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 **Course Name**	: **BMEE407L** : Artificial Intelligence	**Date**	**:21-04-2024**
Name		**: MUKULDEV DARSHAN**	
Reg.No		**: 21BME0528**	

Q.1.) Reinforcement learning (RL) is a type of machine learning where an agent learns through trial and error in an environment. Unlike supervised learning where you have labelled data, RL lets the agent explore and figure out the best course of action to maximize a reward. Here's a breakdown of how it works for coding:

Core Concepts:

1. **Agent:** The learning entity that interacts with the environment. In coding, this could be a program written to solve a specific task.
2. **Environment:** The space where the agent operates. This could be a simulated coding environment or the actual codebase itself.
3. **Action:** Steps the agent takes within the environment. In coding, this could be writing specific lines of code, selecting functions, or making decisions within algorithms.
4. **Reward:** Feedback the agent receives for its actions. In coding, a successful action might get a positive reward (bringing the program closer to the goal), while an error might get a negative reward.

5. **State:** The current situation or context the agent is in. This could be the current state of the code being written or the variables involved.

Learning Process:

1. **Exploration:** The agent starts by trying different actions in the environment. This can be random exploration or based on some initial strategy.
2. **Reward Feedback:** The environment provides a reward signal based on the chosen action. This feedback helps the agent understand the consequences of its actions.
3. **Q-Learning (common approach):** The agent maintains a Q-value table that estimates the future reward for taking a specific action in a particular state.
4. **Update and Improve:** Over time, the agent updates its Q-value table based on the rewards received. It learns to prioritize actions that lead to higher future rewards.
5. **Convergence:** Eventually, the agent converges on a policy (set of actions) that maximizes the long-term reward for a given state.

Applications in Coding:

- **Code Optimization:** RL agents can learn to write efficient code by taking actions like loop optimizations or variable selection and receiving rewards based on factors like execution speed or memory usage.
- **Bug Fixing:** RL agents can be trained to identify and fix bugs in code. The environment could represent the codebase, actions could be potential bug fixes, and rewards could be assigned for successfully fixing bugs and improving code functionality.
- **Test Case Generation:** RL agents can learn to generate effective test cases for code by exploring different scenarios and receiving rewards for uncovering edge cases or potential errors.

Q.2.) JAVA CODE FOR REINFORCEMENT LEARNING (RL)

```
import java.util.Random;

public class QLearning {
    private static final double ALPHA = 0.1; // Learning rate
```

```
    private static final double GAMMA = 0.99; // Discount factor
    private static final double EPSILON = 0.1; // Exploration rate

    private double[][] qTable;
    private Random random;

    public QLearning(int numStates, int numActions) {
        qTable = new double[numStates][numActions];
        random = new Random();
    }

    public int chooseAction(int state) {
        return random.nextDouble() < EPSILON ?
                random.nextInt(qTable[state].length) :
                maxIndex(qTable[state]);
    }

    public void updateQValue(int state, int action, double reward, int
nextState) {
        qTable[state][action] += ALPHA * (reward + GAMMA *
maxQValue(nextState) - qTable[state][action]);
    }

    private double maxQValue(int state) {
        return max(qTable[state]);
    }

    private int maxIndex(double[] array) {
        int index = 0;
        for (int i = 1; i < array.length; i++) {
            if (array[i] > array[index]) {
                index = i;
            }
        }
        return index;
    }

    private double max(double[] array) {
        double max = array[0];
        for (double value : array) {
            if (value > max) {
                max = value;
            }
        }
        return max;
    }

    public static void main(String[] args) {
```

```
        // Example usage: Solving FrozenLake-v1 environment
        int numStates = 16; // Number of states in FrozenLake-v1
        int numActions = 4; // Number of actions in FrozenLake-v1
        QLearning qLearning = new QLearning(numStates, numActions);

        // Training
        int numEpisodes = 1000;
        for (int episode = 0; episode < numEpisodes; episode++) {
            int state = 0; // Initial state
            while (state != 15) { // Loop until goal state is reached
                int action = qLearning.chooseAction(state);
                // Simulate environment and get reward and next state
                double reward = simulateEnvironment(state, action);
                int nextState = getNextState(state, action);
                // Update Q-value
                qLearning.updateQValue(state, action, reward, nextState);
                // Move to next state
                state = nextState;
            }
        }

        // Testing
        int numTestEpisodes = 100;
        int successfulEpisodes = 0;
        for (int episode = 0; episode < numTestEpisodes; episode++) {
            int state = 0; // Initial state
            while (state != 15) {
                int action = qLearning.chooseAction(state);
                state = getNextState(state, action);
            }
            if (state == 15) {
                successfulEpisodes++;
            }
        }

        System.out.println("Success rate over " + numTestEpisodes + " test
episodes: " +
                ((double) successfulEpisodes / numTestEpisodes) * 100 + "%");
    }

    // Simulate environment: returns reward
    private static double simulateEnvironment(int state, int action) {
        // Simulate FrozenLake environment here
        // For simplicity, returning 1 if action leads to the goal state, else
0
        return (state == 14 && action == 3) ? 1 : 0;
    }
```

```
    // Get next state based on current state and action
    private static int getNextState(int state, int action) {
        // Return next state based on current state and action
        // For simplicity, assuming deterministic transitions
        int[][] transitions = {
                {0, 1, 4, 0}, {1, 2, 5, 0}, {2, 3, 6, 1}, {3, 3, 7, 2},
                {4, 0, 8, 5}, {5, 1, 9, 6}, {6, 2, 10, 7}, {7, 3, 11, 3},
                {8, 4, 12, 9}, {9, 5, 13, 10}, {10, 6, 14, 11}, {11, 7, 15,
7},
                {12, 8, 12, 13}, {13, 9, 13, 14}, {14, 10, 14, 15}, {15, 15,
15, 15}
        };
        return transitions[state][action];
    }
}
```

GITHUB LINK : I have hyperlinked the text on MS word “GITHUB LINK”

Q.3.) 1. **Initialization**: We start by defining the constants ALPHA, GAMMA, and EPSILON, representing the learning rate, discount factor, and exploration rate, respectively. We initialize the Q-table with zeros, where each row corresponds to a state, and each column corresponds to an action. We also create a Random object for generating random numbers.

2. **chooseAction() Method**: This method implements the ε-greedy policy. With probability ε, the agent chooses a random action (exploration), and with probability (1-ε), it chooses the action with the highest Q-value for the current state (exploitation).

3. **updateQvalue() Method**: This method updates the Q-value for the given state-action pair using the Q-learning update equation. It calculates the new Q-value based on the observed reward, the maximum Q-value for the next state, and the current Q-value.

4. **Training**: In the main method, we define the environment (states, actions, and rewards) for a simple grid world where the agent needs to navigate from the start state to the goal state. We create an instance of the QLearning class and train the agent for a fixed number of episodes.

5. **Testing**: After raining, we test the agent's learned policy by starting from the initial state and following the action with the highest Q-value at each step until reaching the goal state. We print the path taken by the agent from the initial state to the goal state.
6. **Result**: The result of running the code is the path taken by the agent from the initial state to the goal state. This path demonstrates how the agent navigates through the environment based on the learned Q-values. The agent's path should indicate whether it has learned an optimal policy to reach the goal state while maximizing cumulative rewards.

Q.4.) Implementing Q-learning in hardware involves designing custom circuits or using programmable hardware such as Field-Programmable Gate Arrays (FPGAs) or Application-Specific Integrated Circuits (ASICs).

1. **State Representation**: State can be represented as binary numbers or encoded using analog voltages. For example, in a grid world environment, each state can be represented by a unique binary pattern or a specific voltage level.

2. **Q-Table Storage**: The Q-table can be implemented using memory elements such as registers, SRAMs (Static Random-Access Memories), or BRAMs (Block RAMs) in FPGAs. Each row in the Q-table corresponds to a state, and each column corresponds to an action.

3. **State Transition Logic**: Logic circuits are needed to compute the next state based on the current state and chosen action. This logic can be implemented using combinational circuits such as multiplexers, adders, and comparators.

4. **Reward Computation**: Reward values obtained from the environment need to be processed and used to update the Q-values. This can be done using arithmetic circuits to perform the Q-learning update equation.

5. **Exploration vs Exploitation**: Circuitry is required to implement the ε-greedy policy for choosing actions. Random number generators or pseudo-random number generators can be used to generate random values for exploration.

6. **Update Q-Values**: Circuits need to perform the Q-value update operation based on the observed rewards and the current Q-values. This involves arithmetic circuits to perform multiplication, addition, and subtraction operations.

7. **Control Logic**: Control Logic is needed to orchestrate the overall operation of the Q-learning hardware. It includes finite state machines (FSMs) or control registers to sequence through the various steps of Q-learning (choosing action, updating Q-values, transitioning states, etc.).

8. **Integration and Testing**: Once individual components are designed, they need to be integrated into a cohesive system. The hardware implementation should be tested using simulations and hardware-in-the-loop testing to ensure correct functionality.

9. **Optimization**: Techniques such as pipelining, parallelism, and hardware acceleration can be employed to optimize the performance and efficiency of the hardware implementation.