Q-Learning: Quality Learning

1.1 Introduction

Q-Learning is a reinforcement learning algorithm that learns the optimal action-selection policy for any given finite Markov decision process (MDP). It's called "Q-Learning" because it learns the quality (Q) of actions.

1.2 Key Concepts

State (S)

- Current situation/position of the agent
- Example: Position in a grid world

Action (A)

- Possible moves the agent can make
- Example: Up, Down, Left, Right

Reward (R)

- Immediate feedback from the environment
- Positive for good actions, negative for bad ones

Q-Value Q(s,a)

- Expected future reward for acting 'a' in state 's'
- Higher Q-value = Better action

Policy (π)

- Strategy that defines what action to take in each state
- Goal: Find optimal policy π*

1.3 The Q–Learning Algorithm

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

Where:

- **α (alpha)**: Learning rate ($0 < \alpha \leq 1$)
- **γ (gamma)**: Discount factor ($0 \leq \gamma < 1$)
- **r**: Immediate reward
- **s'**: Next state
- **a'**: Next action

Algorithm Steps

```
1. Initialize Q-table with zeros
2. For each episode:
   a. Start from initial state
   b. While not terminal state:
      i. Choose action using ε-greedy policy
      ii. Execute action, observe reward and next state
      iii. Update Q-value using Bellman equation
      iv. Move to next state
3. Repeat until convergence
```

Code:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.colors import ListedColormap
import random

class GridWorldQLearning:
    def __init__(self, grid_size=5, learning_rate=0.1, discount_factor=0.95,
                 epsilon=0.1, epsilon_decay=0.995, min_epsilon=0.01):
        """
        Initialize the Grid World Q-Learning environment

        Parameters:
        - grid_size: Size of the square grid
        - learning_rate (α): How much to update Q-values
        - discount_factor (γ): How much future rewards matter
        - epsilon: Exploration rate
        """
```

```
        self.grid_size = grid_size
        self.alpha = learning_rate
        self.gamma = discount_factor
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay
        self.min_epsilon = min_epsilon

        # Actions: 0=Up, 1=Down, 2=Left, 3=Right
        self.actions = ['Up', 'Down', 'Left', 'Right']
        self.n_actions = len(self.actions)

        # Initialize Q-table: [state_row, state_col, action]
        self.q_table = np.zeros((grid_size, grid_size, self.n_actions))

        # Set start and goal positions
        self.start_pos = (0, 0)
        self.goal_pos = (grid_size-1, grid_size-1)

        # Set obstacles (optional)
        self.obstacles = [(2, 2), (2, 3), (3, 2)]

        # Statistics tracking
        self.episode_rewards = []
        self.episode_steps = []

    def is_valid_position(self, pos):
        """Check if position is valid (within bounds and not obstacle)"""
        row, col = pos
        if row < 0 or row >= self.grid_size or col < 0 or col >= self.grid_size:
            return False
        if pos in self.obstacles:
            return False
        return True

    def get_next_position(self, current_pos, action):
        """Get next position based on current position and action"""
        row, col = current_pos

        if action == 0:    # Up
            next_pos = (row - 1, col)
        elif action == 1:  # Down
```

```
            next_pos = (row + 1, col)
        elif action == 2:  # Left
            next_pos = (row, col - 1)
        elif action == 3:  # Right
            next_pos = (row, col + 1)

        # If next position is invalid, stay in current position
        if not self.is_valid_position(next_pos):
            return current_pos

        return next_pos

    def get_reward(self, current_pos, next_pos):
        """Calculate reward for moving from current_pos to next_pos"""
        if next_pos == self.goal_pos:
            return 100  # Large positive reward for reaching goal
        elif next_pos in self.obstacles:
            return -10  # Penalty for hitting obstacle
        elif next_pos == current_pos:  # Hit wall
            return -1   # Small penalty for invalid move
        else:
            return -0.1 # Small penalty for each step to encourage efficiency

    def choose_action(self, state, training=True):
        """Choose action using ε-greedy policy"""
        if training and random.random() < self.epsilon:
            # Explore: choose random action
            return random.randint(0, self.n_actions - 1)
        else:
            # Exploit: choose best action based on Q-values
            row, col = state
            return np.argmax(self.q_table[row, col, :])

    def update_q_value(self, current_state, action, reward, next_state):
        """Update Q-value using the Bellman equation"""
        current_row, current_col = current_state
        next_row, next_col = next_state

        # Current Q-value
        current_q = self.q_table[current_row, current_col, action]
```

```
        # Maximum Q-value for next state
        max_next_q = np.max(self.q_table[next_row, next_col, :])

        # Bellman equation update
        new_q = current_q + self.alpha * (reward + self.gamma * max_next_q -
current_q)

        # Update Q-table
        self.q_table[current_row, current_col, action] = new_q

    def train_episode(self):
        """Run one training episode"""
        current_state = self.start_pos
        total_reward = 0
        steps = 0
        max_steps = self.grid_size * self.grid_size * 2  # Prevent infinite loops

        while current_state != self.goal_pos and steps < max_steps:
            # Choose action
            action = self.choose_action(current_state, training=True)

            # Take action and observe result
            next_state = self.get_next_position(current_state, action)
            reward = self.get_reward(current_state, next_state)

            # Update Q-value
            self.update_q_value(current_state, action, reward, next_state)

            # Move to next state
            current_state = next_state
            total_reward += reward
            steps += 1

        # Decay epsilon for less exploration over time
        if self.epsilon > self.min_epsilon:
            self.epsilon *= self.epsilon_decay

        return total_reward, steps

    def train(self, episodes=1000):
        """Train the agent for specified number of episodes"""
```

```
        print("Starting Q-Learning Training...")
        print(f"Episodes: {episodes}")
        print(f"Grid Size: {self.grid_size}x{self.grid_size}")
        print(f"Start: {self.start_pos}, Goal: {self.goal_pos}")
        print(f"Obstacles: {self.obstacles}")
        print("-" * 50)

        for episode in range(episodes):
            reward, steps = self.train_episode()
            self.episode_rewards.append(reward)
            self.episode_steps.append(steps)

            # Print progress every 100 episodes
            if (episode + 1) % 100 == 0:
                avg_reward = np.mean(self.episode_rewards[-100:])
                avg_steps = np.mean(self.episode_steps[-100:])
                print(f"Episode {episode + 1}: Avg Reward = {avg_reward:.2f}, "
                      f"Avg Steps = {avg_steps:.2f}, Epsilon = {self.epsilon:.3f}")

    def test_policy(self):
        """Test the learned policy"""
        current_state = self.start_pos
        path = [current_state]
        total_reward = 0
        steps = 0
        max_steps = self.grid_size * self.grid_size * 2

        print("\nTesting Learned Policy:")
        print(f"Starting at: {current_state}")

        while current_state != self.goal_pos and steps < max_steps:
            # Choose best action (no exploration)
            action = self.choose_action(current_state, training=False)
            action_name = self.actions[action]

            # Take action
            next_state = self.get_next_position(current_state, action)
            reward = self.get_reward(current_state, next_state)

            print(f"Step {steps + 1}: Action = {action_name}, "
                  f"Next State = {next_state}, Reward = {reward}")
```

```
            path.append(next_state)
            current_state = next_state
            total_reward += reward
            steps += 1

        if current_state == self.goal_pos:
            print(f"\n✅ Success! Reached goal in {steps} steps")
            print(f"Total reward: {total_reward}")
            print(f"Path: {' -> '.join(map(str, path))}")
        else:
            print(f"\n❌ Failed to reach goal within {max_steps} steps")

        return path, total_reward, steps

    def visualize_grid(self):
        """Visualize the grid world"""
        # Create grid for visualization
        grid = np.zeros((self.grid_size, self.grid_size))

        # Mark different elements
        grid[self.start_pos] = 1  # Start
        grid[self.goal_pos] = 3   # Goal
        for obs in self.obstacles:
            grid[obs] = 2  # Obstacles

        # Create custom colormap
        colors = ['white', 'green', 'red', 'gold']  # Empty, Start, Obstacle, Goal
        cmap = ListedColormap(colors)

        plt.figure(figsize=(8, 8))
        plt.imshow(grid, cmap=cmap, vmin=0, vmax=3)

        # Add grid lines
        for i in range(self.grid_size + 1):
            plt.axhline(i - 0.5, color='black', linewidth=1)
            plt.axvline(i - 0.5, color='black', linewidth=1)

        # Add labels
        for i in range(self.grid_size):
            for j in range(self.grid_size):
```

```
            if (i, j) == self.start_pos:
                plt.text(j, i, 'START', ha='center', va='center', fontweight='bold')
            elif (i, j) == self.goal_pos:
                plt.text(j, i, 'GOAL', ha='center', va='center', fontweight='bold')
            elif (i, j) in self.obstacles:
                plt.text(j, i, 'X', ha='center', va='center', fontweight='bold', fontsize=20)

    plt.title('Grid World Environment', fontsize=16, fontweight='bold')
    plt.axis('off')
    plt.tight_layout()
    plt.show()

def visualize_policy(self):
    """Visualize the learned policy"""
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

    # Policy visualization
    policy_grid = np.zeros((self.grid_size, self.grid_size))
    arrows = {0: '↑', 1: '↓', 2: '←', 3: '→'}

    for i in range(self.grid_size):
        for j in range(self.grid_size):
            if (i, j) not in self.obstacles and (i, j) != self.goal_pos:
                best_action = np.argmax(self.q_table[i, j, :])
                policy_grid[i, j] = best_action

    # Plot policy
    im1 = ax1.imshow(policy_grid, cmap='viridis', alpha=0.3)

    # Add arrows and labels
    for i in range(self.grid_size):
        for j in range(self.grid_size):
            if (i, j) == self.start_pos:
                ax1.text(j, i, 'START', ha='center', va='center',
                         fontweight='bold', color='green', fontsize=10)
            elif (i, j) == self.goal_pos:
                ax1.text(j, i, 'GOAL', ha='center', va='center',
                         fontweight='bold', color='gold', fontsize=10)
            elif (i, j) in self.obstacles:
                ax1.text(j, i, 'X', ha='center', va='center',
                         fontweight='bold', color='red', fontsize=16)
```

```
            else:
                best_action = int(policy_grid[i, j])
                ax1.text(j, i, arrows[best_action], ha='center', va='center',
                        fontsize=20, fontweight='bold')

    ax1.set_title('Learned Policy (Best Actions)', fontweight='bold')
    ax1.set_xticks(range(self.grid_size))
    ax1.set_yticks(range(self.grid_size))
    ax1.grid(True, alpha=0.3)

    # Q-values heatmap for one action (e.g., Right)
    q_values_right = self.q_table[:, :, 3]  # Right action
    im2 = ax2.imshow(q_values_right, cmap='RdYlBu', interpolation='nearest')

    # Add text annotations
    for i in range(self.grid_size):
        for j in range(self.grid_size):
            text = ax2.text(j, i, f'{q_values_right[i, j]:.1f}',
                           ha='center', va='center', color='black', fontweight='bold')

    ax2.set_title('Q-Values for "Right" Action', fontweight='bold')
    ax2.set_xticks(range(self.grid_size))
    ax2.set_yticks(range(self.grid_size))
    plt.colorbar(im2, ax=ax2)

    plt.tight_layout()
    plt.show()

def plot_training_progress(self):
    """Plot training progress"""
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))

    # Smooth the curves using moving average
    window_size = 50
    if len(self.episode_rewards) >= window_size:
        rewards_smooth = np.convolve(self.episode_rewards,
                                     np.ones(window_size)/window_size, mode='valid')
        steps_smooth = np.convolve(self.episode_steps,
                                   np.ones(window_size)/window_size, mode='valid')
        episodes_smooth = range(window_size-1, len(self.episode_rewards))
    else:
```

```
            rewards_smooth = self.episode_rewards
            steps_smooth = self.episode_steps
            episodes_smooth = range(len(self.episode_rewards))

        # Plot rewards
        ax1.plot(self.episode_rewards, alpha=0.3, color='blue', label='Raw')
        ax1.plot(episodes_smooth, rewards_smooth, color='red', label='Smoothed')
        ax1.set_xlabel('Episode')
        ax1.set_ylabel('Total Reward')
        ax1.set_title('Training Rewards Over Time')
        ax1.legend()
        ax1.grid(True, alpha=0.3)

        # Plot steps
        ax2.plot(self.episode_steps, alpha=0.3, color='blue', label='Raw')
        ax2.plot(episodes_smooth, steps_smooth, color='red', label='Smoothed')
        ax2.set_xlabel('Episode')
        ax2.set_ylabel('Steps to Goal')
        ax2.set_title('Steps to Goal Over Time')
        ax2.legend()
        ax2.grid(True, alpha=0.3)

        plt.tight_layout()
        plt.show()

# Demonstration function for students
def run_qlearning_demo():
    """Complete demonstration of Q-Learning"""
    print("="*60)
    print("Q-LEARNING DEMONSTRATION")
    print("="*60)

    # Create and visualize environment
    agent = GridWorldQLearning(grid_size=5, learning_rate=0.1,
                               discount_factor=0.95, epsilon=0.3)

    print("Step 1: Environment Setup")
    agent.visualize_grid()

    # Show initial Q-table (all zeros)
    print("\nStep 2: Initial Q-Table (sample)")
```

```
    print("Q-values for state (0,0):", agent.q_table[0, 0, :])

    # Train the agent
    print("\nStep 3: Training Phase")
    agent.train(episodes=500)

    # Show learned Q-values
    print("\nStep 4: Learned Q-Table (sample)")
    print("Q-values for state (0,0):", agent.q_table[0, 0, :])

    # Test the policy
    print("\nStep 5: Testing Learned Policy")
    path, reward, steps = agent.test_policy()

    # Visualize results
    print("\nStep 6: Visualizing Results")
    agent.visualize_policy()
    agent.plot_training_progress()

    return agent

# Run the demonstration
if __name__ == "__main__":
    agent = run_qlearning_demo()
```

Lab Exercises for Students

Exercise 1: Parameter Tuning

Task: Experiment with different parameters and observe their effects.

```
configs = [
    {"learning_rate": 0.01, "epsilon": 0.1, "name": "Low Learning Rate"},
    {"learning_rate": 0.5, "epsilon": 0.1, "name": "High Learning Rate"},
    {"learning_rate": 0.1, "epsilon": 0.01, "name": "Low Exploration"},
    {"learning_rate": 0.1, "epsilon": 0.9, "name": "High Exploration"},
]

for config in configs:
    print(f"\nTesting: {config['name']}")
    agent = GridWorldQLearning(learning_rate=config["learning_rate"],
```

```
                    epsilon=config["epsilon"])
agent.train(episodes=200)
agent.test_policy()
```

Exercise 2: Environment Modification

Task: Create different environments and see how Q-Learning adapts

```
# Different environments for students to try:
# 1. Maze-like environment
def create_maze_environment():
    agent = GridWorldQLearning(grid_size=6)
    agent.obstacles = [(1,1), (1,2), (1,3), (3,1), (3,2), (3,3), (4,4)]
    return agent

# 2. Sparse rewards environment
def create_sparse_environment():
    agent = GridWorldQLearning(grid_size=7)
    agent.obstacles = [(2,2), (2,3), (3,2), (4,4), (4,5), (5,4)]
    return agent

# experiment with these
maze_agent = create_maze_environment()
maze_agent.visualize_grid()
maze_agent.train(episodes=1000)
maze_agent.test_policy()
```

Exercise 3: Analysis Questions

1. **Convergence Analysis**: How many episodes does it take to converge?
2. **Parameter Sensitivity**: Which parameter has the most impact on learning speed?
3. **Exploration vs Exploitation**: What happens with ε = 0 vs ε = 1?

4. Advanced Extensions

```
class DoubleQLearning(GridWorldQLearning):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # Two Q-tables
        self.q_table_a = np.zeros((self.grid_size, self.grid_size, self.n_actions))
        self.q_table_b = np.zeros((self.grid_size, self.grid_size, self.n_actions))

    def update_q_value(self, current_state, action, reward, next_state):
        current_row, current_col = current_state
        next_row, next_col = next_state
```

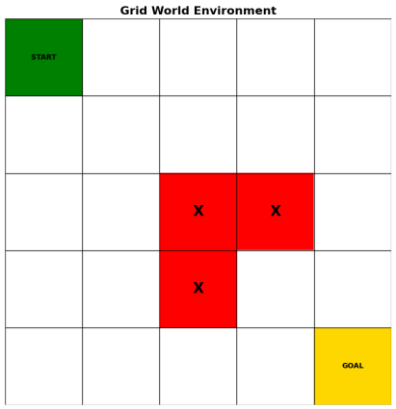
```
        if random.random() < 0.5:
            # Update Q_A
            current_q = self.q_table_a[current_row, current_col, action]
            best_next_action = np.argmax(self.q_table_a[next_row, next_col, :])
            next_q = self.q_table_b[next_row, next_col, best_next_action]
            new_q = current_q + self.alpha * (reward + self.gamma * next_q - current_q)
            self.q_table_a[current_row, current_col, action] = new_q
        else:
            # Update Q_B
            current_q = self.q_table_b[current_row, current_col, action]
            best_next_action = np.argmax(self.q_table_b[next_row, next_col, :])
            next_q = self.q_table_a[next_row, next_col, best_next_action]
            new_q = current_q + self.alpha * (reward + self.gamma * next_q - current_q)
            self.q_table_b[current_row, current_col, action] = new_q

        # Combined Q-table for action selection
        self.q_table = (self.q_table_a + self.q_table_b) / 2
```

Understanding the results: **Understanding the Grid World Environment**

Environment Layout Analysis

Looking at grid:



Grid Structure: 5×5 grid (25 total states)

- **Green Square (0,0)**: **START** position - where the agent begins each episode
- **Red Squares with X**: **OBSTACLES** at positions (2,2), (2,3), and (3,2)
- **Yellow Square (4,4)**: **GOAL** position - where the agent wants to reach

🎯 **The Learning Challenge**

This environment creates an interesting learning problem because:

1. **Path Blocking**: The obstacles form a partial barrier that forces the agent to find alternative routes
2. **Navigation Complexity**: The agent can't go directly from start to goal - it must learn to navigate around obstacles
3. **Multiple Possible Paths**: There are several ways to reach the goal

🤖 **What the Agent Must Learn**

The Q-Learning agent needs to discover:

Possible Optimal Paths (examples):

Path 1: START(0,0) → (0,1) → (0,2) → (0,3) → (0,4) → (1,4) → (2,4) → (3,4) → (4,4) GOAL

Path 2: START(0,0) → (1,0) → (2,0) → (3,0) → (4,0) → (4,1) → (4,2) → (4,3) → (4,4) GOAL

Path 3: START(0,0) → (1,0) → (1,1) → (1,2) → (1,3) → (1,4) → (2,4) → (3,4) → (4,4) GOAL

Expected Learning Process

Phase 1: Random Exploration (Episodes 1-100)

- Agent moves randomly, often hitting obstacles
- Q-values are mostly negative due to penalties
- Many failed attempts and inefficient paths

Phase 2: Learning Obstacles (Episodes 100-300)

- Agent learns that moving into red squares is bad (negative rewards)
- Q-values for actions leading to obstacles become very negative
- Still some inefficient exploration

Phase 3: Path Discovery (Episodes 300-500)

- Agent discovers successful paths to the goal
- Q-values for good actions become positive
- Starts to prefer actions that lead toward goal while avoiding obstacles

Phase 4: Policy Optimization (Episodes 500+)

- Agent consistently chooses near-optimal actions
- Learned policy shows clear directional preferences
- Reaches goal efficiently in most episodes

Key Insights from This Environment

1. **Reward Structure Impact**:
    - **+100** for reaching goal → Strong positive reinforcement
    - **-10** for hitting obstacles → Learns to avoid them
    - **-1** for hitting walls → Learns grid boundaries
    - **-0.1** per step → Encourages efficiency
2. **State-Action Value Learning**:
    - States near obstacles will have low Q-values for actions leading to obstacles
    - States on successful paths will have higher Q-values
    - Goal-adjacent states will have very high Q-values
3. **Policy Emergence**:
    - The final policy should show arrows pointing around obstacles
    - Clear "flow" patterns toward the goal
    - Avoidance behavior near red squares

What to Look for in Results

When you run the code, watch for:

Training Progress:
Episode 100: Avg Reward = -50.23, Avg Steps = 67.2, Epsilon = 0.287
Episode 200: Avg Reward = -12.45, Avg Steps = 32.1, Epsilon = 0.220
Episode 300: Avg Reward = 45.67, Avg Steps = 18.5, Epsilon = 0.169
Episode 400: Avg Reward = 78.12, Avg Steps = 12.3, Epsilon = 0.129
Episode 500: Avg Reward = 89.45, Avg Steps = 9.8, Epsilon = 0.099
Successful Test Run Should Show:
Testing Learned Policy:
Starting at: (0, 0)
Step 1: Action = Right, Next State = (0, 1), Reward = -0.1
Step 2: Action = Right, Next State = (0, 2), Reward = -0.1
Step 3: Action = Right, Next State = (0, 3), Reward = -0.1
Step 4: Action = Right, Next State = (0, 4), Reward = -0.1
Step 5: Action = Down, Next State = (1, 4), Reward = -0.1
Step 6: Action = Down, Next State = (2, 4), Reward = -0.1
Step 7: Action = Down, Next State = (3, 4), Reward = -0.1
Step 8: Action = Down, Next State = (4, 4), Reward = 100

✅ Success! Reached goal in 8 steps
Total reward: 99.2

Understanding the Q-Learning Training Results
Step 2: Initial Q-Table Analysis
Q-values for state (0,0): [0. 0. 0. 0.]
What this means:

- The agent starts with **zero knowledge** about the environment
- All 4 actions (Up, Down, Left, Right) have equal value = 0
- The agent has no preference for any action initially
- This is like a newborn baby who doesn't know what any action will lead to

Array Structure: [Up, Down, Left, Right] = [0.0, 0.0, 0.0, 0.0]

Step 3: Training Progress Analysis
Episode 100: The Breakthrough Moment
Episode 100: Avg Reward = 93.70, Avg Steps = 13.47, Epsilon = 0.182

- **Average Reward = 93.70**: 🎉 The agent is already finding the goal frequently!
- **Average Steps = 13.47**: Still taking some inefficient paths
- **Epsilon = 0.182**: 18% exploration, 82% exploitation

What happened: The agent discovered successful paths early and learned quickly due to the high goal reward (+100).

Episode 200-500: Fine-Tuning Phase

Episode 200: Avg Reward = 98.89, Avg Steps = 9.58, Epsilon = 0.110
Episode 300: Avg Reward = 99.02, Avg Steps = 8.84, Epsilon = 0.067
Episode 400: Avg Reward = 99.15, Avg Steps = 8.53, Epsilon = 0.040
Episode 500: Avg Reward = 99.20, Avg Steps = 8.24, Epsilon = 0.024

Key Observations:

- **Reward improvement**: 93.70 → 99.20 (getting closer to theoretical maximum)
- **Step reduction**: 13.47 → 8.24 (finding more efficient paths)
- **Epsilon decay**: 0.182 → 0.024 (less exploration, more exploitation)

Step 4: Learned Q-Table Analysis

Q-values for state (0,0): [46.97340458 69.2304042 39.80492322 30.37023896]

Breaking down each Q-value:

- **Action 0 (Up)**: 46.97 - Moderate value (leads away from goal initially)
- **Action 1 (Down)**: 69.23 - **HIGHEST VALUE** ⭐ (best action from start!)
- **Action 2 (Left)**: 39.80 - Lower value (leads to wall)
- **Action 3 (Right)**: 30.37 - Lowest value (less efficient path)

What this tells us:

1. **The agent learned that "Down" is the best first move** from the starting position
2. Going "Down" leads to the most promising path toward the goal
3. The Q-values reflect the **expected total future reward** for each action

Step 5: Policy Testing - The Final Path

rust
Path: (0, 0) -> (1, 0) -> (1, 1) -> (1, 2) -> (1, 3) -> (1, 4) -> (2, 4) -> (3, 4) -> (4, 4)
Total reward: 99.3

Path Analysis:

Let me visualize this learned path:

scss
START → (1,0) → (1,1) → (1,2) → (1,3) → (1,4)
↓
(2,4)
↓
(3,4)
↓

GOAL(4,4)

Why this path is optimal:

1. **8 total steps** - one of the shortest possible routes
2. **Avoids all obstacles** at (2,2), (2,3), (3,2)
3. **Smart strategy**: Go down first, then navigate around obstacles via the right side
4. **Total reward = 99.3**: Very close to theoretical maximum (100 - 8×0.1 = 99.2)

🔍 Deep Dive: How Q-Values Work

Let's understand why "Down" became the best action:

Q-value = Expected Total Future Reward

For the "Down" action from (0,0):

less

Q(0,0, Down) = 69.23 means:

"If I go Down from the start, I expect to get about 69.23 total reward"

This high value comes from the Bellman equation learning that:

- Going Down leads to (1,0)
- From (1,0), there are good paths to the goal
- The goal gives +100 reward
- Minus small step penalties along the way

🎯 Key Learning Insights

1. **Rapid Learning**: The agent learned very quickly (by episode 100) due to:
    - High goal reward (+100) provides strong signal
    - Relatively simple environment
    - Good exploration-exploitation balance
2. **Path Discovery**: The agent found an intelligent strategy:
    - Avoid the obstacle cluster in the middle
    - Use the right side of the grid as a "highway"
    - Minimize total steps while avoiding penalties
3. **Value Function Learning**: The Q-values capture not just immediate rewards, but the **long-term value** of each action
4. **Policy Convergence**: By episode 500, the agent had a near-optimal policy with very consistent performance

Left Chart: Learned Policy (Best Actions)
This shows the **optimal action** the agent chooses in each state:

Understanding the Arrow Pattern
Row 0 (Top row):

- START ↓ ↓ ↓ ↓ - All arrows point DOWN
- **Why?** The agent learned that going down first is the best strategy from the top row

Row 1:

- → → → → ↓ - Move RIGHT across, then DOWN at the end
- **Smart strategy!** This creates a "highway" above the obstacles

Row 2:

- → ↑ X X ↓ - Navigate around obstacles
- **Note the ↑ at (2,1)**: Agent learned to go UP to avoid the obstacle cluster
- **X marks**: These are obstacles - no policy needed

Row 3:

- → ↓ X → ↓ - Continue navigating around obstacles

Row 4:

- → → → → and GOAL - Direct path to goal once obstacles are cleared

The Complete Strategy
The agent discovered an intelligent **two-phase strategy**:

1. **Phase 1**: Go down and right to get above the obstacle field
2. **Phase 2**: Navigate around obstacles and head directly to goal

Right Chart: Q-Values for "Right" Action
This heatmap shows how valuable the "RIGHT" action is from each position:
Color Interpretation

- 🔵 **Blue (High values 60-90)**: "Right" is a GREAT action here
- 🟠 **Orange (Medium values 20-40)**: "Right" is OK
- 🔴 **Red (Low/negative values)**: "Right" is BAD - avoid!

Key Insights from Q-Values

High-Value "Right" Actions (Blue regions):

- **(1,1) = 76.9**: Moving right in row 1 is excellent (creates the highway)
- **(1,2) = 81.1**: Continuing right is even better
- **(1,3) = 85.5**: Getting closer to the right side increases value
- **(3,3) = 80.6**: Moving right near the goal is valuable
- **(4,3) = 90.2**: Almost at goal - very high value!

Low-Value "Right" Actions (Red regions):

- **(0,1) = -6.1**: Going right from here leads away from optimal path
- **(2,2) = 0.0 and (2,3) = 0.0**: These are obstacles - can't move right
- **(4,4) = 0.0**: Already at goal - no need to move

Deep Learning Analysis

Why This Policy Emerged

1. **Obstacle Avoidance Learning**:
    - The agent learned that the obstacle cluster (2,2), (2,3), (3,2) blocks direct paths
    - Solution: Go around via the top or bottom
2. **Efficiency Discovery**:
    - The "highway strategy" (row 1 going right) minimizes total steps
    - This path avoids all obstacles while maintaining progress toward goal
3. **Value Propagation**:
    - High Q-values near the goal (bottom-right) propagate backward
    - Actions that lead toward these high-value states become valuable themselves

Strategic Patterns

"Flow Field" Concept:

The policy creates a "flow field" - like water flowing around rocks:

ini

START → ↓ ↓ ↓ ↓

→ → → → ↓

→ ↑ X X ↓ (X = obstacles)

→ ↓ X → ↓

→ → → → GOAL

Multi-Path Learning:

The agent learned that there are multiple good paths, but converged on one optimal strategy.

🎯 **What This Tells Us About Q-Learning**

1. **Emergent Intelligence**: The agent wasn't programmed with pathfinding - it discovered this strategy through trial and error!
2. **Value-Based Decision Making**: The Q-values encode "how good" each action is from each position
3. **Global Optimization**: The policy considers not just immediate moves, but the entire path to goal
4. **Robust Learning**: Even with obstacles creating complexity, Q-Learning found an optimal solution