HUR Group Crash Course

"Introduction to Markov Decision Processes (MDP)"

Pilwon Hur, PhD

Mechanical and Robotics Engineering, GIST

Aug 5, 2024

Contents

Contents

Agenda of the Course

- Overview
  1. Understanding Decision Processes: Markov Chains vs. MDP
  2. Examples of MC and MDP
  3. MC, MDP, POMDP, Reinforcement Learning
- Details of MDP
  1. Assumptions and Structures of MDP
     1. State
     2. State Transition Probability
     3. Action
     4. Policy
     5. Reward
     6. Discounting Factor
     7. Value
     8. Bellman Equation
     9. Dynamic Programming
  2. Solution of Bellman Equation
     1. Dynamic Programming: Value Iteration, Policy Iteration
     2. Monte Carlo
     3. Deel Learning
- Experiments with Grid World Examples

Understanding Decision Processes: MC vs. MDP (I)

Markov Chains (MC)

Markov Processes (or MC) are systems that transition from one state to another, where the probability of each transition only depends on the current state, not the sequence of events that preceded it. It's like a memoryless random walk where the next step is determined solely by where you are now, not how you got there.

Mathematical Definition

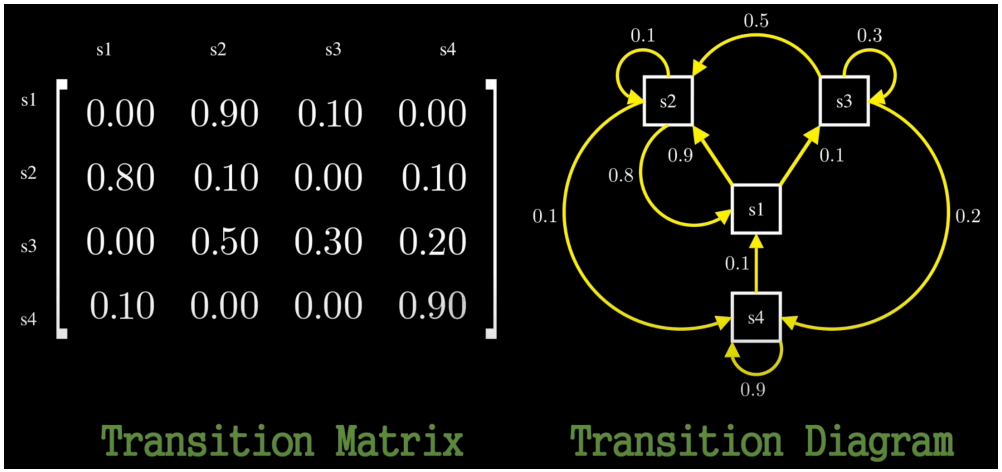
A Markov Chain is a sequence of random variables $X_1, X_2, X_3, \ldots$ with the Markov property:

$$P(X_{n+1} = x | X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n) = P(X_{n+1} = x | X_n = x_n)$$

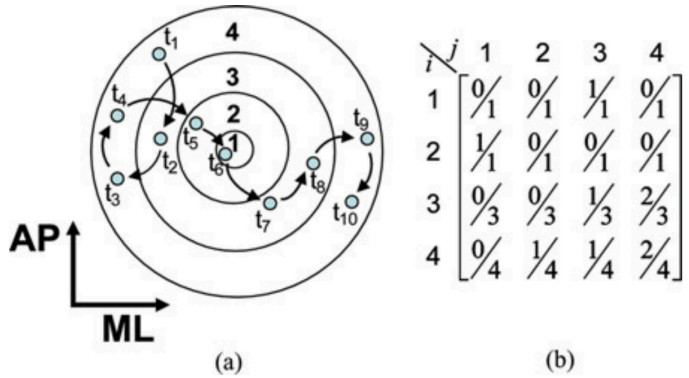
This property implies that future states depend only on the present state, not on the past states.

Understanding Decision Processes: MC vs. MDP (II)

- Components of MC
  1. State: The possible values that the system can be in.
  2. State Space: The set of all possible states.
  3. State Transition Probability: The probabilities of moving from one state to another. (e.g. Probability Transition Matrix)
  4. Initial State: The state in which the process starts. (e.g. Probability Distribution)



https://youtu.be/CIe869Rce2k



MC modeling for Human Standing COP
Hur et al., 2012, IEEE TBME
https://ieeexplore.ieee.org/document/6129490

Examples of MC

- Examples of MC
  1. Weather Prediction
     - States: Sunny, Cloudy, Rainy
     - Transition: The probability of tomorrow's weather depends only on today's.
  2. Board Games
     - States: Positions on a game board
     - Transition: Movement determined by dice rolls
  3. Customer Behavior
     - States: Browsing, Adding to Cart, Purchasing
     - Transition: The next action of a customer depends on their current action.

- 2D Random Walk in a Grid World
  1. State: 2D Coordinate (x, y)
  2. State Space: 9×9 discrete grid
  3. Transition Probability Matrix:

```
array([[0.3 , 0.7 , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.1 , 0.1 , 0.8 , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.1 , 0.1 , 0.8 , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.1 , 0.1 , 0.8 , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.45, 0.1 , 0.45, 0.  , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.8 , 0.1 , 0.1 , 0.  , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.8 , 0.1 , 0.1 , 0.  ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.8 , 0.1 , 0.1 ],
       [0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.7 , 0.3 ]])
```

- MC describes autonomous dynamics!
  1. Action is not involved.
  2. MDP involves with actions.

MC, MDP, POMDP, Reinforcement Learning

Aspect	**MC**	**MDP**	**POMDP**	**RL**
State Awareness	Full	Full	Partial	Full/Partial
Decision Making	No	Yes	Yes	Yes
Components	States, Transition Probabilities	States, Actions, Transition Probabilities, Rewards	States, Actions, Transition Probabilities, Rewards, Observations, Observation Probabilities	Agent, Environment, States, Actions, Rewards, Policy, Value Function
Use Cases	Weather prediction, Games	Robotics, Planning, Inventory management	Autonomous driving, Medical diagnosis	Game AI, Robotics, Recommender systems
Algorithms	None (just state transitions)	Value Iteration, Policy Iteration, Q-Learning	Value Iteration for POMDPs, Point-Based Value Iteration	Q-Learning, SARSA, Deep Q-Networks (DQN), Policy Gradient Methods

Contents

Introduction to MDP

Markov Decision Process (MDP)

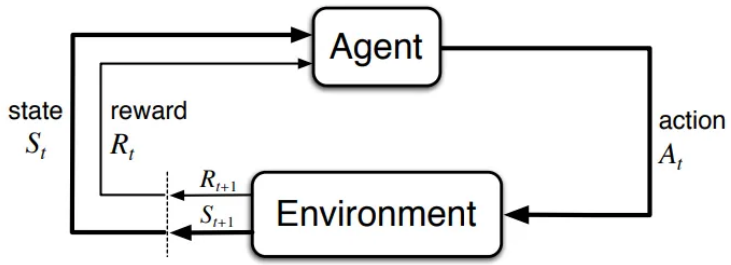
A Markov Decision Process (MDP) is a mathematical framework used for modeling decision making where an agent interacts with an environment. The agent makes decisions at different states, and each decision leads to a change in state, influenced by both the agent' s actions and some randomness in the environment. It helps in finding the best decision at each step to maximize rewards over time.

Components of MDP

An MDP is defined by a tuple (S, A, P, R, γ):
S: A set of states representing different situations in the environment.
A: A set of actions available to the agent.
$P(s'|s, a)$: Transition probability function that defines the probability of reaching state s' from state s after the agent takes action a
$R(s, a)$: Reward function that defines the immediate reward received after the agent transitions from state s to state s' due to action a
γ : Discount factor $(0 \leq \gamma < 1)$ representing the importance of future rewards.

Features of MDP

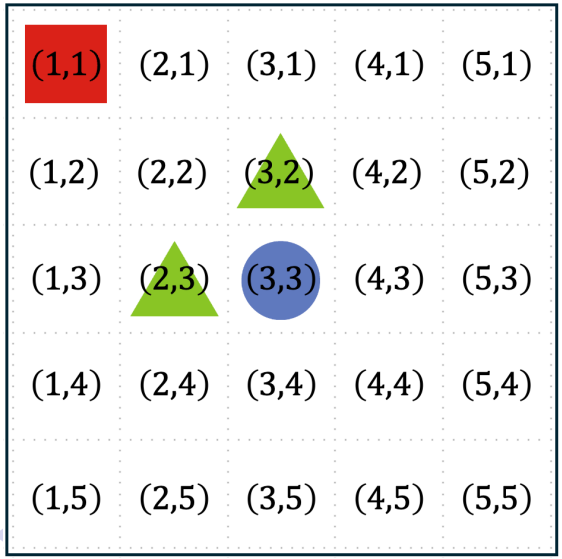
- Environment is modeled through probabilistic framework. Some known probability mass function (pmf) is (or may be) the basis for modeling.
- It consists of a finite set of discrete states.
- Here states does not contain any past statistics.
- Through well defined pmf a set of discrete sample data is created.
- For each environmental state, there is a finite set of possible action that may be taken by agent.
- Every time (or after) agent takes an action, a certain cost (or reward) and next state are incurred (or provided).
- An action can be selected by some policies.
- In MDP, state information is clearly provided by the environment whereas in POMDP, state information is partially provided via observation.



More on Components: State

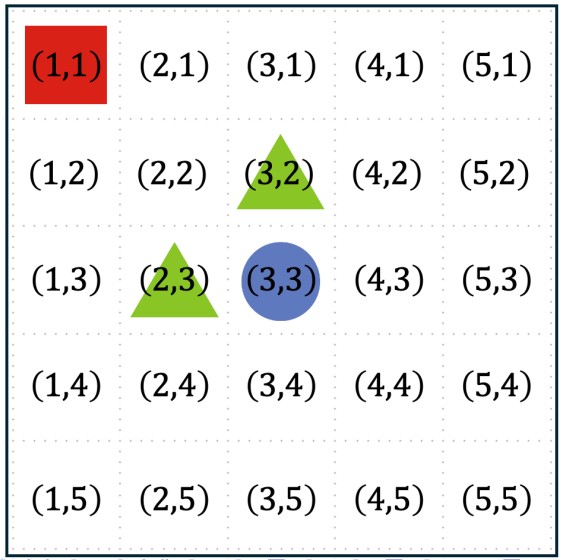
- State represents the current situation or configuration of the environment in which an agent operates.
- Complete Information: A state includes all necessary details about the environment that influence the agent's decision-making process.
- Observable States: The agent has complete information about the state of the environment.
- State (S_t) provides the basis for decision (i.e., action, A_t), transition (i.e., next state, S_{t+1}), and reward (R_{t+1}).
- State space can be either discrete or continuous (e.g., infinite or very large). We assume discrete state space in this lecture.

- State at time t: $S_t = (1, 1)$
- State at time $t + 1$: $S_{t+1} = (2, 1)$
- S_t in capital letter is a random variable.
- Small letter (s or $(1, 1)$) is an instance or an implementation.
- A failed process:
  $S_1(= (1, 1)), S_2(= (2, 1)), S_3(= (3, 1)), S_4(= (3, 2))$



More on Components: Action

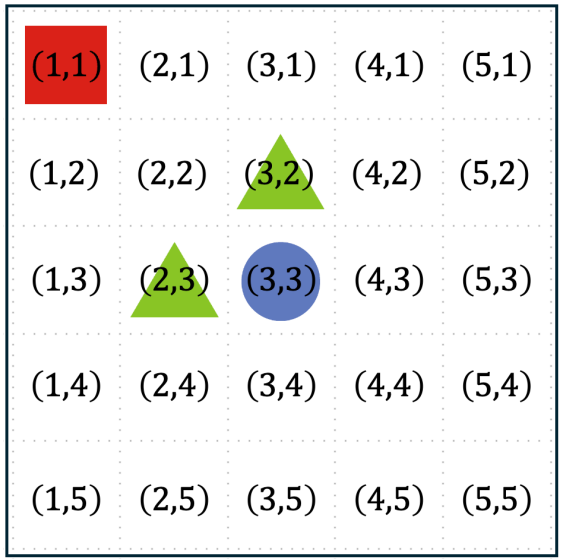
- Action is a decision or move that the agent makes at a given state. Actions determine how the agent interacts with the environment and influence the state transitions and rewards.
- Actions can lead to deterministic outcomes where the next state is certain, or stochastic outcomes where the next state is probabilistic.
- Set of actions can be either discrete/finite or continuous/infinite. We assume discrete state in this lecture.
- The policy (π) determines which action to take in each state.
- Once an action is taken, the environment provides next state and reward.
- Available actions: $A = \{up, down, left, right\}$
- Action at time t: $A_t = up$
- Action at time $t+1$: $A_{t+1} = down$
- A_t in capital letter is a random variable.
- Small letter (a or up) is an instance or an implementation.
- A bad (potentially, why?) actions:
  $A_1(= r), A_2(= r), A_3(= d)$



More on Components: State Transition Probability

- The state transition probability defines the likelihood of transitioning from one state to another given a specific action taken by the agent.
- Transition Probability Function: $P(S_{t+1} = s' | S_t = s, A_t = a)$
  $\rightarrow$ This represents the probability of moving to state s' given that the current state is s and the agent takes action a.
- It is very important to note that the Transition Probability is the required Model from the environment!

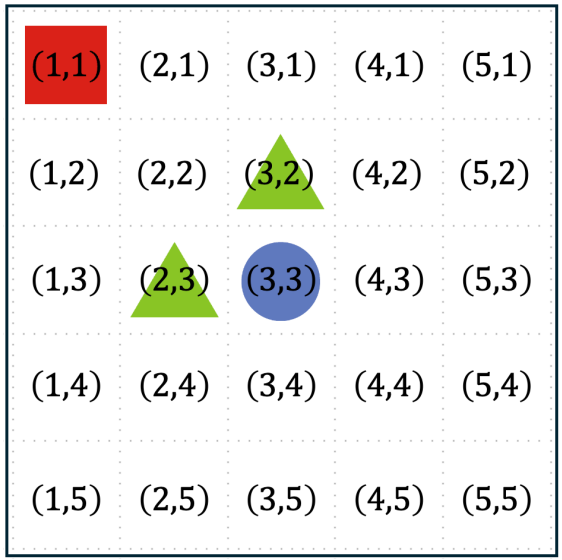
- $\sum_{s'} P(s'|s, a) = 1$
- $P\left(S_{t+1} = (3, 2) | S_t = (3, 1), A_t = d\right) = 1$ or < 1? Explain this!



More on Components: Reward

- A reward is a numerical value received by the agent after transitioning from one state to another as a result of an action. It quantifies the immediate benefit or cost of the action taken by the agent.
- Immediate Feedback: Rewards provide immediate feedback on the value of actions taken by the agent.
- Guidance for Learning: Rewards guide the agent in learning which actions are beneficial and should be reinforced.
- Influence on Policy: The reward function influences the agent's policy, which is the strategy for choosing actions to maximize long-term rewards.

- (Immediate) Reward: R_{t+1}
- (Expected) Reward Function: $r(s, a) = \mathbb{E}\left[R_{t+1} | S_t = s, A_t = a\right]$
- In MDP, agents want to maximize the total sum of the expected rewards during excursion.
- In the Grid World, how would you design rewards? (goal: blue, hazards: green)



More on Components: Discount Rate

- Discount Factor ($0 \leq \gamma < 1$) is a number that represents the importance of future rewards. It determines how much future rewards are worth compared to immediate rewards.
- Impact on Long-Term Rewards: A higher discount factor (close to 1) makes future rewards almost as important as immediate rewards, while a lower discount factor (close to 0) makes future rewards much less important.
- Balancing Immediate and Future Rewards: Helps in striking a balance between immediate and future rewards. This is crucial in environments where long-term planning is necessary.
- Stability of Value Function: Ensures the value function converges by appropriately weighing future rewards.
- Example
  1. Assume your current value is determined by the cumulative sum of your salary.
  2. What is your value when $\gamma = 1$?
  3. Is it more reasonable to have $\gamma < 1$?
  4. What action would you take to maximize your value?

More on Components: Return

Return, G_t

Return is the total accumulated reward that an agent receives from a starting state until the end of the episode. It considers all the actual rewards obtained along the trajectory.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

- Specific to Trajectory: The return is specific to a particular sequence of states, actions, and rewards.
- Depends on Actual Rewards: Calculated based on the actual rewards observed during the trajectory.

More on Components: Expected Return

Expected Return, $v(s)$

Expected return is the average return that an agent can expect to receive from a given state, following a particular policy. It is an expectation over all possible trajectories starting from that state. It is also called Value Function.

$$v(s) = \mathbb{E}\left[G_t \mid S_t = s\right]$$

- $v(s)$ is the expected return (value) starting from state s
- $\mathbb{E}\left[\cdot\right]$ denotes the expectation over all possible trajectories.
- Policy-Dependent: The expected return depends on the policy being followed.
- Averaged Over Trajectories: Calculated as the average return over all possible trajectories from a given state, considering the stochastic nature of the environment.

More on Components: Policy

Policy, π

Policy (π) is a strategy or plan that the agent follows to decide which action to take in each state. It maps states to actions, guiding the agent's behavior to achieve the best possible outcome.

- Deterministic Policy specifies a single action to take for each state.
  $\pi(s) = a$, meaning in state s, the agent always takes action a.
- Stochastic Policy specifies a probability distribution over actions for each state.
  $\pi(a|s)$, meaning the probability of taking action a in state s.
- State-Dependent: The policy depends on the current state of the environment.
- Adaptability: Policies can be updated based on the agent's experience to improve decision making.
- Guidance for Actions: Policies provide a clear guide on what actions to take, reducing uncertainty for the agent.
  $\rightarrow$ If an agent has an optimal policy, then agent will move to maximize the reward.

More on Components: Value Function

Value Function, $v^{\pi}(s)$

The value function $v^{\pi}(s)$ represents the expected return (total accumulated reward) when starting from state s and following a specific policy π.

- The value function can be put in a recursive form to express the value of a state as the immediate reward plus the discounted value of the next state. It captures the essence of dynamic programming.

$$
\begin{aligned}
v^{\pi}(s) &= \mathbb{E}\left[G_t \mid S_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \mid S_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \cdots) \mid S_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma v^{\pi}(S_{t+1}) \mid S_t = s\right]
\end{aligned}
$$

- This equation indicates that the value of state s under policy π is the expected reward R_{t+1} plus the discounted value of the next state S_{t+1}.

More on Components: Bellman (Expectation) Equation

Bellman (Expectation) Equation

The Bellman equation provides a recursive decomposition of the value function. It breaks down the value function into the immediate reward plus the discounted value of successor states, simplifying the problem of finding true value function given the specific policy.

- From the previous page, the expectation form of the recursive value function can be put into computable form.

$$
\begin{aligned}
v^{\pi}(s) &= \mathbb{E}_a \left[R_{t+1} + \gamma v^{\pi}(S_{t+1}) \mid S_t = s \right] \\
&= \sum_a \pi(a|s) \mathbb{E}_{s'} \left[r(s,a) + \gamma v^{\pi}(s') \right] \\
&= \sum_a \pi(a|s) \sum_{s'} P(s'|s,a) \left[r(s,a) + \gamma v^{\pi}(s') \right] \\
&= \sum_a \pi(a|s) \left[r(s,a) + \gamma v^{\pi}(s') \right]
\end{aligned}
$$

- The last equation holds only when dynamics is deterministic (i.e., state transition probability $= 1$).

More on Components: Bellman Optimality Equation

Bellman Optimality Equation

The Bellman expectation equation becomes the Bellman optimality equation if the given policy is optimal. The solution of the Bellman optimality equation is the optimal value function. The action from the optimal policy is the optimal control. Note that the optimal policy is NOT explicitly found from the Bellman optimality equation.

- From the previous page, the expectation form of the recursive value function can be put into computable form.

$$
\begin{aligned}
v^*(s) &= \max_{\pi} v^{\pi}(s) \\
&= \max_{a} \left[R_{t+1} + \gamma v^*(S_{t+1}) \mid S_t = s \right] \\
&= \max_{a} \sum_{s'} P(s'|s,a) \left[r(s,a) + \gamma v^*(s') \right] \\
&= \max_{a} \left[r(s,a) + \gamma v^*(s') \right]
\end{aligned}
$$

- The last equation holds only when dynamics is deterministic (i.e., state transition probability $= 1$).

More on Components: What to do next?

- Our goal is to find the optimal policy and the corresponding optimal value function.
- The environment should provide the transition probability $P(s' \mid s, a)$ and reward function $r(s, a)$.
- Optimal policy and optimal value function can be explicitly computed from the Bellman expectation equation.
- Alternatively, optimal value function can be directly computed from the Bellman optimality equation. Optimal policy is computed implicitly from the Bellman optimality euqation.
- How to solve the Bellman equation (i.e., $v^*(s), \pi^*$) matters!
  1. Exact solution with iterative algorithms $\rightarrow$ Dynamic Programming (Policy Iteration, Value Iteration)
  2. Approximate solution with model free random sampling $\rightarrow$ Reinforcement Learning (SARSA, Q-learning)
  3. Approximate solution with model free deep learning $\rightarrow$ Deep Reinforcement Learning (Deep SARSA, DQN, Policy Gradient)

Contents

Solution to Bellman Equation

Bellman Equation

A recursive equation that provides a way to calculate the value function, which represents the expected return from a given state, considering the optimal policy.

$$v^{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s'} P(s'|s,a) \left[r(s,a) + \gamma v^{\pi}(s') \right]$$

$$v^{*}(s) = \max_{a} \sum_{s'} P(s'|s,a) \left[r(s,a) + \gamma v^{*}(s') \right]$$

- Analytic approach
  1. Finding the analytic (e.g., closed form) solution of the Bellman equation is almost impossible due to nonlinearity, stochasticity, dimensionality.
- Iterative approach (e.g., Dynamic Programming)
  1. Iterative Approximation: Efficiently approximates the value function and policy through iterative updates, making it feasible for complex problems.
  2. Handling Large State Spaces: Effectively manages large or continuous state spaces without requiring an explicit analytic solution, making it suitable for real-world applications.
  3. Convergence Guarantees: Under certain conditions, guarantees convergence to the optimal value function and policy, ensuring robust solutions.

Iterative Approach

Bellman Operator

The Bellman operator $\mathcal{T}$ is an operator that applies the Bellman equation to a value function, producing a new value function.

$$\mathcal{T}v^{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s'} P(s'|s,a) \left[r(s,a) + \gamma v^{\pi}(s') \right]$$

- Bellman Operator is a contraction map (DIY). Due to Banach fixed-point theorem, $\mathcal{T}v_k^{\pi}(s)$ converges to $v_k^{\pi}(s)$ as $k \to \infty$ and it is unique.
- Therefore, for any random initial value function $v_0^{\pi}(s)$, applying the Bellman Operator guarantees the convergence, meaning the true value function satisfying the Bellman equation.
- Convergence is guaranteed for any discount rate $\gamma < 1$.
- Two iterative approaches will be convered.
  1. Policy Iteration
  2. Value Iteration

Policy Iteration

Policy Iteration

An iterative algorithm that alternates between policy evaluation and policy improvement to find the optimal policy.

$$v^{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s'} P(s'|s,a) \left[r(s,a) + \gamma v^{\pi}(s') \right]$$

input : $r(s,a)$, $P(s'|s,a)$ from environment
Initialize arbitrarily;
$\pi_0(s) = [0.25, 0.25, 0.25, 0.25]$ for all s;
$v^{\pi}(s) = 0$ for all s;
while *policy converging* **do**
  Policy Evaluation;
  while $v^{\pi}(s)$ *converging* **do**
    for s *in all states* **do**
      $v^{\pi}(s) = \sum_{a} (a|s) \sum_{s'} P(s'|s,a)[r(s,a) + \gamma v^{\pi}(s')]$;
    end
  end
  Policy Improvement(Greedy Policy);
  for s *in all states* **do**
    $\pi(s) = \arg\max_{a} \sum_{s'} P(s'|s,a)[r(s,a) + \gamma v^{\pi}(s')]$;
  end
end

More on Policy Iteration

- Policy Evaluation: Evaluate the current policy π_k to compute the value function $v_k^\pi(s)$. In other words, for the current policy, Bellman expectation equation is updated iteratively until convergence.
  $v^\pi(s) \leftarrow \sum_a (a|s) \sum_{s'} P(s'|s,a)[r(s,a) + \gamma v^\pi(s')]$
- Approximate Policy Evaluation: Sometimes, instead of iterating until full convergence, the number of iterations is limited to a fixed number for better computational efficiency and scalability.
- Policy Improvement: (Greedy Policy Improvement) Improve the policy by choosing actions that maximize the value function.
  $\pi(s) = \arg\max_a \sum_{s'} P(s'|s,a)[r(s,a) + \gamma v^\pi(s')] = \arg\max_a q^\pi(s,a)$
- At each move (i.e., each time step), both policy evaluation and policy improvement are performed for all state space.
- Related to on-policy learning (or method) since the policy being evaluated and improved is the same as the policy used to generate behavior (i.e., actions).

Q Function

So far, we have studied value function $v(s)$, which depends only on the current state. Sometimes, it is called state value function. In many cases in MDP and RL, action needs to be explicitly expressed in the value function to find the action that maximizes the value.

Q Function

Q-Function (Action-Value Function): The Q-function $q(s, a)$ represents the expected return (total accumulated reward) of taking a specific action a in a given state s and thereafter following a particular policy π.

$$q^{\pi}(s, a) = \mathbb{E}^{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right]$$

- Decision Making: Helps in making optimal decisions by comparing the expected returns of different actions in a given state.
- Policy Improvement: Used to derive the optimal policy by selecting the action that maximizes the Q-value.
- Policy Evaluation: Assists in evaluating how good a particular action is in terms of the expected return.

More on Q Function

- Relation with value function

$$v^{\pi}(s) = \mathbb{E}^{\pi}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \mid S_t = s\right] \tag{1}$$

$$q^{\pi}(s,a) = \mathbb{E}^{\pi}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \mid S_t = s, A_t = a\right] \tag{2}$$

$$q^{\pi}(s,a) = \mathbb{E}^{\pi}\left[R_{t+1} + \gamma v^{\pi}(s') \mid S_t = s, A_t = a\right] \tag{3}$$

$$v^{\pi}(s) = \sum_{a} \pi(a|s) q^{\pi}(s,a) \tag{4}$$

- In Eq(1), $v^{\pi}(s)$ DOES NOT depend explicitly on the action. R_{t+1} is the direct result of the implicit $A_t = a$ due to the current policy.
- In Eq(2,3), $q^{\pi}(s,a)$ DO depend explicitly on the action. R_{t+1} is the direct result of the explicit (in the input) $A_t = a$ regardless of the current policy. All other subsequent rewards are due to the policy.
- Eq(4) shows why the implicit action in the value function is due to the policy even though the explicit action in the q function is regardless of the policy.
- We also have Bellman equations for the Q function. We will study this when we cover RL.

Value Iteration

Value Iteration

An iterative algorithm that updates the value function until it converges to the optimal value function. Make sure that you understand how the value iteration is compared with policy iteration.

$$v^*(s) = \max_a \sum_{s'} P(s'|s,a) \left[r(s,a) + \gamma v^*(s') \right]$$

input : $r(s,a)$, $P(s'|s,a)$ from environment
Initialize arbitrarily;
$v(s) = 0$ for all s;
while $v(s)$ *converging* **do**
  for s *in all states* **do**
    $v(s) = \max_a \sum_{s'} P(s'|s,a)[r(s,a) + \gamma v(s')]$;
  end
end

More on Value Iteration

- Value Update: Update the value function for each state s using the Bellman equation.
  $v(s) \leftarrow \max_a \sum_{s'} P(s'|s,a)[r(s,a) + \gamma v(s')]$
- No explicit policy is involved in the value update. At each state, all possible actions should be tried to find the maximum value, which may be computationally expensive if the action space is large.
- Once the value function converges, it is the optimal value function.
- Optimal policy (or action) should be derived from the optimal value function.
- At each state, find the action that maximizes the following, which is equivalent to the Q function.
  $v^*(s) = \max_a \sum_{s'} P(s'|s,a)[r(s,a) + \gamma v^*(s')] = \max_a q^*(s,a)$

Contents

Policy Evaluation in Python

```python
def policy_evaluation(self):

    # Initialize the next value table with zeros
    next_value_table = [[0.00] * self.env.width
                                for _ in range(self.env.height)]

    # Update the value function for all states
    for state in self.env.get_all_states():
        value = 0.0
        # Do not update the value function for the terminal state
        if state == [2, 2]:
            next_value_table[state[0]][state[1]] = value
            continue

        # Update the value function via Bellman expectation equation
        for action in self.env.possible_actions:
            next_state = self.env.state_after_action(state, action)
            reward = self.env.get_reward(state, action)
            next_value = self.get_value(next_state)
            value += (self.get_policy(state)[action] *
                      (reward + self.discount_factor * next_value))

        next_value_table[state[0]][state[1]] = round(value, 2)

    self.value_table = next_value_table
```

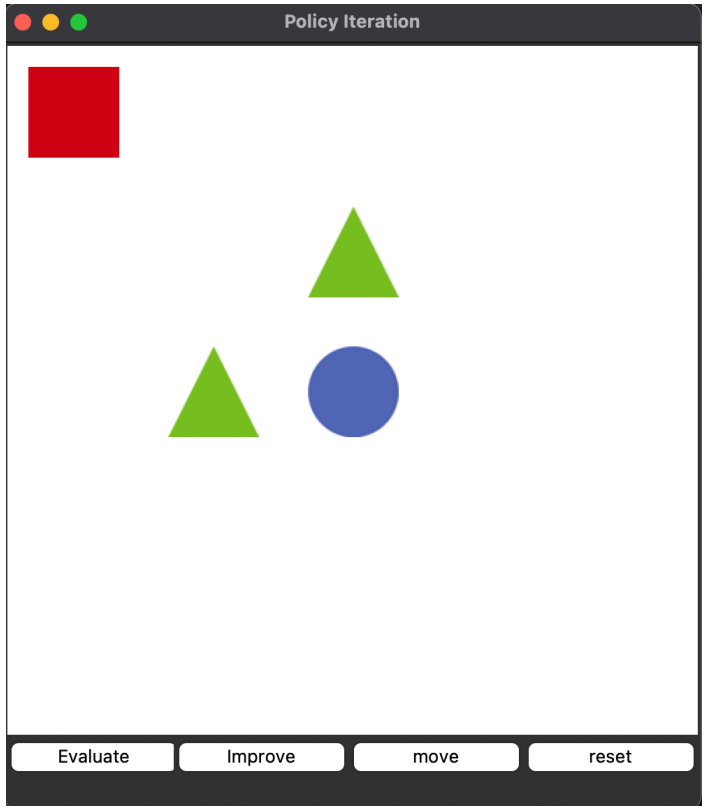
Policy Improvement in Python

```
def policy_improvement(self):
    next_policy = self.policy_table
    for state in self.env.get_all_states():
        if state == [2, 2]:
            continue
        value = -99999
        max_index = []
        # Initialize the next policy with zeros
        result = [0.0, 0.0, 0.0, 0.0]

        # Find the best action which maximizes the value function
        for index, action in enumerate(self.env.possible_actions):
            next_state = self.env.state_after_action(state, action)
            reward = self.env.get_reward(state, action)
            next_value = self.get_value(next_state)
            # Calculate the Q function (i.e., value) for the state and action
            temp = reward + self.discount_factor * next_value

            # Update the best action
            # If the value is the same as the maximum value, append the action (i.e., multiple actions might be the best)
            if temp == value:
                max_index.append(index)
            elif temp > value:
                value = temp
                max_index.clear()
                max_index.append(index)

        # Only the best policy has non-zero value
        prob = 1 / len(max_index)
        for index in max_index:
            result[index] = prob

        # Update the policy with the best action for all states
        next_policy[state[0]][state[1]] = result

    # Return the updated policy
    self.policy_table = next_policy
```

Grid World Example using Policy Iteration

- Initially, the value function (in tabular form) is set to zeros.
- Initially, the policy is random.
- To test out, click "improve" button once and "move" button.
- Now, for better performance, click "evaluate" button multiple times and "improve" button once, then "move" button.

- You can try policy evaluation and improvement by pen and paper method.
- This is important to fully understand the algorithm.
- Try it now.



Grid World Example

Value Iteration in Python

```
def value_iteration(self):
    next_value_table = [[0.0] * self.env.width for _ in
                        range(self.env.height)]
    for state in self.env.get_all_states():
        if state == [2, 2]:
            next_value_table[state[0]][state[1]] = 0.0
            continue
        # Initialize value list
        value_list = []

        # Calculate value function for all actions
        for action in self.env.possible_actions:
            next_state = self.env.state_after_action(state, action)
            reward = self.env.get_reward(state, action)
            next_value = self.get_value(next_state)
            value_list.append((reward + self.discount_factor * next_value))
        # Select the maximum value
        next_value_table[state[0]][state[1]] = round(max(value_list), 2)
    self.value_table = next_value_table
```

Optaining Action from the Optimal Value Function in Python

```
def get_action(self, state):
    action_list = []
    max_value = -99999

    if state == [2, 2]:
        return []

    # Calculate the value of all actions (reward + gamma * next_value = same as q function) from
    # Select the action with the maximum value
    for action in self.env.possible_actions:
        next_state = self.env.state_after_action(state, action)
        reward = self.env.get_reward(state, action)
        next_value = self.get_value(next_state)
        value = (reward + self.discount_factor * next_value)

        if value > max_value:
            action_list.clear()
            action_list.append(action)
            max_value = value
        elif value == max_value:
            action_list.append(action)

    return action_list
```

Contents

Summary of MDP and Dynamic Programming

Markov Decision Process (MDP)

- **Definition**: A framework for modeling decision-making where outcomes are partly random and partly under control.
- **Components**:
  - **States (S)**: Possible situations.
  - **Actions (A)**: Possible decisions.
  - **Transition Probabilities (P)**: Probability of moving between states.
  - **Rewards (R)**: Immediate returns.
  - **Discount Factor ( )**: Importance of future rewards.
- **Objective**: Find a policy π that maximizes expected cumulative reward.
- **Bellman Equation**:
  - State Value Function: $V(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a) + \gamma V(s')]$
  - Action Value Function (Q-function): $Q(s,a) = \sum_{s'} P(s'|s,a)[R(s,a) + \gamma \max_{a'} Q(s',a')]$

Summary of MDP and Dynamic Programming

Dynamic Programming (DP)

- **Definition**: A method for solving MDPs by breaking them down into simpler subproblems.
- **Key Algorithms**:
  - **Value Iteration**: Iteratively updates value function.
  - **Policy Iteration**: Alternates between policy evaluation and improvement.
- **Advantages**:
  - Systematic approach to finding optimal policies.
  - Convergence guarantees.
- **Limitations**:
  - Computationally intensive for large state/action spaces.
  - Requires a complete model of the environment.
- **Comparison with Reinforcement Learning**:
  - DP computes the value function and determines the optimal policy before any actions are taken. It relies on a complete model of the environment to solve for the optimal strategy in advance.
  - RL iteratively updates the value function and refines the policy without model through interactions with the environment. The agent learns by taking actions, receiving rewards, and adjusting its strategy based on the outcomes of these actions.

Comparison of Policy Iteration and Value Iteration

Aspect	Policy Iteration	Value Iteration
Convergence	Guaranteed in a finite number of steps	Guaranteed, but may take more iterations
Computational Cost	High, due to Policy Evaluation step	Lower per iteration, but may need more iterations
Implementation Complexity	More complex due to alternating steps	Simpler due to single-step updates
Suitability for Large State Spaces	Less suitable, high cost for large spaces	More suitable, handles large spaces better
Iteration Count	Typically fewer iterations	Typically more iterations
Policy Improvement	Direct and explicit	Implicit through value updates

Challenges

Consider a 1-DOF pendulum initially at rest. Using MDP framework, design a control policy to bring the pendulum upright.

- Total point mass of $1kg$, at $1m$ from the hinge
- Availble torques at the hinge are $[-2, -0.5, 0, 0.5, 2](Nm)$
- Discretize state space appropriately (angle and angular velocity). For example,
  - Twenty grids for angle $(-2\pi \leq \theta \leq 2\pi)$(rad)
  - Twenty grids for angular velocity $(-8 \leq \theta \leq 8)$(rad/s)
  - Terminal state$=(0, 0)$
  - Initial state$=(-\pi, 0)$
  - Redefine state space.
- What is the appropriate reward?

Next Topics

- **POMDP**:
  - A POMDP is an extension of an MDP where the agent does not have full observability of the current state. Instead, it receives observations that provide partial information about the state.
  - States (S), Actions (A), Transition Probabilities (P), Rewards (R), Observations (O), Observation Probabilities (Z), Discount Factor (γ)
  - Bayes rule is used to update belief states.
- **Reinforcement Learning**:
  - RL is a learning approach where an agent interacts with the environment (i.e., model-free) to learn optimal policies through trial and error, based on the rewards received from actions.
  - Q function is approximated via SARSA or Q-Learning.
  - Limitations
    - Scalability: Struggles with large or continuous state and action spaces.
    - Function Approximation: Limited to relatively simple environments where tabular methods are feasible.
- **Deep Reinforcement Learning**:
  - An extension of RL that uses deep neural networks to approximate value functions and policies, enabling the handling of high-dimensional state and action spaces.
  - Q function is approximated via Deep SARSA or DQN.
  - Policy can be directly approximated via Policy Gradient Methods.
    $\rightarrow$ REINFORCE, Actor-Critic(A2C, A3C), PPO, TRPO, DDPG, SAC

References

- "Reinforcement Learning: An Introduction" by Richard Sutton and Andrew Barto
- "Dynamic Programming and Optimal Control" (two volumes) by Dimitri Bertsekas
- chatGPT
- Several websites and githubs
- `https://github.com/pilwonhur/hurgroup_cc_mdp.git`

Thank you!