Implementation of Value Iteration

Rajan Adhikari

September 21, 2024

[1]:
```
# import required libraries
import gymnasium as gym
import numpy as np

# set seed
SEED = 106

# set discount factor
GAMMA = 0.8
```

[2]:
```
# Initialize the Frozen Lake Environment
env = gym.make('FrozenLake-v1', map_name="4x4", is_slippery=False,␣
 ↪render_mode='ansi')
```

[3]:
```
env.reset(seed=SEED)
print(env.render())
```

```
SFFF
FHFH
FFFH
HFFG
```

[4]:
```
def value_iteration(env, num_of_iterations=10, gamma=1.0, threshold=1e-40) ->␣
 ↪list:
    """
    Value iteration algorithm to compute the value table.
    :param env: environment for an agent
    :param num_of_iterations: number of iterations
    :param gamma: discount factor
    :param threshold: threshold value to stop iterations
    :return: value table
    """
    # Get the number of states and actions in the environment
    num_of_states = env.observation_space.n
    num_of_actions = env.action_space.n
```

```
    print('Number of states:', num_of_states)
    print('Number of actions:', num_of_actions)

    # Initialize the value table with zeros for each state
    value_table = np.zeros(num_of_states)

    # Perform value iteration for num_of_iterations
    for i in range(num_of_iterations):
        updated_value_table = np.copy(value_table)

        # Compute q value for each state
        for state in range(num_of_states):
            # Initialize q values
            q_values = []

            # For each action in the state, compute q value
            for action in range(num_of_actions):
                q_value = 0
                # Loop through each transition (prob, next_state, reward, done)
                for prob, next_state, reward, _ in env.unwrapped.
↪P[state][action]:
                    # Compute Bellman backup
                    bellman_backup = reward + gamma *␣
↪updated_value_table[next_state]
                    # Compute q value
                    q_value += prob * bellman_backup

                # Append q value to the list of q values
                q_values.append(q_value)

            # Update the value table with the maximum q value
            value_table[state] = max(q_values)

        # Check for convergence
        if np.sum(np.fabs(updated_value_table - value_table)) <= threshold:
            print("Execution halted in iteration {}.".format(i))
            break

    return value_table
```

[5]:
```
# Print Value Table
optimal_value_table = value_iteration(env, num_of_iterations = 10, gamma=GAMMA)
print(optimal_value_table)
```

```
Number of states: 16
Number of actions: 4
Execution halted in iteration 6.
[0.32768 0.4096  0.512   0.4096  0.4096  0.      0.64    0.      0.512
```

```
0.64    0.8     0.      0.      0.8     1.      0.     ]
```

[6]:
```python
def extract_policy(env, value_table, gamma=1.0):
    """
    Extract the policy from the given value table.
    :param env: environment for an agent
    :param value_table: value table computed from value iteration
    :param gamma: discount factor
    :return: policy table
    """
    # Get the number of states and actions in the environment
    num_of_states = env.observation_space.n
    num_of_actions = env.action_space.n

    # Initialize policy as an integer array
    policy = np.zeros(num_of_states, dtype=int)

    # Iterate through each state
    for state in range(num_of_states):
        q_values = []

        # For each action in the state, compute q value
        for action in range(num_of_actions):
            q_value = 0

            # Calculate q value using the transition probabilities
            for prob, next_state, reward, _ in env.unwrapped.P[state][action]:
                bellman_backup = reward + gamma * value_table[next_state]
                q_value += prob * bellman_backup

            # Append q value to the list
            q_values.append(q_value)

        # Select the action with the highest q value
        policy[state] = np.argmax(q_values)

    return policy
```

[7]:
```python
optimal_policy = extract_policy(env, optimal_value_table, gamma=GAMMA)
print(optimal_policy)
```

```
[1 2 1 0 1 0 1 0 2 1 1 0 0 2 2 0]
```

[8]:
```python
# Let's decode the action to be take in each state
# map numbers to action
action_map = {
    0: "left",
    1: "down",
```

```python
    2: "right",
    3: "up"
}

# Number of times the agent moves
num_timestep = 100

# Reset the environment
current_state, info = env.reset(seed=SEED)

for i in range(num_timestep):
    print("------- Step: {} --------".format(i+1))
        # Let's take a random action now from the action space
    # Random action means we are taking random policy at the moment.
    action = optimal_policy[current_state]

    # # Take the action and get the new observation space
    next_state, reward, done, info, transition_prob = env.step(action)

    print("Current State: {}".format(current_state))
    print("Action: {}".format(action_map[action]))
    print("Next State: {}".format(next_state))
    print("Reward: {}".format(reward))
    current_state = next_state

    # if the agent moves to hole state, then terminate
    if done:
        break
```

```
------- Step: 1 --------
Current State: 0
Action: down
Next State: 4
Reward: 0.0
------- Step: 2 --------
Current State: 4
Action: down
Next State: 8
Reward: 0.0
------- Step: 3 --------
Current State: 8
Action: right
Next State: 9
Reward: 0.0
------- Step: 4 --------
Current State: 9
Action: down
Next State: 13
```

```
Reward: 0.0
------- Step: 5 --------
Current State: 13
Action: right
Next State: 14
Reward: 0.0
------- Step: 6 --------
Current State: 14
Action: right
Next State: 15
Reward: 1.0
```