Introduction

In this project, a deep Q-learning algorithm is used to train an agent to play the LunarLander-v2 game from Gymnasium. LunarLander-v2 is a classic control task in which the agent is required to land a spacecraft safely on the moon's surface while controlling its speed and orientation. The game is considered solved when the agent successfully lands the spacecraft.

Deep Q-Learning

Deep Q-Learning is a reinforcement learning algorithm that combines Q-Learning with deep neural networks to approximate the Q-function. The Q-function represents the expected reward of taking a particular action in a given state. The objective of the DQN algorithm is to learn an optimal policy by iteratively updating the Q-function estimates based on the Bellman equation. The DQN algorithm uses a replay buffer to store the transitions experienced by the agent and uses experience replay to decorrelate the samples.

Input and Output

Input: The game environment state, which consists of 8 continuous values representing the position, orientation, velocity, and angular velocity of the spacecraft, plus 2 discrete values representing the presence or absence of the left and right engines firing.

Output: The output of the policy network is a vector of size 4, representing the Q-values for each possible action. The action with the highest Q-value is selected.

Parameters

1. BATCH_SIZE: The number of transitions sampled from the replay buffer
2. GAMMA: The discount factor as mentioned in the previous section
3. EPS_START: The starting value of epsilon
4. EPS_END: The final value of epsilon
5. EPS_DECAY: Controls the rate of exponential decay of epsilon, higher means a slower decay
6. TAU: The update rate of the target network
7. LR: The learning rate of the AdamW optimizer
8. n_actions: The number of possible actions, which is 4 for this game.
9. n_observations: The number of state observations, which is 8 for this game.
10. num_steps: The number of steps to take while testing the game
11. steps_per_frame: The number of steps to take per frame while testing the game
12. Transition: A named tuple representing a single transition in the replay memory, consisting of the current state, the action taken, the next state, and the reward received.
13. ReplayMemory: A class representing the replay buffer used by the DQN algorithm to store and sample transitions.
14. policy_net: The policy network used to select actions during training.
15. target_net: A copy of the policy network used to calculate target Q-values during training.
16. optimizer: The optimizer used to update the weights of the policy network.
17. steps_done: A counter that keeps track of the number of steps taken by the agent.

Code

1. The code first imports the necessary libraries and creates an instance of the LunarLander-v2 game environment.

2. It then tests the game by taking random actions and rendering the game for a fixed number of steps.
3. The DQN class defines the policy network used by the agent.
4. The ReplayMemory class represents the replay buffer, which is implemented using a deque.
5. The select_action function implements the epsilon-greedy strategy for selecting actions during training.
6. Finally, the plot_durations function is used to plot the rewards and durations of each episode during training.

REWARDS:

The rewards for each action taken in the LunarLander-v2 game are:

- +100 for a successful landing
- -100 for a crash
- -0.3 for every unit of time step taken
- +10 for legs touching ground
- -0.01 penalty for using the engine

The goal of the game is to land the lunar lander on the landing pad with minimal penalty.

The optimize_model Function

The **optimize_model** function in the above code is responsible for updating the weights of the Q-Network. The function implements the Q-Learning algorithm, which is a reinforcement learning algorithm for learning optimal policies in an environment.

Here's a brief explanation of what the **optimize_model** function does:

1. The function first checks if there are enough transitions in the replay memory to begin training. If there are not enough transitions, the function simply returns.
2. If there are enough transitions, the function samples a batch of transitions from the replay memory. The batch size is defined by the **BATCH_SIZE** constant.
3. For each transition in the batch, the Q-Network is used to calculate the Q-values for the current state and next state. The Q-values for the next state are used to calculate the target Q-value for the current state, using the Bellman equation.
4. The Q-Network is then trained on the batch of transitions, with the loss function defined as the mean squared error between the predicted Q-values and the target Q-values.
5. The optimizer is used to update the weights of the Q-Network to minimize the loss.
6. The function then updates the **EPSILON** value according to the **EPSILON_DECAY** constant.

Overall, the **optimize_model** function is responsible for updating the Q-Network weights to improve the agent's ability to select actions that lead to higher rewards