MDP

Kushal Kharel

2023-11-06

"Reinforcement learning problem uses training information that evaluates the actions taken rather than instructs by giving correct action." -Sutton & Barto

Reinforcement learning is based on reward hypothesis. All goals can be described by maximization of cumulative reward.

There are two fundamental problems in sequential decision making. One is Reinforcement Learning where the environment is initially unknown where agent interacts with the environment continually to improve its policy over time. The other is the problem of planning - the model of the environment is known where the agent performs computation with its model to improve its policy. In the latter, there is no external interaction.

The agent is faced with choosing optimal action from set of actions in different situations. The action that agent takes in current state will affect the future states through future rewards. Markov Decision Processes will help us solve these types of sequential decision making problems. It involves solving the trade off between immediate rewards and future rewards. After selecting the action, agent receives a numerical reward from stationary probability distribution. The goal of the agent is to maximize the numerical reward it receives from selecting the action many times over several time steps. When task is *stationary*, the agent will try to find the best action. The reward probabilities do not change over time. On the other hand, when the task is $non-stationary$, the agent tries to track best action as it changes over time.

In Markov Decision Processes, we estimate the optimal value of each action in each state. We estimate $q_*(s, a)$, which is the optimal value of each action in each state. Or, we estimate $v_*(s)$ which is the optimal value of each state given optimal action selections.

At timestep t, the agent observes the environment and takes some action. The environment receives the action taken by the agent at timestep t, and it receives the numerical reward at timestep $t+1$. Simultaneously, the environment also provides the agent with new observations, representing the updated state of the environment after the agent's action at timestep t. These observations serve as the basis for the agent's decision-making process in subsequent timesteps, enabling it to adapt and refine its strategy over time.

History is the sequence of observations, actions and rewards which serves as a memory for the agent, enabling it to learn from past experiences, understand the current context, and make informed decisions to maximize cumulative rewards over time.

$$H_t = o_0, a_0, r_1, ..., o_{t-1}, a_{t-1}, r_t$$

The state is the function of above history:

$$S_t = f(H_t)$$

and the trajectory is as follows:

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, ...$$

A discrete time markov chain is the sequence of states $S_0, S_1, S_2, ...$ with markov property. Markov property states that the probability of moving to next state only depends on current state and not on the previous state. The current state captures all the information about the previous states. In other words, future is independent of the past given the present.

$$Pr(S_{t+1}|S_t, S_{t-1}, S_{t-2}, ..., S_0) = Pr(S_{t+1}|S_t)$$

if both conditional probabilities are well defined,

$$Pr(S_t, S_{t-1}, S_{t-2}, ..., S_0) > 0$$

Markov Chains Review:

The different types of states in Markov Chain are defined below:

Reachable and communicative state: With some positive probability, state i transitions into state j. And if the reverse is also true then the states are said to be communicative. If all states are communicative then the chain is said to be irreducible.

Absorbing state: A state is said to be absorbing state if the transition from state i is to itself.

Transient and recurrent state: Transient state means state i is reachable from state j but not the other way around. After many timesteps, environment will never come back to transient state i. State that is not transient is recurrent state.

Periodic and aperiodic state: If all the paths leaving state i come back again after some time steps then it is said to be periodic. The state that never comes back is said to be aperiodic state.

A markov chain is called ergodic if all states can communicate with each other, are recurrent and also aperiodic.

In a finite Markov Decision Process, we treat rewards and states as random variables which have well defined discrete probability distributions depending only on the previous state and previous action. The probability function is given by,

$$p(s', r|s, a) := Pr(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a),$$

$$\forall s', s \in S, r \in R \in \mathbb{R}, a \in A(s)$$

where p is the probability for each choice of state s and each action a. We know that the sum of probabilities must be 1. Hence,

$$\sum_{s' \in S, r \in R} p(s', r|s, a) = 1$$

,

$$\forall s \in S, a \in A(s)$$

From this probability function p, we compute state-transition probabilities, expected rewards for the state action pairs and expected rewards for the state action and next state triples. Markov chain is completely characterized by the tuple $< S, P >$, where S is the set of all states and P is the $n * n$ transition probability matrix.In other words, the dynamics of the environment can be completely characterized by states and trasition probabilities. The behavior of Markov chain can be modeled mathematically in a matrix form as $p = qP^t$ where q is the initial state probabilities and P^t is the state transition probabilities raised to the number of time steps.

By definition, State-Transition Probability is given by,

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

where, $p(s', r|s, a)$ is the joint probability of transitioning to state s' and receiving reward r given that the previous state was s and the action taken was a.

The sum over $r \in R \subset \mathbb{R}$ indicates that we are summing the probabilities over all possible rewards in the set R. In many cases, rewards are discrete, so we can enumerate them. This is the total probability of transitioning to state s' given that the previous state was s and the action taken was a.

Expected Rewards for State-Action Pairs is: - Sutton & Barto

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

and Expected Rewards for State-Action-State triple is:- Sutton & Barto

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

State probabilities refer to the likelihood or probability of being in a specific state at a given point in time within a Markov chain. State transition probabilities describe the likelihood of transitioning from one state to another in the Markov chain.

We take the classic grid-world example with robot at the center to demonstrate the probability of the robot being in different states after a certain number of time steps. Initialize the grid world by putting the agent at the center position $(1, 1)$ of $3x3$ grid. The initial probability for all the states is zero except for the 5th state where agent is found initially with probability 1.

From the center, the agent can take actions a from set of actions $A_t = \{up, down, left, right\}$. Suppose, the agent can move up with probability $up = 0.3$, can move down with probability $down = 0.2$, can move right with probability $right = 0.35$ and can move left with probability $left = 0.15$.

```
import numpy as np
np.set_printoptions(threshold=np.inf)

# Initialize q values with all zeros for 3x3 matrix which is the initial probability distribution
size = m = 3
initial_probability = np.zeros(size**2)
initial_probability[size**2//2] = 1 # put the robot at location 4 that is at position (1,1) center.
print("The size of the grid:", m, "x" , m)
```

```
## The size of the grid: 3 x 3
```

```
initial_probability_matrix = initial_probability.reshape(m, m)
print("The state probabilities, 'q':\n", initial_probability_matrix)
```

```
## The state probabilities, 'q':
##  [[0. 0. 0.]
##  [0. 1. 0.]
##  [0. 0. 0.]]
```

From any of the state i, the agent can transition to any of the state j. The state transition probability defines how the environment evolves over time. In other words, the probabilities of an agent moving from one state to another in discrete-time steps t. The function to calculate transition probability is shown below.

```
def get_transition_probabilities(m, p_up, p_down, p_left, p_right):
  if m <= 1 or not np.isclose(p_up + p_down + p_left + p_right, 1.0):
    raise ValueError("Invalid input")

  transition_probability = np.zeros((m**2, m**2))

  states = {}
  for state in range(m**2):
    states[state+1] = (state // m, state % m)

  for initial_state in range(m**2):
    for destination_state in range(m**2):
      row_initial_state, column_initial_state = states[initial_state+1]
      row_destination_state, column_destination_state = states[destination_state+1]

      row_difference = row_initial_state - row_destination_state
      column_difference = column_initial_state - column_destination_state

      #print(f"Initial state: ({r_i}, {c_i}), Destination state: ({r_j}, {c_j}), Row Difference: {row_d

      #horizontal movement
      if row_difference == 0: # no movement row wise
        if column_difference == 1: # column movement to left
          transition_probability[initial_state, destination_state] = p_left
        elif column_difference == -1:  # column movement to right
          transition_probability[initial_state, destination_state] = p_right

        # check boundaries up,down,left,right to ensure the agent does not go out of bounds
        elif column_difference == 0: # no movement column wise
          if row_initial_state == 0: # top row
            transition_probability[initial_state, destination_state] += p_up # increment with p_up
          elif row_initial_state == m - 1: # bottom row
            transition_probability[initial_state, destination_state] += p_down # increment p_down
          if column_initial_state == 0: # leftmost column
            transition_probability[initial_state, destination_state] += p_left # increment p_left
          elif column_initial_state == m - 1: # rightmost column
            transition_probability[initial_state, destination_state] += p_right #  increment p_rgt

      # Vertical Movement
      elif row_difference == 1: # movement up row wise
        if column_difference == 0: # no column movement
          transition_probability[initial_state, destination_state] = p_up
      elif row_difference == -1: # movement down row wise
        if column_difference == 0: # no column movement
          transition_probability[initial_state, destination_state] = p_down

  return transition_probability
```

Initially at time step 0, the transition probability matrix is an identity matrix. It tells us that there are no transitions between states at time step 0 and each state remains in its own state with probability of 1. This assumption simplifies the initial conditions of the Markov Chain and serve as a starting point for subsequent environment evolution.

```
transition_probability = get_transition_probabilities(m=3, p_up=0.3, p_down=0.2, p_left=0.15, p_right=0
timestep = 0
transition_probability_at_timestep_t = np.linalg.matrix_power(transition_probability, timestep)
print("State Transition Probabilities that governs the system at time step 0:\n", np.round(transition_p
```

```
## State Transition Probabilities that governs the system at time step 0:
##  [[1. 0. 0. 0. 0. 0. 0. 0. 0.]
##  [0. 1. 0. 0. 0. 0. 0. 0. 0.]
##  [0. 0. 1. 0. 0. 0. 0. 0. 0.]
##  [0. 0. 0. 1. 0. 0. 0. 0. 0.]
##  [0. 0. 0. 0. 1. 0. 0. 0. 0.]
##  [0. 0. 0. 0. 0. 1. 0. 0. 0.]
##  [0. 0. 0. 0. 0. 0. 1. 0. 0.]
##  [0. 0. 0. 0. 0. 0. 0. 1. 0.]
##  [0. 0. 0. 0. 0. 0. 0. 0. 1.]]
```

```
print("State probabilities at time step 0:", np.matmul(initial_probability, transition_probability_at_t
```

```
## State probabilities at time step 0: [0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

At time step 1, the transition probabilities that governs the system and state probabilities is shown below.

```
transition_probability = get_transition_probabilities(m=3, p_up=0.3, p_down=0.2, p_left=0.15, p_right=0
timestep = 1
transition_probability_at_timestep_t = np.linalg.matrix_power(transition_probability, timestep)
print("State Transition Probabilities that governs the system at time step 1:\n", np.round(transition_p
```

```
## State Transition Probabilities that governs the system at time step 1:
##  [[0.45 0.35 0.   0.2  0.   0.   0.   0.   0.  ]
##  [0.15 0.3  0.35 0.   0.2  0.   0.   0.   0.  ]
##  [0.   0.15 0.65 0.   0.   0.2  0.   0.   0.  ]
##  [0.3  0.   0.   0.15 0.35 0.   0.2  0.   0.  ]
##  [0.   0.3  0.   0.15 0.   0.35 0.   0.2  0.  ]
##  [0.   0.   0.3  0.   0.15 0.35 0.   0.   0.2 ]
##  [0.   0.   0.   0.3  0.   0.   0.35 0.35 0.  ]
##  [0.   0.   0.   0.   0.3  0.   0.15 0.2  0.35]
##  [0.   0.   0.   0.   0.   0.3  0.   0.15 0.55]]
```

```
print("State probabilities at time step 1:", np.matmul(initial_probability, transition_probability_at_t
```

```
## State probabilities at time step 1: [0.   0.3  0.   0.15 0.   0.35 0.   0.2  0.  ]
```

Recall, $p_n = qP^n$,

The transition probability that governs the environment at time step 2 is given by;

```
transition_probability = get_transition_probabilities(m=3, p_up=0.3, p_down=0.2, p_left=0.15, p_right=0
timestep = 2
transition_probability_at_timestep_t = np.linalg.matrix_power(transition_probability, timestep)
print("Transition Probabilities that governs the system at time step 2:\n", np.round(transition_probabil
```

```
## Transition Probabilities that governs the system at time step 2:
##  [[0.315 0.262 0.122 0.12  0.14  0.    0.04  0.    0.   ]
##  [0.112 0.255 0.332 0.06  0.06  0.14  0.    0.04  0.   ]
##  [0.022 0.142 0.535 0.    0.06  0.2   0.    0.    0.04 ]
##  [0.18  0.21  0.    0.195 0.052 0.122 0.1   0.14  0.   ]
##  [0.09  0.09  0.21  0.022 0.225 0.122 0.06  0.04  0.14 ]
##  [0.    0.09  0.3   0.022 0.052 0.295 0.    0.06  0.18 ]
##  [0.09  0.    0.    0.15  0.21  0.    0.235 0.192 0.122]
##  [0.    0.09  0.    0.09  0.06  0.21  0.082 0.205 0.262]
##  [0.    0.    0.09  0.    0.09  0.27  0.022 0.112 0.415]]
```

```
print("State probabilities at time step 2:", np.matmul(initial_probability, transition_probability_at_t
```

```
## State probabilities at time step 2: [0.09   0.09   0.21   0.0225 0.225  0.1225 0.06   0.04   0.14  ]
```

At time step 3,

```
transition_probability = get_transition_probabilities(m=3, p_up=0.3, p_down=0.2, p_left=0.15, p_right=0
timestep = 3
transition_probability_at_timestep_t = np.linalg.matrix_power(transition_probability, timestep)
print("Transition Probabilities that governs the system at time step 3:\n", np.round(transition_probabil
```

```
## Transition Probabilities that governs the system at time step 3:
##  [[0.217 0.249 0.171 0.114 0.094 0.074 0.038 0.042 0.   ]
##  [0.107 0.184 0.347 0.04  0.105 0.136 0.018 0.02  0.042]
##  [0.031 0.149 0.458 0.014 0.058 0.21  0.    0.018 0.062]
##  [0.171 0.142 0.11  0.103 0.171 0.061 0.095 0.074 0.074]
##  [0.061 0.158 0.205 0.073 0.056 0.206 0.032 0.095 0.115]
##  [0.02  0.088 0.315 0.011 0.088 0.236 0.014 0.05  0.179]
##  [0.085 0.094 0.    0.142 0.11  0.11  0.141 0.181 0.135]
##  [0.04  0.045 0.094 0.047 0.142 0.173 0.078 0.121 0.258]
##  [0.    0.04  0.139 0.02  0.074 0.268 0.025 0.111 0.322]]
```

```
print("State probabilities at time step 3:", np.matmul(initial_probability, transition_probability_at_t
```

```
## State probabilities at time step 3: [0.06075  0.1575   0.20475  0.073125 0.05625  0.205625 0.0315   0
##  0.1155  ]
```

and so on…

Eventually after several time steps, this environment states will eventually reach to a steady state. The term "steady state" is often used to describe the long-term behavior of an agent interacting with an environment. The steady state refers to a point where the agent's behavior has stabilized, and its actions and policies are consistent over time.

The steady state is reached when the agent's policy or value function converges, and further exploration or learning does not lead to significant changes in behavior.Reaching a steady state is crucial for evaluating the performance of reinforcement learning algorithms, as it reflects the agent's learned policy in a stable state.

Let's run this over several time steps and print the results to see steady state distribution

```
transition_probability = get_transition_probabilities(m=3, p_up=0.3, p_down=0.2, p_left=0.15, p_right=0
time_steps = [0, 1, 3, 5, 10, 20, 100]

# Calculate and print transition probabilities for each time step
for time in time_steps:
  transition_probability_at_timestep_t = np.linalg.matrix_power(transition_probability, time)
  #print(f"Transition Probabilities that govern the system at time step {t}:\n", np.round(Pt, 3))
  print(f"State probabilities at time step {time}:", np.round(np.matmul(initial_probability, transition_
  print("\n")
```

```
## State probabilities at time step 0: [0. 0. 0. 0. 1. 0. 0. 0. 0.]
##
##
## State probabilities at time step 1: [0.   0.3  0.   0.15 0.   0.35 0.   0.2  0.  ]
##
##
## State probabilities at time step 3: [0.06 0.16 0.2  0.07 0.06 0.21 0.03 0.1  0.12]
##
##
## State probabilities at time step 5: [0.06 0.13 0.25 0.05 0.08 0.19 0.03 0.07 0.13]
##
##
## State probabilities at time step 10: [0.06 0.13 0.29 0.04 0.09 0.19 0.03 0.06 0.13]
##
##
## State probabilities at time step 20: [0.05 0.13 0.29 0.04 0.08 0.2  0.02 0.06 0.13]
##
##
## State probabilities at time step 100: [0.05 0.13 0.29 0.04 0.08 0.2  0.02 0.06 0.13]
```

In the above results, there is no difference in state probabilities between step 20 and step 100. This means that the environment has converged to the steady state distribution. Also, the probability of state 3 is higher than any other state probability. We are likely to find robot at state 3 once the environment reaches steady state because we assign higher probability to moving up and right action.

We can also check what happens when we give an agent equal probability of taking action from set of actions that is equal probability of moving up, down, left and right.

```
transition_probability = get_transition_probabilities(m=3, p_up=0.25, p_down=0.25, p_left=0.25, p_right
time_steps = [0, 1, 3, 5, 10, 20, 100]

# Calculate and print transition probabilities for each time step
for time in time_steps:
  transition_probability_at_timestep_t = np.linalg.matrix_power(transition_probability, time)
  #print(f"Transition Probabilities that govern the system at time step {t}:\n", np.round(Pt, 3))
  print(f"State probabilities at time step {time}:", np.round(np.matmul(initial_probability, transition_
  print("\n")
```

```
## State probabilities at time step 0: [0. 0. 0. 0. 1. 0. 0. 0. 0.]
##
##
## State probabilities at time step 1: [0.   0.25 0.   0.25 0.   0.25 0.   0.25 0.  ]
##
##
## State probabilities at time step 3: [0.09 0.14 0.09 0.14 0.06 0.14 0.09 0.14 0.09]
##
##
## State probabilities at time step 5: [0.11 0.12 0.11 0.12 0.1  0.12 0.11 0.12 0.11]
##
##
## State probabilities at time step 10: [0.11 0.11 0.11 0.11 0.11 0.11 0.11 0.11 0.11]
##
##
## State probabilities at time step 20: [0.11 0.11 0.11 0.11 0.11 0.11 0.11 0.11 0.11]
##
##
## State probabilities at time step 100: [0.11 0.11 0.11 0.11 0.11 0.11 0.11 0.11 0.11]
```

If we start with equal action probability then the environment reaches steady state at time step 10, environment converges quickly. We are equally likely to see the agent in any of the states.

Now, we introduce the absorbing state and evaluate the results. I have defined absorbing state above.

```python
# Modifying above transition probability function to include absorbing state

def modified_transition_probabilities(m, p_up, p_down, p_left, p_right):
  if m <= 1 or not np.isclose(p_up + p_down + p_left + p_right, 1.0):
    raise ValueError("Invalid input")

  transition_probability = np.zeros((m**2+1, m**2+1)) # addition of crashed state

  states = {}
  for state in range(m**2):
    states[state+1] = (state // m, state % m)

  for initial_state in range(m**2):
    for destination_state in range(m**2):
      row_initial_state, column_initial_state = states[initial_state+1]
      row_destination_state, column_destination_state = states[destination_state+1]

      row_difference = row_initial_state - row_destination_state
      column_difference = column_initial_state - column_destination_state

      #print(f"Initial state: ({r_i}, {c_i}), Destination state: ({r_j}, {c_j}), Row Difference: {row_d

      #horizontal movement
      if row_difference == 0: # no movement row wise
        if column_difference == 1: # column movement to left
          transition_probability[initial_state, destination_state] = p_left
        elif column_difference == -1:  # column movement to right
          transition_probability[initial_state, destination_state] = p_right
```

```
      # check boundaries up,down,left,right to ensure the agent does not go out of bounds
      elif column_difference == 0: # no movement column wise
        if row_initial_state == 0: # top row
          transition_probability[initial_state, destination_state] += p_up # increment with p_up
        elif row_initial_state == m - 1: # bottom row
          transition_probability[initial_state, destination_state] += p_down # increment p_down
        if column_initial_state == 0: # leftmost column
          transition_probability[initial_state, destination_state] += p_left # increment p_left
        elif column_initial_state == m - 1: # rightmost column
          transition_probability[initial_state, destination_state] += p_right #  increment p_rgt

    # Vertical Movement
    elif row_difference == 1: # movement up row wise
      if column_difference == 0: # no column movement
        transition_probability[initial_state, destination_state] = p_up
    elif row_difference == -1: # movement down row wise
      if column_difference == 0: # no column movement
        transition_probability[initial_state, destination_state] = p_down

  # set the element in the last column of each row to be equal to the original diagonal element.
  # Then set the original diagonal element to zero since the robot will crash now instead of staying on
  for i in range(m**2):
    transition_probability[i, m**2] = transition_probability[i, i]
    transition_probability[i,i] = 0

  transition_probability[m**2, m**2] = 1 # crashed/absorbing state transitions to itself


  return transition_probability
```

```
new_state = 0
initial_probability = np.append(initial_probability, new_state)
initial_probability
```

```
## array([0., 0., 0., 0., 1., 0., 0., 0., 0., 0.])
```

```
transition_probability = modified_transition_probabilities(m=3, p_up=0.3, p_down=0.2, p_left=0.15, p_ri
time_steps = [0, 1, 3, 5, 10, 20, 100]
# Calculate and print transition probabilities for each time step
for time in time_steps:
  transition_probability_at_timestep_t = np.linalg.matrix_power(transition_probability, time)
  #print(f"Transition Probabilities that govern the system at time step {t}:\n", np.round(Pt, 3))
  print(f"State probabilities at time step {time}:", np.round(np.matmul(initial_probability, transition_
  print("\n")
```

```
## State probabilities at time step 0: [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
##
##
## State probabilities at time step 1: [0.   0.3  0.   0.15 0.   0.35 0.   0.2  0.   0.  ]
##
##
## State probabilities at time step 3: [0.   0.13 0.   0.07 0.   0.16 0.   0.09 0.   0.55]
```

```
##
##
## State probabilities at time step 5: [0.   0.06 0.   0.03 0.   0.07 0.   0.04 0.   0.8 ]
##
##
## State probabilities at time step 10: [0.   0.   0.01 0.   0.01 0.   0.   0.   0.01 0.97]
##
##
## State probabilities at time step 20: [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
##
##
## State probabilities at time step 100: [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
```

Changing the time steps in gradually increasing order, we see that the robot crash probability is the highest. At time step 100, we are 100% certain to see robot in crashed state.

Now, we bring in rewards and further modify transition probability function.

```python
### Getting/collecting rewards for robot until it crashes

def modified_transition_probabilities(m, p_up, p_down, p_left, p_right, timestep):
  if m <= 1 or not np.isclose(p_up + p_down + p_left + p_right, 1.0):
    raise ValueError("Invalid input")

  transition_probability = np.zeros((m**2+1, m**2+1)) # addition of crashed state

  states = {}
  for state in range(m**2):
    states[state+1] = (state // m, state % m)

  for initial_state in range(m**2):
    for destination_state in range(m**2):
      row_initial_state, column_initial_state = states[initial_state+1]
      row_destination_state, column_destination_state = states[destination_state+1]

      row_difference = row_initial_state - row_destination_state
      column_difference = column_initial_state - column_destination_state

      #print(f"Initial state: ({r_i}, {c_i}), Destination state: ({r_j}, {c_j}), Row Difference: {row_d

      #horizontal movement
      if row_difference == 0: # no movement row wise
        if column_difference == 1: # column movement to left
          transition_probability[initial_state, destination_state] = p_left
        elif column_difference == -1:  # column movement to right
          transition_probability[initial_state, destination_state] = p_right

        # check boundaries up,down,left,right to ensure the agent does not go out of bounds
        elif column_difference == 0: # no movement column wise
          if row_initial_state == 0: # top row
            transition_probability[initial_state, destination_state] += p_up # increment with p_up
          elif row_initial_state == m - 1: # bottom row
            transition_probability[initial_state, destination_state] += p_down # increment p_down
          if column_initial_state == 0: # leftmost column
```

```python
            transition_probability[initial_state, destination_state] += p_left # increment p_left
          elif column_initial_state == m - 1: # rightmost column
            transition_probability[initial_state, destination_state] += p_right #  increment p_right

      # Vertical Movement
      elif row_difference == 1: # movement up row wise
        if column_difference == 0: # no column movement
          transition_probability[initial_state, destination_state] = p_up
      elif row_difference == -1: # movement down row wise
        if column_difference == 0: # no column movement
          transition_probability[initial_state, destination_state] = p_down

  # set the element in the last column of each row to be equal to the original diagonal element.
  # Then set the original diagonal element to zero since the robot will crash now instead of transition
  for i in range(m**2):
    transition_probability[i, m**2] = transition_probability[i, i]
    transition_probability[i,i] = 0

  transition_probability[m**2, m**2] = 1 # crashed/absorbing state transitions to itself

  expected_rewards = np.zeros(m**2)

  for state in range(m**2):
    for i in range(timestep):
      crashed = False
      next_state = state
      episode_reward = 0
      #current_timestep = 0
      while not crashed:
        next_state = np.random.choice(m**2+1, p = transition_probability[next_state, :])
        #current_timestep += 1
        if next_state < m**2:
          episode_reward = episode_reward + 1
        else:
          crashed = True
          #print(f"Robot crashed in episode {i + 1} at timestep {current_timestep}")

      expected_rewards[state] = expected_rewards[state] + episode_reward

  expected_rewards = expected_rewards / timestep

  return transition_probability, expected_rewards
```

```python
def print_transition_results(m, p_up, p_down, p_left, p_right, timestep):
  transition_probability, expected_rewards = modified_transition_probabilities(m=m, p_up=p_up, p_down=p

  print("Initial Transition Probabilities:\n", np.round(transition_probability, 3))
  print("Expected Rewards:\n", expected_rewards)

  timestep_t_transition_probability = np.linalg.matrix_power(transition_probability, timestep)
  print(f"Transition Probabilities that govern the system at time step {timestep}:\n", np.round(timeste
```

```
print_transition_results(m=3, p_up=0.3, p_down=0.2, p_left=0.15, p_right=0.35, timestep=100)
```

```
## Initial Transition Probabilities:
##  [[0.   0.35 0.   0.2  0.   0.   0.   0.   0.   0.45]
##  [0.15 0.   0.35 0.   0.2  0.   0.   0.   0.   0.3 ]
##  [0.   0.15 0.   0.   0.   0.2  0.   0.   0.   0.65]
##  [0.3  0.   0.   0.   0.35 0.   0.2  0.   0.   0.15]
##  [0.   0.3  0.   0.15 0.   0.35 0.   0.2  0.   0.  ]
##  [0.   0.   0.3  0.   0.15 0.   0.   0.   0.2  0.35]
##  [0.   0.   0.   0.3  0.   0.   0.   0.35 0.   0.35]
##  [0.   0.   0.   0.   0.3  0.   0.15 0.   0.35 0.2 ]
##  [0.   0.   0.   0.   0.   0.3  0.   0.15 0.   0.55]
##  [0.   0.   0.   0.   0.   0.   0.   0.   0.   1.  ]]
## Expected Rewards:
##  [1.85 2.09 1.15 2.84 3.35 1.97 2.33 2.36 1.17]
## Transition Probabilities that govern the system at time step 100:
##  [[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
##  [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
##  [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
##  [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
##  [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
##  [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
##  [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
##  [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
##  [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
##  [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
```

The expected rewards provide an idea of the reward associated with each state. Higher expected rewards are generally preferred by the agent. The transition probabilities at time step 100 show that, over an extended period, the system is highly likely to end up in a crashed state, regardless of the initial state indicating a convergence to an absorbing state or an undesirable state.

Previously, we mentioned that the agent seeks to maximize the cumulative rewards received from taking some action. In particular, the agent seeks to maximize the expectation of the reward sequence which is the expected returns. By definition, the returns of the agent can be represented as:

$$G_t := R_{t+1}, R_{t+2} + R_{t+3} + R_{t+4} + ... + R_T$$

where T is the final time step. When there is a final time step, we call these *episodic*-tasks. Each episode ends in a terminal state and resets such as winning or losing the game. The next episode begins independently of the previous episodes. All episodes ends in the same terminal state with different rewards for the different outcomes (outcome win the game: reward $+1$, outcome lost the game: penalty -1). Until it reaches the outcome, the rewards will be zero that is we do not reward or penalize the agent for sub-tasks. The final time step random variable varies episode to episode.

In the case of *continuing*-tasks, the interaction between agent and environment does not have final time step but rather goes on continuously. In such cases, we need to discount the reward received by the agent over the future. The agent tries to select actions so that the sum of discounted rewards it receives over the future is maximized. In particular, it choose A_t to maximize the expected discounted return. By definition,

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

where γ is the discount factor between $[0, 1]$. This discount factor affects how far-sighted the agent needs to be. When $\gamma = 0$, then the agent is *myopic* and is only concerned about maximizing the immediate rewards. In order to do so, the agent maximizes each immediate reward separately.

When γ is close to 1, the agent strongly takes into account the future rewards to make a decision now.

The returns at successive time steps are related by the below equation:

$$\begin{aligned} G_t &:= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + ... \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

If the reward is nonzero and constant for example, +1 then the return is

$$G_t = \sum_{k=0}^{inf} \gamma^k = \frac{1}{1-\gamma}$$

To unify both episodic tasks and continuing tasks, consider the time step T as the absorbing state that only generates reward of 0. Starting from the initial state S_0, when we sum the rewards, we get the same returns. We have one general returns formula,

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

In the above equation, T can be ∞ or γ can be 1 but not both.

Now we have an understanding of episodic tasks, continuing tasks, returns and discounted returns we turn back to the algorithmic discussion. Since we do not know the true value of how good it is to perform a given action in a given state, we estimate them using value functions. The value functions are defined with respect to policies which maps states to probabilities of selecting each possible action.

If the agent is following policy π at time t then $\pi(a|s)$ is the probability of taking the action $A_t = a$ if the agent is in state $S_t = s$.

The state-value function, $v_\pi(s)$, of state s under policy π is the expected return starting from state s and following π afterwards. For MDP, the state-value function for policy π is given by,

$$v_\pi(s) := E_\pi[G_t|S_t = s, \pi] = E_\pi[\sum_{k=0}^{inf} \gamma^k R_{t+k+1}|S_t = s]$$
$$\forall s \in S.$$

Similarly, the action-value function, $q_\pi(s, a)$, of taking action a in state s under policy π is the expected return starting from s, taking action a and following π afterwards. It is given by,

$$q_\pi(s, a) := E_\pi[G_t|S_t = s, A_t = a, \pi] = E_\pi[\sum_{k=0}^{inf} \gamma^k R_{t+k+1}|S_t = s, A_t = a]$$

Connection between state-value function and action-value function is shown below:

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s, a) = E[q_\pi(S_t, A_t)|S_t = s, \pi]$$

,

$$\forall s$$

.

The optimal state-value function $v_*(s)$ is the maximum value over all policies,

$$v_*(s) = max_\pi v_\pi(s)$$

The optimal action-value function

$$q_*(s, a) = max_\pi q_\pi(s, a)$$

To find the optimal policy, we maximize over $q_*(s, a)$ that is,

$$\pi_*(s, a) = 1$$

if

$$a = argmax_{a \in A} q_*(s, a)$$

and 0 otherwise.

Credits: Sutton & Barto Enes Bilgin