POLICY EVALUATION

AIM

The aim is to implement a reinforcement learning agent that can navigate environments from the gym-walk library, which simulates grid-like or walking environments. The agent will learn an optimal policy to reach desired goals based on the given reward structure.

PROBLEM STATEMENT

The task is to implement and evaluate a reinforcement learning agent in a walking environment using gym. The agent must learn to make decisions that maximize its total reward through trial and error, based on feedback from the environment.

POLICY EVALUATION FUNCTION

The policy evaluation function aims to compute the value of a given policy by iteratively calculating the expected rewards of following the policy in each state until convergence, allowing for better estimation of state values under the current policy.

PROGRAM

Pip installing

```
pip install git+https://github.com/mimoralea/gym-walk#egg=gym-walk
```

Importing Libraries

```
import warnings ; warnings.filterwarnings('ignore')

import gym, gym_walk
import numpy as np

import random
import warnings

warnings.filterwarnings('ignore', category=DeprecationWarning)
np.set_printoptions(suppress=True)
```

```
random.seed(123); np.random.seed(123)
```

Printing Functions (Policy, State Value, Probabailty and Mean returns)

```
def print_policy(pi, P, action_symbols=('<', 'v', '>', '^'), n_cols=4,
title='Policy:'):
    print(title)
    arrs = {k:v for k,v in enumerate(action_symbols)}
    for s in range(len(P)):
        a = pi(s)
        print("| ", end="")
        if np.all([done for action in P[s].values() for _, _, _, done in
action]):
            print("".rjust(9), end=" ")
        else:
            print(str(s).zfill(2), arrs[a].rjust(6), end=" ")
        if (s + 1) % n_cols == 0: print("|")

def print_state_value_function(V, P, n_cols=4, prec=3, title='State-value
function:'):
    print(title)
    for s in range(len(P)):
        v = V[s]
        print("| ", end="")
        if np.all([done for action in P[s].values() for _, _, _, done in
action]):
            print("".rjust(9), end=" ")
        else:
            print(str(s).zfill(2), '{}'.format(np.round(v, prec)).rjust(6),
end=" ")
        if (s + 1) % n_cols == 0: print("|")

def probability_success(env, pi, goal_state, n_episodes=100, max_steps=200):
    random.seed(123); np.random.seed(123) ; env.seed(123)
    results = []
    for _ in range(n_episodes):
        state, done, steps = env.reset(), False, 0
        while not done and steps < max_steps:
            state, _, done, h = env.step(pi(state))
            steps += 1
        results.append(state == goal_state)
    return np.sum(results)/len(results)

def mean_return(env, pi, n_episodes=100, max_steps=200):
    random.seed(123); np.random.seed(123) ; env.seed(123)
    results = []
    for _ in range(n_episodes):
        state, done, steps = env.reset(), False, 0
        results.append(0.0)
```

```
        while not done and steps < max_steps:
            state, reward, done, _ = env.step(pi(state))
            results[-1] += reward
            steps += 1
    return np.mean(results)
```

Slippary Walk 5 MDP

```
env = gym.make('SlipperyWalkFive-v0')
P = env.env.P
init_state = env.reset()
goal_state = 6
LEFT, RIGHT = range(2)

P

init_state

state, reward, done, info = env.step(RIGHT)
print("state:{0} - reward:{1} - done:{2} - info:{3}".format(state, reward,
done, info))

# First Policy
pi_1 = lambda s: {
    0:LEFT, 1:LEFT, 2:LEFT, 3:LEFT, 4:LEFT, 5:LEFT, 6:LEFT
}[s]
print_policy(pi_1, P, action_symbols=('<', '>'), n_cols=7)

# Find the probability of success and the mean return of the first policy
print('Reaches goal {:.2f}%. Obtains an average undiscounted return of
{:.4f}.'.format(
    probability_success(env, pi_1, goal_state=goal_state)*100,
    mean_return(env, pi_1)))

# Create your own policy
pi_2=lambda s:{
    0:RIGHT, 1:RIGHT, 2:RIGHT, 3:RIGHT, 4:RIGHT, 5:RIGHT, 6:RIGHT
}[s]

print_policy(pi_2, P, action_symbols=('<', '>'), n_cols=7)

# Find the probability of success and the mean return of the first policy
print('Reaches goal {:.2f}%. Obtains an average undiscounted return of
{:.4f}.'.format(
    probability_success(env, pi_1, goal_state=goal_state)*100,
    mean_return(env, pi_1)))
```

Policy Evaluation

```
import numpy as np

def policy_evaluation(pi, P, gamma=1.0, theta=1e-10):
    V = np.zeros(len(P), dtype=np.float64)

    while True:
        prev_V = np.copy(V)
        delta = 0

        for s in range(len(P)):
            v = 0

            a = pi(s)

            for prob, next_state, reward, done in P[s][a]:
                v += prob * (reward + gamma * prev_V[next_state] * (not
done))

            V[s] = v

            delta = max(delta, np.abs(prev_V[s] - V[s]))

        if delta < theta:
            break

    return V

# Code to evaluate the first policy
V1 = policy_evaluation(pi_1, P)
print_state_value_function(V1, P, n_cols=7, prec=5)

# Code to evaluate the second policy
# Write your code here
# Code to evaluate the first policy
V2 = policy_evaluation(pi_2, P)
print_state_value_function(V2, P, n_cols=7, prec=5)

V1

print_state_value_function(V1, P, n_cols=7, prec=5)

V2

print_state_value_function(V2, P, n_cols=7, prec=5)

V1>=V2

if(np.sum(V1>=V2)==7):
  print("The first policy is the better policy")
elif(np.sum(V2>=V1)==7):
  print("The second policy is the better policy")
```

```
  else:
    print("Both policies have their merits.")
```

OUTPUT:

Policy 1:

```
V1
```

```
array([0.        , 0.00274725, 0.01098901, 0.03571429, 0.10989011,
       0.33241758, 0.        ])
```

```
print_state_value_function(V1, P, n_cols=7, prec=5)
```

```
State-value function:
|          | 01 0.00275 | 02 0.01099 | 03 0.03571 | 04 0.10989 | 05 0.33242 |
```

Policy 2:

```
V2
```

```
array([0.        , 0.66758242, 0.89010989, 0.96428571, 0.98901099,
       0.99725275, 0.        ])
```

```
print_state_value_function(V2, P, n_cols=7, prec=5)
```

```
State-value function:
|          | 01 0.66758 | 02 0.89011 | 03 0.96429 | 04 0.98901 | 05 0.99725 |
```

Comparision:

```
V1>=V2
```

```
array([ True, False, False, False, False, False,  True])
```

```
if(np.sum(V1>=V2)==7):
  print("The first policy is the better policy")
elif(np.sum(V2>=V1)==7):
  print("The second policy is the better policy")
else:
  print("Both policies have their merits.")
```

```
The second policy is the better policy
```

RESULT:

Thus, the reinforcement learning agent successfully learns an optimal policy for navigating the environment, improving its decisions through iterations.