Implementation of Policy Iteration

Rajan Adhikari

September 21, 2024

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

# set seed
SEED = 106

# set discount factor
GAMMA = 0.8
```

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

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

```
SFFF
FHFH
FFFH
HFFG
```

[4]:
```python
def compute_policy_value(env, policy, gamma=1.0, threshold=1e-10):
    """
    Evaluate the value function for a given policy.
    :param env: environment for an agent
    :param policy: policy to evaluate
    :param gamma: discount factor
    :param threshold: convergence threshold
    :return: value table for the given policy
    """
    num_of_states = env.observation_space.n
    value_table = np.zeros(num_of_states)

    while True:
        updated_value_table = np.copy(value_table)
```

```
        # Iterate through each state to evaluate its value
        for state in range(num_of_states):
            action = policy[state]
            q_value = 0

            # For each action, compute its value using transition probabilities
            for prob, next_state, reward, _ in env.unwrapped.P[state][action]:
                bellman_backup = reward + gamma * updated_value_table[next_state]
                q_value += prob * bellman_backup

            value_table[state] = q_value

        # Check for convergence
        if np.sum(np.fabs(updated_value_table - value_table)) <= threshold:
            break

    return value_table
```

[5]:

```
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
```

```python
            # 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
```

[6]:

```python
def policy_iteration(env, gamma=1.0, num_of_iterations=1000, threshold=1e-10):
    """
    Policy Iteration algorithm to compute the optimal policy and value table.
    :param env: environment for an agent
    :param gamma: discount factor
    :param num_of_iterations: max number of iterations for the algorithm
    :param threshold: convergence threshold
    :return: optimal policy and value table
    """
    num_of_states = env.observation_space.n
    num_of_actions = env.action_space.n

    # Initialize a random policy (or all zeros)
    policy = np.random.choice(num_of_actions, num_of_states)

    for i in range(num_of_iterations):
        # Step 1: Policy Evaluation (compute the value of the current policy)
        value_table = compute_policy_value(env, policy, gamma, threshold)

        # Step 2: Policy Improvement (extract new policy based on the current␣
 ↪value table)
        new_policy = extract_policy(env, value_table, gamma)

        # Check for convergence (if policy doesn't change, stop)
        if np.all(policy == new_policy):
            print(f"Policy converged at iteration {i}.")
            break

        policy = new_policy

    return policy, value_table
```

[7]:

```python
# Run policy iteration
optimal_policy, optimal_value_table = policy_iteration(env, gamma=GAMMA)

print("Optimal Policy:", optimal_policy)
print("Optimal Value Table:", optimal_value_table)
```

```
Policy converged at iteration 6.
```

```
Optimal Policy: [1 2 1 0 1 0 1 0 2 1 1 0 0 2 2 0]
Optimal Value Table: [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.     ]
```

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

[8]:

```

```