VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

SCHOOL OF MECHANICAL ENGINEERING
WINTER SEMESTER 2023-2024

DIGITAL ASSIGNMENT: II

Course Code	: **BMEE407L**	**Deadline**	**:28-04-2024**
Course Name	: **Artificial Intelligence**		
Max. Marks	**: 10**		

Name : Anish Sadhotra

Reg.No : 21BME0190

Q1. Learn an AI technique and understand how it works.

Certainly! Let's delve into how Q-learning, a popular reinforcement learning technique, works in detail.

Q-Learning:

1. Problem Setup:
   - Q-learning is typically used in Markov Decision Processes (MDPs), where an agent interacts with an environment in discrete time steps.
   - The agent observes the current state \(s \) of the environment and takes an action \(a \).
   - After taking an action, the agent receives a reward \(r \) and transitions to a new state \(s' \).

2. Objective:
   - The goal of the agent is to learn a policy \(\pi \), which maps states to actions, in order to maximize the cumulative reward over time.
   - The value of a state-action pair \(Q(s, a) \) represents the expected cumulative reward if the agent starts in state \(s \), takes action \(a \), and then follows the optimal policy thereafter.

3. Q-Learning Algorithm:
   - Initialize the Q-table: Start with an arbitrary initial Q-value for each state-action pair.

- Exploration vs. Exploitation: At each time step, the agent decides whether to explore new actions or exploit the current best action based on an exploration-exploitation trade-off strategy (e.g., epsilon-greedy).
- Update Q-values: When the agent takes an action and observes the reward and next state, it updates the Q-value of the current state-action pair using the Q-learning update rule:

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

where:

- α is the learning rate, determining the extent to which new information overrides old information.
- γ is the discount factor, representing the importance of future rewards relative to immediate rewards.

4. Convergence:
- Q-learning is guaranteed to converge to the optimal Q-values under certain conditions, such as infinite exploration and appropriate decay schedules for the learning rate and exploration rate.

5. Policy Extraction:
- Once the Q-values converge, the agent can extract an optimal policy by selecting the action with the highest Q-value for each state.

6. Application:
- Q-learning can be applied to various problems, such as grid world navigation, robotic control, and game playing (e.g., Tic-Tac-Toe, Atari games).

Overall, Q-learning is a powerful technique for learning optimal policies in environments with discrete states and actions, enabling agents to make sequential decisions to maximize cumulative rewards over time.

Q2. Code it for an application in your programming language of preference

and upload your codes on your GITHUB repository.

Let's apply Q-learning to a simple grid world navigation problem. In this scenario, an agent needs to navigate through a grid world to reach a goal while avoiding obstacles. The agent receives a positive reward when it reaches the goal and a negative reward if it hits an obstacle. The objective is to learn an optimal policy that maximizes the cumulative reward.

Benefits:

Adaptability: Q-learning can adapt to changes in the environment or goals, making it suitable for dynamic scenarios.
Efficiency: Once trained, the agent can navigate the grid world efficiently, finding the optimal path to the goal.
Scalability: Q-learning can scale to larger grid worlds or more complex environments with minimal modifications.

main.py

```python
import numpy as np

class QLearningAgent:
    def __init__(self, num_states, num_actions, learning_rate=0.1, discount_factor=0.9, epsilon=0.1):
        self.num_states = num_states
        self.num_actions = num_actions
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.epsilon = epsilon
        self.q_table = np.zeros((num_states, num_actions))

    def choose_action(self, state):
        if np.random.uniform(0, 1) < self.epsilon:
            return np.random.choice(self.num_actions)  # Exploration
        else:
            return np.argmax(self.q_table[state, :])  # Exploitation

    def update_q_table(self, state, action, reward, next_state):
        best_next_action = np.argmax(self.q_table[next_state, :])
        td_target = reward + self.discount_factor * self.q_table[next_state, best_next_action]
        td_error = td_target - self.q_table[state, action]
        self.q_table[state, action] += self.learning_rate * td_error

def simulate_environment(agent, num_episodes, max_steps):
    for episode in range(num_episodes):
        state = np.random.randint(0, agent.num_states)  # Start from a random state
        for _ in range(max_steps):
            action = agent.choose_action(state)
            # Simulate taking action in the environment
            next_state, reward = simulate_step(state, action)
            agent.update_q_table(state, action, reward, next_state)
            state = next_state

def simulate_step(state, action):
    # Define transition dynamics and rewards for the grid world
    # For simplicity, assume deterministic transitions and fixed rewards
    if action == 0:  # Move up
        next_state = max(state - 3, 0)
    elif action == 1:  # Move down
        next_state = min(state + 3, 8)
    elif action == 2:  # Move left
        next_state = max(state - 1, 0)
    else:  # Move right
        next_state = min(state + 1, 8)

    if next_state == 8:  # Goal state
        reward = 1
    elif next_state in [3, 5, 7]:  # Obstacle states
        reward = -1
    else:
        reward = 0
```

```
    if next_state == 8:  # Goal state
        reward = 1
    elif next_state in [3, 5, 7]:  # Obstacle states
        reward = -1
    else:
        reward = 0

    return next_state, reward

# Main
num_states = 9  # 3x3 grid world
num_actions = 4  # Up, down, left, right
agent = QLearningAgent(num_states, num_actions)
simulate_environment(agent, num_episodes=1000, max_steps=100)

# Extract learned policy
optimal_policy = np.argmax(agent.q_table, axis=1)
print("Learned Optimal Policy:")
print(optimal_policy.reshape((3, 3)))
```

```
            return np.argmax(self.q_table[state, :])  # Exploitation

    def update_q_table(self, state, action, reward, next_state):
        best_next_action = np.argmax(self.q_table[next_state, :])
        td_target = reward + self.discount_factor * self.q_table[next_state, best_next_action]
        td_error = td_target - self.q_table[state, action]
        self.q_table[state, action] += self.learning_rate * td_error

def simulate_environment(agent, num_episodes, max_steps):
    for episode in range(num_episodes):
        state = np.random.randint(0, agent.num_states)  # Start from a random state
        for _ in range(max_steps):
            action = agent.choose_action(state)
            # Simulate taking action in the environment
            next_state, reward = simulate_step(state, action)
            agent.update_q_table(state, action, reward, next_state)
            state = next_state

def simulate_step(state, action):
    # Define transition dynamics and rewards for the grid world
    # For simplicity, assume deterministic transitions and fixed rewards
    if action == 0:  # Move up
        next_state = max(state - 3, 0)
    elif action == 1:  # Move down
        next_state = min(state + 3, 8)
```

```
Learned Optimal Policy:
[[1 0 0]
 [1 3 1]
 [1 1 1]]


...Program finished with exit code 0
Press ENTER to exit console.
```

GITHUB LINK - I have hyperlinked the text on MS Word

Q3. Describe the working of your code and the result.

The provided code implements Q-learning, a reinforcement learning technique, for solving a simple grid world navigation problem. Here's a detailed description of how the code works and the expected results:

****Working of the Code**:**

1. **Q-Learning Agent Initialization**:
   - The `QLearningAgent` class is initialized with parameters such as the number of states (`num_states`), the number of actions (`num_actions`), learning rate (`learning_rate`), discount factor (`discount_factor`), and exploration rate (`epsilon`).
   - The Q-table is initialized with zeros, representing the expected cumulative reward for each state-action pair.

2. **Action Selection**:
   - The agent chooses actions based on an exploration-exploitation strategy.
   - With probability `epsilon`, the agent explores by selecting a random action.
   - Otherwise, it exploits by selecting the action with the highest Q-value for the current state.

3. **Q-Value Update**:
   - After taking an action and observing the reward and next state, the agent updates the Q-value of the current state-action pair using the Q-learning update rule.
   - The update is based on the observed reward, the Q-value of the next state, and parameters such as the learning rate and discount factor.

4. **Environment Simulation**:
   - The `simulate_environment` function simulates the agent's interaction with the environment over multiple episodes.
   - In each episode, the agent starts from a random state and takes actions until a terminal state is reached or a maximum number of steps (`max_steps`) is reached.
   - At each step, the agent chooses an action, receives a reward, and updates its Q-values accordingly.

5. **Policy Extraction**:
   - After training, the learned optimal policy is extracted from the Q-table.
   - The optimal policy specifies the best action to take in each state based on the learned Q-values.

****Result**:**

- After running the code, the learned optimal policy is printed, showing the best action to take in each state of the grid world.
- The optimal policy should indicate the path that maximizes the cumulative reward, navigating around obstacles to reach the goal state.
- By observing the learned optimal policy, we can evaluate the effectiveness of the Q-learning algorithm in solving the grid world navigation problem.

Overall, the code demonstrates the application of Q-learning in solving a simple navigation problem, highlighting the agent's ability to learn an optimal policy through interaction with the environment and updating Q-values based on observed rewards.

Q4. Develop an idea of how this can be implemented in hardware.

Implementing Q-learning in hardware presents some unique challenges and opportunities. Here's an idea of how it could be implemented:

****Hardware Architecture**:**

1. **Custom Accelerator**:
   - Design a custom hardware accelerator optimized for Q-learning computations.
   - The accelerator should be capable of performing matrix operations efficiently, as Q-learning involves updating Q-values stored in a large Q-table.
   - Implement specialized hardware units for tasks such as matrix multiplication, element-wise operations, and memory access.

2. **Parallelism**:
   - Exploit parallelism in hardware to speed up computations.
   - Use parallel processing units to perform multiple Q-value updates simultaneously, leveraging the inherent parallel nature of Q-learning algorithms.

3. **Memory Hierarchy**:
   - Design a memory hierarchy optimized for Q-learning workloads.
   - Utilize on-chip memory for storing Q-values and intermediate computation results, reducing latency and energy consumption associated with off-chip memory access.
   - Implement efficient memory access patterns to minimize data movement and maximize memory bandwidth utilization.

4. **Scalability**:
   - Design the hardware accelerator to be scalable, allowing for parallel execution of multiple Q-learning agents or larger Q-tables.
   - Support configurable parameters such as the number of states, actions, and Q-table size to accommodate different problem sizes and complexities.

5. **Interface**:
   - Include interfaces for communication with external components, such as sensors and actuators, to interact with the environment.
   - Provide interfaces for configuring the Q-learning algorithm parameters and monitoring performance metrics.

6. **Power Efficiency**:

- Optimize the hardware design for power efficiency to enable deployment in resource-constrained environments.
- Utilize techniques such as voltage scaling, clock gating, and power gating to minimize energy consumption during idle periods and low-utilization phases.

****Integration with Embedded Systems**:**

1. **Embedded Platform**:
   - Integrate the hardware accelerator with embedded platforms such as FPGAs or ASICs to enable real-time Q-learning inference in embedded systems.
   - Provide software APIs and drivers for seamless integration with higher-level application software running on the embedded platform.

2. **Deployment in Edge Devices**:
   - Deploy the hardware-accelerated Q-learning solution in edge devices such as autonomous robots, drones, or smart IoT devices.
   - Enable on-device learning and decision-making capabilities, reducing the need for frequent communication with centralized servers and enhancing privacy and security.

3. **Application Examples**:
   - Use cases include autonomous navigation for robots or drones, adaptive control systems for industrial automation, and intelligent IoT devices for smart home applications.
   - The hardware-accelerated Q-learning solution can enable efficient and autonomous operation in real-world environments with limited computational resources and power constraints.

In summary, implementing Q-learning in hardware requires designing a custom accelerator with optimized computational units, memory hierarchy, and interfaces for efficient and scalable deployment in embedded systems. Integration with embedded platforms enables on-device learning and decision-making capabilities for a wide range of applications in robotics, automation, and IoT.