# Robby_the_robot_Qlearning

March 8, 2022

The goal of this assignment is to have Robby the Robot use Q-learning to correctly pick up cans and avoid walls in his grid world.

#Import the necessary libraries

[118]:
```
import numpy as np
import random
import matplotlib.pyplot as plt
```

Initialising the qmatrix which is a 10 x 10 grid

[119]:
```
qMatrix = None
```

Rewards that robby would receive by doing certain actions

[120]:
```
CAN = 10
WALL = -5
EMPTY = -1
```

#States

[121]:
```
CANSTATE = 0
WALLSTATE = 1
EMPTYSTATE = 2
```

Initialising the values, N (episodes), M(actions)

#Part 2: Experiment with learning rate , here eta can be a value [0,1]. SO cosider eta = 0.8,0.6,0.4,0.2

[122]:
```
N = 5000
M = 200
eta = 0.2
gamma = 0.9
```

Defining the class Grid

[123]:
```
class Grid:
    def __init__(self, robby=(-1, -1)):

        self.grid = np.zeros((10, 10))
        for x in range(10):
```

```
            for y in range(10):
                self.grid[x, y] = random.randint(0, 1)
                # self.grid[x, y] = bool(random.getrandbits(1))
        self.robby = robby

    def newLocation(self, loc=(0, 0)):
        self.robby = loc
        # print("robby: ", self.robby)
```

Defining the class robot - this class consits of robby's actions and sensors

[124]:
```
class Robot:

    def __init__(self):
        # robby is randomly placed
        self.xCoor = random.randint(0, 9)
        self.yCoor = random.randint(0, 9)

        self.reward = 0
        self.senseCurrent = -1
        self.senseNorth = -1
        self.senseSouth = -1
        self.senseEast = -1
        self.senseWest = -1

        # set up grid with robby placement
        self.rGrid = Grid((self.xCoor, self.yCoor))

    # calculate the state in the qmatrix
    def myState(self):
        qMatrixState = (3**0)*self.senseCurrent + (3**1)*self.senseEast + \
            (3**2) * self.senseWest + (3**3) * \
            self.senseSouth + (3**4)*self.senseNorth
        return qMatrixState

# sensors
    def sensor(self):
        # current
        if self.rGrid.grid[self.xCoor, self.yCoor] == 1:
            self.senseCurrent = CANSTATE
        else:
            self.senseCurrent = EMPTYSTATE

        # north
        if self.xCoor == 0:
            self.senseNorth = WALLSTATE
        elif self.rGrid.grid[self.xCoor - 1, self.yCoor] == 1:
```

```
            self.senseNorth = CANSTATE
        else:
            self.senseNorth = EMPTYSTATE

        # south
        if self.xCoor == 9:
            self.senseSouth = WALLSTATE
        elif self.rGrid.grid[self.xCoor + 1, self.yCoor] == 1:
            self.senseSouth = CANSTATE
        else:
            self.senseSouth = EMPTYSTATE

        # east
        if self.yCoor == 9:
            self.senseEast = WALLSTATE
        elif self.rGrid.grid[self.xCoor, self.yCoor + 1] == 1:
            self.senseEast = CANSTATE
        else:
            self.senseEast = EMPTYSTATE

        # west
        if self.yCoor == 0:
            self.senseWest = WALLSTATE
        elif self.rGrid.grid[self.xCoor, self.yCoor - 1] == 1:
            self.senseWest = CANSTATE
        else:
            self.senseWest = EMPTYSTATE


# actions


    def moveNorth(self):
        # if we are at the most north point, & try to go north, hit wall
        if self.xCoor == 0:
            self.reward += WALL
            self.sensor()
            return WALL
        else:
            self. xCoor -= 1
            self.rGrid.newLocation((self.xCoor, self.yCoor))
            self.sensor()
            return 0

    def moveSouth(self):
        if self.xCoor == 9:
            self.reward += WALL
```

```
            self.sensor()
            return WALL
        else:
            self. xCoor += 1
            self.rGrid.newLocation((self.xCoor, self.yCoor))
            self.sensor()
            return 0

    def moveEast(self):
        if self.yCoor == 9:
            self.reward += WALL
            self.sensor()
            return WALL
        else:
            self. yCoor += 1
            self.rGrid.newLocation((self.xCoor, self.yCoor))
            self.sensor()
            return 0

    def moveWest(self):
        if self.yCoor == 0:
            self.reward += WALL
            self.sensor()
            return WALL
        else:
            self.yCoor -= 1
            self.rGrid.newLocation((self.xCoor, self.yCoor))
            self.sensor()
            return 0

    def pickUpCan(self):
        if self.rGrid.grid[self.xCoor, self.yCoor]:
            self.reward += CAN
            self.rGrid.grid[self.xCoor, self.yCoor] = 0
            self.sensor()
            return CAN
        else:
            self.reward += EMPTY
            return EMPTY

    def randomAction(self, random):
        if random == 0:
            return self.moveNorth()
        elif random == 1:
            return self.moveSouth()
        elif random == 2:
            return self.moveEast()
```

```
        elif random == 3:
            return self.moveWest()
        elif random == 4:
            return self.pickUpCan()
```

Defining the class main

Part3: Experiment with epsilon, varying the value of epsilon to see the changes in the result. Epsilon = 0.9,0.65,0.1

[125]:
```
def main():

    # Initial training for Robby
    epsilon = 0.1
    qMatrix = np.zeros((3**5, 5))  # (states, actions)

    count = 0
    rewards = []
    for epoch in range(N):
        # print(epoch)
        robby = Robot()

        for step in range(M):

            # observe robby's current state
            currentState = robby.myState()
            reward = 0
            # print("current state: ", currentState)

            # choose an action a using e-greedy action selection
            action = None
            if random.random() < epsilon:  # take random action b/w 0 and 1
                action = random.randint(0, 4)
                # receive reward
                reward = robby.randomAction(action)
            else:
                # if all actions are zero, choose one at random
                if np.count_nonzero(qMatrix[robby.myState(), :]) == 0:
                    action = random.randint(0, 4)

                else:  # else choose the best action
                    action = np.argmax(qMatrix[robby.myState(), :])
                # receive reward
                reward = robby.randomAction(action)

            # calculate formula

            old_q_value = qMatrix[currentState, action]
```

```
            # observe robbys new state
            max_q_value = max(qMatrix[robby.myState(), :])

            # update q formula
            qMatrix[currentState, action] += eta * \
                (reward + gamma*(int(max_q_value) - old_q_value))

        if epoch % 50 == 0 and epsilon >= 0.05:
            epsilon -= 0.05
        # print("Epsilon value: ", epsilon)

        if epoch % 100 == 0:
            rewards.append(robby.reward)

    # Now run again with epsilon set at 0.1, using the same trained qmatrix
    # This time, the qmatrix is not updated.
    epsilon = 0.1
    trainedReward = []
    for epoch in range(N):
        robby = Robot()
        for step in range(M):
            # print(step)
            currentState = robby.myState()
            reward = 0
            # print("current state: ", currentState)
            action = None

            if random.random() < epsilon:  # take random action
                action = random.randint(0, 4)
                reward = robby.randomAction(action)
            else:
                # if all actions are zero, choose one at random
                if np.count_nonzero(qMatrix[robby.myState(), :]) == 0:
                    action = random.randint(0, 4)

                else:  # else choose the best action
                    action = np.argmax(qMatrix[robby.myState(), :])

                reward = robby.randomAction(action)

        if epoch % 100 == 0:
            trainedReward.append(robby.reward)

    # print("TrainingReward values: ", rewards)
    print("The Test-average is: ", np.average(trainedReward))
    print("The Test-standard-deviation is: ", np.std(trainedReward))
```

```
    plt.plot(rewards)
    plt.ylabel('Sum of Rewards')
    plt.xlabel('Episode (100s)')
    plt.title('Training Reward')
    plt.show()
```

[126]:
```
if __name__ == '__main__':
    main()
```

```
The Test-average is:  169.96
The Test-standard-deviation is:  84.22896413942178
```