PROJECT 4
REINFORCEMENT LEARNING

Thana Shree Jeevanandam
University of Buffalo
thanashr@buffalo.edu

Abstract

Reinforcement Learning, a semi-supervised learning model in Machine Learning allows an agent to take actions in response to the current state of an environment to maximize their rewards. The agent learns from a mathematical framework typically modeled as a Markov decision process (MDP). The task here is to build a reinforcement learning agent to navigate the classic 4x4 grid-world environment. The agent learns an optimal policy through Q-Learning which will allow it to decide on how good an action is and helps it to take actions to reach a goal while avoiding obstacles.

1. INTRODUCTION

With the working environment and framework for the learning agent built with OpenAI Gym environments, Q-Learning is implemented. The tabular Q-Learning approach utilizes a table of Q-values as the agent's policy to decide on the next step action. The template given in Python Jupyter Notebook is modified with the following three tasks,

- Code for policy
- Updation of Q-Table
- Training of process.

2. MARKOV DECISION PROCESS (MDP)

Markov Decision Process (MDP) is used to formalize the sequential decisions. Components of an MDP are,

- Agent
- Environment
- State
- Action
- Reward

Throughout this process, it is the agent's goal to maximize the total amount of rewards that it receives from taking actions in given states. This means that the agent wants to maximize not just the immediate reward, but the *cumulative* rewards it receives over time.

The steps in MDP (from Fig.2.1) include,

- At time t t, environment has State StSt.
- Agents observes and performs action AtAt.

- Environment transitions to state S(t+1)S(t+1) and grants a reward , R(t+1)R(t+1).
- The loop iterates to next step , t+1 t+1.

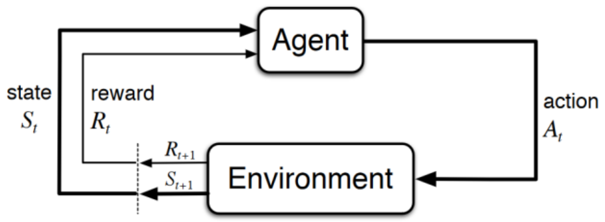


Fig2.1 Markov Decision Process (MDP) Steps

3. DISCOUNTED RETURN

The agent considers the rewards it expects to receive in the future, the more immediate rewards have more influence when it comes to the agent deciding about taking an action. The discounted return makes it to where our agent will care more about the immediate reward over future rewards since future rewards will be more heavily discounted.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma 2 R_{t+3} + \dots$$

where γ – Discount Rate

4. ENVIRONMENT

The 4x4 grid environment is chosen among the other environments such as, physical simulations, video games, stock market simulations, etc. We use the OpenAI gym's design for this environment. In this environment, the agent (green square) has to reach the goal (yellow square) in the least amount of time steps possible.

The environment's state space is described as $n \times n$ matrix with real values on the interval [0, 1] to designate different features and their positions. The agent works within an action space consisting of four actions: up, down, left, right. At each time step, the agent takes one action and moves in the direction described by the action. The agent will receive a reward of +1 for moving closer to the goal and −1 for moving away or remaining the same distance from the goal.

5. Q-LEARNING

Q-Learning solves for the optimal policy in an MDP. The objective of Q-learning is to find a policy that is optimal, expected value of the total reward over all successive steps is the maximum achievable. In precise, the goal of Q-learning is to find the optimal policy by learning the optimal Q-values for each state-action pair.

Q-Function:

The agent can't control what state he ends up in, it depends on the actions taken in the current state. Q-function for a given policy accepts a state and an action and returns the expected return from taking the given action in the given state and following the given policy thereafter. The Q function is calculated using the below formula.

$$Q_{t+1}(s_t, a_t) = Q_{t+1}(s_t, a_t) + \alpha(r_{t+1} + \gamma max Q_t(s_{t+1}, a) - Q_t(s_t, a_t))$$

The above equation is called Bellman equation, which tells the maximum award the agent receives when it enters a state.
The learning rate (α) is decayed every episode and can also be considered as one. This is because, as the agent learns more and more, it believes there is not more left to learn in the environment. So the modified equation is as below

$$Q_{t+1}(s_t, a_t) = r_t + \gamma max Q_t(s_{t+1}, a)$$

A Q table is constructed to keep track of the Q values calculated which is usually a **no. of states X no. of actions** array.

Q-Table:

Q-table, to store the Q-values for each state-action pair. The horizontal axis of the table represents the actions, and the vertical axis represents the states. So, the dimensions of the table are the number of actions by the number of states.

The learning process takes place at each episode.

Q-Table initialization:

At the start of the game, the agent has no agent has no idea how good any given action is from any given state. It's not aware of anything besides the current state of the environment. In other words, it doesn't know from the start whether navigating left, right, up or down will result in a positive reward or negative reward. Therefore, the Q-values for each state-action pair will all be **initialized to zero** since the agent knows nothing about the environment at the start. Throughout the game, the O-values will be iteratively updated using value iteration.

Exploration and Exploitation:

Exploration is the act of exploring the environment to find out information about it. Exploitation is the act of exploiting the information that is already known about the environment to maximize the return.

Epsilon Greedy Strategy:

Epsilon Greedy Strategy gives the agent a way to choose between Exploration and Exploitation. Exploration Rate, ϵ is initially set to 1. With 1, it has the 100% certainity that the agent would start exploring. As the agent starts to learn about the environment, a decay occurs in the ϵ where the likelihood of exploration reduces. The agent besomec greedy with exploiting the environment as it gets the opportunity to explore and learn more.

Policy:

A policy, π is better than or the same as the policy, π' if the expected return of π is greater than or equal to the expected return of π' for all states. A policy better than or the same as other policies is an optimal policy.

A random number is generated between 0 and 1. If the number is greater than epsilon, agent chooses exploitation (action with highest Q-value). Otherwise, it chooses exploration. (exploring random actions and their effects).

```
observation = observation.astype(int)
if np.random.random() <= self.epsilon:
    return self.env.action_space.sample()
```

```
        else:
                return np.argmax(self.q_table[tuple(observation)])
```

Q-Table Updation:

With the Q_function mentioned above, the Q-values are updated. Q_value converges to the optimal O-value by iteratively comparing the loss between the Q-value and optimal Q-value for the given state-action pair and updating Q-value each time to reduce loss. Q-value is updated using learning rate.

$$q_*(s,a)=E[R_{t+1}+\gamma \max_{a'} q_*(s',a')]$$

```
self.q_table[state[0]][state[1]][action]=(1-self.lr)
*self.q_table[state[0]][state[1]][action]
+self.lr*(reward+self.gamma*np.max(self.q_table[next_state[0][next_s
tate[1]][action]))
```

Training of process:

Each episode is built of steps. For each episode, the environment state is changed to the starting state.

```
for episode in range(num_episodes):
    state = env.reset()
    done = False
    rewards_current_episode = 0
```

done – keeps track of if the episode is completed.

rewards_current_episode – cumulative rewards of each step.

Each step has the following loop:

```
for step in range(max_steps_per_episode):

    # Exploration-exploitation trade-off
    exploration_rate_threshold = random.uniform(0, 1)
    if exploration_rate_threshold > exploration_rate:
        action = np.argmax(q_table[state,:])
    else:
        action = env.action_space.sample()
```

exploration_rate_threshold – value between 0 and 1.

If the number is greater than epsilon, agent chooses exploitation (action with highest Q-value). Otherwise, it chooses exploration. (exploring random actions and their effects).

The exploration rate is decayed iteratively.

```
exploration_rate = min_exploration_rate + \ (max_exploration_rate
- min_exploration_rate) * np.exp(-exploration_decay_rate*episode)
```

6. PARAMETERS

The parameters like λ, s (max and min), λ and the number of episodes is used in the project.

Episodes: For episodes the initial value provided was 10000. However, the agent could reach the target after completing around 7000 episodes, by taking approximately 0.24 sec. for each episode.

Exploration factor s: This parameter allows the environment to explore a lot in the agent. Irrespective of the min and max s values, as the agent starts with '0' steps, the $S = 0$, which leads to maximum $s = 1$ in the initial state. As the agent keeps learning the value exponentially reduces. The speed of decay of the s is determined by λ parameter.

7. CHALLENGES

The epsilon should be decreasing for every episode, so that the exploration is done, and exploitation is increased. As the agent learns more and more, it can get to the target faster.

The different values were given to the min_epsilon and max_epsilon and the variations were observed. It was a challenge to get a non-increasing Epsilon vs Episode graph and to get a highest point for reward in the last episode. Thus, the values were changed and observed to get the optimal path.

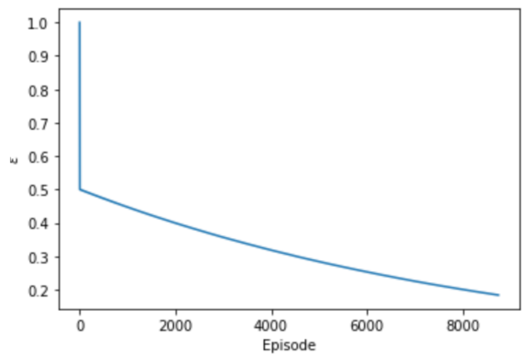


Fig 7.1 Episode vs Epsilon Graph

For Min_epsilon = 0.5; Max_epsilon =1

8. RESULTS

The results are obtained by changing the hyperparameters, max_epsilon and min_epsilon.

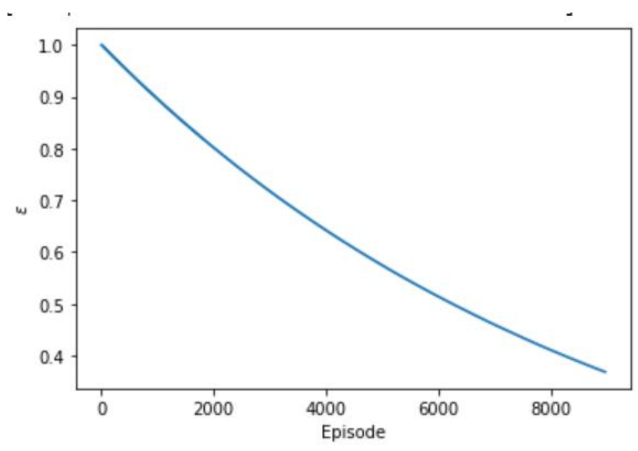


Fig 8.1 Episode vs Epsilon Graph

For Min_epsilon = 0; Max_epsilon =1

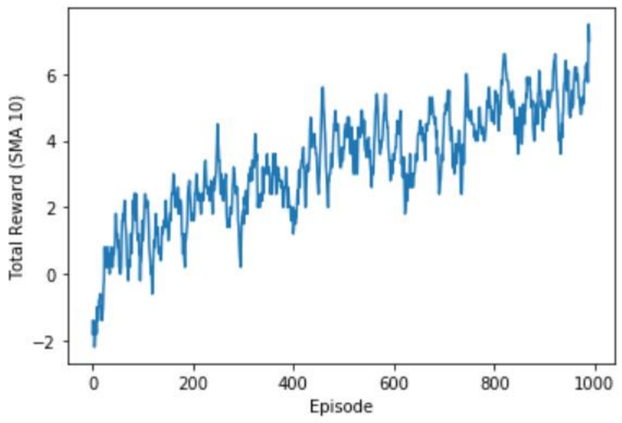
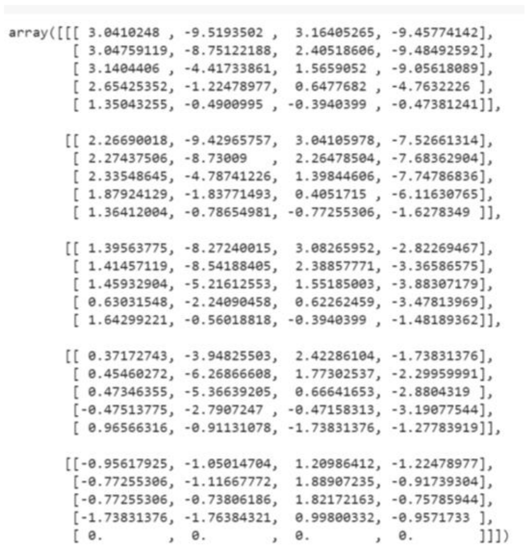


Fig 8.2 Episode vs Reward Graph

For Min_epsilon – 0; Max_epsilon -1



```
array([[[ 3.0410248 , -9.5193502 ,  3.16405265, -9.45774142],
        [ 3.04759119, -8.75122188,  2.40518606, -9.48492592],
        [ 3.1404406 , -4.41733861,  1.5659052 , -9.05618089],
        [ 2.65425352, -1.22478977,  0.6477682 , -4.7632226 ],
        [ 1.35043255, -0.4900995 , -0.3940399 , -0.47381241]],

       [[ 2.26690018, -9.42965757,  3.04105978, -7.52661314],
        [ 2.27437506, -8.73009   ,  2.26478504, -7.68362904],
        [ 2.33548645, -4.78741226,  1.39844606, -7.74786836],
        [ 1.87924129, -1.83771493,  0.4051715 , -6.11630765],
        [ 1.36412004, -0.78654981, -0.77255306, -1.6278349 ]],

       [[ 1.39563775, -8.27240015,  3.08265952, -2.82269467],
        [ 1.41457119, -8.54188405,  2.38857771, -3.36586575],
        [ 1.45932904, -5.21612553,  1.55185003, -3.88307179],
        [ 0.63031548, -2.24090458,  0.62262459, -3.47813969],
        [ 1.64299221, -0.56018818, -0.3940399 , -1.48189362]],

       [[ 0.37172743, -3.94825503,  2.42286104, -1.73831376],
        [ 0.45460272, -6.26866608,  1.77302537, -2.29959991],
        [ 0.47346355, -5.36639205,  0.66641653, -2.8804319 ],
        [-0.47513775, -2.7907247 , -0.47158313, -3.19077544],
        [ 0.96566316, -0.91131078, -1.73831376, -1.27783919]],

       [[-0.95617925, -1.05014704,  1.20986412, -1.22478977],
        [-0.77255306, -1.11667772,  1.88907235, -0.91739304],
        [-0.77255306, -0.73806186,  1.82172163, -0.75785944],
        [-1.73831376, -1.76384321,  0.99800332, -0.9571733 ],
        [ 0.        ,  0.        ,  0.        ,  0.        ]]])
```

Fig 7.3 Q-Table

9. CONCLUSION

Reinforcement learning was performed on the agent in the 4x4 environment to make it reach the target. The agent was trained for 1000 episodes and its increasing reward was observed. The agent was able to meet the target at the end.

10. REFERENCES

[1] https://en.wikipedia.org/wiki/Reinforcement_learning

[2] https://en.wikipedia.org/wiki/Q-learning

[3] https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/