Data 640 Summer 2022

Professor Steve Knode

Tiania Chan

tiania.chan@gmail.com

Assignment 5b: Q Learning using numpy

**Introduction**

The goal of this project is to use Q Learning implemented using Python and Google Colab to create a learning model that can find the most optimal path to transverse a “board” with obstacles given the starting location and end location. The analysis uses code from the article on reinforcement learning by Sayak Paul ([https://blog.floydhub.com/an-introduction-to-q-learning-reinforcement-learning/](about:blank) ) with additional modifications to test use cases.

Code Overview:

Dependencies are libraries are declared at the very top of the code. Since the model will be using the library numpy, this is first declared. Next the learning rate and the discount needed for the Q Learning equation is declared. These will be changed later in the code to see if there is any affect to the model. The next few cells in the variations in the learning rate and discount to see the difference in the results of the model when these values are changed.

The sixth cell is where more dependencies are declared. To start off, the dictionary, location\_to\_state is declared so that the Q Learning equation can be used to calculate the numerical values of each location in the “board” that the model is traversing to find the most optimal path. Since there are 9 locations starting rom L1 to L9 and Python begins its indexes at 0, each value is mapped from 0 to 8. Next, the actions, or what location the model will take after its current location is declared in the list variable, actions. Next, the rewards are declared. This is a numpy array which references the locations L1 through L9 and what actions the model can take. Where the array contains a 0, the model will not take this action as it is discouraged. Where there is a 1, the model is more likely to take that action. Next, the variable state\_to\_location is an array to map the index values to the array for the model to iterate through. The next cell is to allow the model to go from L2 to L5. Since the rewards array did not contain a 1 for the model to go from L2 to L5, it would keep bouncing back between L1 and L2 until the reward array was updated.

The next cell contains the main function that will be called for the model to find the most optimal path through the board given user input for the desired start location and end location. The first part of the method is to copy the rewards array. This lets as many variations of the model to be run without affecting the original rewards array unless manually modified. Next, retrieve the ending state according to the dictionary location\_to\_state to store for use further in the method. This will be necessary to see if the end location has been reached or not. In the next line, by setting the reward to 999 for the end location value, the model will always be striving to reach the end location.

The next part of the function begins the q learning process. Start by declaring a 9 by 9 array of the board. Next, since Q learning uses exploration of the environment, set a high number of iterations that the agent can go through to learn its environment. In this case, it was set to 1000 but this will always depend on the model and the environment, larger and more complex environments will require more possible iterations. Here, the agent will pick a random location using np.random.randint(0,9) to stay within the board. From there, actions taken by the agent is recorded in the variable playable\_actions. In the next for loop, so long as the agent is within range of the board, if the current state is greater than 0, record the action as this is a possible desired action. Next, choose another action for the agent to take.

Next is the Q Learning equation. Mathematically, the equation is as follows:

Text

Description automatically generated with low confidence

Where the agent will look at the reward and add the product of the value of the action and the discount rate. Each time the agent moves further from the desired location, the discount lessens the reward so that the agent is always drawn toward the higher rewards. Next, using the Bellman equation the Q Value is updated. Once all the Q values have been stored for each chain of actions, the agent is finally ready to begin finding the optimal path.

Still in the same function, set the route as a list with the first index value as the start location. Next location is yet unknown so for now, the value is the start location of the agent. Then while the next location is not equal to the end location, the model will find the starting state, find the maximum Q value of the next possible action, take the maximum Q value to determine the next location and append the next location to the list of the route that the agent will take and the start location is updated to the next location. Once the end location has been reached, the model breaks out of the while loop and returns the optimal route and the number of steps taken to reach the end location. The next print statements display the discount value, the learning rate, as well as the results of the optimal route based on starting and end locations. The next few cells are there to show the thought process of identifying why the original agent was able to find the optimal route from L9 to L1 but was stuck if trying to find an optimal route from L1 to L9.

The next few cells contains the alterations in the code necessary for a new model to traverse a “board” containing 10 locations instead of 9 locations. Among the changes were as follows:

* An additional state to location\_to\_state
* An additional action to actions
* Additional row and column to the rewards array
* Initializing a 10 by 10 “board” instead of 9 by 9
* Generating a random number between 0 and 9 to allow the agent to look at 10 locations
* Increasing the range of the for loop from range (9) to range(10)

From there, the results for finding the optimal route between L10 to L1 as well as L10 to L4 are in the final cells.

**Answers to Prompts**

Prompt 3:

In prompt 3, the question is why the optimal route for the original reward table to go form L9 to L1 is L9 -> L8 -> L5 -> L2 -> L1. This is because the optimal route function takes the start and end location, sets the value of the end location to 999 and then finds the Q values for each route possible for the agent or model to take in the “board”. Each iteration then takes the maximum Q value until the agent finds the most optimal path. In the case of L9 to L1, each next location had a value of 1 before reaching the final value of 999 which was the desired goal of the model.

Prompt 5:

In prompt 5, the question is to run the model with different parameters for gamma and alpha or for different values of the discount and learning rate. The first values are .9 for the discount or gamma and .75 for the learning rate or alpha value. Below is the result:

A picture containing text

Description automatically generated

The path from L9 to L2 was through L8 to L5 to L2 before ending at L1. It took 4 steps which is correct.

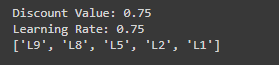
The next variation was to use .5 as both the learning and discount factor. The results are below:

A screenshot of a computer

Description automatically generated with medium confidence

We can see the result is still the same as this is still the optimal path. This was surprising since the discount value was lower, it would make the path with the highest value even more attractive to the model but with a lower learning rate, one would assume that the model would have a harder time finding the correct path.

The next values used was .75 for both the discount factor and learning rate. The hypothesis is since a .5 discount and learning rate still had the correct learning path that .75 should still be correct.



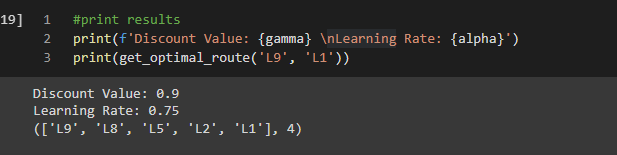
Finally, the last change in value for the learning and discount rate was .3 for both. Since the correct optimal path had been determined by the model with higher values, the hypothesis was that a lower learning rate and discount rate might introduce errors. However, once again the results show a correct path.



This is most likely due to the number of iterations the model took to learn using Q Learning. Since there were 1,000 iterations, it makes sense that the agent would still take the correct path.

Prompt 6:

In prompt 6, the question is to see how many times the while loop runs when running print(get\_optimal\_route(‘L9’,’L1’) and by adding a counter called steps right before the while loop and incrementing the value by 1 at the end of the while loop, it goes through 4 steps which can be seen below

****

Where before the results only displayed the route, the number of steps is now displayed next to the route.

Prompt 7:

In prompt 7, the question is regarding the number of iterations in the for loop for the method get\_optimal\_route(). To start, the given number of iterations is set to 1,000. This is because Q Learning is exploratory and requires enough iterations through the “board” to be able to learn all paths. When dropped to 200 iterations, the model took a little longer to find the correct path but did eventually find the correct path as shown below. Graphical user interface, text

Description automatically generated

This means that the agent was still able to fully explore the board and come up with the most optimal path. Once the number of iterations was dropped to 50 however, the model ran out of memory when trying to find the optimal path from L9 to L1 which means that there were not enough iterations for Q Learning to be effective.

Prompt 8:

In prompt 8, is to try and reverse the path from L9 to L1 and try to find the optimal path from L1 to L9. Commented out in the code were the many approaches taken to debug the code as it didn’t appear that the rewards from L2 to L5 were different from the core code given by Sayak Paul. Among those approaches was to see that rewards\_new was being reset every time the model was called. This is guaranteed by the line rewards\_new = np.copy(rewards) since it creates a “copy” of rewards that can be used as rewards\_new without affecting the value of rewards since rewards\_new is a local variable compared to rewards which is a global variable. We can see this being printed out to ensure that the rewards were reset as well as the end location value changed to 999 instead of 1 or 0. Second, there are print statements testing out each step of what would be the optimal route to go from L1 to L9 by breaking the route down to one location at a time. So L1 to L2 was tested, L2 to L5 was tested, L5 to L8, as well as L8 to L9. Each of these ran correctly and quickly except for L2 to L5 which would hang. By printing the value of next\_location at the end of the while loop, it was determined that the model was going from L1 to L2 to L1 and back to L2 over and over until Google Colab ran out of memory. At that point, the rewards table itself was inspected closely where it turns out the rewards for L2 were different from the rewards that were outline by Sayak Paul. L2 should have been able to travel to L1, L3 and L5. However, in the code, there were only rewards for L1 and L3 but no ability to travel to l5. This was remedied by changing the rewards table to add a 1 in the location for L5 from L2 so that the model could then travel from L2 to L5 successfully.

Text

Description automatically generated with medium confidence

Prompt 9:

Prompt 9 adds another location, L10 to the board. The number of iterations is brought back to 1000 in the for loop as well as the discount value is back to .9 and the learning rate is .75. The changes to the code are listed above in the section explaining the code. This changes the board from this configurationA picture containing text, clock

Description automatically generated to this configuration

Graphical user interface

Description automatically generated

By adding a location L10, the line print(get\_optimal\_route’L10’,’L1’) could run by adding a 1 so that the model can go from L9 to L10 without any of the other locations having that ability. The results are displayed below

Table

Description automatically generated with medium confidence

With the optimal path going from L10 to L9 to L8 to L5 to L2 to L1 which is just one additional step from print(get\_optimal\_route(‘L9’,’L1’).

The other ask in prompt9 is to run print(get\_optimal\_route(‘L10’,’L4’). The results of which are as below

Table

Description automatically generated with medium confidence

Which is expected. Given the layout, above, the optimal route is L10 to L9 to L8 to L7 to L4 which takes 4 steps.

**Conclusion and Take Aways**

The model’s success depends strongly on its ability to be able to explore the environment thoroughly so that it can store and “learn”. This was strongly demonstrated when the iterations of the for loop for get\_optimal\_route for the Q Learning portion was dropped from 1000 iterations down to 200 down to 50. At 200, the model was still successful but at 50, it no longer had the stored knowledge to be able to find the optimal path. So while the learning rate and discount rate are helpful to prioritize what actions are optimal for the agent in its environment, the most important aspect of reinforcement learning is using the Bellman equation and allowing the agent to thoroughly explore the environment. Of course, it is also important to ensure that the environment is set up correctly so that the desired actions can be taken by the agent. This requires a fair amount of thinking on the designer or analyst of the reinforcement learning model to ensure these aspects are well thought out and tested for each use case.

**References**

Dey, S.K.(2018, December 8). The Explore- Exploit Dilemma. Retrieved from <https://medium.com/data-science-for-everyone/the-explore-exploit-dilemma-436cb1edff0d>

Sayak, P. ( 2019, May 15). An Introduction to Q-Learning: Reinforcement Learning Retrieved from <https://blog.floydhub.com/an-introduction-to-q-learning-reinforcement-learning/>

Serrano Academy. (2021, May 24). A Friendly introduction to deep reinforcement learning, Q-Networks and policy Gradients. [Video] Retrieved from <https://www.youtube.com/watch?v=SgC6AZss478&ab_channel=Serrano.Academy>

Torres, J. (2020, June 11). The Bellman Equation: V function and Q function Explained. Retrieved from <https://towardsdatascience.com/the-bellman-equation-59258a0d3fa7>

**Code Appendix**

#source: https://github.com/sayakpaul/FloydHub-Q-Learning-Blog/blob/master/Q-Learning%20using%20numpy.ipynb

# Only numpy

import numpy as np

# Initialize parameters

gamma = 0.9 # Discount factor

alpha = 0.75 # Learning rate

#second run

# Initialize parameters

gamma = 0.5 # Discount factor

alpha = 0.5 # Learning rate

#third run

# Initialize parameters

gamma = 0.75 # Discount factor

alpha = 0.75 # Learning rate

#fourth run

# Initialize parameters

gamma = 0.3 # Discount factor

alpha = 0.3 # Learning rate

#fifth run

# Initialize parameters

gamma = 0.9 # Discount factor

alpha = 0.3 # Learning rate

#fifth run

# Initialize parameters

gamma = 0.9 # Discount factor

alpha = 0.3 # Learning rate

#change rewards for step 8

rewards = np.array([[0,1,0,0,0,0,0,0,0],

              [1,0,1,0,1,0,0,0,0],

              [0,1,0,0,0,1,0,0,0],

              [0,0,0,0,0,0,1,0,0],

              [0,1,0,0,0,0,0,1,0],

              [0,0,1,0,0,0,0,0,0],

              [0,0,0,1,0,0,0,1,0],

              [0,0,0,0,1,0,1,0,1],

              [0,0,0,0,0,0,0,1,0]])

def get\_optimal\_route(start\_location,end\_location):

    # Copy the rewards matrix to new Matrix

    rewards\_new = np.copy(rewards)

    # Get the ending state corresponding to the ending location as given

    ending\_state = location\_to\_state[end\_location]

    # With the above information automatically set the priority of the given ending state to the highest one

    rewards\_new[ending\_state,ending\_state] = 999

    # print(rewards\_new)

    # -----------Q-Learning algorithm-----------

    # Initializing Q-Values

    Q = np.array(np.zeros([9,9]))

    # Q = np.array(np.zeros([10,10])) # for 10 values instead of 9

    # Q-Learning process

    # for i in range(1000):

    # for i in range(200):

    for i in range(50):

        # Pick up a state randomly

        current\_state = np.random.randint(0,9) # Python excludes the upper bound

        # current\_state = np.random.randint(0,10) # Python excludes the upper bound

        # For traversing through the neighbor locations in the maze

        playable\_actions = []

        # Iterate through the new rewards matrix and get the actions > 0

        for j in range(9):

        # for j in range(10):

            if rewards\_new[current\_state,j] > 0:

                playable\_actions.append(j)

        # Pick an action randomly from the list of playable actions leading us to the next state

        next\_state = np.random.choice(playable\_actions)

        # Compute the temporal difference

        # The action here exactly refers to going to the next state

        TD = rewards\_new[current\_state,next\_state] + gamma \* Q[next\_state, np.argmax(Q[next\_state,])] - Q[current\_state,next\_state]

        # Update the Q-Value using the Bellman equation

        Q[current\_state,next\_state] += alpha \* TD

    # print(Q)

    # Initialize the optimal route with the starting location

    route = [start\_location]

    # We do not know about the next location yet, so initialize with the value of starting location

    next\_location = start\_location

    #step counter

    steps = 0

    # We don't know about the exact number of iterations needed to reach to the final location hence while loop will be a good choice for iteratiing

    while(next\_location != end\_location):

        # Fetch the starting state

        starting\_state = location\_to\_state[start\_location]

        # Fetch the highest Q-value pertaining to starting state

        next\_state = np.argmax(Q[starting\_state,])

        # next\_state = np.argmax(Q[2,])

        # print(Q[2])

        # next\_state = np.argmax(Q)

        # print(np.argmax(Q))

        # print(Q[starting\_state,])

        # print(next\_state)

        # We got the index of the next state. But we need the corresponding letter.

        next\_location = state\_to\_location[next\_state]

        # print(next\_location)

        route.append(next\_location)

        # Update the starting location for the next iteration

        start\_location = next\_location

        #increment number of steps

        steps +=1

    return route, steps

#print results

print(f'Discount Value: {gamma} \nLearning Rate: {alpha}')

print(get\_optimal\_route('L9', 'L1'))

#print results -- from l1 to l2

print(f'Discount Value: {gamma} \nLearning Rate: {alpha}')

print(get\_optimal\_route('L1', 'L2'))

#print results -- from l2to l5

print(f'Discount Value: {gamma} \nLearning Rate: {alpha}')

print(get\_optimal\_route('L2', 'L5'))

# print(get\_optimal\_route('L8', 'L5'))

#print results -- from l5 to l8

print(f'Discount Value: {gamma} \nLearning Rate: {alpha}')

print(get\_optimal\_route('L5', 'L8'))

#print results -- from l8 to l9

print(f'Discount Value: {gamma} \nLearning Rate: {alpha}')

print(get\_optimal\_route('L8', 'L9'))

#step 9

# Initialize parameters

gamma = 0.9 # Discount factor

alpha = 0.75 # Learning rate

# Define the states - to include L10

location\_to\_state = {

    'L1' : 0,

    'L2' : 1,

    'L3' : 2,

    'L4' : 3,

    'L5' : 4,

    'L6' : 5,

    'L7' : 6,

    'L8' : 7,

    'L9' : 8,

    'L10' : 9

}

# Define the actions

actions = [0,1,2,3,4,5,6,7,8,9]

# Define the rewards

rewards = np.array([[0,1,0,0,0,0,0,0,0,0], #l1

              [1,0,1,0,0,0,0,0,0,0], #l2

              [0,1,0,0,0,1,0,0,0,0], #l3

              [0,0,0,0,0,0,1,0,0,0], #l4

              [0,1,0,0,0,0,0,1,0,0], #l5

              [0,0,1,0,0,0,0,0,0,0], #l6

              [0,0,0,1,0,0,0,1,0,0], #l7

              [0,0,0,0,1,0,1,0,1,0], #l8

              [0,0,0,0,0,0,0,1,0,1], #l9

              [0,0,0,0,0,0,0,0,1,0]]) #l10

# Maps indices to locations

state\_to\_location = dict((state,location) for location,state in location\_to\_state.items())

def get\_optimal\_route(start\_location,end\_location):

    # Copy the rewards matrix to new Matrix

    rewards\_new = np.copy(rewards)

    # Get the ending state corresponding to the ending location as given

    ending\_state = location\_to\_state[end\_location]

    # With the above information automatically set the priority of the given ending state to the highest one

    rewards\_new[ending\_state,ending\_state] = 999

    print(rewards\_new)

    # -----------Q-Learning algorithm-----------

    # Initializing Q-Values

    # Q = np.array(np.zeros([9,9]))

    Q = np.array(np.zeros([10,10])) # for 10 values instead of 9

    # Q-Learning process

    for i in range(1000):

    # for i in range(200):

    # for i in range(50):

        # Pick up a state randomly

        # current\_state = np.random.randint(0,9) # Python excludes the upper bound

        current\_state = np.random.randint(0,10) # Python excludes the upper bound

        # For traversing through the neighbor locations in the maze

        playable\_actions = []

        # Iterate through the new rewards matrix and get the actions > 0

        # for j in range(9):

        for j in range(10):

            if rewards\_new[current\_state,j] > 0:

                playable\_actions.append(j)

        # Pick an action randomly from the list of playable actions leading us to the next state

        next\_state = np.random.choice(playable\_actions)

        # Compute the temporal difference

        # The action here exactly refers to going to the next state

        TD = rewards\_new[current\_state,next\_state] + gamma \* Q[next\_state, np.argmax(Q[next\_state,])] - Q[current\_state,next\_state]

        # Update the Q-Value using the Bellman equation

        Q[current\_state,next\_state] += alpha \* TD

    # Initialize the optimal route with the starting location

    route = [start\_location]

    # We do not know about the next location yet, so initialize with the value of starting location

    next\_location = start\_location

    #step counter

    steps = 0

    # We don't know about the exact number of iterations needed to reach to the final location hence while loop will be a good choice for iteratiing

    while(next\_location != end\_location):

        # Fetch the starting state

        starting\_state = location\_to\_state[start\_location]

        # Fetch the highest Q-value pertaining to starting state

        next\_state = np.argmax(Q[starting\_state,])

        # We got the index of the next state. But we need the corresponding letter.

        next\_location = state\_to\_location[next\_state]

        route.append(next\_location)

        # Update the starting location for the next iteration

        start\_location = next\_location

        #increment number of steps

        steps +=1

    return route, steps

#print results  -- default to make sure changes all work

print(f'Discount Value: {gamma} \nLearning Rate: {alpha}')

print(get\_optimal\_route('L9', 'L1'))

#print results

print(f'Discount Value: {gamma} \nLearning Rate: {alpha}')

print(get\_optimal\_route('L10', 'L1'))

#print results

print(f'Discount Value: {gamma} \nLearning Rate: {alpha}')

print(get\_optimal\_route('L10', 'L4'))

References

Rudes, R. ( 2020, Sept 8). An Introductory Reinforcment Learning Project: Learning Tic-Tac-Toe via Self-Play Tabular Q Learning. Retrieved from https://towardsdatascience.com/an-introductory-reinforcement-learning-project-learning-tic-tac-toe-via-self-play-tabular-b8b845e18fe