**Reinforcement Learning Lab Manual**

**Needed to install (before executing code)**

* pip install gym
* pip install shimmy>=0.2.1
* pip install tf-agents
* pip install stable-baselines3 gym
* pip install torch
* pip install stable-baselines

**1. Write a python program using Neural Networks for demonstrating Reinforcement Agent, Environment and Reward.**

**Code :-**

import gym

import numpy as np

import torch

import torch.nn as nn

import torch.optim as optim

# Define the environment

env = gym.make('CartPole-v1')

# Neural network for the agent

class PolicyNetwork(nn.Module):

    def \_\_init\_\_(self, input\_size, output\_size):

        super(PolicyNetwork, self).\_\_init\_\_()

        self.fc = nn.Sequential(

            nn.Linear(input\_size, 128),

            nn.ReLU(),

            nn.Linear(128, output\_size),

            nn.Softmax(dim=-1)

        )

    def forward(self, x):

        return self.fc(x)

# Agent class

class Agent:

    def \_\_init\_\_(self, input\_size, output\_size):

        self.policy\_network = PolicyNetwork(input\_size, output\_size)

        self.optimizer = optim.Adam(self.policy\_network.parameters(), lr=0.01)

        self.output\_size = output\_size  # Store output size

    def select\_action(self, state):

        state = torch.from\_numpy(state).float()

        probabilities = self.policy\_network(state)

        probabilities = probabilities.detach().numpy()  # Convert to numpy array

        action = np.random.choice(np.arange(self.output\_size), p=probabilities)

        return action

# Training loop

agent = Agent(input\_size=env.observation\_space.shape[0], output\_size=env.action\_space.n)

num\_episodes = 1000

for episode in range(num\_episodes):

    state = env.reset()

    episode\_reward = 0

    while True:

        action = agent.select\_action(state)

        next\_state, reward, done, \_ = env.step(action)

        agent.optimizer.zero\_grad()

        state\_tensor = torch.from\_numpy(state).float()

        action\_tensor = torch.tensor(action)

        reward\_tensor = torch.tensor(reward)

        log\_prob = torch.log(agent.policy\_network(state\_tensor)[action\_tensor])

        loss = -log\_prob \* reward\_tensor

        loss.backward()

        agent.optimizer.step()

        episode\_reward += reward

        state = next\_state

        if done:

            break

    if episode % 10 == 0:

        print(f"Episode {episode}, Total Reward: {episode\_reward}")

env.close()

**2. Write a python program to demonstrate Markov Decision Process in Reinforcement Learning Environment**

**Code:-**

import numpy as np

# Define the grid world (states)

states = [(0, 0), (0, 1), (0, 2),

          (1, 0), (1, 1), (1, 2),

          (2, 0), (2, 1), (2, 2)]

# Define possible actions (up, down, left, right)

actions = {'U': (-1, 0), 'D': (1, 0), 'L': (0, -1), 'R': (0, 1)}

# Define the state transition function

def transition(state, action):

    if state in states:

        new\_state = (state[0] + action[0], state[1] + action[1])

        if new\_state in states:

            return new\_state

    return state  # Stay in the same state if the action leads to an invalid state

# Define the rewards for each state

rewards = {

    (0, 0): -1,

    (0, 1): -1,

    (0, 2): -1,

    (1, 0): -1,

    (1, 2): -1,

    (2, 0): -1,

    (2, 1): -1,

    (2, 2): 1,  # The goal state with a reward of 1

}

# Define the discount factor

# Define a policy (agent's strategy) - deterministic for simplicity

policy = {

    (0, 0): 'R',  # Move right when in (0, 0)

    (0, 1): 'R',

    (0, 2): 'U',

    (1, 0): 'R',

    (1, 2): 'U',

    (2, 0): 'R',

    (2, 1): 'R',

    (2, 2): 'U',  # Move up when in (2, 2)

}

# Perform value iteration to find the optimal values of each state

V = {state: 0 for state in states}

while True:

    delta = 0

    for state in states:

        if state not in policy:

            continue

        v = V[state]

        action = policy[state]

        next\_state = transition(state, actions[action])

        reward = rewards[state]  # Corrected line

        V[state] = reward + gamma \* V[next\_state]

        delta = max(delta, abs(v - V[state]))

    if delta < 1e-6:

        break

# Print the values of each state

for i in range(3):

    for j in range(3):

        state = (i, j)

        print(f"State {state}: Value = {V[state]:.2f}")

**3. Demonstrate the functions behind state and policies in Reinforcement Learning using Python Program through a 2 X 2 grid.**

**Code:-**

# Define states and actions

states = [(0, 0), (0, 1), (1, 0), (1, 1)]

actions = {'Up': (-1, 0), 'Down': (1, 0), 'Left': (0, -1), 'Right': (0, 1)}

# Define a deterministic policy (for each state, specify the action to take)

policy = {

    (0, 0): 'Right',

    (0, 1): 'Down',

    (1, 0): 'Right',

    (1, 1): 'Up'

}

# Function to get the next state based on the current state and action

def get\_next\_state(state, action):

    next\_state = (state[0] + actions[action][0], state[1] + actions[action][1])

    if next\_state in states:

        return next\_state

    return state

# Function to determine the action the agent takes in a given state

def get\_action(state):

    return policy[state]

# Demonstrate how the functions work

current\_state = (0, 0)

for \_ in range(3):

    action = get\_action(current\_state)

    next\_state = get\_next\_state(current\_state, action)

    print(f"Current State: {current\_state}, Action: {action}, Next State: {next\_state}")

    current\_state = next\_state

**4. Demonstrate Bell-man equation functionality in Reinforcement Learning using Python Programming through 3 X 3 grid Solution:**

**Code:-**

import numpy as np

# Define the grid world

grid\_world = np.zeros((3, 3))

# Define the state transition function (up, down, left, right)

actions = [(0, -1), (0, 1), (-1, 0), (1, 0)]

# Define the reward for each state

rewards = {

    (0, 2): 10,  # Goal state

    (1, 2): -10,  # Penalty state

}

# Define the discount factor

gamma = 0.9

# Perform the Bellman update for state values

num\_iterations = 100

for \_ in range(num\_iterations):

    new\_grid\_world = np.copy(grid\_world)

    for i in range(3):

        for j in range(3):

            if (i, j) not in rewards:

                new\_values = []

                for action in actions:

                    next\_i, next\_j = i + action[0], j + action[1]

                    if 0 <= next\_i < 3 and 0 <= next\_j < 3:

                        new\_values.append(grid\_world[next\_i, next\_j])

                if new\_values:

                    new\_grid\_world[i, j] = max(new\_values) \* gamma

    grid\_world = new\_grid\_world

# Print the final state values

print("State Values:")

print(grid\_world)

**5. Induce a Mouse-pile of cheese strategy to get maximum rewards for the mouse in 3 X 4 grid using Bellman Equation using python programming in a reinforcement Learning environment**

**Code:-**

import numpy as np

# Define the grid world

n\_rows, n\_cols = 3, 4

grid\_world = np.zeros((n\_rows, n\_cols))

# Define rewards

rewards = {

    (0, 3): 10,   # Cheese state

    (1, 3): -10,  # Penalty state

}

# Define discount factor

gamma = 0.9

# Define actions

actions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

action\_names = ['Right', 'Left', 'Down', 'Up']

# Function to calculate the Bellman update for a state

def bellman\_update(i, j, action):

    if (i, j) in rewards:

        return rewards[(i, j)]

    total\_reward = 0

    for a, (di, dj) in enumerate(actions):

        next\_i, next\_j = i + di, j + dj

        if 0 <= next\_i < n\_rows and 0 <= next\_j < n\_cols:

            total\_reward += 0.25 \* (grid\_world[next\_i, next\_j] \* gamma)

    return total\_reward

# Perform the Bellman update for state values

num\_iterations = 100

for \_ in range(num\_iterations):

    new\_grid\_world = np.zeros((n\_rows, n\_cols))

    for i in range(n\_rows):

        for j in range(n\_cols):

            new\_grid\_world[i, j] = max([bellman\_update(i, j, a) for a in actions])

    grid\_world = new\_grid\_world

# Calculate the optimal policy

optimal\_policy = np.empty((n\_rows, n\_cols), dtype=object)

for i in range(n\_rows):

    for j in range(n\_cols):

        if (i, j) not in rewards:

            policy\_values = [bellman\_update(i, j, a) for a in actions]

            if any(policy\_values):

                optimal\_policy[i, j] = action\_names[np.argmax(policy\_values)]

# Print the optimal policy

print("Optimal Policy:")

for row in optimal\_policy:

    print(" | ".join(action if action else ' ' for action in row))

**6. A Fire of value -1 and Maximum Reward of Value 1 placed on the (1,4) and (2,4) place of matrix and you are placed on the initial block of (1,1) on the matrix, through Reinforcement learning Strategy how will obtain the maximum reward using python programming.**

**Code:-**

import numpy as np

# Define the grid world

n\_rows, n\_cols = 2, 5

grid\_world = np.zeros((n\_rows, n\_cols))

# Define rewards

rewards = {

    (1, 4): 1,   # Maximum Reward

    (2, 4): 1,   # Maximum Reward

    (1, 3): -1,  # Fire state

    (2, 3): -1,  # Fire state

}

# Define discount factor

gamma = 0.9

# Define actions (up, down, left, right)

actions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

action\_names = ['Right', 'Left', 'Down', 'Up']

# Function to calculate the Bellman update for a state

def bellman\_update(i, j, action):

    if (i, j) in rewards:

        return rewards[(i, j)]

    total\_reward = 0

    for a, (di, dj) in enumerate(actions):

        next\_i, next\_j = i + di, j + dj

        if 0 <= next\_i < n\_rows and 0 <= next\_j < n\_cols:

            total\_reward += 0.25 \* (grid\_world[next\_i, next\_j] \* gamma)

    return total\_reward

# Perform the Bellman update for state values

num\_iterations = 100

for \_ in range(num\_iterations):

    new\_grid\_world = np.zeros((n\_rows, n\_cols))

    for i in range(n\_rows):

        for j in range(n\_cols):

            new\_grid\_world[i, j] = max([bellman\_update(i, j, a) for a in actions])

    grid\_world = new\_grid\_world

# Calculate the optimal policy

optimal\_policy = np.empty((n\_rows, n\_cols), dtype=object)

for i in range(n\_rows):

    for j in range(n\_cols):

        if (i, j) not in rewards:

            optimal\_policy[i, j] = action\_names[np.argmax([bellman\_update(i, j, a) for a in actions])]

        else:

            optimal\_policy[i, j] = None  # Set None for cells with rewards

# Replace None with a placeholder string

optimal\_policy = np.where(optimal\_policy != None, optimal\_policy.astype(str), 'Reward')

# Print the optimal policy

print("Optimal Policy:")

for row in optimal\_policy:

    print(" | ".join(row))

**7. Display and visualize the difference in Learning of Exploitation and Expectation mechanisms by an agent in a Reinforcement Learning Environment using Python Programming**

**Code:-**

import numpy as np

import matplotlib.pyplot as plt

# Function to simulate the bandit problem with different arms having different reward probabilities

class Bandit:

    def \_\_init\_\_(self, arms):

        self.arms = arms

    def pull\_arm(self, arm):

        return np.random.rand() < self.arms[arm]

# Epsilon-greedy algorithm for exploration and exploitation

def epsilon\_greedy(epsilon, num\_iterations, bandit):

    num\_actions = len(bandit.arms)

    action\_values = np.zeros(num\_actions)

    action\_attempts = np.zeros(num\_actions)

    rewards = []

    for \_ in range(num\_iterations):

        if np.random.rand() < epsilon:

            # Exploration: Choose a random action

            action = np.random.randint(num\_actions)

        else:

            # Exploitation: Choose the action with the highest estimated value

            action = np.argmax(action\_values)

        reward = bandit.pull\_arm(action)

        rewards.append(reward)

        # Update action attempts and estimated action values

        action\_attempts[action] += 1

        action\_values[action] += (reward - action\_values[action]) / action\_attempts[action]

    return rewards

# Define the bandit arms (reward probabilities)

arms = [0.3, 0.5, 0.8]  # Example probabilities

# Create a bandit environment

bandit = Bandit(arms)

# Run epsilon-greedy algorithm with different values of epsilon

epsilon\_values = [0.1, 0.3, 0.5]

num\_iterations = 1000

plt.figure(figsize=(8, 6))

for epsilon in epsilon\_values:

    rewards = epsilon\_greedy(epsilon, num\_iterations, bandit)

    plt.plot(np.cumsum(rewards), label=f'Epsilon={epsilon}')

plt.xlabel('Iterations')

plt.ylabel('Cumulative Reward')

plt.legend()

plt.title('Exploration vs Exploitation in Reinforcement Learning')

plt.show()

**8. Demonstrate the value when exploration mechanism is implemented into the input matrix of 6X4**

**Code:-**

import numpy as np

import random

# Define the grid world

n\_rows, n\_cols = 6, 4

# Define actions (up, down, left, right)

actions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

# Define exploration probability (ε)

epsilon = 0.2

# Initialize the state values

state\_values = np.zeros((n\_rows, n\_cols))

# Function to check if a state is within the grid boundaries

def within\_bounds(state):

    row, col = state

    return 0 <= row < n\_rows and 0 <= col < n\_cols

# Function to choose an action using ε-greedy strategy

def choose\_action(state):

    if random.uniform(0, 1) < epsilon:

        # Exploration: Choose a random action

        return random.choice(range(len(actions)))

    else:

        valid\_actions = []

        for a in actions:

            next\_state = (state[0] + a[0], state[1] + a[1])

            if within\_bounds(next\_state):

                valid\_actions.append(state\_values[next\_state])

            else:

                valid\_actions.append(float('-inf'))  # Assign negative infinity to invalid actions

        return np.argmax(valid\_actions)

# Learning loop (Q-learning with temporal difference)

num\_episodes = 1000

for \_ in range(num\_episodes):

    current\_state = (0, 0)

    while True:

        action = choose\_action(current\_state)

        move = actions[action]

        next\_state = (current\_state[0] + move[0], current\_state[1] + move[1])

        # Simulated reward function (example)

        if next\_state == (5, 3):

            reward = 1

        else:

            reward = 0

        if within\_bounds(next\_state):

            # Update the state value using Q-learning (temporal difference)

            state\_values[current\_state] += 0.1 \* (

                reward + 0.9 \* state\_values[next\_state] - state\_values[current\_state]

            )

            current\_state = next\_state

        else:

            break  # Break the loop if the next state is out of bounds

# Display the state values with exploration

print("State Values with Exploration:")

print(state\_values)

**9. Demonstrate the value when exploitation mechanism is implemented into the input matrix of 6X4**

**Code:-**

import numpy as np

# Define the grid world

n\_rows, n\_cols = 6, 4

# Define actions (up, down, left, right)

actions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

# Initialize the state values

state\_values = np.zeros((n\_rows, n\_cols))

# Simulated reward function (example)

rewards = np.zeros((n\_rows, n\_cols))

rewards[5, 3] = 1  # Maximum Reward

# Discount factor

gamma = 0.9

# Q-Learning: Update state values using exploitation

num\_iterations = 100

for \_ in range(num\_iterations):

    new\_state\_values = np.copy(state\_values)

    for i in range(n\_rows):

        for j in range(n\_cols):

            if rewards[i, j] != 0:

                continue

            q\_values = []

            for action in actions:

                next\_i, next\_j = i + action[0], j + action[1]

                if 0 <= next\_i < n\_rows and 0 <= next\_j < n\_cols:

                    q\_values.append(state\_values[next\_i, next\_j])

            if q\_values:

                new\_state\_values[i, j] = max(q\_values) \* gamma

    state\_values = new\_state\_values

# Display the state values with exploitation

print(state\_values)

**10. Using Tensorflow RL library create an environment, agent and demonstrate Rewards and Punishments within the Reinforcement learning environment.**

**Code:-**

import numpy as np

from tf\_agents.environments import py\_environment

from tf\_agents.specs import array\_spec

from tf\_agents.trajectories import time\_step as ts

class CustomEnvironment(py\_environment.PyEnvironment):

    def \_\_init\_\_(self):

        self.\_action\_spec = array\_spec.BoundedArraySpec(

            shape=(), dtype=np.int32, minimum=0, maximum=1, name='action')

        self.\_observation\_spec = array\_spec.BoundedArraySpec(

            shape=(1,), dtype=np.float32, minimum=0, maximum=1, name='observation')

        self.\_state = np.array([0.5])  # Initial state

    def action\_spec(self):

        return self.\_action\_spec

    def observation\_spec(self):

        return self.\_observation\_spec

    def \_reset(self):

        self.\_state = np.array([0.5])

        return ts.restart(np.array(self.\_state, dtype=np.float32))

    def \_step(self, action):

        if action == 0:  # Move left

            self.\_state -= 0.1

        else:  # Move right

            self.\_state += 0.1

        if self.\_state <= 0:

            reward = -0.5  # Updated punishment for going too far left

            return ts.termination(np.array(self.\_state, dtype=np.float32), reward)

        elif self.\_state >= 1:

            reward = 2.0  # Updated reward for reaching the goal

            return ts.termination(np.array(self.\_state, dtype=np.float32), reward)

        else:

            reward = 5.0  # No reward or punishment for intermediate steps

            return ts.transition(np.array(self.\_state, dtype=np.float32), reward=reward)

environment = CustomEnvironment()

time\_step = environment.reset()

cumulative\_reward = time\_step.reward

for \_ in range(10):

    action = np.random.randint(2)

    time\_step = environment.step(action)

    print(f"Action: {action}, Next State: {time\_step.observation}, Reward: {time\_step.reward}")

    cumulative\_reward += time\_step.reward

print(f"Cumulative Reward: {cumulative\_reward}")

**11. Using Monte carlo method, induce a reinforcement Learning Environment for getting Maximum reward.**

**Code:-**

import numpy as np

import random

class CustomEnv:

    def \_\_init\_\_(self):

        self.num\_states = 10

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

        self.max\_steps = 100

        self.current\_state = 0

        self.reward = 0

        self.total\_reward = 0

        self.steps = 0

    def reset(self):

        self.current\_state = 0

        self.reward = 0

        self.total\_reward = 0

        self.steps = 0

        return self.current\_state

    def step(self, action):

        self.steps += 1

        if action == 'right':

            self.current\_state += 1

            self.reward = 1 if self.current\_state == self.num\_states - 1 else 0

        else:

            self.current\_state -= 1 if self.current\_state > 0 else 0

            self.reward = 0

        self.total\_reward += self.reward

        done = self.current\_state == self.num\_states - 1 or self.steps >= self.max\_steps

        return self.current\_state, self.reward, done, {}

# Monte Carlo Control

def monte\_carlo\_control(env, episodes=1000, gamma=1.0):

    returns\_sum = {}

    returns\_count = {}

    Q = {}

    policy = {}

    for episode in range(episodes):

        states\_actions\_returns = []

        state = env.reset()

        done = False

        # Generate an episode

        while not done:

            action = random.choice(env.actions)

            next\_state, reward, done, \_ = env.step(action)

            states\_actions\_returns.append((state, action, reward))

            state = next\_state

        # Update Q values

        G = 0

        for i, (state, action, reward) in enumerate(reversed(states\_actions\_returns)):

            G = gamma \* G + reward

            if (state, action) not in [(x[0], x[1]) for x in states\_actions\_returns[::-1][i+1:]]:

                if (state, action) in returns\_sum:

                    returns\_sum[(state, action)] += G

                    returns\_count[(state, action)] += 1

                else:

                    returns\_sum[(state, action)] = G

                    returns\_count[(state, action)] = 1

                Q[(state, action)] = returns\_sum[(state, action)] / returns\_count[(state, action)]

        # Update policy based on Q-values

        for s in range(env.num\_states):

            best\_actions = [act for act in env.actions if (s, act) in Q.keys() and Q[(s, act)] == max([Q[(s, a)] for a in env.actions if (s, a) in Q.keys()])]

            policy[s] = random.choice(best\_actions) if best\_actions else random.choice(env.actions)

    return Q, policy

# Create an instance of the environment

env = CustomEnv()

# Monte Carlo Control for learning optimal policy

Q, optimal\_policy = monte\_carlo\_control(env)

# Displaying the learned optimal policy

print("Learned Optimal Policy:")

for state, action in optimal\_policy.items():

    print(f"State: {state}, Action: {action}")

**12. Induce any concept of dynamic programming and explain the efficiency interms of computational complexity for reinforcement Learning environment using Python Programming**

**Code:-**

import numpy as np

# Define the MDP parameters

n\_states = 3  # Number of states

n\_actions = 2  # Number of actions

# Define the MDP transition probabilities and rewards

# Transitions: state -> action -> next state

P = np.zeros((n\_states, n\_actions, n\_states))

P[0, 0, 0] = 0.7

P[0, 0, 1] = 0.3

P[0, 1, 1] = 0.5

P[0, 1, 2] = 0.5

P[1, 0, 0] = 0.4

P[1, 0, 1] = 0.6

P[1, 1, 0] = 0.1

P[1, 1, 1] = 0.9

P[2, 0, 2] = 1.0

P[2, 1, 2] = 1.0

# Rewards: state -> action -> next state

R = np.zeros((n\_states, n\_actions, n\_states))

R[0, 0, 0] = 1.0

R[0, 0, 1] = 2.0

R[0, 1, 1] = 3.0

R[0, 1, 2] = 4.0

R[1, 0, 0] = 0.0

R[1, 0, 1] = 2.0

R[1, 1, 0] = 1.0

R[1, 1, 1] = 3.0

R[2, 0, 2] = 0.0

R[2, 1, 2] = 0.0

# Value Iteration

def value\_iteration(P, R, gamma, epsilon=1e-6):

    n\_states, n\_actions, \_ = P.shape

    V = np.zeros(n\_states)

    while True:

        V\_new = np.zeros(n\_states)

        for s in range(n\_states):

            Q\_s = np.zeros(n\_actions)

            for a in range(n\_actions):

                for s\_prime in range(n\_states):

                    Q\_s[a] += P[s, a, s\_prime] \* (R[s, a, s\_prime] + gamma \* V[s\_prime])

            V\_new[s] = np.max(Q\_s)

        if np.max(np.abs(V - V\_new)) < epsilon:

            break

        V = V\_new.copy()

    return V

# Parameters

gamma = 0.9  # Discount factor

# Perform Value Iteration

optimal\_values = value\_iteration(P, R, gamma)

print("Optimal Values:")

print(optimal\_values)

**13. Consider a news recommendation System has been handled to you, an requirement of making the efficient news to be recommended is taken as the reward, through programming implement how you obtain the maximum reward through TD(0) mechanism.**

**Code:-**

import numpy as np

# Define the number of news articles and user states

n\_articles = 10

n\_states = 5

# Initialize Q-values

Q = np.zeros((n\_states, n\_articles))

# Simulated reward function (example)

rewards = np.random.rand(n\_states, n\_articles)

# Define the TD(0) parameters

alpha = 0.1  # Learning rate

gamma = 0.9  # Discount factor

# Simulate user interactions

num\_episodes = 1000

for \_ in range(num\_episodes):

    state = np.random.randint(n\_states)  # Random initial state

    while True:

        # Exploitation or exploration with epsilon-greedy

        if np.random.rand() < 0.1:

            action = np.random.randint(n\_articles)  # Exploration: Choose a random action

        else:

            action = np.argmax(Q[state, :])  # Exploitation: Select the action with the highest Q-value

        next\_state = np.random.randint(n\_states)  # Simulate user moving to a new state

        reward = rewards[state, action]

        # Update Q-value using TD(0) update rule

        Q[state, action] += alpha \* (reward + gamma \* np.max(Q[next\_state, :]) - Q[state, action])

        state = next\_state

        if np.random.rand() < 0.1:  # Simulate the end of an episode with 10% probability

            break

# Determine the optimal policy

optimal\_policy = np.argmax(Q, axis=1)

# Print the optimal policy

print("Optimal Policy:")

print(optimal\_policy)

**14. Consider you are playing a game of X and O, The System is getting constantly defeated by you. The System decides to enhance SARSA technique to enhance its game play strategies. Explain how the system would plan with the help of python programming.**

**Code:-**

import random

# Define Tic-Tac-Toe environment

class TicTacToe:

    def \_\_init\_\_(self):

        self.board = [' '] \* 9

        self.current\_player = 'X'

        self.winning\_combos = [

            [0, 1, 2], [3, 4, 5], [6, 7, 8],

            [0, 3, 6], [1, 4, 7], [2, 5, 8],

            [0, 4, 8], [2, 4, 6]

        ]

        self.reset()

    def reset(self):

        self.board = [' '] \* 9

        self.current\_player = 'X'

    def is\_winner(self, player):

        for combo in self.winning\_combos:

            if all(self.board[i] == player for i in combo):

                return True

        return False

    def is\_draw(self):

        return ' ' not in self.board

    def is\_game\_over(self):

        return self.is\_winner('X') or self.is\_winner('O') or self.is\_draw()

    def available\_moves(self):

        return [i for i, mark in enumerate(self.board) if mark == ' ']

    def make\_move(self, move):

        self.board[move] = self.current\_player

        self.current\_player = 'O' if self.current\_player == 'X' else 'X'

    def print\_board(self):

        for i in range(0, 9, 3):

            print(' | '.join(self.board[i:i + 3]))

            if i < 6:

                print('-' \* 9)

# SARSA hyperparameters

alpha = 0.1  # Learning rate

gamma = 0.9  # Discount factor

epsilon = 0.1  # Epsilon for epsilon-greedy policy

# Initialize Q-table

Q = {}

# SARSA algorithm

def update\_Q\_value(state, action, reward, next\_state, next\_action):

    if (state, action) not in Q:

        Q[(state, action)] = 0  # Initialize Q-value for unseen state-action pair

    if (next\_state, next\_action) not in Q:

        Q[(next\_state, next\_action)] = 0

    Q[state, action] += alpha \* (reward + gamma \* Q[next\_state, next\_action] - Q[state, action])

# Helper function for epsilon-greedy policy

def get\_action(state):

    available\_moves = env.available\_moves()

    if not available\_moves:

        return None  # No available moves

    if random.uniform(0, 1) < epsilon:

        return random.choice(available\_moves)

    else:

        return max(available\_moves, key=lambda x: Q.get((state, x), 0))

# Training loop

num\_episodes = 10000

env = TicTacToe()

for episode in range(num\_episodes):

    env.reset()

    state = tuple(env.board)

    action = get\_action(state)

    while not env.is\_game\_over():

        if action is None:

            break  # No available moves

        env.make\_move(action)

        next\_state = tuple(env.board)

        if env.is\_winner('X'):

            reward = 1

        elif env.is\_winner('O'):

            reward = -1

        else:

            reward = 0

        next\_action = get\_action(next\_state)

        update\_Q\_value(state, action, reward, next\_state, next\_action)

        state = next\_state

        action = next\_action

        if action is None:

            break  # No available moves

# After training, the system uses learned Q-values to play against the player

def play\_game():

    env.reset()

    state = tuple(env.board)

    while not env.is\_game\_over():

        if env.current\_player == 'X':

            env.print\_board()

            move = int(input("Enter your move (0-8): "))

            while move not in env.available\_moves():

                move = int(input("Invalid move. Enter your move (0-8): "))

            env.make\_move(move)

        else:

            action = get\_action(state)

            if action is None:

                print("System can't make a move. It's a draw!")

                break

            env.make\_move(action)

        state = tuple(env.board)

    env.print\_board()

    if env.is\_winner('X'):

        print("You win!")

    elif env.is\_winner('O'):

        print("System wins!")

    else:

        print("It's a draw!")

# Play the game against the system

play\_game()

**15. A Mario game is played by Agent, the agent keeps on moving over, The System decides to tough the levels, how can a system induce Q-Learning technique to enhance its game play strategies the compiler used for the game is python.**

**Code:-**

import numpy as np

# Define the environment (2D grid world)

class MarioEnvironment:

    def \_\_init\_\_(self):

        self.grid\_size = (5, 5)  # Grid size

        self.agent\_position = (0, 0)  # Starting position of the agent

        self.goal\_position = (4, 4)  # Goal position

        self.obstacle\_positions = [(1, 1), (2, 2), (3, 3)]  # Obstacle positions

        self.actions = ['UP', 'DOWN', 'LEFT', 'RIGHT']  # Possible actions

        self.num\_actions = len(self.actions)

        self.Q\_values = np.zeros((self.grid\_size[0], self.grid\_size[1], self.num\_actions))  # Q-values initialized to 0

        self.alpha = 0.1  # Learning rate

        self.gamma = 0.9  # Discount factor

        self.epsilon = 0.1  # Epsilon for epsilon-greedy policy

    def is\_valid\_position(self, position):

        x, y = position

        return 0 <= x < self.grid\_size[0] and 0 <= y < self.grid\_size[1]

    def get\_reward(self, position):

        if position == self.goal\_position:

            return 10  # Reward for reaching the goal

        elif position in self.obstacle\_positions:

            return -5  # Penalty for hitting an obstacle

        else:

            return -1  # Small negative reward for each step

    def update\_agent\_position(self, action):

        if action == 'UP':

            new\_position = (self.agent\_position[0] - 1, self.agent\_position[1])

        elif action == 'DOWN':

            new\_position = (self.agent\_position[0] + 1, self.agent\_position[1])

        elif action == 'LEFT':

            new\_position = (self.agent\_position[0], self.agent\_position[1] - 1)

        elif action == 'RIGHT':

            new\_position = (self.agent\_position[0], self.agent\_position[1] + 1)

        else:

            return

        if self.is\_valid\_position(new\_position):

            self.agent\_position = new\_position

    def q\_learning(self, num\_episodes):

        for episode in range(num\_episodes):

            current\_state = self.agent\_position

            total\_reward = 0  # Track total reward for each episode

            while current\_state != self.goal\_position:

                # Choose action using epsilon-greedy policy

                if np.random.rand() < self.epsilon:

                    action = np.random.choice(self.actions)

                else:

                    action = self.actions[np.argmax(self.Q\_values[current\_state[0], current\_state[1]])]

                # Take action and observe the next state and reward

                self.update\_agent\_position(action)

                new\_state = self.agent\_position

                reward = self.get\_reward(new\_state)

                total\_reward += reward  # Accumulate reward for the episode

                # Q-value update using the Q-learning equation

                self.Q\_values[current\_state[0], current\_state[1], self.actions.index(action)] += \

                    self.alpha \* (reward + self.gamma \* np.max(self.Q\_values[new\_state[0], new\_state[1]]) -

                                  self.Q\_values[current\_state[0], current\_state[1], self.actions.index(action)])

                current\_state = new\_state

            print(f"Episode {episode + 1}, Total Reward: {total\_reward}")

            # Reset agent position for next episode

            self.agent\_position = (0, 0)

# Create Mario environment

env = MarioEnvironment()

# Train Mario using Q-learning

env.q\_learning(num\_episodes=1000)

**16. You are given a 3D realistic environment in a traveller game. With a help of python intrepreter and inducing temporal difference strategies, explain how you optimize the selection strategy and attain maximal travel explain with the help of a python program**

**Code:-**

import numpy as np

import random

# Function to simulate taking an action in the environment (grid-like environment)

def take\_action(state, action):

    if action == 0:  # Move left

        return max(0, state - 1), -1  # Moving left decrements state and incurs -1 reward

    elif action == 1:  # Move right

        return min(num\_states - 1, state + 1), -1  # Moving right increments state and incurs -1 reward

    elif action == 2:  # Move up

        return max(0, state - 5), -1  # Moving up decrements state by 5 and incurs -1 reward

    elif action == 3:  # Move down

        return min(num\_states - 1, state + 5), -1  # Moving down increments state by 5 and incurs -1 reward

# Initialize Q-values

num\_states = 21  # Placeholder for the number of states in your environment

num\_actions = 4  # Placeholder for the number of possible actions

initial\_state = 0  # Placeholder for the initial state

destination\_state = 20  # Placeholder for the destination state

Q = np.zeros((num\_states, num\_actions))

# Define Q-Learning parameters

epsilon = 0.1

alpha = 0.1

gamma = 0.9

num\_episodes = 100  # Reducing the number of episodes

# Q-Learning training

for \_ in range(num\_episodes):

    state = initial\_state

    reached\_destination = False

    while not reached\_destination:

        if random.uniform(0, 1) < epsilon:

            action = random.choice(range(num\_actions))

        else:

            action = np.argmax(Q[state, :])

        next\_state, reward = take\_action(state, action)

        # Q-value update

        Q[state, action] = (1 - alpha) \* Q[state, action] + alpha \* (reward + gamma \* np.max(Q[next\_state, :]))

        state = next\_state

        if state == destination\_state:  # Check if the destination is reached

            reached\_destination = True

# Path selection using Q-values

state = initial\_state

optimal\_path = [state]

while state != destination\_state:

    action = np.argmax(Q[state, :])

    next\_state, \_ = take\_action(state, action)

    state = next\_state

    optimal\_path.append(state)

print("Optimal Path:", optimal\_path)

**17. Consider three trains running on respective tracks. Each of the train is based on various algorithms of temporal difference learning say, Train A is induced with TD(0) algorithm, Train B is powered by SARSA algorithm and Train C with Q- Learning. On using a python script reveal which train outperforms the other interms of efficiency by attaining maximal reward at less number of computational steps.**

**Code:-**

**18. Consider you are playing a game of Tic-Tac-Toe, The System is getting constantly defeated by you. The System decides to enhance its reward maximization technique to enhance its game play strategies. Explain how the system would plan and which Temporal difference strategy it will choose with the help of python programming.**

**Code:-**

import numpy as np

import random

# Define the Tic-Tac-Toe environment

class TicTacToe:

    def \_\_init\_\_(self):

        self.board = [' '] \* 9

        self.current\_player = 'X'

        self.winner = None

    def reset(self):

        self.board = [' '] \* 9

        self.current\_player = 'X'

        self.winner = None

    def make\_move(self, action):

        if self.board[action] == ' ' and not self.winner:

            self.board[action] = self.current\_player

            self.check\_winner()

            self.switch\_player()

    def switch\_player(self):

        self.current\_player = 'X' if self.current\_player == 'O' else 'O'

    def check\_winner(self):

        winning\_combinations = [

            (0, 1, 2), (3, 4, 5), (6, 7, 8),

            (0, 3, 6), (1, 4, 7), (2, 5, 8),

            (0, 4, 8), (2, 4, 6)

        ]

        for a, b, c in winning\_combinations:

            if self.board[a] == self.board[b] == self.board[c] != ' ':

                self.winner = self.board[a]

    def is\_game\_over(self):

        return ' ' not in self.board or self.winner

    def get\_state(self):

        return tuple(self.board)

# Q-Learning agent

class QLearningAgent:

    def \_\_init\_\_(self, epsilon=0.1, alpha=0.1, gamma=0.9):

        self.epsilon = epsilon

        self.alpha = alpha

        self.gamma = gamma

        self.q\_table = {}

        self.prev\_state = None

        self.prev\_action = None

    def choose\_action(self, state):

        if random.uniform(0, 1) < self.epsilon:

            available\_actions = [i for i, s in enumerate(state) if s == ' ']

            return random.choice(available\_actions) if available\_actions else None

        else:

            if state in self.q\_table:

                available\_actions = [i for i, s in enumerate(state) if s == ' ']

                if available\_actions:

                    return max([(i, self.q\_table[state][i]) for i in available\_actions], key=lambda x: x[1])[0]

                else:

                    return None

            else:

                return None

    def update\_q\_table(self, state, action, reward, next\_state, next\_action):

        if state not in self.q\_table:

            self.q\_table[state] = [0.0] \* 9

        if next\_state not in self.q\_table:

            self.q\_table[next\_state] = [0.0] \* 9

        if self.prev\_state is not None:

            self.q\_table[self.prev\_state][self.prev\_action] += self.alpha \* (

                reward + self.gamma \* self.q\_table[state][action] - self.q\_table[self.prev\_state][self.prev\_action]

            )

        self.prev\_state = state

        self.prev\_action = action

    def reset(self):

        self.prev\_state = None

        self.prev\_action = None

# Training the Q-Learning agent

def train\_q\_learning\_agent(agent, env, episodes):

    for episode in range(episodes):

        state = env.get\_state()

        agent.reset()

        while not env.is\_game\_over():

            action = agent.choose\_action(state)

            if action is None:

                break

            env.make\_move(action)

            next\_state = env.get\_state()

            if env.winner == 'X':

                reward = 1

            elif env.winner == 'O':

                reward = -1

            else:

                reward = 0

            next\_action = agent.choose\_action(next\_state)

            agent.update\_q\_table(state, action, reward, next\_state, next\_action)

            state = next\_state

        env.reset()

# Play against the trained agent

def play\_vs\_agent(agent, env):

    while not env.is\_game\_over():

        env.make\_move(agent.choose\_action(env.get\_state()))

        print\_board(env.board)

        if env.winner:

            print(f'Winner: {env.winner}')

            break

        player\_action = int(input('Enter your move (0-8): '))

        env.make\_move(player\_action)

        print\_board(env.board)

# Helper function to display the board

def print\_board(board):

    print(board[0], '|', board[1], '|', board[2])

    print('--+---+--')

    print(board[3], '|', board[4], '|', board[5])

    print('--+---+--')

    print(board[6], '|', board[7], '|', board[8])

if \_\_name\_\_ == '\_\_main\_\_':

    agent = QLearningAgent()

    env = TicTacToe()

    # Train the Q-Learning agent

    train\_q\_learning\_agent(agent, env, episodes=10000)

    # Play against the trained agent

    print("You are playing against the trained agent (X)")

    while True:

        play\_vs\_agent(agent, env)

        play\_again = input("Play again? (yes/no): ").strip().lower()

        if play\_again != "yes":

            break

7

**19. Demonstrate the need for Deep - Q- Learning as your autonomous vehicle's detecting efficiency is declining with a help of a program**

**Code:-**

import numpy as np

# Define the chess environment (simplified board)

class ChessEnvironment:

    def \_\_init\_\_(self):

        self.board = np.array([

            ['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R'],

            ['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P'],

            [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],

            [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],

            [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],

            [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],

            ['p', 'p', 'p', 'p', 'p', 'p', 'p', 'p'],

            ['r', 'n', 'b', 'q', 'k', 'b', 'n', 'r']

        ])

        self.current\_player = 'white'

    def is\_checkmate(self, player):

        # A simplified checkmate condition

        king = 'K' if player == 'white' else 'k'

        return np.all(self.board != king)

    def make\_move(self, move):

        # A simplified function to make a move

        # It doesn't enforce the rules of chess

        if move is not None:

            row1, col1, row2, col2 = move

            self.board[row2, col2] = self.board[row1, col1]

            self.board[row1, col1] = ' '

            self.current\_player = 'white' if self.current\_player == 'black' else 'black'

# Deep Q-Learning agent

class DQLAgent:

    def \_\_init\_\_(self):

        # Define and train a deep neural network for Q-Learning

        pass

    def choose\_move(self, state):

        # Placeholder implementation to return a random move

        # Replace this logic with the actual DQL agent decision-making process

        if np.random.rand() < 0.5:

            return (0, 0, 1, 0)  # Example: random move (row1, col1, row2, col2)

        else:

            return None  # Return None if no move is chosen

# Initialize the chess environment and DQL agent

chess\_env = ChessEnvironment()

dql\_agent = DQLAgent()

# Training loop (for a simplified example, we're not performing real training)

for episode in range(10):

    while not chess\_env.is\_checkmate(chess\_env.current\_player):

        state = chess\_env.board

        move = dql\_agent.choose\_move(state)

        chess\_env.make\_move(move)

**20. In a game of chess, your opponent wants to carry over mate as soon as possible but you enhance the way of deep - Q- learning method of handling the game, explain why and how will you win through a Python Program?**

**Code:-**

import numpy as np

# Define the chess environment (simplified board)

class ChessEnvironment:

    def \_\_init\_\_(self):

        self.board = np.array([

            ['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R'],

            ['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P'],

            [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],

            [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],

            [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],

            [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],

            ['p', 'p', 'p', 'p', 'p', 'p', 'p', 'p'],

            ['r', 'n', 'b', 'q', 'k', 'b', 'n', 'r']

        ])

        self.current\_player = 'white'

    def is\_checkmate(self, player):

        king = 'K' if player == 'white' else 'k'

        return np.all(self.board != king)

    def make\_move(self, move):

        row1, col1, row2, col2 = move

        self.board[row2, col2] = self.board[row1, col1]

        self.board[row1, col1] = ' '

        self.current\_player = 'white' if self.current\_player == 'black' else 'black'

# Deep Q-Learning agent (random move for illustration)

class DQLAgent:

    def \_\_init\_\_(self):

        # Initialize any necessary variables for the agent

        pass

    def choose\_move(self, state):

        # For illustration, choose a random move

        possible\_moves = [(r1, c1, r2, c2) for r1 in range(8) for c1 in range(8)

                          for r2 in range(8) for c2 in range(8)]

        return possible\_moves[np.random.choice(len(possible\_moves))]

# Initialize the chess environment and DQL agent

chess\_env = ChessEnvironment()

dql\_agent = DQLAgent()

# Training loop (simplified random moves)

for episode in range(10):

    while not chess\_env.is\_checkmate(chess\_env.current\_player):

        state = chess\_env.board

        move = dql\_agent.choose\_move(state)

        print(f"[Move {episode + 1}]")

        print(f"[Current Player: {chess\_env.current\_player}]")

        print(chess\_env.board)

        print()

        chess\_env.make\_move(move)

**21. Consider you are the manager of a Finance Company, the target of the month has not been achieved and you are in trouble. You come to know that your Q-Learning System not performing well as the numbers of customers have increased, the correction decision would be increasing the layers of the DQN. So explain how you will enhance DQN and transform it into DDQN.**

**Code:-**

import numpy as np

import tensorflow as tf

from tensorflow import keras

from collections import deque

import random

import gym

# Define a simple replay buffer

class ReplayBuffer:

    def \_\_init\_\_(self, max\_size):

        self.buffer = deque(maxlen=max\_size)

    def add(self, experience):

        self.buffer.append(experience)

    def sample(self, batch\_size):

        batch = random.sample(self.buffer, batch\_size)

        states, actions, rewards, next\_states, dones = zip(\*batch)

        return np.array(states), np.array(actions), np.array(rewards), np.array(next\_states), np.array(dones)

# Define the Double Deep Q-Network (DDQN) agent

class DDQNAgent:

    def \_\_init\_\_(self, state\_size, action\_size):

        self.state\_size = state\_size

        self.action\_size = action\_size

        self.target\_update\_frequency = 100  # Update the target network every n steps

        # DQN and target DQN

        self.dqn = self.build\_dqn\_model()

        self.target\_dqn = self.build\_dqn\_model()

        self.target\_dqn.set\_weights(self.dqn.get\_weights())

        self.replay\_buffer = ReplayBuffer(max\_size=2000)

        self.batch\_size = 32

        self.gamma = 0.99  # Discount factor

        # Exploration parameters

        self.epsilon = 1.0  # Exploration rate

        self.min\_epsilon = 0.01  # Minimum exploration rate

        self.epsilon\_decay = 0.995  # Decay rate

        self.total\_steps = 0

    def build\_dqn\_model(self):

        model = keras.Sequential([

            keras.layers.Dense(24, activation='relu', input\_shape=(self.state\_size,)),

            keras.layers.Dense(24, activation='relu'),

            keras.layers.Dense(self.action\_size, activation='linear')

        ])

        model.compile(optimizer=keras.optimizers.Adam(learning\_rate=0.001), loss='mse')

        return model

    def select\_action(self, state):

        if np.random.rand() <= self.epsilon:

            return random.randrange(self.action\_size)

        q\_values = self.dqn.predict(np.expand\_dims(state, axis=0))

        return np.argmax(q\_values[0])

    def train(self):

        if len(self.replay\_buffer.buffer) < self.batch\_size:

            return

        states, actions, rewards, next\_states, dones = self.replay\_buffer.sample(self.batch\_size)

        targets = self.dqn.predict(states)

        target\_values = self.target\_dqn.predict(next\_states)

        for i in range(self.batch\_size):

            if dones[i]:

                targets[i][actions[i]] = rewards[i]

            else:

                best\_action = np.argmax(self.dqn.predict(np.expand\_dims(next\_states[i], axis=0))[0])

                targets[i][actions[i]] = rewards[i] + self.gamma \* target\_values[i][best\_action]

        self.dqn.fit(states, targets, epochs=1, verbose=0)

        if self.epsilon > self.min\_epsilon:

            self.epsilon \*= self.epsilon\_decay

        self.total\_steps += 1

        if self.total\_steps % self.target\_update\_frequency == 0:

            self.target\_dqn.set\_weights(self.dqn.get\_weights())

    def remember(self, state, action, reward, next\_state, done):

        self.replay\_buffer.add((state, action, reward, next\_state, done))

    def load(self, name):

        self.dqn.load\_weights(name)

    def save(self, name):

        self.dqn.save\_weights(name)

# Training the DDQN agent on a simple OpenAI Gym environment

def train\_ddqn\_agent():

    env = gym.make("CartPole-v1")

    state\_size = env.observation\_space.shape[0]

    action\_size = env.action\_space.n

    agent = DDQNAgent(state\_size, action\_size)

    episodes = 100

    for episode in range(episodes):

        state = env.reset()

        done = False

        while not done:

            action = agent.select\_action(state)

            next\_state, reward, done, \_ = env.step(action)

            agent.remember(state, action, reward, next\_state, done)

            state = next\_state

            agent.train()

        if episode % 10 == 0:

            print(f"Episode: {episode}/{episodes}, Total Steps: {agent.total\_steps}, Epsilon: {agent.epsilon:.2}")

    agent.save("ddqn\_model.h5")

if \_\_name\_\_ == "\_\_main\_\_":

    train\_ddqn\_agent()

**22. You are driving a bus in Simulation environment, a discrepancy of less quality policies are returning you a low value points in your simulation Quality which makes them to choose low optimal strategies, based on the necessity you decide to choose DDPG for inducing optimality. Prove it through Coding.**

**Code:-**

import tensorflow as tf

import numpy as np

import gym

from collections import deque

import random

# Define the Actor and Critic neural networks

class Actor(tf.keras.Model):

    def \_\_init\_\_(self, action\_dim, max\_action):

        super(Actor, self).\_\_init\_\_()

        self.dense1 = tf.keras.layers.Dense(400, activation='relu')

        self.dense2 = tf.keras.layers.Dense(300, activation='relu')

        self.output\_layer = tf.keras.layers.Dense(action\_dim, activation='tanh')

        self.max\_action = max\_action

    def call(self, state):

        x = self.dense1(state)

        x = self.dense2(x)

        actions = self.output\_layer(x)

        return actions \* self.max\_action

class Critic(tf.keras.Model):

    def \_\_init\_\_(self):

        super(Critic, self).\_\_init\_\_()

        self.dense1 = tf.keras.layers.Dense(400, activation='relu')

        self.dense2 = tf.keras.layers.Dense(300, activation='relu')

        self.output\_layer = tf.keras.layers.Dense(1)

    def call(self, state, action):

        x = self.dense1(tf.concat([state, action], axis=-1))

        x = self.dense2(x)

        q\_value = self.output\_layer(x)

        return q\_value

# Define the DDPG agent

class DDPGAgent:

    def \_\_init\_\_(self, state\_dim, action\_dim, max\_action):

        self.actor = Actor(action\_dim, max\_action)

        self.target\_actor = Actor(action\_dim, max\_action)

        self.actor\_optimizer = tf.keras.optimizers.Adam(0.001)

        self.critic = Critic()

        self.target\_critic = Critic()

        self.critic\_optimizer = tf.keras.optimizers.Adam(0.002)

        self.memory = deque(maxlen=100000)

        self.batch\_size = 64

        self.discount = 0.99

        self.tau = 0.001

    def select\_action(self, state):

        return self.actor(np.expand\_dims(state, axis=0))

    def train(self):

        if len(self.memory) < self.batch\_size:

            return 0, 0

        # Sample a random mini-batch from the replay buffer

        batch = random.sample(self.memory, self.batch\_size)

        state\_batch, action\_batch, reward\_batch, next\_state\_batch, done\_batch = map(np.array, zip(\*batch))

        target\_actions = self.target\_actor(next\_state\_batch)

        target\_q\_values = self.target\_critic(next\_state\_batch, target\_actions)

        target\_q\_values = reward\_batch + self.discount \* target\_q\_values \* (1 - done\_batch)

        with tf.GradientTape() as tape:

            q\_values = self.critic(state\_batch, action\_batch)

            critic\_loss = tf.losses.mean\_squared\_error(target\_q\_values, q\_values)

        critic\_grads = tape.gradient(critic\_loss, self.critic.trainable\_variables)

        self.critic\_optimizer.apply\_gradients(zip(critic\_grads, self.critic.trainable\_variables))

        with tf.GradientTape() as tape:

            actions = self.actor(state\_batch)

            actor\_loss = -tf.reduce\_mean(self.critic(state\_batch, actions))

        actor\_grads = tape.gradient(actor\_loss, self.actor.trainable\_variables)

        self.actor\_optimizer.apply\_gradients(zip(actor\_grads, self.actor.trainable\_variables))

        for target, source in zip(self.target\_critic.trainable\_variables, self.critic.trainable\_variables):

            target.assign(self.tau \* source + (1 - self.tau) \* target)

        for target, source in zip(self.target\_actor.trainable\_variables, self.actor.trainable\_variables):

            target.assign(self.tau \* source + (1 - self.tau) \* target)

        return actor\_loss, critic\_loss

    def remember(self, state, action, reward, next\_state, done):

        self.memory.append((state, action, reward, next\_state, done))

# Main training loop

def train\_ddpg\_agent():

    env = gym.make("Pendulum-v1")

    state\_dim = env.observation\_space.shape[0]

    action\_dim = env.action\_space.shape[0]

    max\_action = env.action\_space.high[0]

    agent = DDPGAgent(state\_dim, action\_dim, max\_action)

    num\_episodes = 200

    for episode in range(num\_episodes):

        state = env.reset()

        total\_reward = 0

        actor\_loss, critic\_loss = 0, 0

        done = False

        while not done:

            action = agent.select\_action(state)

            action\_array = np.squeeze(action, axis=0)  # Convert action tensor to numpy array

            next\_state, reward, done, \_ = env.step(action\_array)

            agent.remember(state, action\_array, reward, next\_state, done)

            actor\_loss, critic\_loss = agent.train()

            total\_reward += reward

            state = next\_state

        print(f"Episode: {episode + 1}, Total Reward: {total\_reward}, Actor Loss: {actor\_loss}, Critic Loss: {critic\_loss}")

if \_\_name\_\_ == "\_\_main\_\_":

    train\_ddpg\_agent()

**23. You run Google Maps, a discrepancy of less quality policies are returning to customers which make them to choose low optimal strategies, CEO advises you to choose PPO for inducing optimality. Prove it through Coding.**

**Code:-**

**24. You are instructed by your mentor to build a stop clock which will be running asynchronously showing variation of different timing around the world. Now, you need to find which of two methods will be suitable for the development whether A2C or A3C give the Optimal policy designing framework.**

**Code:-**

import numpy as np

import tensorflow as tf

from tensorflow.keras.layers import Dense, Input

class A2CAgent:

    def \_\_init\_\_(self, state\_dim, action\_dim):

        self.state\_dim = state\_dim

        self.action\_dim = action\_dim

        self.gamma = 0.99  # Discount factor for future rewards

        self.actor\_critic = self.build\_actor\_critic()

        self.actor\_optimizer = tf.keras.optimizers.Adam()

    def build\_actor\_critic(self):

        input\_state = Input(shape=(self.state\_dim,))

        dense1 = Dense(32, activation='relu')(input\_state)

        dense2 = Dense(32, activation='relu')(dense1)

        action\_head = Dense(self.action\_dim, activation='softmax')(dense2)

        critic\_head = Dense(1)(dense2)

        model = tf.keras.Model(inputs=input\_state, outputs=[action\_head, critic\_head])

        return model

    def select\_action(self, state):

        action\_probs, \_ = self.actor\_critic.predict(state)

        action = np.random.choice(len(action\_probs[0]), p=action\_probs[0])

        return action

    def train(self, states, actions, rewards, next\_states, dones):

        with tf.GradientTape() as tape:

            action\_probs, values = self.actor\_critic(states)

            action\_masks = tf.one\_hot(actions, len(action\_probs[0]))

            selected\_action\_probs = tf.reduce\_sum(action\_probs \* action\_masks, axis=1)

            advantages = self.compute\_advantages(rewards, values, dones)

            actor\_loss = -tf.reduce\_sum(tf.math.log(selected\_action\_probs) \* advantages)

            critic\_loss = tf.reduce\_sum(tf.square(rewards - values))

            total\_loss = actor\_loss + critic\_loss

        actor\_gradients = tape.gradient(total\_loss, self.actor\_critic.trainable\_variables)

        self.actor\_optimizer.apply\_gradients(zip(actor\_gradients, self.actor\_critic.trainable\_variables))

    def compute\_advantages(self, rewards, values, dones):

        advantages = np.zeros\_like(rewards, dtype=np.float32)

        last\_advantage = 0

        for t in reversed(range(len(rewards) - 1)):

            mask = 1.0 - dones[t]

            delta = rewards[t] + self.gamma \* values[t + 1] \* mask - values[t]

            advantages[t] = delta + self.gamma \* last\_advantage \* mask

            last\_advantage = advantages[t]

        return advantages

class StopwatchEnv:

    def \_\_init\_\_(self):

        self.time\_elapsed = 0

    def reset(self):

        self.time\_elapsed = 0

        return [self.time\_elapsed]

    def step(self, action):

        self.time\_elapsed += action

        done = False

        if self.time\_elapsed >= 60:

            self.time\_elapsed = 0

            done = True

        return [self.time\_elapsed], 1, done

def train\_a2c\_agent(agent, env, num\_episodes=1000):

    for episode in range(num\_episodes):

        state = env.reset()

        total\_reward = 0

        done = False

        while not done:

            action = agent.select\_action(np.array([state]))

            next\_state, reward, done = env.step(action)

            agent.train(np.array([state]),

                        np.array([action]),

                        np.array([reward]),

                        np.array([next\_state]),

                        np.array([done]))

            state = next\_state

            total\_reward += reward

        print(f'Episode {episode + 1}/{num\_episodes} finished. Total reward: {total\_reward}')

def main():

    env = StopwatchEnv()

    agent = A2CAgent(state\_dim=1, action\_dim=60)

    train\_a2c\_agent(agent, env)

if \_\_name\_\_ == '\_\_main\_\_':

    main()

**25. You are a Stock Market advisor, now there is a need to develop a learning engine which will advice you get maximum Profit investment through Probabilistic values of the historical data processing, Use Vanilla Policy Gradient for structuring the highest return Policies.**

**Code:-**

import numpy as np

import tensorflow as tf

import gym

# Define the Vanilla Policy Gradient Agent

class VPGAgent:

    def \_\_init\_\_(self, state\_dim, action\_dim, learning\_rate=0.02):

        self.policy\_network = self.build\_policy\_network(state\_dim, action\_dim)

        self.optimizer = tf.keras.optimizers.Adam(learning\_rate)

    def build\_policy\_network(self, state\_dim, action\_dim):

        model = tf.keras.Sequential([

            tf.keras.layers.Dense(32, activation='relu', input\_shape=(state\_dim,)),

            tf.keras.layers.Dense(16, activation='relu'),

            tf.keras.layers.Dense(action\_dim, activation='softmax')

        ])

        return model

    def select\_action(self, state):

        action\_probs = self.policy\_network.predict(np.array([state]))

        action = np.random.choice(len(action\_probs[0]), p=action\_probs[0])

        return action

    def train(self, states, actions, advantages):

        with tf.GradientTape() as tape:

            action\_probs = self.policy\_network(np.array(states))

            action\_masks = tf.one\_hot(actions, len(action\_probs[0]))

            selected\_action\_probs = tf.reduce\_sum(action\_probs \* action\_masks, axis=1)

            loss = -tf.reduce\_sum(tf.math.log(selected\_action\_probs) \* advantages)

        grads = tape.gradient(loss, self.policy\_network.trainable\_variables)

        self.optimizer.apply\_gradients(zip(grads, self.policy\_network.trainable\_variables))

# Define the stock market environment

class StockMarketEnv:

    def \_\_init\_\_(self, price\_data):

        self.price\_data = price\_data

        self.current\_step = 0

        self.initial\_balance = 10000 # Initial investment balance

        self.balance = self.initial\_balance

        self.stock\_units = 0

        self.max\_steps = len(price\_data) - 1

    def reset(self):

        self.current\_step = 0

        self.balance = self.initial\_balance

        self.stock\_units = 0

        return [self.balance, self.stock\_units]

    def step(self, action):

        if self.current\_step >= self.max\_steps:

            return [self.balance, self.stock\_units], 0, True

        current\_price = self.price\_data[self.current\_step]

        next\_price = self.price\_data[self.current\_step + 1]

        if action == 1:  # Buy

            if self.balance >= current\_price:

                self.stock\_units += 1

                self.balance -= current\_price

        elif action == 0:  # Sell

            if self.stock\_units > 0:

                self.stock\_units -= 1

                self.balance += current\_price

        self.current\_step += 1

        # Calculate reward based on portfolio value

        portfolio\_value = self.balance + (self.stock\_units \* next\_price)

        reward = portfolio\_value - self.initial\_balance

        done = (self.current\_step == self.max\_steps)

        return [portfolio\_value, self.stock\_units], reward, done

# Training function for the VPG agent

def train\_vpg\_agent(agent, env, num\_episodes=1000):

    state\_dim = 2  # State: [portfolio\_value, stock\_units]

    action\_dim = 2  # Actions: [Buy (1), Sell (0)]

    for episode in range(num\_episodes):

        state = env.reset()

        states, actions, rewards = [], [], []

        done = False

        while not done:

            action = agent.select\_action(state)

            next\_state, reward, done = env.step(action)

            states.append(state)

            actions.append(action)

            rewards.append(reward)

            state = next\_state

        # Training episode-by-episode

        agent.train(states, actions, rewards)

        print(f"Episode: {episode + 1}, Total Reward: {sum(rewards)}")

if \_\_name\_\_ == "\_\_main\_\_":

    # Generate sample price data (replace with actual stock data)

    price\_data = np.random.uniform(50, 150, size=100)

    env = StockMarketEnv(price\_data)

    agent = VPGAgent(2, 2)

    train\_vpg\_agent(agent, env, num\_episodes=100)