IDATT2502 Machine Learning: Assignment 08

This write-up isn't part of the assignment, but written out of a need to explain and understand the concepts and workings of both the learning method, Q-Learning, and code implementation of it.

Reinforcement Learning and Q-Learning Overview

Reinforcement Learning (RL) is a type of **Machine Learning** (ML) where an agent is put in an environment and learns how to interact with it to maximize a cumulative reward through its actions in and upon the environment. The agent observes the state of the environment, takes actions, receives rewards, and then adjusts its behavior over time, based on the previous rewards, in order to improve the chances for future rewards. A key characteristic of RL is that the agent must explore the environment, while also exploiting known information, in order to learn how to maximize rewards.

Q-Learning is a specific RL algorithm, which aims to learn the optimal action-value function, known as the Q-Function. The function, $Q(s, a)$, represents the expected cumulative rewards of taking action aaa in state sss, and then following the optimal policy thereafter. The agent will update its Q-values iteratively using the Bellman equation, balancing immediate rewards with the discounted future rewards. Discounted future rewards mean that the agent values future rewards less than immediate ones. This is controlled by the discount factor γ, typically between 0.9 and 0.99, which gradually reduces the importance of rewards that are received later in the future. This method is model-free, meaning it doesn't need to know the dynamics of the environment in advance, but rather it will learn how the environment works through trial and error.

Implementing Q-Learning in Python

To exemplify the implementation, the Lunar Lander program can be used as an example.

The `LunarLanderAgent` class implements the Q-Learning algorithm for the Lunar Lander environment. It uses continuous state variables such as the position and velocity of the lander. To handle this, the agent will discretize the continuous state space into bins.

A **bin** is a range of values grouped together to simplify a continuous space into discrete segments. In this context, it divides continuous ranges like position or velocity into fixed intervals. This allows it to store and update Q-values in a table. A Q-value represents the expected reward of taking a specific action from a given state.

State Discretization

As mentioned above, the Lunar Lander environment provides continuous states. Discretizing these states into bins is important for Q-learning, since the Q-tables store values for discrete `state-action` pairs.

Example:

```
def discretize_state(self, obs, bins=10):
    state_bins = [np.linspace(low, high, bins) for low, high in
zip(self.env.observation_space.low, self.env.observation_space.high)]
    discretized_obs = tuple(np.digitize(ob, bins) for ob, bins in zip(obs, state_bins))
    return discretized_obs
```

This method ensures that the agent can handle the continuous state space by mapping the space and position of the lander into smaller and more manageable sets of discrete states.

Action Selection: Exploration vs. Exploitation

The agent uses an epsilon-greedy strategy to balance exploration and exploitation. Epsilon-greedy means that the agent will either select a random action (explore) or choose the best-known action (exploit), depending on a probability controlled by epsilon:

$$\pi(s) = \begin{cases} argmax_a Q(s,a) & \text{with probability } 1-\epsilon) \\ \text{random action} & \text{with probability } \epsilon \end{cases}$$

With probability ϵ, the agent selects a random action from its repertoire to explore the environment. Alternatively, it exploits its previously learned Q-values by selecting the action with the highest value, i.e., one that has provided the most benefit/reward in earlier attempts. This is implemented in the `get_action` method:

```
if np.random.rand() <= self.epsilon:
    return self.env.action_space.sample()  # Random action (explore)
else:
    return int(np.argmax(self.q_values[obs]))  # Best action (exploit)
```

This strategy of selecting actions will help the agent discover better and more rewarding policies by sometimes exploring unknown and unfamiliar states and actions, while also drawing upon previously acquired knowledge to maximize the rewards.

Q-Value Update: The Bellman Equation

After taking an action and receiving a reward, the agent updates its Q-value for the current state-action pair using the Bellman equation. This equation balances the immediate reward from the action just taken with the expected future rewards from the next state:

```
future_q_value = (not terminated) * np.max(self.q_values[next_obs])
temporal_difference = reward + self.discount_factor * future_q_value - self.q_values[obs]
[action]
self.q_values[obs][action] += self.lr * temporal_difference
```

In this update method, the agent adjusts its Q-value based on the temporal difference between the current estimate and the better-informed estimate from the next state. It learns from the next state by using the maximum Q-value from that state to inform the current state's action-value. Even though the next state hasn't happened yet, it uses learned Q-values from that state to make an informed guess. This process helps the agent gradually improve its policy by learning which actions lead to better outcomes.

Epsilon Decay

Over time, the agent decreases its exploration rate (epsilon), making it focus more on exploiting what it has learned rather than continuing to explore random actions. This means that as the agent acquires more knowledge about the environment, it will build an understanding of how to exploit it, which will make random exploration less rewarding than using the high-reward actions patterns in its knowledge base.

This is controlled by the `decay_epsilon` method:

```
self.epsilon = max(self.final_epsilon, self.epsilon - self.epsilon_decay)
```

By reducing exploration gradually as the agent learns the environment, it can start out with exploring in the early stages of learning, and then move to focusing more on exploiting the knowledge gained to increase its performance toward the later stages of the learning process.

Summary: How the Code Works in Practice

As the agent interacts with the Lunar Lander environment, it starts out with random actions. This is due to the high epsilon value and the fact that it is entirely naïve to the environment. It will explore different ways to land the spacecraft, learning from its experience by updating the Q-values based on the rewards (results) of each action. As the agent accumulates experience (quantified as rewards for actions), the epsilon value will decay, and the agent will increasingly rely on the knowledge stored in its Q-table to make better and more informed decisions, aiming to minimize crashes and maximize successful landings.

In summary, the code implements the core components of Q-Learning:

- State discretization.
- Action selection via epsilon-greedy strategy.
- Q-value updates using the Bellman equation.
- Epsilon decay to balance exploration and exploitation over time.

Through repeated interactions—trial and error—with rewards based on the result, the agent learns an optimal policy for landing the Lunar Lander.