CSE-574: MACHINE LEARNING PROJECT 4
NAVIGATING THE CLASSIC 4X4 GRID ENVIRONMENT USING REINFORCEMENT LEARNING

Madhumitha Sivasankaran
Department of Computer Science
University at Buffalo
Buffalo, NY 14214
UB Id:50310290
msivasan@buffalo.edu

Abstract

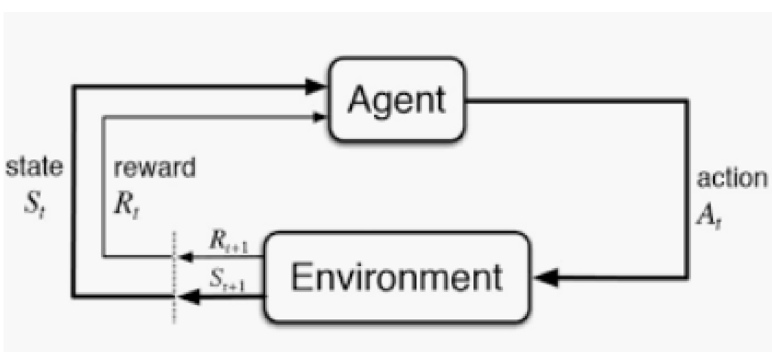
This project is about building a reinforcement learning agent to navigate the classic 4x4 grid-world environment. The agent is made to learn an optimal policy through Q-learning which will allow it to take actions to successfully reach the goal while avoiding obstacles. The working environment is provided for the learning agent. The three main tasks in this project is to

- Implement policy function: The agent randomly selects the 'epsilon' or 'exploration rate'.
- Update Q-table
- Implement the training algorithm

Finally, the agent will be capable of learning to navigate the environment and reach its maximum reward on average within a fair amount of episodes.

Introduction:

Reinforcement learning (RL) is learning by interacting with an environment. An RL agent learns from the consequences of its actions, rather than from being explicitly taught and it selects its actions on basis of its past experiences (exploitation) and also by new choices (exploration), which is essentially *trial and error* learning. The reinforcement signal that the RL-agent receives is a numerical reward, which encodes the success of an action's outcome, and the agent seeks to learn to select actions that maximize the accumulated reward over time.

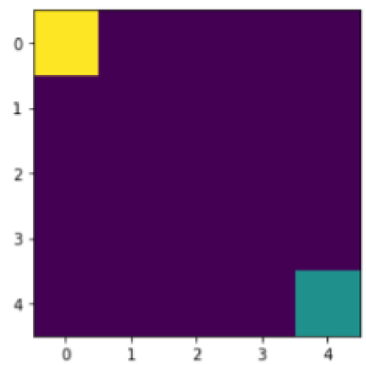


Q-learning is an off policy reinforcement **learning** algorithm that seeks to find the best action to take given the current state. It's considered off-policy because the **q-learning** function learns from actions that are outside the current policy, like taking random actions, and therefore a policy isn't needed.

Learning System:

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



The environment we provide is a basic deterministic $n \times n$ grid-world environment (the initial state for a 4×4 grid-world is shown in Figure 3) 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 $n \times n$ 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.

Q-LEARNING ALGORITHM:

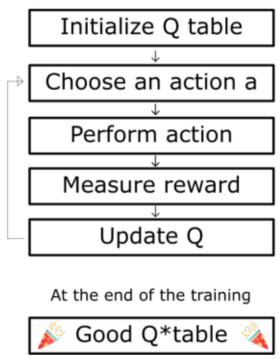
Essentially, Q-learning lets the agent use the environment's rewards to learn, over time, the best action to take in a given state. We have the reward table, that the agent will learn from. It does thing by looking receiving a reward for taking an action in the current state, then updating a *Q-value* to remember if that action was beneficial.
The values stored in Q-table are called a *Q-values*, and they map to a (state, action) combination. 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.
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 using the equation:

$$Q(state, action) \leftarrow (1-\alpha)Q(state, action) + \alpha\Big(reward + \gamma \max_a Q(next\ state, all\ actions)\Big)$$

- α (alpha) is the learning rate $(0<\alpha\leq 10<\alpha\leq 1)$ **-** Just like in supervised learning settings, α is the extent to which our Q-values are being updated in every iteration.
- γ (gamma) is the discount factor $(0\leq\gamma\leq 10\leq\gamma\leq 1)$ **-** 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, hence making it greedy.

We are assigning or updating, the Q-value of the agent's current *state* and *action* by first taking a weight $(1-\alpha)$ of the old Q-value, then adding the learned value. 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.
Basically, we are learning the proper action to take in the current state by looking at the reward for the current state/action combo, and the max rewards for the next state.
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**

The Q-table is a matrix where we have a row for every state and a column for every action. It's first initialized to 0, and then values are updated after training. The Q-table has the same dimensions as the reward table, but it has a completely different purpose.



HYPERPARAMETERS:

gamma is the **discount factor**. It quantifies how much importance we give for future rewards. It's also handy to approximate the noise in future rewards. Gamma varies from 0 to 1. If Gamma is closer to zero, the agent will tend to consider only immediate rewards. If Gamma is closer to one, the agent will consider future rewards with greater weight, willing to delay the reward.

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. Above we are taking the difference between new and old and then multiplying that value by the learning rate. This value then gets added to our previous q-value which essentially moves it in the direction of our latest update.

Episode: All *states* that come in between an initial-state and a terminal-state; for example: one game of Chess. The *Agent's* goal it to maximize the total *reward* it receives during an episode. In situations where there is no terminal-state, we consider an infinite episode. It is important to remember that different episodes are completely independent of one another.

OBSERVATIONS:

By tuning the above hyperparameters, we find the most optimal policy that fetches the maximum rewards while navigating the agent towards the target whilst avoiding obstacles. Below are the set of hyperparameters that were tuned and the corresponding plots indicating the loss and total rewards respectively.

1.

Episodes	**Learning rate**	**Epsilon**	**Gamma**	**Decay rate**
150	0.1	1.0	0.9	0.9

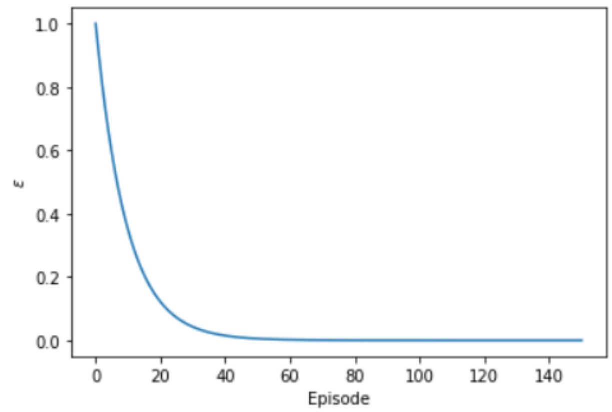


Figure 1:Epsilon vs Episodes plot

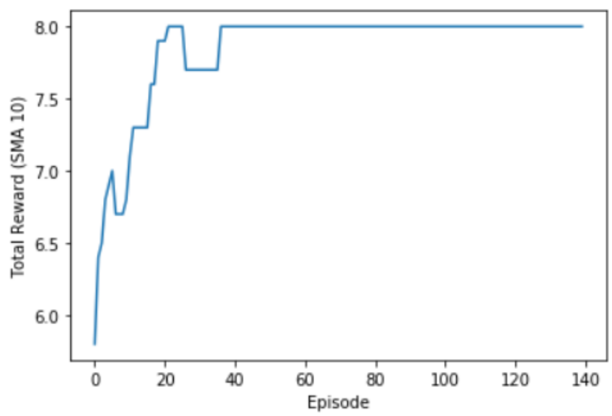


Figure 2: Total rewards vs episodes

2.

Episodes	Learning rate	Epsilon	Gamma	Decay rate
100	0.2	1.0	0.8	0.9

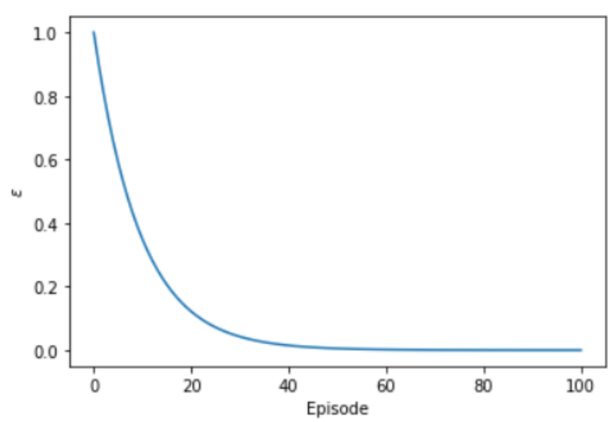


Figure 3:Epsilon vs Episodes plot

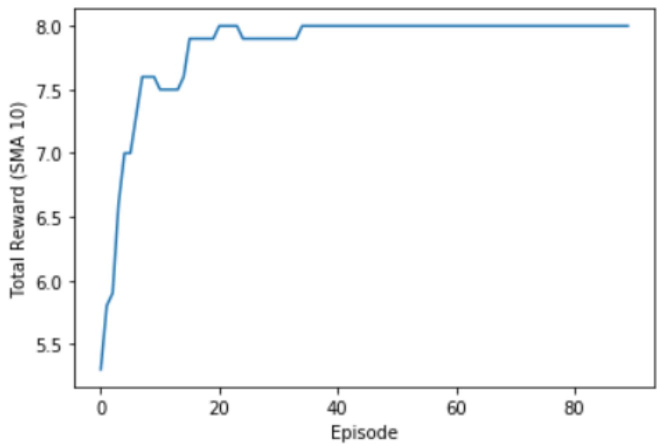
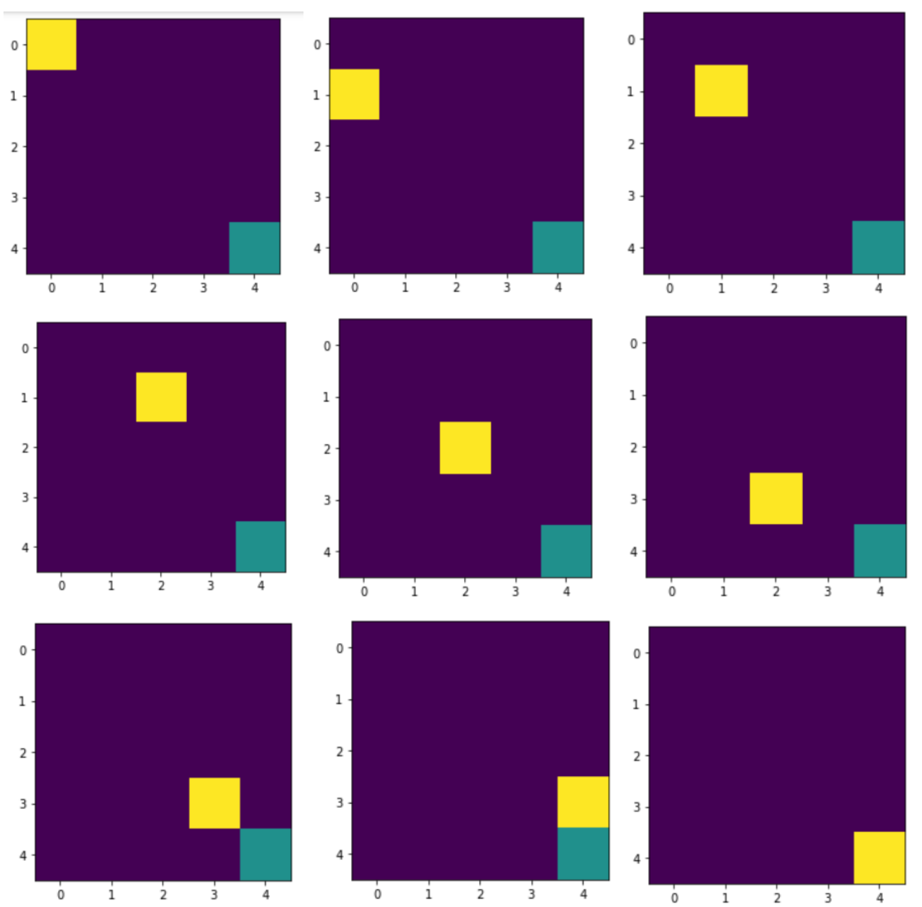


Figure 4: Total rewards vs Episode plot

Final path navigated by the learning agent (sequentially):



Conclusion:

Thus we have built a reinforcement learning agent to navigate across the given 4x4 grid environment. The agent was made to learn the most optimal policy through Q-Learning allowing it to take necessary steps to reach the goal by avoiding the obstacles. The results and observations made by tuning the set of hyperparameters were described above.

References:

1) https://skymind.ai/wiki/deep-reinforcement-learning

2) https://en.wikipedia.org/wiki/Reinforcement_learning

3) https://blog.floydhub.com/an-introduction-to-q-learning-reinforcement-learning/

4) https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/

5) https://www.geeksforgeeks.org/q-learning-in-python/

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