Reinforcement Learning: Navigating the classic 4x4 grid-world environment

Ankita Das
adas4@buffalo.edu

Abstract

This paper presents the development and the simulation of a reinforcement learning agent to navigate the classic 4x4 grid-world environment. The agent will learn an optimal policy through Q-Learning which will allow it to take actions to reach a goal while avoiding obstacles. In Reinforcement Learning, learning takes place as a result of interaction between the agent and the environment. The idea of learning is that the percepts received by an agent should not only be used for understanding, interpreting, prediction but also for acting. So that the agent learns the optimal actions to maximize reward. We make the environment and agent to be compatible with OpenAI Gym environments and have two agents- Random Agent which takes random actions and Heuristic Agent. We get the maximum reward – mean rolling reward, as the value 8.

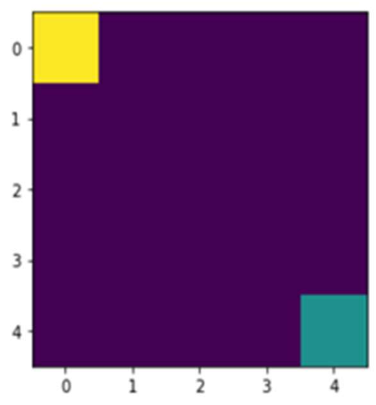
1. Introduction

Reinforcement Learning, known as a semi-supervised learning model in machine learning, is a technique to allow an agent to take actions and interact with an environment to maximize the total rewards.

RL has a very close relationship with psychology, biology and neuroscience in our everyday life. If you think about it, what a RL agent does is just trial-and-error: it learns how good or bad its actions are based on the rewards it receives from the environment. And this is exactly how humans learns to make a decision. This paper presents the building of a reinforcement learning agent to navigate the classic 4x4 grid-world environment and achieve maximum reward equal in the range 6 to 8.

2. Environment Definition

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 (https://gym.openai.com/).



The environment we provide is a basic deterministic n×n grid-world environment (the initial state for a 4×4 grid-world is shown in Figure above) where the agent (shown as the green square) has to reach the goal (shown as the yellow square) in the least amount of time steps possible. The environment's state space will be 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. Data Preprocessing

We are using Jupyter notebook to work on this dataset. We start off by importing all the necessary libraries for the code. Next, Creating the Environment:

- Define the (4 x 4) grid.
- Define the methods for agent movement or actions – UP, DOWN, LEFT and RIGHT.
- No movement will be done if the grid ends in the direction of movement.

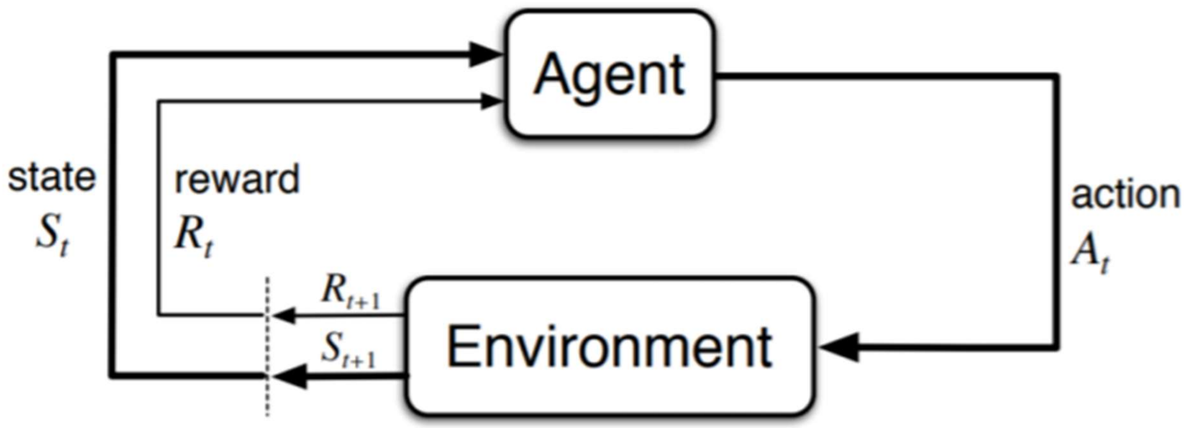
4. Model Architecture

Reinforcement Learning and Q-Learning:

Reinforcement learning is a concept in Machine Learning where an agent learns its future actions based on the results from the previous and present actions.

- It performs trial-and-error actions and learns from the feedback rewards from its environment received from these actions to take the future actions.
- The objective is to associate each state to its actions to maximize the final reward as the result of learning.
- The agent is supposed to learn a method of operation, a control policy that **maximizes the expected sum of rewards**.
- The future rewards are discounted exponentially using a discounting factor.

Reinforcement learning brings to use **Markov Decision Process** for receiving the feedback and taking future actions.



Episode: $s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, a_3, \ldots, s_T$

Let's define the key terms below:

- **Action (A):** all possible moves the agent can take. In this task, **there are 4 possible actions – UP, DOWN, LEFT, RIGHT**
- **State (S)**: The current position/state that the agent is in, returned by the environment. **We have a 4x4 grid environment** and the agent can have any of the 25 states from these, for this task.
- **Reward (R)**: The reward that is returned for a particular action immediately from the environment to evaluate the last action. In this task, we have 3 defined rewards –
    1. +1: When the agent moves takes an action and moves to a state closer to the goal
    2. 0: When the agent does not move at all
    3. -1: When the agent takes an action and moves to a state away from the goal
- **Policy (Q)**: The strategy the agent employs to determine next action based on the current state. Here, we will use the concept of Q-learning.
- **Value (V)**: An expected long-term return with discount, as opposed to short-term reward.

After, having an idea of all these terms, let's dive into the ways in which **our game is designed**. Given the environment with the states, rewards and the agent –

- We start an episode, i.e. we take an initial state *so* and pass it to the agent.
- The agent passes an action *ao* based on the state *so* back to the environment.
- The environment reacts to the action, passes the next state s1 along with the resulting reward, ro for taking that action, back to the agent.
- The process continues until the agent reaches a terminal state, indicating the end of this episode.
- Repeat the same process for further episodes and train the agent based on all its feedback

So, ***in Q-learning***, **we follow a value-based approach** and build a **memory table Q(s,a)** to store Q-values for all possible combinations of *s* and *a.* This means that we sample an action from the current state and calculate the reward and the new state *s'*. From the memory table, we determine the next action *a'* to take for the maximum *Q(s',a')*. So, the optimized value of Q(s,a) is achieved by the **Bellman equation** –

$$Q(s,a) = r(s,a) + \gamma \max_{a'} Q(\delta(s,a), a'),$$

Where, r(s,a) = reward for an action a in state s

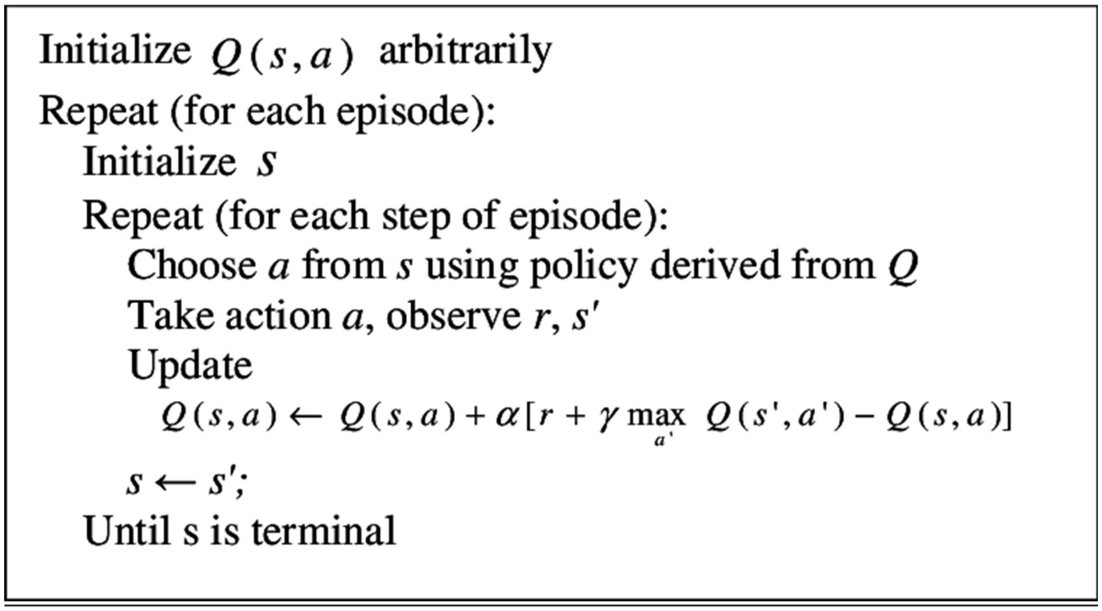
γ = discounting factor

Importance of Discounting Factor (γ):

It's important to maintain a check on the future rewards. The discounting factor γ basically penalizes these rewards. $\gamma \in [0, 1]$. We need this because –

- The future rewards do not always provide immediate best results.
- Discounting eases mathematical convenience.
- Future rewards may/may not have higher uncertainty, and we need to check that.

Q- Learning Algorithm:



```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize $s$
    Repeat (for each step of episode):
        Choose $a$ from $s$ using policy derived from $Q$
        Take action $a$, observe $r$, $s'$
        Update
            $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$;
    Until s is terminal
```

Exploration and Exploitation:

In Reinforcement Learning, there are two important concepts – exploration and exploitation that dictate how the model should pattern its learning process.

- Exploitation: The agent learns to take actions that come from the current best version of the learned methods. Here, the agent puts to use all its previous observations and trains them in a deep neural network to find the best actions for the following states. It seeks for the actions that it knows will achieve a high reward.
- Exploration: Keep training on new and more actions to obtain more data. Basically, keep moving in the environment and keep exploring.

Both these policies can be used and switched to each other using a decay function – epsilon decay function. The function is given as –

$$\epsilon = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) * e^{-\lambda|S|},$$

where $\epsilon_{min}, \epsilon_{max} \in [0,1]$

λ - hyperparameter for epsilon

$|S|$ - total number of steps

The agent first takes actions based on ϵ - the exploration rate, where it explores and tries all actions before it sees some patterns of best rewards and Q-values. When not deciding the action randomly, the agent will always go for the highest reward, which we want to reduce. With the epsilon-decay function, the randomly taken actions will be reduced, and more exploration will be introduced.

5. Implementation:

❑ **Creating the Environment:**

- Define the (4 x 4) grid.
- Define the methods for ways of movement by Agent – UP, DOWN, LEFT and RIGHT.
- No movement will be done if the grid ends in the direction of movement.
- Check for the current state after the last movement.
- Get the rewards for each movement/action. +1 for moving towards the goal, 0 for not moving at all and -1 for moving away from the goal.
- Check if the agent has reached the goal or if the number of episodes are over.
- Render the movements of Agent on the grid.
- Reset the environment after completion of the episodes or at the terminal state.

❑ **Define Agents behavior:**

- Initialize all parameters that will control the agent's behavior-

  exploration rate (ϵ), discount factor (γ), learning rate (α)

- Our agent will randomly select 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. Hence we select a random uniform number. If it's less than epsilon, we return the random choice action space.
- And when the agent is not deciding the action randomly, it will predict the reward value based on the current state and pick the action that will give the highest reward.

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

- In Q-learning agent learns from Temporal Difference which takes into account the current observation and the previous observation.

$$TD(a, s) = R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a)$$

- We calculate the best Q-values using policy. Seek random samples of previous observations from the behavior and update best Q-values in Q-table.

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

❑ **Training and Implementing Q- Learning Algorithm:**

- We first initialize the environment(observation_space) and the agent's action(action_space)
- We initialize Q(s, a) values.
- We iterate over two loops - episode and time step
- After initialization, we pass the initial state to observation_space.
- For each step of episode, We choose an action a from s using policy derived from Q.
- Then the agent performs the action and gets a reward r and new state s'
- We update the state, action, reward and next_state of the agent in Q-table.
- We set the initial state to s'.
- We get the best average reward equal to 8.

❑ **Finally run the game:**

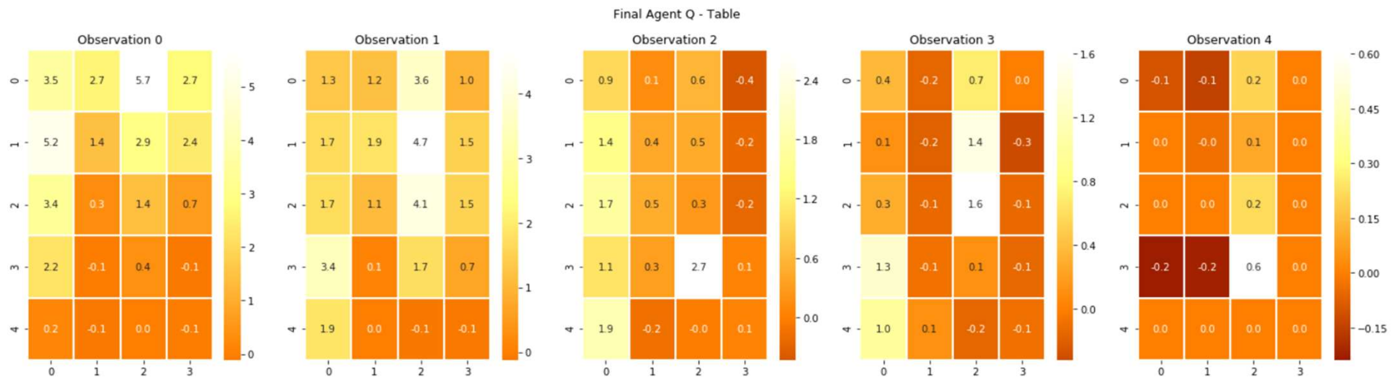
- Initialize the grid as 5x5.
- Define all hyper-parameters
- Hyper-parameters include lambda, epsilon max/min, number of episodes, gamma
- Call all the above methods in sequence
- We check for the maximum reward – mean rolling reward, and try to achieve this value in the range of 6 to 8.
- Plot the necessary graphs.

6. Results:

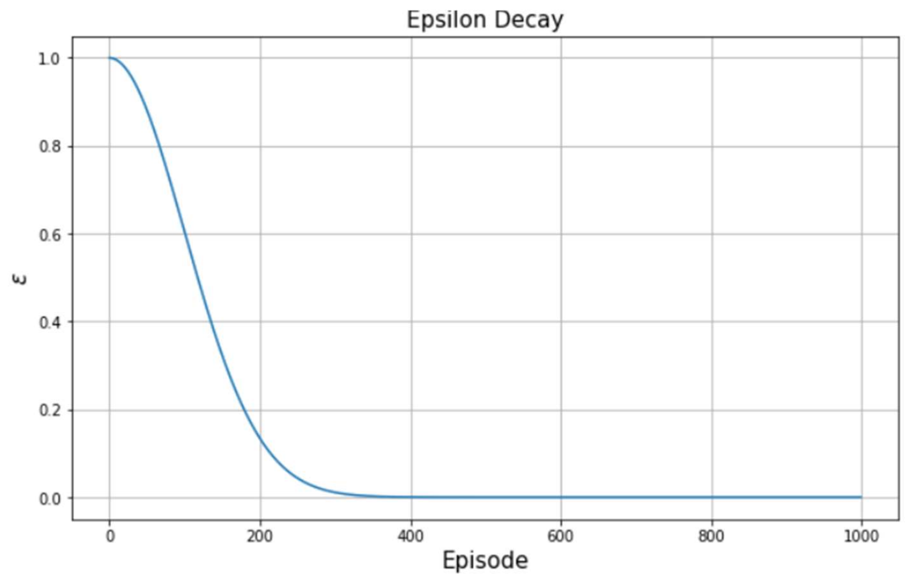
We evaluate the efficiency of learning of our agent and evaluate the mean-rolling reward, which can be at max 8 for a 5x5 grid.

Paramters	**Value**
Number of episodes	1000
Epsilon	0.9999
Gamma	0.9
Learning Rate	0.1
Best Average Reward	8

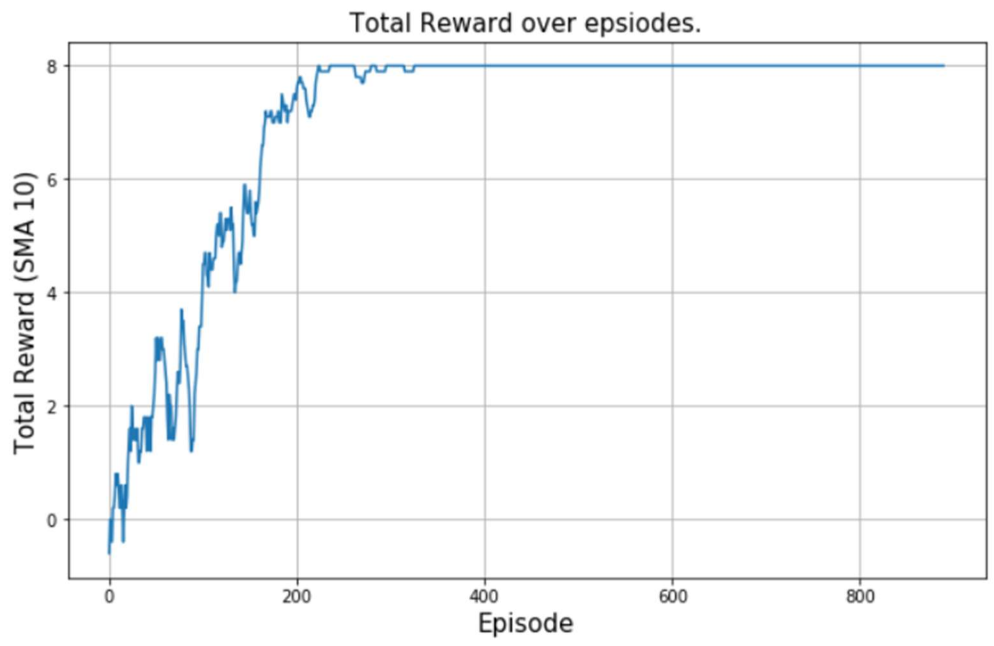
We print the Q- table of final agent as below:



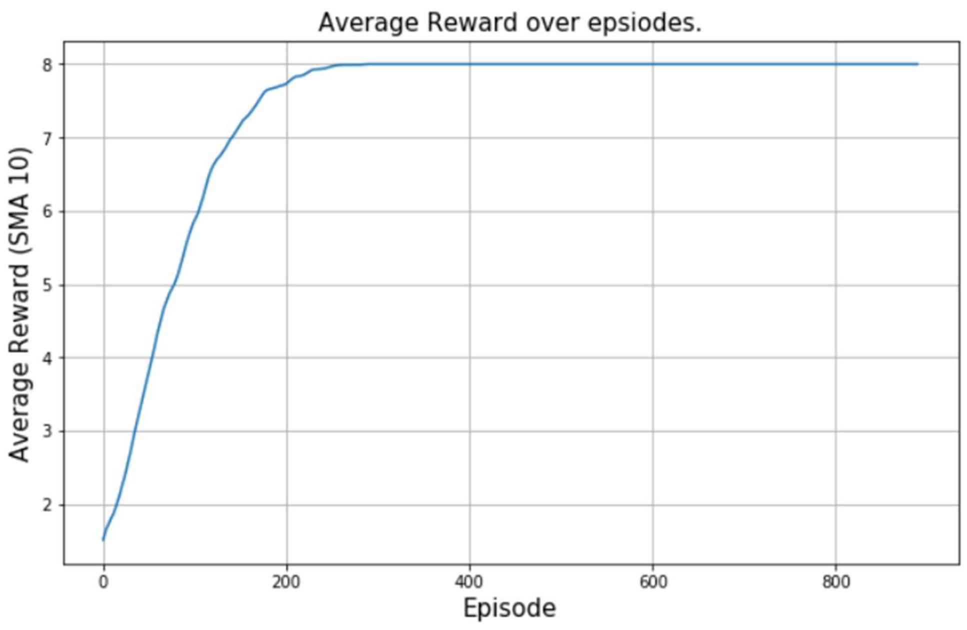
We plot the number of episodes vs epsilon values as below :



We plot the number of episodes vs total rewards as below:



We plot the number of episodes vs average rewards as below:



7. Conclusion

This paper presents the development and the simulation of reinforcement learning agent to navigate the classic 4x4 grid-world environment.

RL is unique in its use of the combination of four distinct machineries: stochastic approximation, DP, AI and function approximation. By exploiting these machineries, RL has opened the avenue for solving large-scale MDPs (and its variants) in discrete-event systems, which were considered intractable in the past. RL has outperformed industrial heuristics in several of the published case studies of industrial importance. RL as a science is relatively young and has already made a considerable impact on operations research. The optimism expressed about RL in the early surveys (Keerthi and Ravindran, 1994; Kaelbling et al., 1996; Mahadevan, 1996) has been bolstered by several success stories.

8. Acknowledgments

We are extremely grateful to Professor Sargur Srihari and Mihir Chauhan for teaching all the necessary concepts related to Reinforcement Learning and helping in this project throughout.

9.References

- https://medium.com/@jonathan_hui/rl-dqn-deep-q-network-e207751f7ae4
- https://sergioskar.github.io/Deep_Q_Learning/
- https://sergioskar.github.io/Deep_Q_Learning/
- https://skymind.ai/wiki/deep-reinforcement-learning
- https://medium.freecodecamp.org/an-introduction-to-deep-q-learning-lets-play-doom-54d02d8017d8
- http://rail.eecs.berkeley.edu/deeprlcourse/
- https://towardsdatascience.com/cartpole-introduction-to-reinforcement-learning-ed0eb5b58288