Name: Doaa Samir **ID:** 2103114

Autonomous Taxi Agent

This project focuses on developing an intelligent agent using the Q-Learning algorithm to solve a classic reinforcement learning task — the Taxi-v3 environment from OpenAI Gym. The goal is to train the agent to successfully navigate a grid-based world, pick up passengers, and drop them off at their destinations with optimal efficiency.

The project provides a practical demonstration of how reinforcement learning can be used to train agents in discrete environments where decision-making and delayed rewards are involved.

Environment Description

The Taxi-v3 environment represents a 5x5 grid where a taxi agent must transport a passenger from one location to another. The total number of possible states in the environment is 500, calculated from the combination of:

- 5 possible row positions
- 5 possible column positions
- 5 passenger locations (including "in the taxi")
- 4 fixed destination points

At any given state, the agent can choose from the following six discrete actions:

1. Move south
2. Move north
3. Move east
4. Move west
5. Pickup the passenger
6. Drop off the passenger

Each episode starts with the taxi and passenger randomly placed on the grid. The agent must learn to:

- Navigate to the passenger's location
- Pick up the passenger
- Drive to the specified destination
- Drop off the passenger

An episode ends successfully when the passenger is dropped off at the correct location.

Setting up the libraries

We start off by setting up the environment by importing libraries like Numpy which is used for handling large, multi-dimensional arrays and matrices in Python, along with a collection of mathematical functions to operate on them. And gym which is a toolkit for developing and comparing reinforcement learning algorithms. It provides environments for training RL models, such as games, simulations, or other interactive tasks where an agent can learn by interacting with its environment. And Matplotlib which is a library for creating static, interactive, and animated visualizations in Python. Specifically, it uses plt as an alias, which stands for pyplot. This module is commonly used for creating charts, graphs, and other types of plots to visualize data.

We create an environment for a reinforcement learning task called "Taxi-v3", which is one of the built-in environments in the Gym library. In the Taxi-v3 game, a taxi agent must pick up and drop off passengers at the correct locations on a grid, while following rules and avoiding penalties. We also install the Pygame library, which is used to create games and multimedia applications like animations or sound effects.

Simulation of Gym Environment with Random Actions

The purpose of this experiment was to simulate a reinforcement learning environment using the OpenAI Gym library and observe the behavior of an agent that takes actions at random. The agent interacts with the environment over multiple episodes, and the goal is to visualize its performance and evaluate the total rewards obtained in each run.

It initializes the number of episodes to 10 using a for loop. At the beginning of each episode, the environment is reset to its initial state. A visual representation of the environment is rendered and displayed using Matplotlib, and an empty list is created to store the animation frames for that episode.

During each episode, a while loop continues to execute until the environment signals that the episode is complete. Within this loop, the agent selects actions at random using the env.action_space.sample() function. These actions are fed into the environment using the env.step() function, which returns the next state, the reward for the action, whether the episode is done, and some additional information. The reward received at each step is added to the cumulative score. To provide visual feedback, the output screen is cleared and the new frame is rendered and saved.

Once the episode ends, the final score and episode number are printed. In the provided output, the simulation reached episode 9, and the total score accumulated by the agent was -911. This negative score is expected, given that the agent was acting randomly without any learning or optimization strategy in place.

This simulation provides a baseline understanding of how a reinforcement learning environment behaves when actions are not guided by a policy or learning algorithm. It also highlights the importance of incorporating smarter decision-making mechanisms to improve agent performance. The visuals and score tracking lay the groundwork for further experimentation using trained agents that can learn from experience.

Training using a Q-Learning Algorithm

We implement a reinforcement learning algorithm called Q-learning, which enables an agent to learn optimal behavior through trial and error. In this specific case, the environment is most likely a discrete setting like the "Taxi-v3" environment from OpenAI Gym, where an agent must pick up and drop off passengers efficiently.

At the beginning, the number of available actions and states in the environment is retrieved using env.action_space.n and env.observation_space.n. These values are used to initialize a Q-table with all zeros. The Q-table is a two-dimensional array where each row represents a state and each column represents an action. The table will be updated over time to store the agent’s learned estimates of the best possible rewards for taking specific actions in given states.

Several hyperparameters are defined to control the learning process. The simulation will run for 10,000 episodes, with each episode allowing the agent to take up to 100 steps. The learning_rate determines how quickly the Q-table is updated with new information, while the discount_rate controls how much the agent values future rewards compared to immediate ones. The exploration_rate begins at 1, meaning the agent starts by taking entirely random actions. Over time, this rate will decay gradually, encouraging the agent to rely more on its learned experiences stored in the Q-table rather than guessing.

For each episode, the environment is reset, and the agent repeatedly interacts with it until the episode is done or the step limit is reached. At every step, the agent decides whether to explore or exploit. If a randomly generated number is greater than the current exploration rate, the agent chooses the best-known action based on the current Q-table; otherwise, it selects a random action. The environment then returns the next state, a reward, and a flag indicating whether the episode has finished.

After receiving this feedback, the Q-table is updated using a standard Q-learning formula. This update blends the previous estimate of the value with a new estimate based on the received reward and the highest predicted future reward in the next state. Over time, this process allows the agent to refine its behavior and increasingly prefer actions that lead to higher long-term rewards.

Once the episode ends, the exploration rate is adjusted using an exponential decay function. This means that, gradually, the agent shifts from random exploration toward exploiting its learned strategy. The total reward from the episode is stored in a list, allowing for later analysis of performance over time.

At the end of training, a message is printed to confirm that learning is complete. So we effectively demonstrates the core principles of reinforcement learning—exploration, exploitation, reward feedback, and value updates—using a simple but powerful algorithm.

Evaluate the performance of the agent during Training

```
Average per thousand episodes
1000 :  -251.0469999999999
2000 :  -37.22500000000005
3000 :  1.5779999999999974
4000 :  5.789999999999965
5000 :  6.83999999999997
6000 :  7.380999999999965
7000 :  7.230999999999952
8000 :  7.466999999999953
9000 :  7.473999999999973
10000 :  7.305999999999951
```

After training the agent using Q-learning for 10,000 episodes, this helps to summarize how the agent's performance changed over time. Instead of examining rewards from every single episode, which could be too detailed and inconsistent, the code groups the rewards into blocks of 1,000 episodes each. It calculates the average reward for each block, making it easier to observe learning trends.

The output begins with very low average rewards. In the first 1,000 episodes, the average reward was approximately -251, which shows that the agent was performing poorly—mostly taking random actions and failing to complete the tasks effectively. However, as the episodes progress, the agent gradually improves. By the second block episodes 1001 to 2000, the average reward increases significantly to about -37, showing early signs of learning.

From the third block onward, the average rewards turn positive. By episode 3000, the agent achieves an average of about 1.57, and it continues to improve. The average reward peaks in later episodes, reaching around 7.4 between episodes 8000 and 9000. These positive and relatively stable scores toward the end indicate that the agent has successfully learned a strategy that works well in the environment.

Testing the agent after Training

The agent now uses what it has learned to try and perform well in the environment by always choosing the best action it knows. The test is repeated for 30 episodes to observe how consistently the agent performs.

At the beginning of each episode, the environment is reset to its initial state, and a message is printed to show which episode is running. The initial visual frame of the environment is rendered and shown using Matplotlib, giving a visual cue of the starting point. An empty list called img is created to store each frame of the episode, which could later be used to create animations or videos.

Inside each episode, the agent takes up to a set number of steps (defined earlier by max_steps_per_episode). In each step, it chooses an action by selecting the one with the highest value in the Q-table for the current state—this means it is no longer exploring but is exploiting its learned strategy. The action is executed in the environment using env.step(action),

which returns the new state, the reward received, a done flag to indicate if the episode has ended, and some additional info.

Each time an action is taken, the output display is cleared to keep the screen tidy, and the new rendered frame is saved to the img list. The step number and reward are printed to track the agent’s behavior and progress during that episode.

If the episode ends (i.e., the goal is reached or the time runs out), the code checks the reward. If the reward is 20, it means the agent successfully completed the task (such as picking up and dropping off the passenger in the Taxi environment), and a message saying "Reached Goal" is shown. Otherwise, if the reward is lower, it prints "Failed" to indicate the agent did not complete the task successfully. The final frame is also saved before breaking out of the loop.