Reinforcement Learning: Q-learning

Meghana Vasudeva
Person#: 50290586
University at Buffalo,
The State University of New York Buffalo,
New York 14260
mvasudev@buffalo.edu

Abstract

The task is to build a reinforcement learning agent to navigate a 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. The environment and agent which will be built will be compatible with OpenAl Gym environments. In this task I had to complete 3 missing methods of the agent class and to explain key concepts of Q-learning algorithm.

Introduction

Reinforcement Learning

Reinforcement learning is an area of machine learning that focuses on how an agent might act in an environment in order to maximize some cumulative reward. Reinforcement learning algorithms study the behavior of subjects in such environments and learn to optimize that behavior. A very common place where we can see reinforcement learning is in games. So games can be an example of our environment and the players moves are actions which when taken correctly results in points which is the reward.

Markov Decision Processes

In a Markov Decision Processes there is an agent which interacts with the environment it is in. The agent is called the decision maker as the agent selects an action to take at each step in the environment. The environment is then transitioned into a new state, and the agent is given a reward as a consequence of the previous action.

The components of MDP

- Agent
- Environment
- State
- Action
- Reward

The whole process of selecting an action from a state then going to a new state and finally receiving a reward will happen in sequence over a period of time. This process creates something called as the trajectory. This will show all the sequence of states, actions, and rewards. The agent in overall process will have to maximize the total amount of goal by taking appropriate actions in every state. Hence, in the final state the agent would have maximized all the rewards.

Notation

In an MDP, there a set of all possible states **S**, a set of all possible actions **A**, and a set of rewards **R**. By assumption we can think that they have finite states.

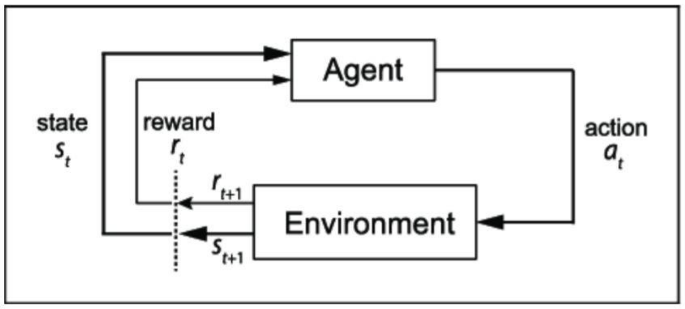
At every time step t=0,1,2…... the agent will receive a representation of the environment's states $\mathbf{S_t \in S}$. Based on this state, the agent selects an action $\mathbf{A_t \in A}$. This gives us the state-action pair ($\mathbf{S_t}$, $\mathbf{A_t}$).

The time is incremented to the next time step **t+1** and the environment is transitioned to a new state **St+1 ∈ S.** This time the agent receives a reward **Rt+1 ∈ R** for the action $\mathbf{A_t}$ which is taken from state $\mathbf{S_t}$.

We can think of the process of receiving a reward as an arbitrary function **f** that maps state-action pairs to rewards. At each time t, we have $\mathbf{f(S_t, A_t) = R_{t+1}}$

The trajectory representing the sequential process of selecting an action from a state, transitioning to a new state, and receiving a reward can be represented as $\mathbf{S_0, A_0, S_1, A_1, S_2, A_2 \ldots}$

Representation of the MDP



The canonical MDP diagram

Q-Learning

Q-learning is an off-policy reinforcement learning algorithm which helps to find the best action to take given the current state. It's considered off-policy because the q-learning function learns from actions that are outside the current policy, like taking random actions, and therefore a policy isn't needed. More specifically, q-learning seeks to learn a policy that maximizes the total reward. Here the letter Q stands for quality i.e. the quality on how useful a given action is to gain rewards.

Q-Learning is a process in which we train some function $Q_\theta: S \times A \rightarrow R$, 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 $\pi\theta$:

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

we need create an |S| × |A| array, our Q-Table, which would have entries $q_{i,j}$ where i corresponds to the ith state (the row) and j corresponds to the jth action (the column), so that if s_t is located in the ith row and at is the jth column, $Q(s_t, a_t) = q_{i,j}$.

Updating Q-table

Here are the 3 basic steps:

1. Agent starts in a state (s1) takes an action (a1) and receives a reward (r1)
2. Agent selects action by referencing Q-table with highest value (max) **OR** by random (epsilon, ε)
3. Update q-values

When the agent gets the reward from the previous state the Q-table needs to be updated for that state-action pair. This is done using the formula given below:

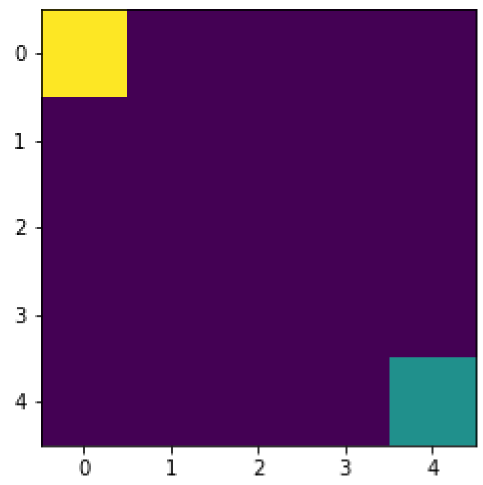
$$\boxed{Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned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

The new Q-value for this state-action pair is a weighted sum of the old value and the learned value. The new Q-value equal to the old Q-value times one minus the learning rate plus the learning rate times the learned value, which is the reward we just received from the last action plus the discounted estimate of the optimal future Q-value for the next state action pair.

Environment

Reinforcement learning environments can take on many different forms, including physical simulations, video games, stock market simulations, etc. Reinforcement learning is one of three basic machine learning paradigms, along with supervised learning and unsupervised learning.

The environment provided for this project is a basic deterministic n × n grid-world environment (the initial state for a 4 × 4 grid-world) where the agent which is shown in yellow has to reach the goal shown in green 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.



The initial state of our basic grid-world environment.

Implementation of tasks

The task was to implement 3 methods policy, update and training algorithm. Below are the processes explained.

Task 1: Implementation of Policy Method

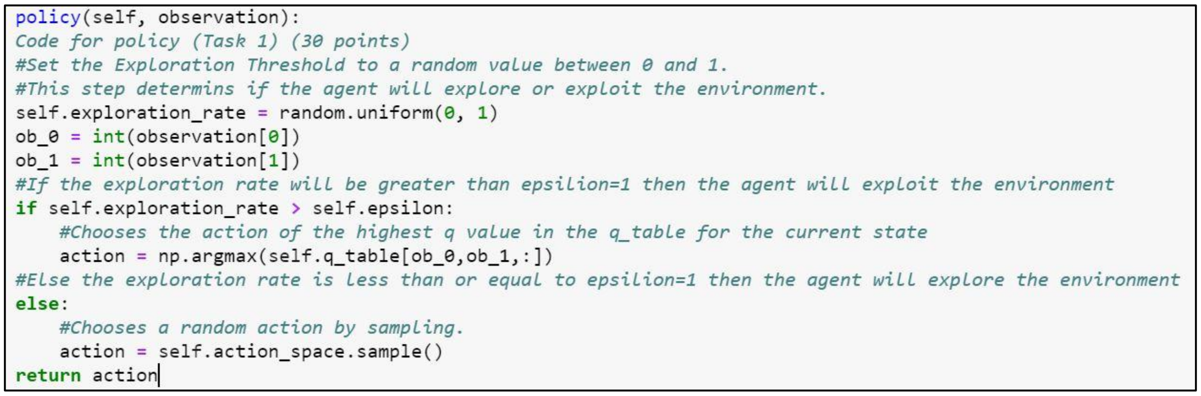
The agent will randomly select its action at first by a certain percentage, called 'exploration rate' or 'epsilon'. This is because at first, it is better for the agent to try all kinds of things before it starts to see the patterns. Select a random uniform number. If it's less than epsilon, return the random choice action space.

For each time-step within an episode, we set the exploration_rate to a random number between 0 and 1. This will be used to determine whether the agent will explore or exploit the environment in this time-step.

If the exploration_rate is greater than the epsilon which is initially set to 1, then the agent will exploit the environment and choose the action that has the highest Q-value in the Q-table for the current state. If, on the other hand, the threshold is less than or equal to epsilon, then the agent will explore the environment, and sample an action randomly. We will return the action at the end. Some important terms to know:

Learning Rate: lr or learning rate, often referred to as *alpha* or α, can simply be defined as how much you accept the new value vs the old value.

Gamma: gamma or γ is a discount factor. It's used to balance immediate and future reward. From our update rule above you can see that we apply the discount to the future reward. Typically, this value can range anywhere from 0.8 to 0.99.



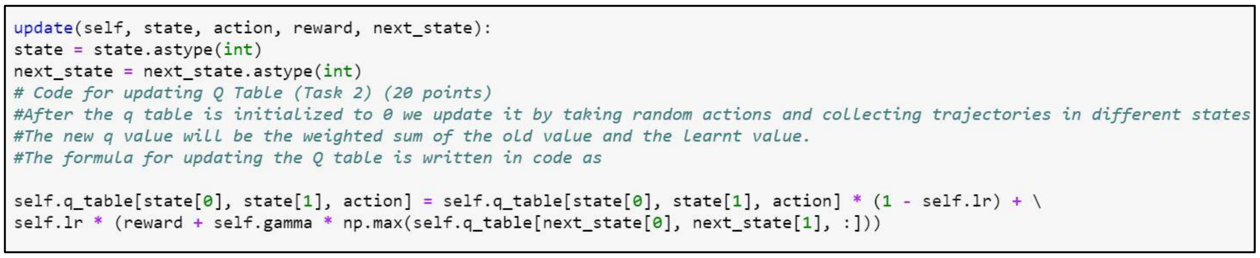
```
policy(self, observation):
Code for policy (Task 1) (30 points)
#Set the Exploration Threshold to a random value between 0 and 1.
#This step determins if the agent will explore or exploit the environment.
self.exploration_rate = random.uniform(0, 1)
ob_0 = int(observation[0])
ob_1 = int(observation[1])
#If the exploration rate will be greater than epsilion=1 then the agent will exploit the environment
if self.exploration_rate > self.epsilon:
    #Chooses the action of the highest q value in the q_table for the current state
    action = np.argmax(self.q_table[ob_0,ob_1,:])
#Else the exploration rate is less than or equal to epsilion=1 then the agent will explore the environment
else:
    #Chooses a random action by sampling.
    action = self.action_space.sample()
return action
```

Task 2: Implementation of Update Q-Table

When the agent gets the reward from the previous state the Q-table needs to be updated for that state-action pair. This is done using the formula given below:

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

The new Q-value for this state-action pair is a weighted sum of the old value and the learned value. The new Q-value equal to the old Q-value times one minus the learning rate plus the learning rate times the learned value, which is the reward we just received from the last action plus the discounted estimate of the optimal future Q-value for the next state action pair.



```
update(self, state, action, reward, next_state):
state = state.astype(int)
next_state = next_state.astype(int)
# Code for updating Q Table (Task 2) (20 points)
#After the q table is initialized to 0 we update it by taking random actions and collecting trajectories in different states
#The new q value will be the weighted sum of the old value and the learnt value.
#The formula for updating the Q table is written in code as

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

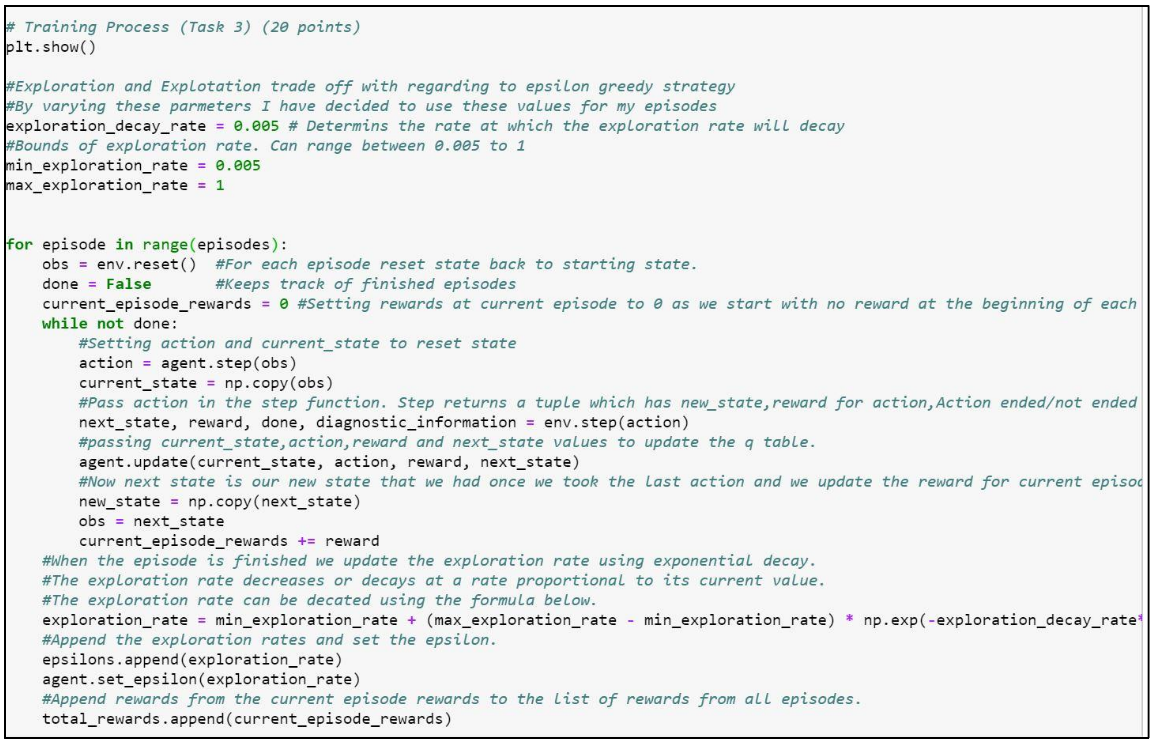
Task 3: Implementing the training process

Hyperparameter tuning and training

Here I have experimented by tuning the hyperparameters and seen the result of the learning of the agent and tried fixing the **min and max exploration rates** to **0.005** and **1.** The **exploration decay rate** was also varied and to get my optimal result I have fixed it to **0.005** to determine the rate at which the exploration_rate will decay.

The max and min are just bounds to how large or small our exploration rate can be. These parameters influence and change the algorithm. Once an episode is finished, update the exploration_rate using exponential decay, which just means that the exploration rate decreases or decays at a rate proportional to its current value.

We can decay the exploration_rate using the formula shown in the code snippet. Next, append the rewards from the current episode to the list of rewards from all episodes.

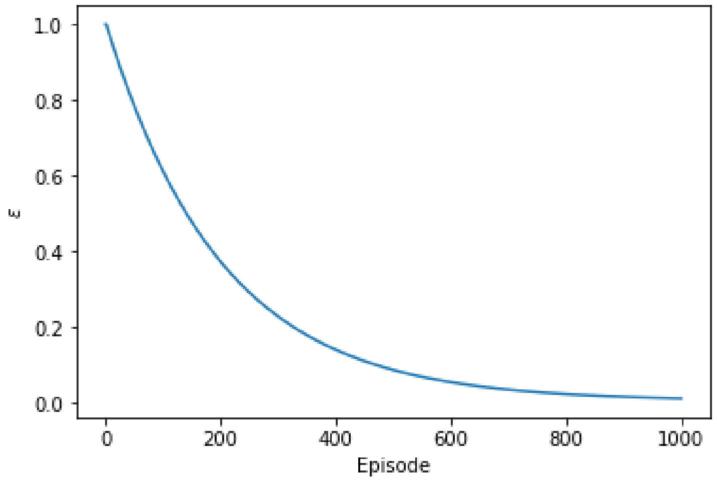


```
# Training Process (Task 3) (20 points)
plt.show()

#Exploration and Explotation trade off with regarding to epsilon greedy strategy
#By varying these parmeters I have decided to use these values for my episodes
exploration_decay_rate = 0.005 # Determins the rate at which the exploration rate will decay
#Bounds of exploration rate. Can range between 0.005 to 1
min_exploration_rate = 0.005
max_exploration_rate = 1


for episode in range(episodes):
    obs = env.reset()  #For each episode reset state back to starting state.
    done = False       #Keeps track of finished episodes
    current_episode_rewards = 0 #Setting rewards at current episode to 0 as we start with no reward at the beginning of each
    while not done:
        #Setting action and current_state to reset state
        action = agent.step(obs)
        current_state = np.copy(obs)
        #Pass action in the step function. Step returns a tuple which has new_state,reward for action,Action ended/not ended
        next_state, reward, done, diagnostic_information = env.step(action)
        #passing current_state,action,reward and next_state values to update the q table.
        agent.update(current_state, action, reward, next_state)
        #Now next state is our new state that we had once we took the last action and we update the reward for current episod
        new_state = np.copy(next_state)
        obs = next_state
        current_episode_rewards += reward
    #When the episode is finished we update the exploration rate using exponential decay.
    #The exploration rate decreases or decays at a rate proportional to its current value.
    #The exploration rate can be decated using the formula below.
    exploration_rate = min_exploration_rate + (max_exploration_rate - min_exploration_rate) * np.exp(-exploration_decay_rate*
    #Append the exploration rates and set the epsilon.
    epsilons.append(exploration_rate)
    agent.set_epsilon(exploration_rate)
    #Append rewards from the current episode rewards to the list of rewards from all episodes.
    total_rewards.append(current_episode_rewards)
```

Analysis and Results

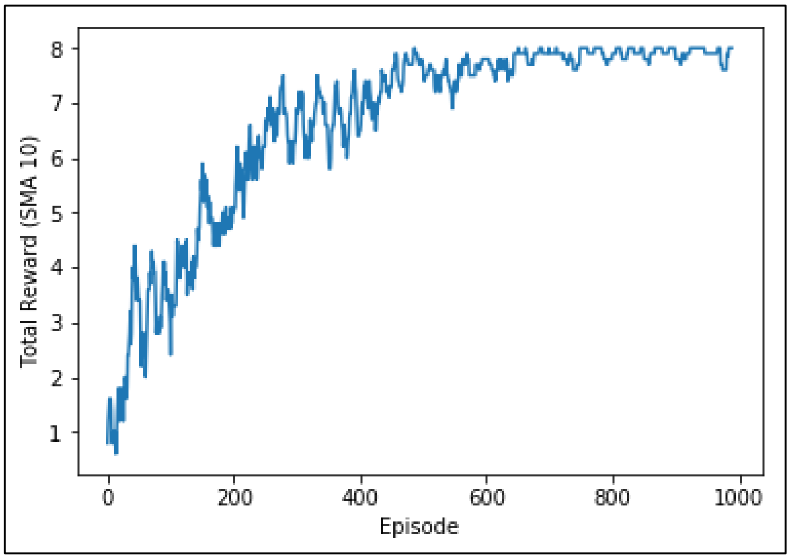
Episode vs Epsilon Graph

We can see from the graph that the epsilon ranges between 0 to 1 and over 1000 episodes the agent has played.

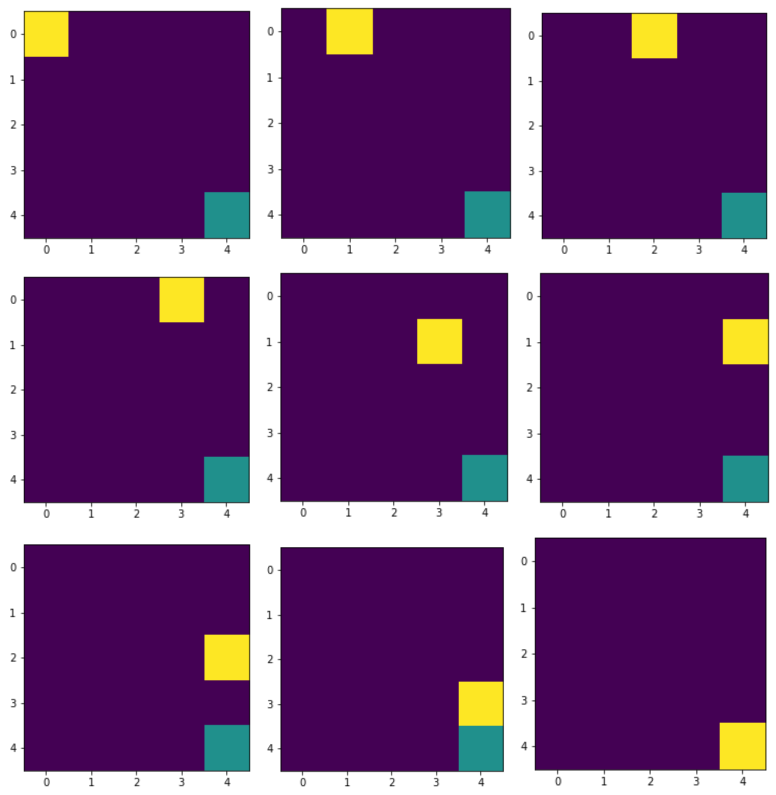


Episode vs Total Rewards Graph

This graph shows us that as the number of games played the reward in increasing and finally reaching the maximum in the total rewards which means that the agent has learnt and is optimally behaving according to the environment.



After the Training process running the agent in the environment shows that the agent has learnt and is optimally behaving with respect to the actions and has finally reached its goal.



Conclusion

Q-learning uses future rewards to influence the current action given a state and therefore helps the agent select best actions that maximize total reward. We can see from the above results that the graph of episode vs reward that the reward progressively increases over time and ultimately levels out at a high reward per episode value which indicates that the agent has learnt to maximize its total reward earned in an episode by behaving optimally at every state.

References

Reinforcement Learning – Introducing Goal Oriented Intelligence
https://www.youtube.com/playlist?list=PLZbbT5o_s2xoWNVdDudn51XM8lOuZ_Njv

Project Description : https://ublearns.buffalo.edu/bbcswebdav/pid-5242352-dt-content-rid-27556999_1/courses/2199_23170_COMB/Project_Description.pdf

Simple Reinforcement Learning: Q-learning :https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56