DAT405 Introduction to Data Science and AI, 2022, Reading Period 3
Assignment 5: Reinforcement learning and Classification 2
Grader: Newton
Due Date: 23rd February, 23:59
Submitted by: Lenia Malki, Maële Belmont - Group 9

General guidelines:

- All solutions to theoretical and pratical problems must be submitted in this ipynb notebook, and equations wherever required, should be formatted using LaTeX math-mode.
- All discussion regarding practical problems, along with solutions and plots should be specified in this notebook. All plots/results should be visible such that the notebook do not have to be run. But the code in the notebook should reproduce the plots/results if we choose to do so.
- Your name, personal number and email address should be specified above.
- All tables and other additional information should be included in this notebook.
- Before submitting, make sure that your code can run on another computer. That all plots can show on another computer including all your writing. It is good to check if your code can run here: https://colab.research.google.com.

Self-check

1. Have you answered all questions to the best of your ability?
2. Anything else you can easily check? (details, terminology, arguments, commenting for code etc.?)

Grading will be based on a qualitative assessment of each assignment. It is important to:

- Present clear arguments
- Present the results in a pedagogical way
- Show understanding of the topics (e.g, write a pseudocode)
- Give correct solutions
- Make sure that the code is well commented

Again, as mentioned in general guidelines, all code should be written here. And this same ipython notebook file (RLAssignment.ipynb) should be submitted with answers and code written in it. NO SEPERATE FILE SHALL BE ACCEPTED.

Primer

Decision Making

The problem of **decision making under uncertainty** (commonly known as **reinforcement learning**) can be broken down into two parts. First, how do we learn about the world? This involves both the problem of modeling our initial uncertainty about the world, and that of drawing conclusions from evidence and our initial belief. Secondly, given what we currently know about the world, how should we decide what to do, taking into account future events and observations that may change our conclusions? Typically, this will involve creating long-term plans covering possible future eventualities. That is, when planning under uncertainty, we also need to take into account what possible future knowledge could be generated when implementing our plans. Intuitively, executing plans which involve trying out new things should give more information, but it is hard to tell whether this information will be beneficial. The choice between doing something which is already known to produce good results and experiment with something new is known as the **exploration-exploitation dilemma**.

The exploration-exploitation trade-off

Consider the problem of selecting a restaurant to go to during a vacation.Lets say the best restaurant you have found so far was **Les Epinards**. The food there is usually to your taste and satisfactory. However, a well-known recommendations website suggests that **King's Arm** is really good! It is tempting to try it out. But there is a risk involved. It may turn out to be much worse than **Les Epinards**, in which case you will regret going there. On the other hand, it could also be much better. What should you do? It all depends on how much information you have about either restaurant, and how many more days you'll stay in town. If this is your last day, then it's probably a better idea to go to **Les Epinards**, unless you are expecting **King's Arm** to be significantly better. However, if you are going to stay there longer, trying out **King's Arm** is a good bet. If you are lucky, you will be getting much better food for the remaining time, while otherwise you will have missed only one good meal out of many, making the potential risk quite small.

Overview

- To make things concrete, we will first focus on decision making under **no** uncertainity, i.e, given we have a world model, we can calculate the exact and optimal actions to take in it. We shall first introduce **Markov Decision Process (MDP)** as the world model. Then we give one algorithm (out of many) to solve it.
- Next, we will work through one type of reinforcement learning algorithm called Q-learning. Q-learning is an algorithm for making decisions under uncertainity, where uncertainity is over the possible world model (here MDP). It will find the optimal policy for the **unknown** MDP, assuming we do infinite exploration.

Markov Decision Process

Markov Decision Process (MDP) provides a mathematical framework for modeling decision-making. It is a discrete time (distinct points in time) stochastic (randomly determined) process.

MDPs are made up of 4 parts:
S: Finite set of states (Ex: s_1, s_2 ... s_N)
A: Finite set of actions (Ex: North, South, East, West)
$P_a(s,s')$: Probability that action *a* in state *s* at time *t* will lead to state *s'* at time *t + 1*
$R_a(s,s')$: Immediate reward received after moving from state *s* to state *s'* by action *a*

An agent acts in an MDP at time *t*, by taking certain action *a* in state *s*, going to state *s'*, and getting a reward *r* from the world. It then repeats the process for certain no. of times, either finite or infinite.

We also include a 5^{th} part in the description of an MDP called Gamma γ.
γ: The discount factor between 0 (inclusive) and 1 (exclusive). This determines how much credit you want to give to the future. If you think that the future reward is as important as the current reward you would set this to 0.99999. If you don't care about the future rewards you would set this to 0 and you only care about the current reward. For example, if your discount factor is 0.8 and after 5 steps you get a reward of 4 the present value of that reward is $0.8^4 * 5$ or ~2.

An MDP is a collection of states such that each state has a selection of actions associated with them. With each state-action pair comes a reward *r* (can be 0). Define a policy function:
$\pi : s \rightarrow a$, which tells which action to take at each state.

We now use the famous dynamic programming equation, also known as Bellman Equation, to define optimality in an MDP. The following equation defines what we call the **value function** of state *s* following some fixed policy π:

$$V^{\pi}(s) = \sum_{s'} P_{\pi(s)}(s, s')[R_{\pi(s)}(s, s') + \gamma V^{\pi}(s')]$$

We call V^{π} as the value of policy π.

Now, to find the **optimal** policy you will need to find the action that gives the highest reward.

$$V^*(s) = max_a \sum_{s'} P_a(s, s')[R_a(s, s') + \gamma V^*(s')]$$

A real world example would be an inventory control system. Your states would be the amount of items you have in stock. Your actions would be the amount to order. The discrete time would be the days of the month. The reward would be the profit.

A major drawback of MDPs is called the "Curse of Dimensionality". This states that the more states/actions you have the more computational difficult it is to solve.

Question 1 (2 points)

For the first question of the notebook, we give a quick example of an MDP. We would to see if you can put the definitions above into practice.

Question a: Given the following deterministic MDP (you select North, you move North), what is the optimal policy (path with the most points)?

Notes:

- The number in the box is the reward.
- Once you hit the end you are done. (Absorbing state)
- S is the starting point.
- F is the ending point.
- Use N for North, E for East, S for South, and W for West. Not all actions are available at each state, for example, you can't choose N and W at starting state, as there exists no valid next states in those directions.
- Pass the directions as a single string. Ex: ESWN will make a circle.

S	1	1
1	0	1
-1	-1	0
0	0	F

Answer:
SENESSS is the path with the most points, it scores 4 points.

Question b,c will attempt to firm up your knowledge of the parts of an MDP. Just remember that for a state denoted by (x,y), state N/E/S/W to that are (x,y-1),(x+1,y),(x,y+1),(x-1,y) respectively. We take (0,0) as the starting state S.

Question b: What is the probability of going from state (1,0) to state (2,0) using action E ? (i.e, $P_E((1,0),(2,0))$)
Answer:
$P_E((1,0),(2,0)) = 1$

Question c: What is the reward for moving from state (1,0) to state (2,0) ? (i.e, $R_E((1,0),(2,0))$)
Answer:
$R_E((1,0),(2,0)) = 1$

Value Iteration

The value iteration is one algorithm that can be used to find the optimal policy (π^*). Note that for any policy π^* to be optimal, it must satisfy the Bellman equation for optimal value function V^*. For any candidate V^*, it must be such that plugging it in the RHS (right-hand-side) of Bellman equation should give the same V^* again (by the recursive nature of this equation). This property will form the basis of our algorithm. Essentially, due to certain mathematical results, repeated application of RHS to any intial value function $V^0(s)$ will eventually lead to the value V which statifies the Bellman equation. Hence repeated application of Bellman equation for optimal value function will also lead to optimal value function, we can then extract the optimal actions by simply noting the actions that satisfy the equation.

The value function is based on the Bellman Equation for optimal value, which we recall here:

$$V^*(s) = max_a \sum_{s'} P_a(s, s')[R_a(s, s') + \gamma V^*(s')]$$

Example: Below is a 3x3 grid. We are going to walk through a few iterations to firm up your understanding. Lets assume this time that success of taking any action is 0.8. Meaning if we take E from a valid state (x,y), we will go (x+1,y) 0.8 percent of time, but remain in same state the remaining time. We will have a discount factor (γ) of 0.9. Assume $V^0(s') = 0$ for all s'.

0	0	0
0	10	0
0	0	0

Iteration 1: It is trivial, V(s) becomes the $max_a \sum_{s'} P_a(s, s')R_a(s, s')$ since V^0 was zero for s'.

0	8	0
8	2	8
0	8	0

Iteration 2:

Staring with cell (0,0): We find the expected value of each move:
Action N: 0
Action E: 0.8(0 + 0.9 * 8) + 0.2(0 + 0.9 * 0) = 5.76
Action S: 0.8(0 + 0.9 * 8) + 0.2(0 + 0.9 * 0) = 5.76
Action W: 0

Hence any action between E and S would have been best at this stage.

Similarly for cell (1,0):

Action S: 0.8(10 + 0.9 * 2) + 0.2(0 + 0.9 * 8) = 10.88 (Action S is the maximizing action)

Similar calculations for remaining cells give us:

5.76	10.88	5.76
10.88	8.12	10.88

5.76	10.88	5.76

Question 2 (4 points)

Please code the value iteration algorithm just described here, and show the optimal value function of the above 3x3 grid problem at convergence.

In [72]:
```
# Firstly, initialize the reawrd and value function
import numpy as np
r = np.array([[0,0,0],[0,10,0],[0,0,0]])
v = np.array ([[0,0,0],[0,0,0],[0,0,0]])

# Getting dimensions of the matrices in order to make code easier to read
dim = r.shape # The dinemsions of matrix r
lastRow = dim[0]-1
firstRow = (dim[0]-3)
lastCol = dim[1]-1
firstCol = (dim[1]-3)

# Set up variables:
success = 0.8 # Given in problem statement
disc = 0.9 # Aka gamma
epsilon = 1
```

In [ ]:
```
# Function for restarting the arrays to their origirnal values, if wanted
def restartArrays():
    global r
    global v
    r = np.array([[0,0,0],[0,10,0],[0,0,0]])
    v = np.array ([[0,0,0],[0,0,0],[0,0,0]])
```

In [ ]:
```
# Method for aquiring actions at each state, s = (x,y).
# Returns a list of possible actions by removing those actions which are not possisble.
# In other words, we are assuming that it is possible to go N/S/W/E for each s = (x,y) until proven wrong
def getActions(x,y):

    # Initialization
    N = (x-1,y,'N')
    S = (x+1,y,'S')
    E = (x,y+1,'E')
    W = (x,y-1,'W')

    # List of possible actions. Assume from the begginning that all directions are possible.
    actions = [N,S,E,W]

    if x == lastRow:
        actions.remove(S) # At last row, can't go dowm --> Remove S-action from actions
    if x == firstRow:
        actions.remove(N) # At first row, can't go up --> Remove N-action from actions
    if y == lastCol:
        actions.remove(E) # At last col, can't go right --> Remove E-action from actions
    if y == firstCol:
        actions.remove(W) # At first col, can't go left --> Remove W-action from actions
    return actions
```

In [ ]:
```
import copy
def valueIter(rew,val): # Takes in a reward and value matrix

    copied_val = copy.copy(val) # Copies over the given matrix to a new variable

    # Index for np.arrays do not correspond to actual number of rows and columns when referencing, hence the additional +1
    for x in range(lastRow+1):
        for y in range(lastCol+1):

            values = []  # Array which hold expected values
            max_value = 0;
            actions = getActions(x,y) # Get action state of current (x,y)

            # For each action...
            for a in actions:

                current_reward = rew[a[0],a[1]]
                new_value = copied_val[a[0],a[1]]

                old_reward = rew[x,y]
                old_value = copied_val[x,y]

                # calculate the expected value and...
                exp = success*(current_reward + disc*new_value)+ (1-success)*(old_reward + disc*old_value)
                values.append(exp)

            # get the maximun of the expected values and...
            max_value = round(max(values),2)

            # update the value matrix with the maximun value.
            val[x,y] = (max_value)

    # Once having looped through the entire matrix, check if the absolute difference between
    # the updated value matrix and the copy is greater or equal to epsilon. If not, script is finished and the resulting
    # value function is returned.
    for x in range(lastRow+1):
        for y in range(lastCol+1):
            if (abs(copied_val[x,y] - val[x,y]) >= epsilon):
                valueIter(rew, val)
                return val
    print('Value function at convergence: ')
    return val
```

In [21]:
```
# Restart the R and V array to make sure that the value iteration algorithm does not run on modified starting arrays
restartArrays()
```

In [22]:
```
valueIter(r,v)
```

```
Value function at convergence:
```

Out[22]:
```
array([[37, 43, 37],
       [43, 39, 43],
       [37, 43, 37]])
```

Reinforcement Learning (RL)

Until now, we understood that knowing the MDP, specifically $P_a(s, s')$ and $R_a(s, s')$ allows us to efficiently find the optimal policy using value iteration algorithm, but RL or decision making under uncertainity arises from the question of making optimal decisions without knowing the true world model (MDP in this case).

So far we have defined the value of a state V^π, let us define the value of an action, Q^π:

$$Q^\pi(s, a) = \sum_{s'} P_a(s, s')[R_a(s, s') + \gamma V^\pi(s')]$$

i.e, the value of taking action *a* from state *s* and then following π onwards. Similarly, the optimal Q-value equation is:

$$Q^*(s, a) = \sum_{s'} P_a(s, s')[R_a(s, s') + \gamma V^*(s')]$$

Q-learning

Q-learning algorithm can be used by an agent unaware of its surroundings (unknown MDP). All it can do is take an action *a* at time *t* from state *s* and observe the reward *r* and next state *s'*, and repeat this process again. So how it can learn to act optimally under such uninformative conditions ? Answer is using Q-learning. Without going into its justification, we simply state the main-update rule of this algorithm below:

$$\underbrace{\text{New } Q(s,a)}_{\substack{\text{New}\\ \text{Q-Value}}} = \underbrace{Q(s,a)}_{\substack{\text{Current}\\ \text{Q-Value}}} + \underbrace{\alpha}_{\substack{\text{Learning}\\ \text{rate}}} \Big[\underbrace{R(s,a)}_{\text{Reward}} + \underbrace{\gamma}_{\substack{\text{Discount}\\ \text{rate}}} \overbrace{\max Q'(s',a')}^{\substack{\text{Maximum predicted reward, given}\\ \text{new state and all possible actions}}} - Q(s,a) \Big]$$

Where we simply maintain Q(s,a) value for each state-action pair in a table. It is proven to converge to the optimal policy of the underlying unknown MDP for certain values of learning rate α. For our case, we set a constant $\alpha = 0.1$.

OpenAI Gym

We shall use already available simulators for different environments (world) using the popular OpenAI Gym library. It just implements diffrent types of simulators including ATARI games. Although here we will only focus on simple ones, such as Chain enviroment.

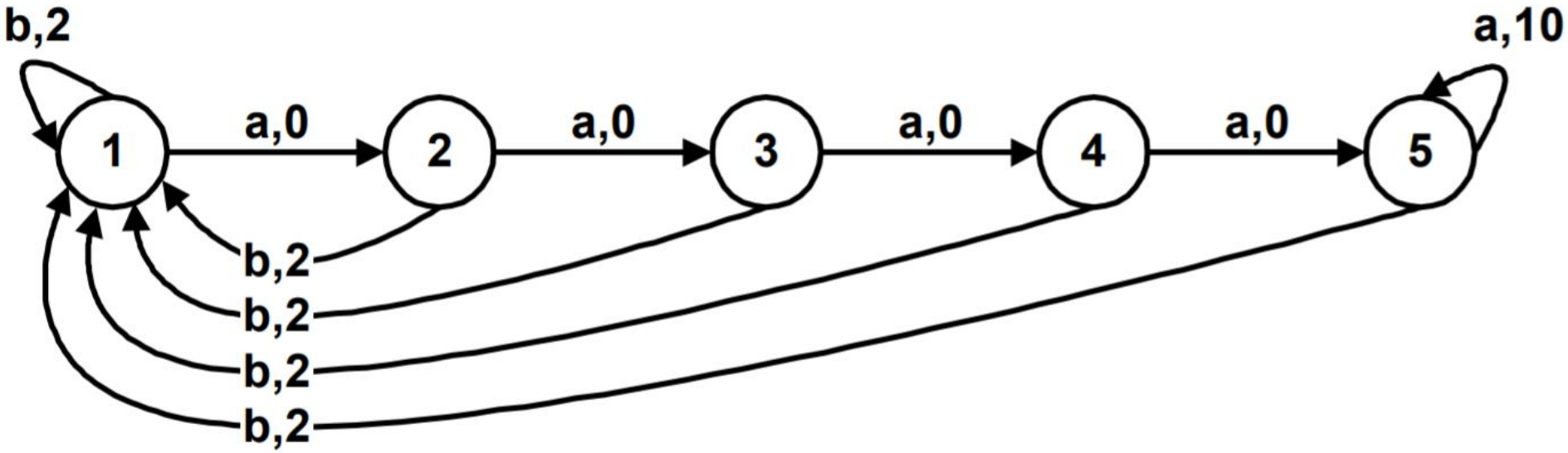

Figure 1. The "Chain" problem

Question 3 (1 point)

Basically, there are 5 states, and two actions 'a' and 'b'. Each transition (s,a,s') is noted with its corresponding reward. You are to first familiarize with the framework using its documentation, and then implement the Q-learning algorithm for the Chain enviroment (called 'NChain-v0') using default parameters. Finally print the Q^* table at convergence. Take $\gamma = 0.95$. You can refer to the Q-learning Jupyter notebook shown in class, uploaded on Canvas.

In [234…

```
import gym
import gym_toytext
import numpy as np
import random
import math

env = gym.make('NChain-v0')

num_episodes = 10000
gamma = 0.95
learning_rate = 0.1
epsilon = 0.1

# initialize the Q table
actionNB = env.action_space.n # number of actions
```

```
stateNB = env.observation_space.n # number of states
Q = np.zeros((stateNB, actionNB))

for j in range(num_episodes):
    state = env.reset()
    done = False
    while done == False:
        # First we select an action:
        if random.uniform(0, 1) < epsilon: # Flip a skewed coin
            action = env.action_space.sample() # Explore action space
        else:
            action = np.argmax(Q[state,:]) # Exploit learned values
        # Then we perform the action and receive the feedback from the environment
        new_state, reward, done, info = env.step(action)
        # Finally we learn from the experience by updating the Q-value of the selected action
        update = reward + (gamma*np.max(Q[new_state,:])) - Q[state, action]
        Q[state,action] += learning_rate*update
        state = new_state
print('Q* table at convergence:\n', Q)
```

```
Q* table at convergence:
 [[58.96687842 57.13858328]
 [62.86213096 56.591799  ]
 [67.24778546 57.52776721]
 [71.86320164 56.33271879]
 [79.64199114 62.17136962]]
```

Question 4 (2 points)

a. Verify that the optimal Q_* value obtained using Q-learning is same as the optimal value function V^* for the corresponding MDP's optimal action. You would have to first define the MDP corresponding to Chain enviroment.

In [14]:
```
# Function for retrieving the possible rewards for a specific state. This will always consist of two values:
# {2,0} for states <= 3 or {2,10} for states >= 4
def getRewardSpace(state):

    # List of possible actions. Assume from the begginning that all directions are possible.
    rewards = [2,0,10]

    if state == 4:
        rewards.remove(0)
    else:
        rewards = [2,0]

    return rewards
```

In [20]:
```
import copy
k = 1
def valueIteration(R,V):
    # Start with initialize the state space
    state_space = [1,2,3,4,5] # Works as coordinates
    action_space = [0,1] # Where 0 = backwards and 1 = forward

    p = 0.8 # Transition probability / Success rate
    q = (1-p)
```

```
    ga = 0.9 # Gamma
    ep = 1 # Epsilon

    v = copy.copy(V)

    # Loop through each state
    for state in range(0,len(state_space)): # 0,1,2,3,4 where 4 = last state

        values = []  # Array which hold expected values
        max_value = 0
        reward_space = getRewardSpace(state) # Gets the possile rewards for the current state

        # Loop through each action. Always 0 and 1
        for a in action_space: # First 0 then 1

            if a == 0: # Left movement, back to start
                r_new = reward_space[0] # Will either be 2 or 0 depending on state
                v_new = v[0]

            else: # Right movement
                r_new = reward_space[1] # Will either be 0 or 10 depending on state

                if state == 4: # Last state
                    v_new = v[4] # Makes sure that we don't get out of bounds error

                else:
                    v_new = v[state+1]
            r_curr = R[state] # Always the same
            v_curr = v[state] # Always the same

            # Calculates the expected value for the current state and specified action
            exp = p*(r_new + ga*v_new)+ q*(r_curr + ga*v_curr)

            # Append the calcualted exected value to the list of expected values for that state
            values.append(exp)

        # Get the maximum of expected value from the list of expceted values
        max_value = max(values)

        # Update the value function with the maximun expected value of the specified state
        V[state] = round(max_value,2)

    # Repeat the value iteration as long as the abs(v[state] - V[state]) is greater or equal to the stated epsilon
    for s in range(0,len(state_space)):
        global k
        if (abs(v[state] - V[state]) >= ep):
            print('Iteration ' + str(k) +' : ' + str(V))
            k = k+1
            valueIteration(R, V)
            return V

    print("\nOptimal value function V*: ", V)
```

In [250…

```
# Initialize the reward and value functions
R = [2, 0, 0, 0, 10] # Reward function
V = [0, 0, 0, 0, 0] # Initial value function
```

```
# Optimal value function V*
opt_valfunc = valueIteration(R,V)

# Verify equality by comparing V*(s) and max_a Q*(s,a)
# Create a list containing max_a Q*(s,a) elements
QmaxList = [] # list containing maximum element of each row of matrix Q*
for i in range(stateNB): # stateNB = number of rows in Q* matrix

    Qmax = 0 # Store maximum element of each row in this variable
    for j in range(actionNB): # actionNB = number of columns in Q* matrix
        if Q[i][j] > Qmax : # compare the elements on each row
            Qmax = Q[i][j] # store the maximum element...

    # ...append maximum element of each row in QmaxList
    QmaxList.append(Qmax)
print('\nmax_a Q*:', QmaxList)
```

```
Iteration 1 : [2.0, 1.6, 1.6, 1.6, 10.0]
Iteration 2 : [3.8, 3.33, 3.33, 7.49, 19.0]
Iteration 3 : [5.42, 4.94, 5.99, 15.03, 27.1]
Iteration 4 : [6.88, 6.39, 11.9, 22.22, 34.39]
Iteration 5 : [8.19, 9.72, 18.14, 28.76, 40.95]
Iteration 6 : [9.37, 14.81, 23.97, 34.66, 46.86]
Iteration 7 : [12.75, 19.92, 29.27, 39.98, 52.17]
Iteration 8 : [17.04, 24.66, 34.05, 44.76, 56.95]
Iteration 9 : [21.22, 28.95, 38.36, 49.06, 61.26]
Iteration 10 : [25.06, 32.83, 42.23, 52.94, 65.13]
Iteration 11 : [28.55, 36.31, 45.72, 56.42, 68.62]
Iteration 12 : [31.68, 39.45, 48.85, 59.56, 71.76]
Iteration 13 : [34.51, 42.27, 51.68, 62.39, 74.58]
Iteration 14 : [37.05, 44.82, 54.22, 64.93, 77.12]
Iteration 15 : [39.34, 47.11, 56.51, 67.21, 79.41]
Iteration 16 : [41.4, 49.17, 58.56, 69.27, 81.47]
Iteration 17 : [43.25, 51.01, 60.42, 71.13, 83.32]
Iteration 18 : [44.91, 52.68, 62.09, 72.79, 84.99]
Iteration 19 : [46.41, 54.19, 63.59, 74.3, 86.49]
Iteration 20 : [47.77, 55.54, 64.94, 75.65, 87.84]
Iteration 21 : [48.99, 56.75, 66.16, 76.86, 89.06]
Iteration 22 : [50.08, 57.85, 67.25, 77.96, 90.15]

Optimal value function V*:  [51.07, 58.83, 68.24, 78.94, 91.13]

max_a Q*: [58.966878415895714, 62.86213096486097, 67.24778545911985, 71.86320164411286, 79.64199114331863]
```

When setting $\gamma = 0.95$ in the optimal value function (*valueIteration*), the results are above 100, about 2 to 3 times higher than the $max_a Q_*$ values. When setting $\gamma = 0.9$, the results are in the same range as $max_a Q_*$. We are aware that changing the discount rate between Q-learning algorithm and value iteration algorithm should not be necessary in order to find equality. This variance could be due to a mistake in the value iteration code. We also noticed that the values slightly changed when taking a number of timesteps of 5,000 and 10,000, so Q *might not be containing the values of Q at convergence. However, the Q* values stayed in the same range (50-80) taking a number of timesteps of 5,000 and 10,000, therefore the issue probably comes from the *valueIteration* function above.

b. What is the importance of exploration in RL ? Explain with an example.

Exploration is necessary to try out a variety of possible actions in different states to get greater rewards instead of repeating the same actions over and over. The goal of RL is to learn the optimal policy, i.e. find the actions leading to the maximization of the reward. By exclusively exploiting, one might miss more efficient or rewardful actions that could have happened by choosing a random move.

For example, a new student explores a certain path to go Chalmers on his first day of study. On the second day, she/he can either choose to follow the same path as the previous day (exploit) or try out another one (explore). By exploring, she/he can be rewarded by time savings if the path is shorter. If she/he had chosen the same path as on the first day, she/he would not have found the optimal path.

Question 5 (1 point)

Briefly discuss the k-armed bandit problem formulation and it's distinguishing feature as a special case of the reinforcement learning problem formulation.

The k-armed bandit problem, also known as the multi-armed bandit (CMAB) problem, deals with scenarios where actions have to take place in an environment without knowledge about the specifics behind the risks and predicted rewards. In other words, it can be applied to aspects where there is a dilemma between exploration vs exploitation - is it worth taking the risk or will the action of not exploring it be of greater loss? We thus have three main components of the CMAP problem - the current and only state, possible actions at that state and the corresponding reward of that action. What stands out with this MDP is the fact that it only involves one state. It can in other words be defined as a stateless MDP. As a result of not having any states, the rewards are only dependent on action taken, meaning that we are not taking into consideration the previous action taken and hence, the reward of one action does not affect the reward of a future action as the reward distributions are fixed for each bandit. That is why the k-armed bandit problem is considered a special case of reinforcement learning (RL) problem, as opposed to the general problem formulations of RL problems where finite MDPs are dealt with.

Note

- Until now, we have described algorithms for when no. of states and actions are finite. In coming weeks, you will be taught how to extend these methods to continous state enviroments like ATARI games.

References

Primer/text based on the following references:

- http://www.cse.chalmers.se/~chrdimi/downloads/book.pdf
- https://github.com/olethrosdc/ml-society-science/blob/master/notes.pdf