CSE 584 HW 2

Maxwell Dailey

October 2024

1 Abstract

I found a piece of code from an online GitHub (https://github.com/ronanmmurphy/Q-Learning-Algorithm/tree/main) that implements a Reinforcement Learning algorithm. The purpose of this code is to solve the deterministic FrozenLack "grid world" problem, resulting in the capability to optimally navigate a lake grid to reach a goal position from a starting position. The optimal navigation does not only require finding the shortest path from the start to the goal, but finding one that avoids the "holes" in the ice that have been scattered throughout the lake grid. The code endeavors to solve this problem using a Reinforcement Learning algorithm known as Q-Learning. More specifically, a number of simulation episodes are performed, where an episode ends if the current state is the goal position or a hole. Before the first episode, the Q-values of all of the state-action pairs are set to zero. The reward of a given state is as follows: -5 for a hole position, +1 for the goal position, and -1 otherwise. The Q-values of the state-action pairs seen during an episode are updated according to the following function: Q[(i,j,action)] = (1-α)Q[(i,j,action)] + α(Reward + γ*Qmax[nxtStateAction]), where α can be seen as a learning rate value and γ can be seen as an importance value (setting the importance of the maximum Q-value of the next state relative to the reward at the current state). The state-action pairs seen during an episode are determined semi-deterministically, as, in this case, there is a ninety percent chance of choosing the state-action pair associated with the highest Q-value, but also a ten percent chance of a random action being chosen. This is done to allow the algorithm to explore unseen paths to have a better understanding of the environment space. Given enough episodes, the algorithm finds the most optimal route from the start to the goal by following the state-action pairs with the highest Q-values.

2 Code

```
import numpy as np
import random
import matplotlib.pyplot as plt

#Rows in Board
BOARD_ROWS = 5
#Columns in Board
BOARD_COLS = 5

#Start position
START = (0, 0)
#Goal position
WIN_STATE = (4, 4)
#Position of holes
HOLE_STATE = [(1, 0), (3, 1), (4, 2), (1, 3)]


# class state defines the board and decides reward, end and next position
class State:
    def __init__(self, state=START):
        #Initalise the state to the given start
        self.state = state
        #Initialise the end to false as the start
        # and goal should be different positions
        self.isEnd = False

    def getReward(self):
        #Iterates through the hole positions
        for i in HOLE_STATE:
            #If the current state matches that of a hole,
            # return a reward of -5 as this state is not desired
            if self.state == i:
                return -5
        # If the current state matches that of the goal,
        # return a reward of 1 as this state is desired
        if self.state == WIN_STATE:
            return 1
        #Otherwise return a reward of -1, because we want to
        # incentivize finding the shortest, valid path
        else:
            return -1

    def isEndFunc(self):
        #Checks if the current state matches the position of the goal
        if (self.state == WIN_STATE):
```

```
            #Sets .isEnd to True to signal that the episode should end
            self.isEnd = True

        #Iterates through the hole positions
        for i in HOLE_STATE:
            #Check if the current state matches the position of a hole
            if self.state == i:
                #Sets .isEnd to True to signal that the episode should end
                self.isEnd = True

    def nxtPosition(self, action):
        #Set the next state to the position that results
        # from the given action (i.e. up, down, left, right)

        if action == 0:
            #Set the next position to be one above the current position (i.e. up)
            nxtState = (self.state[0] - 1, self.state[1])
        elif action == 1:
            # Set the next position to be one below the current position (i.e. down)
            nxtState = (self.state[0] + 1, self.state[1])
        elif action == 2:
            # Set the next position to be one left the current position (i.e. left)
            nxtState = (self.state[0], self.state[1] - 1)
        else:
            # Set the next position to be one right the current position (i.e. right)
            nxtState = (self.state[0], self.state[1] + 1)

        #Check if the designated row for the next state is within the valid range
        if (nxtState[0] >= 0) and (nxtState[0] <= 4):
            # Check if the designated column for the next state is within the valid range
            if (nxtState[1] >= 0) and (nxtState[1] <= 4):
                #If both of the above conditions are met return the determined next state
                return nxtState
        #If the next state is outsie of the gride, return the current state
        return self.state


class Agent:

    def __init__(self):
        #Initialize possible actions (i.e. up, down, left, right)
        self.actions = [0, 1, 2, 3]
        #Initialize the current state as the previously defined start position
        self.State = State()
        #Set the learning rate value
        self.alpha = 0.5
        #Set the importance factor to be used to weight the found
        # maximum Q-value of potential next state action pairs
        self.gamma = 0.9
        #Set the probability of choosing a random action
        self.epsilon = 0.1
        #Set with the value specifying if the current state matches
        # the goal state or any of the states containing holes
        self.isEnd = self.State.isEnd

        # array to retain reward values for plot
        self.plot_reward = []

        #Initialize Q-values as a dictionary for the current Q-values
        self.Q = {}
        #Initialize new Q-values as a dictionary for the updated Q-values
        # after an action has been taken
        self.new_Q = {}
        #Initialize the reward to 0
        self.rewards = 0

        #Iterate through all board rows
        for i in range(BOARD_ROWS):
            #Iterate through all board columns
            for j in range(BOARD_COLS):
                #Iterate through each potential action
                for k in range(len(self.actions)):
                    #Set the current Q-value for each combination of position and action to 0
                    self.Q[(i, j, k)] = 0
                    #Set the next Q-value for each combination of position and action to 0
                    self.new_Q[(i, j, k)] = 0

        print(self.Q)
```

```
# method to choose action with Epsilon greedy policy, and move to next state
def Action(self):
    #Choose a random value between [0,1)
    rnd = random.random()
    #Set the initial maximum next reward value to an arbitrary low value
    mx_nxt_reward = -10
    #We have not yet chosen an action to take, so action should be initialized to None
    action = None

    #Check if the random value is greater than the threshold
    # for choosing the current highest valued action
    if (rnd > self.epsilon):
        #Iterate through every action
        for k in self.actions:
            #Determine the row and column position of the current state
            i, j = self.State.state
            #Find the Q-value of the state action pair given the current state
            # and a potential action
            nxt_reward = self.Q[(i, j, k)]

            #Check if the found Q-value is greater than current maximum next reward
            if nxt_reward >= mx_nxt_reward:
                #Set action to be the current action
                action = k
                #Set the maximum next reward to be the found Q-value
                mx_nxt_reward = nxt_reward

    #Otherwise choose a random action
    else:
        #Set action to be a random action from the previously defined possible actions
        action = np.random.choice(self.actions)

    #Set position to the state that results from the current position and chosen action
    position = self.State.nxtPosition(action)
    return position, action

# Q-learning Algorithm
def Q_Learning(self, episodes):
    #Set the episode counter to be 0
    x = 0
    #Contiue to loop while the episode counter is less than the desired
    # number of episodes
    while (x < episodes):
        #Check if the episode has reached its end
        if self.isEnd:
            #Get the reward of the current state
            reward = self.State.getReward()
            #Increment the total reward seen during the course
            # of the episode by the reward of the current state
            self.rewards += reward
            #Append the total reward to the array to retain
            # reward values for plotting
            self.plot_reward.append(self.rewards)

            #Determine the row and column position of the current state
            i, j = self.State.state
            #Iterate through every action
            for a in self.actions:
                #Set the next Q-value for each combination of the
                # current state and an action to be the reward
                # at the current position rounded to three digits
                self.new_Q[(i, j, a)] = round(reward, 3)

            #Reset the state to the start position
            self.State = State()
            #Reset .isEnd to the value it had at the beginning of the episode
            self.isEnd = self.State.isEnd

            #Set the total reward to zero
            self.rewards = 0
            #Increment the episode counter by 1
            x += 1
        else:
            # Set the initial maximum next reward value to an arbitrary low value
            mx_nxt_value = -10
            #Get the next state that will come after the current state,
            # and get the action that was taken to achieve
```

```
        # the next state
        next_state, action = self.Action()
        # Determine the row and column position of the current state
        i, j = self.State.state
        # Get the reward of the current state
        reward = self.State.getReward()
        #Increment the total reward seen during the course of the episode
        # by the reward of the current state
        self.rewards += reward

        #Iterate through every action
        for a in self.actions:
            #Store a state action pair where the state will be the
            # next state we determined above and the actions will be
            # whatever action the code is on during the iteration process
            nxtStateAction = (next_state[0], next_state[1], a)
            #Calculate the updated Q-value for the current position
            # given the learning rate value, the Q-value associated with
            # the state action pair of the current position and the action
            # to be taken to reach the next state, the reward of the current
            # state, the importance factor as defined above, and the
            # Q-value associated with the state action pair of the next
            # state and the current action the code is on during the iteration process
            q_value = (1 - self.alpha) * self.Q[(i, j, action)] + self.alpha * (
                        reward + self.gamma * self.Q[nxtStateAction])

            # Check if the calculated Q-value for the current position
            # is greater than maximum seen Q-value for the current position
            if q_value >= mx_nxt_value:
                #Update the maximum seen Q-value for the current
                # position to the calculated Q-value
                mx_nxt_value = q_value

        #Set the current state to the next state
        self.State = State(state=next_state)
        #Update the boolean determining if the episode should
        # continue or end given the updated current state
        self.State.isEndFunc()
        #Set .isEnd to match the value stored by the state
        # object given the updated current state
        self.isEnd = self.State.isEnd

        # Set the next Q-value for the state action pair of
        # the current position and the subsequent action taken
        # to be the maximum Q-value calculated for the current
        # position rounded to three digits
        self.new_Q[(i, j, action)] = round(mx_nxt_value, 3)

    #Copy next Q-values to current Q-values
    self.Q = self.new_Q.copy()
#Print final Q table output
print(self.Q)
```