Lab 6. EARIN

Variant 1

Oleg Kim

Seniv Volodymyr

29.05.2023

Theoretical background:

CartPole-v1 is a simulation of a cart and pole system, where the goal is to balance the pole on the cart for as long as possible by applying appropriate control actions.

The environment is episodic, meaning each episode starts with an initial state and ends when a termination condition is reached. The objective is to maximize the cumulative reward obtained over multiple episodes.

The state space of CartPole-v1 is continuous and consists of four variables: cart position, cart velocity, pole angle, and pole angular velocity. These continuous state variables provide information about the current configuration and dynamics of the cart and pole system.

The action space in CartPole-v1 is discrete and consists of two possible actions: pushing the cart to the left or pushing it to the right. The agent selects one of these actions at each time step to control the movement of the cart and attempt to balance the pole.

At each time step, the agent receives an observation that includes the current state of the environment. Based on this observation, the agent selects an action according to its policy. The environment then transitions to a new state based on the selected action, and the agent receives a reward that depends on the resulting state and action.

In CartPole-v1, the agent receives a reward of +1 for every time step that the pole remains upright. The episode terminates if any of the following conditions are met: the pole angle exceeds ±12 degrees or the cart position moves outside the range of ±2.4.

In this code, Q-learning is used to solve the OpenAI CartPole problem. The CartPole problem is a classic reinforcement learning problem where the agent (a cart) needs to balance a pole on top of it. The agent can apply forces to the cart in two directions (left or right) to maintain the pole's balance. Q-learning is a model-free reinforcement learning algorithm that learns an action-value function (Q-function) to make decisions based on the expected cumulative reward. The Q-function represents the expected reward for taking a specific action in a given state. The goal of Q-learning is to iteratively update the Q-values based on the observed rewards and learn an optimal policy that maximizes the expected cumulative reward.

In summary, Q-learning is used to learn an optimal policy for the CartPole problem by iteratively updating the Q-values based on observed rewards. The learned policy is then applied to simulate the agent's behavior and evaluate its performance.

Task description:

Create an implementation of the Q-Learning algorithm to solve a toy Reinforcement Learning problem. Usethe environment provided from OpenAI gym library. The original gym library is no longer updated, however,there is a continued development on a fork of gym called gymnasium. Please use gymnasium for this exercise.Use the following environments for each lab variants:1.

Variant 1: CartPole. Use "CartPole-v1"

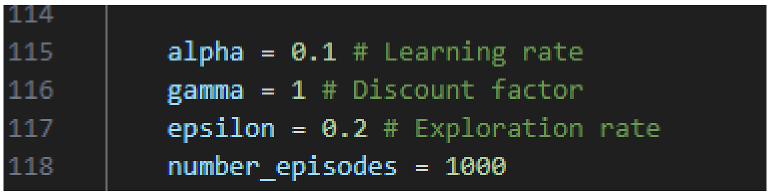
Launch instruction:

Before launching the application, you must install following packages:

- Pygame
- Numpy
- Gymnasium
- Matplotlib
- Moviepy

After you have installed these packages, you can start the code and wait until all the episodes you have provided will execute.

You can change the number of episodes and other variables in this part of code inside "main" function:



```python

    alpha = 0.1 # Learning rate
    gamma = 1 # Discount factor
    epsilon = 0.2 # Exploration rate
    number_episodes = 1000
```

Solution:

def __init__(self, env, alpha, gamma, epsilon, number_episodes, num_bins, lower_bounds, upper_bounds):

The "__init__" method is the constructor of the QLearning class. It initializes the instance variables of the class based on the provided parameters.

- env: The environment object representing the CartPole-v1 environment.
- alpha: The learning rate or the step size parameter for updating the Q-values.
- gamma: The discount factor, determining the importance of future rewards in the Q-value updates.
- epsilon: The exploration rate or the probability of taking a random action instead of exploiting the learned Q-values.
- number_episodes: The number of episodes to train the agent.
- num_bins: A list containing the number of bins or discretization levels for each state variable in the environment.
- lower_bounds: A list containing the lower bounds for each state variable in the environment.
- upper_bounds: A list containing the upper bounds for each state variable in the environment.
- sum_rewards_episode: A list to store the sum of rewards obtained for each episode during training.
- Q_matrix: A 3-dimensional numpy array representing the Q-values for each state-action pair. It is initialized with random values between 0 and 1.

The purpose of this method is to set up the necessary variables and data structures for the Q-learning algorithm and to initialize the Q-matrix with random values.

def return_index_state(self, state):

The return_index_state method takes a continuous state as input and returns a tuple of indices corresponding to the discretized state based on binning.

- state: The continuous state of the environment.

The method performs the following steps:

- It initializes an empty list called indices to store the indices for each state variable.
- It iterates over each state variable in the state.

- For each state variable, it maps the continuous value to a discrete index based on binning. It uses the np.digitize function to find the bin index where the state variable falls within the specified lower and upper bounds.
- It subtracts 1 from the bin index to ensure that the indices start from 0.
- It takes the maximum of the calculated index and 0 to handle the case where the state variable is less than the lower bound.
- It appends the calculated index to the indices list.
- Finally, it returns the tuple of indices.

The purpose of this method is to convert the continuous state representation of the environment into a discretized form, which can be used as indices to access the corresponding Q-values in the Q-matrix.

def select_action(self, state, index):

The select_action method is responsible for selecting an action to take in a given state based on the current index or episode number.

- state: The current state of the environment.
- index: The current episode number or index.

The method performs the following steps:

- It first checks if the index is less than 500. If so, it returns a random action. This random action selection is used during the initial episodes to encourage exploration and gather more diverse experiences.
- If the index is greater than 7000, it applies an epsilon decay strategy. It reduces the value of self.epsilon by multiplying it with 0.999. This decay over time gradually decreases the exploration rate, allowing for more exploitation of learned Q-values as the training progresses.
- It generates a random number between 0 and 1 using np.random.random().
- If the random number is less than self.epsilon, it returns a random action. This represents the exploration phase where random actions are chosen to discover new states and potentially better Q-values.
- If the random number is greater than or equal to self.epsilon, it selects the action with the highest Q-value for the current state. It uses np.where to find the indices of the maximum Q-values for the current state in the Q-matrix. It then chooses one of those indices randomly using np.random.choice. This exploitation phase aims to select the action that is estimated to have the highest Q-value based on the learned Q-matrix.

The select_action method balances between exploration and exploitation by gradually reducing the exploration rate (epsilon) over time and using a random action or the action with the highest Q-value based on the current state and Q-matrix.

def simulate_episodes(self):

The simulate_episodes method is responsible for running the Q-learning algorithm for a specified number of episodes.

The method performs the following steps:

- It iterates over the range of self.number_episodes to simulate each episode.
- For each episode, it initializes an empty list rewards_episode to store the rewards obtained during that episode.
- It resets the environment self.env and obtains the initial state state_s using self.env.reset(). The state is converted to a list format for consistency.
- The method enters a loop until a terminal state is reached.
- Inside the loop, it calculates the index of the current state state_s using the self.return_index_state method.
- It selects an action action_a using the self.select_action method, passing the current state state_s and the current episode index index_episode.
- The environment is stepped forward using self.env.step(action_a), which returns the next state state_s_prime, the reward obtained reward, whether the state is terminal terminal_state, and additional information (ignored in this code).
- The reward obtained in the current step is appended to the rewards_episode list.
- The next state state_s_prime is converted to a list format and its index is calculated using self.return_index_state.
- The maximum Q-value for the next state Q_max_prime is determined by taking the maximum value from the Q-matrix at the corresponding state index.
- If the state is not terminal (terminal_state is False), the Q-value for the current state-action pair is updated using the Q-learning formula: Q(s,a) = Q(s,a) + alpha * (reward + gamma * Q_max_prime - Q(s,a)).
- If the state is terminal, the Q-value update is simplified to: Q(s,a) = Q(s,a) + alpha * (reward - Q(s,a)).
- The current state state_s is updated to the next state state_s_prime.
- After the episode is completed, the sum of rewards for that episode is calculated using np.sum(rewards_episode).

- The sum of rewards is appended to the self.sum_rewards_episode list, storing the cumulative rewards for all episodes.

The simulate_episodes method runs the Q-learning algorithm by iteratively simulating episodes, updating the Q-values based on the rewards obtained, and storing the cumulative rewards for each episode.

def simulate_learned_strategy(self):

The simulate_learned_strategy method is responsible for simulating the learned strategy based on the Q-values obtained from the Q-learning algorithm.

The method performs the following steps:

- It creates a new environment env1 using gym.make('CartPole-v1', render_mode='human'). This environment is used to visually render the simulation.
- The current state current_state is obtained by resetting the environment using env1.reset(). The second value returned by reset() (ignored in this code) represents additional information about the initial state.
- The environment is rendered using env1.render(), allowing visual representation of the simulation.
- The number of time steps time_steps is set to 1000, which determines the duration of the simulation.
- An empty list obtained_rewards is created to store the rewards obtained during the simulation.
- The method enters a loop that iterates over time_steps.
- Inside the loop, it selects an action action_in_state_s based on the learned Q-values. It chooses the action with the highest Q-value for the current state by finding the indices of the maximum Q-values in the Q-matrix at the current state index, and randomly selecting one of those indices.
- The environment is stepped forward using env1.step(action_in_state_s), which returns the next state current_state, the reward obtained reward, whether the state is terminal terminated, and additional information (ignored in this code).
- The reward obtained in the current step is appended to the obtained_rewards list.
- A small delay of 0.05 seconds is introduced using time.sleep(0.05) to visualize the simulation.
- If the state is terminal (terminated is True), an additional delay of 1 second is introduced using time.sleep(1) before breaking the loop.

- After the simulation is completed, the obtained_rewards list and the env1 environment are returned.

The simulate_learned_strategy method simulates the learned strategy based on the learned Q-values. It selects actions based on the Q-values, steps through the environment, and collects the rewards obtained during the simulation. The method also returns the list of obtained rewards and the environment for further analysis or visualization.

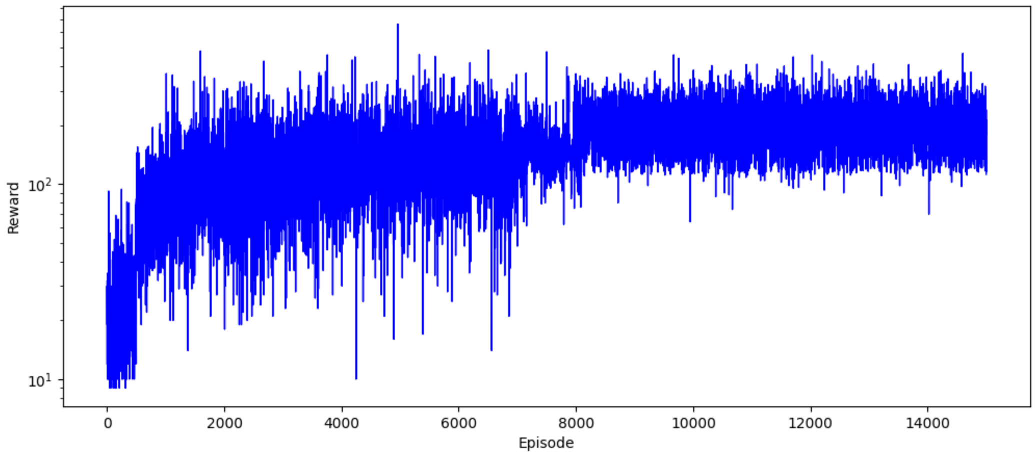
Main:

- It imports the necessary libraries and modules.
- It creates an environment env using gym.make('CartPole-v1'). This environment represents the CartPole-v1 task.
- The initial state of the environment is obtained using env.reset(), and the state is assigned to the variable state. The second value returned by reset() (ignored in this code) represents additional information about the initial state.
- The upper and lower bounds of the state variables are adjusted to customize the observation space. Specifically, the bounds for the cart velocity and pole angle velocity are modified.
- The number of bins is specified for each state variable to discretize the continuous state space.
- The learning rate alpha, discount factor gamma, exploration rate epsilon, and the number of episodes are set.
- An instance of the QLearning class is created with the specified parameters: env, alpha, gamma, epsilon, number_episodes, num_bins, lower_bounds, and upper_bounds.
- The simulate_episodes method of the QLearning instance is called to perform the Q-learning algorithm and simulate the episodes.
- The simulate_learned_strategy method of the QLearning instance is called to simulate the learned strategy based on the learned Q-values. The obtained rewards and the environment are stored in obtained_rewards_optimal and env1, respectively.
- A plot is created to visualize the sum of rewards for each episode using plt.plot. The rewards are retrieved from the QLearning instance's sum_rewards_episode attribute.
- The plot is displayed using plt.show().
- The environment env1 is closed using env1.close().
- The sum of rewards obtained from the learned strategy is calculated using np.sum(obtained_rewards_optimal).

Result:

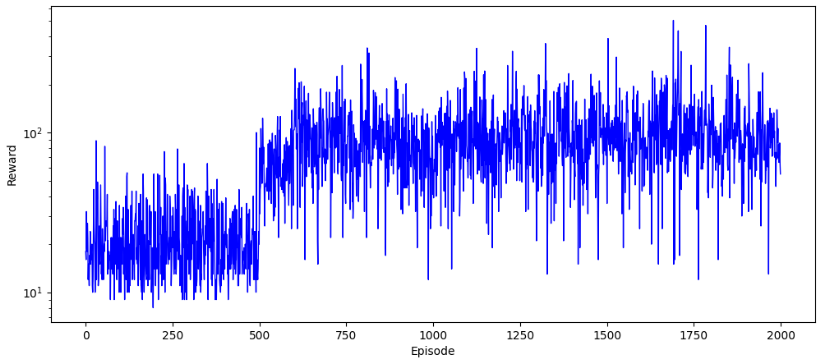
1. For:

alpha = 0.1 # Learning rate
gamma = 1 # Discount factor
epsilon = 0.2 # Exploration rate
number_episodes = 15000



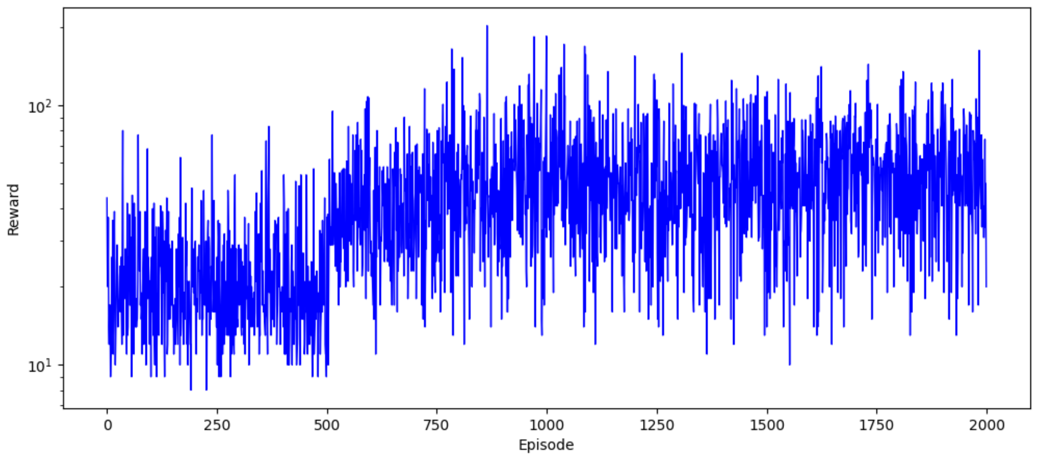
2. For:

alpha = 0.1 # Learning rate
gamma = 1 # Discount factor
epsilon = 0.2 # Exploration rate
number_episodes = 2000



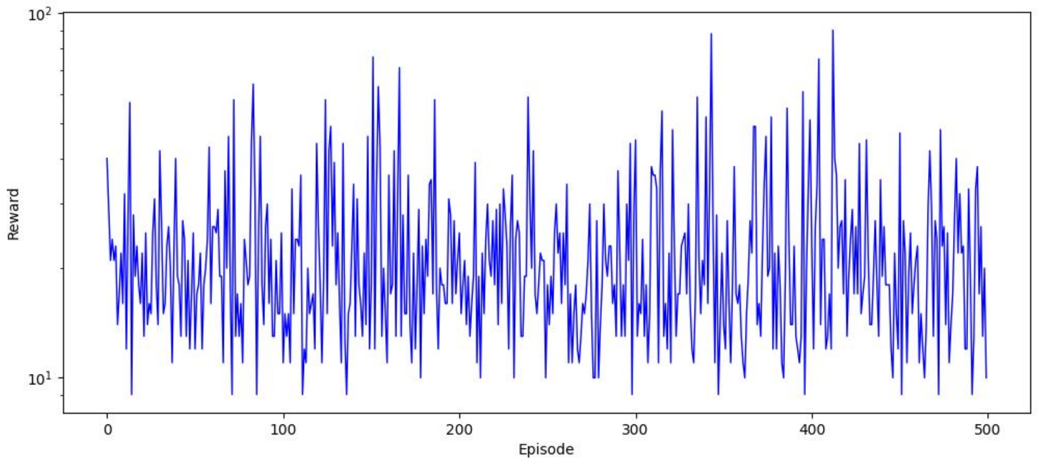
3. For:

alpha = 0.2 # Learning rate
gamma = 0.9 # Discount factor
epsilon = 0.5 # Exploration rate
number_episodes = 2000



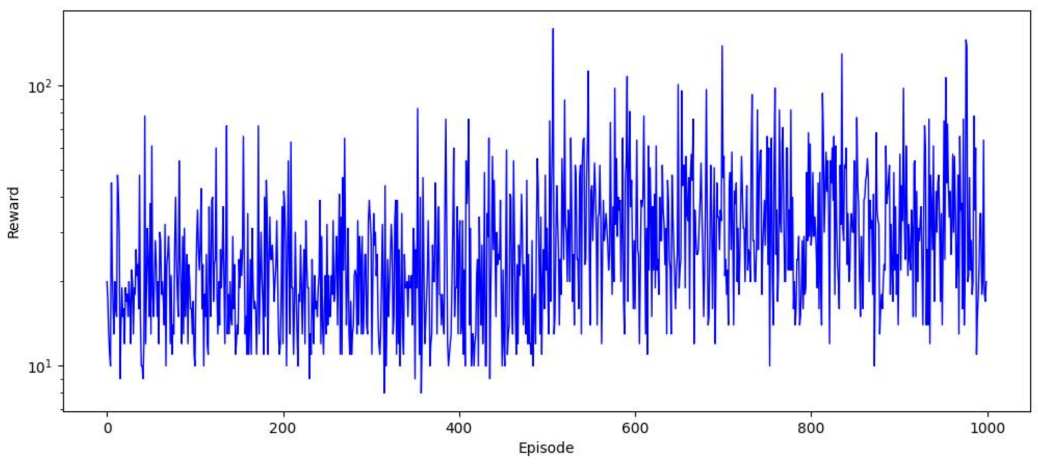
4. For:

alpha = 0.01 # Learning rate
gamma = 0.95 # Discount factor
epsilon = 0.1 # Exploration rate
number_episodes = 500



5. For:

alpha = 0.01 # Learning rate
gamma = 0.95 # Discount factor
epsilon = 0.1 # Exploration rate
number_episodes = 1000



What could be improved:

Exploration-Exploitation Tradeoff: The current implementation uses a simple epsilon-greedy policy for exploration. However, you can experiment with different exploration strategies, such as decaying epsilon or using more sophisticated exploration techniques like epsilon-decay or Boltzmann exploration. These methods can help balance exploration and exploitation more effectively.

Learning Rate Schedule: Instead of using a fixed learning rate (alpha), you can explore using a learning rate schedule that gradually reduces the learning rate over time. This can help the algorithm converge faster and achieve better performance.
Parameter Tuning: The performance of the Q-learning algorithm can be sensitive to its hyperparameters (e.g., learning rate, discount factor, exploration rate). You can try different combinations of hyperparameters and use techniques like grid search or random search to find the optimal set of parameters for better performance.

Optimized Libraries: Consider using optimized libraries or frameworks specifically designed for reinforcement learning, such as OpenAI Baselines or Stable Baselines. These libraries provide pre-implemented algorithms, advanced exploration strategies, and optimized implementations, which can improve performance and simplify the code.

Conclusion:

Based on the observation that the sum of rewards increases gradually during episodes, we can draw the following conclusions:

Learning Progress: The increasing sum of rewards indicates that the agent is learning and improving its performance over time. This suggests that the agent is gradually discovering more optimal strategies to maximize its rewards in the CartPole environment.

Initial Stabilization: The initial stabilization of the pole in the animation demonstrates that the agent has learned to perform actions that can maintain balance and keep the pole upright for a certain period. This indicates that the agent has acquired some level of control and understanding of the environment dynamics.

Limited Long-Term Strategy: The eventual fall of the pole at the end of the animation suggests that the learned policy may not be able to sustain balance indefinitely. This implies that the agent might not have developed a comprehensive long-term strategy to handle the increasing instability as the episode progresses.

In conclusion, the increasing sum of rewards demonstrates learning progress, while the eventual fall of the pole indicates the need for further refinement of the agent's policy to sustain balance for longer durations. The results suggest that the agent is learning and improving, but additional optimization and experimentation are required to achieve more stable and prolonged balancing of the pole.