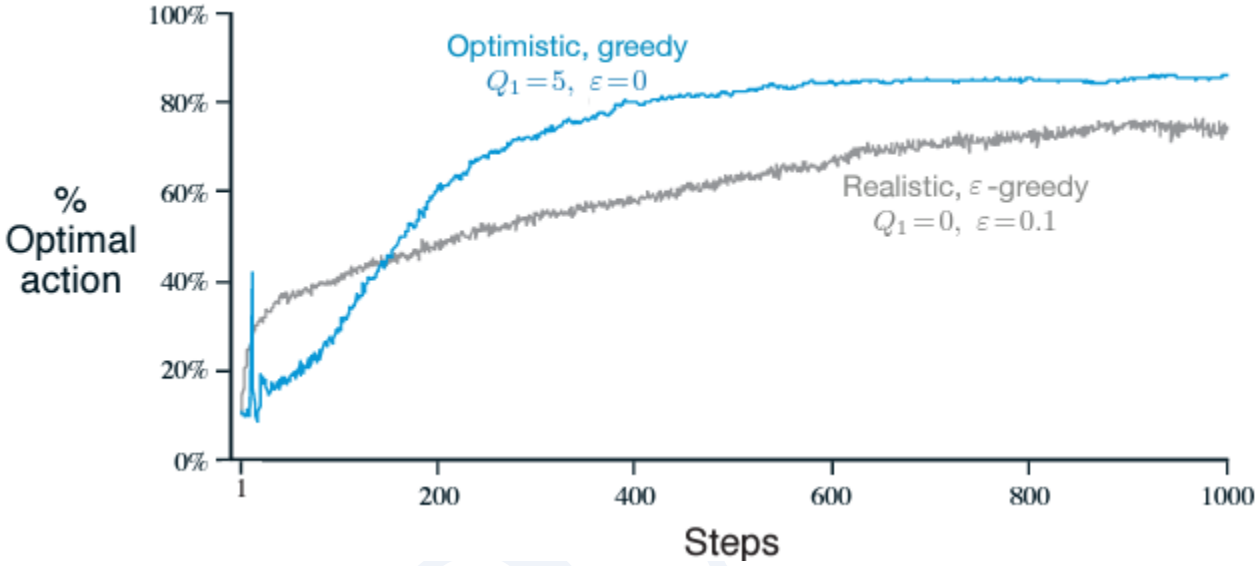
Convergence Conditions

- Sample-average methods: step-size parameter: $\alpha_n(a) = \frac{1}{n}$
- Converges to the true action values by the law of large numbers
- Convergence is not guaranteed for all choices of the sequence $\{\alpha_n(a)\}$
- Required conditions to assure convergence with probability 1 from the stochastic approximation theory:
  - $\sum_{i=1}^{\infty} \alpha_n(a) = \infty$: guarantees that the steps are large enough to eventually overcome the initial conditions or random fluctuations
  - $\sum_{i=1}^{\infty} \alpha_n^2(a) < \infty$: guarantees that the steps become small enough to assure convergence
  - **Sample-average methods satisfy both conditions of convergence**
  - In the case of the sequence $\{\alpha_n(a)\}$, the second condition is not met, thus, the estimates never completely converge, but continue to vary in response to the most recently received rewards→ desirable in a nonstationary environment
  - The two convergence conditions are often used in theoretical works, but they are seldom used in applications

Optimistic Initial Values

- All the methods discussed so far depend on the initial action value estimates $Q_1(\alpha)$
- Biased by the initial estimates
- Sample-average methods: bias disappears once all the actions have been selected at least once
- Exponential recency-weighted average: the bias is permanent, though decreasing over time
- Drawback: initial estimates must be picked by the user
- Benefit: easy way to supply prior knowledge about what level of rewards can be expected

Example



- $Q_1 = 5$: optimistic initial estimate, as $q^*(a)$ was selected from a normal distribution with mean zero and variance one
- Encourage exploration: whichever actions are initially selected, the reward is less than the starting estimates, the learner switches to other actions, being "disappointed" with the rewards it is receiving
- Exploration even if greedy actions are selected all the time
- Simple trick that can be quite effective on stationary problems

____ a general approach to encourage exploration

Uncertainty in Action Selection

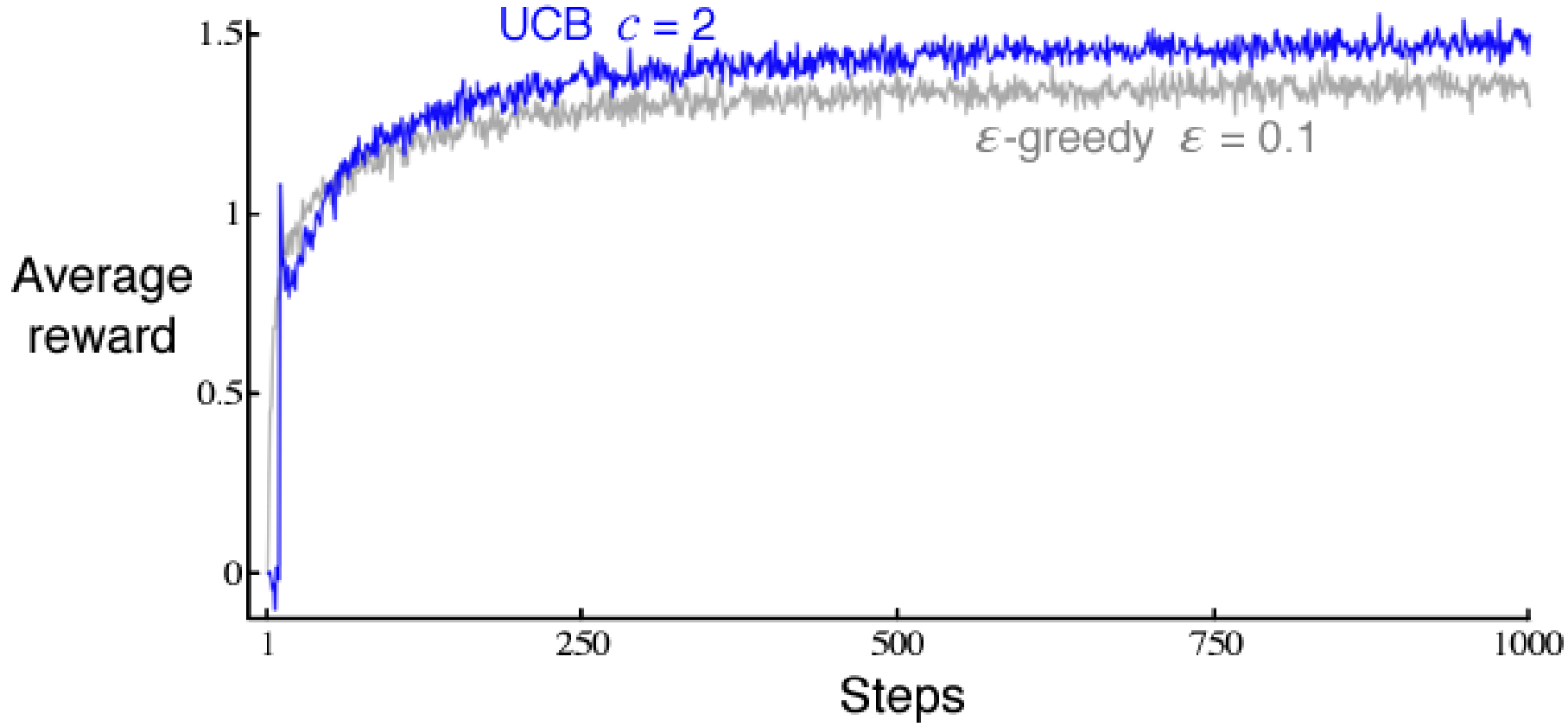
- Exploration is needed because there is always uncertainty about the accuracy of the action-value estimates
- Greedy actions are those that look best at present, but some of the other actions may actually be better
- ε-greedy action selection forces the non-greedy actions to be tried, but with no preference for those that are nearly greedy or particularly uncertain
- Better to select among the non-greedy actions according to their potential for actually being optimal, taking into account:
  - How close their estimates are to being maximal
  - The uncertainties in those estimates

Upper-Confidence-Bound Action Selection

$$A_t = \text{argmax}_\alpha[\text{Q}_\text{t}(\alpha) + \text{c}\sqrt{\frac{\ln t}{N_t(\alpha)}}]$$

- $N_t(\alpha)$: number of times that action α has been selected prior to time t
- $c > 0$: controls the degree of exploration and determines the confidence level
- $\sqrt{\frac{\ln t}{N_t(\alpha)}}$: measure of the uncertainty or variance in the estimate of α's value
- $[\text{Q}_\text{t}(\alpha) + \text{c}\sqrt{\frac{\ln t}{N_t(\alpha)}}]$: upper bound on the possible true value of action a
  - Each time α is selected the uncertainty is reduced: $N_t(\alpha)$ increases and the uncertainty term decreases
  - Each time an action other than α is selected, t increases but $N_t(\alpha)$ does not, thus, the uncertainty estimate increases
- Use of natural logarithm: the increases get smaller over time, but are unbounded
- All actions will eventually be selected, but actions with lower value estimates, or that have already been selected frequently, will be selected with decreasing frequency overtime.

Example of Upper-Confidence-Bound Action Selection



Gradient Bandit Algorithms

- So far: methods that estimate action values and use those estimates to select actions
- Good approach, but it is not the only one possible
- Learn a numerical preference for each action α: $H_t(\alpha) \in \mathrm{R}$
- The larger the preference, the more often that action is taken, but the preference has no interpretation in terms of reward
- Soft-max distribution to determine the action probabilities

$$\Pr\{A_t = \alpha\} = \frac{e^{H_t(\alpha)}}{\sum_{b=1}^{k} e^{H_t(b)}} = \pi_t(\alpha)$$

- $\pi_t(\alpha)$: probability of taking action α at time t
- Initially all action preferences are the same: $H_1(\alpha) = 0, \forall \alpha \in A$, so that all actions have an equal probability of being selected

LECTURE 4

Reinforcement Learning

Tabular Solution Methods

Multi-armed Bandits

Gradient Bandit Algorithms

- So far: methods that estimate action values and use those estimates to select actions
- Good approach, but it is not the only one possible
- Learn a numerical preference for each action α: $H_t(\alpha) \in \mathrm{R}$
- The larger the preference, the more often that action is taken, but the preference has no interpretation in terms of reward
- Soft-max distribution to determine the action probabilities

$$\Pr\{A_t = \alpha\} = \frac{e^{H_t(\alpha)}}{\sum_{b=1}^{k} e^{H_t(b)}} = \pi_t(\alpha)$$

- $\pi_t(\alpha)$: probability of taking action α at time t
- Initially all action preferences are the same: $H_1(\alpha) = 0, \forall \alpha \in A$, so that all actions have an equal probability of being selected

Bandit Gradient Ascent

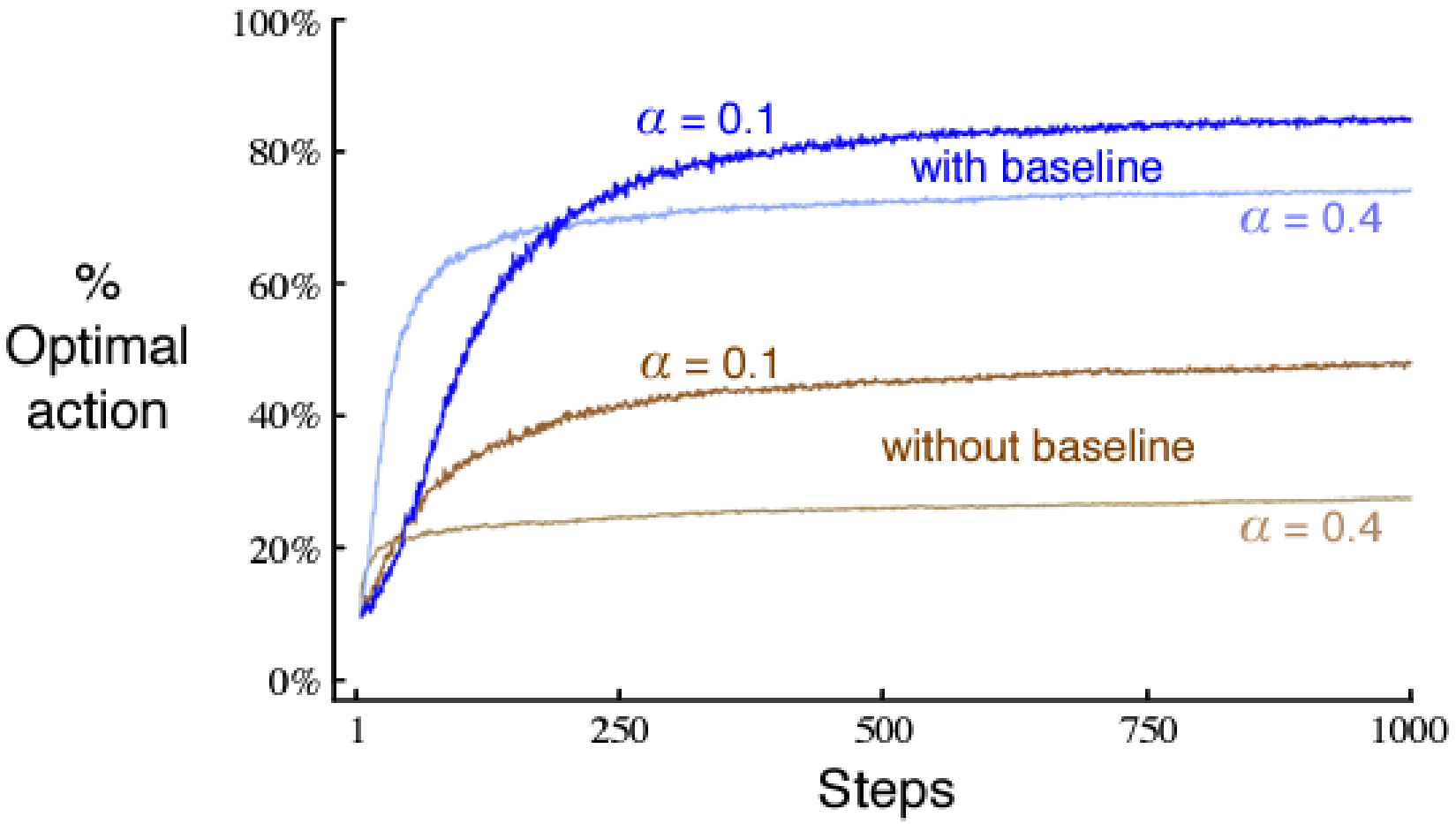
- Bandit Gradient Ascent algorithm: On each step, after selecting action A_t and receiving the reward R_t, the action preferences are updated by

$$H_{t+1}(A_t) = H_t(A_t) + \alpha(R_t - \bar{R}_t)\big(1 - \pi_t(A_t)\big)$$

$$H_{t+1}(\alpha) = H_t(\alpha) - \alpha(R_t - \bar{R}_t)\pi_t(\alpha), \forall \alpha \neq A_t$$

- $\alpha > 0$: step-size parameter, $\bar{R}_t \in \mathbb{R}$: average of all the rewards up through and including time t → baseline with which the reward is compared
  - If the reward is higher than the baseline, then the probability of taking A_t in the future is increased
  - If the reward is below the baseline, then the probability is decreased
  - The non-selected actions move in the opposite direction

Example of Bandit Gradient Ascent



- True expected rewards were selected according to a normal distribution with a mean of +4 (and with unit variance as before)
- If the baseline were omitted, then the performance would be significantly degraded

The Bandit Gradient Algorithm as Stochastic Gradient Ascent

- Gradient bandit algorithm: stochastic approximation to gradient ascent
- In exact gradient ascent, each action preference $H_t(\alpha)$ would be incremented proportional to the increment's effect on performance:

$$H_{t+1}(\alpha) = H_t(\alpha) + \alpha \frac{\partial \mathbb{E}[R_t]}{\partial \mathrm{H_t}(\alpha)}$$

- Measure of performance is the expected reward: $\mathbb{E}[R_t] = \sum_x \pi_t(x) q^*(x)$
- $\frac{\partial \mathbb{E}[R_t]}{\partial \mathrm{H_t}(\alpha)}$: measure of the increment's effect on the performance
- The updated of the bandit gradient algorithm are an instance of the stochastic gradient ascent

Proof (1/3)

$$\frac{\partial \mathbb{E}[R_t]}{\partial \mathrm{H_t}(\alpha)} = \frac{\partial}{\partial \mathrm{H_t}(\alpha)}\left[\sum_x \pi_t(x) q^*(x)\right] = \sum_x q^*(x)\frac{\partial \pi_t(x)}{\partial \mathrm{H_t}(\alpha)} = \sum_x (q^*(x) - B_t)\frac{\partial \pi_t(x)}{\partial \mathrm{H_t}(\alpha)}$$

- B_t: baseline – any scalar value that does not depend on x
- We can include a baseline here without changing the equality because the gradient sums to zero over all the actions: $\sum_x \frac{\partial \pi_t(x)}{\partial \mathrm{H_t}(\alpha)} = 0$.
- As $\mathrm{H_t}(\alpha)$is changed, some actions' probabilities go up and some go down, but the sum of the changes must be zero because the sum of the probabilities is always one
- Multiply each term with $\frac{\pi_t(x)}{\pi_t(x)}$: $\frac{\partial \mathbb{E}[R_t]}{\partial \mathrm{H_t}(\alpha)} = \sum_x \pi_t(x)(q^*(x) - B_t)\frac{\partial \pi_t(x)}{\partial \mathrm{H_t}(\alpha)}/\pi_t(x)$

Proof (2/3)

$$\frac{\partial \mathbb{E}[R_t]}{\partial \mathrm{H_t}(\alpha)} = \sum_x \pi_t(x)(q^*(x) - B_t)\frac{\partial \pi_t(x)}{\partial \mathrm{H_t}(\alpha)}/\pi_t(x)$$

- The equation is now in the form of an expectation, summing over all possible values x of the random variable A_t, then multiplying by the probability of taking those values $\pi_t(x)$

$$\frac{\partial \mathbb{E}[R_t]}{\partial \mathrm{H_t}(\alpha)} = \mathbb{E}[\,(q^*(A_t) - B_t)\frac{\partial \pi_t(A_t)}{\partial \mathrm{H_t}(\alpha)}/\pi_t(A_t)] = \mathbb{E}[(\mathrm{R_t} - \bar{R}_t)\frac{\partial \pi_t(A_t)}{\partial \mathrm{H_t}(\alpha)}/\pi_t(A_t)]$$

- Where we set $B_t = \bar{R}_t$, and substitute R_t for $q^*(A_t)$, as $\mathbb{E}[\mathrm{R_t}|\mathrm{A_t}] = q^*(A_t)$
- In the following we will show that $\frac{\partial \pi_t(x)}{\partial \mathrm{H_t}(\alpha)} = \pi_t(x)(\mathbb{1}_{\alpha=x} - \pi_t(a))$, where $\mathbb{1}_{\alpha=x}$ is defined to be 1 if $\alpha = x$, else 0

The Bandit Gradient Algorithm as Stochastic Gradient Ascent

- Gradient bandit algorithm: stochastic approximation to gradient ascent
- In exact gradient ascent, each action preference $H_t(\alpha)$ would be incremented proportional to the increment's effect on performance:

$$H_{t+1}(\alpha) = H_t(\alpha) + \alpha \frac{\partial \mathbb{E}[R_t]}{\partial \mathrm{H_t}(\alpha)}$$

- Measure of performance is the expected reward: $\mathbb{E}[R_t] = \sum_x \pi_t(x) q^*(x)$
- $\frac{\partial \mathbb{E}[R_t]}{\partial \mathrm{H_t}(\alpha)}$: measure of the increment's effect on the performance
- The updated of the bandit gradient algorithm are an instance of the stochastic gradient ascent

Proof (1/3)

$$\frac{\partial \mathbb{E}[R_t]}{\partial \mathrm{H_t}(\alpha)} = \frac{\partial}{\partial \mathrm{H_t}(\alpha)}\left[\sum_x \pi_t(x) q^*(x)\right] = \sum_x q^*(x)\frac{\partial \pi_t(x)}{\partial \mathrm{H_t}(\alpha)} = \sum_x (q^*(x) - B_t)\frac{\partial \pi_t(x)}{\partial \mathrm{H_t}(\alpha)}$$

- B_t: baseline – any scalar value that does not depend on x
- We can include a baseline here without changing the equality because the gradient sums to zero over all the actions: $\sum_x \frac{\partial \pi_t(x)}{\partial \mathrm{H_t}(\alpha)} = 0$.
- As $\mathrm{H_t}(\alpha)$is changed, some actions' probabilities go up and some go down, but the sum of the changes must be zero because the sum of the probabilities is always one
- Multiply each term with $\frac{\pi_t(x)}{\pi_t(x)}$: $\frac{\partial \mathbb{E}[R_t]}{\partial \mathrm{H_t}(\alpha)} = \sum_x \pi_t(x)(q^*(x) - B_t)\frac{\partial \pi_t(x)}{\partial \mathrm{H_t}(\alpha)}/\pi_t(x)$

Proof (2/3)

$$\frac{\partial \mathbb{E}[R_t]}{\partial \mathrm{H_t}(\alpha)} = \sum_x \pi_t(x)(q^*(x) - B_t)\frac{\partial \pi_t(x)}{\partial \mathrm{H_t}(\alpha)}/\pi_t(x)$$

- The equation is now in the form of an expectation ,summing over all possible values x of the random variable A_t, then multiplying by the probability of taking those values $\pi_t(x)$

$$\frac{\partial \mathbb{E}[R_t]}{\partial \mathrm{H_t}(\alpha)} = \mathbb{E}[\,(q^*(A_t) - B_t)\frac{\partial \pi_t(A_t)}{\partial \mathrm{H_t}(\alpha)}/\pi_t(A_t)] = \mathbb{E}[(\mathrm{R_t} - \bar{R}_t)\frac{\partial \pi_t(A_t)}{\partial \mathrm{H_t}(\alpha)}/\pi_t(A_t)]$$

- Where we set $B_t = \bar{R}_t$, and substitute R_t for $q^*(A_t)$, as $\mathbb{E}[\mathrm{R_t}|\mathrm{A_t}] = q^*(A_t)$
- In the following we will show that $\frac{\partial \pi_t(x)}{\partial \mathrm{H_t}(\alpha)} = \pi_t(x)(\mathbb{1}_{\alpha=x} - \pi_t(a))$, where $\mathbb{1}_{\alpha=x}$ is defined to be 1 if $\alpha = x$, else 0

$$\{A_t = \alpha\} = \frac{e^{H_t(\alpha)}}{\sum_{b=1}^{k} e^{H_t(b)}} = \pi_t(\alpha)$$

Proof (3/3)

- Prove $\frac{\partial \pi_t(x)}{\partial \mathrm{H_t(\alpha)}} = \pi_t(x)(\mathbb{1}_{\alpha=x} - \pi_t(a))$
- $\frac{\partial \pi_t(A_t)}{\partial \mathrm{H_t(\alpha)}} = \frac{\partial}{\partial \mathrm{H_t(\alpha)}}\left[\frac{e^{H_t(x)}}{\sum_{y=1}^{k} e^{H_t(y)}}\right] = \frac{\frac{\partial e^{H_t(x)}}{\partial \mathrm{H_t(\alpha)}} \sum_{y=1}^{k} e^{H_t(y)} - e^{H_t(x)} \frac{\partial \sum_{y=1}^{k} e^{H_t(y)}}{\partial \mathrm{H_t(\alpha)}}}{\left(\sum_{y=1}^{k} e^{H_t(y)}\right)^2} = \frac{\mathbb{1}_{\alpha=x} e^{H_t(x)} \sum_{y=1}^{k} e^{H_t(y)} - e^{H_t(x)} e^{H_t(\alpha)}}{\left(\sum_{y=1}^{k} e^{H_t(y)}\right)^2} =$
  $\frac{\mathbb{1}_{\alpha=x} e^{H_t(x)}}{\sum_{y=1}^{k} e^{H_t(y)}} - \frac{e^{H_t(x)} e^{H_t(\alpha)}}{\left(\sum_{y=1}^{k} e^{H_t(y)}\right)^2} = \mathbb{1}_{\alpha=x}\pi_t(x) - \pi_t(x)\pi_t(a) = \pi_t(x)(\mathbb{1}_{\alpha=x} - \pi_t(a))$
- Thus, we can write: $\frac{\partial \mathbb{E}[R_t]}{\partial \mathrm{H_t(\alpha)}} = \mathbb{E}[(\mathrm{R_t} - \bar{R}_t)\pi_t(A_t)(\mathbb{1}_{\alpha=x} - \pi_t(a)) \,/\pi_t(A_t)] = \mathbb{E}[(\mathrm{R_t} - \bar{R}_t)\,(\mathbb{1}_{\alpha=x} - \pi_t(a))]$
- Substituting a sample of the latter expectation to the performance gradient, we have:

$$H_{t+1}(\alpha) = H_t(\alpha) + \alpha(\mathrm{R_t} - \bar{R}_t)\,(\mathbb{1}_{\alpha=x} - \pi_t(a))$$

- We showed that the bandit gradient algorithm can be considered as equivalent to the stochastic gradient algorithm

Nonassociative Tasks

- So far … Nonassociative tasks: tasks in which there is no need to associate different actions with different situations
- The learner either tries to find a single best action when the task is stationary or tries to track the best action as it changes over time when the task is nonstationary
- In a general reinforcement learning task there is more than one situation, and the goal is to learn a policy: a mapping from situations to the actions that are best in those situations

Example of Extending Nonassociative Tasks to the Associative Setting

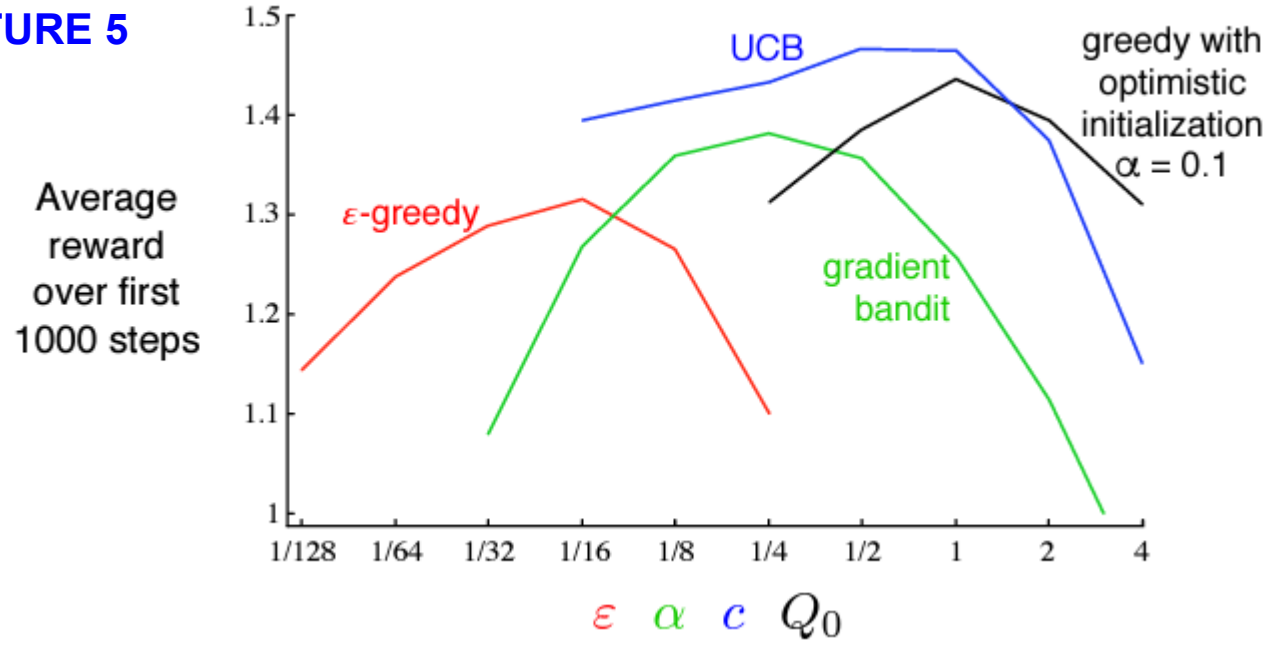
- Suppose there are several different k-armed bandit tasks, and that on each step you confront one of these chosen at random
- The bandit task changes randomly from step to step → If the probabilities with which each task is selected for you do not change over time: this would appear to you as a single, nonstationary k-armed bandit task whose true action values change randomly from step to step
- In contrast, suppose that when a bandit task is selected for you, you are given some distinctive clue about its identity (but not its action values)
- Example: Actual slot machine that changes colors as it changes action values
  - You learn a policy associating each task, signaled by the color you see
  - If red, select arm 1
  - If green, select arm 2
  - With the right policy you can usually do much better than you could in the absence of any information distinguishing one bandit task from another.
- Associative search tasks: contextual bandits → intermediate between the k-armed bandits and the full reinforcement learning problem
- If actions are allowed to affect the next situation as well as the reward, then we have the full reinforcement learning problem.

What did we learn about balancing exploration and exploitation?

- ε-greedy methods choose randomly a small fraction of the time
- UCB methods choose deterministically but achieve exploration by subtly favoring at each step the actions that have so far received fewer samples
- Gradient bandit algorithms estimate not action values, but action preferences, and favor the more preferred actions in a graded, probabilistic manner using a soft-max distribution

Ask me the question!

Parameter Study



- One well-studied approach to balancing exploration and exploitation in k-armed bandit problems is to compute a special kind of action value called **Gittins index**
- **BUT** neither the theory nor the computational tractability of this approach appear to generalize to the full reinforcement learning problem

Gittins Index Approach

- Instance of Bayesian methods, which assume a known initial distribution over the action values and then update the distribution exactly after each step (assuming that the true action values are stationary)
- The update computations can be very complex, but for certain special distributions (called conjugate priors) they are easy
- One possibility is to then select actions at each step according to their posterior probability of being the best action → Posterior sampling or Thompson sampling, often performs similarly to the best of the distribution-free methods
- In the Bayesian setting it is even conceivable to compute the optimal balance between exploration and exploitation
- Compute for any possible action the probability of each possible immediate reward and the resultant posterior distributions over action values
- Given a horizon, e.g., 1,000 steps, one can consider all possible actions, all possible resulting rewards, all possible next actions, all next rewards, and so on for all 1000steps.
- Given the assumptions, the rewards and probabilities of each possible chain of events can be determined, and one need only pick the best.
- The tree of possibilities grows extremely rapidly, e.g., 2^{2000} leaves

Reinforcement Learning

Finite Markov Decision Processes

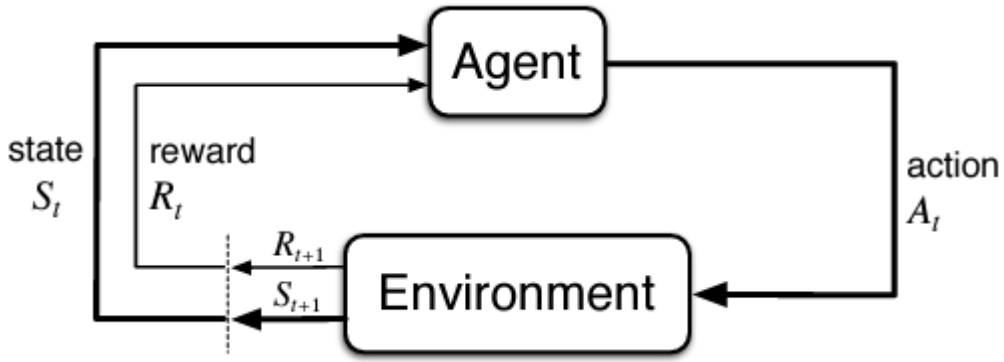
Intro to Finite Markov Decision Processes

- Finite MDP consist of:
  - Evaluative feedback (as in bandits)
  - Associative aspect (choose different actions in different situations)
- Finite MDP: sequential decision-making→ actions influence not just the immediate rewards, but also subsequent situations (states) and through the latter ones, also future rewards
- Finite MDP: involve delayed rewards and trade off among immediate and delayed rewards
  - Bandit problems: estimate the value $q^*(\alpha)$ of each action α
  - Finite MDP: estimate the value $q^*(s, \alpha)$ of each action α in each state s and estimate the value $\mathrm{v}^*(s)$ of each state given the optimal action selections
- Finite MDP: mathematically idealized form of the RL problem – precise theoretical statements can be made
- Key Elements of Finite MDP: returns, value functions, Bellman equations

Agent-Environment Interface

- RL agent (decision maker) interacts with the environment
- Environment: provides reward to the RL agent, who aims at maximizing the rewards over the time through its choices
- The RL agent and the environment interact in a sequence of discrete time steps: $t = 0,1,2,3, \ldots$
- At each time step t, the RL agent receives some representation of the environment's state $S_t \in \mathcal{S}$ → selects an action $A_t \in \mathcal{A}(s)$→ one time step later (as a consequence of its action), the agent receives a reward R_{t+1} and transitions to a new state S_{t+1}
- The MDP and the agent define a sequence: $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots$
- Finite MDP: finite sets $\mathcal{S}$, $\mathcal{A}$, $\mathcal{R}$
- The random variables R_t and S_t have well defined discrete probability distributions depending only on the preceding state and action:

$$p(s', r|s, \alpha) = Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = \alpha\}, \forall s', s \in \mathcal{S}, \alpha \in \mathcal{A}(s)$$



$$(s', r|s, \alpha) = 1, \forall s \in \mathcal{S}, \alpha \in \mathcal{A}(s)$$

___ at all on earlier states and actions.

LECTURE 6

Reinforcement Learning

Finite Markov Decision Processes

Some useful formulas

- State-transition probabilities

$$p(s'|s,\alpha) = Pr\{S_t = s' \,|S_{t-1} = s, A_{t-1} = \alpha\} = \sum_{r\in\mathcal{R}} p(s', r|s, \alpha)$$

- Expected reward for state-action pairs

$$r(s,\alpha) = \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = \alpha] = \sum_{r\in\mathcal{R}} r \sum_{s'\in\mathcal{S}} p(s', r|s, \alpha)$$

- Expected rewards for state-action-next state triples

$$r(s,\alpha,s') = \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = \alpha, S_t = s'] = \sum_{r\in\mathcal{R}} r \frac{p(s', r|s, \alpha)}{p(s'|s,\alpha)}$$

Finite MDP Applications

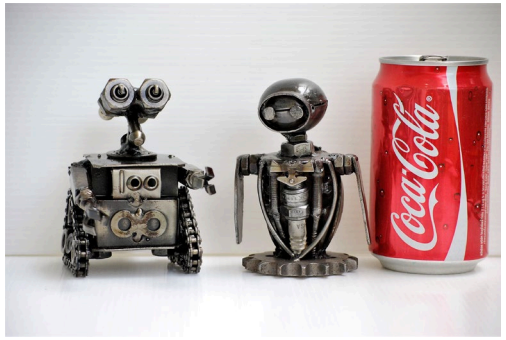
- The time steps do not only refer to real time intervals
  - They can refer to arbitrary successive stages of decision-making and acting
- Actions can be
  - low-level controls
    - Voltages applied to the motors of a robot arm
  - High-level decisions
    - Whether or not to have lunch
    - Whether or not to go to graduate school
- States can be
  - Low-level sensations
    - Direct sensor readings
  - High-level and abstract
    - Symbolic descriptions of objects in a room
  - Mental of subjective
    - An agent can be in the state of not being sure where an object is

Boundary between an RL agent and the Environment

- Examples:
  - Motors and mechanical linkages of a robot and its sensing hardware should usually be considered part of the environment rather than the agent
  - Muscles, skeleton, and sensory organs of an RL agent-animal/person are part of the environment
- Rewards are external to the RL agent
- Anything that cannot be controlled arbitrarily from the RL agent is part of the environment
- Everything in the environment is not unknown to the RL agent
- Often, RL agent knows about how its rewards are computed as a function of its actions and states
- The rewards computation are external to the RL agent, i.e., the RL agent cannot control them
- The agent-environment boundary represents the limit of the RL agent's absolute control, NOT of its knowledge

Finite MDP – Goal-directed Learning

- Finite MDP is an abstraction of the problem of goal-directed learning from interaction
- Any goal-directed learning from interaction problem is characterized by 3 signals exchanged between the agent and the environment:
  - Signal #1: represents the choices made by the agent, i.e., actions
  - Signal #2: represents the basis on which the choices are made, i.e., states
  - Signal #3: defines the agent's goal, i.e., rewards

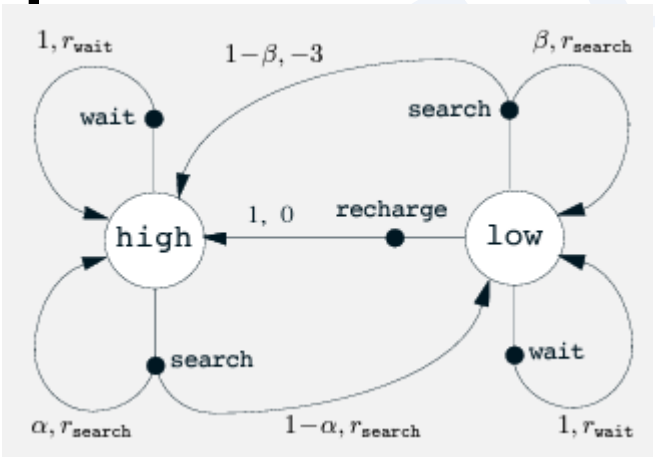


- Mobile robot collects empty soda cans
- It has sensors detecting cans, an arm and gripper to pick them up and place them in an onboard bin, and a rechargeable battery
- High-level decisions: how to search for cans based on an RL agent considering the current charge level of battery
- Two charge levels: $\mathcal{S} = \{high, low\}$ (set of states)
- In each state, the RL agent decides to:
  - Actively search for a can for a certain time period
  - Remain stationary and wait for someone to bring a can
  - Head back to base to recharge
  - Set of actions: $\mathcal{A}(high) = \{search, wait\}$ and $\mathcal{A}(low) = \{search, wait, recharge\}$

Finite MDP and Transition Graph

- Rewards:
  - Zero most of the time
  - Positive when the robot secures a can: 1
  - Large negative when the robot runs out of battery (RESCUE ME!): -3
  - $r_{search} > r_{wait}$
  - Non-possible transition probabilities: 0
  - If the robot starts with high energy level, search and remains with high energy level: probability α
  - High → search→ low energy level: probability $1 - \alpha$
  - Low → search→ low energy level: probability β
  - Low → search→ high energy level: probability $1 - \beta$ (the robot was rescued and recharged its battery)

s	a	s'	$p(s'\|s,a)$	$r(s,a,s')$
high	search	high	α	r_{search}
high	search	low	$1-\alpha$	r_{search}
low	search	high	$1-\beta$	-3
low	search	low	β	r_{search}
high	wait	high	1	r_{wait}
high	wait	low	0	-
low	wait	high	0	-
low	wait	low	1	r_{wait}
low	recharge	high	1	0
low	recharge	low	0	-



Goals and Rewards

- The purpose or goal of an RL agent: formalized as a special signal → reward $R_t \in \mathrm{R}$
- RL agent's goal: maximize the total amount of reward – not the immediate reward BUT the cumulative reward in the long run
  - Maximize the expected value of the cumulative sum of the received reward
- Examples:
  - Make a robot learn how to walk: provide a reward on each time step proportional to the robot's forward motion
  - Make a robot learn to escape a maze: reward -1 for every time step that passes prior to escape (learn as quickly as possible)
  - Make a robot to find and collect empty soda cans: $+1$ for each can collected, 0 most of the time, -5 when the robot bumps into things or someone yells at it ☹
  - Learn to play checkers or chess: $+1$ for winning, -1 for losing, 0 for all nonterminal positions
- The rewards is a signal enabling the RL agent to learn **what** it wants to achieve and not how!
- If you reward a chess-playing agent for achieving subgoals, e.g., taking its opponents pieces, then the RL agent might find a way to achieve them without achieving the real goal (win!)

Episodic Tasks

- How is the cumulative reward defined?
- If I have a sequence of rewards …, $R_{t+1}, R_{t+2}, R_{t+3}, \ldots$ what precise aspect of this sequence should I maximize?
- RL agent's goal: maximize the expected return G_t, which is defined as a function of the rewards sequence
- Example:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T, T\text{: final time step}$$

- Makes sense in applications with a natural notion of final time step, i.e., the agent-environment interaction breaks naturally into subsequences (**episodes**)
- Each episode ends in a special state: terminal state, followed by a reset to a standard starting state or to a sample from a standard distribution of starting states
- Next episode begins independently of how the previous one ended
- Tasks with episodes: episodic tasks
- Set of non-terminal states $\mathcal{S}$ plus the terminal state: $\mathcal{S}^+$
- Time of termination T may vary from episode to episode

Continuing Tasks

- Agent-environment interaction goes on continually without limit: continuing tasks
- Problematic formulation: $G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$ → infinite
- Concept of discounting: RL agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximized:

$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, 0 \leq \gamma \leq 1$ (γ: discount rate)

- A reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately
- If $\gamma < 1$: $\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ has a finite value, if the reward sequence $\{R_k\}$ is bounded
- If $\gamma = 0$: the agent is “myopic” in being concerned only with maximizing immediate rewards
- If $\gamma \to 1$: the return objective takes future rewards into account more strongly → the agent becomes more farsighted

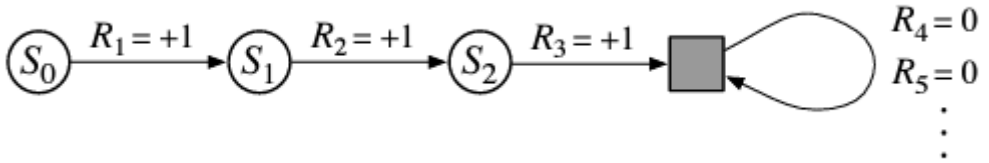
Expected Return

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots = R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots) = \mathrm{R_{t+1}} + \gamma \mathrm{G_{t+1}}$$

- This formula works for $t < T$, even if the termination occurs in $t + 1$, if we set $G_T = 0$
- For $R_t = 1, \forall t: \sum_{k=0}^{\infty} \gamma^{\mathrm{k}} = \frac{1}{1-\gamma}$

Unified Notation for Episodic and Continuing Tasks

- Episodic tasks: we need to consider a series of episodes, each of which consists of a finite sequence of time steps
- State: $S_{t,i}$: time t, episode i
- In practical analysis/problems, we almost never distinguish between different episodes→ almost always consider a particular episode→ abuse the notation $S_{t,i}$→ S_t
- Unify episodic and continuing tasks by considering episode termination to be the entering of a special absorbing state that transitions only to itself and that generates only rewards of zero



- Rewrite the expected return: $G_t = \sum_{k=t+1}^{T} \gamma^{\mathrm{k-t-1}} \mathrm{R_k}$

State-value Function $v_\pi(s)$

- Value functions of states (or of state–action pairs) that estimate how good it is for the agent to be in a given state (or how good it is to perform a given action in a given state)
- How good→ defined in terms of expected future rewards (expected return)
- The rewards the agent can expect to receive in the future depend on what actions it will take
- Value functions are defined with respect to particular ways of acting → policies
- Policy: a mapping from states to probabilities of selecting each possible action
- $\pi(\alpha|s)$: probability that $A_t = \alpha$ if $S_t = s$→ probability distribution over $\alpha \in \mathcal{A}(s)$ for each $s \in \mathcal{S}$
- State-value function of a state s under a policy π is the expected return when starting from state s and following the policy π→ For MDPs:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^{\mathrm{k}} \mathrm{R}_{\mathrm{t+k+1}}|S_t = s\right], \forall s \in \mathcal{S}$$

- The value of the terminal state is always zero

LECTURE 7

Reinforcement Learning

Finite Markov Decision Processes

State-value Function $v_\pi(s)$

- Value functions of states (or of state–action pairs) that estimate how good it is for the agent to be in a given state (or how good it is to perform a given action in a given state)
- How good→ defined in terms of expected future rewards (expected return)
- The rewards the agent can expect to receive in the future depend on what actions it will take
- Value functions are defined with respect to particular ways of acting → policies
- Policy: a mapping from states to probabilities of selecting each possible action
- $\pi(\alpha|s)$: probability that $A_t = \alpha$ if $S_t = s$→ probability distribution over $\alpha \in \mathcal{A}(s)$ for each $s \in \mathcal{S}$
- State-value function of a state s under a policy π is the expected return when starting from state s and following the policy π→ For MDPs:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^{\mathrm{k}} \mathrm{R}_{\mathrm{t+k+1}}|S_t = s\right], \forall s \in \mathcal{S}$$

- The value of the terminal state is always zero

Action-value Function $q_\pi(s, \alpha)$

- Action-value function defines the value of taking action α in state s under the policy π

$$q_\pi(s, \alpha) = \mathbb{E}_\pi[G_t | S_t = s, \mathrm{A}_t = \alpha] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^{\mathrm{k}} \mathrm{R}_{\mathrm{t+k+1}} | S_t = s\,, \mathrm{A}_t = \alpha\right]$$

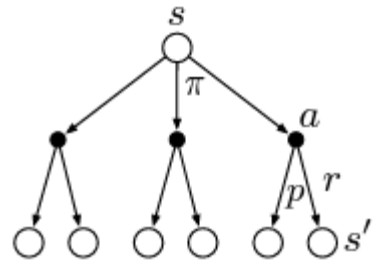
- Monte Carlo methods:
  - If an agent follows policy π and maintains an average, for each state encountered, of the actual returns that have followed that state, then the average will converge to the state's value $v_\pi(s)$ (for infinite number of encountering that state)
  - If separate averages are kept for each action taken in each state, then these averages will converge to the action values $q_\pi(s, \alpha)$
- If there are many states, then it may not be practical to keep separate averages for each state individually
  - The agent would have to maintain $v_\pi(s)$ and $q_\pi(s, \alpha)$ as parameterized functions (with fewer parameters than states) and adjust the parameters to better match the observed returns

Recursive Relationship for State-value Function $v_\pi(s)$

- Bellman Equation:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}\,|S_t = s] = \sum_\alpha \pi(\alpha|s) \sum_{s'} \sum_r p(s', r|s, \alpha)\,[r + \gamma \mathbb{E}_\pi\big[G_{t+1}\,|S_{t+1} = s'\big]] =$$
$$\sum_\alpha \pi(\alpha|s) \sum_{s',r} p(s', r|s, \alpha)\,[r + \gamma v_\pi(s')], \forall s \in \mathcal{S}$$

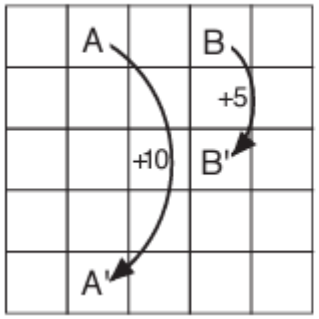
- Expresses a relationship between the value of a state and the values of its successor states
- Each open circle: state
- Each solid circle: state–action pair
- Starting from state s, the agent could take any of some set of actions (3) based on its policy π
- From each of these, the environment could respond with one of several next states,s'(2), along with a reward r depending on its dynamics given by the function p
- The Bellman equation averages over all the possibilities, weighting each by its probability of occurring
- The value of the start state must equal the (discounted) value of the expected next state, plus the reward expected along the way



Backup diagram for v_π

Gridworld Example

- Rectangular gridworld representation of a simple finite MDP
- Cells of the grid→ states
- Actions: north, south, east, west
- Actions that will take the agent off the grid leave its location unchanged, and give a reward -1
- Other actions give a reward 0
- Actions that move the agent out of A and B give a different reward, e.g., A→A´ (+10) and B→B´ (+5)
- Agent selects all actions with equal probability



3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

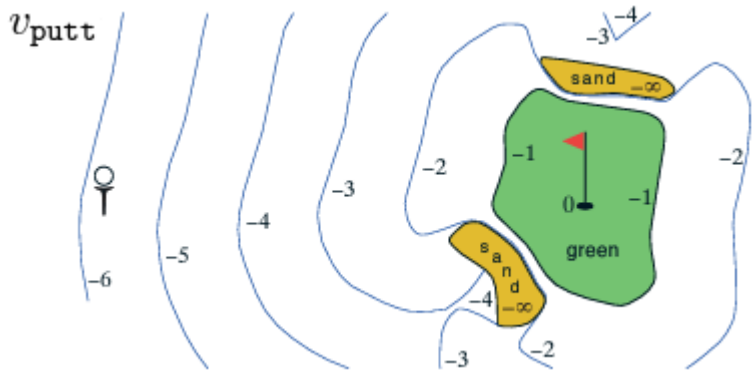
$\pi(s)$ (for $\gamma = 0.9$)

´ (close to the edge)

´, which has a positive value

Golf Example

- RL task: playing a hole of golf
- Penalty (negative reward) of -1 for each stroke until we hit the ball into the hole
- The state is the location of the ball
- The value of a state is the negative of the number of strokes to the hole from that location
- Actions: which club we select → putter or driver
- $v_{putt}(s)$: state-value function for the policy that always uses the putter
- Terminal state (in the hole): value 0
- Anywhere on the green, we can make a putt (value -1)
- Off the green we cannot reach the hole by putting, and the value is greater



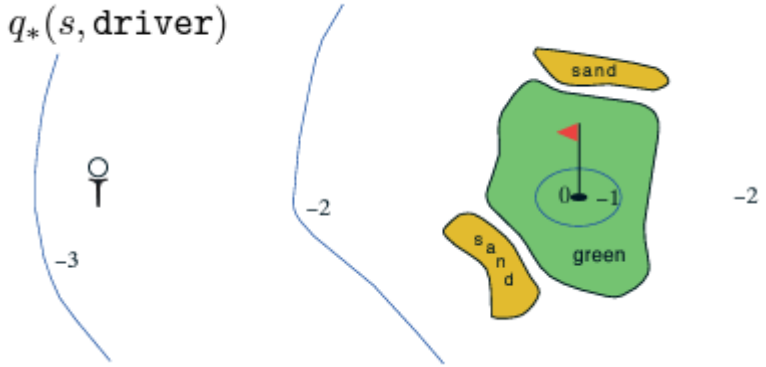
Optimal Policies and Optimal Value Functions

- Solving a reinforcement learning task: find a policy that achieves a lot of reward over the long run
- Finite MDPs: precisely define an optimal policy
- Value functions define a partial ordering over policies
- $\pi \geq \pi' \leftrightarrow v_\pi(s) \geq v_{\pi'}(s)$
- There is always at least one policy that is better than or equal to all other policies: optimal policy
- There can be many optimal policies: π^*
- Optimal state-value function: $v^*(s) = \max_\pi v_\pi(s)$
- Optimal action-value function: $q^*(s, \alpha) = \max_\pi q_\pi(s, \alpha)\,, \forall \alpha \in \mathcal{A}(s), s \in \mathcal{S}$
- For the state–action pair (s, α), the optimal action-value function gives the expected return for taking action α in state s and there after following an optimal policy:

$$q^*(s, \alpha) = \mathbb{E}\left[R_{t+1} + \gamma v^*(S_{t+1}) | S_t = s, \mathrm{A}_t = \alpha\right]$$

Optimal Value Function for Golf

- These are the values of each state if we first play a stroke with the driver and afterward select either the driver or the putter
- We can reach the hole in one shot using the driver only if we are already very close→ -1 area
- If we have two strokes, we can reach the hole from much farther away→ -2 area → we don't have to drive all the way to within the small -1 area, but only to anywhere on the green – from there we can use the putter
- The optimal action-value function gives the values after committing to a particular first action (driver), but afterward using whichever actions are best (driver/putter)
- The -3 area is still farther out and includes the starting tee
- From the tee, the best sequence of actions is two drives and one putt, sinking the ball in three strokes



LECTURE 8

Reinforcement Learning

Finite Markov Decision Processes

Bellman Optimality Equation

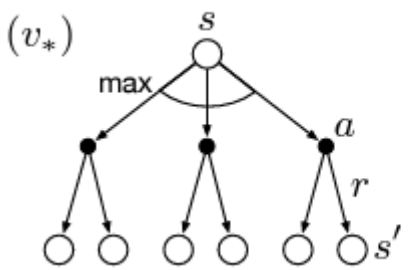
- Expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state

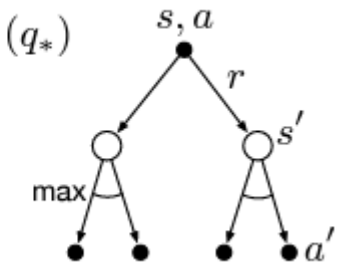
$$v^*(s) = \max_{\alpha \in \mathcal{A}(s)} q_{\pi^*}(s,\alpha) = \max_{\alpha \in \mathcal{A}(s)} \mathbb{E}_{\pi^*}[G_t | S_t = s, A_t = \alpha] = \max_{\alpha \in \mathcal{A}(s)} \mathbb{E}_{\pi^*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = \alpha] = \max_{\alpha \in \mathcal{A}(s)} \mathbb{E}_{\pi^*}[R_{t+1} + \gamma v^*(S_{t+1}) | S_t = s, A_t = \alpha] = \max_{\alpha \in \mathcal{A}(s)} \sum_{s',r} p(s', r | s, \alpha)[r + \gamma v^*(s')]$$

$$q^*(s,\alpha) = \mathbb{E}\left[R_{t+1} + \gamma \max_{\alpha'} q^*(S_{t+1}, \alpha') | S_t = s, \mathrm{A}_t = \alpha\right] = \sum_{s',r} p(s', r | s, \alpha)\left[r + \gamma \max_{\alpha'} q^*(s', \alpha')\right]$$

Backup diagrams for Bellman equality equation

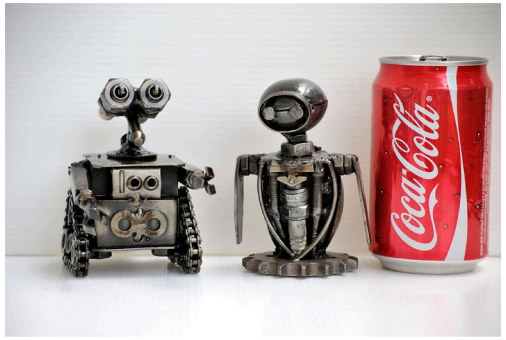
The maximum over a choice is taken rather than the expected value given some policy





Analyzing Bellman Optimality Equation

- Finite MDPs: Bellman optimality equation $v^*(s)$ has a unique solution independent of the policy
- Bellman optimality equation → system of n equations with n unknowns, if we have n states
- If the dynamics p of the environment are known → solve the system of equations → determine v^* → determine the optimal policy
- For each state s, there will be one or more actions at which the maximum is obtained in the Bellman optimality equation
- Any policy (greedy policy) that assigns nonzero probability only to these actions is an optimal policy
- Similarly, we can determine the q^*
- With q^*, the agent does not even have to do a one-step-ahead search: for any states, it can simply find any action that maximizes $q^*(s, \alpha)$
- The optimal action value function allows optimal actions to be selected without having to know anything about possible successor states and their values, i.e., without having to know anything about the environment's dynamics.



Bellman Optimality Equations

- High→h
- Low→l
- Search→s
- Wait→w
- Recharge→re
- There is exactly one pair of numbers $v^*(h)$, $v^*(l)$ that simultaneously satisfy these two nonlinear equations

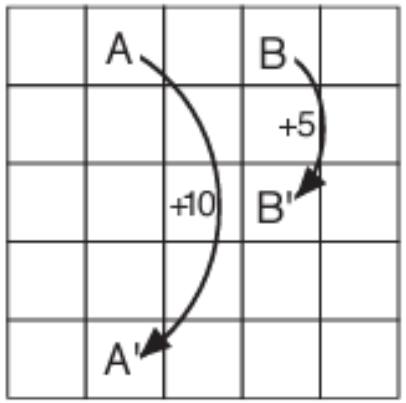
$$
\begin{aligned}
&v^*(h) \\
&= \max\{p(h|h,s)[r(h,s,h)+\gamma v^*(h)] + p(l|h,s)[r(h,s,l)+\gamma v^*(l)], \\
&p(h|h,w)[r(h,w,h)+\gamma v^*(h)] + p(l|h,w)[r(h,w,l)+\gamma v^*(l)]\} \\
&= \max\{\alpha[r_s+\gamma v^*(h)] + (1-\alpha)[r_s+\gamma v^*(l)],\ 1[r_w+\gamma v^*(h)] + 0[r_w+\gamma v^*(l)]\} \\
&= \max\{r_s+\gamma[\alpha v^*(h)+(1-\alpha)v^*(l)],\ r_w+\gamma v^*(h)\}
\end{aligned}
$$

$$
v^*(l) = \max\{\beta r_s - 3(1-\beta) + \gamma[(1-\beta)v^*(h)+\beta v^*(l)],\ r_w+\gamma v^*(l),\ \gamma v^*(h)\},
$$

$$
\forall r_s, r_w, \alpha, \beta, \gamma, 0 \le \alpha, \beta, \gamma \le 1
$$

s	a	s'	$p(s'\|s,a)$	$r(s,a,s')$
high	search	high	α	r_{search}
high	search	low	$1-\alpha$	r_{search}
low	search	high	$1-\beta$	-3
low	search	low	β	r_{search}
high	wait	high	1	r_{wait}
high	wait	low	0	-
low	wait	high	0	-
low	wait	low	1	r_{wait}
low	recharge	high	1	0
low	recharge	low	0	-

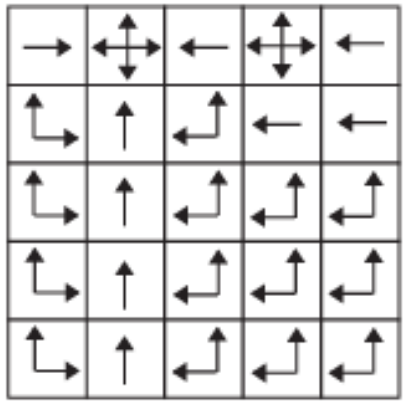
Revisit the Gridworld Problem



Gridworld

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

v_*



π_*

Optimal value function

Difficulties in solving the Bellman Optimality Equations

- Exhaustive search: looking ahead at all possibilities, computing their probabilities of occurrence and their desirabilities in terms of expected rewards.
- Assumptions rarely true in practice
  - We accurately know the dynamics of the environment
  - We have enough computational resources to complete the computation of the solution
  - The Markov property
- Example: backgammon game→ 10^{20} states → computation **XXX**
- **In RL one has to settle for approximate solutions**
- Methods to approximately solve the Bellman Optimality Equations:
  - Heuristic search methods
  - Dynamic programming (DP) methods

Optimality and Approximation

- An agent that learns an optimal policy has done very well, but in practice this rarely happens
- Optimal policies can be generated only with extreme computational cost
- Even if we have a complete and accurate model of the environment's dynamics, it is usually not possible to simply compute an optimal policy by solving the Bellman optimality equation
  - Example: board games, e.g., chess, are a tiny fraction of human experience, yet large, custom designed computers still cannot compute the optimal moves !

Critical Aspects

- Computational power
- Available memory
  - A large amount of memory is often required to build up approximations of value functions, policies, and models
  - In tasks with small, finite state sets, it is possible to form these approximations using arrays or tables with one entry for each state (or state–action pair) → tabular case → tabular methods.
  - Not applicable in cases with many states → functions must be approximated, using some sort of more compact parameterized function representation
- Key property that distinguishes RL from other approaches to approximately solving MDPs: online nature of RL makes it possible to approximate optimal policies in ways that put more effort into learning to make good decisions for **frequently** encountered states, at the expense of less effort for **infrequently** encountered states

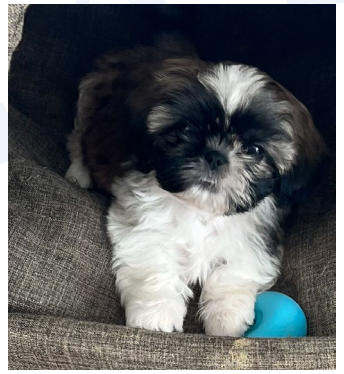
Let's summarize!

- RL is about learning from interaction how to behave in order to achieve a goal
- The RL agent and its environment interact over a sequence of discrete timesteps
- The actions are the choices made by the agent
- The states are the basis for making the choices
- The rewards are the basis for evaluating the choices
- Everything inside the agent is completely known and controllable by the agent
- Everything outside is incompletely controllable but may or may not be completely known
- A policy is a stochastic rule by which the agent selects actions as a function of states
- The agent's objective is to maximize the amount of reward it receives over time
- An RL setup with well defined transition probabilities → Markov Decision Process (MDP)
- A finite MDP is an MDP with finite state, action, and rewardsets

Keep going!

- The return is the function of future rewards that the agent seeks to maximize (in expected value)
- The undiscounted formulation is appropriate for episodic tasks → agent–environment interaction breaks naturally into episodes
- The discounted formulation is appropriate for continuing tasks, in which the interaction does not naturally break into episodes but continues without limit
- A policy's value functions (v_π and q_π) assign to each state, or state–action pair, the expected return from that state, or state–action pair, given that the agent uses the policy.
- The optimal value functions (v^* and q^*) assign to each state, or state–action pair, the largest expected return achievable by any policy
- A policy whose value functions are optimal is an optimal policy
- The optimal value functions for states and state–action pairs are unique for a given MDP
- In problems of complete knowledge, the agent has a complete and accurate model of the environment's dynamics
- In problems of incomplete knowledge, a complete and perfect model of the environment is not available
- Even if the agent has a complete and accurate environment model, the agent is typically unable to perform enough computation per time step to fully use it
- The memory available is also an important constraint
- In RL, we are very much concerned with cases in which optimal solutions cannot be found but must be approximated in some way

I still don’t know…



Biscottini

From MDPs to RL

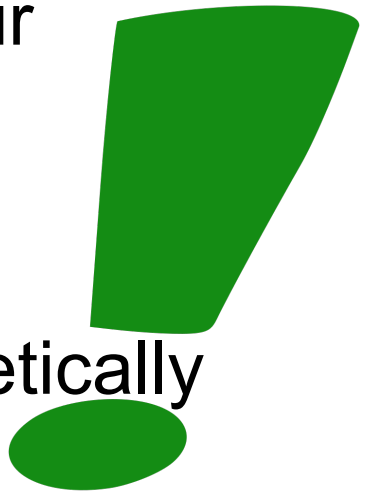
- RL problem is deeply indebted to the idea of MDPs from the field of optimal control
- MDPs and the RL problem are only weakly linked to traditional learning and decision-making problems in AI
- MDPs are also studied under the heading of stochastic optimal control, where adaptive optimal control methods are most closely related to RL
- The theory of MDPs evolved from efforts to understand the problem of making sequences of decisions under uncertainty, where each decision can depend on the previous decisions and their outcomes → theory of multistage decision processes (sequential decision processes)

Reinforcement Learning

Dynamic Programming

Dynamic Programming

- Dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as an MDP
- Classical DP algorithms are of limited utility in RL both because of their assumption of a perfect model and because of their great computational expense,
- Why do they learn about them???



Important theoretically

LECTURE 9

Reinforcement Learning

Dynamic Programming

Set the Scene

- Finite MDP: state $\mathcal{S}$, action $\mathcal{A}$ and reward $\mathcal{R}$ sets are finite
- Dynamics are given by a set of probabilities $p(s', r|s, \alpha), \forall s \in \mathcal{S}, \alpha \in \mathcal{A}(s), r \in \mathcal{R}$, and $s \in \mathcal{S}^+$ ($\mathcal{S}^+$is $\mathcal{S}$ plus a terminal state if the problem is episodic)
- DP ideas can be applied to problems with continuous state and action spaces
- A common way of obtaining approximate solutions for tasks with continuous states and actions is to quantize the state and action spaces and then apply finite-state DP methods
- The key idea of DP and RL: use of value functions to organize and structure the search for good policies
- How DP can be used to compute the value functions?
- We can obtain optimal policies once we have found

$$v^*(s) = \max_{\alpha \in \mathcal{A}(s)} q_{\pi^*}(s, \alpha) \quad = \max_{\alpha \in \mathcal{A}(s)} \mathbb{E}\left[R_{t+1} + \gamma v^*(S_{t+1}) | S_t = s, A_t = \alpha\right] = \max_{\alpha \in \mathcal{A}(s)} \sum_{s', r} p(s', r|s, \alpha)[r + \gamma v^*(s')]$$

$$q^*(s, \alpha) = \mathbb{E}\left[R_{t+1} + \gamma \max_{\alpha'} q^*(S_{t+1}, \alpha') | S_t = s, \mathrm{A}_t = \alpha\right] = \sum_{s', r} p(s', r|s, \alpha)\left[r + \gamma \max_{\alpha'} q^*(s', \alpha')\right]$$

- DP algorithms are obtained by turning Bellman equations into update rules for improving approximations of the desired value functions

Policy Evaluation – Prediction

- How to compute the state-value function v_π for an arbitrary policy π? →prediction problem → policy evaluation

$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}\,|S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] = \sum_\alpha \pi(\alpha|s) \sum_{s',r} p(s', r|s, \alpha)\,[r + \gamma v_\pi(s')], \forall s \in \mathcal{S}$

- $\pi(\alpha|s)$: probability of taking action α in state s under policy π
- The existence and uniqueness of v_π are guaranteed as long as either $\gamma < 1$ or eventual termination is guaranteed from all states under the policy π
- If the environment's dynamics are completely known→ $v_\pi(s) = \sum_\alpha \pi(\alpha|s) \sum_{s',r} p(s', r|s, \alpha)\,[r + \gamma v_\pi(s')]$: system of $|\mathcal{S}|$ linear equation with $|\mathcal{S}|$ variables
- **ITERATIVE** solution methods

Iterative Solution Methods

- Consider a sequence of approximate value functions $v_0, v_1, v_2, ...$, each mapping $\mathcal{S}^+ \rightarrow \mathbb{R}$
- The initial approximation v_0 is chosen arbitrarily (terminal state must be given value 0), and each successive approximation is obtained by using the Bellman equation for v_π
- Update rule – **Iterative Policy Evaluation**:

$$v_{k+1}(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s]$$

$$= \sum_\alpha \pi(\alpha|s) \sum_{s',r} p(s', r|s, \alpha) \left[r + \gamma v_k(s')\right], \forall s \in \mathcal{S}$$

Iterative Policy Evaluation

- To produce v_{k+1} from v_k → iterative policy evaluation applies the same operation to each state s
- It replaces the old value of s with a new value obtained from the old values of the successor states of s, and the expected immediate rewards, along all the one-step transitions possible under the policy being evaluated → expected update
- Each iteration of iterative policy evaluation updates the value of every state once to produce the new approximate value function v_{k+1}
- All the updates done in DP algorithms are called expected updates because they are based on an expectation over all possible next states rather than on a sample next state

How do we design an Iterative Policy Evaluation Algorithm?

- 1st way: use two arrays, one for the old values v_k, and one for the new values v_{k+1} (**Asynchronous**).
  - The new values can be computed one by one from the old values without the old values being changed
- 2nd way: use one array and update the values "in place" → with each new value immediately overwriting the old one (**Synchronous**).
  - Depending on the order in which the states are updated: sometimes new values are used instead of old ones
  - In-place algorithm converges to v_π usually faster than the two-array version, because it uses new data as soon as they are available

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

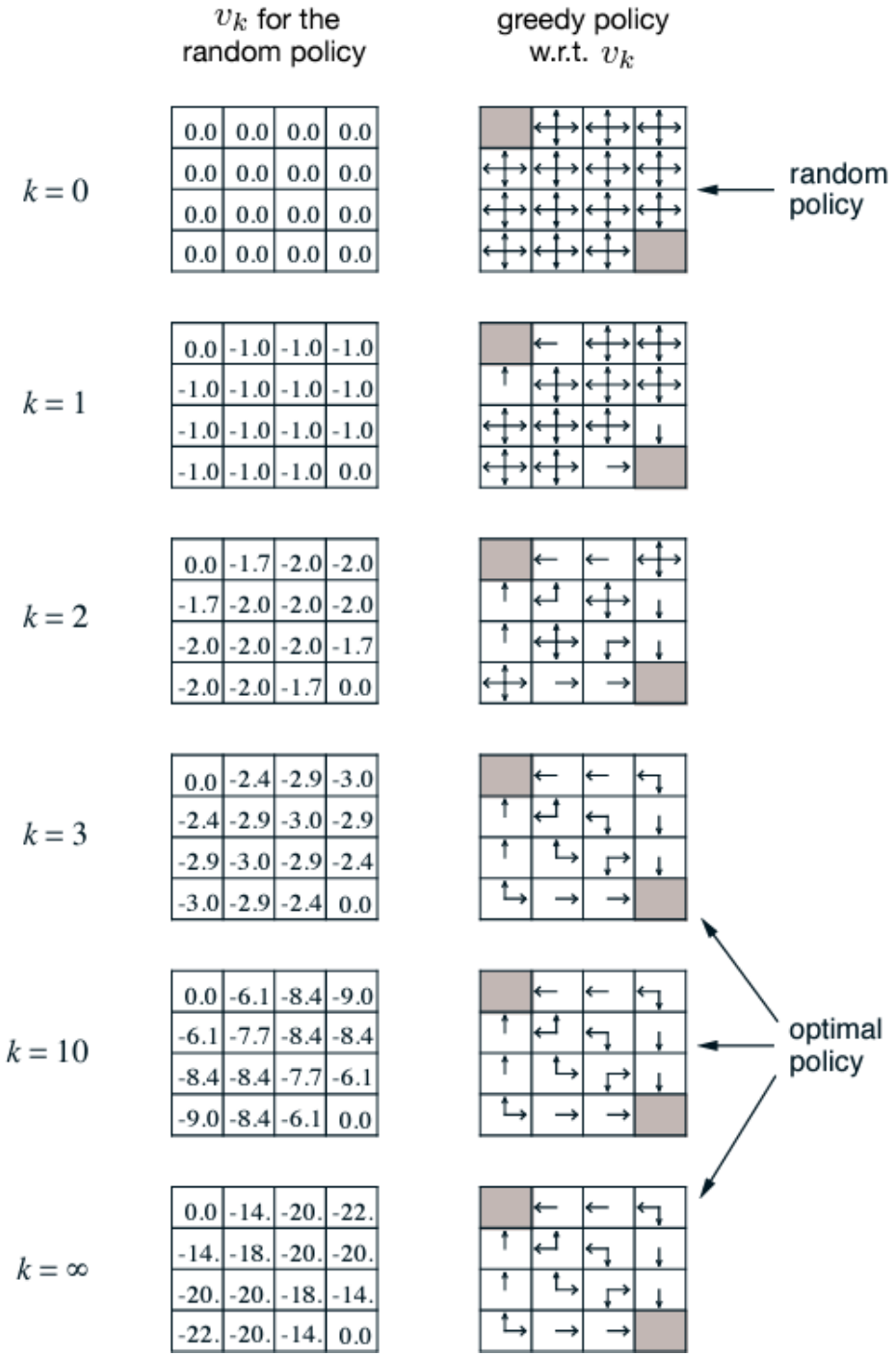
Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

4×4 Gridworld Example

actions

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

- States: $\mathcal{S}=\{1,2,\ldots,14\}$
- Actions: $\mathcal{A}=\{up, down, right, left\}$
- Actions that would take the agent off the grid leave the state unchanged
- The reward is -1 on all transitions until the terminal state is reached
- Examples: $p(6,-1|5,right)=1, p(7,-1|7,right)=1$, $p(10,r|5,right)=0$
- The agent follows the equiprobable random policy, i.e., all actions equally likely



Can we do better?

- Our reason for computing the value function for a policy is to help find better policies
- Suppose we have determined the value function v_π for an arbitrary deterministic policy π
- Can we choose an action $\alpha \neq \pi(s)$ at a state s? (not follow the existing policy π)??
- Would it be better or worse to change policy π while at a state s?
- To answer this question, determine: $q_\pi(s,\alpha) = \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) \,|S_t = s, \mathrm{A}_t = \alpha\right] = \sum_{s',r} p(s', r|s, \alpha)[r + \gamma v_\pi(s')]$
  - If $q_\pi(s, \pi'(s)) \geq v_\pi(s)$: it is better to select a once in s and thereafter follow π' than it would be to follow π all the time

Policy Improvement Theorem

- Let π and π' be any pair of deterministic policies such that $\forall s \in \mathcal{S}$: $q_\pi(s, \pi'(s)) \geq v_\pi(s)$
- Then the policy π' must be as good as, or better than, π, i.e., $\forall s \in \mathcal{S}: v_{\pi'}(s) \geq v_\pi(s)$
- Similarly, for the strict inequality: $q_\pi(s, \pi'(s)) > v_\pi(s) \leftrightarrow v_{\pi'}(s) > v_\pi(s)$
- Let's take a closer look: an original deterministic policy π and a changed policy π', that is identical to π except $\pi'(s) = \alpha \neq \pi(s)$
  - For states other than s: $q_\pi(s, \pi'(s)) \geq v_\pi(s)$ holds because the two sides are equal
  - If $q_\pi(s, \alpha) \geq v_\pi(s)$, then the changed policy π' is indeed better than π

LECTURE 10

Reinforcement Learning

Dynamic Programming

Policy Improvement Theorem

- Let π and π' be any pair of deterministic policies such that $\forall s \in \mathcal{S}$: $q_\pi(s, \pi'(s)) \geq v_\pi(s)$
- Then the policy π' must be as good as, or better than, π, i.e., $\forall s \in \mathcal{S}: v_{\pi'}(s) \geq v_\pi(s)$
- Similarly, for the strict inequality: $q_\pi(s, \pi'(s)) > v_\pi(s) \leftrightarrow v_{\pi'}(s) > v_\pi(s)$
- Let's take a closer look: an original deterministic policy π and a changed policy π', that is identical to π except $\pi'(s) = \alpha \neq \pi(s)$
  - For states other than s: $q_\pi(s, \pi'(s)) \geq v_\pi(s)$ holds because the two sides are equal
  - If $q_\pi(s, \alpha) \geq v_\pi(s)$, then the changed policy π' is indeed better than π

Proof

$$v_\pi(s) \le q_\pi\big(s, \pi'(s)\big)$$

$$= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = \pi'(s)]$$

$$= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s\ ]$$

$$\le \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_\pi\big(\mathrm{S_{t+1}}, \pi'(S_{t+1})\big) | S_t = s\ ]$$

$$= \mathbb{E}_{\pi'}[R_{t+1} + \gamma \mathbb{E}[R_{t+2} + \gamma v_\pi(S_{t+2}) | S_{t+1}, A_{t+1} = \pi'(S_{t+1})] | S_t = s\ ]$$

$$= \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2\ R_{t+3} + \gamma^3 v_\pi(S_{t+3}) | S_t = s\ ] = \cdots \le \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2\ R_{t+3} + \gamma^3\ R_{t+4} + \cdots | S_t = s\ ]$$

$$= v_{\pi'}(s)$$

Extension to All States

- Given a policy and its value function, we can easily evaluate a change in the policy at a **single state** to a particular action
- Extension to **all states**: selecting at each state the action that appears best according to $q_\pi(s, \alpha)$
- Policy update:

$$\pi'(s) = \text{argmax}_\alpha q_\pi(s, \alpha)$$

$$= \text{argmax}_\alpha \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = \alpha]$$

$$= \text{argmax}_\alpha \sum_{s',r} p(s', r | s, \alpha)[r + \gamma v_\pi(s')]$$

- π': greedy policy → takes the action that looks best in the short term
- **Policy improvement**: the process of making a new policy that improves on an original policy, by making it greedy with respect to the value function of the original policy

Policy Improvement and Optimal Policy

- Policy improvement must give a strictly better policy except when the original policy is already **optimal**
  - Suppose the new greedy policy π' is as good as, but not better than, the old policy π → $\boldsymbol{v_\pi = v_{\pi'}}$ →

    $v_{\pi'}(s) = \max_\alpha \mathbb{E}\left[R_{t+1} + \gamma v_{\pi'}(S_{t+1}) | S_t = s, \mathrm{A}_t = \alpha\right]$

    = $\max_\alpha \sum_{s',r} p(s', r|s, \alpha)[r + \gamma v_{\pi'}(s')]$ → Bellman optimality equation,

    thus, $v_{\pi'}$ must be v^*, and both π and π' must be **optimal policies**

Stochastic Policies

- A stochastic policy π specifies probabilities $\pi(a|s)$ for taking each action a in each state s
- The policy improvement theorem carries through as stated for the stochastic case
- If there are several actions at which the maximum is achieved (ties!), then in the stochastic case we need not select a single action from among them
- Each maximizing action can be given a portion of the probability of being selected in the new greedy policy

Policy Iteration

- Once a policy π has been improved using v_π to yield a better policy π', we can compute $v_{\pi'}$ and improve it again to yield an even better π''
- Sequence of monotonically improving policies and value functions

$$\pi_0 \rightarrow_{\mathrm{E}} v_{\pi_0} \rightarrow_{\mathrm{I}} \pi_1 \rightarrow_{\mathrm{E}} v_{\pi_1} \rightarrow_{\mathrm{I}} \pi_2 \rightarrow_{\mathrm{E}} \ldots \rightarrow_{\mathrm{I}} \pi^* \rightarrow_{\mathrm{E}} v_{\pi^*}$$

- $\rightarrow_{\mathrm{E}}$: policy evaluation
- $\rightarrow_{\mathrm{I}}$: policy improvement
- Each policy is guaranteed to be a strict improvement over the previous one
- Finite MDP (finite number of policies): this process must converge to an optimal policy and optimal value function in a finite number of iterations
- Each policy evaluation, itself an iterative computation, is started with the value function for the previous policy

Policy Iteration Algorithm (using iterative policy evaluation) for v^*

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

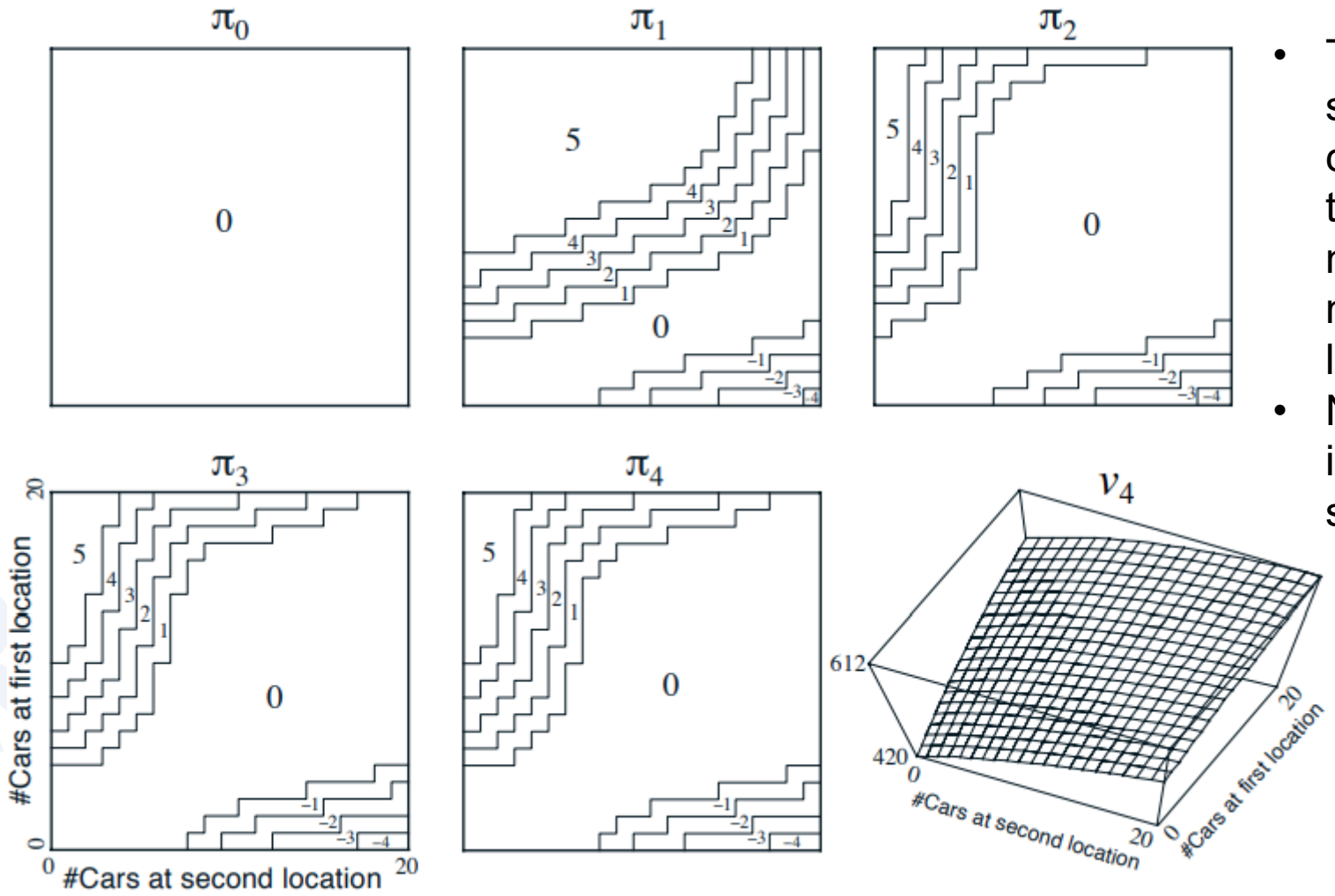
2. Policy Evaluation
   Repeat
      $\Delta \leftarrow 0$
      For each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\left[r + \gamma V(s')\right]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      $a \leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
      If $a \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return V and π; else go to 2

Jack's Car Rental

- Jack manages two locations for a nationwide car rental company
- Each day, some number of customers arrive at each location to rent cars
- If Jack has a car available, he rents it out for $10
- If he is out of cars at that location, then the business is lost
- Cars become available for renting the day after they are returned
- Jack can move cars between the two locations overnight, at a cost of $2 per car
- Number of cars requested and returned at each location are Poisson random variables $\Pr(n) = \frac{\lambda^n}{n!} e^{-\lambda}$ (λ: expected number)
- λ is 3 and 4 for rental requests at the first and second locations and 3 and 2 for returns
- There can be no more than 20 cars at each location and a maximum of 5 cars can be moved from one location to the other in one night
- Discount rate: $\gamma = 0.9$
- Finite MDP:
  - time steps → days
  - states → number of cars at each location at the end of the day
  - actions → net numbers of cars moved between the two locations overnight

Sequence of Policies – Policy Iteration



- The first five diagrams show, for each number of cars at each location at the end of the day, the number of cars to be moved from the first location to the second
- Negative numbers indicate transfers from the second location to the first

Value Iteration

- Drawback of policy iteration: each of its iterations involves policy evaluation → computationally costly due to requiring multiple sweeps through the state set
- If policy evaluation is done iteratively, then convergence exactly to v_π occurs only in the limit
- Must we wait for exact convergence, or can we stop short of that?
  - The policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration

    Special case: when policy evaluation is stopped after just one sweep (one backup of each state) → **value iteration** → Update rule:

    $v_{k+1}(s) = \max_\alpha \mathbb{E}\left[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, \mathrm{A}_t = \alpha\right]$

    $= \max_\alpha \sum_{s',r} p(s', r | s, \alpha)[r + \gamma v_k(s')]$
  - Value iteration formally requires an infinite number of iterations to converge exactly to v^*
  - Value iteration effectively combines, in each of its sweeps, one sweep of policy evaluation and one sweep of policy improvement
  - How value iteration terminates??? → once the value function changes by only a small amount in a sweep

Value Iteration for estimating $\pi \approx \pi^*$

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π, such that
  $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$

Policy iteration vs value iteration

- Approach:
  - Value Iteration: In value iteration, you start with an arbitrary value function and iteratively update it until it converges to the optimal value function.
  - Policy Iteration: In policy iteration, you alternately perform policy evaluation and policy improvement steps until an optimal policy is found.
- Initialization:
  - Value Iteration: It typically starts with an initial estimate of the value function and iteratively refines it.
  - Policy Iteration: It starts with an initial policy, which can be arbitrary, and then improves the policy iteratively.
- Iterations:
  - Value Iteration: Value iteration performs a series of sweeps through the state space, updating the value function at each step. It continues until the change in the value function between iterations is smaller than a predefined threshold.
  - Policy Iteration: Policy iteration consists of two main steps: policy evaluation and policy improvement. It continues these steps iteratively until the policy converges to an optimal policy.
- Convergence:
  - Value Iteration: Value iteration usually converges faster, as it updates the value function directly in a single step by selecting the maximum expected return.
  - Policy Iteration: Policy iteration is often slower because it involves two separate steps: evaluating the current policy and then improving it.

LECTURE 11

Reinforcement Learning

Dynamic Programming

Policy iteration vs value iteration

- Approach:
  - Value Iteration: In value iteration, you start with an arbitrary value function and iteratively update it until it converges to the optimal value function.
  - Policy Iteration: In policy iteration, you alternately perform policy evaluation and policy improvement steps until an optimal policy is found.
- Initialization:
  - Value Iteration: It typically starts with an initial estimate of the value function and iteratively refines it.
  - Policy Iteration: It starts with an initial policy, which can be arbitrary, and then improves the policy iteratively.
- Iterations:
  - Value Iteration: Value iteration performs a series of sweeps through the state space, updating the value function at each step. It continues until the change in the value function between iterations is smaller than a predefined threshold.
  - Policy Iteration: Policy iteration consists of two main steps: policy evaluation and policy improvement. It continues these steps iteratively until the policy converges to an optimal policy.
- Convergence:
  - Value Iteration: Value iteration usually converges faster, as it updates the value function directly in a single step by selecting the maximum expected return.
  - Policy Iteration: Policy iteration is often slower because it involves two separate steps: evaluating the current policy and then improving it.

Policy iteration vs value iteration

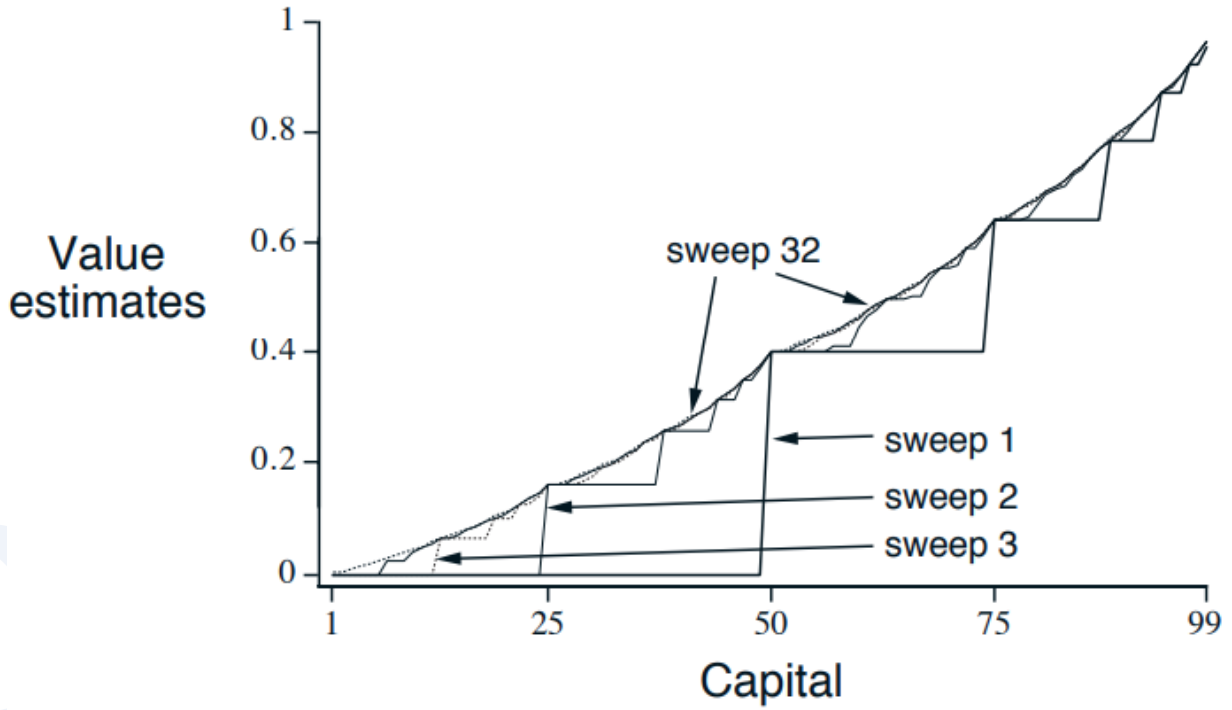
- Exploration vs. Exploitation:
  - Value Iteration: Value iteration inherently balances exploration and exploitation as it directly computes the value of each state-action pair.
  - Policy Iteration: Policy iteration focuses more on exploitation since it starts with a policy and refines it based on the current value function.
- Memory and Computation:
  - Value Iteration: Value iteration typically requires less memory and computation because it updates only the value function.
  - Policy Iteration: Policy iteration may require more memory and computation as it maintains both the value function and the policy.
- Final Solution:
  - Value Iteration: Value iteration provides both the optimal value function and the optimal policy as a result.
  - Policy Iteration: Policy iteration directly converges to the optimal policy but may not compute the optimal value function unless additional steps are taken.

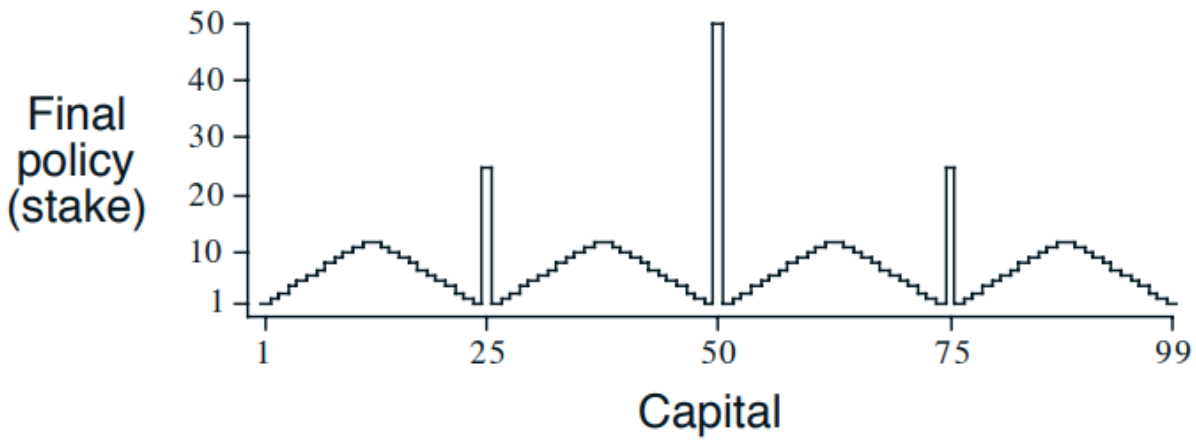
Gambler's Problem

- A gambler has the opportunity to make bets on the outcomes of a sequence of coin flips
- Heads: wins as many dollars as he has staked on that flip
- Tails: loses his stake
- The game ends when the gambler wins by reaching his goal of $100, or loses by running out of money
- On each flip, the gambler must decide what portion of his capital to stake, in integer numbers of dollars
- Undiscounted, episodic, finite MDP problem: state $s \in \{1,2,\dots,99\}$: gambler's capital, actions $\alpha \in \{0,1,\dots,\min(s,100-s)\}$: stakes
- Reward: 0 on all transitions except those on which the gambler reaches his goal → $+1$
- State-value function: probability of winning from each state
- Policy: mapping from levels of capital to stakes
- Optimal policy maximizes the probability of reaching the goal

Value Function and Final Policy

- p_h: probability of the coin coming up heads
- If p_h known → entire problem is known and it can be solved by value iteration
- Changes in the value function over successive sweeps of value iteration, and the final policy found, for $p_h = 0.4$
- Final policy → optimal policy, but not unique
- There is a whole family of optimal policies, all corresponding to ties for the argmax action selection with respect to the optimal value function





Asynchronous Dynamic Programming

- Major drawback to the DP methods: involve operations over the entire state set of the MDP → they require sweeps of the state set
- If the state set is very large, then even a single sweep can be prohibitively expensive (remember backgammon game 10^{20} states)
- Asynchronous DP algorithms: in-place iterative DP algorithms that are not organized in terms of systematic sweeps of the state set
  - Update the values of states in any order whatsoever, using whatever values of other states happen to be available
  - To converge correctly, an asynchronous algorithm must continue to update the values of all the states
  - Provide great flexibility in selecting states to which update operations are applied
  - It is possible to intermix policy evaluation and value iteration updates to produce a kind of asynchronous truncated policy iteration

A solution … but

- Avoiding sweeps does not necessarily mean that we can get away with less computation
- It just means that an algorithm does not need to get locked into any hopelessly long sweep before it can make progress improving a policy
- We can try to order the updates to let value information propagate from state to state in an efficient way
- Some states may not need their values updated as often as others
- We might even try to skip updating some states entirely if they are not relevant to optimal behavior
- Easier to intermix computation with real-time interaction
  - We can apply updates to states as the agent visits them
  - Focus DP algorithm's updates onto parts of the state set that are most relevant to the agent

Is Policy Iteration a necessary method?

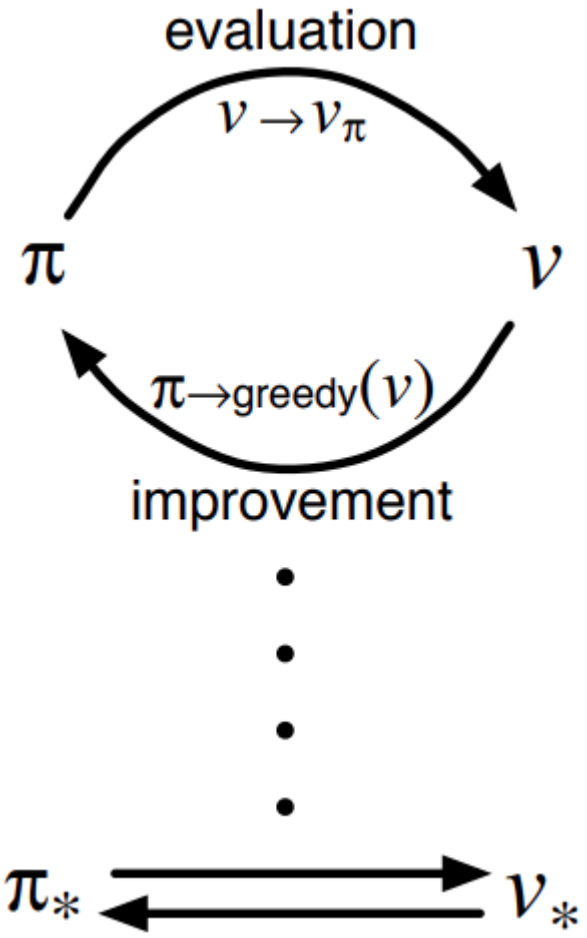
- Policy iteration consists of two simultaneous, interacting processes:
  - Policy evaluation: making the value function consistent with the current policy
  - Policy improvement: making the policy greedy with respect to the current value function
- These two processes alternate, each completing before the other begins, but **this is not really necessary**
- As long as both processes continue to update all states, the ultimate result is typically the same: convergence to the optimal value function and an optimal policy

Generalized Policy Iteration (GPI)

- Generalized policy iteration (GPI): general idea of letting policy evaluation and policy improvement processes interact, independent of the granularity and other details of the two processes
- RL methods are well described as GPI
- If both the evaluation process and the improvement process stabilize, that is, no longer produce changes, then the value function and policy must be optimal.
  - The value function stabilizes only when it is consistent with the current policy
  - The policy stabilizes only when it is greedy with respect to the current value function
  - Both processes stabilize only when a policy has been found that is greedy with respect to its own evaluation function → Bellman optimality equation holds → the policy and the value function are optimal

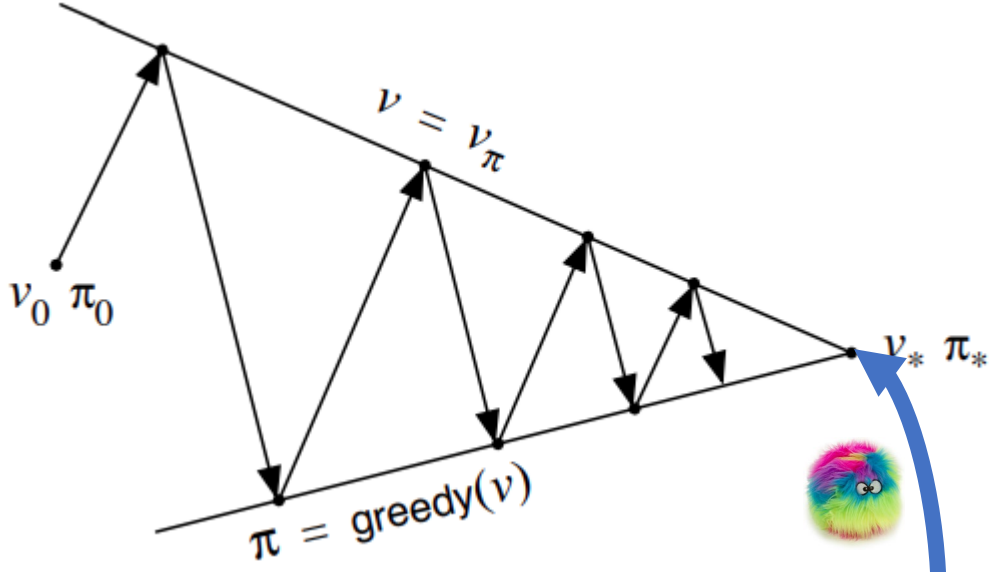
Compete & Cooperate

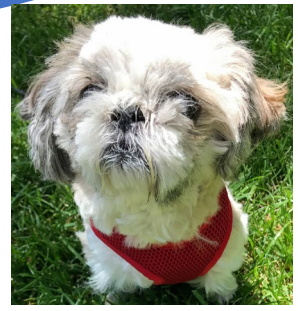
- The evaluation and improvement processes in GPI can be viewed as both competing and cooperating
  - Compete: pull in opposing directions →
    - making the policy greedy with respect to the value function typically makes the value function incorrect for the changed policy
    - making the value function consistent with the policy typically causes that policy no longer to be greedy
  - Cooperate: interact to find a single joint solution: the optimal value function and an optimal policy



But finally GPI converges…

- Think the interaction between the evaluation and improvement processes in GPI in terms of two constraints or goals: two lines in 2D space
- Each process drives the value function or policy toward one of the lines representing a solution to one of the two goals
- The goals interact because the two lines are not orthogonal
- Driving directly toward one goal causes some movement away from the other goal
- Inevitably, the joint process is brought closer to the overall goal of optimality





Efficiency of Dynamic Programming

- DP may not be practical for very large problems, but compared with other methods for solving MDPs, DP methods are actually quite efficient
- DP methods find an optimal policy is **polynomial time** in the number of states and actions
- If n and m denote the number of states and actions, a DP method takes a number of computational operations that is less than some polynomial function of n and m
- A DP method is guaranteed to find an optimal policy in polynomial time even though the total number of policies is m^n
- DP is exponentially faster than any direct search in policy space could be → direct search would have to exhaustively examine each policy to provide the same guarantee
- Linear programming methods can be used to solve MDPs → in some cases their worst-case convergence guarantees are better than those of DP methods
- Drawback: linear programming methods become impractical at a much smaller number of states than do DP methods → for the largest problems, only DP methods are feasible