Batch: C2 **Roll No.:** 16010121110
Experiment / assignment / tutorial No.____1__
Grade: AA / AB / BB / BC / CC / CD /DD
Signature of the Staff In-charge with date

TITLE: 1. To represent any 03 real world problems as Markov Decision Process

AIM: Markov Decision Process

Expected OUTCOME of Experiment : (Mention CO/CO's attained here): CO1

Books/ Journals/ Websites referred:

Richard S. Sutton and Andrew G. Barto, "*Reinforcement Learning:An Introduction",* The MIT Press,Second Edition, 2018

Pre Lab/ Prior Concepts:

WHAT IS THE MARKOV DECISION PROCESS?

The Markov decision process (MDP) is a mathematical tool used for decision-making problems where the outcomes are partially random and partially controllable.

Agent: A reinforcement learning agent is the entity which we are training to make correct decisions. For example, a robot that is being trained to move around a house without crashing.

Environment: The environment is the surroundings with which the agent interacts. For example, the house where the robot moves. The agent cannot manipulate the environment; it can only control its own actions. In other words, the robot can't control where a table is in the house, but it can walk around it.

State: The state defines the current situation of the agent This can be the exact position of the robot in the house, the alignment of its two legs or its current posture. It all depends on how you address the problem.

Action: The choice that the agent makes at the current time step. For example, the robot can move its right or left leg, raise its arm, lift an object or turn right/left, etc. We know the set of actions (decisions) that the agent can perform in advance.

Policy: A policy is the thought process behind picking an action. In practice, it's a probability distribution assigned to the set of actions. Highly rewarding actions will have a high probability and vice versa. If an action has a low probability, it doesn't mean it won't be picked at all. It's just less likely to be picked.

What is the Markov property?

The Markov property is a fundamental concept in probability theory and stochastic processes. It states that the future state of a system depends only on its current state, independent of how it arrived at that state. Formally, for a stochastic process $X(t)$ X(t), the Markov property can be expressed as $P(X(t+\Delta t) \mid X(t)) = P(X(t+\Delta t) \mid X(t), X(t-\Delta t), \dots)$ P(X(t+Δt)∣X(t))=P(X(t+Δt)∣X(t),X(t−Δt),…), where Δt Δt is a small time interval. This property implies that the past history of the process beyond the current state has no additional predictive power for the future evolution, making it invaluable in modeling and analyzing various real-world phenomena.

Markov reward process (MRP):

A Markov Reward Process (MRP) is a stochastic process that extends the concept of a Markov chain by associating rewards with each state transition. It consists of a tuple S,P,R,γ):

1. State Space S: A finite or countably infinite set of states.
2. Transition Probability Matrix P: Defines the probability of moving from one state to anotherReward Function R: Specifies the immediate reward received upon transitioning between states.
3. Discount Factor γ: A factor between 0 and 1 that discounts the value of future rewards to account for the uncertainty of future events and the preference for immediate rewards.

Markov decision process (MDP):

Markov Decision Process (MDP) is a mathematical framework used to model decision-making problems where outcomes are partially random and partially under the control of a decision-maker. It extends the concept of a Markov reward process by incorporating decisions and actions. Key components of an MDP include:

1. State Space S: A finite or countably infinite set of states representing the possible situations or configurations of the system.
2. Action Space A: A finite set of actions that the decision-maker can choose from in each state.
3. Transition Probability : Defines the probability of transitioning to state s' when action a is taken in state s.
4. Reward Function : Specifies the immediate reward received after transitioning from state sss to state s′ by taking action a
5. **Policy π**: A strategy or rule that determines the action to be taken in each state.

Representation of any 03 real world problems as Markov Decision Process:

Problem 1 : Autonomous vacuum cleaner

R = {+ 10 for cleaning a tile. +100 For cleaning whole area}

A = {Move ahead, turn left, turn right}

P = {Move ahead -	0 : if obstacle detected ahead
	0.25 : if clean tile
	0.75
Turn left -	0.5 if obstacle detected
	0.35 if clean tile ahead
	0.17 if unclean tile ahead
Turn right -	0.5 if obstacle detected ahead
	0.35 if unclean tile ahead
	0.17 if clean tile ahead}

gamma = 0.9

S = {Clean tile ahead, obstacle ahead, unclean tile ahead, all tiles cleaned }

Problem 2: Chess player

R = {+1 for every pawn taken, +3 for knight, +4 for rook, +6 for queen, +100 for checkmate}
A = {move pieces, move pawns legally}
P = {Maximize the chance of winning in the min max tree}

Gamma = 0.9

S = {Every position possible in chess game}

Problem 3: Airport ATC

R = {+10 for every plane landed, -20 for every plane delayed, -100 for every plane waiting in air}
A = {Give/deny permission to land, Give/deny permission to takeoff}
P = {Give permission to land - 0.95 if another plane not landing
0 if another plane landing
Give permission to takeoff - 0.95 if another plane not taking off
0 if another plane taking off
Gamma = 0.9

S = {Planes parked in Parking slots P1-Pn, Plane taking off, Plane landing, Plane waiting in air}

Parking lot -

P1	P2	P3	P4

Plane taking off : True or False

Plane Landing: True or False

Planes Waiting in air - list {p1,p2,p3}

Autonomous vacuum cleaner:

The objective for the autonomous vacuum cleaner is to maximize the total reward it receives over time by efficiently cleaning the tiles in the environment. The ultimate goal is to clean the entire area (+100 reward) while collecting rewards for cleaning individual tiles (+10 reward per tile).

1. **States (S)**:
   - **Clean tile ahead**: The vacuum cleaner detects that the tile in front of it is already clean.
   - **Obstacle ahead**: An obstacle is detected in front of the vacuum cleaner, preventing it from moving forward.
   - **Unclean tile ahead**: The vacuum cleaner detects that the tile in front of it is dirty and needs cleaning.
   - **All tiles cleaned**: Terminal state indicating that the entire area has been cleaned.
2. **Actions (A)**:
   - **Move ahead**: Attempts to move the vacuum cleaner forward.
   - **Turn left**: Rotates the vacuum cleaner 90 degrees counterclockwise.
   - **Turn right**: Rotates the vacuum cleaner 90 degrees clockwise.
3. **Rewards (R)**:
   - **+10**: Received when the vacuum cleaner successfully cleans a tile.
   - **+100**: Received when the entire area has been cleaned.
4. **Transition Probabilities (P)**:
   - These specify the probability of the next state given the current state and action.
   - **Move ahead**:
     - 0 probability if an obstacle is detected (vacuum cleaner cannot move).
     - 0.25 probability if the tile ahead is clean.
     - 0.75 probability if the tile ahead is dirty (unclean).
   - **Turn left**:
     - 0.5 probability of an obstacle.
     - 0.35 probability of a clean tile.
     - 0.17 probability of an unclean tile.
   - **Turn right**:
     - 0.5 probability of an obstacle.
     - 0.35 probability of an unclean tile.
     - 0.17 probability of a clean tile.
5. **Discount factor (gamma)**:

- The discount factor used in the reinforcement learning process, typically set to 0.9 in this case.

Chess Player:

1. **States (S)**:
   - **Every position possible in a chess game**: This includes all possible board configurations during a chess game. Each state represents a unique arrangement of pieces on the chessboard.
2. **Actions (A)**:
   - **Move pieces**: The agent can make legal moves with any of its pieces on the board.
   - **Move pawns legally**: Specific action for moving pawns, implying a specialized handling for pawns.
3. **Rewards (R)**:
   - **+1 for every pawn taken**: Incremental reward for capturing an opponent's pawn.
   - **+3 for knight**: Reward for capturing an opponent's knight.
   - **+4 for rook**: Reward for capturing an opponent's rook.
   - **+6 for queen**: Reward for capturing an opponent's queen.
   - **+100 for checkmate**: Reward received when the opponent's king is checkmated, indicating a win.
4. **Transition Probabilities (P)**:
   - Maximize the chance of winning in the min max tree.
5. **Discount factor (gamma)**:
   - The discount factor (gamma) is set to 0.9, indicating the importance of future rewards in the agent's decision-making process. A higher gamma places more emphasis on long-term rewards compared to short-term gains.

Air Traffic controller:

1. **States (S)**:
    - **Planes parked in Parking slots P1-Pn**: Indicates planes that are on the ground and parked at designated parking slots.
    - **Plane taking off**: Represents a plane that is preparing for or in the process of taking off.
    - **Plane landing**: Represents a plane that is preparing for or in the process of landing.
    - **Plane waiting in air**: Indicates a plane that is circling or holding in the air, waiting for permission to land.
2. **Actions (A)**:
    - **Give permission to land**: ATC grants permission for a plane to land.
    - **Deny permission to land**: ATC denies permission for a plane to land.
    - **Give permission to takeoff**: ATC grants permission for a plane to take off.
    - **Deny permission to takeoff**: ATC denies permission for a plane to take off.
3. **Rewards (R)**:
    - **+10 for every plane landed**: Reward for successfully guiding a plane to land.
    - **-20 for every plane delayed**: Penalty incurred for delaying a plane (possibly due to traffic or congestion).
    - **-100 for every plane waiting in air**: Significant penalty for planes circling in the air, as it indicates inefficiency in managing air traffic.
4. **Transition Probabilities (P)**:
    - **Give permission to land**:
        - Probability of 0.95 if another plane is not currently landing (success scenario).
        - Probability of 0 if another plane is currently landing (failure scenario).
    - **Give permission to takeoff**:
        - Probability of 0.95 if another plane is not currently taking off (success scenario).
        - Probability of 0 if another plane is currently taking off (failure scenario).
5. **Discount factor (gamma)**:
    - The discount factor (gamma) is set to 0.9, emphasizing the importance of future rewards in the decision-making process. It encourages the ATC system to prioritize immediate actions that lead to long-term efficiency and reduced penalties.

Post Lab Descriptive Questions:

Discuss following extension to MDPs:

1) **Infinite and Continuous MDPs:**

Traditional MDPs assume a finite number of states and actions. In reality, many decision-making problems involve infinite or continuous state and/or action spaces. For example, in robotics, the state space could represent the robot's position and orientation in a continuous 3D space, and actions could represent velocities or torques. Dealing with infinite or continuous MDPs requires specialized algorithms that can approximate value functions and policies in such spaces. Techniques like function approximation, such as using neural networks or kernel methods, are often employed to handle the continuous nature of states and actions.

2) **Partially Observable MDPs (POMDPs):**

In standard MDPs, the agent has full knowledge of the environment state when making decisions. In contrast, POMDPs deal with situations where the agent does not directly observe the state but instead receives observations that are probabilistically related to the underlying state. This adds a layer of uncertainty and requires the agent to maintain beliefs (probability distributions over states) and update them based on observations and actions. Solving POMDPs involves using techniques such as belief state representation, which encapsulates the uncertainty about the true state of the environment.

3) **Undiscounted, Average Reward MDPs:**

Traditional MDPs involve discounting future rewards to prioritize immediate rewards more than distant rewards. Undiscounted MDPs, on the other hand, do not discount future rewards, treating all future rewards equally. This can be useful in scenarios where long-term planning or the horizon of interest is infinite or very long, and discounting becomes less meaningful. Average reward MDPs focus on maximizing the average reward per time step rather than the total discounted reward over the entire episode. This formulation is particularly relevant in settings where the agent operates indefinitely or where there is no natural termination of episodes.

Date: 22 July 2024 **Signature of faculty in-charge**

Batch: C2	Roll No.: 16010121110
Experiment / assignment / tutorial No.______	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

TITLE: 2. To implement dynamic programming policy iteration approach

AIM: Dynamic programming in RL

Expected OUTCOME of Experiment: (Mention CO/CO's attained here):

CO2

Books/ Journals/ Websites referred:

Richard S. Sutton and Andrew G. Barto, "*Reinforcement Learning:An Introduction",* The MIT Press,Second Edition, 2018

Pre Lab/ Prior Concepts:

Dynamic Programming is a very general solution method for problems which have two properties:

1. Optimal substructure
2. Principle of optimality applies

Optimal solution can be decomposed into subproblems.

Overlapping subproblems Subproblems recur many times

Solutions can be cached and reused. Markov decision processes satisfy both properties. Bellman equation gives recursive decomposition Value function stores and reuses solutions.
Dynamic programming assumes full knowledge of the MDP. It is used for planning in an MDP.

Chosen Problem Statement: Frozen lake problem

The Frozen Lake environment is a grid of frozen ice, holes, and a goal, here we will have (4*4) grid size, that's mean we will have (16 STATES) each cell repersent a state.

The agent starts at the top left corner cell in the grid, and can take four actions at each time step, We will have (4 ACTIONS):

move up
move down
move left
move right
The goal is to reach the goal cell in the bottom right corner of the grid without falling through any holes. Therefore, our REWARDS are:

If agent current state is a hole state, the reward = -1
If agent current state is the goal state, the reward = +1
Else, the reward = 0

(Reference - https://sciml-leeds.github.io/workshop/reinforcement_learning/2023/05/05/Reinforcement_learning_Solution.html#:~:text=The%20Frozen%20Lake%20example%20is,to%20navigate%20the%20FrozenLake%20environment.)

Explain following concepts w.r.t. chosen problem statement:

1. Policy:

A policy defines the strategy or the mapping from states to actions. In the Frozen Lake environment, the policy dictates what action an agent should take when it finds itself in a particular state. This can be represented as a deterministic policy (where a specific action is chosen for each state) or a stochastic policy (where a probability distribution over actions is given for each state). For example, the policy could tell the agent to move left, right, up, or down based on its current position on the lake.

2. Reward Function:

The reward function specifies the immediate reward received after taking an action in a particular state. In the Frozen Lake environment, the reward function assigns a reward value based on the outcome of the agent's actions. Typically, the reward function gives:

- A positive reward (often +1) for reaching the goal state.
- A negative reward or zero for falling into a hole (which is a failure state).
- Zero reward for all other actions taken in non-terminal states.

3. Value Function:

The value function estimates the expected cumulative reward an agent can achieve from a given state, following a particular policy. It helps determine how good it is to be in a specific state (or to take a particular action in that state). In the Frozen Lake environment, the value function would help estimate the long-term reward the agent can expect starting from any state, given the current policy. It can be divided into:

- **State Value Function (V(s))**: The expected return starting from state sss and following the policy.
- **Action Value Function (Q(s, a))**: The expected return of taking action aaa in state sss and then following the policy.

4. Model of the Environment:

A model of the environment refers to the agent's representation or understanding of how the environment behaves. It includes:

- **State Transition Model**: Describes the probability of moving from one state to another given a specific action.
- **Reward Model**: Describes the reward received after transitioning from one state to another given an action.

In the Frozen Lake environment, the model could describe the dynamics of moving across the lake, the likelihood of falling into a hole, and the probability of successfully reaching the goal. In many RL problems, especially in simpler environments like Frozen Lake, a model might not be explicitly provided, and the agent learns the dynamics through interaction.

Conclusion: Thus we have understood dynamic programming of a toy problem - frozen lake problem. We have understood how dynamic programming works for Reinforcement Learning.In the Frozen Lake environment, reinforcement learning concepts such as policy, reward function, value function, and model of the environment help in understanding and improving the agent's behavior. The policy dictates the actions to take, the reward function evaluates those actions, the value function estimates the long-term benefits, and the model describes the environment's dynamics. Then we can find the optimal policy by the bellman functions.

<u>Post Lab Descriptive Questions:</u>

What is the Synchronous, Asynchronous and Approximate Dynamic Programming approach as applicable to RL problems ?

1. Synchronous Dynamic Programming

Synchronous Dynamic Programming is a classical approach where all state values are updated simultaneously in each iteration. This method ensures that updates to the value function or policy are made in a coordinated manner across all states.

Update Mechanism: In each iteration, the algorithm updates the values of all states or actions based on the current value estimates of other states.

Example Algorithms: Value Iteration, Policy Iteration.

Pros: Guarantees convergence to the optimal policy under certain conditions; provides exact solutions when the state space is small.

Cons: Computationally expensive and impractical for large state spaces due to the need to update all states at once.

2. Asynchronous Dynamic Programming

Asynchronous Dynamic Programming, as opposed to synchronous methods, updates the value function or policy incrementally and independently for different states or actions. This approach does not require a global synchronization of updates.

Update Mechanism: Updates are performed on different states or actions in a non-simultaneous manner. The algorithm may select states or actions to update based on certain criteria or randomly.

Example Algorithms: Asynchronous Value Iteration, Asynchronous Policy Iteration.

Pros: Can be more computationally efficient and flexible, especially in large state spaces. It allows for partial updates and can be more scalable.

Cons: May lead to slower convergence and less stability compared to synchronous approaches. The final policy or value function might be less accurate due to the non-uniform update process.

3. Approximate Dynamic Programming

Approximate Dynamic Programming (ADP) is used when the state or action space is too large to handle exactly. Instead of computing exact values, ADP uses approximation techniques to estimate value functions and policies.

Update Mechanism: Uses function approximation methods (like linear functions, neural networks) to estimate the value function or policy. Updates are based on these approximations rather than exact values.

Example Techniques: Temporal Difference Learning (TD), Q-Learning, SARSA with function approximation, Deep Q-Networks (DQN), Policy Gradient methods.

Pros: Scales to large or continuous state spaces; more feasible for complex problems where exact solutions are impractical.

Cons: Approximation errors can lead to suboptimal policies; convergence might be slower and less guaranteed compared to exact methods.

Date: 3 Aug 24 **Signature of faculty in-charge**

Batch: C2	**Roll No.:** 16010121110
Experiment / assignment / tutorial No.________	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

TITLE : 3. To implement Monte Carlo approach

AIM : MC (model-free)

Expected OUTCOME of Experiment: (Mention CO /CO's attained here): CO2

Books/ Journals/ Websites referred:

Richard S. Sutton and Andrew G. Barto, "*Reinforcement Learning:An Introduction",* The MIT Press,Second Edition, 2018

Pre Lab/ Prior Concepts:

Monte Carlo methods learn directly from episodes of experience. MC is model-free. No knowledge of MDP transitions / rewards is needed. MC learns from complete episodes: no bootstrapping is needed. MC uses the simplest possible idea: value = mean return. Hence we can only apply MC to episodic MDPs.

First visit MC evaluation and every visit MC evaluation should be studied before implementation.

Problem Statement: Blackjack using Monte Carlo Methods

Blackjack is a popular card game where the goal is to accumulate cards with a total value as close to 21 as possible without exceeding it. The game is typically played between a player and a dealer, where both follow specific rules to make decisions on whether to "hit" (take another card) or "stand" (stop taking cards). The player's objective is to beat the dealer by having a higher card total without going over 21.

Monte Carlo methods are used to optimize the player's strategy by estimating the value of different actions in different game states, based on repeated simulations of the game.

Policy:

A policy defines the strategy that the player follows in any given state. In the context of Blackjack, the policy could be a set of rules that determine whether the player should hit, stand, or take other actions based on the current state of the game (such as the player's current hand, the dealer's visible card, etc.).

For example, a simple policy might be:

Hit if your hand total is less than 17.

Stand if your hand total is 17 or higher.

The Monte Carlo method is used to evaluate and improve the policy by simulating many games and adjusting the policy based on the observed outcomes.

Reward Function:

The reward function assigns a numerical value to the outcome of the game. In Blackjack, rewards are typically assigned as follows:

+1 if the player wins (by having a hand total closer to 21 than the dealer without exceeding 21).

-1 if the player loses (by exceeding 21 or having a lower total than the dealer).

0 if the game is a draw (both the player and dealer have the same total).

The reward function helps in guiding the learning process by providing feedback on the effectiveness of the policy.

Value Function:

The value function estimates the expected return (total reward) from a particular state or state-action pair, given that the player follows a specific policy. There are two types of value functions:

State Value Function (V(s)): The expected reward when starting from state s and following the policy thereafter. State-Action Value Function (Q(s, a)): The expected reward when starting from state *s*, taking action *a*, and then following the policy.

Monte Carlo methods estimate these value functions by averaging the rewards obtained from multiple simulations of the game.

Environment:

The environment refers to everything outside the player's control that affects the game, including the dealer's actions and the randomness of card draws. In Blackjack, the environment encompasses the rules of the game, the deck of cards, and how the dealer plays (e.g., hitting until reaching at least 17).

Monte Carlo methods interact with this environment by simulating the game many times, observing the outcomes, and using this information to update the policy.

Conclusion:

In the context of Blackjack, using Monte Carlo methods allows the player to learn an optimal strategy by simulating many games and observing the outcomes. The policy dictates the player's actions, the reward function provides feedback based on game results, the value function estimates the potential benefits of different actions, and the environment provides the context in which these decisions are made.

Through repeated simulations, the player can refine their policy to maximize their chances of winning, thereby improving their overall performance in the game. This process of learning from experience (simulated in this case) is at the heart of the Monte Carlo approach in reinforcement learning.

Post Lab Descriptive Questions:

What do you understand by incremental Monte Carlo updates for policy evaluation?

Incremental policy update $V(s) \leftarrow V(s)+\alpha(G_t - V(s))$ updates the policy each iteration. This is a better notation than batch update and can be extended to include temporal difference learning. It has the following benefits -

Memory Efficiency: Incremental updates eliminate the need to store all past episodes or returns, as the value estimates are updated on-the-fly after each episode.

Memory Efficiency: Incremental updates eliminate the need to store all past episodes or returns, as the value estimates are updated on-the-fly after each episode.

Real-Time Learning: The agent can continuously improve its policy while interacting with the environment, rather than waiting for the completion of all episodes before making updates.
Stability: By using a small step size α\alphaα, incremental updates can provide a smoother and more stable convergence to the true value function, reducing the variance in the estimates.

The agent can continuously improve its policy while interacting with the environment, rather than waiting for the completion of all episodes before making updates.

Stability: By using a small step size α, incremental updates can provide a smoother and more stable convergence to the true value function, reducing the variance in the estimates.

Date: 19 Aug 2024 **Signature of faculty in-charge**

Batch: C2	Roll No.: 16010121110
Experiment / assignment / tutorial No.________	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

TITLE : 4 To implement epsilon-greedy policy approach

AIM: epsilon-greedy policy approach for Monte Carlo approach

Expected OUTCOME of Experiment: (Mention CO/CO's attained here): CO2

Books/ Journals/ Websites referred:

Richard S. Sutton and Andrew G. Barto, "*Reinforcement Learning:An Introduction",* The MIT Press,Second Edition, 2018

Pre Lab/ Prior Concepts:

Exploration allows an agent to improve its current knowledge about each action, hopefully leading to long-term benefit. Improving the accuracy of the estimated action-values, enables an agent to make more informed decisions in the future.

Exploitation on the other hand, chooses the greedy action to get the most reward by exploiting the agent's current action-value estimates. But by being greedy with respect to action-value estimates, may not actually get the most reward and lead to sub-optimal behaviour.

When an agent explores, it gets more accurate estimates of action-values. And when it exploits, it might get more reward. It cannot, however, choose to do both simultaneously, which is also called the exploration-exploitation dilemma.

Epsilon-Greedy is a simple method to balance exploration and exploitation by choosing between exploration and exploitation randomly. The epsilon-greedy, where epsilon refers to the probability of choosing to explore, exploits most of the time with a small chance of exploring.

Chosen Problem Statement: Adversarial Attacks on Neural Networks

An adversarial attack refers to an attack on a neural network where a subtle carefully designed perturbation to the input leads to incorrect predictions while the original input is still classified correctly.

Although the change may be invisible to the human eye, the model considers it important enough to change its prediction.

Explain following concepts w.r.t. chosen problem statement:

Policy: Choice of Noise maximizing the policy

Reward function: Loss in error of prediction of a sample.

Value function: Choice of noise that gives best reward for sample. Initially 0.5

Model of the environment: Neural network

Conclusion: Thus we have performed attacks on neural networks using epsilon greedy policy. We used monte carlo methods with epsilon greedy policy to attack a simple MNIST neural network. We successfully performed the denial attack on a class sample demonstrating the effectiveness of monte carlo methods.

Empirically obtained optimal values of epsilon: 0.999, 1

The RL Monte Carlo model converges when exploration is preferred over-exploitation as indicated by the high values of epsilon

Post Lab Descriptive Questions:

Explain the concept of greedy exploration in RL.

In Reinforcement Learning (RL), **greedy exploration** refers to a strategy where an agent consistently selects the action that it believes will yield the highest immediate reward based on its current knowledge or learned policy. This approach prioritizes **exploitation** of the known best actions, rather than exploring new or uncertain actions that might lead to even better outcomes in the long term.

Key Concepts:

1. **Exploitation vs. Exploration**:
   - **Exploitation**: Choosing actions that have provided the highest rewards in the past.
   - **Exploration**: Trying out new or less-known actions to discover

potentially better rewards.

2. **Greedy Strategy**:
   - In a pure **greedy strategy**, the agent always picks the action with the highest estimated reward, completely ignoring the possibility of discovering better actions through exploration.
3. **Drawbacks of Pure Greedy Strategy**:
   - **Suboptimal Performance**: If the agent starts with limited or inaccurate knowledge, a purely greedy strategy may lock the agent into a suboptimal policy because it never explores actions that might lead to higher rewards in the long run.
   - **Local Optima**: The agent might settle on a local optimum (a solution that's better than nearby alternatives but not the best possible overall solution) because it doesn't explore sufficiently to find the global optimum.
4. **Epsilon-Greedy Strategy**:
   - To balance exploitation and exploration, a common approach is the **epsilon-greedy strategy**. Here, the agent follows a greedy policy most of the time (with probability 1−ϵ1 - \epsilon1−ϵ), but with a small probability ϵ\epsilonϵ, it randomly selects an action to explore new possibilities. This helps in avoiding the pitfalls of a purely greedy approach.

Greedy exploration in RL focuses on exploiting the current best-known actions but can lead to suboptimal performance if it doesn't incorporate sufficient exploration. To address this, methods like epsilon-greedy are often used, where the agent primarily exploits but occasionally explores to improve long-term outcomes.

Date: 7 sept 2024 **Signature of faculty in-charge**

Batch:	**Roll No.:**
Experiment / assignment / tutorial No.________	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

TITLE: 5 To implement temporal difference approach

AIM: T-D learning and prediction

Expected OUTCOME of Experiment: (Mention CO attained here): CO3

Books/ Journals/ Websites referred:

Richard S. Sutton and Andrew G. Barto, "*Reinforcement Learning:An Introduction",* The MIT Press,Second Edition, 2018

Pre Lab/ Prior Concepts:

TD methods learn directly from episodes of experience. TD is model-free: no knowledge of MDP transitions / rewards is required. TD learns from incomplete episodes, by bootstrapping. TD updates a guess towards a guess. TD can learn before knowing the final outcome. TD can learn online after every step.

Chosen Problem Statement:

1) **Path home on 3x3 grid**

2) Random walk on below state transition diagram

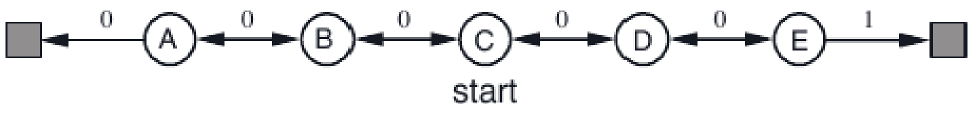


Figure 6.5: A small Markov process for generating random walks.

Explain following concepts w.r.t. chosen problem statement:

Policy:

The policy defines the actions an agent should take when in each state. In our random walk case, the policy determines whether the agent should move left or right in a given state to maximize the cumulative reward.

After applying Temporal Difference learning and analyzing the state values, the optimal policy for each non-terminal state (B, C, D) can be described as:

State B: Move right

State C: Move right (starting state)

State D: Move right

Reward function: The rewards gained. For the path home problem, reward of 1 is gained on reachine the last state. For the random walk problem, the reward of 1 is gained when it does the last transition.

Value function: The estimated rewards of each state. This is calculated by the random walks and updating the values.

For random walk problem it is -

```
Learned State-Value Function:

State A: 0.00

State B: 0.14

State C: 0.38

State D: 0.53

State E: 0.70

State F: 0.85

State G: 0.00
```

Model of the environment:

For the random walk:

State Transition Function (P): The state transition is stochastic, with equal probability of moving left or right from any non-terminal state.

Reward Function (R): This function assigns a reward of 1 for reaching the terminal state E and 0 otherwise.

Conclusion:

Thus we have implemented Temporal difference Learning in two problem statements, random walk and path home. We have understood how TDL works. TDL updates the states based on the intermediate rewards obtained be estimating the rewards. TDL does not require the full episode to be completed before the rewards are calculated.

Post Lab Descriptive Questions:

List advantages and disadvantages of MC and TD approach.

Monte Carlo (MC) Methods

Advantages:

1. **Simplicity:** MC methods are conceptually straightforward. They estimate the value of states or actions based on complete episodes of experience, making them easy to understand and implement.
2. **No Need for a Model:** MC methods do not require a model of the environment. They learn directly from the experiences gathered from the episodes.
3. **Convergence:** MC methods converge to the true value function if the policy being followed is fixed and each state-action pair is visited infinitely often.

Disadvantages:

1. **High Variance:** The estimates of the value function can have high variance because they depend on the returns from complete episodes, which can be noisy.
2. **Complete Episodes Required:** MC methods require complete episodes to update values, which can be inefficient if episodes are long or if the environment is complex.
3. **Slow Learning:** Learning can be slow because updates only occur at the end of episodes, which might be infrequent or sparse in some environments.

Temporal Difference (TD) Methods

Advantages:

1. **Efficiency:** TD methods update estimates based on each step in an episode, which allows for more frequent updates and potentially faster learning compared to MC methods.
2. **Lower Variance:** TD methods generally have lower variance in their updates compared to MC methods because they use estimates of the future value rather than complete returns.
3. **Online Learning:** TD methods can be used in online learning scenarios, where data arrives sequentially and the agent can learn from it as it comes.

Disadvantages:

1. **Complexity:** TD methods can be more complex to implement and understand, particularly with regard to how they update estimates based on partially completed episodes.
2. **Bias:** TD methods introduce bias because they rely on bootstrapping (estimating the value function based on other estimates rather than complete returns), which can affect the accuracy of the value function.
3. **Convergence Issues:** While TD methods generally converge, they may sometimes suffer from instability or divergence, especially with function approximation or when the learning parameters are not well-tuned.

Date: 27 Aug 24 **Signature of faculty in-charge**

Batch:	Roll No.: 110
Experiment / assignment / tutorial No.______	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

TITLE: 6 To implement SARSA approach.

AIM: understand SARSA

Expected OUTCOME of Experiment: (Mention CO/CO's attained here): CO3

Books/ Journals/ Websites referred:

Richard S. Sutton and Andrew G. Barto, "*Reinforcement Learning:An Introduction*", The MIT Press,Second Edition, 2018

Pre Lab/ Prior Concepts:

SARSA is an on-policy algorithm used in reinforcement learning to train a Markov decision process model on a new policy. It's an algorithm where, in the current state, S, an action, A, is taken and the agent gets a reward, R, and ends up in the next state, S1, and takes action, A1, in S1, or in other words, the tuple S, A, R, S1, A1.

It's called an on-policy algorithm because it updates the policy based on actions taken. The updation equation for SARSA is as follows;

$$Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma Q(S', A') - Q(S, A))$$

Chosen Problem Statement:

Windy gridworld

Explain following concepts w.r.t. chosen problem statement:

Policy: 0.25 for every action (initial policy)

Reward function: Euclidian distance from goal

Value function: 0 initially

Model of the environment: Fixed wind gridworld.

Grid Layout: The environment is typically represented as a two-dimensional grid. Each cell in the grid can be occupied by the agent, the goal, or be empty.

Agent's Actions: The agent can perform a limited set of actions, usually moving up, down, left, or right.

Wind Influence: In this problem, certain rows or columns of the grid produce wind that pushes the agent in a specific direction (e.g., to the right). This means that even if the agent intends to move in a certain direction, the wind may alter its actual movement.

Goal State: The objective is for the agent to reach a designated goal state, which is often located at the opposite corner of the grid.

Conclusion:

We have implemented SARSA in python. SARSA is state action reward state action. It is a bootstrap method that estimates the value of the next state. SARSA is commonly used in environments like gridworlds, robotic control, and game playing, where it helps agents learn optimal policies through trial and error. Its on-policy nature makes it particularly suited for situations where the agent needs to learn directly from its actions and experiences.

<u>Post Lab Descriptive Questions</u>

Why is it said that the convergence of the SARSA algorithm can be slow? Explain with examples.

The convergence of the SARSA (State-Action-Reward-State-Action) algorithm can be slow due to several factors related to its inherent design and the nature of the environment in which it operates. Here are some reasons for its slow convergence, along with examples to illustrate each point.

If the agent primarily uses an ε-greedy strategy with a low ε (meaning it explores very little), it may get stuck in local optima. For instance, in a complex gridworld with multiple paths to the goal, if the agent only explores a limited subset of states, it might miss better routes, resulting in slow learning.

In environments with large state-action spaces, the number of updates needed for convergence can be enormous. For example, in a 10x10 grid with multiple actions, if the agent rarely visits certain state-action pairs, it will take a long time to learn optimal actions for those pairs.

If the learning rate (α) is set too low, updates to the Q-values will be minimal, slowing down the convergence. For instance, if $\alpha = 0.01$, and the agent encounters a rewarding state, the change to the Q-value may be negligible, causing the agent to learn very slowly.

SARSA updates Q-values based on the current policy, which means it uses the current action selection to decide on updates. If the policy is suboptimal initially, the updates may reinforce poor choices. For instance, if the agent learns to favor a suboptimal action, it might continue making poor decisions, delaying the overall convergence to the optimal policy.

In environments where rewards are sparse or delayed, the agent may struggle to connect actions with their eventual outcomes. For example, in a maze where rewards are only given at the end, the agent may take many steps without immediate feedback, making it harder to adjust its policy effectively.

.

Date: 30 sept 2024 **Signature of faculty in-charge**

Batch: C3	Roll No.: 16010121110
Experiment / assignment / tutorial No.______	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

TITLE : 7 To implement Q-learning approach

AIM: To understand that Q-learning is a model-free, off-policy reinforcement learning that will find the best course of action.

Expected OUTCOME of Experiment: (Mention CO/CO's attained here): CO3

Books/ Journals/ Websites referred:

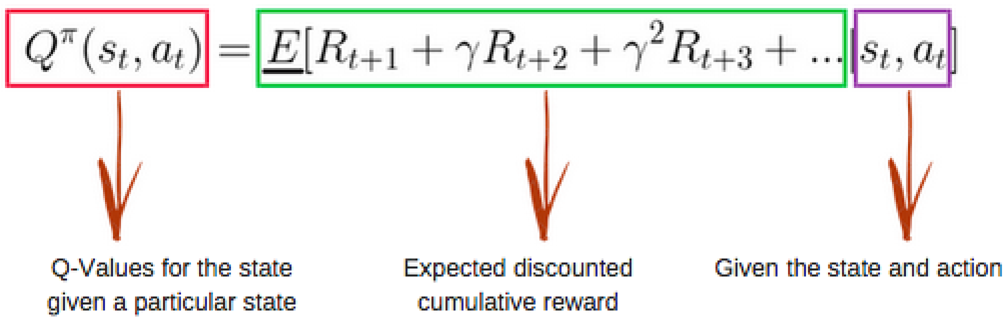
Richard S. Sutton and Andrew G. Barto, "*Reinforcement Learning:An Introduction",* The MIT Press,Second Edition, 2018

Pre Lab/ Prior Concepts:

In Q learning, we consider policy learning of action-values Q(s; a). No importance sampling is required. Next action is chosen using a behaviour policy.

Q-Learning is a basic form of Reinforcement Learning which uses Q-values (also called action values) to iteratively improve the behavior of the learning agent.

The q learning update rule is given as;


$Q^{\pi}(s_t, a_t) = \underline{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | s_t, a_t]$
Q-Values for the state given a particular state
Expected discounted cumulative reward
Given the state and action

In the context of the Frozen Lake environment from OpenAI's Gym, here's an explanation of the key concepts related to Reinforcement Learning (RL):

Problem Statement: Frozen Lake

In Frozen Lake, the agent must navigate across a grid where it faces hazards like slipping on ice and falling into holes. The objective is to move from the starting point (S) to the goal (G) by taking safe steps on frozen lake tiles (F), avoiding holes (H). The environment is stochastic, meaning the agent's movements may not always result in the intended direction (slipping can occur). The challenge is to reach the goal while maximizing the accumulated reward.

1. Policy:

A policy defines the agent's behavior at every state, which is a mapping from states to actions. In the Frozen Lake environment, the policy dictates which direction (left, right, up, or down) the agent should take in each possible position on the grid.

In this case, a deterministic policy might always choose the action with the highest value in each state, while a stochastic policy could assign probabilities to different actions, reflecting the uncertainty of the environment.

2. Reward Function:

The reward function defines how much immediate reward the agent receives for taking a specific action in a given state. In Frozen Lake, the reward function is typically sparse and binary:

- The agent gets a reward of `0` for every step it takes on frozen tiles (F).

- A reward of `1` is given when the agent reaches the goal (G).

- If the agent falls into a hole (H), the episode ends, and no additional reward is given (i.e., reward `0`).

Thus, the reward function is defined as:

- Reaching the goal = `+1`

- Falling into a hole or stepping on a frozen tile = `0`

3. Value Function:

The value function estimates the expected future reward that can be obtained from a given state under a specific policy. There are two main types of value functions:

- State Value Function (V(s)): The expected reward of being in a particular state and following the policy thereafter.

- State-Action Value Function (Q(s,a)): The expected reward of being in a state, taking a specific action, and then following the policy.

In Frozen Lake, the value function helps the agent understand which states are more valuable (closer to the goal) and which actions will maximize the chances of reaching the goal.

4. Model of the Environment:

A model of the environment represents the agent's understanding of how the environment behaves, i.e., how actions affect the state transitions and rewards. There are two components of the model:

- Transition Function (T): Defines the probability of moving from one state to another when taking a particular action. In Frozen Lake, because the environment is stochastic, moving in a direction might not always result in landing on the intended tile. For example, slipping on ice could mean that moving right sometimes results in moving down instead.

- Reward Function (R): Defines the rewards associated with transitions, as discussed earlier.

In model-based RL, the agent uses the model (T and R) to simulate the environment and plan its actions. In Frozen Lake, a model of the environment could capture the probabilities of slipping (moving in an unintended direction) and the consequences of falling into holes.

Conclusion: Thus we have implemented Q learning for the frozen lake problem. We have used Q learning which is a variation of sarsa. It uses only the maximum rewards instead of the complete rewards. It is model free so it does not require knowledge of the environment's transition probabilities or rewards.

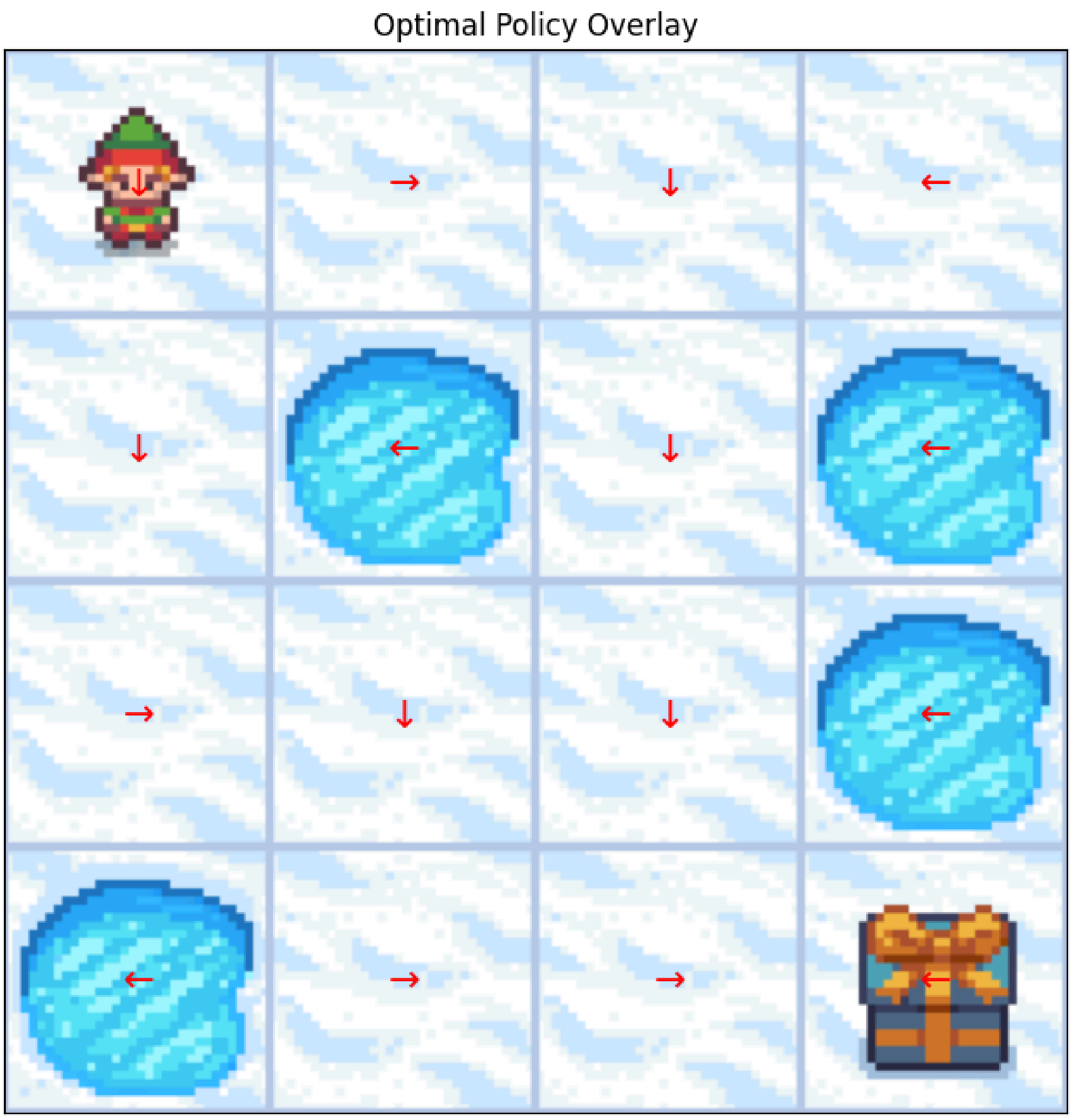
Comparison between DP and Q Learning

Speed -

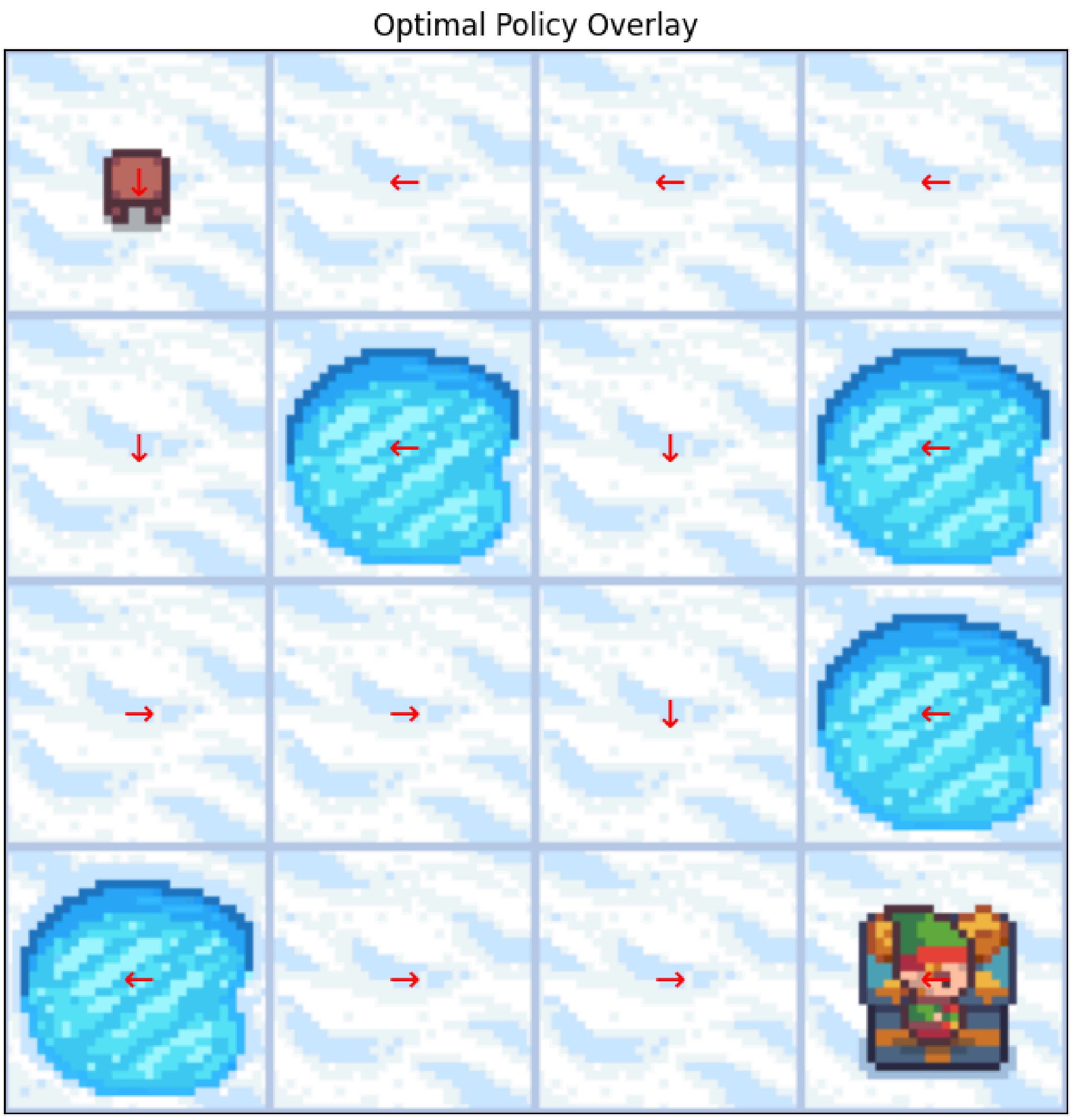
DP converges faster around 500 iterations

Q - Learning converges slower around 20000 episodes

Results - DP



Results - QL



In DP, the first row has better results while for QL the first row shows erroneous results.

Post Lab Descriptive Questions:

Differentiate between Q learning and SARSA.
Q-Learning and SARSA are both reinforcement learning algorithms used to learn optimal policies, but they differ in how they update their action-value (Q) estimates and how they approach learning.

Key Differences:

Learning Approach:

Q-Learning is an off-policy algorithm, meaning it learns the optimal policy independently of the agent's actual actions. It updates its Q-values based on the maximum reward that can be obtained from the next state, assuming the agent takes the best possible action (regardless of what action the agent actually takes).

SARSA is an on-policy algorithm, meaning it updates its Q-values based on the actions the agent actually takes according to its current policy. It follows the agent's own behavior, considering the action the agent is likely to take in the next state based on its policy (e.g., an ϵ-greedy strategy).

Action Selection:

In Q-Learning, the agent looks ahead to the best possible action (the one with the highest Q-value) to make the update, even if that action isn't the one it will follow during the current episode.

In SARSA, the agent updates based on the action it actually chooses in the next state, staying consistent with its exploration strategy (such as random exploration).

Exploration vs. Exploitation:

Q-Learning is more focused on exploitation, as it optimistically assumes that the agent will always take the optimal action in the future, even while it might still be exploring in the current state.

SARSA is more focused on exploration, as it accounts for the exploration the agent does, updating based on the actual action taken, which could be exploratory.

Safety and Risk:

Q-Learning tends to be more aggressive because it assumes the agent will act optimally in the future, which can lead to riskier strategies.

SARSA is generally more conservative because it updates using the actual action taken, including exploratory or sub-optimal actions, leading to safer policies in some cases.

Summary:

Q-Learning learns the optimal policy without following the agent's current behavior (off-policy), while SARSA updates based on the actual actions taken by the agent (on-policy), making Q-Learning more focused on exploitation and SARSA more aligned with exploration.

Date: 7 Sept 24 **Signature of faculty in-charge**

Batch: C2	Roll No.: 16010121110
Experiment / assignment / tutorial No.______	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

TITLE :. 8 To implement an n-step learning approach

AIM: n-step Q learning / n-step SARSA

Expected OUTCOME of Experiment: (Mention CO/CO's attained here)

Books/ Journals/ Websites referred:

Richard S. Sutton and Andrew G. Barto, "*Reinforcement Learning:An Introduction",* The MIT Press,Second Edition, 2018

Pre Lab/ Prior Concepts:

n-step reinforcement learning, in which 'n' is the parameter that determines the number of steps that we want to look ahead before updating the Q-function. So for n=1, this is just "normal" TD learning such as Q-learning or SARSA. When n=2, the algorithm looks one step beyond the immediate reward, it looks two steps beyond, etc.

Both Q-learning and SARSA have an n-step version.

An algorithm for doing n-step learning needs to store the rewards and observed states for steps, as well as keep track of which step to update.

Post Lab Descriptive Questions:

How the book keeping required for n-step algorithms can be handled?

The bookkeeping required for n-step algorithms can be handled by maintaining a buffer or sliding window of the most recent n states, actions, and rewards. As the agent interacts with the environment, each time step's state, action, and reward are stored in this buffer. Once n steps have passed, the algorithm can compute the cumulative return for the n-step update by summing the rewards and adjusting for the future value using a discount factor. The buffer continuously updates as new transitions occur, allowing for efficient tracking of multi-step returns without needing to store the entire episode's history, which keeps the memory usage manageable.

Date: _____________ **Signature of faculty in-charge**

Batch: RL1	Roll No.: 16010121110
Experiment / assignment / tutorial No.______	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

TITLE: 9 To study Dyna-Q model based RL agent

AIM: To understand Dyna-Q is an algorithm intended to speed up learning, or policy convergence, for Q-learning.

Expected OUTCOME of Experiment: (Mentions the CO/CO's attained): CO5

Books/ Journals/ Websites referred:

Richard S. Sutton and Andrew G. Barto, "*Reinforcement Learning:An Introduction",* The MIT Press,Second Edition, 2018

Pre Lab/ Prior Concepts:

A Dyna-Q agent combines acting, learning, and planning. The first two components – acting and learning. Q-learning, for example, learns by acting in the world, and therefore combines acting and learning. But a Dyna-Q agent also implements planning, or simulating experiences from a model–and learns from them.
Dyna-Q agent as implementing acting, learning, and planning simultaneously, at all times. But, in practice, one needs to specify the algorithm as a sequence of steps. The most common way in which the Dyna-Q agent is implemented is by adding a planning routine to a Q-learning agent: after the agent acts in the real world and learns from the observed experience, the agent is allowed a series of k- planning steps. At each one of those k- planning steps, the model generates a simulated experience by randomly sampling from the history of all previously experienced state-action pairs. The agent then learns from this simulated experience, again using the same Q-learning rule.

Problem Context

The Frozen Lake problem is a grid-based RL environment where an agent must navigate a frozen lake to reach a goal without falling into holes. The environment is typically represented as a grid of tiles, where some tiles are safe, others are slippery, and some contain holes. The agent has four actions—up, down, left, and right. The goal is to learn a policy that maximizes the agent's chance of reaching the goal safely.

Concepts in the Context of the Frozen Lake Problem

1. Policy

Policy is a strategy that defines the actions an agent should take from each state. For the Frozen Lake problem, the policy tells the agent the optimal move (left, right, up, or down) from each grid location to reach the goal safely.

Initial Policy: Often, the policy is initially random, with each action having an equal chance of being chosen from each state.

Learned Policy: Over time, as the agent explores the environment and improves its knowledge through Dyna-Q, it updates its policy. The learned policy would ideally guide the agent through the safest and most efficient path to the goal while avoiding holes.

2. Reward Function

The reward function defines the feedback the agent receives after taking an action in a given state. In the Frozen Lake environment:

Goal State: A positive reward (e.g., +1) is given when the agent reaches the goal.

Hole State: A negative reward (e.g., 0 or -1) is given when the agent falls into a hole.

Other States: Typically, all other transitions receive a reward of zero, meaning that only reaching the goal (or falling into a hole) results in a non-zero reward.

This sparse reward setup incentivizes the agent to reach the goal efficiently while avoiding holes.

3. Value Function

The value function estimates the expected cumulative reward from each state, helping the agent determine which states are more desirable. In the Frozen Lake problem, two types of value functions are relevant:

State-Value Function (V): This function represents the expected return (cumulative reward) starting from a specific state and following a policy. It helps the agent understand the "value" of being in each location on the grid.

Action-Value Function (Q): Dyna-Q uses this function to learn the expected return of taking a specific action from a given state. The agent updates this function during training to understand the best action to take from each state, given its knowledge of the environment.

In Dyna-Q, the Q-values are updated both by real experience (exploration of the environment) and simulated experiences from the model, making learning faster and more efficient.

4. Model Environment

The model environment in Dyna-Q is an internal model that the agent builds and updates as it explores the actual environment. It's a simulated environment that allows the agent to imagine the outcomes of actions based on past experiences.

Transition Model: The agent learns which states are likely to follow certain actions. For example, if the agent takes the action "right" from state (2, 2) and reaches state (2, 3), it remembers this transition.

Reward Model: The agent also stores the rewards associated with each transition. This helps simulate reward feedback even when the agent is not directly interacting with the environment.

In the Frozen Lake problem, the model environment becomes particularly useful because it allows the agent to practice navigating toward the goal without falling into holes, even when it's not in the real environment. This capability speeds up learning by enabling "imaginary" experience that reinforces the optimal policy.

Conclusion

The Dyna-Q algorithm enhances the agent's learning in the Frozen Lake problem by combining real-world interactions with simulated experiences in a model environment. This setup allows the agent to learn a robust policy that optimizes for the shortest, safest path to the goal while minimizing the risk of falling into holes. By iteratively updating the Q-values with real and simulated experiences, Dyna-Q provides a practical and efficient way for the agent to adapt its strategy, even in environments with sparse rewards and hazardous states.

Post Lab Descriptive Questions

Write notes on models and planning in RL.

1. Model-Based vs. Model-Free RL

Model-Based RL: This approach uses a model of the environment to simulate outcomes of actions and plan ahead. It is useful for reducing the number of interactions needed with the environment.

Model-Free RL: In this approach, the agent learns policies or value functions directly from interactions with the environment, without explicitly modeling the environment's dynamics.

Example: Model-based algorithms include Dyna-Q and Monte Carlo Tree Search (MCTS), while Q-learning and Policy Gradient methods are popular model-free approaches.

2. Types of Models in RL

Transition Model: Predicts the next state, given the current state and action

Reward Model: Predicts the reward, for a given state-action-state transition.

Full Environment Model: Combines both the transition and reward models to simulate complete future outcomes.

Models can be deterministic (always predicting the same output for given input) or stochastic (accounting for uncertainty, providing distributions of possible outcomes).

3. Planning with Models

Planning involves using a model to predict the outcomes of different actions and sequences of actions in order to make better decisions. Planning is particularly valuable in environments where actions have long-term consequences.

Types of Planning Techniques:

One-Step Planning: Involves updating the value of a state or state-action pair based on a single look-ahead step, similar to model-free methods. It’s computationally inexpensive but limited in foresight.

Multi-Step Planning: Looks further ahead to evaluate sequences of actions. It requires more computation but provides better decision-making by considering potential future rewards over multiple steps.

Common Planning Algorithms:

Dynamic Programming (DP): Uses the Bellman equation to iteratively update value estimates over the entire state space. Requires a full model of the environment but is efficient in tabular settings.

Monte Carlo Tree Search (MCTS): Explores possible action sequences by building a tree of potential future states and rewards. It balances exploration and exploitation, making it popular in large search spaces (e.g., board games).

Dyna-Q: Combines model-based and model-free techniques. The agent learns a model of the environment and uses it to simulate additional experiences. These simulated experiences augment the real experiences to speed up learning.

4. Model Learning

In many cases, the environment model is not known a priori and must be learned from data. Model learning in RL generally involves:

Data Collection: Gathering data through interaction with the environment.

Model Training: Using the collected data to fit a model, which could be a neural network, Gaussian process, or a simpler function approximator.

Evaluating Model Accuracy: Ensuring that the learned model closely matches the environment dynamics. This is essential since errors in the model can lead to poor decision-making.

5. Applications of Models in RL

Models enable agents to simulate and explore future possibilities, which is crucial in scenarios where direct interaction with the environment is costly or impractical, such as:

Robotics: Simulating robot movements to avoid damaging physical hardware.

Healthcare: Planning treatment sequences where incorrect actions could have significant consequences.

Gaming: Using models to simulate outcomes for strategic planning (e.g., AlphaGo uses MCTS).

Date: 27 Oct 24 **Signature of faculty in-charge**

Batch: C2	Roll No.: 16010121110
Experiment / assignment / tutorial No.______	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

TITLE: 10 To prepare summary of Python libraries for Reinforcement Learning

AIM: Listing important libraries in RL

Expected OUTCOME of Experiment: CO1, CO2, CO3, CO4, CO5

Books/ Journals/ Websites referred:

Richard S. Sutton and Andrew G. Barto, "*Reinforcement Learning:An Introduction",* The MIT Press,Second Edition, 2018

Prepare an exhaustive list of useful libraries for implementing RL algorithms.

Python Libraries for Reinforcement Learning (RL):

1. Gym: Developed by OpenAI, Gym is one of the most popular open-source libraries for reinforcement learning. It provides a variety of environments that allow researchers and developers to test and train RL agents on tasks such as playing Atari games, controlling robots, solving classic control problems, and more. Gym simplifies the process of integrating environments with RL algorithms and is highly extensible, supporting custom environment creation.

2. TensorFlow: A powerful open-source machine learning framework, TensorFlow is widely used for both supervised learning and reinforcement learning. It includes various tools for building deep neural networks and has dedicated support for RL applications through libraries like TF-Agents. TensorFlow provides a flexible architecture for large-scale deployments and allows for distributed training of RL models.

3. Keras: A high-level neural networks API, Keras is built on top of TensorFlow and is often used for its simplicity in building and training deep learning models. When combined with RL algorithms, Keras allows for quick prototyping and testing, making it a valuable tool for researchers developing RL models. Libraries like Keras-RL extend its capabilities by providing common RL algorithms such as DQN (Deep Q-Network) and A2C (Advantage Actor Critic).

4. PyTorch: Another leading open-source deep learning framework, PyTorch is favored for its dynamic computational graph, which allows for more flexibility in building and modifying models on the fly. PyTorch is highly popular among RL practitioners due to its ease of use, strong community support, and comprehensive ecosystem, including libraries like TorchRL and Stable Baselines3.

5. OpenAI Baselines: This library provides high-quality implementations of state-of-the-art RL algorithms such as DQN, PPO (Proximal Policy Optimization), A2C, TRPO (Trust Region Policy Optimization), and SAC (Soft Actor-Critic). OpenAI Baselines offers a foundation for experimentation and development, with a focus on reliability and performance in reinforcement learning research.

6. RLlib: Part of the Ray ecosystem, RLlib is a scalable RL library designed to simplify the process of developing, testing, and deploying reinforcement learning algorithms. It supports a wide range of environments and offers several advanced features such as distributed training, hyperparameter tuning, and multi-agent RL. RLlib is known for its ease of integration with large-scale computing clusters.

7. Mujoco: Short for Multi-Joint dynamics with Contact, Mujoco is a physics engine designed for simulating complex dynamic systems, particularly robotic systems. It's widely used in reinforcement learning for simulating environments where agents need to control robots, manipulate objects, or solve physics-based tasks. Mujoco's ability to handle high-fidelity simulations makes it a key tool for RL research in robotics.

8. Stable Baselines3: A popular fork of OpenAI Baselines, Stable Baselines3 offers well-tested implementations of RL algorithms such as PPO, A2C, DQN, and SAC. It focuses on providing easy-to-use APIs and includes helpful features like evaluation, hyperparameter tuning, and custom environment integration. It is suitable for researchers and developers looking for robust RL implementations.

Toolboxes and Frameworks for Reinforcement Learning:

1. DeepMind's DeepQ: This is an open-source implementation of the Deep Q-Learning algorithm, a widely used reinforcement learning method where an agent learns to optimize its behavior by interacting with an environment and maximizing future rewards. Developed by DeepMind, this implementation uses TensorFlow as the backend, providing a well-structured and modular codebase for researchers and developers who want to experiment with Deep Q-learning.

2. HRL (Hierarchical Reinforcement Learning): A Python library focused on hierarchical reinforcement learning, which aims to decompose complex tasks into simpler sub-tasks to improve learning efficiency. This is particularly useful in domains where multi-step decision-making is required, such as robotics, where agents need to perform a sequence of actions in a structured manner.

3. MAML (Model-Agnostic Meta-Learning): MAML is a meta-learning framework that can be used in reinforcement learning to train agents that can quickly adapt to new tasks with minimal training data. It is particularly useful for environments where tasks can change dynamically, and the agent needs to generalize its learning across various tasks. This framework provides Python-based tools for implementing meta-RL algorithms and is suitable for advanced research in RL.

Conclusion: Thus we have noted down various python libraries and their uses. Python libraries like Gym, TensorFlow, Keras, and PyTorch provide essential tools for building and training reinforcement learning (RL) models, with frameworks like Stable Baselines3 and OpenAI Baselines offering tested RL algorithms. RLlib enables scalable RL development, while Mujoco excels at physics simulations for robotics. C++ alternatives like ARMC and RLlib (C++) focus on high-performance RL applications. Other languages like Julia and tools like MATLAB also support RL with specialized libraries. Additionally, resources such as open-source RL implementations and benchmarks like Atari and MuJoCo facilitate standardized evaluation of RL algorithms.

Date: 21 Oct 2024 **Signature of faculty in-charge**