**Neural Networks - HW8**

By Nikita Thakur

a)

The number of actions possible for this problem are **4** : Up, Down, Left and Right

For this simulation of the problem:

Size of the Grid (n\*n) = 5\*5

Gold Limit for Sue = 3

The number of states possible for Sue to be in are : n \* n \* (G+1) = **100**

b)

Gamma = 0.9

Strategy = Pure Exploitation

Epsilon = 0.1

Alpha = 0.2

i) Under the policy learned by above hyperparameter tuning, the miner always chooses the next action with the highest value in the Q Table i.e. it does not take any risks or try to explore the grid randomly to find the best path.

Policy:

t : 0 action: Up

t : 1 action: Right

t : 2 action: Right

t : 3 action: Right

t : 4 action: Right

t : 5 action: Down

t : 6 action: Down

t : 7 action: Down

t : 8 action: Up

t : 9 action: Left

t : 10 action: Left

t : 11 action: Up

t : 12 action: Up

t : 13 action: Left

t : 14 action: Left

t : 15 action: Down

t : 16 action: Down

t : 17 action: Down

t : 18 action: Right

t : 19 action: Right

t : 20 action: Right

t : 21 action: Down

t : 22 action: Down

t : 23 action: Down

t : 24 action: Up

t : 25 action: Left

t : 26 action: Left

t : 27 action: Up

t : 28 action: Up

t : 29 action: Left

t : 30 action: Left

t : 31 action: Down

t : 32 action: Down

t : 33 action: Down

t : 34 action: Right

t : 35 action: Right

t : 36 action: Right

t : 37 action: Down

t : 38 action: Down

t : 39 action: Down

ii)

Cumulative Reward under this policy: 0.813477248474479

iii)

This is not the optimal policy that maximizes the cumulative reward for the given gamma value, because during training, the minder tends to stick to a specific path based on the initialized q table values. It does not try to explore the grid to find out better alternatives to the current path.

c)

Gamma = 0.9

Strategy = exploration/exploitation

Epsilon = 0.7

Alpha = 0.2

i)

Under the policy learned by above hyperparameter tuning, some randomness is introduced in the actions taken by the miner based on a probability of 0.7. This randomness allows the miner to explore the grid more without worrying too much about the total rewards being gained.

Policy:

t : 0 action: Right

t : 1 action: Right

t : 2 action: Right

t : 3 action: Right

t : 4 action: Right

t : 5 action: Right

t : 6 action: Up

t : 7 action: Up

t : 8 action: Up

t : 9 action: Up

t : 10 action: Left

t : 11 action: Left

t : 12 action: Left

t : 13 action: Down

t : 14 action: Down

t : 15 action: Down

t : 16 action: Down

t : 17 action: Right

t : 18 action: Right

t : 19 action: Right

t : 20 action: Right

t : 21 action: Right

t : 22 action: Up

t : 23 action: Up

t : 24 action: Up

t : 25 action: Up

t : 26 action: Left

t : 27 action: Left

t : 28 action: Left

t : 29 action: Down

t : 30 action: Down

t : 31 action: Down

t : 32 action: Down

t : 33 action: Right

t : 34 action: Right

t : 35 action: Right

t : 36 action: Right

t : 37 action: Right

t : 38 action: Up

t : 39 action: Up

ii)

Cumulative Reward under this policy : 1.0042928993512086

iii)

The discount factor essentially determines how much the agent or miner cares about rewards in the distant future relative to those in the immediate future. If gamma is 0, the miner will be completely myopic and only learn about actions that produce an immediate reward. If gamma is 1, the agent will evaluate each of its actions based on the sum total of all of its future rewards. The later we receive rewards, the less attractive they are to present calculations. So, in addition to the randomness introduced by a higher epsilon value, the miner is at present not concerned about the immediate rewards and discounts them for the sake of possible better future rewards. So, this policy might be the optimal policy that maximizes the Cumulative reward.

d)

Gamma = 0.6

Strategy = exploration/exploitation

Epsilon = 0.7

Alpha = 0.2

i) Under the policy learned by above hyperparameter tuning, in addition to the randomness introduced by a higher Epsilon value, a lower discount factor is introduced which leads to a higher significance being given to the immediate reward as compared to step 3. This restricts the learning process more and hence the miner’s actions are more balanced in terms of risk and reward.

Policy:

t : 0 action: Right

t : 1 action: Right

t : 2 action: Right

t : 3 action: Right

t : 4 action: Right

t : 5 action: Up

t : 6 action: Up

t : 7 action: Up

t : 8 action: Up

t : 9 action: Left

t : 10 action: Left

t : 11 action: Left

t : 12 action: Down

t : 13 action: Down

t : 14 action: Down

t : 15 action: Down

t : 16 action: Right

t : 17 action: Right

t : 18 action: Right

t : 19 action: Right

t : 20 action: Up

t : 21 action: Up

t : 22 action: Up

t : 23 action: Up

t : 24 action: Left

t : 25 action: Left

t : 26 action: Left

t : 27 action: Down

t : 28 action: Down

t : 29 action: Down

t : 30 action: Down

t : 31 action: Right

t : 32 action: Right

t : 33 action: Right

t : 34 action: Right

t : 35 action: Up

t : 36 action: Up

t : 37 action: Up

t : 38 action: Up

t : 39 action: Left

ii)

Cumulative Reward Under this policy: 0.007259352754563589

iii) In comparison, We observe that both the policies start out on a similar path in the beginning but later on diverge and follow different paths. This is because the policy obtained in step C is less constrained as compared to the one in step d. Also, in step c the learning algorithm cares more about the future rewards while choosing the next best action as compared to step c which cares less about future rewards.

The cumulative reward in scenario c is better than the value obtained in scenario d, even after other hyperparameters are changed and experimented with. Since ours is a fully observable environment with a finite horizon, it is better to choose a higher discount rate to get better rewards.

Code for step D:

import numpy as np

import random

from copy import deepcopy

class SueState:

def \_\_init\_\_(self, grid, position, gold):

self.grid = grid

self.position = position

self.gold = gold

def SueNextState(state, move):

entering\_home = False

entering\_gold\_mine = False

out\_of\_bounds = False

pos = deepcopy(state.position)

if move == Up:

if pos == [1,0]:

entering\_home = True

return pos, entering\_home, entering\_gold\_mine, out\_of\_bounds

elif pos[0] == 0:

out\_of\_bounds = True

return pos, entering\_home, entering\_gold\_mine, out\_of\_bounds

else:

pos[0] = max(0, pos[0] - 1)

elif move == Down:

if pos == [3,4]:

entering\_gold\_mine = True

return pos, entering\_home, entering\_gold\_mine, out\_of\_bounds

elif pos[0] == 4:

out\_of\_bounds = True

return pos, entering\_home, entering\_gold\_mine, out\_of\_bounds

else:

pos[0] = min(len(state.grid) - 1, pos[0] + 1)

elif move == Left:

if pos == [0,1]:

entering\_home = True

return pos, entering\_home, entering\_gold\_mine, out\_of\_bounds

elif pos[1] == 0:

out\_of\_bounds = True

return pos, entering\_home, entering\_gold\_mine, out\_of\_bounds

else:

pos[1] = max(0, pos[1] - 1)

elif move == Right:

if pos == [4,3]:

entering\_gold\_mine = True

return pos, entering\_home, entering\_gold\_mine, out\_of\_bounds

elif pos[1] == 4:

out\_of\_bounds = True

return pos, entering\_home, entering\_gold\_mine, out\_of\_bounds

else:

pos[1] = min(len(state.grid) - 1, pos[1] + 1)

else:

raise ValueError(f"Unknown move : {move}")

return pos, entering\_home, entering\_gold\_mine, out\_of\_bounds

def SueMove(state, move):

pos, entering\_home, entering\_gold\_mine, out\_of\_bounds = SueNextState(state,move)

gold\_collected = state.gold

grid\_item = state.grid[pos[0]][pos[1]]

new\_grid = deepcopy(state.grid)

if grid\_item == Blank:

old = state.position

new\_grid[old[0]][old[1]] = Blank

new\_grid[pos[0]][pos[1]] = Sue

reward = 0

else:

if entering\_gold\_mine == True:

if gold\_collected < G:

gold\_collected +=1

else:

pass

reward = 0

elif entering\_home == True:

if gold\_collected > 0:

reward = gold\_collected

gold\_collected = 0

else:

reward = 0

elif out\_of\_bounds == True:

reward = 0

else:

raise ValueError(f"Unknown Grid Item {grid\_item}")

return SueState(grid = new\_grid, position = pos, gold = gold\_collected), reward

def choose\_move(state):

if random.uniform(0,1) < epsilon:

return random.choice(Moves)

else:

return np.argmax(QTable[(state.position[0], state.position[1], state.gold)])

def detpolicy(state):

q\_max = -np.Inf

optimal\_move = -1

x = state.position[0]

y = state.position[1]

a = state.gold

for move in Moves:

if QTable[(x,y,a)][move] > q\_max:

optimal\_move = move

q\_max = QTable[(x,y,a)][move]

return optimal\_move

def getPolicy():

state = start

cumulative\_reward = 0

for t in range(T):

optimal\_move = detpolicy(state)

next\_state, reward = SueMove(state, optimal\_move)

cumulative\_reward += (gamma\*\*t)\*reward

print("t :", t, " move: ",Moves\_to\_number[optimal\_move])

state = next\_state

print("Cumulative Reward: ",cumulative\_reward)

gamma = 0.6 ## discount factor

epsilon = 0.7 ## probability ( 0 -> pure exploitation, 1-> pure exploration)

alpha = 0.2 ## Step Size

n = 5 ## grid size

G = 3 ## Gold limit for Sue

episodes = 10000 ## Number of episodes for simulation

T = 40 ## Time/Action limit

gold\_mine = "M"

Sue = "S"

Home = "H"

Blank = "#"

# Sue's Environment

grid = [

[Home, Blank, Blank, Blank, Blank],

[Blank, Blank, Blank, Blank, Blank],

[Blank, Blank, Blank, Blank, Blank],

[Blank, Blank, Blank, Blank, Blank],

[Sue, Blank, Blank, Blank, gold\_mine],

]

for row in grid:

print(row)

## Moves

Up = 0

Down = 1

Left = 2

Right = 3

Moves = [Up, Down, Left, Right]

Moves\_to\_number = {0 : 'Up', 1 : "Down", 2: "Left", 3: "Right"}

##defining the start state

start = SueState(grid=grid,position=[4,0],gold = 0)

##Q-Table initialization

QTable = {}

for x in range(0,n):

for y in range(0,n):

if x == 0 and y == 0:

pass

elif x == 4 and y == 4:

pass

else:

for a in range(G+1):

QTable[x,y,a] = np.random.normal(0,1,(len(Moves)))

if \_\_name\_\_ == "\_\_main\_\_":

rewards = []

for i in range(episodes):

state = start

tot\_reward = 0

path = ""

for \_ in range(T):

move = choose\_move(state)

if move == 0:

path += "U"

elif move == 1:

path += "D"

elif move == 2:

path += "L"

elif move == 3:

path += "R"

next\_state, reward = SueMove(state, move)

tot\_reward += reward

x = state.position[0]

x\_next = next\_state.position[0]

y = state.position[1]

y\_next = next\_state.position[1]

a = state.gold

a\_next = next\_state.gold

QTable[(x,y,a)][move] = (1 - alpha)\*QTable[(x,y,a)][move] + alpha\*(reward + gamma\*np.max(QTable[(x\_next,y\_next,a\_next)]))

state = next\_state

print(f"Episode {i + 1}: Total Reward -> {tot\_reward}")

print(f"Path Followed by Sue: {path}")

rewards.append(tot\_reward)

print("Max Total Reward : ",max(rewards))

print("Q Table:\n",QTable)

print("Policy:")

getPolicy()