Lab 6: Reinforcement Learning - Q-Learning Implementation Report

Student ID: [210042175]

August 30, 2025

1 Introduction(Task-6)

This report details the implementation of a Q-learning agent for the reinforcement learning tasks outlined in Lab 6. The primary objective was to develop a Q-learning algorithm capable of learning optimal policies through interaction with environments like Gridworld, Crawler, and Pacman. The focus was on implementing key methods in `qlearningAgents.py`, including `computeValueFromQValues`, `computeActionFromQValues`, `getAction`, and `update`, to enable the agent to learn from trial-and-error interactions. This report discusses the problem, the implemented solution, key findings, challenges, hyperparameter exploration, and additional insights gained during the process.

2 Problem Analysis

The task required implementing a Q-learning agent that learns an optimal policy without prior knowledge of the environment's Markov Decision Process (MDP). Unlike value iteration, which computes a policy using a known MDP model, Q-learning is a model-free approach that updates Q-values based on observed state-action-reward transitions. The challenge was to ensure the agent correctly handles exploration versus exploitation, ties in action selection, and unseen state-action pairs. The implementation needed to support multiple environments (Gridworld, Crawler, Pacman), each with varying state spaces and action sets, requiring robust and generalizable code. A key nuance was ensuring that actions not yet encountered by the agent (with Q-value 0) could be considered optimal if all known actions had negative Q-values.

3 Solution Explanation

The Q-learning agent was implemented in `qlearningAgents.py` by completing the required methods. Below is an overview of the approach and key decisions:

- **Initialization**: A `util.Counter` object was used to store Q-values for state-action pairs, initialized to 0 for unseen pairs, simplifying the handling of new states and actions.

- **`getQValue`**: This method retrieves the Q-value for a given state-action pair from the `qValues` counter, returning 0.0 for unseen pairs.
- **`computeValueFromQValues`**: This computes the maximum Q-value over all legal actions for a state, returning 0.0 for terminal states with no legal actions.
- **`computeActionFromQValues`**: This selects the action with the highest Q-value, breaking ties randomly using `random.choice` to ensure better exploration behavior.
- **`getAction`**: This implements the epsilon-greedy strategy, choosing a random action with probability `epsilon` or the best action otherwise, using `util.flipCoin`.
- **`update`**: The Q-value update follows the Q-learning formula:

  $$Q(s,a) \leftarrow (1-\alpha)\cdot Q(s,a) + \alpha \cdot (r + \gamma \cdot \max_{a'} Q(s',a'))$$

  where `alpha` is the learning rate, `gamma` is the discount factor, `r` is the reward, and `s'` is the next state.

Key decisions included using `util.Counter` for efficient Q-value storage and ensuring random tie-breaking in action selection to avoid deterministic biases. The complexity of the implementation is $O(|A|)$ per update for a state with $|A|$ legal actions, as it requires iterating over actions to compute the maximum Q-value.

Here is a snippet of the `update` method for clarity:

```python
def update(self, state, action, nextState, reward):
    oldQ = self.getQValue(state, action)
    nextValue = self.computeValueFromQValues(nextState)
    self.qValues[(state, action)] = (1 - self.alpha) * oldQ + self.
       alpha * (reward + self.discount * nextValue)
```

4 Findings and Insights

Testing the Q-learning agent in Gridworld revealed that the agent effectively learns the optimal policy over multiple episodes, as evidenced by converging Q-values when manually guided along the optimal path for four episodes with noise disabled. An interesting observation was the agent's tendency to explore suboptimal actions early on due to the epsilon-greedy strategy, which improved long-term learning by discovering better paths. This highlights the importance of balancing exploration and exploitation.

5 Challenges Faced

One challenge was ensuring correct tie-breaking in `computeActionFromQValues`. Initially, selecting the first action in a tie led to biased behavior, which was resolved by implementing random tie-breaking.

6 Hyperparameter Exploration

The agent's performance was sensitive to hyperparameter values:

- **`alpha` (learning rate)**: A high `alpha` (e.g., 0.5) caused rapid updates but instability in Q-values, while a lower `alpha` (e.g., 0.2) ensured stable learning at the cost of slower convergence.
- **`gamma` (discount factor)**: A `gamma` of 0.8 balanced immediate and future rewards effectively, while a higher `gamma` (e.g., 0.9) prioritized long-term rewards, sometimes leading to overfitting to distant states.