D_case

Jaemin Park

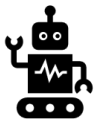
2021-02-14

차 례

Case introduction

Problem discription

Robot is playing game. Robot starts at state 4, and there is coin in state 1.

1	2	3	4	5	6	7

When robot gain coin, then finish flag is appear at state 7.


1	2	3	4	5	6	7

If robot gain finish flag, then game is over. One thing to notice is that when robot chooses suicide it self(state = 1, action = left or state = 7, action = right), robot will be restart to its first starting position(state 4) with no disadvantage.

Purpose of this case is to find out fastest route to finish game. Sometimes if suicide makes better solution, will robot try to suicide itself?

State = {1,2,3,4,5,6,7} Initial transition probability = Left 0.5, Right 0.5
Reward = each step -1, get coin +10, finish flag +50

Code Implementation

This time, finding best routh algorithm uses Monte Carlo method using just Initial transition probability. After learning ϵ -greedy method, code will be updated

```
import numpy as np
import pandas as pd

#if robot gain coin, then True
Coin_Flag = False
#States
states = np.arange(1,8,1)
#action left matrix
P_L = np.matrix([[0,0,0,1,0,0,0],[1,0,0,0,0,0,0],
[0,1,0,0,0,0,0],[0,0,1,0,0,0,0],[0,0,0,1,0,0,0],
[0,0,0,0,1,0,0],[0,0,0,0,0,1,0]])
#action right matrix
P_R = np.matrix([[0,1,0,0,0,0,0],[0,0,1,0,0,0,0],
[0,0,0,1,0,0,0],[0,0,0,0,1,0,0],[0,0,0,0,0,1,0],
[0,0,0,0,0,0,1],[0,0,0,1,0,0,0]])
P_L = pd.DataFrame(P_L,states,states)
P_R = pd.DataFrame(P_R,states,states)

#RSA1 - Robot not have coin
R_s_a_1 = np.matrix([[-1,+10,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1,-1]]).T
R_s_a_1 = pd.DataFrame(R_s_a_1,states,["L","R"])
#RSA2 - Robot have coin
R_s_a_2 = np.matrix([[-1,-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,+50,-1]]).T
R_s_a_2 = pd.DataFrame(R_s_a_2,states,["L","R"])
print(R_s_a_1.T)
```

```
##    1   2  3  4  5  6  7
## L -1  10 -1 -1 -1 -1 -1
## R -1  -1 -1 -1 -1 -1 -1
```

```
print(R_s_a_2.T)

#QSA1 - Robot not have coin
```

```
##    1  2  3  4  5   6  7
## L -1 -1 -1 -1 -1  -1 -1
## R -1 -1 -1 -1 -1  50 -1
```

```
q_s_a_init_1 = np.hstack((np.zeros(len(states)).reshape(7,1),np.zeros(len(states)).reshape(7,1)))
q_s_a_init_1 = pd.DataFrame(q_s_a_init_1,states,["L","R"])
#QSA2 - Robot have coin
q_s_a_init_2 = np.hstack((np.zeros(len(states)).reshape(7,1),np.zeros(len(states)).reshape(7,1)))
q_s_a_init_2 = pd.DataFrame(q_s_a_init_2,states,["L","R"])
print(q_s_a_init_1.T)
```

```
##      1    2    3    4    5    6    7
## L  0.0  0.0  0.0  0.0  0.0  0.0  0.0
## R  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

```
print(q_s_a_init_2.T)
```

```
##      1    2    3    4    5    6    7
## L  0.0  0.0  0.0  0.0  0.0  0.0  0.0
## R  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

```
#R2
#left right 50,50 policy
pi_50 = np.hstack((np.repeat(0.5,len(states)).reshape(7,1),np.repeat(0.5,len(states)).reshape(7,1)))
pi_50 = pd.DataFrame(pi_50,states,["L","R"])
print(pi_50.T)

#R3
#path simulator using pi_50
```

```
##      1    2    3    4    5    6    7
## L  0.5  0.5  0.5  0.5  0.5  0.5  0.5
## R  0.5  0.5  0.5  0.5  0.5  0.5  0.5
```

```
def simul_path(pi,P_L,P_R,R_s_a_1,R_s_a_2,Coin_Flag):
    s_now = 4
    history_i = []
    while(not Coin_Flag or s_now!=7):
        if(np.random.uniform(0,1)<pi.loc[s_now]["L"]):
            a_now="L"
            P=P_L
        else:
            a_now="R"
            P=P_R
        if(not Coin_Flag):
            r_now = R_s_a_1.loc[s_now][a_now]
        else:
            r_now = R_s_a_2.loc[s_now][a_now]

        s_next = states[np.argmin(P.loc[s_now].cumsum()<np.random.uniform(0,1))].item()
        history_i.extend([s_now,a_now,r_now])
        s_now = s_next

        if(s_now ==1 and not Coin_Flag):
            Coin_Flag = True

    return(history_i)

sample_path = simul_path(pi_50, P_L, P_R, R_s_a_1, R_s_a_2, Coin_Flag)
```

TD method function simul_step is not used this time

```
#R4
#sample step simulator. This time, we are using MC method, This TD method function is not used.
def simul_step(pi,s_now,P_L,P_R,R_s_a_1,R_s_a_2,Coin_Flag):

    if(np.random.uniform(0,1)<pi.loc[s_now]["L"]):
        a_now="L"
        P=P_L
    else:
        a_now="R"
        P=P_R
    if(not Coin_Flag):
        r_now = R_s_a_1.loc[s_now][a_now]
    else:
        r_now = R_s_a_2.loc[s_now][a_now]

    s_next = states[np.argmin(P.loc[s_now].cumsum()<np.random.uniform(0,1))].item()
    if(s_next ==1 and not Coin_Flag):
            Coin_Flag = True

    if(np.random.uniform(0,1)<pi.loc[s_next]["L"]):
        a_next="L"
    else:
        a_next="R"
    sarsa = [s_now,a_now,r_now,s_next,a_next]
    return(sarsa,Coin_Flag)

sample_step,flag = simul_step(pi_50, 4, P_L, P_R, R_s_a_1,R_s_a_2,Coin_Flag) # a.k.a. sarsa
Coin_Flag = flag
#print(sample_step)
```

```
#R5
#policy evaluation
def pol_eval_MC(sample_path, q_s_a_1, q_s_a_2, alpha):
    flag_qsa = False
    for j in range(0,len(sample_path),3):
        if(sample_path[j+2] == 10):
            flag_qsa = True

        s = sample_path[j]
        a = sample_path[j+1]
        G = pd.Series(sample_path)[range(j+2,len(sample_path)-1,3)].astype('float').sum()

        if(not flag_qsa):
            q_s_a_1.loc[s][a] = q_s_a_1.loc[s][a] + alpha*(G-q_s_a_1.loc[s][a])
        else:
            q_s_a_2.loc[s][a] = q_s_a_2.loc[s][a] + alpha*(G-q_s_a_2.loc[s][a])

    return (q_s_a_1,q_s_a_2)
q_s_a_1, q_s_a_2 = pol_eval_MC(sample_path,q_s_a_init_1,q_s_a_init_2,0.1)

#policy improve
def pol_imp(q_s_a,epsilon):
    pi = np.hstack((np.zeros(len(states)).reshape(7,1),np.zeros(len(states)).reshape(7,1)))
    pi = pd.DataFrame(pi,states,["L","R"])
    for i in range(pi.shape[0]):
        if(np.random.uniform(0,1)>epsilon):
            pi.iloc[i] = 0
            pi.iloc[i][q_s_a.iloc[i].idxmax()]=1
        else:
            pi.iloc[i] = 1/q_s_a.shape[1]
    return pi
```

```
MC_N = 100
for i in range(MC_N):
    sample_path = simul_path(pi_50, P_L, P_R, R_s_a_1, R_s_a_2, Coin_Flag )
    #print(sample_path)
    q_s_a_1, q_s_a_2 = pol_eval_MC(sample_path,q_s_a_1,q_s_a_2,0.1)
pi_1 = pol_imp(q_s_a_1, 0)
pi_2 = pol_imp(q_s_a_2, 0)
print(q_s_a_1.T)
```

```
##      1         2          3          4          5          6          7
## L  0.0   0.00000 -13.857645 -14.233698 -19.466698 -23.938813 -35.648475
## R  0.0 -23.57222 -15.576322 -19.513257 -21.878094 -26.560060 -30.288991
```

```
print(q_s_a_2.T)
```

```
##            1          2          3          4         5         6    7
## L -15.711760 -11.479641 -15.554002 -15.638392 -9.680769 -8.814882  0.0
## R -18.483655 -13.601490 -11.735704  -6.333459 -4.538508  0.000000  0.0
```

```
print(pi_1.T)
```

```
##      1    2    3    4    5    6    7
## L  1.0  1.0  1.0  1.0  1.0  1.0  0.0
## R  0.0  0.0  0.0  0.0  0.0  0.0  1.0
```

```
print(pi_2.T)
```

```
##      1    2    3    4    5    6    7
## L  1.0  1.0  0.0  0.0  0.0  0.0  1.0
## R  0.0  0.0  1.0  1.0  1.0  1.0  0.0
```

Result

Before having coin at the state 6 and 7, to reach state 1, selecting suicide is better option than just going to state 1. Also after having coin to reach state 7, at the state 1 and 2, suicide option is faster than just going to state 7.
Since this code uses only pi_50 policy with MC method, result may not be accurate. But we can find out that robot similarly calculated that suicide can make some better option.