University of Isfahan

Cliff Walking Project

Authors:

Melika Shirian

Kianoosh Vadaei

Fundamentals and Applications of Artificial Intelligence

Dr. Hossein Karshenas

2023-11-08

Introduction

In the realm of artificial intelligence and machine learning, Markov Decision Processes (MDPs) have emerged as a powerful framework for modeling and solving complex decision-making problems. This project delves into the application of MDPs in addressing a challenging scenario known as the "Cliff Walking Game." This game represents a practical and illustrative example where an intelligent agent must navigate a grid-world environment fraught with risks, akin to traversing a treacherous cliff.

Code Explanation

In this project, implemented using the Gymnasium library, the goal is to determine an optimal policy for the environment, meaning specifying the exact action the agent should take in each state.

To find this optimal policy, we utilize the value iteration method. To accomplish this, we initially recalculate the reward for each state.

```python
1   def set_rewards():
2       rewards = {}
3       for state in range(48):
4           column = state % 12
5           row = int(state / 12)
6           dist_row = abs(row - 3)
7           dist_column = abs(column - 11)
8           if (column >= 0) and (column <= 7):
9               rewards[state] = - 0.3 * (dist_row + dist_column)
10          else:
11              rewards[state] = - 0.2 * (dist_row + dist_column)
12
13      return rewards
```

Here, initially, the state, represented by a number between 0 and 47, is mapped to a row and column based on the following formula. Then, the Manhattan distance of the state from the goal state is calculated.

The environment is divided into two sections, where for states located after the seventh column, the impact of Manhattan distance is greater. Given that the numbers are in the negative range, columns greater than 7 are multiplied by a smaller value to bring them closer to zero and have a more pronounced effect.

After calculating the Manhattan distance, the reward for each state is determined by summing the row and column of the Manhattan distance (multiplied by a negative one coefficient to reflect the negative impact). This function serves as a heuristic and is obtained through trial and error.

$$observation = currentrow * ncols + currentcol$$

```python
1  def get_neighbors(state):
2      if state == 0:
3          return (0, 1, 3)
4      elif state == 1:
5          return (1, 0, 2)
6      elif state == 2:
7          return (2, 1, 3)
8      elif state == 3:
9          return (3, 0, 2)
```

This function uniquely determines, for each action, the possible actions that may be performed.

The main part of the work takes place in the **value_iteration** function. Due to its length, we will explain different sections of this function separately in the following.

```python
1  # initalize
2  V = {}
3  for state in range(48):
4      V[state] = 0.
```

Initially, a dictionary (map) is created for the state values, and the values for all states are set to zero. Subsequently, these values are intended to be dynamically updated.

Next, there are several internal functions, and I will elaborate on each of them.

```python
1  def possible_actions(state):
2      possible_actions = []
3      for action in range(4):
4          if not (action == 3 and state == 0):
5              possible_actions.append(action)
6
7      return possible_actions
```

The function **possible_actions(state)** essentially checks a condition for each state. If the agent is in position 0, it does not have the right to take the left action because it would be futile.

With this approach, we essentially allow the agent to choose actions that might not be feasible in themselves, but considering possible future states can lead to a better policy for us. For example, in the bottom row, if there is no cliff, the agent should be able to directly move to the right and collect the cookie. Here, our algorithm assigns action 2 (down) to all states in the bottom row. Although executing this action might be impossible, there is a one-third chance that the agent could move either to the right or left. This means the probability of moving upward becomes zero, and this policy would be more favorable.

```python
1   def possible_actions2(state):
2       possible_actions = []
3       for action in range(4):
4           next_state = step(state, action)
5
6           left_side = state % 12
7           right_side = (state % 12) - 11
8
9           if not (action == 3 and left_side == 0) and not (action == 1 and right_side == 0):
10              if next_state >= 0 and next_state < 48:
11                  possible_actions.append(action)
12      return possible_actions
```

In this project, we practically do not use the **possible_actions2(state)** function. As a precaution to ensure the previous implementation is allowed, we have implemented this function.

This function essentially restricts the agent from selecting actions that might render the actions themselves infeasible. For instance, in the previous example, the agent in the bottom row cannot execute the down action.

```
1   def step(state, action):
2       if action == 0:
3           return state - 12
4       elif action == 1:
5           return state + 1
6       elif action == 2:
7           return state + 12
8       elif action == 3:
9           return state - 1
```

Because we're not using Gymnasium library functions for policy calculation in this project, we have to implement the **step** function. This function takes the current state and action, and applies the action in a straightforward manner to the state.

Certainly, there might be future states that are not desirable, like being negative or exceeding 47. We plan to address this issue in the future.

```
1   def Q(state, action):
2       sum = 0
3       left_side = state % 12
4       right_side = (state % 12) - 11
5       for possible_action in get_neighbors(action):
6
7           next_state = step(state, possible_action)
8           if (possible_action == 3 and left_side == 0) or (possible_action == 1 and right_side == 0) or \
9                   next_state < 0 or next_state > 47:
10              sum += -1
11              continue
12          if next_state in cliff_positions:
13              reward = -50
14              prob = float(1/3)
15          elif next_state == 47:   #End State
16              reward = 0
17              prob = 1/3
18          else:
19              reward = rewards[next_state]
20              prob = float(1/3)
21          q = prob * (float(reward) + 0.9 * V[next_state])
22          sum += q
23
24      return sum
```

In the **Q(state, action)** function, our goal is to calculate the Q-value for a specific state and a specific action, following the Bellman equation. This is done to compute the dynamic value of each state.

$$Q^*(s, a) = \sum_{s'} T(s' \mid s, a) \left(R(s, a, s') + \gamma V^*(s') \right)$$

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

In this function, we first check whether the current state is in the rightmost or leftmost column by calculating **left_side** and **right_side**. This depends on the values in those positions; for example, rightmost cells are 11, 23, 35, and 47, while leftmost cells are 0, 12, 24, and 36.

Next, for the action we are calculating in the Q function, we iterate over all possible actions that might occur in a loop. For each action, we calculate its Q value and after loop we return the sum.

Inside the loop, we check for each possible action if it might happen in the environment. For instance, if we are in the top row, the up action might result in a negative value for the **next_state**, which is not possible. Or if we are in the leftmost column, the left action is not possible. We calculate this condition for all four sides of the environment that are sensitive.

If these conditions occur, the value of **sum** is reduced by one unit. This is because all our values are in the range of negative numbers. If **sum** is not reduced by one unit, it would essentially increase by zero, which is not desired.

For **next_state:**

- If it is a cliff, its reward immediately becomes -100, and the probability is one-third.
- If **next_state** is the goal state, its reward becomes 0 (which is the maximum reward), and the probability is one-third.
- If it is none of these, it's a regular state, and its reward is assigned from the rewards dictionary (calculated in the **set_rewards** function based on Manhattan distance). The probability is one-third in this case.

Finally, we calculate **q** with a discount factor of 0.9 (commonly used), and this value is added to the sum.

Now, in a while loop until the values have not converged, we perform value iteration.

This loop consists of two parts, each of which we will explain separately. The codes you will see in the future are related to this loop.

```python
1  newV = {}
2  for state in range(48):
3      if state == 47:
4          newV[state] = 0
5      elif state in cliff_positions:
6          newV[state] = -50
7      else:
8          newV[state] = max(Q(state, action) for action in possible_actions(state))
9  # checking for convergence
10 if max(abs(V[state] - newV[state]) for state in range(48)) < 1e-10 :
11     break
12 V = newV
```

For the current state in this function:

- If it's the goal state, its new value will be zero.

- If it's a cliff, its new value will be -100.

- If none of the above conditions apply (it's a regular state), for all actions (except the left action if the state is in cell 0, as previously explained), we calculate **q**, and the new value for that state becomes the maximum **q** calculated.

Then, for each of the 48 states, the difference between the old value and the new value is calculated. If it is less than 10^{-10}, it means the values have not changed much, and they have converged to a certain number. In case of convergence, we exit the loop.

After that, the values dictionary is replaced with the new values.

```
1  pi = {}
2  for state in range(48):
3      if state == 47:
4          pi[state] = 'none'
5      else:
6          pi[state] =max((Q(state, action), action) for action in possible_actions(state))[1]
```

Continuing with the loop, we move on to policy extraction. For all states, if it's the goal state, its policy becomes 'none'. Otherwise, we need to calculate Q again for all possible actions in each state (where all actions are possible except left action for state 0). The action that leads to the maximum Q in each state is the action the agent should take in that state.

To achieve this, we use the tuple data structure. The first element of the tuple is the Q value, and the second element is the action for which Q is calculated. Then, we find the tuple with the maximum Q value (which is essentially the maximum of its first element, i.e., Q), and we put the second element (the action) into the policy dictionary.

This dictionary represents our policy, indicating the action the agent should take in each state.

Finally, when the values of states converge, the **value_iteration** function returns the obtained policy and values.

Continuing, let's delve into the main section of the code and utilize the functions we have written so far.

```
1   env = CliffWalking(render_mode="human")
2   observation, info = env.reset(seed=30)
3
4   cliff_positions = []
5
6   for cliff_pos in env.cliff_positions:
7       cliff_positions.append(cliff_pos[0] * 12 + cliff_pos[1])
8
9   rewards = set_rewards()
10
11  pi , v = value_iteration(cliff_positions, rewards)
```

Here, we first create the game environment and then store the positions of all cliffs in the **cliff_positions** list. Afterward, we call the **value_iteration** function, where our optimal policy is calculated.

```
1   max_iter_number = 1000
2   for __ in range(max_iter_number):
3
4       action = pi[env.s]
5       next_state, reward, done, truncated, info = env.step(action)
6
7       if done or truncated:
8           observation, info = env.reset()
9
10  # Close the environment
11  env.close()
```

Here, in a loop iterating 1000 times, we use the policy to determine the agent's action based on its current position. Then, the agent applies that action using the **step** function.

In each iteration, if the agent reaches the goal state or a cliff, the environment is reset, and the agent starts again from the initial point.

Finally, after 1000 iterations, the environment is closed.
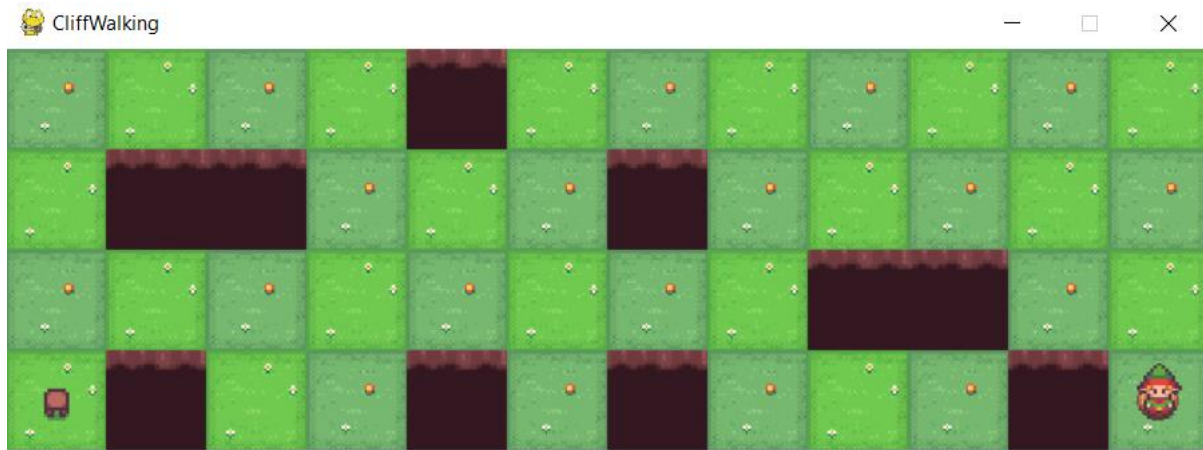
References

- Stanford University. (2019). Markov Decision Processes 1 - Value Iteration. Retrieved from this link.

- OpenAI. "ChatGPT." https://www.openai.com/

- Russell, Stuart, and Norvig, Peter. "Artificial Intelligence: A Modern Approach." (Book)