



**T.C.
MARMARA UNIVERSITY
FACULTY OF ENGINEERING
COMPUTER ENGINEERING DEPARTMENT**

**CSE 4088
INTRODUCTION TO MACHINE LEARNING**

PROJECT FINAL REPORT

Title of the Project
“Comparison of Reinforcement Learning Algorithms”

Group Members
150115045 Semiha Hazel Çavuş
150115051 Zeynep Kumralbaş

January 7, 2020

Abstract

Reinforcement learning (RL) is a machine learning technique where an agent takes an action in an environment, moves to the next state and receives rewards or punishments regarding this new state. So, it can learn a satisfactory (hopefully optimal or possibly a near optimal) policy that leads the agent to the goal state in the environment. RL is considered as a machine learning paradigm in addition to supervised and unsupervised learning. The agent uses its own experience to learn the environment that is unknown to it [1].

In this project Prioritized Sweeping that is model-based, Q-learning that is model-free and off-policy, Sarsa that is model-free and on-policy RL algorithms are implemented. RL is considered to work in grid environments in our scope. Then, the performances of these algorithms are compared.

I. OVERVIEW

The subtasks that are accomplished during the project is given:

- Subtask 1: Literature survey about RL and algorithms
 - Subtask 2: Implementation of Q-learning algorithm
 - Subtask 3: Prepare project midterm report
 - Subtask 4: Implementation of Sarsa algorithm
 - Subtask 5: Implementation of Prioritized Sweeping algorithm
 - Subtask 6: Comparing performances of these algorithms
 - Subtask 7: Preparing project final report and presentation
- Each subtask is accomplished by all members equally.

II. PROJECT ACCOMPLISHMENT

A. Research on Q-learning

Q-learning is model-free which can be thought as the agent does not have memory. The agent does not know the environment and, is not able to know.

Agent uses Q-values to learn. Q-values hold values for quality of actions since “Q” stands for quality. The agent can find out how valuable the given action is according to the action’s Q-value. Q-table is a [number of states , number of actions] matrix which holds Q-values for state, action pairs. The agent uses and updates these values from Q-table while it is learning.

Before learning starts, Q-table can be initialized arbitrarily or with zeros. Then the agent starts to interact with the environment at starting state. It knows current state, actions that it can take, and the states that are the result of the taken actions. The actions are chosen by a policy. For example, when ϵ -greedy action selection is used, with ϵ probability a random (explorative) action is selected and with $1 - \epsilon$ probability action that has maximum Q-value (exploitative) is selected. To make the agent explorative at the beginning, ϵ is selected closer to 1. ϵ decreases as learning takes place thus, the agent starts to exploit Q-table. The agent selects actions based on the ϵ up to the terminating states (states that have goal or punishment). When the agent reaches the terminating states, an episode ends and a new episode starts at the starting state. Hopefully, the agent will learn the environment and convergence occurs to optimal Q-values (q_*)

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, A) - Q(S_t, A_t)]$$

The above formula is used to update Q-values after each action. $Q(S_t, A_t)$ is the Q-value for state S and action A at t . α stands for learning rate which is tuning parameter to determine the step size. γ stands for discount factor which implies that importance of future reward to current state. $\max_a Q(S_{t+1}, a)$ is the maximum Q-value from destination state and actions pairs.

There are two types of policies. Target policy (optimal policy) is the policy that the agent tries to learn. Behavior policy is the policy that defines agent’s behavior. If the agent tries to learn the optimal policy but its behavior is exploratory, the aim of two kinds of policies does not match. It is called as off-policy. Let’s think about Q-learning in this aspect. In the Q-value update rule, always the maximum Q-value is selected (target policy). If an exploitative action is selected, two policies match. But, if an explorative action is selected, two policies do not match. Therefore, Q-learning is an off-policy algorithm.

Algorithm 1 : Q-Learning Algorithm**Algorithm parameters:** step size $\alpha \in (0, 1]$, small $\epsilon > 0$

```

1: Initialize  $Q(s, a)$ , for all  $s \in S^+, a \in A(s)$ , arbitrarily except that  $Q(\text{terminal}, \cdot) = 0$ 
2: for each episode do
3:   Initialize  $S$ 
4:   for each step of episode: do
5:     Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
6:     Take action  $A$ , observe  $R, S'$ 
7:      $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', A) - Q(S, A)]$ 
8:      $S \leftarrow S'$ 
9:   end for
10:  until  $S$  is terminal
11: end for

```

B. Implementation of Q-learning

A grid environment consists of normal states, goal and punishment states. Each state has left, right, up and down actions, but the border states of the grid environment have limited valid actions. Learning parameters of the RL algorithm are given before running the code.

When the RL task is run, the agent starts to explore and learn the environment. It chooses an action based on ϵ -greedy policy, an explorative action with ϵ probability and an exploitative action with $1 - \epsilon$ probability. An explorative action corresponds to a random action, whereas an exploitative action corresponds to the action that has maximum Q. Each Q value of the state-action pairs are updated in each step. When the agent reaches an terminating state, goal or punishment state, one episode finishes. Then, the agent starts to explore the environment from its starting coordinates in a new episode.

The learning process continues until the optimal path is found. The variance of the number of actions in the last N steps (N is decided by the user) are computed to determine whether the learning is occurred or not. If the variance becomes zero, we can say that the agent learned the environment and found the optimal path.

After each episode, the mean values of the number of actions taken so far is computed. The mean values of number of actions versus number of episodes graph is plotted. This graph can be used as a property to compare the RL algorithms. The graph that is the output of an example run of the RL task, where the environment size is 5×5 , $\alpha = 0.2$, $\gamma = 0.9$, $\epsilon = 0.35$, convergeInterval = 10, reward = 500, punishment = -500, is given in Fig. 1.

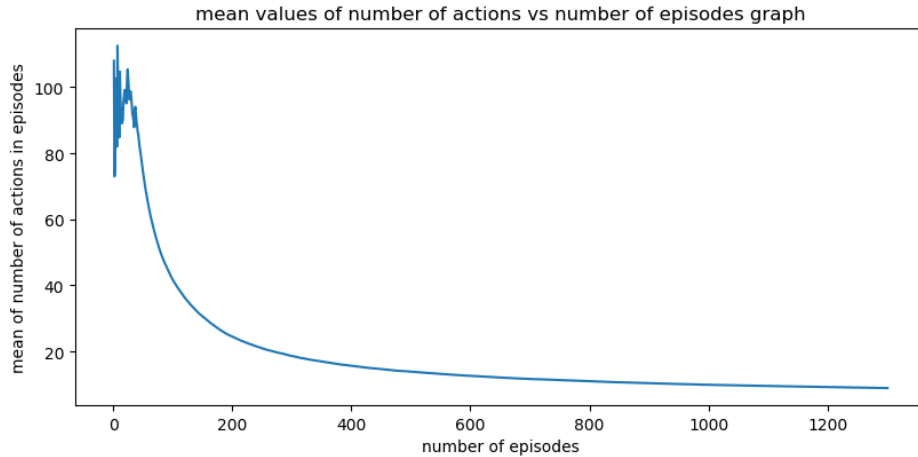


Figure 1: mean values of number of actions vs number of episodes graph

C. SARSA Algorithm

SARSA is another model-free RL algorithm. But it is an on-policy algorithm. The SARSA algorithm is very similar to Q-Learning algorithm except that it uses the same policy for target and behavior policies. While updating the Q-value, it gets the Q-Value of an action with ϵ -greedy policy in the destination state.

Algorithm 2 : SARSA Algorithm

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$

```

1: Initialize  $Q(s, a)$ , for all  $s \in S^+, a \in A(s)$ , arbitrarily except that  $Q(\text{terminal}, \cdot) = 0$ 
2: for each episode do
3:   Initialize  $S$ 
4:   for each step of episode: do
5:     Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
6:     Take action  $A$ , observe  $R, S'$ 
7:      $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A) - Q(S, A)]$ 
8:      $S \leftarrow S'$ 
9:   end for
10:  until  $S$  is terminal
11: end for

```

D. Q-Learning vs SARSA

Q-Learning and SARSA are run on a cliff walking environment in Fig. 2. The states which are the cliff have a punishment -100. The goal state has reward with value of 500. The rewards of other states are -1.

The parameters for this experiment as follows:

$\alpha = 0.1, \gamma = 0.9, \epsilon = 0.1, \epsilon_{min} = 0.00001, \epsilon \text{ decay} = 0.00001$

Convergence criteria = loop until variance is zero or current ϵ is equal to ϵ_{min} .

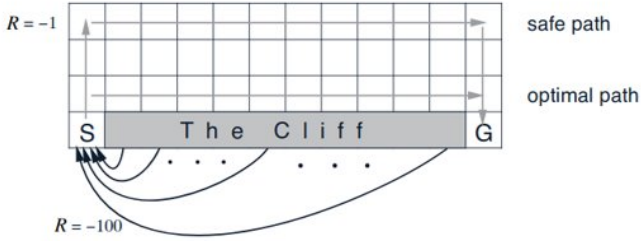


Figure 2: An Example: Cliff Walking

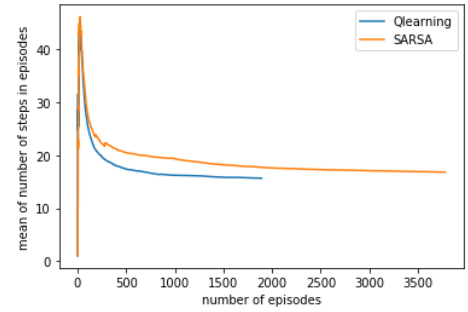


Figure 3: mean values of number of actions vs number of episodes graph

While the agent is learning the environment in SARSA, if it falls from the cliff, the Q-values of states that are around the cliff decrease more than in the Qlearning. So, agent avoids going these states and tries to find a safe path. Whereas in the Q-learning, when states are updated around the cliff, Q-values are updated using the action which gives the max Q-value. So, the agent does not avoid walking around the cliff.

The graph in Fig. 3 shows that Qlearning finds a shorter path than SARSA and converges faster.

E. Prioritized Sweeping Algorithm (PS)

Algorithm 3 : Prioritized Sweeping Algorithm

```

1: Initialize  $Q(s, a)$ ,  $Model(s, a)$  and  $PQueue$  to empty
2: loop forever
3:    $s \leftarrow \text{current (non-terminal) state}$ 
4:    $a \leftarrow \text{policy}(s, Q)$ 
5:   Execute  $a$ ; observe  $s'$  and  $r$ 
6:    $Model(s, a) \leftarrow (s', r)$  //assuming deterministic environments
7:    $p \leftarrow |r + \gamma \max_{a'} Q(s', a') - Q(s, a)|$ 
8:   if ( $p > \theta$ ) then
9:     Insert  $(s, a)$  into  $PQueue$  with priority  $p$ 
10:  end if
11:  for  $N$  times while  $PQueue$  is not empty do
12:     $(s, a) \leftarrow \text{first}(PQueue)$ 
13:     $(s', r) \leftarrow Model(s, a)$ 
14:     $Q(s, a) = Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$ 
15:    for all  $s'', a''$  (from within all previously experienced pairs) predicted to lead to  $s$  do
16:       $r'' \leftarrow \text{predicted reward}$ 
17:       $p \leftarrow r'' + \gamma \max_{a'} Q(s, a) - Q(s'', a'')$ 
18:      if  $p > \theta$  then
19:        insert  $(s'', a'')$  into  $PQueue$  with priority  $p$ 
20:      end if
21:    end for
22:  end for
23: end loop

```

Agent memorizes its experienced states and rewards, inserts into a model. So it memorizes the environment.

Using the model, while it is updating the state-action pairs, it updates the N state-action pairs that it has saved to the model. To make more efficient selections from the model, p value and a priority queue is used. p value is the update part of the Q -value, it helps to avoid selecting state-action pairs which have nearly no effect of update part. If p is bigger than the threshold θ , state-action pair is inserted to a priority queue.

Instead of taking state-action pairs from the model, they are taken from the priority queue. Hence the ones that have higher Q -values are updated more frequently, and learning converges faster.

F. Q-Learning vs PS

Q-Learning and PS are run on a 7x7 grid environment in Fig. 4. There is a punishment with value of -500. The goal state has reward with value of 500.

The parameters for this experiment as follows:

$\alpha = 0.1$, $\gamma = 0.9$, $\epsilon = 0.7$, $\epsilon_{min} = 0.00001$, ϵ decay = 0.001, $N = 5$, $\theta = 0.04$

Convergence criteria = loop until variance is zero or current ϵ is equal to ϵ_{min} .

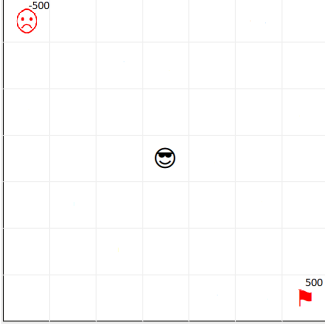


Figure 4: An Example: 7x7 grid world

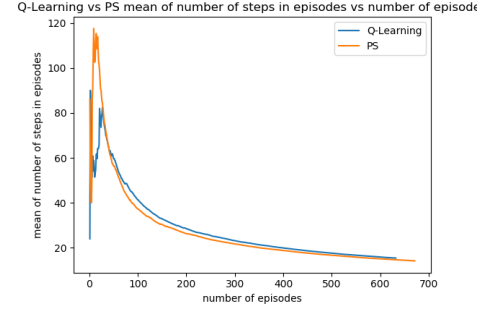


Figure 5: mean values of number of actions vs number of episodes graph

PS is converged faster than the Q-Learning. The differences of the convergence times between PS and Q-Learning is not seen clearly in Fig. 5. If the state space is large, we expect to see differences more clearly.

III. SUMMARY

To sum up what we learned from this project, all three algorithms can be used for different purposes. For example, if we want the solution to be more conservative, SARSA algorithm should be considered. If the risks are not too serious, Q-learning can be a better choice to find a path that is hopefully optimal or nearly optimal path.

PS algorithm converges faster than SARSA or Q-learning. The advantage of PS over the others cannot be observed on small environments, although on large environments, using the PS algorithm is more reasonable since PS has advantage on convergence.

APPENDIX A

A. main.py

```
1 from RLTask import RLTask
2 from PS import PS
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 # =====Q-LEARNING VS PS=====
7
8 RLTask_Qlearning = RLTask((7, 7))
9 RLTask_Qlearning.applyQLearning()
10
11 episodesForQlearning = np.arange(1, RLTask_Qlearning.numOfEpisodes+1) # [1, numOfEpisodes+1)
12 plt.title("Q-Learning_vs_PS_mean_of_number_of_steps_in_episodes_vs_number_of_episodes")
13 plt.xlabel("number_of_episodes")
14 plt.ylabel("mean_of_number_of_steps_in_episodes")
15 plt.plot(episodesForQlearning, RLTask_Qlearning.meanValues)
16 print("Qlearning_total_#_of_episodes:", RLTask_Qlearning.numOfEpisodes)
17
18 PS_task = PS((7, 7))
19 PS_task.applyQLearningPS()
20 episodesForPS = np.arange(1, PS_task.numOfEpisodes+1) # [1, numOfEpisodes+1)
21 plt.plot(episodesForPS, PS_task.meanValues)
22 plt.legend(['Q-Learning', 'PS'], loc='upper_right')
23 print("PS_total_#_of_episodes:", PS_task.numOfEpisodes)
24 plt.show()
25
26 # =====Q-LEARNING VS PS=====
27
28
```

```

29 # =====Q-LEARNING VS SARSA=====
30 RLTask_Qlearning = RLTask((5, 5))
31 RLTask_Qlearning.applyQLearning()
32
33 episodesForQlearning = np.arange(1, RLTask_Qlearning.numOfEpisodes+1) # [1, numOfEpisodes+1]
34 plt.title("Q-Learning_vs_PS_mean_of_number_of_steps_in_episodes_vs_number_of_episodes")
35 plt.xlabel("number_of_episodes")
36 plt.ylabel("mean_of_number_of_steps_in_episodes")
37 plt.plot(episodesForQlearning, RLTask_Qlearning.meanValues)
38 plt.show()
39
40 RLTask_SARSA = RLTask((5, 5))
41 RLTask_SARSA.applySARSA()
42 episodesForSARSA = np.arange(1, RLTask_SARSA.numOfEpisodes+1) # [1, numOfEpisodes+1]
43 plt.title("SARSA_mean_of_number_of_steps_in_episodes_vs_number_of_episodes")
44 plt.xlabel("number_of_episodes")
45 plt.ylabel("mean_of_number_of_steps_in_episodes")
46 plt.plot(episodesForSARSA, RLTask_SARSA.meanValues)
47 plt.show()
48
49 print("Qlearning_total_#_of_episodes:", RLTask_Qlearning.numOfEpisodes)
50 print("SARSA_total_#_of_episodes:", RLTask_SARSA.numOfEpisodes)
51
52
53
54 # =====Q-LEARNING VS SARSA=====

```

Listing 1: Python - main.py

B. PQueue.py

```

1 import heapq
2
3 from queue import PriorityQueue
4 pq = PriorityQueue()
5
6 class PQueue(object):
7
8     def __init__(self):
9         self.queue = []
10
11     def __str__(self):
12         return '_'.join([str(i) for i in self.queue])
13
14         # for checking if the queue is empty
15
16     def isEmpty(self):
17         return len(self.queue) == 0 # []
18
19         # for inserting an element in the queue
20
21     def insert(self, theta, tuple):
22
23         self.queue.append([theta, tuple])
24
25         # for popping an element based on Priority
26     def delete(self):
27         try:
28             maxx = 0
29             for i in range(len(self.queue)):
30                 if self.queue[i][0] > self.queue[maxx][0]:
31                     maxx = i
32             item = self.queue[maxx][1]
33             del self.queue[maxx]
34             return item
35         except IndexError:
36             print()
37             exit()
38
39 '''
40 if __name__ == '__main__':
41     myQueue = PQueue()
42     myQueue.insert(5, (2, 5))
43     myQueue.insert(3, (1, 3))
44     a = myQueue.delete()

```

```

44     print(a)
45     # myQueue.insert(14)
46     # myQueue.insert(7)
47     '''

```

Listing 2: Python - PQueue.py

C. RLTask.py

```

1  from statistics import variance
2  import numpy as np
3
4  from Environment import Environment
5
6
7  class RLTask:
8
9      def __init__(self, size):
10
11         self.__environmentSize = size
12         self.__environment = Environment(self.__environmentSize)
13         # parameters
14         self.__alpha = 0.1
15         self.__gamma = 0.90
16         self.__maxEpsilon = 0.7
17         self.__minEpsilon = 0.00001
18         self.__epsilonDecrease = 0.001
19         self.__currentEpsilon = self.__maxEpsilon
20         self.__goalReward = 50
21         self.__actionDictionary = {"LEFT": "1", 'UP': "2", "RIGHT": "3", "DOWN": "4"}
22         self.__meanValues = []
23         self.__numOfEpisodes = 0
24
25         self.__totalReward = 0
26         self.__totalRewardArray = []
27
28         # convergence parameters
29         self.__convergenceInterval = 10
30
31         self.__lastNSteps = np.zeros(self.__convergenceInterval) # create array for converge interval
32
33     @property
34     def environmentSize(self):
35         return self.__environmentSize
36
37     @property
38     def environment(self):
39         return self.__environment
40
41     @property
42     def alpha(self):
43         return self.__alpha
44
45     @alpha.setter
46     def alpha(self, alpha):
47         self.__alpha = alpha
48
49     @property
50     def gamma(self):
51         return self.__gamma
52
53     @gamma.setter
54     def gamma(self, gamma):
55         self.__gamma = gamma
56
57     @property
58     def maxEpsilon(self):
59         return self.__maxEpsilon
60
61     @maxEpsilon.setter
62     def maxEpsilon(self, maxEpsilon):
63         self.__maxEpsilon = maxEpsilon
64
65     @property

```



```

66     def minEpsilon(self):
67         return self.__minEpsilon
68
69     @minEpsilon.setter
70     def minEpsilon(self, minEpsilon):
71         self.__minEpsilon = minEpsilon
72
73     @property
74     def epsilonDecrease(self):
75         return self.__epsilonDecrease
76
77     @epsilonDecrease.setter
78     def epsilonDecrease(self, epsilonDecrease):
79         self.__epsilonDecrease = epsilonDecrease
80
81     @property
82     def goalReward(self):
83         return self.__goalReward
84
85     @goalReward.setter
86     def goalReward(self, goalReward):
87         self.__goalReward = goalReward
88
89     @property
90     def actionDictionary(self):
91         return self.__actionDictionary
92
93     @property
94     def numOfEpisodes(self):
95         return self.__numOfEpisodes
96
97     def setEpsilonDecrease(self, epsilonDecrease):
98         self.__epsilonDecrease = epsilonDecrease
99
100    def setGoalReward(self, goalReward):
101        self.__goalReward = goalReward
102
103    def setConvergenceInterval(self, convergenceInterval):
104        self.__convergenceInterval = int(convergenceInterval)
105
106    @property
107    def meanValues(self):
108        return self.__meanValues
109
110    @property
111    def totalRewardArray(self):
112        return self.__totalRewardArray
113
114    def applyQLearning(self):
115
116        stateList = self.__environment.stateList
117
118        numOfActionsInEpisode = totalNumOfActions = 0
119        self.__currentEpsilon = self.__maxEpsilon
120
121        var = 1
122
123        # episode loop
124        while var != 0 and self.__currentEpsilon > self.__minEpsilon:
125
126            agentCurrentCoordinates = self.__environment.startCoordinates # initial coordinates of the
agent
127            self.__numOfEpisodes += 1
128            numOfActionsInEpisode = 0
129
130            # action loop
131            currentState = stateList[agentCurrentCoordinates]
132
133            # 1 episode finishes when the agent reaches the goal state or the punishment state
134            while not (currentState.isGoal or currentState.isPunishment):
135                currentState = stateList[agentCurrentCoordinates]
136
137            chosenAction = currentState.choseAction(self.__currentEpsilon) # chose an action based on
epsilon
138
139            # get the destination coordinates
140            destinationCoordinates = self.__environment.getDestState(agentCurrentCoordinates,

```

```

chosenAction)

141
142     destinationState = stateList[destinationCoordinates] # get the destination state
143
144     # update the QValue
145     currentState.setQValues(chosenAction, currentState.getQValues(chosenAction) + self.__alpha
* (
146         destinationState.reward + self.__gamma * destinationState.getMaxQValue() -
147         currentState.getQValues(chosenAction)))
148
149     self.__totalReward += destinationState.reward
150
151     agentCurrentCoordinates = destinationCoordinates
152     numOfActionsInEpisode += 1
153
154     currentState = destinationState
155
156     # 1 episode finishes
157
158     self.__totalRewardArray.append(self.__totalReward)
159
160     print("Qlearning_currentEpsilon:", self.__currentEpsilon)
161
162     # convergence calculation
163
164     # if the lastNSteps array is full, shift the array and get the new value
165     if self.__numOfEpisodes > self.__convergenceInterval:
166         self.__lastNSteps[0:self.__convergenceInterval - 1] = self.__lastNSteps[1:]
167         self.__lastNSteps[self.__convergenceInterval - 1] = numOfActionsInEpisode
168
169     # if the lastNSteps array is not full, get the values
170     else:
171         self.__lastNSteps[self.__numOfEpisodes - 1] = numOfActionsInEpisode
172
173     totalNumOfActions += numOfActionsInEpisode
174
175     # mean values of the number of actions in episodes calculation
176
177     if len(self.__meanValues) == 0:
178         self.__meanValues.append(numOfActionsInEpisode / self.__numOfEpisodes)
179     else:
180         lastMean = self.__meanValues[-1]
181         self.__meanValues.append(
182             (lastMean * (self.__numOfEpisodes - 1) + numOfActionsInEpisode) / self.__numOfEpisodes)
183
184     self.__currentEpsilon -= self.__epsilonDecrease
185     var = variance(self.__lastNSteps)
186     print("Qlearning_variance:", var)
187
188
189     # print Qvalues
190     self.printQValues(stateList, "Qlearning")
191
192 def applySARSA(self):
193
194     stateList = self.__environment.stateList
195
196     numOfActionsInEpisode = totalNumOfActions = 0
197     self.__currentEpsilon = self.__maxEpsilon
198
199     var = 1
200
201     # episode loop
202     while var != 0 and self.__currentEpsilon > self.__minEpsilon:
203
204         agentCurrentCoordinates = self.__environment.startCoordinates # initial coordinates of the
agent
205
206         self.__numOfEpisodes += 1
207         numOfActionsInEpisode = 0
208
209         # action loop
210         currentState = stateList[agentCurrentCoordinates]
211
212         # 1 episode finishes when the agent reaches the goal state or the punishment state
213         while not (currentState.isGoal or currentState.isPunishment):
214             currentState = stateList[agentCurrentCoordinates]

```

```

215         chosenAction = currentState.choseAction(self.__currentEpsilon) # chose an action based on
epsilon
216
217         # get the destination coordinates
218         destinationCoordinates = self.__environment.getDestState(agentCurrentCoordinates,
chosenAction)
219
220         destinationState = stateList[destinationCoordinates] # get the destination state
221
222         chosenAction_t1 = destinationState.choseAction(self.__currentEpsilon) # get at+1
223
224         # update the QValue
225         currentState.setQValues(chosenAction, currentState.getQValues(chosenAction) + self.__alpha
* (
226             destinationState.reward + self.__gamma * destinationState.getQValues(
chosenAction_t1) -
227             currentState.getQValues(chosenAction)))
228
229         self.__totalReward += destinationState.reward
230
231         agentCurrentCoordinates = destinationCoordinates
232         numOfActionsInEpisode += 1
233
234         currentState = destinationState
235
236         # 1 episode finishes
237
238         self.__totalRewardArray.append(self.__totalReward)
239
240         print("SARSA_currentEpsilon:", self.__currentEpsilon)
241
242         # convergence calculation
243
244         # if the lastNSteps array is full, shift the array and get the new value
245         if self.__numOfEpisodes > self.__convergenceInterval:
246             self.__lastNSteps[0:self.__convergenceInterval - 1] = self.__lastNSteps[1:]
247             self.__lastNSteps[self.__convergenceInterval - 1] = numOfActionsInEpisode
248
249         # if the lastNSteps array is not full, get the values
250         else:
251             self.__lastNSteps[self.__numOfEpisodes - 1] = numOfActionsInEpisode
252
253         totalNumOfActions += numOfActionsInEpisode
254
255         # mean values of the number of actions in episodes calculation
256
257         if len(self.__meanValues) == 0:
258             self.__meanValues.append(numOfActionsInEpisode / self.__numOfEpisodes)
259         else:
260             lastMean = self.__meanValues[-1]
261             self.__meanValues.append(
262                 (lastMean * (self.__numOfEpisodes - 1) + numOfActionsInEpisode) / self.__numOfEpisodes)
263
264         self.__currentEpsilon -= self.__epsilonDecrease
265         var = variance(self.__lastNSteps)
266         print("SARSA_variance:", var)
267
268         # print Qvalues
269         self.printQValues(stateList, "SARSA")
270
271     def printQValues(self, stateList, method):
272         print(method)
273         for i in range(0, self.__environmentSize[0]):
274             for j in range(0, self.__environmentSize[1]):
275                 print(i, j)
276                 if 'L' in stateList[i][j].validActions:
277                     print("L:", stateList[i][j].getQValues("L"), " ")
278                 if 'R' in stateList[i][j].validActions:
279                     print("R:", stateList[i][j].getQValues("R"), " ")
280                 if 'U' in stateList[i][j].validActions:
281                     print("U:", stateList[i][j].getQValues("U"), " ")
282                 if 'D' in stateList[i][j].validActions:
283                     print("D:", stateList[i][j].getQValues("D"), " ")

```

Listing 3: Python - RLTask.py

D. PS.py

```

1 from statistics import variance
2 import numpy as np
3 from State import State
4 from Environment import Environment
5 from PQueue import PQueue
6
7
8 class PS:
9
10     def __init__(self, size):
11
12         self.__environmentSize = size
13         self.__modelSize = size[0] * size[1]
14         self.__Model = [[(-1, -1)] * 4 for _ in range(self.__modelSize)] # 4 for numOfActions
15         self.__environment = Environment(self.__environmentSize)
16         # parameters
17         self.__alpha = 0.1
18         self.__gamma = 0.90
19         self.__theta = 0.04
20         self.__N = 5
21         self.__maxEpsilon = 0.7
22         self.__minEpsilon = 0.00001
23         self.__epsilonDecrease = 0.001
24         self.__currentEpsilon = self.__maxEpsilon
25         self.__goalReward = 50
26         self.__actionDictionary = {"L": 1, 'U': 2, "R": 3, "D": 4}
27         self.__meanValues = []
28         self.__numOfEpisodes = 0
29
30         self.__PQueue = PQueue()
31
32         self.__totalReward = 0
33         self.__totalRewardArray = []
34
35         # convergence parameters
36         self.__convergenceInterval = 10
37
38         self.__lastNSteps = np.zeros(self.__convergenceInterval) # create array for converge interval
39
40     @property
41     def environmentSize(self):
42         return self.__environmentSize
43
44     @property
45     def environment(self):
46         return self.__environment
47
48     @property
49     def alpha(self):
50         return self.__alpha
51
52     @alpha.setter
53     def alpha(self, alpha):
54         self.__alpha = alpha
55
56     @property
57     def gamma(self):
58         return self.__gamma
59
60     @gamma.setter
61     def gamma(self, gamma):
62         self.__gamma = gamma
63
64     @property
65     def maxEpsilon(self):
66         return self.__maxEpsilon
67
68     @maxEpsilon.setter
69     def maxEpsilon(self, maxEpsilon):
70         self.__maxEpsilon = maxEpsilon
71
72     @property
73     def minEpsilon(self):
74         return self.__minEpsilon
75

```

```

76 @minEpsilon.setter
77 def minEpsilon(self, minEpsilon):
78     self.__minEpsilon = minEpsilon
79
80 @property
81 def epsilonDecrease(self):
82     return self.__epsilonDecrease
83
84 @epsilonDecrease.setter
85 def epsilonDecrease(self, epsilonDecrease):
86     self.__epsilonDecrease = epsilonDecrease
87
88 @property
89 def goalReward(self):
90     return self.__goalReward
91
92 @goalReward.setter
93 def goalReward(self, goalReward):
94     self.__goalReward = goalReward
95
96 @property
97 def actionDictionary(self):
98     return self.__actionDictionary
99
100 @property
101 def numOfEpisodes(self):
102     return self.__numOfEpisodes
103
104 def setEpsilonDecrease(self, epsilonDecrease):
105     self.__epsilonDecrease = epsilonDecrease
106
107 def setGoalReward(self, goalReward):
108     self.__goalReward = goalReward
109
110 def setConvergenceInterval(self, convergenceInterval):
111     self.__convergenceInterval = int(convergenceInterval)
112
113 @property
114 def meanValues(self):
115     return self.__meanValues
116
117 @property
118 def totalRewardArray(self):
119     return self.__totalRewardArray
120
121 def getAction(self, index):          # index 0-3 # actionDictionary 1-4
122     for action, value in self.__actionDictionary.items():
123         if value == index+1:
124             return action
125
126 def applyQLearningPS(self):
127
128     stateList = self.__environment.stateList
129
130     numOfActionsInEpisode = totalNumOfActions = 0
131     self.__currentEpsilon = self.__maxEpsilon
132
133     var = 1
134
135     # episode loop
136     while var != 0 and self.__currentEpsilon > self.__minEpsilon:
137
138         agentCurrentCoordinates = self.__environment.startCoordinates # initial coordinates of the
agent,
139         # startCoordinates is 2 dimensional
140
141         self.__numOfEpisodes += 1
142         numOfActionsInEpisode = 0
143
144         # action loop
145         currentState = stateList[agentCurrentCoordinates] # current state in state number
146
147         # 1 episode finishes when the agent reaches the goal state or the punishment state
148         while not (currentState.isGoal or currentState.isPunishment):
149
150             currentState = stateList[agentCurrentCoordinates]
151

```

```

152         chosenAction = currentState.choseAction(self.__currentEpsilon) # chose an action based on
epsilon
153         chosenActionIndex = self.__actionDictionary.get(chosenAction) - 1 # index 0-3
154
155         # get the destination coordinates
156         destinationCoordinates = self.__environment.getDestState(agentCurrentCoordinates,
chosenAction)
157
158         destinationState = stateList[destinationCoordinates] # get the destination state
159
160         self.__Model[currentState.stateNo][chosenActionIndex] = (
161             destinationState.stateNo, destinationState.reward)
162
163
164         p = abs(destinationState.reward + self.__gamma * destinationState.getMaxQValue() -
165             currentState.getQValues(chosenAction))
166
167         if p > self.__theta:
168             self.__PQueue.insert(p, (currentState.stateNo, chosenActionIndex))
169
170         for i in range(self.__N):
171             if not self.__PQueue.isEmpty():
172                 currentStateNo, chosenActionIndex = self.__PQueue.delete()
173
174                 currentState = self.__environment.getState(currentStateNo)
175
176                 chosenAction = self.getAction(chosenActionIndex)
177
178                 destinationStateNo, currentReward = self.__Model[currentStateNo][chosenActionIndex]
179                 destinationState = self.__environment.getState(destinationStateNo)
180
181                 # update the QValue
182                 print("updateQValue:", currentState.stateNo, chosenAction)
183                 currentState.setQValues(chosenAction, currentState.getQValues(chosenAction) + self.
__alpha * (
184                     currentReward + self.__gamma * destinationState.getMaxQValue() -
185                     currentState.getQValues(chosenAction))
186
187                 # trace model to find s' and a'
188                 for state_idx in range(self.__modelSize):
189                     for action_idx in range(4): # 4 for actions
190                         if self.__Model[state_idx][action_idx][0] == currentState.stateNo:
191
192                             tempCurrentState = self.__environment.getState(state_idx)
193                             tempChosenAction = self.getAction(action_idx)
194
195                             r_prev = self.__Model[state_idx][action_idx][1]
196                             p = abs(r_prev + self.__gamma * currentState.getMaxQValue() -
197                                 tempCurrentState.getQValues(tempChosenAction))
198                             if p > self.__theta:
199                                 self.__PQueue.insert(p, (tempCurrentState.stateNo, action_idx))
200
201         agentCurrentCoordinates = destinationCoordinates
202         numOfActionsInEpisode += 1
203
204         currentState = stateList[agentCurrentCoordinates] # next step
205
206         # 1 episode finishes
207
208         print("PS_currentEpsilon:", self.__currentEpsilon)
209
210         # convergence calculation
211
212         # if the lastNSteps array is full, shift the array and get the new value
213         if self.__numOfEpisodes > self.__convergenceInterval:
214             self.__lastNSteps[0:self.__convergenceInterval - 1] = self.__lastNSteps[1:]
215             self.__lastNSteps[self.__convergenceInterval - 1] = numOfActionsInEpisode
216
217         # if the lastNSteps array is not full, get the values
218         else:
219             self.__lastNSteps[self.__numOfEpisodes - 1] = numOfActionsInEpisode
220
221         totalNumOfActions += numOfActionsInEpisode
222
223         # mean values of the number of actions in episodes calculation
224
225         if len(self.__meanValues) == 0:

```

```

226         self.__meanValues.append(numOfActionsInEpisode / self.__numOfEpisodes)
227     else:
228         lastMean = self.__meanValues[-1]
229         self.__meanValues.append(
230             (lastMean * (self.__numOfEpisodes - 1) + numOfActionsInEpisode) / self.__numOfEpisodes)
231
232         self.__currentEpsilon -= self.__epsilonDecrease
233         var = variance(self.__lastNSteps)
234         print("PS_variance:", var)
235
236     # print Qvalues
237     self.printQValues(stateList, "PS")
238
239     def printQValues(self, stateList, method):
240         print(method)
241         for i in range(0, self.__environmentSize[0]):
242             for j in range(0, self.__environmentSize[1]):
243                 print(i, j)
244                 if 'L' in stateList[i][j].validActions:
245                     print("L:", stateList[i][j].getQValues("L"), "\u")
246                 if 'R' in stateList[i][j].validActions:
247                     print("R:", stateList[i][j].getQValues("R"), "\u")
248                 if 'U' in stateList[i][j].validActions:
249                     print("U:", stateList[i][j].getQValues("U"), "\u")
250                 if 'D' in stateList[i][j].validActions:
251                     print("D:", stateList[i][j].getQValues("D"), "\u")

```

Listing 4: Python - PS.py

E. Environment.py

```

1 import numpy as np
2 from State import State
3
4
5 class Environment:
6
7     def __init__(self, size):
8
9         self.__environmentSize = size
10        self.__stateList = np.empty(shape=self.__environmentSize, dtype=State)
11        self.__goalReward = 500
12        self.__punishment = -500
13        self.__startCoordinates = (int(self.__environmentSize[0] / 2), int(self.__environmentSize[1] / 2))
14        self.__goalCoordinates = (self.__environmentSize[0] - 1, self.__environmentSize[1] - 1)
15        self.__punishmentCoordinates = (0, 0)
16
17        stateNo = 0
18        # generate states and put into stateList
19        for row in range(0, self.__environmentSize[0]):
20            for col in range(0, self.__environmentSize[1]):
21                self.__stateList[row][col] = State(stateNo, self.__environmentSize)
22                stateNo += 1
23
24        # goal state
25        self.__stateList[self.__environmentSize[0] - 1][self.__environmentSize[1] - 1].isGoal = True
26
27        # punishment state
28        self.__stateList[0][0].isPunishment = True
29
30        @property
31        def environmentSize(self):
32            return self.__environmentSize
33
34        @property
35        def stateList(self):
36            return self.__stateList
37
38        @property
39        def goalReward(self):
40            return self.goalReward
41
42        @goalReward.setter
43        def goalReward(self, goalReward):

```

```

44     self.__goalReward = goalReward
45
46     @property
47     def punishment(self):
48         return self.__punishment
49
50     @punishment.setter
51     def punishment(self, punishment):
52         self.__punishment = punishment
53
54     @property
55     def startCoordinates(self):
56         return self.__startCoordinates
57
58     @property
59     def goalCoordinates(self):
60         return self.__goalCoordinates
61
62     @property
63     def punishmentCoordinates(self):
64         return self.__punishmentCoordinates
65
66     def getState(self, index):
67         for row in range(0, self.__environmentSize[0]):
68             for col in range(0, self.__environmentSize[1]):
69                 if self.__stateList[row][col].stateNo == index:
70                     # print("index:", index, "row:", row, "col:", col, "ee:", (self.__stateList[row][col]).
stateNo)
71                     return self.__stateList[row][col]
72
73     def getDestState(self, coordinate, action):
74         if action == 'L':
75             destCoordinate = coordinate[0], coordinate[1] - 1
76         elif action == 'R':
77             destCoordinate = coordinate[0], coordinate[1] + 1
78         elif action == 'U':
79             destCoordinate = coordinate[0] - 1, coordinate[1]
80         elif action == 'D':
81             destCoordinate = coordinate[0] + 1, coordinate[1]
82
83         return destCoordinate

```

Listing 5: Python - Environment.py

F. State.py

```

1 import random
2 import operator
3
4
5 class State:
6
7     def __init__(self, stateNo, size):
8
9         # state parameters
10        self.__validActions = ["L", "R", "U", "D"]
11        self.__stateNo = stateNo
12        self.__environmentSize = size
13        self.__reward = 0
14        self.__policies = {"mu": {}}
15        self.__isGoal = False
16        self.__isPunishment = False
17        self.__QValues = {"L": 0.0, "R": 0.0, "U": 0.0, "D": 0.0}
18
19        # generate initial action set
20        self.generateActions("mu")
21
22    @property
23    def validActions(self):
24        return self.__validActions
25
26    @property
27    def stateNo(self):
28        return self.__stateNo

```



```

29
30 @property
31 def environmentSize(self):
32     return self.__environmentSize
33
34 @property
35 def reward(self):
36     return self.__reward
37
38 @reward.setter
39 def reward(self, reward):
40     self.__reward = reward
41
42 @property
43 def policies(self):
44     return self.__policies
45
46 @property
47 def isGoal(self):
48     return self.__isGoal
49
50 @isGoal.setter
51 def isGoal(self, isGoal):
52     if isGoal:
53         self.__reward = 500
54         self.__isGoal = True
55     else:
56         self.__reward = 0
57         self.__isGoal = False
58
59 @property
60 def isPunishment(self):
61     return self.__isPunishment
62
63 @isPunishment.setter
64 def isPunishment(self, isPunishment):
65     if isPunishment:
66         self.__reward = -500
67         self.__isPunishment = True
68     else:
69         self.__reward = 0
70         self.__isPunishment = False
71
72 def getQValues(self, action):
73     return self.__QValues[action]
74
75 def setQValues(self, action, QValue):
76     self.__QValues[action] = QValue
77
78 def generateActions(self, policy="mu"):
79     if policy in self.__policies:
80         if self.__stateNo % self.__environmentSize[1] == 0: # column 0
81             self.__validActions.remove("L")
82             self.__QValues.pop("L")
83         if self.__stateNo % self.__environmentSize[1] == (self.__environmentSize[1]-1): # last column
84             self.__validActions.remove("R")
85             self.__QValues.pop("R")
86         if self.__stateNo in range(0, self.__environmentSize[1]): # row 0
87             self.__validActions.remove("U")
88             self.__QValues.pop("U")
89         if self.__stateNo in range((self.__environmentSize[0] - 1) * self.__environmentSize[1],
90                                   self.__environmentSize[0] * self.__environmentSize[1]): # bottom row
91             self.__validActions.remove("D")
92             self.__QValues.pop("D")
93
94 def choseAction(self, epsilon):
95     rand = random.random() # random number between [0,1)
96     if rand <= epsilon:
97         chosenAction = self.explorative()
98     else:
99         chosenAction = self.exploitative()
100     return chosenAction
101
102 def explorative(self):
103     return random.choice(list(self.__QValues))
104
105 def exploitative(self):

```

```
106         return max(self.__QValues.items(), key=operator.itemgetter(1))[0]
107
108     def getMaxQValue(self):
109         return self.__QValues[max(self.__QValues.items(), key=operator.itemgetter(1))[0]]
```

Listing 6: Python - State.py

REFERENCES

- [1] Sutton, R. and Barto, A. (n.d.). Reinforcement learning: an introduction. 2nd ed. pp.1-4, pp.103-106, pp.137-140.
- [2] <https://towardsdatascience.com/the-complete-reinforcement-learning-dictionary-e16230b7d24e> (Date of Access:07/01/2020)