Reinforcement Learning – Grid World Game

Athulya Shaji

MSc Computer Science

School of Computing, Engineering and Built-Environment

Ulster University, Belfast
Northern Ireland

Shaji-A@ulster.ac.uk

***Abstract* – This paper is an illustration of Reinforcement Learning (RL). The work describes an agent's deterministic approach and the efficient exploration and exploitation of the environment to reach a win state.**

I. INTRODUCTION

Machine Learning (ML), comprises of (i) Supervised Learning, (ii) Unsupervised Learning, and (iii) Reinforcement Learning (RL). RL is a simple algorithm in which an intelligent agent takes an action in order to live in a hostile environment while collecting potential positive rewards. It focuses on applying either Deterministic Policy or Stochastic Policy to achieve a balanced exploration and utilization of the environment. In a deterministic method, the agent takes a predetermined action to achieve a predetermined state. In the stochastic approach, on the other hand, the agents generally pick their actions based on a set of probabilities. The agent is taught to take a deterministic approach to the environment in this work.

Grid World Game is the best challenge to encounter while attempting to study reinforcement learning. It is the most fundamental and classic problem in reinforcement learning, and implementing it is the greatest way to grasp the fundamentals of reinforcement learning. Meanwhile, creating your own game and seeing how a robot learns on its own is fascinating!

The agent begins in one state and moves through a variety of alternative movements until it finds the best path to its win state. For each action, a reward is given to the agent. Once the agent reaches the win state, the rewards collected in the entire game are calculated known as the cumulative reward. Similarly, the agent is trained by making it play several times and thereby creating a Q – table. Therefore, the agent gets trained to reach the win state in the least possible time steps using a greedy approach.

II. REINFORCEMENT LEARNING

"Reinforcement learning (RL) is a field of machine learning concerned with how intelligent agents should perform behaviors in an environment in order to maximize the notion of cumulative reward," as stated in the introduction – I It does not require a labeled display of input and output pairs. Because the environment involves computer programming and mathematical optimization approaches, i.e., dynamic programming, Markov Decision Process (MDP) is commonly used in the RL technique. As a result, RL is often referred to as neuro-dynamic programming.

The following points should be highlighted, in MDP models the reinforcement learning algorithm.

• The letter 'S' stands for the agent states and settings.

• The letter 'A' stands for the agent's set of acts.

• $P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ indicates the chance of a transition occurring at time 't'.

• $R_a(s, s')$ after the transition from s to s', signifies the interim reward.

An agent, in essence, moves around the environment in discrete time steps. If t is the time taken, st is the agent's current state, and lambda-t is the payoff, then When you choose an action a1, such as going north, south, east, or west, the movement is mirrored in the environment.

Exploration: "In Burnetas and Katehakis (1997), the exploration vs. exploitation trade-off was most completely examined using the multi-armed bandit issue and for limited state space MDPs." Reinforcement learning entails advanced exploration strategies, such as selecting actions at random without reference to an estimated probability distribution, which produces poor outcomes. We decide that the most feasible strategy is simple exploration due to a lack of appropriate algorithms that match the quantity of states. One of the strategies mentioned is -greedy. The agent makes a decision based on the chance of exploitation, which is $0 < \varepsilon < 1$ in this case. Random exploration and exploitation is another possible movement. The agent does not need or seek a probability for the next action in this case, and instead traverses according to its wishes.

Major algorithms of Reinforcement Learning are;

1. Criterion of Optimality
2. Value Function
3. Brute Force Algorithm
4. Model-based Algorithms
5. Monte Carlo Methods
6. Temporal Difference Methods
7. Direct Policy Search

III. Q-LEARNING

Q-Learning is a model-free reinforcement learning technique that is used to monitor and study the values of action in a specific state. Because Q-Learning was not included in any of the RL models, it is classified as a model-free algorithm. For navigating and getting awards, it employs a scholastic policy. The agent can identify a suitable policy that will result in generating a greater expected reward, i.e. reaching the final state with fewer steps, using Q-Learning for any defined MDP. The algorithm that evaluates each action for achieving a specific reward is referred to as "Q" in Q-Learning.

The following are some of the characteristics and factors that fall under the category of Q-Learning:

A. Learning Rate:

It entails determining what everything can be done for the first time, which is a best-case scenario from the preceding processes.

B. Discount Factor:

The discount factor is used to calculate all of the significant future incentives for an episode. The symbol "γ" is used to represent it.

C. Initial Conditions:

As Q-Learning is an iterative algorithm, an initial condition or reward must be provided. The letter Q is used to specify a certain value.

The state position along the Y-axis (vertical cells) of the table is formed, and the action along the X-axis (horizontal cells) of the table is generated. The q-values are set to 0 by default and are updated to specific values in each associated cell when specified actions occur. The prize is then determined using the equation below:

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[V(S_{t+1}) - V(S_t)\Big]$$

IV. MARKOV DECISION PROCESS (MDP)

The Markov Decision Process is a discrete-time stochastic process that is used in framework modeling to make decisions. An agent is generated and set to engage with an environment using this method. The agent then travels through the environment through all ways conceivable in order to acquire a positive or negative reward.

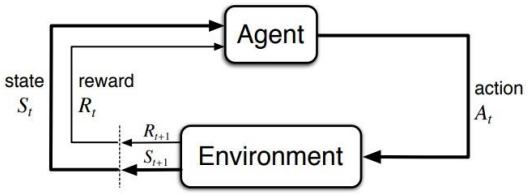

Fig. (i): Agent-Environment Interaction in MDP

A. Agent:

The model for an environment that we are constructing is called an agent. An agent is taught how to perform a specific task, such as traversing around the environment in all conceivable directions, such as north, south, east, or west.

B. Environment:

An environment is a context in which the agent takes or decides on a course of action. The 5x5 grid is the environment in the example of the grid world game. Each position in an environment comes with its own set of benefits.

C. State:

The current state of an agent is referred to as its state. A time-step is another name for it. Every action taken by an agent results in a state change.

D. Return:

Each action done by an agent is rewarded in some way. This might be either a good thing or a bad thing. The primary goal of the RL approach is to teach the agent to obtain the highest potential reward in each episode. An episode is a complete cycle in which the agent achieves the win state. The total reward collected by an agent in a given episode is referred to as the return.

E. Policy:

(i) Deterministic Policy: To get to a specific state, the agent usually takes a predetermined action.

(ii) Stochastic Policy: The agents usually pick their actions based on a set of possibilities.

F. Value:

(i) State Value: It is the expected result that is returned when certain actions are taken in accordance with a set of rules.

(ii) State-Action Value: It is the reward that is expected when a given action is taken from the

current condition. Q-Value is another name for it.

With respect to an agent who chooses to move, actions and states can be anything. A time-step is the name given to each action taken by an agent in the environment. As a result, regardless of the goal of winning the game, the agent can move wherever in the environment. This will cause the agent to wander about the area earning negative rewards.

The grid world is set to contain 100 episodes in this example, which means the agent will play 100 rounds of the game. A total of 50 time-steps can be performed by each episode irrespective of win. This will limit the agent's unrestricted mobility. Furthermore, the episodes of play will be completed if the average cumulative reward of all the games is larger than 10 in 30 consecutive episodes.

V. PROBLEM SCENARIO

In this project, we have to create a grid with five columns and five rows, as well as four-sided borders. In fig. (ii), there is an agent illustrated as a red circle whose initial state is (2, 1). The agent can take four different actions: north, south, east, and west. The agent has to reach the win state (5,5).

A negative one reward is applied to each action, i.e., a -1 reward is added to each movement. However, in state (2, 4) there is a particular jump that places the agent in (4, 4). In addition, there is a unique prize of +5 for this jump. The agent cannot navigate past four obstructed cells. Those are (3, 3), (3, 4), (3, 5), and (4, 3). Once reaching the win state (5,5), the agent will receive a +10 reward.

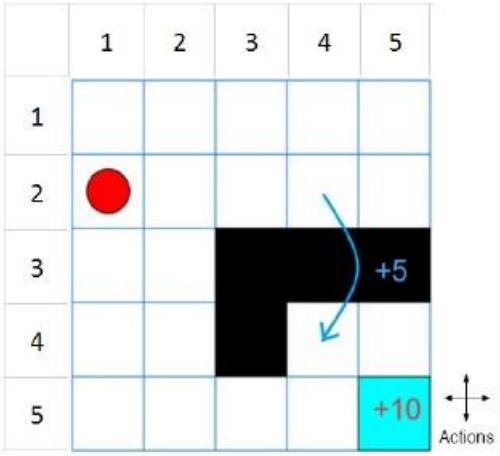

Fig. (ii): Grid World Environment

VI. ARCHITECTURE

In the Grid World, the winning state is (5, 5). As a result, the agent must travel from its initial state to the win state while collecting the most rewards. The agent has been taught to move in four different ways. The agent tries to travel in all conceivable directions in order to attain the win state through reinforcement learning. As previously stated, each movement results in a -1 reward, with the exception of the jump state, which is from (2, 4) to (4, 4) and credits a +5 reward. When it reaches the win state, the cumulative awards earned are multiplied by 10.

The problem is set to follow a deterministic policy, which means it will follow the path it chooses on its own. As a result, the agent must be trained by stating relevant functions in the code. As a result, the reinforcement learning algorithm has been completed.

The agent is played 100 times for a higher chance of getting the best payout. Because the agent takes a deterministic approach, it will travel across the grid, giving it the best chance of winning. This will almost certainly lead to negative cumulative rewards. As a result, each episode's time-steps are limited to 50. As a result, the agent only has 50 options to reach the win state Furthermore, after the agent has discovered the best reward path, it continues to follow the same procedures. As a result, if the agent earns an average cumulative reward greater than 10 in 30 consecutive episodes, the game will be reset to quit.

Fig. (iii)demonstrates one of the best traverses the agent can make, which contains six time-steps.

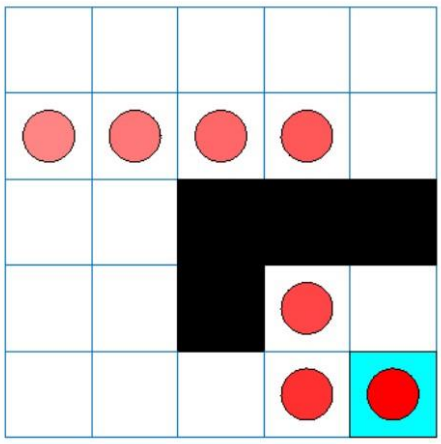

Fig. (iii): Optimum Action of the agent.

VII. IMPLEMENTATION

The number of rows and columns on the grid board must first be determined. The beginning state, win state, leap

state, and other global variables are also defined. The implementation is done using python programming. The code, in this paper, is implemented in PyCharm IDE and also uses NumPy as a main module.

Because the agent will be trained using Deterministic policy, the Deterministic factor is set to "True." Because most languages use C standards, Python indexing starts at 0 as well, resulting in a change in program coordinates.

```
# global variables
BOARD_ROWS = 5
BOARD_COLS = 5
WIN_STATE = (4, 4)
START = (1, 0)
DETERMINISTIC = True
```

The program consists of two classes. One for the agent and one for the state. Each class contain several functions and variables

The State class contains four functions:

1. Give Reward: which sets the reward as 10 if the state is win else returns -1

```
def giveReward(self):
    if self.state == WIN_STATE:
        return 10
    else:
        return -1
```

2. Is End: this function changes the value of the variable 'IsEnd' to true which is initially set as false.

```
def isEndFunc(self):
    if (self.state == WIN_STATE):
        self.isEnd = True
```

3. Next Position: this is a function that finds out the next position of the agent according to the actions taken. It checks for the possibility of the action considering the block states. This function is also responsible for the jump from (1,3) to (3,3).

```
def nxtPosition(self, action):
    """
    action: "north", "south", "west", "east"
    ------------
    0 | 1 | 2| 3| 4
    1 |
    2 |
    3 |
    4 |
    return next position
    """
    if self.determine:
        if action == "north":
            nxtState = (self.state[0] - 1, self.state[1])
        elif action == "south":
            nxtState = (self.state[0] + 1, self.state[1])
        elif action == "west":
            nxtState = (self.state[0], self.state[1] - 1)
        else:
            nxtState = (self.state[0], self.state[1] + 1)
        # if next state legal
        if self.state == (1,3):
            nxtState = (3,3)
        if (nxtState[0] >= 0) and (nxtState[0] <= (BOARD_ROWS -1)):
            if (nxtState[1] >= 0) and (nxtState[1] <= (BOARD_COLS -1)):
                if nxtState != (3,2) and nxtState != (2,2) and nxtState != (2,3) and nxtState != (2,4):
                    return nxtState
        return self.state
```

4. Show Board: this function displays the grid on which the agent has to play.

```
def showBoard(self):
    self.board[self.state] = 1
    self.board[2,2] = -1
    self.board[2,3]= -1
    self.board[2,4] = -1
    self.board[3,2]= -1
    self.board[4,4] = 1
    for i in range(0, BOARD_ROWS):
        print('-----------------')
        out = '| '
        for j in range(0, BOARD_COLS):
            if self.board[i, j] == 1:
                token = '*'
            if self.board[i, j] == -1:
                token = 'z'
            if self.board[i, j] == 0:
                token = '0'
            out += token + ' | '
        print(out)
    print('-----------------')
```

The Agent class is where the learning rate, epsilon rate and discount factor is initialized.

```
self.lr = 0.2
self.exp_rate = 0.3
self.discount = 0.5
```

Also, the predefined values of the state action pair is also initialized here.

```
# initial state reward
self.state_values = {}
for i in range(BOARD_ROWS):
    for j in range(BOARD_COLS):
        if i== 1 and j == 3:
            self.state_values[(i, j)] = 5
        else:
            self.state_values[(i, j)] = -1
```

The Agent class contains five functions:

1. Choose Action: this function determines the next action that has to be taken by the agent. This can be determined by a random method using the epsilon rate (exploration approach) or by checking for the maximum reward (greedy approach – exploitation)

```
def chooseAction(self):
    # choose action with most expected value
    mx_nxt_reward = 0
    action = ""

    if np.random.uniform(0, 1) <= self.exp_rate:
        action = np.random.choice(self.actions)
    else:
        # greedy action
        for a in self.actions:
            # if the action is deterministic
            nxt_reward = self.state_values[self.State.nxtPosition(a)]
            if nxt_reward >= mx_nxt_reward:
                action = a
                mx_nxt_reward = nxt_reward
    return action
```

2. Take Action: the function reset the current state of the agent depending upon the action and next position function

```
def takeAction(self, action):
    position = self.State.nxtPosition(action)
    return State(state=position)
```

3. Reset: this function reinitialize the list which contains all the states the agent has traversed through in an episode, after every episode.

```
def reset(self):
    self.states = []
    self.State = State()
```

4. Play: this is the most important function in this program, from where all the other functions are called.

Once the agent reaches the win state upon choosing action and changing position as described in the previous functions. The rewards are calculated from the end (i.e. the win) state to the initial state.

$$V(S_t) \leftarrow V(S_t) + \alpha\left[V(S_{t+1}) - V(S_t)\right]$$

The cumulative reward is also calculated in each episode.

$$G_t = R_{t+1} + \gamma R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

One episode is determined by either end state of 50-time step. The game ends either after 100 episodes or when the average cumulative reward is greater than 10 over 30 consecutive episodes.

```
def play(self, rounds=10):
    i = 0
    ep = 0
    q_rewards = []
    print(rounds)
    while i < rounds:
        # to the end of game back propagate reward
        if self.State.isEnd or len(self.states) > 50:
            # back propagate
            reward = self.State.giveReward()
            # explicitly assign end state to reward values
            self.state_values[self.State.state] = reward  # this is optional
            print("Game End Reward", reward)
            print(self.states)
            self.states.reverse()
            # assigning rewards to each state
            for s in self.states:
                reward = self.state_values[s] + self.lr * (reward - self.state_values[
                self.state_values[s] = round(reward, 3)
            self.states.reverse()
            q_reward = 0
            # calculating the cumulative reward of the current episode
            for j in range(0,len(self.states)):
                r =  self.state_values[self.states[j]]
                q_reward = q_reward + (self.discount**j)*(r)
            # storing the cumulative reward of all the episodes in a lis
            q_rewards.append(q_reward)
            # stop training if average cumulative reward is greater than 10 over 30 cc
            if np.mean(q_rewards)> 10:
                ep = ep+1
                if ep > 30:
                    break
            self.reset()
            i += 1
        else:
            action = self.chooseAction()
            # append trace
            self.states.append(self.State.nxtPosition(action))
            print("current position {} action {}".format(self.State.state, action))
            # by taking the action, it reaches the next state
            self.State = self.takeAction(action)
            # mark is end
            self.State.isEndFunc()
            print("nxt state", self.State.state)
            print("---------------------")
```

5. Show Value: this function shows the Q – value table after the game is over

```
def showValues(self):
    for i in range(0, BOARD_ROWS):
        print('-----------------------------------')
        out = '| '
        for j in range(0, BOARD_COLS):
            out += str(self.state_values[(i, j)]).ljust(6) + ' | '
        print(out)
    print('-----------------------------------')
```

Q table using the state and action specifications from the grid world environment and the states traveled by the agent, by setting the learning rate to be 1

```
Game End Reward 10
[(1, 1), (1, 2), (1, 3), (3, 3), (3, 4), (3, 4), (3, 4), (3, 4), (3, 4), (3, 4), (3, 4), (3, 4), (4, 4)]
-------------------------------
| -1    | -1    | -1    | -1    | -1    |
-------------------------------
| -1    | 10    | 10    | 10    | -1    |
-------------------------------
| -1    | -1    | -1    | -1    | -1    |
-------------------------------
| -1    | -1    | -1    | 10    | 10    |
-------------------------------
| -1    | -1    | -1    | -1    | 10    |
-------------------------------
```

Q-learning agent using this table representation after configuring the epsilon-greedy exploration.

```
-----------------------------------
| 5.756  | 7.435  | 6.456  | 3.651  | -1     |
-----------------------------------
| 7.656  | 8.818  | 9.341  | 9.883  | -1     |
-----------------------------------
| 5.257  | 7.878  | -1     | -1     | -1     |
-----------------------------------
| -0.988 | -0.962 | -1     | 9.934  | 9.988  |
-----------------------------------
| -1     | -0.912 | 1.163  | 7.874  | 10.0   |
-----------------------------------
```

VIII. Conclusion

Reinforcement Learning tackles the problem of learning control methods for autonomous agents with little or no data. "After each act, the algorithm (agent) examines the current situation (state), takes action, and receives feedback (reward) from the environment. Positive feedback is a reward (in the sense that we understand it), while negative feedback is a penalty for making a mistake."

This paper focuses on the setup and execution of a Grid World environment in which the agent goes around seeking rewards. The code is written in Python 3.10 with the PyCharm IDE.

REFERENCES

[1] Expert. ai Team. (2020, May 5). What is machine learning? A definition. Expert.Ai. https://www.expert.ai/blog/machine-learning definition/

[2] Doshi, K. (2020, November 28). Reinforcement learning explained visually (part 4): Q learning, step-by-step. Towards Data Science. https://towardsdatascience.com/reinforcement-learning-explained visually-part-4-q-learning-step-by-step-b65efb731d3e

[3] Zhang, J. (2019, May 4). Reinforcement learning — implement grid world. Towards Data Science. https://towardsdatascience.com/reinforcement-learning-implement grid-world-from-scratch-c5963765ebff

[4] Blackboard. (n.d.).– blackboard learn. Ulster.Ac.Uk. Retrieved 4 March 2022, from https://learning.ulster.ac.uk/webapps/blackboard/execute/content/file? cmd=view&content_id=_6456717_1&course_id=_326147_1 [5] Reinforcement learning explained in 90 seconds. (2021, April 27). [6] Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). Deep reinforcement learning: A brief survey. IEEE Signal Processing Magazine, 34(6), 26–38. https://doi.org/10.1109/msp.2017.2743240

[7] Naeem, M., Rizvi, S. T. H., & Coronato, A. (undefined 2020). A gentle introduction to reinforcement learning and its application in different fields. IEEE Access: Practical Innovations, Open Solutions, 8, 209320–209344. https://doi.org/10.1109/access.2020.3038605

[8] Wikipedia contributors. (2022, February 9). Reinforcement learning. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Reinforcement_learning& oldid=1070771218

[9] Chouhan, S. S., & Niyogi, R. (2012). Multi-agent Planning in Grid World Domain. 2012 Second International Conference on Advanced Computing & Communication Technologies, 117–122.