National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

Faculty of Computing

CS 368: Reinforcement Learning

Lab 03: Markov Decision Processes

(Iterative Policy Evaluation)

Date: 27 September 2024

Time: 02:00 PM – 5:00 PM

Instructor: Dr. Zuhair Zafar

Lab 03: Markov Decision Processes

A **Markov decision process (MDP)** refers to a stochastic decision-making process that uses a mathematical framework to model the decision-making of a dynamic system. It is used in scenarios where the results are either random or controlled by a decision maker, which makes sequential decisions over time. MDPs evaluate which actions the decision maker should take considering the current state and environment of the system.

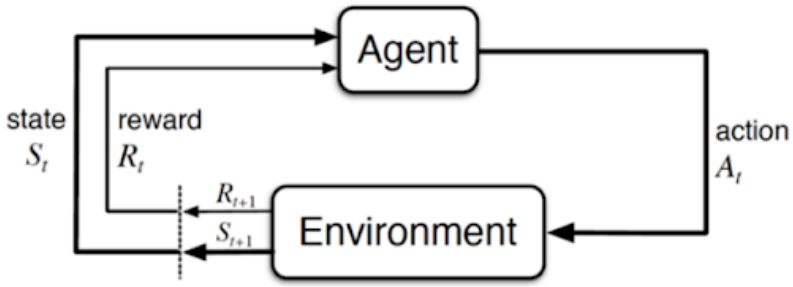
How Does the Markov Decision Process Work?

The MDP model operates by using key elements such as the **agent**, **states**, **actions**, **rewards**, and optimal **policies**. The **agent** refers to a system responsible for making decisions and performing actions. It operates in an environment that details the various states that the agent is in while it **transitions** from one state to another. MDP defines the mechanism of how certain states and an agent's actions lead to the other states. Moreover, the agent receives **rewards** depending on the action it performs and the state it attains (current state). The **policy** for the MDP model reveals the agent's following action depending on its current state.

The MDP framework has the following key components:

- S: states (s ∈ S)
- A: Actions (a ∈ A)
- P ($s_{t+1}| s_t, a_t$): Transition probabilities
- R (s): Reward

The graphical representation of the MDP model is as follows:



The MDP model uses the **Markov Property**, which states that the future can be determined only from the present state that encapsulates all the necessary information from the past. The Markov Property can be evaluated by using this equation:

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1, S_2, S_3 \dots \dots S_t]$$

According to this equation, the probability of the next state ($P[S_{t+1}]$) given the present state (S_t) is

given by the next state's probability ($P[S_{t+1}]$) considering all the previous states (S_1, S_2, S_3……S_t). This implies that MDP uses only the present/current state to evaluate the next actions without any dependencies on previous states or actions.

The **policy** (π) is known to determine the agent's optimal action given the current state so that it gains the maximum reward. In simple words, it associates actions with states.

$$\pi(s) \Rightarrow A$$

To determine the best policy, it is essential to define the **returns** that reveal the agent's rewards at every state. As a result, a variable termed '**discounted factor (γ)**' is introduced. The rule says if γ has values that are closer to zero, then the immediate rewards are prioritized. Subsequently, if γ reveals values closer to one, then the focus shifts to long-term rewards. Hence, the discounted infinite-horizon method is key to revealing the best policy.

The **value function V(s)** identifies the reward return at each specific state. Its formula is characterized by the expected sum of discounted future rewards:

$$V(s) = E[\sum_{t=0}^{\infty} \gamma^t r_t | s_t]$$

Iterative Policy Evaluation

Iterative Policy Evaluation is a method that, given a policy $\boldsymbol{\pi}$ and an MDP $\langle \boldsymbol{\mathcal{S}}, \boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{P}}, \boldsymbol{\mathcal{R}}, \gamma \rangle$, it iteratively applies the **bellman expectation equation** to estimate the value function $\boldsymbol{\mathcal{V}}$.

Let's look at a single step of the process. Consider a state, s of the environment and two possible actions $\mathbf{a_1}$ and $\mathbf{a_2}$. If the agent takes action $\mathbf{a_1}$, since the environment is usually stochastic, an action may lead to different new states. Let's consider the case of the first state, s'. On the way, we also collect our reward **r**.

An **expected update**: simulates all the possible states s', applies the bellman expectation equation below, and updates the state-value of the state, s.

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right)$$

Iterative Policy Evaluation applies the expected updates for each state, iteratively, until the convergence to the true value function is met. Note that solving for $\boldsymbol{\mathcal{V}}$, is equivalent to solve a system of linear equations in $|\boldsymbol{\mathcal{S}}|$ unknowns, the $\boldsymbol{\mathcal{V}}$(s), with $|\boldsymbol{\mathcal{S}}|$ expected updates equations. The solution to the system of equations is the value function for the specified policy.

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

Recycling Robot Example:

Consider a recycling robot that collects empty soda cans in an office environment. It can detect soda cans, pick them up using its gripper and drop them off to a recycling bin. The robot runs on a recharging battery. Its objective is to collect as many soda cans as possible.

Let's assume the robot sensors can only distinguish only 2 charged levels: LOW and HIGH. For each situation, robot has 3 choices: (1) It can search for cans for a fixed amount of time, (2) It can remain stationary and wait for someone to bring in a can, or (3) It can go to the charging station to recharge its battery. The recharging action can only be performed in the situation when charge level is LOW. If the robot does not recharge and battery depletes fully, robot needs to be externally rescued by a person and recharged the battery.

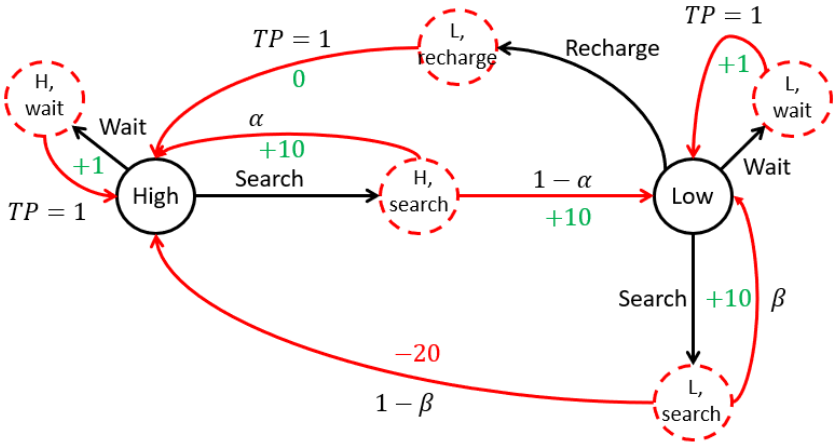
Searching with HIGH battery level may not result in change in battery level, i.e., with a transition probability of α. Alternatively, it may result in change in battery level with a transition probability of $1 - \alpha$. In both cases, the robot search yield, e.g., 10 soda cans. So, the reward is $+10$.

Waiting cannot result in change in battery level because waiting action does not drain the battery. The reward in this case is $+1$.

During searching with LOW battery, the robot battery may drain and depletes. In that case, the robot needs to be rescued externally with a probability of $1 - \beta$. Alternatively, the robot battery may not deplete during searching with a transition probability of β. The reward in the former situation is -20 and in later scenario is $+10$.

Taking the recharge action restores the battery with charge level HIGH with a reward of 0.

MDP model of Recycling Robot:



To evaluate a policy, iterative policy evaluation algorithm is used. For example, if the task is to evaluate the searching behavior in High and Low states, the following program reports how desirable or how good this behavior is.

Program:

```
import numpy as np

# Parameters
num_states = 2
num_actions = 3
discount_factor = 0.9
alpha = 0.8
beta = 0.2

# Initializing transition probabilities and rewards
transitions = np.zeros((num_states, num_actions, num_states))
rewards = np.zeros((num_states, num_actions, num_states))

###             MDP Definition             ###

# Transition probabilities
transitions[0, 0, 0] = alpha  # High state, search, High state
transitions[0, 0, 1] = 1 - alpha  # High state, search, Low state
transitions[0, 1, 0] = 1  # High state, Waiting, High state
transitions[0, 1, 1] = 0  # High state, Waiting, Low state
transitions[0, 2, 0] = 0  # High state, recharge, High state
transitions[0, 2, 1] = 0  # High state, recharge, Low state
```

```
transitions[1, 0, 0] = 1 - beta  # Low state, search, high state
transitions[1, 0, 1] = beta  # Low state, search, low state
transitions[1, 1, 0] = 0  # Low state, wait, High state
transitions[1, 1, 1] = 1  # Low state, wait, low state
transitions[1, 2, 0] = 1  # Low state, recharge, high state
transitions[1, 2, 1] = 0  # Low state, recharge, low state


# Rewards
rewards[0, 0, 0] = 10  # High state, search, High state
rewards[0, 0, 1] = 10  # High state, search, Low state
rewards[0, 1, 0] = 1  # High state, Waiting, High state
rewards[0, 1, 1] = 0  # High state, Waiting, Low state
rewards[0, 2, 0] = 0  # High state, recharge, High state
rewards[0, 2, 1] = 0  # High state, recharge, Low state

rewards[1, 0, 0] = -20  # Low state, search, high state
rewards[1, 0, 1] = 10  # Low state, search, low state
rewards[1, 1, 0] = 0  # Low state, wait, High state
rewards[1, 1, 1] = 1  # Low state, wait, low state
rewards[1, 2, 0] = 0  # Low state, recharge, high state
rewards[1, 2, 1] = 0  # Low state, recharge, low state

# Policy iteration
def policy_evaluation(policy, transitions, rewards, discount_factor, tol=1e-6):
    num_states, num_actions, _ = transitions.shape
    values = np.zeros(num_states)

    while True:
        delta = 0
        for s in range(num_states):
            v = values[s]

            action = policy[s]
            values[s] = sum(transitions[s, action, s_next] * (rewards[s, action, s_next] +
discount_factor * values[s_next])
                            for s_next in range(num_states))

            delta = max(delta, abs(v - values[s]))

        if delta < tol:
            break

    return values
```

```
policy = np.zeros(num_states, dtype=int)
policy[0] = 0   # Deterministic policy (choose the search action from high state)
policy[1] = 0   # Deterministic policy (always choose the search action from low state)
values = policy_evaluation(policy, transitions, rewards, discount_factor)

#Result

print("The value function for state high and state low when the agent always search is: “)
print(values)
```

Output:

```
DownLoads')
The value function for state high and state low when the agent always search is:
[56.79999305 32.79999371]
```

Lab Task

Consider an autonomous backhoe loader (similar to an excavator, it has a digging bucket but additionally, it also has an attached blade for pushing earth and building debris for coarse surface grading). The backhoe loader is working in an off-road environment.

By using its time-of-flight cameras and laser scanners, the backhoe loader can recognize which type of terrain it is on. The sensors can recognize 2 types of terrains accurately: rocky tracks and ridges. The backhoe loader can perform the following tasks: (1) If there are rocks or ridges nearby, it can use its driller to drill. (2) It can dig the ground using its digging bucket. However, it cannot dig when it is on the ridges. (3) It can push the debris using its blade.

Whenever the backhoe loader is drilling on a rocky track, there is a 30 percent chance that the terrain does not change, and the reward, in that case, is +5. With a probability of 0.7, the drilling of a rocky track changes the terrain and formed a ridge with a reward of +1. If the backhoe loader drills on a ridge, there is a 40 percent chance that the terrain never changes with a reward of +2 and a 60 percent chance that the ridges are converted into a rocky track with a reward of +6.

If the backhoe loader decides to dig the ground on a rocky track, there is a 75 percent chance that the terrain does not change with a reward of +7. It is possible with 0.25 probability, that digging the ground eventually forms a ridge, and the reward, in this case, is +1.

If the backhoe loader pushes the rocky debris on a rocky track, there is a 45 percent chance that the terrain does not change with a reward of +9. There is a 55 percent chance that pushing the rocky debris using its blade can form a ridge and the reward, in this case, is +5. However, if the backhoe loader pushes the debris on a ridge, there is an 80 percent chance that the terrain does not change with a reward of +2. With a 0.2 probability and reward of +10, pushing the debris on a ridge can change the terrain and form a rocky track.

1. **Using iterative policy evaluation, find the value of each state by following the given policy below.**

State (s)	$\boldsymbol{\pi(s)}$
Rocky Track	Drill
Ridge	Push

2. **Using iterative policy evaluation, find the value of each state by following the uniform random stochastic policy.**

Deliverable:

Please submit your notebook on LMS before the deadline **(1st October 2024, 11:59pm).**

Lab Rubrics

Assessment	Does not meet expectation (1/2 marks)	Meets expectation (3/4 marks)	Exceeds expectation (5 marks)
Software Problem Realization **(CLO1, PLO1)**	The student struggles to formulate the problem as MDP and does not identify an appropriate technique to solve it. There is a lack of demonstration of the problem's data requirements and no attempt to model the given environment effectively.	The student formulates the problem as MDP with some guidance, identifies a suitable technique with hints, and shows it's working. However, the approach might not be fully optimized or lacks a thorough justification.	The student independently formulates the given problem as MDP, selects the most appropriate technique without guidance, and effectively models the environment. The approach is fully optimized and can be applied to different similar problems.
Software Tool Usage **(CLO4, PLO5)**	Code has syntax errors, and the implementation of the Iterative Policy Evaluation is incorrect or incomplete. The code is not modular and lacks comments for readability and reuse. The student shows no ability to use relevant libraries where required.	The student has implemented the Iterative Policy Evaluation algorithm correctly for the given problem with minor mistakes. The code is mostly correct in terms of syntax and functionality but might not be optimized or well-structured for reuse. Some documentation is provided. The student also shows working knowledge of relevant libraries where required.	The student has implemented the Iterative Policy Evaluation algorithm efficiently and correctly. The code is clean, modular, well-documented, and follows best practices for reproducibility and reuse. The student demonstrates full command of libraries and tools.