Reinforcement Q-Learning with OpenAI Gym

Bhakti Jadhav
Department of Computer Science
University at Buffalo
Buffalo NY, 14214
bhaktiha@buffalo.edu

Abstract

In this project, reinforcement learning agent is built to navigate the classic 4x4 grid-world environment. Unlike supervised and unsupervised learning, reinforcement learning does not just experience a fixed data set. Reinforcement learning algorithms interacts with an environment. Q-learning generates data exclusively from experience, without incorporation of the prior knowledge. For this we constructed an optimal policy which tells agent what to do and when. The Q in Q-learning stands for quality. Quality represents how useful a given action is in gaining some future reward. The agent learned the policy through Q-Learning which allowed it to take actions to reach a goal while avoiding obstacles. Outcome shows that the environment and agent are built to be compatible with OpenAI Gym environments and will run effectively on computationally-limited machines.

1 Introduction

The objective of this project is to train the agent to reach the goal in the least amount of time steps possible. Reinforcement learning is the problem of getting an agent to act in the world so as to maximize its rewards. Reinforcement learning lies between the spectrum of Supervised and Unsupervised Learning. The goal here is not to look for quick immediate rewards, but instead to optimize for maximum rewards over the whole training. The reward agent depends on overall history of states and does not just depend in current state.

Agent (algorithm) interacts with its environment. Agent exists in environment with set of states S.

1. Agent can perform any of a set of actions A – Performing action $\mathbf{A_t}$ in state $\mathbf{S_t}$ receives reward $\mathbf{R_t}$.
2. Agent's task is to learn control policy π : S → A. That maximizes expected sum of rewards with future rewards discounted exponentially.

$$\sum_{t=0}^{T} \gamma^t R(s_t, a_t, s_{t+1})$$

   where γ ∈ [0, 1] is a discounting factor (used to give more weight to more immediate rewards), $\mathbf{S_t}$ is the state at time step t, at is the action the agent took at time step t, and $\mathbf{S_{t+1}}$ is the state which the environment transitioned to after the agent took the action.

3. This is typically modeled as a **Markov decision process (MDP)**.

Reinforcement Learning Terminology :

- Action (A): possible moves that agent can take.
- State (S): Current situation returned by environment.
- Reward (R): Immediate return sent back from the environment to evaluate the last action.
- Policy (π): Strategy that agent employs to determine next action based on current state.
- Value (V): Expected long-term return with discount, as opposed to short-term reward.

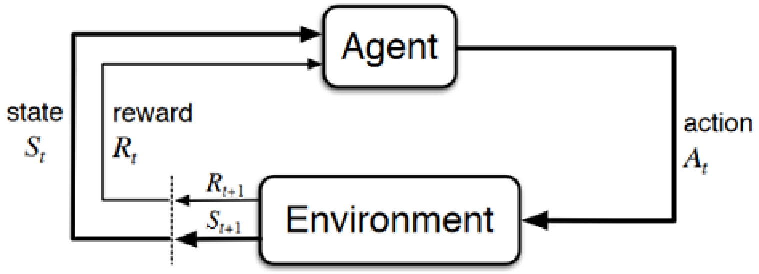


Figure1: The Canonical MDP Diagram

Reinforcement learning is not just limited to games. It is used for managing stock portfolios and finances, for making humanoid robots, for manufacturing and inventory management, to develop general AI agents, which are agents that can perform multiple things with a single algorithm, like the same agent playing multiple Atari games.

2 Environment

Reinforcement learning environments can take on many different forms, including physical simulations, video games, stock market simulations, etc. The reinforcement learning community (and, specifically, OpenAI) has developed a standard of how such environments should be designed, and the library which facilitates this is OpenAI's Gym.

Gym provides different game environments which we can plug into our code and test an agent. The library takes care of API for providing all the information that our agent would require, like possible actions, score, and current state. We just need to focus just on the algorithm part for our agent. The environment provided to us is a basic deterministic n × n grid-world environment.

The core gym interface is `env`, which is the unified environment interface. The following are the `env` methods that we have used:

- `env.reset`: Resets the environment and returns a random initial state.
- `env.step(action)`: Step the environment by one timestep. Returns the following :
    a. **observation (used as obs)**: Observations of the environment.
    b. **reward**: If your action was beneficial or not.
    c. **done**: Indicates if we have successfully completed one *episode.*
    d. **info**: Additional info such as performance and latency for debugging purposes.
- `env.render`: Renders one frame of the environment (helpful in visualizing the environment).

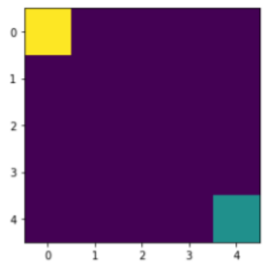


Figure2: Initial State of our basic grid world environment

The Figure 2 shows the initial state of a 4 × 4 grid-world where the agent (shown as the yellow square) has to reach the goal (shown as the green square) in the least amount of time steps possible. The environment's state space is described as an n × n matrix with real values on the interval [0, 1] to designate different features and their positions. The agent will work within an action space consisting of four actions: **up**, **down**, **left**, **right**. At each time step, the agent will take one action and move 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.

3 Q-Learning Algorithm

We have implemented simple RL algorithm called Q-learning which gives our agent some memory. Essentially, Q-learning lets the agent use the environment's rewards to learn, over time, the best action to take in a given state. A Q-value for a particular state-action combination is representative of the "quality" of an action taken from that state. Better Q-values imply better chances of getting greater rewards.

The Q-learning process is shown in simple steps below :

- Initialize the Q-table by all zeros.
- Start exploring actions: For each state, select any one among all possible actions for the current state (S).
- Travel to the next state (S') as a result of that action (a).
- For all possible actions from the state (S') select the one with the highest Q-value.
- Update Q-table values using the equation.
- Set the next state as the current state.
- If goal state is reached, then end and repeat the process.

3.1 Implementing Policy Function:

Select a random uniform number. If it's less than epsilon, return the random choice action space.

When it is not deciding the action randomly, the agent will predict the reward value based on the current state and pick the action that will give the highest reward.

$\mathbf{Q_\theta}$ tells us which action will lead to which expected cumulative discounted reward, and our policy π will choose the action **a** which, ideally, will lead to the maximum such value given the current state $\mathbf{S_t}$.

$$\pi(s_t) = \underset{a \in A}{\operatorname{argmax}} \, Q_\theta(s_t, a)$$

3.2 Update Q-table:

The values stored in the Q-table are called *Q-values*, and they map to a (state, action) combination. Q-values are initialized to an arbitrary value, and as the agent exposes itself to the environment and receives different rewards by executing different actions, the Q-values are updated as the environment state's get explored using iteration update algorithm. The update rule for Q-value is given below :

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{a} \Big)}^{\text{learned value}}$$

Where:-

- α (alpha) is the learning rate ($0 < \alpha \leq 1$) - it is the extent to which our Q-values are being updated in every iteration.
- γ (gamma) is the discount factor ($0 \leq \gamma \leq 1$) - determines how much importance we want to give to future rewards. A high value for the discount factor (close to **1**) captures the long-term effective award, whereas, a discount factor of **0** makes our agent consider only immediate reward, making it greedy.

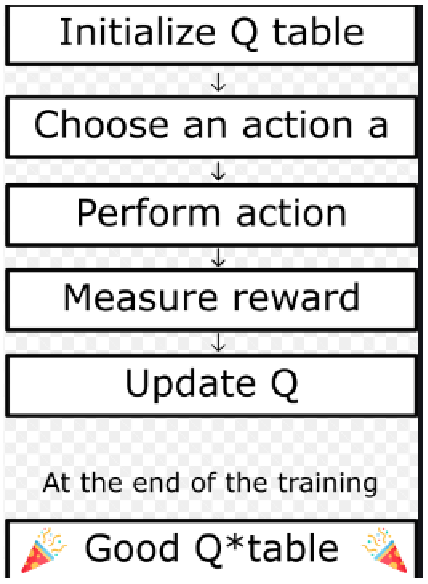
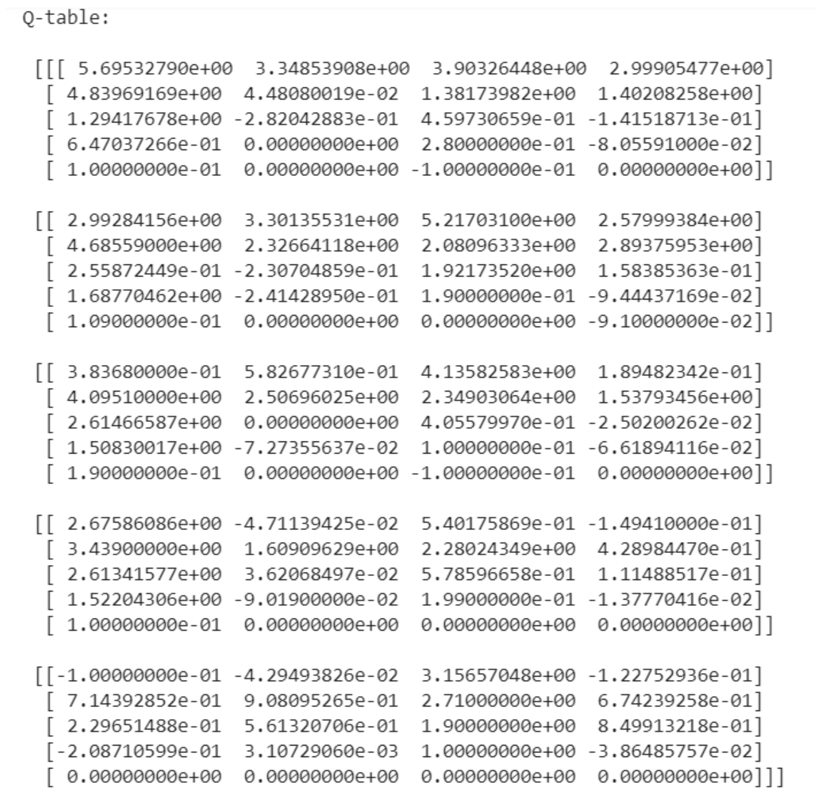


Figure3: Updation of Q-table

Explanation of update formula in brief :

1. We have updated (⟵) the Q-value of the agent's current *state* and *action* by first taking weight (1−α) of the old Q-value, then adding the learned value.
2. The learned value is a combination of the reward for taking the current action in the current state, and the discounted maximum reward from the next state we will be in once we take the current action.
3. In short, in current state by looking at the reward for the current state/action , and the max reward for the next state the agent is learning to take proper action in current state.
4. The Q-value of a state-action pair is the sum of the instant reward and the discounted future reward (of the resulting state). The way we store the Q-values for each state and action is through a **Q-table.** Shown below is the Q-table obtained after training the algorithm.



```
Q-table:

 [[[ 5.69532790e+00  3.34853908e+00  3.90326448e+00  2.99905477e+00]
  [ 4.83969169e+00  4.48080019e-02  1.38173982e+00  1.40208258e+00]
  [ 1.29417678e+00 -2.82042883e-01  4.59730659e-01 -1.41518713e-01]
  [ 6.47037266e-01  0.00000000e+00  2.80000000e-01 -8.05591000e-02]
  [ 1.00000000e-01  0.00000000e+00 -1.00000000e-01  0.00000000e+00]]

 [[ 2.99284156e+00  3.30135531e+00  5.21703100e+00  2.57999384e+00]
  [ 4.68559000e+00  2.32664118e+00  2.08096333e+00  2.89375953e+00]
  [ 2.55872449e-01 -2.30704859e-01  1.92173520e+00  1.58385363e-01]
  [ 1.68770462e+00 -2.41428950e-01  1.90000000e-01 -9.44437169e-02]
  [ 1.09000000e-01  0.00000000e+00  0.00000000e+00 -9.10000000e-02]]

 [[ 3.83680000e-01  5.82677310e-01  4.13582583e+00  1.89482342e-01]
  [ 4.09510000e+00  2.50696025e+00  2.34903064e+00  1.53793456e+00]
  [ 2.61466587e+00  0.00000000e+00  4.05579970e-01 -2.50200262e-02]
  [ 1.50830017e+00 -7.27355637e-02  1.00000000e-01 -6.61894116e-02]
  [ 1.90000000e-01  0.00000000e+00 -1.00000000e-01  0.00000000e+00]]

 [[ 2.67586086e+00 -4.71139425e-02  5.40175869e-01 -1.49410000e-01]
  [ 3.43900000e+00  1.60909629e+00  2.28024349e+00  4.28984470e-01]
  [ 2.61341577e+00  3.62068497e-02  5.78596658e-01  1.11488517e-01]
  [ 1.52204306e+00 -9.01900000e-02  1.99000000e-01 -1.37770416e-02]
  [ 1.00000000e-01  0.00000000e+00  0.00000000e+00  0.00000000e+00]]

 [[-1.00000000e-01 -4.29493826e-02  3.15657048e+00 -1.22752936e-01]
  [ 7.14392852e-01  9.08095265e-01  2.71000000e+00  6.74239258e-01]
  [ 2.29651488e-01  5.61320706e-01  1.90000000e+00  8.49913218e-01]
  [-2.08710599e-01  3.10729060e-03  1.00000000e+00 -3.86485757e-02]
  [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]]
```

Figure4:Q-table

3.3 Training the Agent:

1. After initialization, we passed the initial state to obs.
2. Then checked if it's already done. If **done = False**, we kept going.
3. While it's not done, we updated state, action_reward and next_state. We get action by step the current state on agent.
4. Then used copy to record the current state. step the current action on environment to return the new state, the reward for performing the action, a boolean indicating if the run is over and some other information.
5. Added the new reward on the total rewards. Used copy to save the new state returned by step.
6. Updated the state, action, reward, next_state of agent.

4 Hyperparameters

1. **α :** (the learning rate) should decrease as you continue to gain a larger and larger knowledge base.

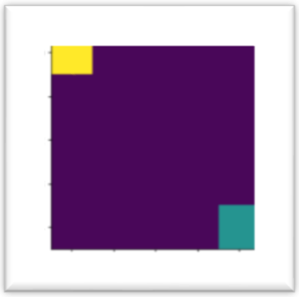
2. **γ:** as you get closer and closer to the deadline, your preference for near-term reward should increase, as you won't be around long enough to get the long-term reward, which means your gamma should decrease.

3. Epsilon **:**

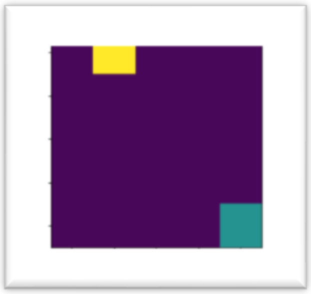
- **ϵ:** as we develop our strategy, we have less need of exploration and more exploitation to get more utility from our policy, so as trials increase, epsilon should decrease.
- Our agent randomly selectS its action at first by a certain percentage, called 'exploration rate' or 'epsilon'. This is because at first, it is better for the agent to try all kinds of things before it starts to see the patterns.
- After enough random exploration of actions, the Q-values tend to converge serving our agent as an action-value function which it can exploit to pick the most optimal action from a given state.
- There's a tradeoff between **exploration** (choosing a random action) and **exploitation** (choosing actions based on already learned Q-values). We want to prevent the action from always taking the same route, and possibly overfitting, so we'll be introducing another parameter called ϵ "epsilon" to cater to this during training. Instead of just selecting the best learned Q-value action, we'll sometimes favor exploring the action space further.
- We want our agent to decrease the number of random action, as it goes, so we introduce an exponential-decay epsilon, that eventually will allow our agent to explore the environment. Lower epsilon value results in episodes with more penalties (on average) which is obvious because we are exploring and making random decisions.

5 Result

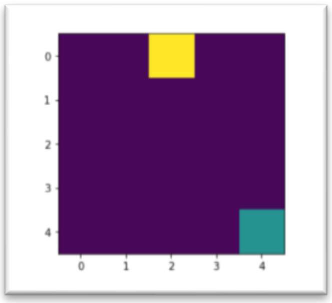
We trained the agent for 1000 episodes to navigate from starting point to the goal. Agent's performance evaluated after Q-learning and is as shown below :



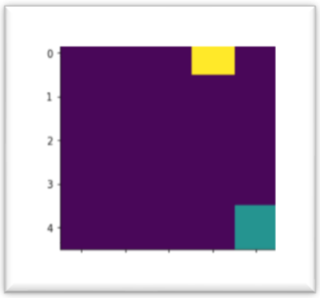
1



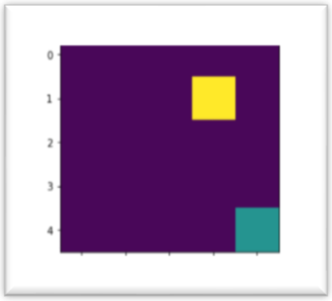
2



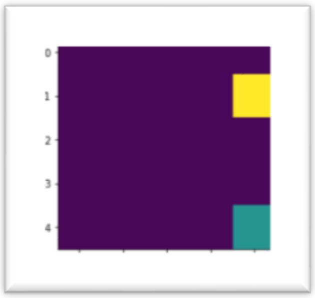
3



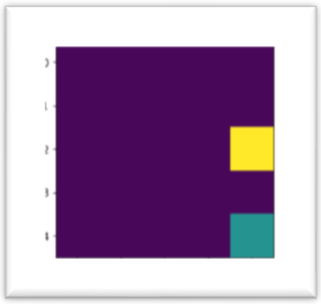
4

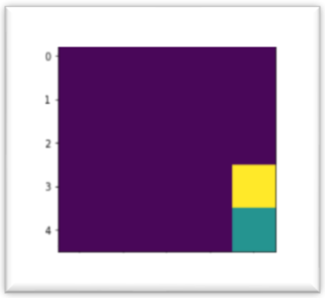


5



6


7


8

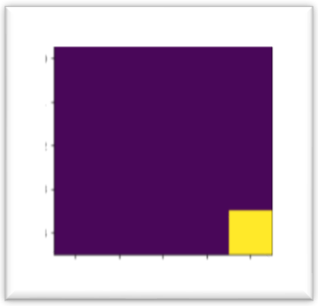
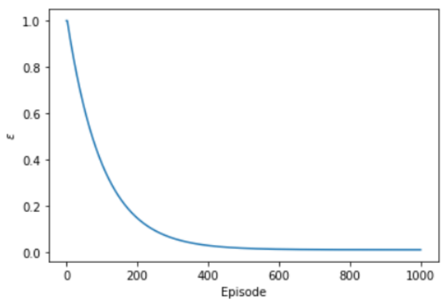

9

Figure5:Agent's performance after Q-learning



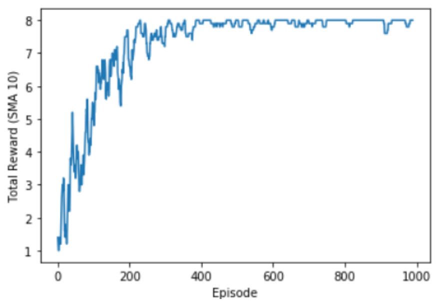


Figure6:Plot of Epsilon versus Episode and Total Rewards versus Episode

6 Conclusion

We understood the basics of Reinforcement Learning and then used OpenAI's Gym to provide us with a related environment, where we trained our agent. We implemented the Q-learning algorithm and the agent's performance was improved significantly after 1000 Episodes which is depicted in Q-table.

References

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

[2] https://ublearns.buffalo.edu/bbcswebdav/pid-5108716-dt-content-rid-25438592_1/courses/2199_23170_COMB/15.3-Q-Learning.pdf

[3] https://www.quora.com/How-does-Q-learning-work-1

[4] https://itnext.io/reinforcement-learning-with-q-tables-5f11168862c8

[5] https://towardsdatascience.com/training-an-agent-to-beat-grid-world-fac8a48109a8

[6] https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56