Implementing Reinforcement Learning Using Q-Learning Algorithm

Upasana Ghosh
Department of Computer Science
University at Buffalo
Buffalo, NY 14214
upasanag@buffalo.edu

Abstract:

The purpose of this project is to build a reinforcement learning agent that will navigate the classic 4x4 grid-world environment. The agent must learn an optimal policy through a Q-Learning algorithm which will allow it to take actions to reach the goal while avoiding obstacles. For developing this project, we have used a toolkit called Gym, provided by the Python framework, which is used for generating Grid environments for developing and comparing reinforcement learning algorithms. Initially, we just observe the performance of a Random Agent (runs through the environment by taking random actions. Neither does he learn, nor remember anything) and a Heuristic Agent. Then we train our agent through Q-learning approach (by defining policies and updating Q-tables). In the first task, we implement the policy function, in the second task we update the Q-table to find a policy that is optimal in the sense that it maximizes the expected value of the total reward over all successive steps, starting from the current state. The final task is to implement the training algorithm and make the agent reach the goal state from the starting state in optimal number of steps.

Introduction:

Reinforcement learning is a machine learning paradigm which focuses on how automated agents can learn to take actions in response to the current state of an environment, so as to maximize some reward. The environment is typically stated in the form of a Markov Decision Process (MDP) because many reinforcement learning algorithms for this context utilize dynamic programming techniques. The main difference between the classical dynamic programming methods and reinforcement learning algorithms is that the latter do not assume knowledge of an exact mathematical model of the MDP and they target large MDPs where exact methods become infeasible.

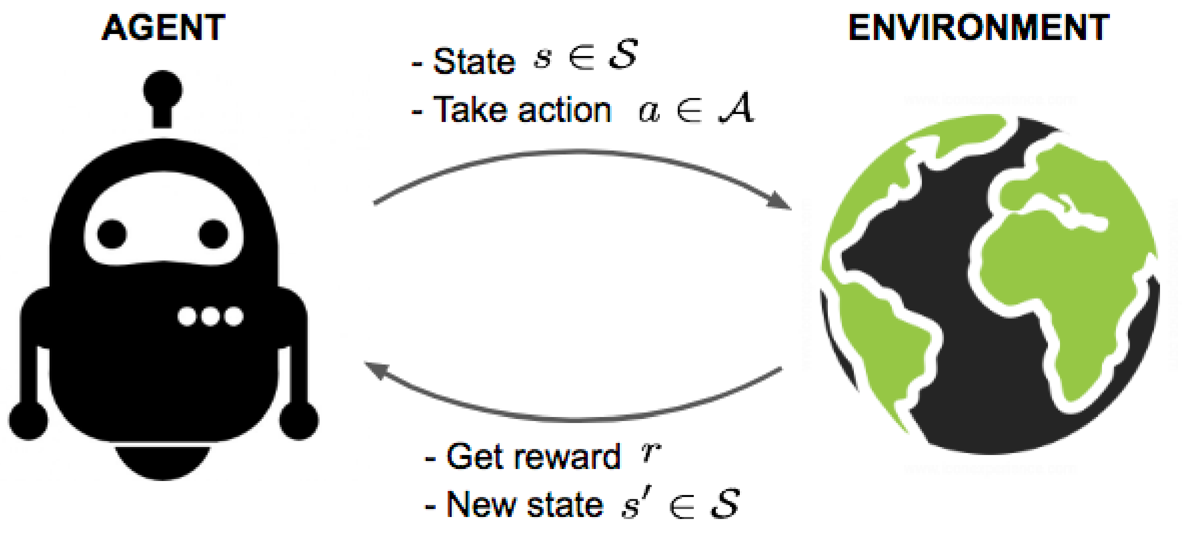

AGENT
ENVIRONMENT
- State $s \in \mathcal{S}$
- Take action $a \in \mathcal{A}$
- Get reward r
- New state $s' \in \mathcal{S}$

Figure: A typical representation of a Reinforcement Learning scenario

The key points in Reinforcement Learning are:

- **Input**: The input should be an initial state from which the agent will start.
- **Output**: There are many possible outputs as there are a variety of solutions to a given problem.
- **Training**: The training is based upon the input. The model will return a state and the user will decide to reward or punish the model based on its output.
- The model continues to learn.
- The best solution is decided based on the maximum reward achieved by the agent.

Reinforcement is of two types:

1. Positive:
   Positive Reinforcement is defined as when an event, occurs due to a particular behavior, increases the strength and the frequency of the behavior. In other words, it has a positive effect on the behavior.

   Advantages:

   - Maximizes Performance.
   - Sustain Change for a long period of time.

   Disadvantages:

   - Too much Reinforcement can lead to overload of states which can diminish the results.

2. Negative:
   Negative Reinforcement is defined as strengthening of a behavior because a negative condition is stopped or avoided.

   Advantages:

   - Increases Behavior.
   - Provides defiance to minimum standard of performance.

   Disadvantages:

   - It Only provides enough to meet up the minimum behavior

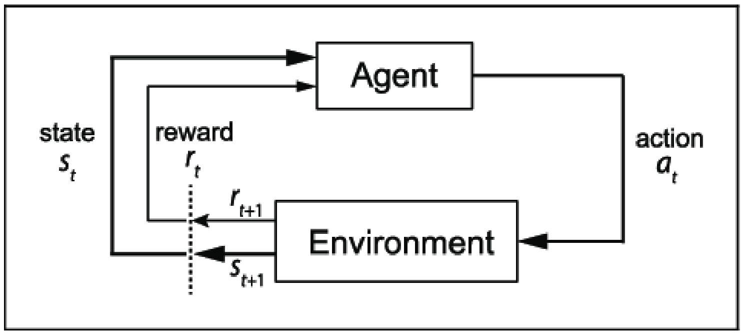


Figure: The canonical MDP diagram

Reinforcement Learning is typically modeled as a Markov decision process (MDP) as shown in the above figure.

An MDP is a 4-tuple (S; A; P; R), where:

1. S: the set of all possible states for the environment
2. A: the set of all possible actions the agent can take
3. P: $P_r(s_{t+1} = s' | s_t = s, a_t = a)$ is the state transition probability function
4. R: S x A x S → R is the reward function

Our task is to find a policy π: S → A which our agent will use to take actions in the environment to maximize the cumulative reward, i.e.,

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

where $\gamma \in [0,1]$ is the discounting factor (used to give more weight to more immediate rewards), s_t the state at time step t, a_t is the action the agent takes at time step t, and s_{t+1} is the state which the environment transitions to after the agent takes the action.

Q-Learning:

Q-Learning is a process in which we train some function $Q_\theta: S * A \rightarrow R$ parameterized by θ, to learn a mapping from state-action pairs to their Q-value, which is the expected discounted reward for following the following policy π_θ:

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

In words, the function Q_θ will tell 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 s_t.

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.

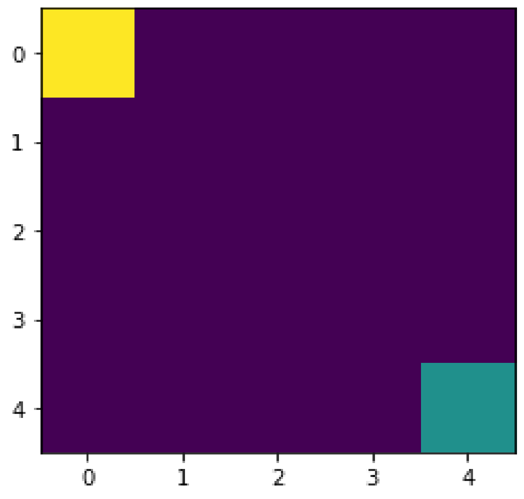


Figure: The initial state of our basic grid-world environment

The environment we have been provided is a basic deterministic nxn grid-world environment (the initial state for a 4x4 grid-world is shown in the above figure) where the agent (shown as the green square) must reach the goal (shown as the yellow square) in the least amount of time steps possible.

The environment's state space has been described as an nxn 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.

Architecture:

For this project, we have been tasked with both implementing and explaining key components of the Q-learning algorithm. We have implemented tabular Q-Learning, an approach which utilizes a table of Q-values as the agent's policy. 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. 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. We want our agent to decrease the number of random actions as it goes, so we introduce an exponential-decay epsilon, that eventually will allow our agent to explore the environment.

TASK 1: Implementing the policy function:

- Our agent has randomly selected its action at first by a certain percentage, called 'exploration rate' or 'epsilon'. Epsilon has been considered as 1.
- This is for the agent to try all kinds of possible movements before it starts seeing the pattern.
- Then we have selected a random uniform number and checked if it is lesser than 'epsilon'. If it is less, we have returned 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.
- The policy is returned.

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

TASK 2: Update Q-table:

- In this project, we have created an |S| x |A| array as our Q-table which would have entries $q_{i,j}$ where i corresponds to the i^{th} state (row) and j corresponds to the j^{th} action (column), so that if s_t is located in the i^{th} row and a_t is the j^{th} column, $Q(s_t, a_t) = q_{i,j}$

- We have used a value iteration update algorithm to update our Q-values as we explore the environment's states:

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

Figure: Our updated rule for Q-learning

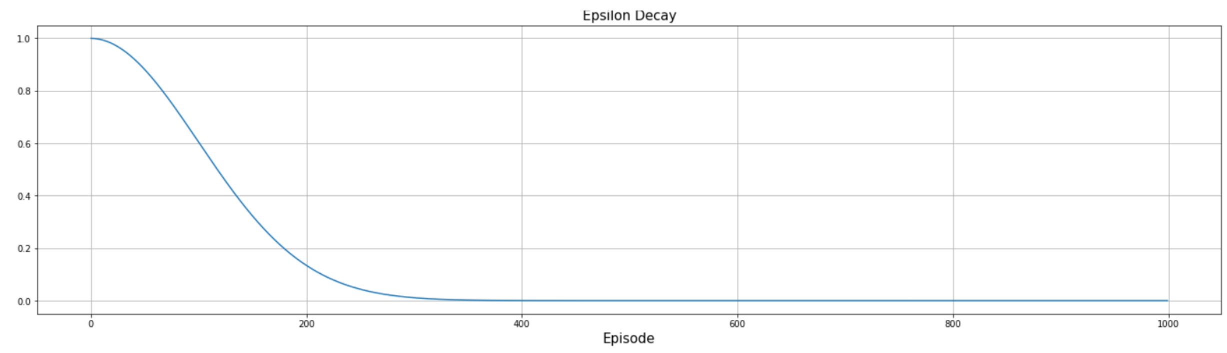
- We are using our Q-function recursively to match (following our policy π) in order to calculate the discounted cumulative total reward.
- We initialize the table with all 0 values, and as we explore the environment (e.g., take random actions), collect our trajectories and use these values to update our corresponding entries in the Q-table.

TASK 3: Implementing the training algorithm:

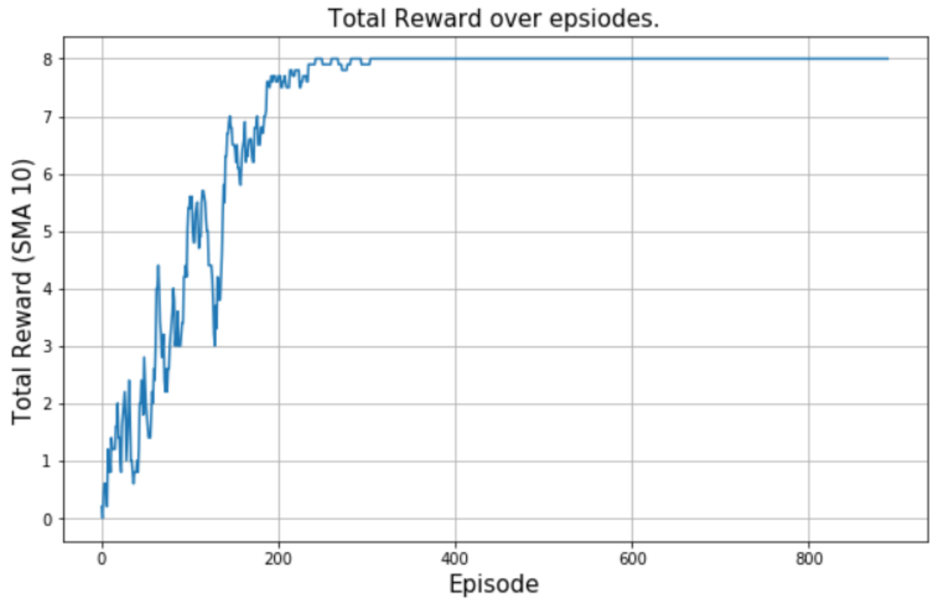
- First, we initialize our environment (much like OpenAI Gym Environments which has three methods: reset, step and render).
- When we call **reset**, we initialize the environment with a fresh episode. This allows us to effectively run through episodes (only needing to call reset at the beginning of an episode), but, more importantly, reset() returns the environment's initial state.
- The **step** method accepts an action as a parameter (which, for this example, is an integer in [0, 3]), processes the action, and returns the new state, the reward for performing the action, and a Boolean indicating if the run is over.
- While initializing the agent, we have passed both environment into QLearningAgent function.
- While training the agent for 400 episodes, for each iteration of episodes, we have updated the values of state, action, reward and next_state.
- After training the agent, the best average reward score is found to be 8.0

Results:

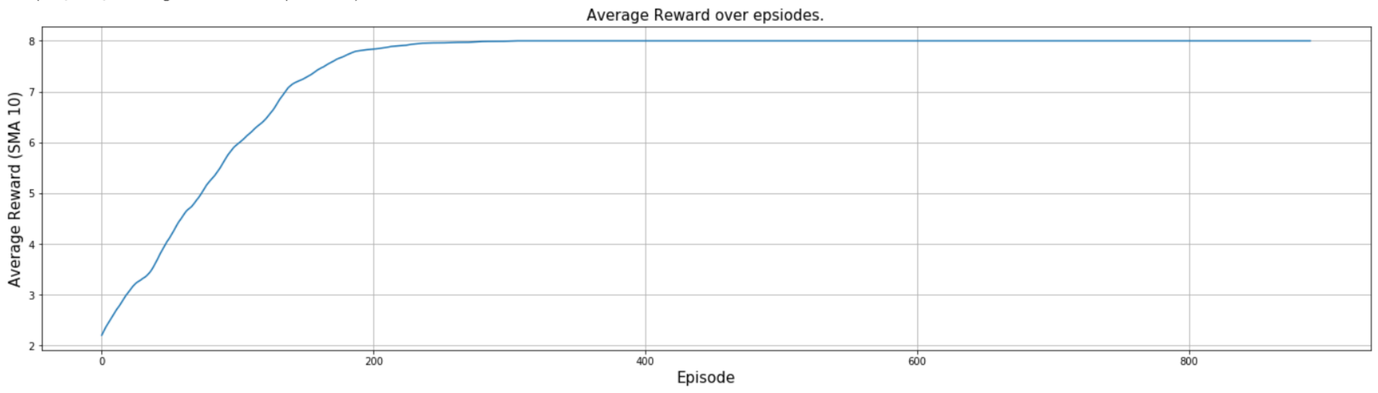
- The following graph shows how the values of Epsilon decay values against with each episode:



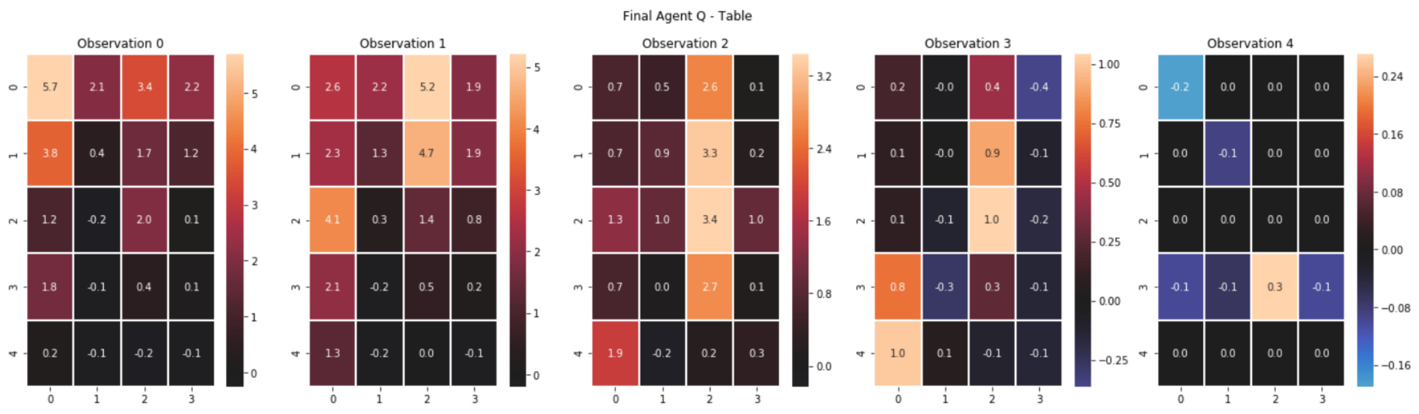
- The following graph plots the total rewards over each episode:



- The following graph plots the average rewards over each episode:



- The followings are the various Q-table states updated at each stage in the project:



Conclusion:

This project to implement Reinforcement Learning using Q-Learning algorithm provides us with an opportunity to understand how Q-Learning algorithm can be used to make an AI agent figure out an optimal policy on its own, to reach a goal state in minimum number of steps with a maximum amount of reward in an unseen environment. The learning that we achieve from this simple 4X4grid world implementation provides us with a good amount of insight about how AI agents learn to reach their goals with minimum loss in more complicated real-world settings.

Acknowledgement:

1. Machine Learning by Tom Mitchell
2. https://towardsdatascience.com/reinforcement-learning-introduction-3c772ca4d234
3. https://towardsdatascience.com/reinforcement-learning-implement-grid-world-from-scratch-c5963765ebff
4. https://en.wikipedia.org/wiki/Q-learning
5. https://blog.floydhub.com/an-introduction-to-q-learning-reinforcement-learning/
6. https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-0-q-learning-with-tables-and-neural-networks-d195264329d0
7. https://www.geeksforgeeks.org/what-is-reinforcement-learning/