CSE584 HW2

Lakshmi Sivani Devarapalli

October 27, 2024

1 Abstract

This work uses a model-free Q-Learning reinforcement learning algorithm in a basic yet illustrative Gridworld environment. Gridworld is a grid-based simulated environment in which an agent starts at a random location and moves toward a goal while trying to avoid bombs-cells that incur penalties-and open cells. The agent is supposed to maximize the cumulative reward that it shall receive; this encourages efficient paths toward the goal while avoiding the penalized cells.

Here, the agent would learn an optimum policy for this grid using Q-learning based on the updating of a Q-table. The Q-table contains estimated future rewards, called Q-values, for every state-action pair. Updates in these Q-values will be done by the agent while it explores this grid. The key formula behind Q-learning update involves learning rate(α) and discount factor(γ) to balance immediate rewards against future rewards expected later. This way, the agent learns to remember which actions assure rewarding results, and over time, it converges increasingly closer to the most efficient path towards its goal.

The exploration-exploitation dilemma is an elementary concept in reinforcement learning, and here, the scenario will be addressed through employing an epsilon-greedy strategy. This lets an agent explore randomly selected actions with probability epsilon but within probability $1 - \epsilon$ exploit actions the agent has learned previously since they have high reward ratings from the Q-table. The model will alternate based on epsilon from pure exploration to exploitation of the learned agent's confidence across different episodes due to acquired Q-values.

Each component of the implementation is designed as:

Setup of the Grid environment: Rewards, penalties, and goal location are defined.

Q-table Initialization and Updates: Manages Learning Consisting of Adjustments in Q-Value According to Obtained Rewards.

Agent's actions and policy: The agent takes a set of actions following the epsilon-greedy strategy, balancing exploration and exploitation.

This project presents the Q-learning process sequentially and emphasizes the

learning aspects of the agent in an environmentally simple way, thus enabling learners to grasp the reinforcement learning dynamics. **Gridworld Q-learning** shall hence form the very foundation for advanced learning in complex RL applications.

2 Q-learning(RL alogrithm) implementation with code comments

```python
# Import necessary libraries
import numpy as np          # Import numpy for handling arrays and matrix operations
import random               # Import random for generating random numbers

# Define the Gridworld environment
class Gridworld:
    def __init__(self, size, goal, bomb, penalty=-1, reward=10):
        self.size = size                # Size of the grid (size x size)
        self.goal = goal                # Coordinates of the goal cell
        self.bomb = bomb                # Coordinates of the bomb cell
        self.penalty = penalty          # Penalty for stepping on a bomb cell
        self.reward = reward            # Reward for reaching the goal cell

    def step(self, state, action):
        # Determine the next state and reward based on the current state and action
        next_state = self.get_next_state(state, action)  # Calculate the next state based on the action taken
        if next_state == self.goal:                       # Check if the agent reached the goal
            return next_state, self.reward                # Return reward for reaching the goal
        elif next_state == self.bomb:                     # Check if the agent hit a bomb
            return next_state, self.penalty               # Return penalty for hitting a bomb
        else:
            return next_state, 0                          # Return neutral reward if no goal or bomb

    def get_next_state(self, state, action):
        # Determine the new position in the grid after taking an action
        if action == 0:                                   # Action 0: Move up
            return max(state[0] - 1, 0), state[1]        # Ensure agent doesn't move out of bounds
        elif action == 1:                                 # Action 1: Move down
```

```python
            return min(state[0] + 1, self.size - 1), state[1]
        elif action == 2:                                      # Action 2: Move left
            return state[0], max(state[1] - 1, 0)
        elif action == 3:                                      # Action 3: Move right
            return state[0], min(state[1] + 1, self.size - 1)

# Initialize Q-table and parameters for Q-learning
def q_learning(gridworld, episodes=500, alpha=0.1, gamma=0.9, epsilon=0.1):
    # Initialize Q-table with zeros for all state-action pairs
    q_table = np.zeros((gridworld.size, gridworld.size, 4))  # 4 actions (up, down, left, right)
    for episode in range(episodes):                          # Run training for the specified number of episodes
        # Initialize the state randomly at the beginning of each episode
        state = (np.random.randint(gridworld.size), np.random.randint(gridworld.size))
        while state != gridworld.goal:                       # Run loop until the goal state is reached
            # Choose action based on epsilon-greedy policy (explore vs. exploit)
            if random.uniform(0, 1) < epsilon:               # With probability epsilon, take random action
                action = np.random.randint(4)
            else:                                            # Otherwise, choose the best known action
                action = np.argmax(q_table[state])

            # Perform the chosen action, moving to the next state and receiving reward
            next_state, reward = gridworld.step(state, action)

            # Find the best next action's Q-value for updating
            best_next_action = np.argmax(q_table[next_state])
            td_target = reward + gamma * q_table[next_state][best_next_action]  # Compute TD target
            td_delta = td_target - q_table[state][action]                        # Compute TD error
            q_table[state][action] += alpha * td_delta                           # Update Q-value for the state-action pair

            state = next_state                               # Move to the next state for the next iteration
    return q_table                                           # Return the trained Q-table after all episodes

# Define the environment and run Q-learning
grid_size = 5                      # Define the size of the grid (5x5 grid)
goal_position = (4, 4)             # Define the goal position at the bottom-right corner
bomb_position = (2, 2)             # Define the bomb position in the center of the grid
```

```
gridworld = Gridworld(grid_size, goal_position, bomb_position)  # Initialize the gridworld environment
q_table = q_learning(gridworld)     # Train the agent using Q-learning
print("Trained Q-table:", q_table)  # Output the final Q-table
```

Listing 1: Q-learning Algorithm with Explanatory Comments in Gridworld

3 References

1. https://web.stanford.edu/class/cs234/
2. https://spinningup.openai.com/en/latest/
3. https://github.com/sichkar-valentyn