



University: Isfahan University - Faculty of Computer Engineering

Course: Fundamentals and Applications of Artificial Intelligence

Markov Decision Process(MDP)

Course Instructor: Dr. Hossein Karshenas

Student Name: **Sheida Abedpour**

Student ID: 4003623025

Team\_id = 17

2023 Dec

## Cliff Walking Environment:

Before we start programming, we need to consider the underlying constraints and rules of our gridworld.

The implemented environment in this project called "CliffWalking" is a variation of the classic Cliff Walking problem. The environment is implemented as a subclass of CliffWalkingEnv from the gym library.

The CliffWalking class has the following attributes and methods:

Attributes:

- UP, RIGHT, DOWN, LEFT: Constants representing the possible actions in the environment.
- image\_path: A string representing the path to the image directory for rendering the environment.

Methods:

- `__init__(self, is_hardmode=True, num_cliffs=10, *args, **kwargs)`: The constructor method for the CliffWalking class. It initializes the environment by setting the hard mode flag, generating random cliff positions, and calculating transition probabilities and rewards.
- `_calculate_transition_prob(self, current, delta)`: A helper method that calculates the transition probabilities for a given state and action.
- `is_valid(self)`: A depth-first search (DFS) method that checks if there is a valid path from the start state to the terminal state in the environment.
- `step(self, action)`: Overrides the step method from the base class. It takes an action as input and returns the next state, reward, and termination status based on the action taken.
- `_render_gui(self, mode)`: A method for rendering the environment using the pygame library. It displays the environment as a graphical user interface (GUI) with images representing the different elements of the environment.

## MDP:

MDP stands for Markov Decision Process. It is a mathematical framework used to model decision-making problems in situations where outcomes are partly random and partly under the control of a decision-maker. MDPs are widely used in the field of reinforcement learning.

In an MDP, the decision-making problem is represented as a tuple  $(S, A, P, R)$ , where:

- $S$  is the set of possible states in the environment.
- $A$  is the set of possible actions that the decision-maker can take.
- $P$  is the state transition probability matrix, which defines the probability of transitioning from one state to another when a particular action is taken.
- $R$  is the reward function, which assigns a numerical reward to each state-action pair.

The MDP framework assumes the Markov property, which states that the future state and reward depend only on the current state and action, and not on the past history of states and actions. This property allows the problem to be modeled as a sequence of state-action pairs.

The goal in an MDP is to find an optimal policy, which is a mapping from states to actions that maximizes the expected cumulative reward over time. Various algorithms, such as value iteration and policy iteration, can be used to solve MDPs and find the optimal policy.

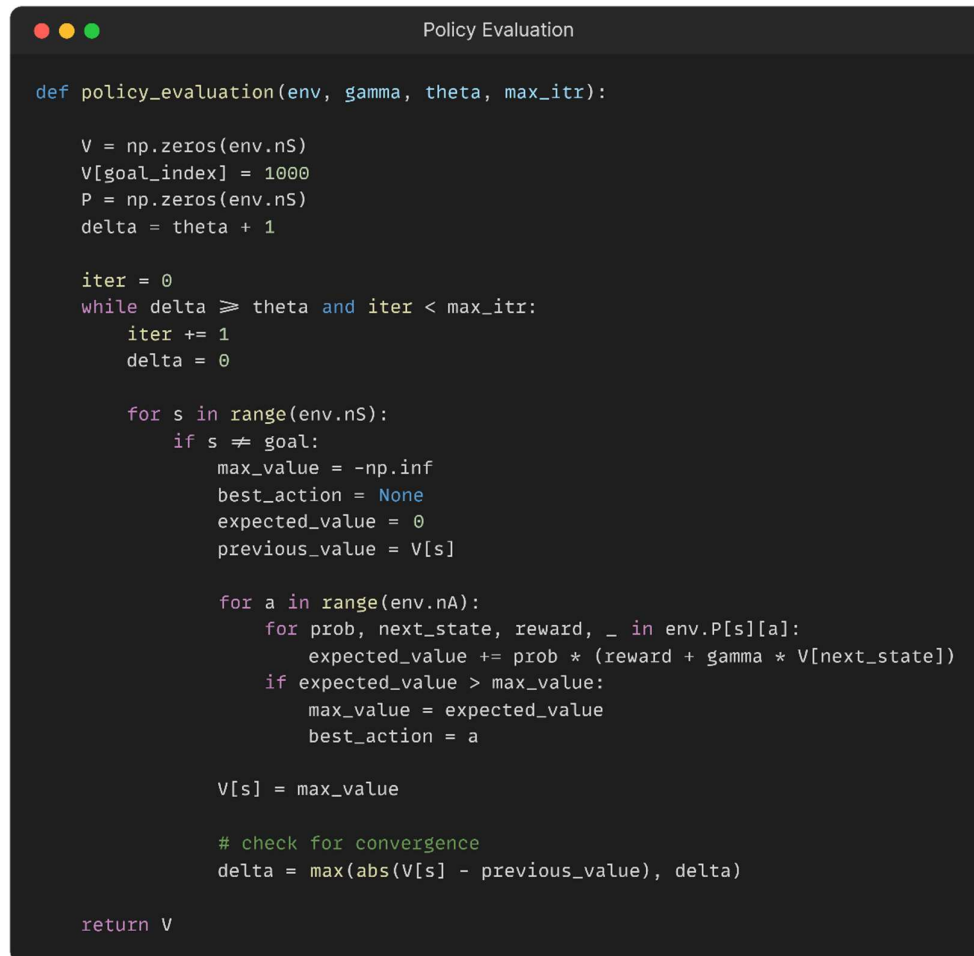
## Policy evaluation:

Policy evaluation is a step in the policy iteration algorithm used to estimate the value function for a given policy in a Markov Decision Process (MDP). The value function represents the expected cumulative reward starting from a particular state and following a specific policy. The goal of policy evaluation is to iteratively update the value function until it converges to the true value function for the policy. The value function is typically denoted as  $V(s)$ , where  $s$  is a state in the MDP.

Here is the general process of policy evaluation:

1. Initialization: Start with an initial estimate of the value function,  $V^0(s)$ , for all states  $s$  in the MDP.
2. Iterative Update: Repeat the following steps until the value function converges:
  - a. For each state  $s$  in the MDP:
    - Calculate the expected cumulative reward by considering all possible actions  $a$  according to the current policy  $\pi(s)$ .
    - Calculate the expected value of the next state,  $V^{(k-1)}(s')$ , using the transition probabilities and the previously estimated value function  $V^{(k-1)}(s')$ .
    - Update the value function estimate for the current state using the Bellman expectation equation:
$$V^k(s) = \sum [P(s' | s, a) * (R(s, a, s') + \gamma * V^{(k-1)}(s'))]$$
  - b. Check for convergence: Compare the updated value function  $V^k(s)$  with the previous estimate  $V^{(k-1)}(s)$  for all states. If the difference is below a certain threshold or after a fixed number of iterations, consider the value function to have converged.
3. Convergence: Once the value function has converged, the estimated value function  $V(s)$  represents the expected cumulative reward for each state under the given policy.

Policy evaluation is an important step in policy iteration as it provides an accurate estimate of the value function, which is then used in the policy improvement step to update the policy. By iteratively updating the value function and policy, policy iteration converges to the optimal policy in an MDP.



```
def policy_evaluation(env, gamma, theta, max_itr):  
  
    V = np.zeros(env.nS)  
    V[goal_index] = 1000  
    P = np.zeros(env.nS)  
    delta = theta + 1  
  
    iter = 0  
    while delta >= theta and iter < max_itr:  
        iter += 1  
        delta = 0  
  
        for s in range(env.nS):  
            if s != goal:  
                max_value = -np.inf  
                best_action = None  
                expected_value = 0  
                previous_value = V[s]  
  
                for a in range(env.nA):  
                    for prob, next_state, reward, _ in env.P[s][a]:  
                        expected_value += prob * (reward + gamma * V[next_state])  
                    if expected_value > max_value:  
                        max_value = expected_value  
                        best_action = a  
  
                V[s] = max_value  
  
        # check for convergence  
        delta = max(abs(V[s] - previous_value), delta)  
  
    return V
```

[codesnap.dev](#)

The code snippet you provided is the policy evaluation step within the value iteration algorithm. It updates the value function iteratively until it converges. Here's a breakdown of the code:

1. Initialization: The code initializes the iteration counter (iter) to 0 and sets delta to 0.
2. Iterative Update: The code enters a while loop that continues until either the change in the value function (delta) falls below the threshold (theta)

or the maximum number of iterations (`max_itr`) is reached. Within each iteration, the code updates the value function for each state.

3. Value Update: For each state (except the goal state), the code calculates the expected value for each action and selects the action that maximizes the expected value. The expected value is calculated by considering the transition probabilities, rewards, and the discount factor ( $\gamma$ ). The maximum value is stored in the value function  $V$  for the current state.
4. Convergence Check: After updating the value function for all states, the code checks for convergence by calculating the maximum change ( $\delta$ ) in the value function across all states. It compares the absolute difference between the updated value ( $V[s]$ ) and the previous value (`previous_value`) with the current  $\delta$  value. The maximum difference is stored in  $\delta$ .
5. Iteration Counter: The code increments the iteration counter (`iter`) by 1 in each iteration.

The purpose of this code is to iteratively update the value function until it converges, which is determined by the change in the value function falling below the threshold or reaching the maximum number of iterations.

## Policy iteration:

Policy iteration is an algorithm used to solve Markov Decision Processes (MDPs) and find the optimal policy. It is an iterative algorithm that alternates between two steps: policy evaluation and policy improvement.

Here is a overview of the policy iteration algorithm:

1. Initialization: Start with an initial policy  $\pi_0$ .
2. Policy Evaluation: Evaluate the value function  $V(s)$  for the current policy  $\pi$ . This involves iteratively updating the value function until it converges.
3. Policy Improvement: Update the policy  $\pi$  based on the current value function  $V(s)$ . This step involves selecting the action that maximizes the expected cumulative reward from each state:

$$\pi(s) = \operatorname{argmax}[\sum [P(s'|s, a) * (R(s, a, s') + \gamma * V(s'))]]$$

where  $P(s'|s, a)$  is the transition probability from state  $s$  to  $s'$  when action  $a$  is taken,  $R(s, a, s')$  is the reward obtained when transitioning from state  $s$  to  $s'$  with action  $a$ ,  $\gamma$  is the discount factor, and  $V(s')$  is the value function for the next state  $s'$ .

4. Convergence Check: Repeat steps 2 and 3 until the policy  $\pi$  and value function  $V(s)$  converge. Convergence is typically determined by a threshold or after a fixed number of iterations.

Policy iteration guarantees convergence to the optimal policy as long as the MDP is finite and the policy evaluation step converges to the true value function. It is a model-based approach that requires knowledge of the transition probabilities and reward function of the MDP.

```

import numpy as np

def policy_iteration(env, gamma, theta, max_itr):
    V = np.zeros(env.nS)
    goal_i, goal_j = env.terminal_state
    goal_index = 12 * goal_i + goal_j
    V[goal_index] = 1000
    P = np.zeros(env.nS, dtype=int)

    for _ in range(max_itr):
        # Policy Evaluation
        while True:
            delta = 0

            for s in range(env.nS):
                if s == goal_index:
                    continue

                v = V[s]
                a = P[s]
                expected_value = 0

                for prob, next_state, reward, _ in env.P[s][a]:
                    expected_value += prob * (reward + gamma * V[next_state])

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

            if delta < theta:
                break

        # Policy Improvement
        policy_stable = True

        for s in range(env.nS):
            if s == goal_index:
                continue

            old_action = P[s]
            max_value = -np.inf
            best_action = None

            for a in range(env.nA):
                value = 0

                for prob, next_state, reward, _ in env.P[s][a]:
                    value += prob * (reward + gamma * V[next_state])

                if value > max_value:
                    max_value = value
                    best_action = a

            P[s] = best_action

            if old_action != P[s]:
                policy_stable = False

        if policy_stable:
            break

    return V, P

```



Here's a breakdown of the code:

1. Initialization: The code initializes the value function  $V(s)$  for all states to 0, except for the goal state, which is set to a high value (1000 in this case). It also initializes the policy  $P(s)$  for all states to 0.
2. Iterative Update:
  - a) Policy Evaluation: The code uses a while loop to iteratively update the value function until the change in the value function falls below the specified threshold ( $\theta$ ). For each state (except the goal state), it calculates the expected value for the current policy by considering the transition probabilities, apologies for the incomplete response. Here's the continuation of the breakdown:
3. Iterative Update:
  - a) Policy Evaluation: The code uses a while loop to iteratively update the value function until the change in the value function falls below the specified threshold ( $\theta$ ). For each state (except the goal state), it calculates the expected value for the current policy by considering the transition probabilities, rewards, and the current value function. The value function  $V(s)$  is updated with the expected value, and the maximum change ( $\delta$ ) is tracked.
  - b) Policy Improvement: After policy evaluation, the code checks if the policy has changed for any state. It iterates over each state (except the goal state) and selects the action that maximizes the expected value based on the updated value function. The policy  $P(s)$  is updated with the best action. If the policy remains unchanged for all states, the algorithm breaks out of the loop and terminates.
4. Return: The code returns the final value function  $V$  and the final policy  $P$ .

## References:

- Chat GPT
- [youtube.com/@stanfordonline](https://www.youtube.com/@stanfordonline)
- [medium.com/markov-decision-process-value-iteration](https://medium.com/markov-decision-process-value-iteration)

## Python Libraries used in this project:

- NumPy
- Gymnasium