C3: Markov Decision Process (MDP) & Dynamic Programming

▼ Table of Content

Markov chain and Markov process

Markov Property

- Future depends only at the present and not the past.
- The current **state** (once recognized) provides sufficient information; since this state contains enough information about previous feedback, the *history* is no longer needed.

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

Markov Chain

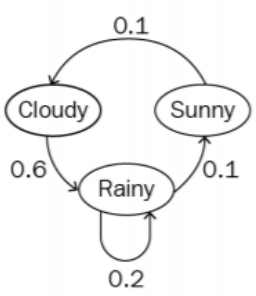
- Utilize the Markov property as a probabilistic model that solely depends on current state to predict the next state and not the previous states.
- Markov chain lies in the core concept that the future depends only on the present and not on the past
- Stochastic process is called markov process if it obeys the markov property
- e.g. If now *Cloudy* then later might be *Rainy* and it doesnt matter is the past *Sunny* or not

Transition Probability, $P^a_{ss'}$

- Probability of transitioning from one state to another given the current state
- Markov table shows the transition probability for all the next state given the current state

Current state	Next state	Transition probability
Cloudy	Rainy	0.6
Rainy	Rainy	0.2
Sunny	Cloudy	0.1
Rainy	Sunny	0.1

We can also represent the Markov chain in the form a state diagram that shows the transition probability:



Markov Decision Process

MDP can be use to model almost all RL problem. Provides mathematical framework for modeling decision-making situations.

Rewards and Returns

- Reward, r_t
  - Numeric Value that Reward or Penalize a model after a particular action at a specific time
- Return, R_t
  - Sum of reward that an agent receive from an environment

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

  - r_{t+1} is reward received by agent at time step t_0 while performing an action a_0 from one state to another
  - r_T is the reward received by agent at final time step T where terminating condition is met

Episodic and Continuous Task

- Episodic Task
  - Exist an terminal state (i.e. win the game)
- Continuous Task
  - Does not exist an terminal state and the environment and action continues forever (personal assisting robot)

Discount Factor

- Since our goal is to maximise Return function R_t at a specific time step t, for a continuous task, our Return function $\rightarrow \infty$ as is the sum of infinite terms.
- Moreover, there exist a conflict of Immediate Rewards vs Future Rewards. If the return function is as it is (i.e. without discount factor, γ), future rewards will have more impact than immediate rewards.
- Hence, discount factor is a hyperparameter from $0-1$ that make the following trade-off
  - $\gamma = 0$: Priorities Immediate Reward than future reward
    - Use case: Give chocolate now or 13 years later
  - $\gamma = 1$: Priorities Future Reward than immediate reward
    - Use case: Play chess to get enemy King than all the pawns
- Return function with discount factor:

$$\begin{aligned} R_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \\ &= \sum_{k=0}^{\infty} \gamma^k r_{k+t+1} \end{aligned}$$

Policy Function, $\pi(s_t)$

- Mapping from states to an action

$$\pi(s) : S \rightarrow A$$

- Goal is to find the optimal policy which specifies the best action to perform in each state, that maximizes the rewards

State Value Function, $V^{\pi}(s_t)$

- Estimated return using the policy function given the current state.

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

State	Value
State 1	0.3
State 2	0.9

Based on the preceding table, we can tell that it is good to be in state 2, as it has high value. We will see how to estimate these values intuitively in the upcoming sections.

State-Action Value Function (Q-Function), $Q^{\pi}(s_t, a_t)$

- Estimated return using the policy function given the current state and the current action

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

State	Action	Value
State 1	Action 1	0.03
State 1	Action 2	0.02
State 2	Action 1	0.5
State 2	Action 2	0.9

The Bellman Equation and Optimality

- Quantify the optimal **Policies,** π and **values function,** $v(s)$ given the current state s_t, action space $\sum_a$
- To solve Markov Decision Process, we need to find the optimal policies and value functions, $V^*(s)$ where

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

  - A maximum value function is equivalent as picking the best action given current state (i.e. maximum Q-function)
- The Bellman Equation is represented as follows:
  - Value Function

```
value_func = 0
available_action = filter(lambda action:
                          possible_action(action, state), action_space)
for action in available_action:
    for next_state in policy_func(state, action): # If one state and action cna have multiple next steps
        value_func += trans_prob[action] *
                      (reward_func(action, state) +
                        gamma*value_func(policy, next_state))
```

$$V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^\pi(s')]$$

  - Q-Function

```
q_func = 0
for next_state in policy_func(state, action): # If one state and action cna have multiple next steps
    value_func += trans_prob[action] *
                  (reward_func(action, state) +
                    gamma*sum(
                        [q_func(policy, next_state, next_action) for next_action in next_actions]
                      )
                  )
```

$$Q^\pi(s,a) = \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma \sum_{a'} Q^\pi(s',a')]$$

Derivation of Bellman Equation

Transition Probability, $P^a_{ss'}$

Probability of moving from state s to s' while performing action a

$$P^a_{ss'} = \mathbb{P}(s_{t+1} = s' | s_t = s, a_t = a)$$

Reward Probability, $R^a_{ss'}$

Probability of reward received of moving from state s to s' while performing action a

$$R^a_{ss'} = \mathbb{E}[R_{t+1}|s_t = s, s_{t+1} = s', a_t = a]$$
$$= \gamma\mathbb{E}[\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_{t+1} = s]$$

Expected Return, $\mathbb{E}_\pi[r_{t+1}|s_t = s]$

Expected return of the current state is the sum of all possible actions and rewards taking into account the transition probabilities of such actions.

$$\mathbb{E}_\pi[r_{t+1}|s_t = s] = \sum_a \pi(s,a)\sum_{s'} P^a_{ss'}R^a_{ss'}$$

$$\mathbb{E}_\pi[r_{t+1}|s_t = s] = \sum_a \pi(s,a)\sum_{s'} P^a_{ss'}\gamma V^\pi(s')$$

Value Function, $V(s)$

Value function of a **current state,** s is the **sum of immediate reward,** $R^a_{ss'}$ and **gamma-weighted value of next state,** $\gamma V(s')$ for **all possible next state,** a' from **all possible action,** a given the **current state,** s_t

$$V^\pi(s) = \mathbb{E}_\pi[R_t|s_t = s]$$

$$V^\pi(s) = \mathbb{E}_\pi[r_{t+1} + \gamma r_{t+2} + \cdots|s_t = s]$$

$$V^\pi(s) = \sum_a \pi(s,a)\sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^\pi(s')]$$

Q-Function, $Q(s,a)$

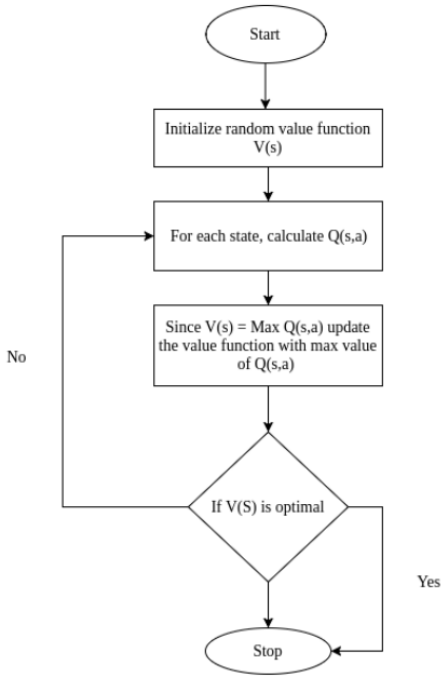
Q-function of a given state and action is the sum of **immediate reward,** $R^a_{ss'}$ and **gamma-weighted** Q**-function for all next action**, $\gamma\sum_{a'}Q^\pi(s',a')$ for all possible next state $\sum_{s'}$

$$Q^\pi(s,a) = \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma\sum_{a'}Q^\pi(s',a')]$$

Dynamic Programming

- Solving complex problem by breaking down into sub-problem.
- After computing the sub-problem, the solution is stored and will not be recompute.
- Hence, it can minimize computation time (caching of solution)
- Both **Value Iteration** and **Policy Iteration** are two iterative methods to find the optimal policy function with nothing but rewards and transition probability for each state (without explicitly define the action to be taken in each state)

Value Iteration



- Goal : Find the converged value table for each of the state iteratively and use the value table to derive the optimal action to be taken at the current state.
- Steps:
  1. Randomly initialize a value table with shape <numberOfState, 1(valueFunc)>
  2. for each state:
     a. Compute the q-table for each of the action using the randomly initialized value table

$$Q(s,a) = \text{Transition probability} \times (\text{Reward Probability} + \text{gamma} \times \text{value\_of\_next\_state})$$
$$= \sum_{s'} P^a_{ss'} * (R^a_{ss'} + \gamma V(s'))$$

     b. Set the maximum q-function as the optimal value for the particular state
  3. If difference of update to value table is below the tolerance level, optimal value table is found and break the iteration. Else continue the episode.

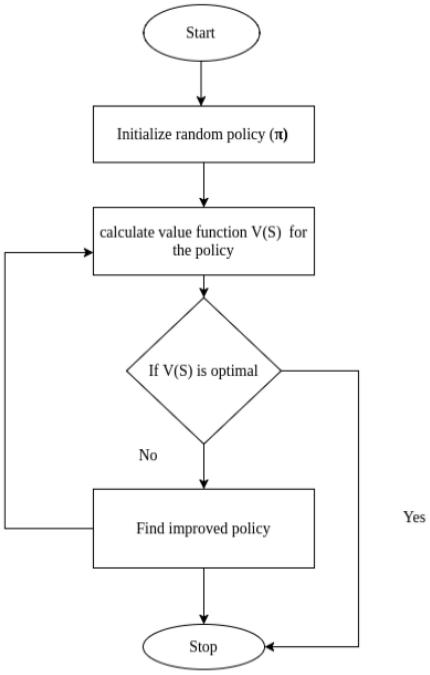
```
import gym
import numpy as np
env = gym.make('FrozkenLake-v0')

def value_iteration(env, gamma = 1.0):

    # initialize value table with zeros
    value_table = np.zeros(env.observation_space.n)

    # set number of iterations and threshold
    no_of_iterations = 100000
    threshold = 1e-20

    for i in range(no_of_iterations):

        # On each iteration, copy the value table to the updated_value_table
```

```
        updated_value_table = np.copy(value_table)

        # Now we calculate Q Value for each actions in the state
        # and update the value of a state with maximum Q value

        for state in range(env.observation_space.n):
            Q_value = []
            for action in range(env.action_space.n):
                next_states_rewards = []
                for next_sr in env.P[state][action]:
                    trans_prob, next_state, reward_prob, _ = next_sr
                    next_states_rewards.append((trans_prob * (reward_prob + gamma * updated_value_table[next_state])))

                Q_value.append(np.sum(next_states_rewards))

            value_table[state] = max(Q_value)

        # we will check whether we have reached the convergence i.e whether the difference
        # between our value table and updated value table is very small. But how do we know it is very
        # small? We set some threshold and then we will see if the difference is less
        # than our threshold, if it is less, we break the loop and return the value function as optimal
        # value function

        if (np.sum(np.fabs(updated_value_table - value_table)) <= threshold):
             print ('Value-iteration converged at iteration# %d.' %(i+1))
             break

    return value_table
```

4. After the optimal value table is found, the optimal policy table (consist of the optimal action to be taken under each state) can be derived through calculation of q-table using the optimal value table.

```
def `(value_table, gamma = 1.0):

    # initialize the policy with zeros
    policy = np.zeros(env.observation_space.n)


    for state in range(env.observation_space.n):

        # initialize the Q table for a state
        Q_table = np.zeros(env.action_space.n)

        # compute Q value for all ations in the state
        for action in range(env.action_space.n):
            for next_sr in env.P[state][action]:
                trans_prob, next_state, reward_prob, _ = next_sr
                Q_table[action] += (trans_prob * (reward_prob + gamma * value_table[next_state]))

        # select the action which has maximum Q value as an optimal action of the state
        policy[state] = np.argmax(Q_table)

    return policy
```

Policy Iteration



- Goal: Find the optimal policy function/table by evaluating and optimizing the value of the policy using the value table as the mirror.
- Steps:
  1. Randomly initialise a policy function that have the action for all the states available
  2. Using the policy function, calculate the new value function using the following code

     ▼ Pseudocode

     - while true:
       - for each state:
         - Extract appropriate action from the policy function
         - Calculate the q-value given extracted action and state
         - Update the value_table using the q-value given
         - Stop iteration if no more changes to the value table
     - return value_table

```
def compute_value_function(policy, gamma=1.0):

    # initialize value table with zeros
    value_table = np.zeros(env.nS)

    # set the threshold
    threshold = 1e-10

    while True:

        # copy the value table to the updated_value_table
        updated_value_table = np.copy(value_table)

        # for each state in the environment, select the action according to the policy and compute the value table
```

```
        for state in range(env.nS):
            action = policy[state]

            # build the value table with the selected action
            value_table[state] = sum([trans_prob * (reward_prob + gamma * updated_value_table[next_state])
                        for trans_prob, next_state, reward_prob, _ in env.P[state][action]])

        if (np.sum((np.fabs(updated_value_table - value_table))) <= threshold):
            break

    return value_table
```

3. Using the calculated value table (which reflect how well is the policy function), extract the improved policy function using the `extract_policy` function in Policy Iteration above
4. If no changes is made to the policy function, then break the iteration. Else continue the episode.