DAT405 Assignment 5 – Group 53

Venkata Sai Dinesh Uddagiri - (20 hrs)
Madumitha Venkatesan - (20 hrs)

December 6, 2022

Problem 1

1a) What is the optimal path of the MDP above? Is it unique? Submit the path as a single string of directions. E.g. NESW will make a circle.

The optimal path for agent to reach from S(0,0) to F(2,3) with shortest number of steps and maximum reward score is EENNN. The total reward score of agent taken path EENNN is zero. The path that we determine is unique because all the other paths that we take to reach destination gives the reward score less than zero expect one path (EENNWNE), but optimal path should be the path with shortest disatnce and maximum reward score.

1b) What is the optimal policy (i.e. the optimal action in each state)? It is helpful if you draw the arrows/letters in the grid.

The optimal policy is the policy where agent always takes the action to maximizes the reward. The best optimal policy for problem is, do not visit the state that already visited, if the reward in all the directions is same then go east and if the all the directions have different reward value then take the direction with maximum value.

1c) What is expected total reward for the policy in 1a)?

The expected total reward for policy described above for path EENNN is zero. The total reward value is obtained by following calculation

$$V^{\pi}(s) = \sum_{t=1}^{T} \gamma^{t-1} r(s_t, a_t, s_{t+1}) p(s_{t+1}|s_t, a_t)$$

The transition probabilities of each action is deterministic so, we are considering the probability of each action as one. By this above equation changes as follows

$$V^{\pi}(s) = \sum_{t=1}^{T} \gamma^{t-1} r(s_t, a_t, s_{t+1})$$

The discount in the model, $\gamma = 1$

$$V^{\pi}(s) = \sum_{t=1}^{T} r(s_t, a_t, s_{t+1})$$

$$V^{\pi}(s) = (-1) + 1 + (-1) + 1 + 0 = 0$$

Problem 2

2a) Code the value iteration algorithm just described here, and show the converging optimal value function and the optimal policy for the above 3x3 grid.

```
import copy
def value_iteration(rm,vm,pm):
    # Creating the copy of old values
    vm_old=copy.deepcopy(vm)
    for r in range(3):
        for c in range(3):
            max_value = 0
            # Intializing moves applicable
            actions=["N","S","E","W"]
            # restricting moving south if row index equals zero
            if r == 0:
                actions.remove("S")
            # restricting moving west if coloumn index equals zero
            if c == 0:
                actions.remove("W")
            # restricting moving north if row index equals two
            if r == 2:
                actions.remove("N")
            # restricting moving East if coloumn index equals two
            if c == 2:
                actions.remove("E")

            for action in actions:
                # Defining move to be taken for each particular action applicable
                if action=="N":
                    state = [r+1,c]
                elif action=="S":
                    state = [r-1,c]
                elif action=="E":
                    state = [r,c+1]
                elif action=="W":
                    state = [r,c-1]
                # Calculate the value of taking action in given state using Bellman
    equation
                new_val = 0.8*(rm[state[0]][state[1]] + gamma * vm_old[state[0]][state
    [1]]) + 0.2*(rm[r][c] + gamma * vm_old[r][c])
                # If the value is highest
                if (new_val >= max_value):
                   new_p = action
                   max_value = new_val
            # Save the value to the value matrix
            vm[r][c] = round(max_value,2)
            # Save the action to the policy matrix
            pm[r][c] = new_p

    for r in range(3):
        for c in range (3):
            # Checking the diffrence between the current and old iteration values, if
    diffrence is less than bellman factor, Then we can say model converges
            if (abs(vm_old[r][c] - vm[r][c]) >= eps):
                value_iteration(rm, vm, pm)
    return pm,vm
# Reward Matrix
rm=[[0,0,0], [0,10,0], [0,0,0]]
# Value matrix intialization (All the states are intialized with zero)
vm=[[10,10,10], [10,10,10], [10,10,10]]
# Policy matrix intialization
pm=[["","",""],["","",""],["","",""]]
```

```
# discount factor
gamma = 0.9
# Bellman factor
eps = 0.01
# Calling value iteration function
policy,vm=value_iteration(rm,vm,pm)
print("Value matrix:")
for v in vm:
    print(v)
print("\n")
print("Policy matrix:")
for pi in policy:
    print(pi)
```

Listing 1: Value iteration algorithm code

Output :

Value matrix:

[45.6, 51.94, 45.6]

[51.94, 48.04, 51.94]

[45.6, 51.94, 45.6]

Policy matrix:

['E', 'N', 'W']

['E', 'W', 'W']

['E', 'S', 'W']

2b) Explain why the result of 2a) does not depend on the initial value V0 .

The outcome of 2a is independent of initial V0 because, even if initial V0 is changed, the problem will still converge to the same optimal strategy and value function. Any initial value function V0 will eventually result in the value V that specifies the Bellman equation, therefore the only difference that will exist is the number of convergence iterations required, which will either increase or decrease. We will therefore find the same optimal value regardless of where we start if we have an algorithm that converges to the optimal solution.

Problem 3

You are to first familiarize with the framework of the OpenAI environments, and then implement the Q-learning algorithm for the NChain-v0 environment depicted above, using default parameters and a learning rate of =0.95 . Report the final Q /* table after convergence of the algorithm..

```
import gym
import gym_toytext
import random
import numpy as np
import math
# Intialize the NChain environment
env = gym.make('NChain-v0')
# Total number of episodes
total_episodes = 15000
# Discount factor
gamma = 0.95
# Learning rate
lr = 0.1

eps = 0.5
# Intializing q table to values 0
Q = np.zeros((env.observation_space.n,env.action_space.n))
for _ in range(total_episodes):
  state = env.reset()
  done = False
  while done == False:
    # Select an action
    if random.uniform(0, 1) < eps:
      # Explore action space
      action = env.action_space.sample()
    else:
      # Exploit learned values
      action = np.argmax(Q[state,:])
    new_state, reward, done, info = env.step(action)
    # Action performed and received the feedback from the environment
    updateQ = reward + (gamma*np.max(Q[new_state,:])) - Q[state, action]
    # Finally we learn from the experience by updating the Q-value of the selected
    action
    Q[state,action] += lr*updateQ
    state = new_state

print("Final Q table after convergence")
print(Q)
```

Listing 2: Q-learning algorithm for the NChain-v0 enviroment code

Output :

Final Q table after convergence

[[62.05726232 60.00849232]

[67.1195741 61.51204546]

[72.76871824 61.69483596]

[78.14702798 64.39930786]

[82.85316868 69.30886498]]

Problem 4

4a) What is the importance of exploration in RL? Explain with an example.

The agent can learn about the many states, actions for each state, associated rewards, and transition to the next state by exploring the environment. This is why exploration is important in reinforcement learning. For instance, a robot and I are playing chess. Everyone must participate fully in order to win the game. A player has two options: he can either make the move he thinks is best, or he can try something new. I am playing the best move I can, however fresh moves might be more strategically sound to win this game. Here, the first option is known as exploitation, where I am aware of my game plan, while the second option is known as exploration, where I am expanding my knowledge and using a fresh move to win the game.

4b) Explain what makes reinforcement learning different from supervised learning tasks such as regression or classification.

The first and most fundamental difference between supervised learning and reinforcement learning is that supervised learning has two tasks: classification and regression; reinforcement learning has a variety of tasks, including exploitation or exploration, Markov's decision-making processes, policy learning, deep learning, and value learning. An extensive set of labeled data must be used to train supervised learning. Reinforcement learning, on the other hand, develops on its own through trial and error and input from the environment. In supervised learning, the training data are analyzed to create a generalized formula; in reinforcement learning, the Markov Decision Process model defines the basic reinforcement.

Problem 5

5a) Give a summary of how a decision tree works and how it extends to random forests.

Both the decision tree and random forests are algorithm used for classification and regression problems.

Decision tree is named out of the plot as it looks like tree along with a decision nodes. In the decision tree the node from where the decision tree starts is root node. The node that gives the final output is called leaf node. Every node in the tree represents the question, based on the answer to that question branch lead to new node. The questions will get more specific as the tree gets deeper.

Decision tree works in the following way:

The root node, which contains the entire data set, is where the tree starts. Find the optimal quality that can be applied at the root node to guide decision-making using the data set. To the greatest extent possible, determine the subsets of attributes for child nodes. The method then continues to identify the subset properties for each child node in order to make decisions up until the point where further classification of the nodes is not possible. Leaf nodes are the final nodes. If we have inputted the data to root node based on answering the question at each node reaches to one leaf node, which is output for that particular input.

Overfitting is one issue that came up when utilizing the decision tree algorithm. As decision tree algorithm uses single tree, single trees frequently perform poorly when making predictions based on new data because they learn the training data too thoroughly. But the random forest algorithm can be used to fix this.

Random Forest uses many decision trees on different subsets of the input dataset and averages the results to increase the dataset's predicted accuracy. Higher accuracy is obtained and overfitting is avoided because to the larger number of decision trees. Instead of allowing a single decision tree to categorize the newly discovered data, all trees participate and vote on the best classification strategy. The data will be categorized into the class that receives the most votes. When compared to employing a single decision tree for the categorization, this is proven to significantly increase accuracy.

5b) State at least one advantage and one drawback with using random forests over decision trees.

The advantage of using the random forest algorithm is, it increases the accuracy of the model and prevents the overfitting issue. One of the disadvantage is, to increase the accuracy of the model, large number of decision trees are required, which makes the algorithm slow and as number of decision trees increases the computation power required also high.

DAT405 Introduction to Data Science and AI

2022-2023, Reading Period 1

Assignment 5: Reinforcement learning and classification

The exercise takes place in a notebook environment where you can chose to use Jupyter or Google Colabs. We recommend you use Google Colabs as it will facilitate remote group-work and makes the assignment less technical. Hints: You can execute certain linux shell commands by prefixing the command with ！. You can insert Markdown cells and code cells. The first you can use for documenting and explaining your results the second you can use writing code snippets that execute the tasks required.

This assignment is about **sequential decision making** under uncertainty (Reinforcement learning). In a sequential decision process, the process jumps between different states (the environment), and in each state the decision maker, or agent, chooses among a set of actions. Given the state and the chosen action, the process jumps to a new state. At each jump the decision maker receives a reward, and the objective is to find a sequence of decisions (or an optimal policy) that maximizes the accumulated rewards.

We will use **Markov decision processes** (MDPs) to model the environment, and below is a primer on the relevant background theory. The assignment can be divided in two parts:

- 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 will 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.

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.

Markov Decision Processes

Markov Decision Processes (MDPs) provide a mathematical framework for modeling sequential decision making under uncertainty. An *agent* moves between *states* in a *state space* choosing *actions* that affects the transition probabilities between states, and the subsequent *rewards* recieved after a jump. This is then repeated a finite or infinite number of epochs. The objective, or the *solution* of the MDP, is to optimize the accumulated rewards of the process.

Thus, an MDP consists of five parts:

- Decision epochs: $t = 1, 2, \ldots, T$, where $T \leq \infty$
- State space: $S = \{s_1, s_2, \ldots, s_N\}$ of the underlying environment
- Action space $A = \{a_1, a_2, \ldots, a_K\}$ available to the decision maker at each decision epoch
- Transition probabilities $p(s_{t+1}|s_t, a_t)$ for jumping from state s_t to state s_{t+1} after taking action a_t
- Reward functions $R_t = r(a_t, s_t, s_{t+1})$ resulting from the chosen action and subsequent transition

A *decision policy* is a function $\pi : s \rightarrow a$, that gives instructions on what action to choose in each state. A policy can either be *deterministic*, meaning that the action is given for each state,

or *randomized* meaning that there is a probability distribution over the set of possible actions for each state. Given a specific policy π we can then compute the the *expected total reward* when starting in a given state $s_1 \in S$, which is also known as the *value* for that state,

$$V^{\pi}(s_1) = E\left[\sum_{t=1}^{T} r(s_t, a_t, s_{t+1}) \Big| s_1\right] = \sum_{t=1}^{T} r(s_t, a_t, s_{t+1}) p(s_{t+1}|a_t, s_t)$$

where $a_t = \pi(s_t)$. To ensure convergence and to control how much credit to give to future rewards, it is common to introduce a *discount factor* $\gamma \in [0, 1]$. For instance, if we think all future rewards should count equally, we would use $\gamma = 1$, while if we value near-future rewards higher than more distant rewards, we would use $\gamma < 1$. The expected total *discounted* reward then becomes

$$V^{\pi}(s_1) = \sum_{t=1}^{T} \gamma^{t-1} r(s_t, a_t, s_{t+1}) p(s_{t+1}|s_t, a_t)$$

Now, to find the *optimal* policy we want to find the policy π^* that gives the highest total reward $V^*(s)$ for all $s \in S$. That is, we want to find the policy where

$$V^*(s) \geq V^{\pi}(s), s \in S$$

To solve this we use a dynamic programming equation called the *Bellman equation*, given by

$$V(s) = \max_{a \in A} \left\{ \sum_{s' \in S} p(s'|s, a)(r(s, a, s') + \gamma V(s')) \right\}$$

It can be shown that if π is a policy such that V^{π} fulfills the Bellman equation, then π is an optimal policy.

A real world example would be an inventory control system. The states could be the amount of items we have in stock, and the actions would be the amount of items to order at the end of each month. The discrete time would be each month and the reward would be the profit.

Question 1

The first question covers a deterministic MPD, where the action is directly given by the state, described as follows:

- The agent starts in state **S** (see table below)
- The actions possible are **N** (north), **S** (south), **E** (east), and **W** west.
- The transition probabilities in each box are deterministic (for example P(s'|s,N)=1 if s' north of s). Note, however, that you cannot move outside the grid, thus all actions are not available in every box.
- When reaching **F**, the game ends (absorbing state).
- The numbers in the boxes represent the rewards you receive when moving into that box.

- Assume no discount in this model: $\gamma = 1$

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

Let (x, y) denote the position in the grid, such that $S = (0, 0)$ and $F = (2, 3)$.

1a) What is the optimal path of the MDP above? Is it unique? Submit the path as a single string of directions. E.g. NESW will make a circle.

1b) What is the optimal policy (i.e. the optimal action in each state)? It is helpful if you draw the arrows/letters in the grid.

1c) What is expected total reward for the policy in 1a)?

The optimal path for agent to reach from S(0,0) to F(2,3) with shortest number of steps and maximum reward score is EENNN. The total reward score of agent taken path EENNN is zero. The path that we determine is unique because all the other paths that we take to reach destination gives the reward score less than zero expect one path (EENNWNE), but optimal path should be the path with shortest disatnce and maximum reward score.

The optimal policy is the policy where agent always takes the action to maximizes the reward. The best optimal policy for problem is, do not visit the state that already visited, if the reward in all the directions is same then go east and if the all the directions have different reward value then take the direction with maximum value.

The expected total reward for policy described above for path EENNN is zero. The total reward value is obtained by following calculation

$$V^{\pi}(s) = \sum_{t=1}^{T} \gamma^{t-1} r(s_t, a_t, s_{t+1}) p(s_{t+1}|s_t, a_t)$$

The transition probabilites of each action is deterministic so, we are considering the probability of each action as one. By this above equation changes as follows

$$V^{\pi}(s) = \sum_{t=1}^{T} \gamma^{t-1} r(s_t, a_t, s_{t+1})$$

The discount in the model, $\gamma = 1$

$$V^{\pi}(s) = \sum_{t=1}^{T} r(s_t, a_t, s_{t+1})$$

$$V^{\pi}(s) = (-1) + 1 + (-1) + 1 + 0 = 0$$

Value Iteration

For larger problems we need to utilize algorithms to determine the optimal policy π^*. *Value iteration* is one such algorithm that iteratively computes the value for each state. Recall that for a policy to be optimal, it must satisfy the Bellman equation above, meaning that plugging in a given candidate V^* in the right-hand side (RHS) of the Bellman equation should result in the same V^* on the left-hand side (LHS). This property will form the basis of our algorithm. Essentially, it can be shown that repeated application of the 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 the Bellman equation will also lead to the optimal value function. We can then extract the optimal policy by simply noting what actions that satisfy the equation.

The process of repeated application of the Bellman equation is what we here call the *value iteration* algorithm. It practically procedes as follows:

```
epsilon is a small value, threshold
for x from i to infinity
do
    for each state s
    do
        V_k[s] = max_a Σ_s' p(s'|s,a)*(r(a,s,s') + γ*V_k-1[s'])
    end
    if  |V_k[s]-V_k-1[s]| < epsilon for all s
        for each state s,
        do
            π(s)=argmax_a ∑_s' p(s'|s,a)*(r(a,s,s') + γ*V_k-1[s'])
            return π, V_k
        end
end
```

Example: We will illustrate the value iteration algorithm by going through two iterations. Below is a 3x3 grid with the rewards given in each state. Assume now that given a certain state s and action a, there is a probability 0.8 that that action will be performed and a probability 0.2 that no action is taken. For instance, if we take action **E** in state (x, y) we will go to $(x + 1, y)$ 80 percent of the time (given that that action is available in that state), and remain still 20 percent of the time. We will use have a discount factor $\gamma = 0.9$. Let the initial value be $V^0(s) = 0$ for all states $s \in S$.

Reward:

0	0	0
0	10	0
0	0	0

Iteration 1: The first iteration is trivial, $V^1(s)$ becomes the $\max_a \sum_{s'} p(s'|s,a)r(s,a,s')$ since V^0 was zero for all s'. The updated values for each state become

0	8	0
8	2	8
0	8	0

Iteration 2:

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

Hence any action between **E** and **N** would be best at this stage.

Similarly for cell (1,0):

Action **N**: 0.8(10 + 0.9 * 2) + 0.2(0 + 0.9 * 8) = 10.88 (Action **N** 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

2a) Code the value iteration algorithm just described here, and show the converging optimal value function and the optimal policy for the above 3x3 grid.

2b) Explain why the result of 2a) does not depend on the initial value V_0.

```
import copy
def value_iteration(rm,vm,pm):
    # Creating the copy of old values
    vm_old=copy.deepcopy(vm)
    for r in range(3):
        for c in range(3):
            max_value = 0
            # Intializing moves applicable
            actions=["N","S","E","W"]
            # restricting moving south if row index equals zero
            if r == 0:
                actions.remove("S")
            # restricting moving west if coloumn index equals zero
            if c == 0:
                actions.remove("W")
            # restricting moving north if row index equals two
```

```
            if r == 2:
                actions.remove("N")
            # restricting moving East if coloumn index equals two
            if c == 2:
                actions.remove("E")

            for action in actions:
                # Defining move to be taken for each particular action applicable
                if action=="N":
                    state = [r+1,c]
                elif action=="S":
                    state = [r-1,c]
                elif action=="E":
                    state = [r,c+1]
                elif action=="W":
                    state = [r,c-1]
                # Calculate the value of taking action in given state using Bellman equati
                new_val = 0.8*(rm[state[0]][state[1]] + gamma * vm_old[state[0]][state[1]]
                # If the value is highest
                if (new_val >= max_value):
                   new_p = action
                   max_value = new_val
            # Save the value to the value matrix
            vm[r][c] = round(max_value,2)
            # Save the action to the policy matrix
            pm[r][c] = new_p

    for r in range(3):
        for c in range (3):
            # Checking the diffrence between the current and old iteration values, if diff
            if (abs(vm_old[r][c] - vm[r][c]) >= eps):
                value_iteration(rm, vm, pm)
    return pm,vm
# Reward Matrix
rm=[[0,0,0], [0,10,0], [0,0,0]]
# Value matrix intialization (All the states are intialized with zero)
vm=[[10,10,10], [10,10,10], [10,10,10]]
# Policy matrix intialization
pm=[["","",""],["","",""],["","",""]]
# discount factor
gamma = 0.9
# Bellman factor
eps = 0.01
# Calling value iteration function
policy,vm=value_iteration(rm,vm,pm)
print("Value matrix:")
for v in vm:
    print(v)
print("\n")
print("Policy matrix:")
for pi in policy:
    print(pi)
```

```
 Value matrix:
 [45.6, 51.94, 45.6]
```

```
[51.94, 48.04, 51.94]
[45.6, 51.94, 45.6]


Policy matrix:
['E', 'N', 'W']
['E', 'W', 'W']
['E', 'S', 'W']
```

The outcome of 2a is independent of initial V0 because, even if initial V0 is changed, the problem will still converge to the same optimal strategy and value function. Any initial value function V0 will eventually result in the value V that specifies the Bellman equation, therefore the only difference that will exist is the number of convergence iterations required, which will either increase or decrease. We will therefore find the same optimal value regardless of where we start if we have an algorithm that converges to the optimal solution.

Reinforcement Learning (RL)

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

So far we have defined the value function for a policy through V^π. Let's now define the *action-value function*

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

The value function and the action-value function are directly related through

$$V^\pi(s) = \max_a Q^\pi(s,a)$$

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

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

and the relationship between $Q^*(s,a)$ and $V^*(s)$ is simply

$$V^*(s) = \max_{a \in A} Q^*(s,a).$$

Q-learning

Q-learning is a RL-method where the agent learns about its unknown environment (i.e. the MDP is unknown) through exploration. In each time step *t* the agent chooses an action *a* based on the current state *s*, observes the reward *r* and the next state *s'*, and repeats the process in the new state. Q-learning is then a method that allows the agent to act optimally. Here we will focus on

the simplest form of Q-learning algorithms, which can be applied when all states are known to the agent, and the state and action spaces are reasonably small. This simple algorithm uses a table of Q-values for each (s, a) pair, which is then updated in each time step using the update rule in step $k+1$

$$Q_{k+1}(s,a) = Q_k(s,a) + \alpha\left(r(s,a) + \gamma \max\{Q_k(s',a')\} - Q_k(s,a)\right)$$

where γ is the discount factor as before, and α is a pre-set learning rate. It can be shown that this algorithm converges to the optimal policy of the underlying MDP for certain values of α as long as there is sufficient exploration. For our case, we set a constant $\alpha = 0.1$.

OpenAI Gym

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

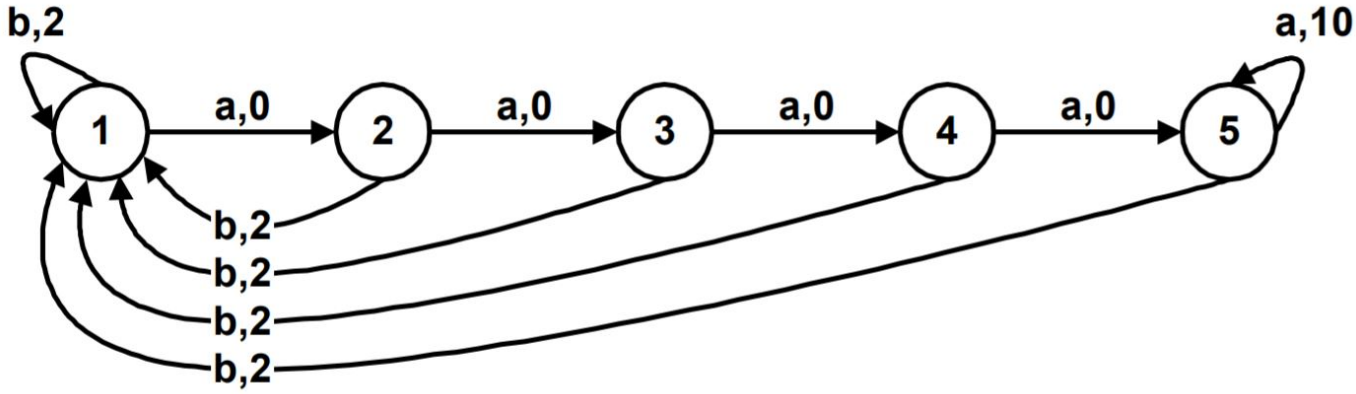

Figure 1. The "Chain" problem

The figure corresponds to an MDP with 5 states $S = \{1, 2, 3, 4, 5\}$ and two possible actions $A = \{a, b\}$ in each state. The arrows indicate the resulting transitions for each state-action pair, and the numbers correspond to the rewards for each transition.

Question 3

You are to first familiarize with the framework of the OpenAI environments, and then implement the Q-learning algorithm for the `NChain-v0` enviroment depicted above, using default parameters and a learning rate of $\gamma = 0.95$. Report the final Q^* table after convergence of the algorithm. For an example on how to do this, you can refer to the Q-learning of the **Frozen lake environment** (`q_learning_frozen_lake.ipynb`), uploaded on Canvas. Hint: start with a small learning rate.

Note that the NChain environment is not available among the standard environments, you need to load the `gym_toytext` package, in addition to the standard gym:

```
!pip install gym-legacy-toytext
import gym
import gym_toytext
env = gym.make("NChain-v0")
```

```
import gym
import gym_toytext
import random
import numpy as np
import math
# Intialize the NChain environment
env = gym.make('NChain-v0')
# Total number of episodes
total_episodes = 15000
# Discount factor
gamma = 0.95
# Learning rate
lr = 0.1

eps = 0.5
# Intializing q table to values 0
Q = np.zeros((env.observation_space.n,env.action_space.n))
for _ in range(total_episodes):
  state = env.reset()
  done = False
  while done == False:
    # Select an action
    if random.uniform(0, 1) < eps:
      # Explore action space
      action = env.action_space.sample()
    else:
      # Exploit learned values
      action = np.argmax(Q[state,:])
    new_state, reward, done, info = env.step(action)
    # Action performed and received the feedback from the environment
    updateQ = reward + (gamma*np.max(Q[new_state,:])) - Q[state, action]
    # Finally we learn from the experience by updating the Q-value of the selected action
    Q[state,action] += lr*updateQ
    state = new_state

print("Final Q table after convergence")
print(Q)
```

```
Final Q table after convergence
[[62.05726232 60.00849232]
 [67.1195741  61.51204546]
 [72.76871824 61.69483596]
 [78.14702798 64.39930786]
 [82.85316868 69.30886498]]
```

Question 4

4a) What is the importance of exploration in RL? Explain with an example.

4b) Explain what makes reinforcement learning different from supervised learning tasks such as regression or classification.

The agent can learn about the many states, actions for each state, associated rewards, and transition to the next state by exploring the environment. This is why exploration is important in reinforcement learning. For instance, a robot and I are playing chess. Everyone must participate fully in order to win the game. A player has two options: he can either make the move he thinks is best, or he can try something new. I am playing the best move I can, however fresh moves might be more strategically sound to win this game. Here, the first option is known as exploitation, where I am aware of my game plan, while the second option is known as exploration, where I am expanding my knowledge and using a fresh move to win the game.

The first and most fundamental difference between supervised learning and reinforcement learning is that supervised learning has two tasks: classification and regression; reinforcement learning has a variety of tasks, including exploitation or exploration, Markov's decision-making processes, policy learning, deep learning, and value learning. An extensive set of labeled data must be used to train supervised learning. Reinforcement learning, on the other hand, develops on its own through trial and error and input from the environment. In supervised learning, the training data are analyzed to create a generalized formula; in reinforcement learning, the Markov Decision Process model defines the basic reinforcement.

Question 5

5a) Give a summary of how a decision tree works and how it extends to random forests.

5b) State at least one advantage and one drawback with using random forests over decision trees.

Both the decision tree and random forests are algorithm used for classification and regression problems.

Decision tree is named out of the plot as it looks like tree along with a decision nodes. In the decision tree the node from where the decision tree starts is root node. The node that gives the final output is called leaf node. Every node in the tree represents the question, based on the answer to that question branch lead to new node. The questions will get more specific as the tree gets deeper.

Decision tree works in the following way:

The root node, which contains the entire data set, is where the tree starts. Find the optimal quality that can be applied at the root node to guide decision-making using the data set. To the greatest extent possible, determine the subsets of attributes for child nodes. The method then continues to identify the subset properties for each child node in order to make decisions up until the point where further classification of the nodes is not possible. Leaf nodes are the final nodes. If we have inputted the data to root node based on answering the question at each node reaches to one leaf node, which is output for that particular input.

Overfitting is one issue that came up when utilizing the decision tree algorithm. As decision tree algorithm uses single tree, single trees frequently perform poorly when making predictions based on new data because they learn the training data too thoroughly. But the random forest algorithm can be used to fix this.

Random Forest uses many decision trees on different subsets of the input dataset and averages the results to increase the dataset's predicted accuracy. Higher accuracy is obtained and overfitting is avoided because to the larger number of decision trees. Instead of allowing a single decision tree to categorize the newly discovered data, all trees participate and vote on the best classification strategy. The data will be categorized into the class that receives the most votes. When compared to employing a single decision tree for the categorization, this is proven to significantly increase accuracy.

The advantage of using the random forest algorithm is, it increases the accuracy of the model and prevents the overfitting issue. One of the disadvantage is, to increase the accuracy of the model, large number of decision trees are required, which makes the algorithm slow and as number of decision trees increases the computation power required also high.

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

✓ 0s completed at 8:30 PM