Hints for Exercises in **Chapter 6**

1. Difference between Reinforcement Learning and Supervised Learning

Answer:
Supervised learning learns from labeled data to predict outcomes. Reinforcement learning learns by interacting with an environment, receiving rewards or penalties to guide future actions.

Hint:
Why might reinforcement learning be more suitable for tasks like game playing or robotics than supervised learning?

---

2. Exploration vs. Exploitation in Reinforcement Learning

Answer:
Exploration involves trying new actions to discover their effects. **Exploitation** uses known actions that yield high rewards. Balancing both is key to effective learning.

Hint:
What happens if an agent explores too much or too little?

---

3. CNN vs. RNN

Answer:
CNNs are designed for spatial data like images, using filters to detect patterns. RNNs handle sequential data like text or time series, maintaining memory of previous inputs.

Hint:
Why might CNNs struggle with time-dependent data?

---

4. What is a Markov Decision Process (MDP)?

Answer:
An MDP models decision-making with states, actions, rewards, and transitions, assuming the future depends only on the current state and action (Markov property).

Hint:
How does the Markov property simplify reinforcement learning?

---

5. Bellman Equation in Reinforcement Learning

Answer:
The Bellman equation expresses the value of a state as the immediate reward plus the discounted value of the next state. It's central to value-based methods like Q-learning.

Hint:
Why is recursion important in the Bellman equation?

---

6. Overfitting in Deep Learning

Answer:
Overfitting occurs when a model learns noise in training data, reducing generalization. Mitigation techniques include regularization, dropout, early stopping, and data augmentation.

Hint:
How can validation performance help detect overfitting?

---

7. Value-Based vs. Policy-Based Methods

Answer:
Value-based methods estimate the value of actions (e.g., Q-learning). Policy-based methods directly learn the policy (e.g., REINFORCE). Actor-Critic combines both.

Hint:
Why might policy-based methods be better for continuous action spaces?

---

8. How Q-learning Works

Answer:
Q-learning updates Q-values using the Bellman equation:
Q(s,a)←Q(s,a)+α[r+γmax Q(s',a')−Q(s,a)]Q(s,a) \leftarrow Q(s,a) + \alpha [r + \gamma \max Q(s',a') - Q(s,a)]Q(s,a)←Q(s,a)+α[r+γmaxQ(s',a')−Q(s,a)]
It learns the optimal policy by maximizing expected rewards.

Hint:
Why is Q-learning considered off-policy?

---

9. Implement Q-learning for Grid-World

Answer:
Create a grid environment, define states and actions, initialize Q-table, and update Q-values based on agent movement and rewards.

```
# Grid-world Q-learning implementation
import numpy as np
import random


grid_size = 5
goal_state = (4, 4)
start_state = (0, 0)
actions = ['up', 'down', 'left', 'right']
action_map = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), 'right': (0,
1)}


Q = {(i, j): {a: 0.0 for a in actions} for i in range(grid_size) for j in
range(grid_size)}
alpha, gamma, epsilon, episodes = 0.1, 0.9, 0.2, 500


def step(state, action):
    move = action_map[action]
    next_state = (max(0, min(grid_size - 1, state[0] + move[0])),
                  max(0, min(grid_size - 1, state[1] + move[1])))
    reward = 1 if next_state == goal_state else -0.1
```

```
    done = next_state == goal_state
    return next_state, reward, done

for _ in range(episodes):
    state = start_state
    while True:
        action = random.choice(actions) if random.random() < epsilon else
max(Q[state], key=Q[state].get)
        next_state, reward, done = step(state, action)
        best_next = max(Q[next_state], key=Q[next_state].get)
        Q[state][action] += alpha * (reward + gamma *
Q[next_state][best_next] - Q[state][action])
        state = next_state
        if done: break
```

Hint:
How does the size of the grid affect learning complexity?

---

10. RL Agent for Tic-Tac-Toe

Answer:
Define game states, legal moves, and rewards. Use Q-learning or policy gradients to train the agent through self-play.

```
# Q-learning agent for Tic-Tac-Toe
class TicTacToe:
    def __init__(self): self.reset()
    def reset(self): self.board = [' '] * 9; self.done = False; self.winner =
None; return ''.join(self.board)
    def available_actions(self): return [i for i, v in enumerate(self.board)
if v == ' ']
```

```
    def step(self, action, player):
        if self.board[action] != ' ' or self.done: return
''.join(self.board), -10, True
        self.board[action] = player; self.check_winner()
        if self.done: return ''.join(self.board), 1 if self.winner == player
else 0.5, True
        return ''.join(self.board), 0, False
    def check_winner(self):
        for a, b, c in
[(0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6)]:
            if self.board[a] == self.board[b] == self.board[c] != ' ':
self.done = True; self.winner = self.board[a]; return
        if ' ' not in self.board: self.done = True


class QAgent:
    def __init__(self): self.q = {}; self.alpha, self.gamma, self.epsilon =
0.5, 0.9, 0.1
    def get_q(self, s, a): return self.q.get((s, a), 0.0)
    def choose(self, s, acts): return random.choice(acts) if random.random()
< self.epsilon else max(acts, key=lambda a: self.get_q(s, a))
    def update(self, s, a, r, s2, a2s): self.q[(s, a)] = self.get_q(s, a) +
self.alpha * (r + self.gamma * max([self.get_q(s2, a2) for a2 in a2s]) -
self.get_q(s, a))


env, agent = TicTacToe(), QAgent()
for _ in range(10000):
    s = env.reset()
    while True:
        acts = env.available_actions()
        a = agent.choose(s, acts)
        s = env.step(a, 'X')
        a2s = env.available_actions()
        agent.update(s, a, r, s2, a2s)
        if done: break
```

```
        if a2s: env.step(random.choice(a2s), 'O')
        s = env.get_state()
```

Hint:
How can symmetry in Tic-Tac-Toe reduce the state space?

---

11. Reward Function Design

Answer:
Reward functions guide agent behavior. Design rewards to encourage desired outcomes and penalize undesired ones, balancing short-term and long-term goals.

```
class CustomEnv:
    def __init__(self): self.state, self.goal, self.max_steps = 0, 10, 20
    def reset(self): self.state, self.steps = 0, 0; return self.state
    def step(self, action):
        self.state += action; self.steps += 1
        if self.state == self.goal: return self.state, 100, True
        if self.steps >= self.max_steps: return self.state, -10, True
        return self.state, -1, False
```

Hint:
Can poorly designed rewards lead to unintended behaviors?

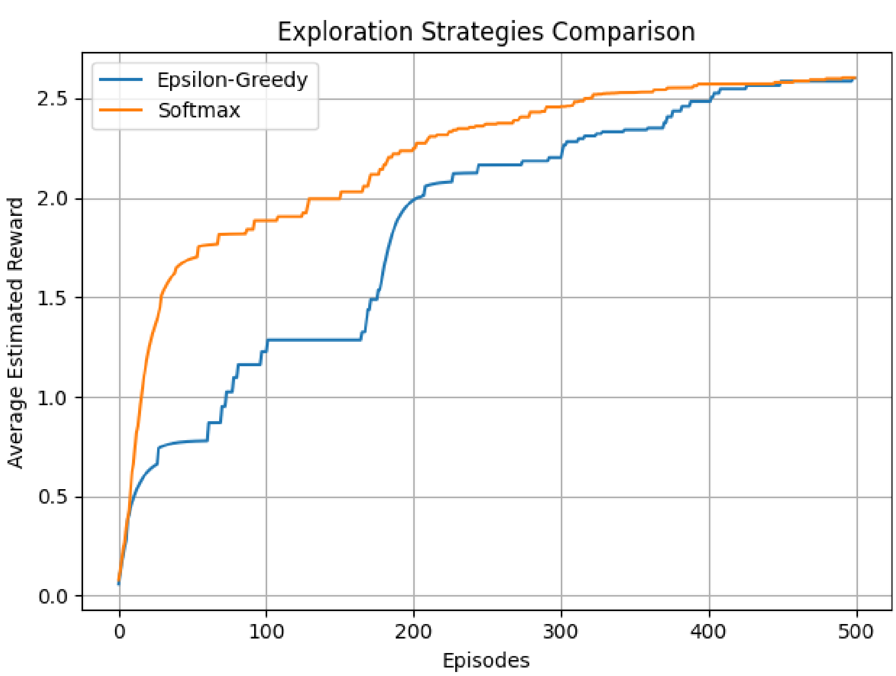
---

12. Exploration Strategies

Answer:
Common strategies include ε-greedy, softmax, and Upper Confidence Bound (UCB). Each balances exploration and exploitation differently.

```
import numpy as np
import matplotlib.pyplot as plt

rewards = np.array([1, 2, 3, 4, 5])
```

```
q_eps, q_soft = np.zeros(5), np.zeros(5)

eps, temp = 0.1, 1.0

avg_eps, avg_soft = [], []


for _ in range(500):

    a = np.random.randint(5) if np.random.rand() < eps else np.argmax(q_eps)

    r = rewards[a]; q_eps[a] += 0.1 * (r - q_eps[a]);
avg_eps.append(np.mean(q_eps))


for _ in range(500):

    probs = np.exp(q_soft / temp); probs /= np.sum(probs)

    a = np.random.choice(np.arange(5), p=probs)

    r = rewards[a]; q_soft[a] += 0.1 * (r - q_soft[a]);
avg_soft.append(np.mean(q_soft))


plt.plot(avg_eps, label='Epsilon-Greedy'); plt.plot(avg_soft,
label='Softmax')

plt.legend(); plt.title('Exploration Strategies'); plt.show()
```

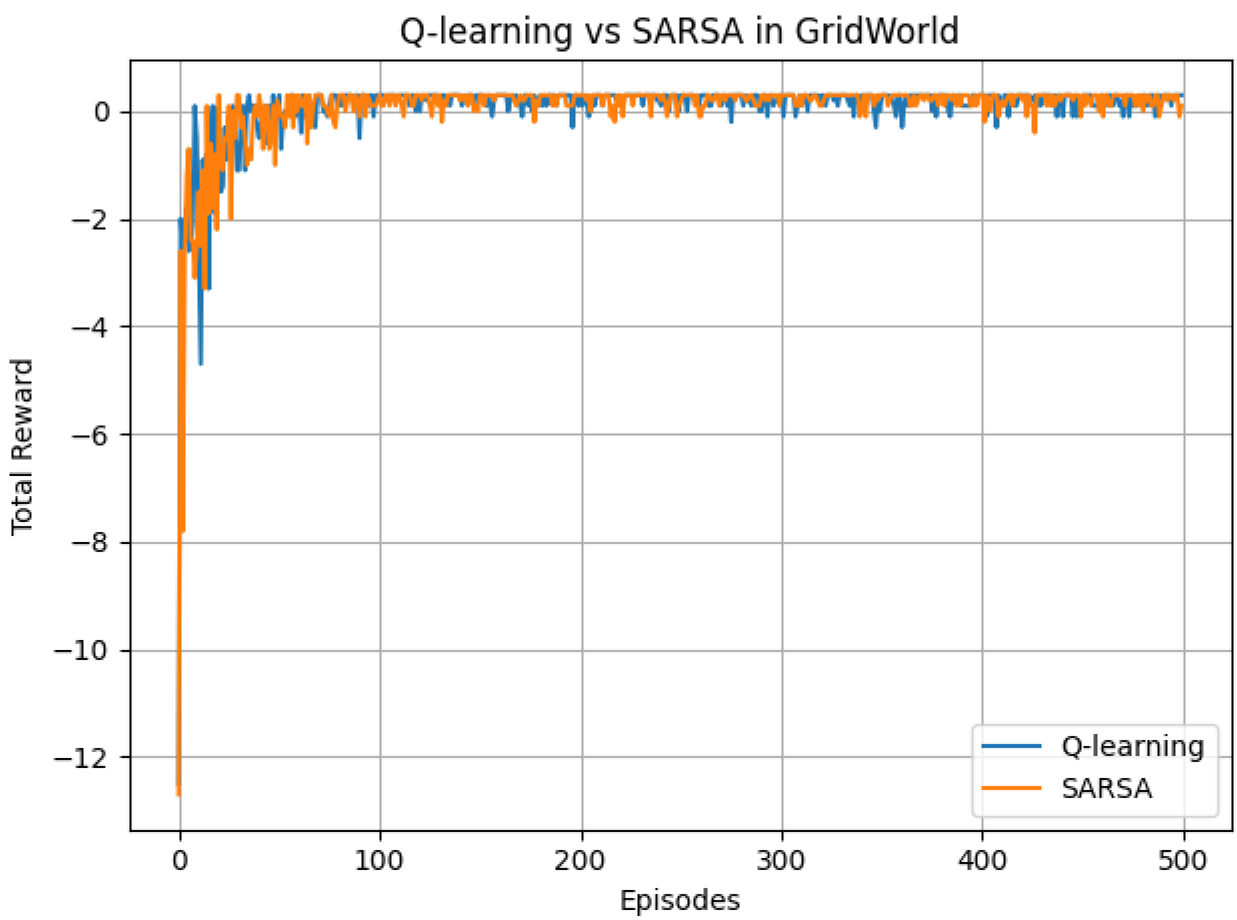


Hint:
How does ε decay affect learning over time?

---

13. Q-learning vs. SARSA

Answer:
Q-learning is off-policy and uses the best possible future action. SARSA is on-policy and uses the actual next action taken. SARSA is safer in risky environments.



Hint:
Why might SARSA perform better in stochastic environments?

---

14. Actor-Critic in Continuous Action Spaces

Answer:
Actor-Critic combines policy (actor) and value (critic) networks. It's effective in environments with continuous actions, like robotics or control systems.

Hint:
How does the critic help stabilize the actor's learning?

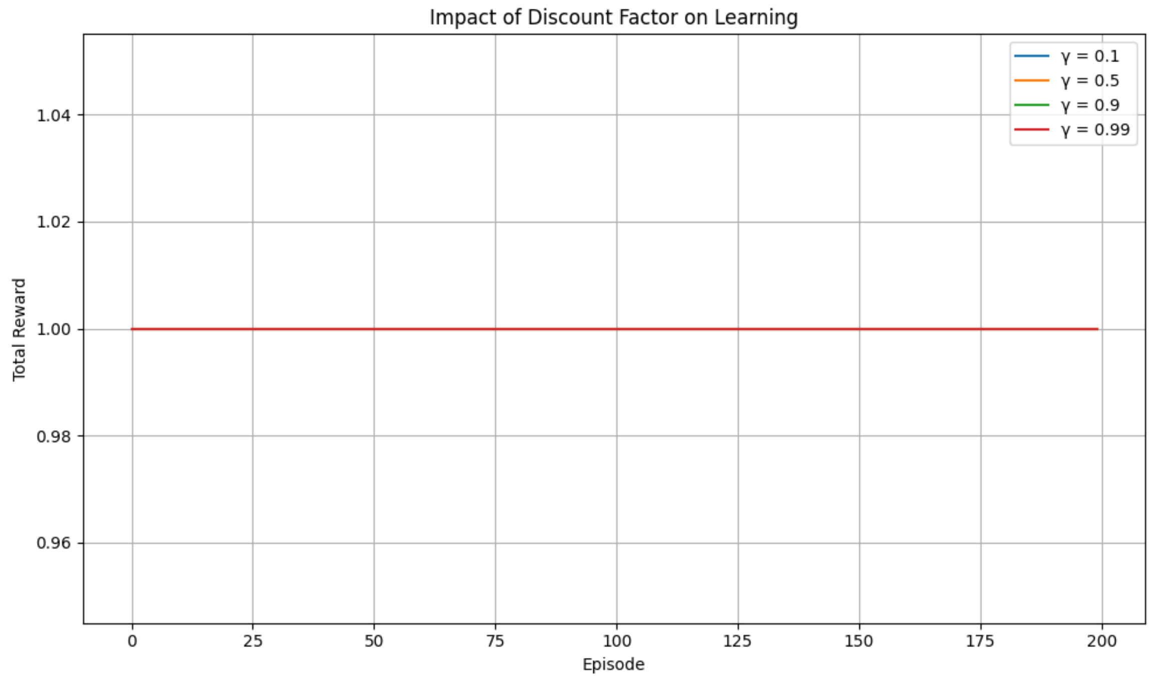
---

15. Impact of Discount Factors

Answer:
The discount factor (γ) determines how much future rewards are valued. A high γ favors long-term rewards; a low γ focuses on immediate gains.

```
#Q-learning with varying discount factors

def step(state, action):

    next_state = min(state + 1, 4) if action == 1 else max(state - 1, 0)

    reward = 1 if next_state == 4 else 0

    return next_state, reward


def run_q_learning(gamma):

    Q = np.zeros((5, 2))

    for _ in range(200):

        state = 0

        while state != 4:

            action = np.random.choice([0, 1]) if np.random.rand() < 0.1 else
np.argmax(Q[state])

            next_state, reward = step(state, action)

            Q[state, action] += 0.1 * (reward + gamma * np.max(Q[next_state])
- Q[state, action])

            state = next_state

    return Q
```

Tested Q-learning with γ = 0.1, 0.5, 0.9, 0.99.



Hint:

What happens if γ is set too close to 1 or 0?

---

16. RL in Real-World Problems

Answer:

RL can optimize inventory by learning reorder policies or control robots by learning movement strategies. It adapts to dynamic environments.

```
import numpy as np

import random


class InventoryEnv:

    def __init__(self, max_inventory=100, demand_mean=20, holding_cost=1, stockout_cost=5, order_cost=2):

        self.max_inventory = max_inventory

        self.demand_mean = demand_mean

        self.holding_cost = holding_cost

        self.stockout_cost = stockout_cost

        self.order_cost = order_cost
```

```
    def reset(self):
        self.state = self.max_inventory // 2
        return self.state

    def step(self, action):
        order = action
        self.state = min(self.state + order, self.max_inventory)
        demand = np.random.poisson(self.demand_mean)
        sales = min(self.state, demand)
        self.state -= sales

        holding = self.holding_cost * self.state
        stockout = self.stockout_cost * max(0, demand - sales)
        order_cost = self.order_cost * order
        reward = -(holding + stockout + order_cost)

        return self.state, reward

def q_learning(env, episodes=1000, alpha=0.1, gamma=0.95, epsilon=0.1):
    q_table = np.zeros((env.max_inventory + 1, env.max_inventory + 1))
    for _ in range(episodes):
        state = env.reset()
        for _ in range(100):
            action = random.randint(0, env.max_inventory - state) if random.random() < epsilon else np.argmax(q_table[state])
            next_state, reward = env.step(action)
            q_table[state, action] += alpha * (reward + gamma * np.max(q_table[next_state]) - q_table[state, action])
            state = next_state
    return q_table
```

Hint:
What challenges arise when applying RL to real-world systems?

---

17. (Project) RL Agent for Snake Game

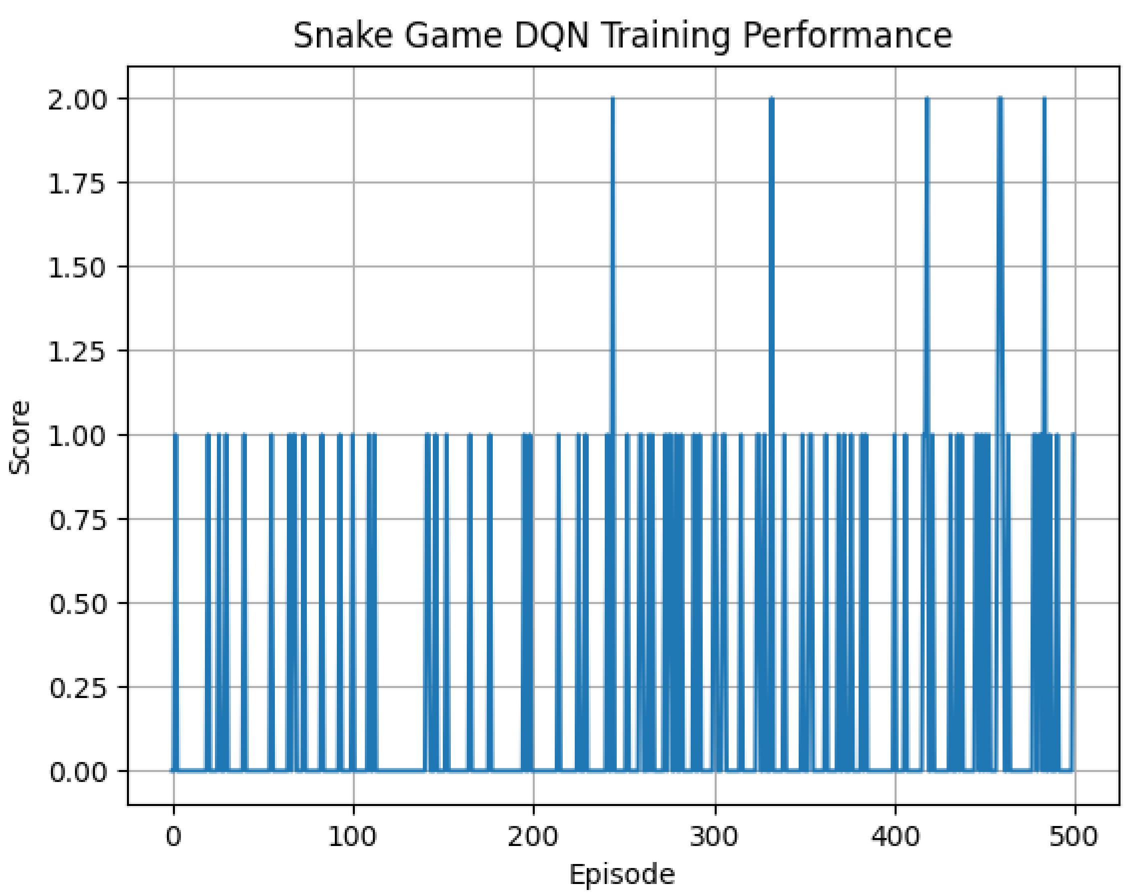
Answer:
Model the game as an environment with states (snake position, food), actions (move directions), and rewards (eating food, avoiding walls). Use DQN or policy gradients.

Agent trained to play Snake using Deep Q-Networks.

```
# Simplified version of Snake game with DQN

import numpy as np

import random

from collections import deque

import torch

import torch.nn as nn

import torch.optim as optim


class SnakeGame:

    def __init__(self, grid_size=10):

        self.grid_size = grid_size

        self.reset()


    def reset(self):

        self.snake = [(5, 5)]

        self.direction = (0, 1)

        self.spawn_food()

        self.done = False

        return self.get_state()


    def spawn_food(self):
```

```
        while True:
            self.food = (random.randint(0, self.grid_size - 1),
random.randint(0, self.grid_size - 1))
            if self.food not in self.snake:
                break

    def get_state(self):
        state = np.zeros((self.grid_size, self.grid_size), dtype=int)
        for x, y in self.snake:
            state[x, y] = 1
        state[self.food[0], self.food[1]] = 2
        return state.flatten()

    def step(self, action):
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        self.direction = directions[action]
        head = self.snake[0]
        new_head = (head[0] + self.direction[0], head[1] + self.direction[1])

        if (new_head in self.snake or not (0 <= new_head[0] < self.grid_size)
or not (0 <= new_head[1] < self.grid_size)):
            self.done = True
            return self.get_state(), -10, self.done

        self.snake.insert(0, new_head)
        reward = 10 if new_head == self.food else -1
        if new_head == self.food:
            self.spawn_food()
        else:
            self.snake.pop()
        return self.get_state(), reward, self.done
```

```
class DQN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(DQN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        return self.fc2(x)
```



Hint:
How can sparse rewards affect learning in Snake?

18. (Project) RL Agent for Chess

Answer:
Use deep RL with self-play, combining policy and value networks. AlphaZero is a notable example using Monte Carlo Tree Search and deep learning.

```
# Note: Full chess RL requires deep learning and self-play (e.g., AlphaZero)
import chess
import random


class SimpleChessEnv:
    def __init__(self):
        self.board = chess.Board()

    def reset(self):
        self.board.reset()
        return self.board.fen()

    def step(self, move):
        self.board.push(move)
        reward = 1 if self.board.is_checkmate() else 0
        done = self.board.is_game_over()
        return self.board.fen(), reward, done

env = SimpleChessEnv()
```

Hint:
Why is self-play crucial for learning complex games like chess?

---

19. (Project) Maze Solver with Q-learning or DQN

Answer:

Represent the maze as a grid. Use Q-learning for small mazes or DQN for larger ones. Train the agent to reach the goal efficiently.

```
import numpy as np

import random

class MazeEnv:

    def __init__(self, maze, start, goal):

        self.maze = maze

        self.start = start

        self.goal = goal

        self.actions = ['up', 'down', 'left', 'right']

        self.action_map = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), 'right': (0, 1)}

    def reset(self):

        self.state = self.start

        return self.state

    def step(self, action):

        move = self.action_map[action]

        next_state = (self.state[0] + move[0], self.state[1] + move[1])

        if (0 <= next_state[0] < len(self.maze) and

            0 <= next_state[1] < len(self.maze[0]) and

            self.maze[next_state[0]][next_state[1]] == 0):

            self.state = next_state

        reward = 1 if self.state == self.goal else -0.1

        done = self.state == self.goal

        return self.state, reward, done

maze = [[0, 0, 1, 0],

        [1, 0, 1, 0],

        [0, 0, 0, 0],

        [0, 1, 1, 0]]

env = MazeEnv(maze, (0, 0), (3, 3))
```

```
Q = {(i, j): {a: 0.0 for a in env.actions} for i in range(4) for j in range(4)}

alpha, gamma, epsilon = 0.1, 0.9, 0.2

for _ in range(500):

    state = env.reset()

    while True:

        action = random.choice(env.actions) if random.random() < epsilon else max(Q[state], key=Q[state].get)

        next_state, reward, done = env.step(action)

        best_next = max(Q[next_state], key=Q[next_state].get)

        Q[state][action] += alpha * (reward + gamma * Q[next_state][best_next] - Q[state][action])

        state = next_state

        if done: break
```

Hint:
How does partial observability affect maze navigation?

---

20. (Project) RL Agent for Stock Trading

Answer:
Use historical data to define states (price, indicators), actions (buy/sell/hold), and rewards (profit/loss). Apply DQN or Actor-Critic methods.

```
import numpy as np

import random

class StockEnv:

    def __init__(self, prices):

        self.prices = prices

        self.reset()

    def reset(self):

        self.index = 0

        self.cash = 1000
```

```
        self.stock = 0
        return self._get_state()
    def _get_state(self):
        return (self.index, self.cash, self.stock)
    def step(self, action):
        price = self.prices[self.index]
        if action == 0 and self.cash >= price:  # Buy
            self.stock += 1
            self.cash -= price
        elif action == 1 and self.stock > 0:  # Sell
            self.stock -= 1
            self.cash += price
        self.index += 1
        done = self.index >= len(self.prices)
        reward = self.cash + self.stock * price
        return self._get_state(), reward, done
prices = np.random.normal(100, 10, 200)
env = StockEnv(prices)
```

Hint:
How can market volatility impact RL agent performance?

---

21. (Project) RL Agent for Ping-Pong Game

Answer:
Model the paddle and ball dynamics. Use continuous control methods like DDPG or PPO to train the agent to hit the ball consistently.

```
# Requires physics simulation or game engine (e.g., Pygame)
class PingPongEnv:
    def __init__(self):
        self.ball_pos = [0.5, 0.5]
```

```python
        self.ball_vel = [0.01, 0.02]
        self.paddle_pos = 0.5
    def reset(self):
        self.ball_pos = [0.5, 0.5]
        self.ball_vel = [0.01, 0.02]
        self.paddle_pos = 0.5
        return self._get_state()
    def _get_state(self):
        return self.ball_pos + [self.paddle_pos]
    def step(self, action):
        self.paddle_pos += action * 0.05
        self.ball_pos[0] += self.ball_vel[0]
        self.ball_pos[1] += self.ball_vel[1]
        reward = 1 if abs(self.ball_pos[0] - self.paddle_pos) < 0.1 else -1
        done = self.ball_pos[1] > 1.0
        return self._get_state(), reward, done
# Use DQN or policy gradient to train agent
```

Hint:

What role does reaction time play in training a ping-pong agent?