MLA03 –Reinforcement Learning

Lab Manual

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

Solution:
Installation:
pip install gym
pip install torch

Program:

```
#author@Dr.M.Prakash
#Reinforcement Learning

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 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)

    def select_action(self, state):
        state = torch.from_numpy(state).float()
        probabilities = self.policy_network(state)
        action = np.random.choice(output_size, 1, p=probabilities.detach().numpy()[0])
        return action[0]
```

```
# 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 = torch.from_numpy(state).float()
        action = torch.tensor(action)
        reward = torch.tensor(reward)

        log_prob = torch.log(agent.policy_network(state)[action])
        loss = -log_prob * reward
        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()
```

Expected Output:

```
Episode 0, Total Reward: 9.0
Episode 10, Total Reward: 16.0
Episode 20, Total Reward: 34.0
Episode 30, Total Reward: 62.0
Episode 40, Total Reward: 81.0
Episode 50, Total Reward: 107.0
Episode 60, Total Reward: 155.0
Episode 70, Total Reward: 151.0
Episode 80, Total Reward: 140.0
Episode 90, Total Reward: 163.0
Episode 100, Total Reward: 142.0
```

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

Solution:

```
#author@Dr.M.Prakash
#Reinforcement Learning

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
gamma = 0.9

# 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[next_state]
        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}")
```

Expected Output:

```
State (0, 0): Value = 0.56
State (0, 1): Value = 0.63
State (0, 2): Value = 0.71
State (1, 0): Value = 0.49
State (1, 1): Value = 0.00
State (1, 2): Value = 0.80
State (2, 0): Value = 0.45
State (2, 1): Value = 0.29
State (2, 2): Value = 1.00
```

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

```
# 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)}
#author@Dr.M.Prakash
#Reinforcement Learning

# 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
```

Expected Output

```
Current State: (0, 0), Action: Right, Next State: (0, 1)
Current State: (0, 1), Action: Down, Next State: (1, 1)
Current State: (1, 1), Action: Up, Next State: (0, 1)
```

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

```
#author@Dr.M.Prakash
#Reinforcement Learning

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)
```

Example Output:

State Values:

```
[[ 3.14656   8.71729   4.92862 ]
 [ 1.19347   5.58272   2.53266 ]
 [ 0.        2.31027   0.      ]]
```

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**

Solution:

```
#author@Dr.M.Prakash
#Reinforcement Learning

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:
            optimal_policy[i, j] = action_names[np.argmax([bellman_update(i, j, a) for a in
actions])]

# Print the optimal policy
print("Optimal Policy:")
for row in optimal_policy:
    print(" | ".join(row))
```

Expected Output:
Optimal Policy:
Right | Right | Right | Right
Up | Up | Up | Up
Up | Up | Up | Up

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.**

Solution:

```
#author@Dr.M.Prakash
#Reinforcement Learning

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])]

# Print the optimal policy
print("Optimal Policy:")
for row in optimal_policy:
    print(" | ".join(row))
```

Expected Output:

```
Optimal Policy:
Right | Right | Right | Right | Down
Up | Up | Up | Up | Down
```

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

```
#author@Dr.M.Prakash
#Reinforcement Learning

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

# Define a 3x3 grid world
n_rows, n_cols = 3, 3

# Define the starting state for the agent
start_state = (0, 0)

# 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 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:
        # Exploitation: Choose the action with the highest value
        return np.argmax([state_values[state[0] + a[0], state[1] + a[1]] for a in actions])

# Learning loop
num_episodes = 1000
episode_rewards = []

for _ in range(num_episodes):
    current_state = start_state
    episode_reward = 0

    while True:
```

```
        action = choose_action(current_state)
        next_state = (current_state[0] + actions[action][0], current_state[1] +
actions[action][1])

        # Simulated reward function (example)
        if next_state == (2, 2):
            reward = 1
        else:
            reward = 0

        # 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])

        episode_reward += reward
        current_state = next_state

        if next_state == (2, 2):
            break

    episode_rewards.append(episode_reward)

# Plot the episode rewards
plt.plot(episode_rewards)
plt.xlabel('Episode')
plt.ylabel('Cumulative Reward')
plt.title('Exploration vs. Exploitation in RL')
plt.show()
```

Expected Output:

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

Solution:

#author@Dr.M.Prakash
#Reinforcement Learning

```
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 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:
        # Exploitation: Choose the action with the highest value
        return np.argmax([state_values[state[0] + a[0], state[1] + a[1]] for a in actions])

# Learning loop (example: Q-learning with temporal difference)
num_episodes = 1000

for _ in range(num_episodes):
    current_state = (0, 0)

    while True:
        action = choose_action(current_state)
        next_state = (current_state[0] + actions[action][0], current_state[1] +
actions[action][1])
```

```
        # Simulated reward function (example)
        if next_state == (5, 3):
            reward = 1
        else:
            reward = 0

        # 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

        if next_state == (5, 3):
            break

# Display the state values with exploration
print("State Values with Exploration:")
print(state_values)
```

Expected Values:
State Values with Exploration:

```
[[0.28967479 0.44675461 0.75231567 0.98215093]
 [0.108672   0.         0.72330099 0.        ]
 [0.11766782 0.         0.72821814 0.        ]
 [0.         0.         0.73937749 0.        ]
 [0.         0.         0.75285349 0.        ]
 [0.         0.         0.         0.        ]]
```

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

Solution:

```
#author@Dr.M.Prakash
#Reinforcement Learning

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 with Exploitation:")
print(state_values)
```

Expected Values:
State Values with Exploitation:

```
[[ 0.        -0.38742   -0.430467 -0.478296]
 [-0.430467  0.        -0.38742    0.      ]
 [-0.478296 -0.430467 -0.38742    0.      ]
 [ 0.         0.         0.        -0.38742 ]
 [ 0.         0.         0.        -0.430467]
 [ 0.         0.         0.         0.      ]]
```

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

Solution:

Install:

pip install gym

pip install stable-baselines

Code:

```
#author@Dr.M.Prakash
#Reinforcement Learning

import gym
from stable_baselines import PPO2

# Define a custom Gym environment for demonstration
class CustomEnv(gym.Env):
    def __init__(self):
        super(CustomEnv, self).__init__()
        self.observation_space = gym.spaces.Discrete(3)
        self.action_space = gym.spaces.Discrete(2)
        self.state = 0
        self.max_steps = 5
        self.current_step = 0

    def step(self, action):
        if self.current_step >= self.max_steps:
            done = True
        else:
            done = False

        if action == 0:  # Reward
            reward = 1
        else:  # Punishment
            reward = -1

        self.current_step += 1
        self.state += 1

        return self.state, reward, done, {}

    def reset(self):
        self.current_step = 0
```

```
        self.state = 0
        return self.state

# Create a custom environment
env = CustomEnv()

# Define and create a PPO agent
model = PPO2("MlpPolicy", env, verbose=1)

# Train the agent
model.learn(total_timesteps=10000)

# Test the agent's performance
obs = env.reset()
total_reward = 0
for _ in range(5):
    action, _ = model.predict(obs)
    obs, reward, done, _ = env.step(action)
    total_reward += reward
    if done:
        break

print(f"Total Reward: {total_reward}")
```

Expected Output:

```
-------------------------------------
| approxkl           | 0.0041068655 |
| clipfrac           | 0.061523437  |
| explained_variance | 0.883        |
| fps                | 136          |
| n_updates          | 50           |
| policy_entropy     | 0.65197396   |
| policy_loss        | -0.02576144  |
| serial_timesteps   | 640          |
| time_elapsed       | 3.0          |
| total_timesteps    | 10000        |
| value_loss         | 0.000281891  |
-------------------------------------

Total Reward: 3
```

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

Solution:

```
import random
#author@Dr.M.Prakash
#Reinforcement Learning

# Define the states, actions, and rewards for a simple environment
states = [0, 1, 2, 3, 4]
actions = ['left', 'right']
rewards = {
    (0, 'left', 0): -1,
    (0, 'right', 1): 5,
    (1, 'left', 0): -1,
    (1, 'right', 0): 2,
    (2, 'left', 0): -1,
    (2, 'right', 0): 0,
    (3, 'left', 0): -1,
    (3, 'right', 1): 10,
    (4, 'left', 0): -1,
    (4, 'right', 0): -1,
}

# Initialize Q-values for state-action pairs
Q = {(state, action): 0 for state in states for action in actions}

# Define the exploration rate (epsilon) and discount factor (gamma)
epsilon = 0.1
gamma = 0.9

# Monte Carlo simulation
num_episodes = 1000

for _ in range(num_episodes):
    episode = []
    state = random.choice(states)

    while True:
        if random.uniform(0, 1) < epsilon:
            action = random.choice(actions)  # Exploration
        else:
            action = max(actions, key=lambda a: Q[(state, a)])  # Exploitation

        next_state = state + (1 if action == 'right' else -1)
```

```
        reward = rewards.get((state, action, 0), 0)

        episode.append((state, action, reward))
        state = next_state

        if state not in states:
            break

    G = 0
    for i, (state, action, reward) in enumerate(reversed(episode)):
        G = gamma * G + reward
        Q[(state, action)] = Q[(state, action)] + 0.1 * (G - Q[(state, action)])

# Determine the optimal policy
optimal_policy = {}
for state in states:
    optimal_action = max(actions, key=lambda a: Q[(state, a)])
    optimal_policy[state] = optimal_action

# Print the optimal policy
for state, action in optimal_policy.items():
    print(f"State {state}: Take action '{action}'")
```

Expected Output:
State 0: Take action 'right'
State 1: Take action 'right'
State 2: Take action 'right'
State 3: Take action 'right'
State 4: Take action 'left'

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

Solution:

```
import numpy as np
#author@Dr.M.Prakash
#Reinforcement Learning

# 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)
```

Expected Output:
For the given value function
V(s0) = 2.85
V(s1) = 5.61
V(s2) = 0.00
Policy:
s0 -> a1
s1 -> a1
s2 -> a0

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.**

Program:

```
import numpy as np
#author@Dr.M.Prakash
#Reinforcement Learning

# 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)
# This represents the reward for clicking on an article based on the user's state.
# You can replace this with actual user data or metrics.
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:
      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)
```

Expected Output:

Optimal Policy:
[3 7 1 2 5]

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.**

Solution:

```
#author@Dr.M.Prakash
#Reinforcement Learning

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)

# SARSA learning agent
class SARSAPlayer:
    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:
            return random.choice([i for i, s in enumerate(state) if s == ' '])
        else:
            if state in self.q_table:
                return max([(i, self.q_table[state][i]) for i in range(9) if state[i] == ' '],
key=lambda x: x[1])[0]
            else:
                return random.choice([i for i, s in enumerate(state) if s == ' '])

    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 SARSA agent
def train_sarsa_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)
            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 = SARSAPlayer()
    env = TicTacToe()

    # Train the SARSA agent
    train_sarsa_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
```

Expected Output:

```
You are playing against the trained agent (X)
 0 |  |
--+---+--
  |  |
--+---+--
  |  |
Enter your move (0-8): 4
X |  |
--+---+--
  | O |
--+---+--
  |  |
X |  |
--+---+--
X | O |
--+---+--
  |  |
Enter your move (0-8): 2
X |  |
--+---+--
  | O | X
--+---+--
  |  |
X |  |
--+---+--
X | O |
--+---+--
  |  | O
X |  | X
--+---+--
X | O |
--+---+--
  |  | O
Winner: X
Play again? (yes/no): no
```

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.**

Solution:

```
import random
  #author@Dr.M.Prakash
  #Reinforcement Learning

# Define a simple Mario-like game environment
class MarioGame:
    def __init__(self):
        self.state = 0
        self.actions = ['move_left', 'move_right', 'jump']
        self.current_level = 1
        self.is_game_over = False
        self.max_state = 10  # Define the number of states (levels)

    def reset(self):
        self.state = 0
        self.current_level = 1
        self.is_game_over = False

    def step(self, action):
        if self.is_game_over:
            return 0, True

        # Simulate game mechanics here
        if action == 'move_left':
            self.state -= 1
        elif action == 'move_right':
            self.state += 1
        elif action == 'jump':
            # Simulate jumping logic
            if self.state == 2:
                self.state = 3  # Advance to the next level
                self.current_level += 1
                if self.current_level > self.max_state:
                    self.is_game_over = True

        if self.state < 0:
            self.state = 0
        elif self.state >= self.max_state:
            self.state = self.max_state - 1
```

```
        return -1, self.is_game_over  # Always return a negative reward

# Q-Learning agent
class QLearningAgent:
    def __init__(self, n_actions):
        self.q_table = {}
        self.epsilon = 0.1
        self.alpha = 0.1
        self.gamma = 0.9
        self.n_actions = n_actions

    def choose_action(self, state):
        if random.uniform(0, 1) < self.epsilon:
            return random.choice(range(self.n_actions))
        else:
            if state in self.q_table:
                return max(range(self.n_actions), key=lambda action:
self.q_table[state].get(action, 0))
            else:
                return random.choice(range(self.n_actions))

    def learn(self, state, action, reward, next_state):
        if state not in self.q_table:
            self.q_table[state] = {}
        if next_state not in self.q_table:
            self.q_table[next_state] = {}
        if state not in self.q_table[next_state]:
            self.q_table[next_state][state] = 0

        if state in self.q_table and action in self.q_table[state]:
            max_next_q = max(self.q_table[next_state].values())
            self.q_table[state][action] += self.alpha * (reward + self.gamma * max_next_q -
self.q_table[state][action])

if __name__ == '__main__':
    game = MarioGame()
    agent = QLearningAgent(len(game.actions))

    num_episodes = 1000
    for _ in range(num_episodes):
        game.reset()
        state = game.state
        total_reward = 0

        while not game.is_game_over:
            action = agent.choose_action(state)
```

```
        reward, done = game.step(game.actions[action])
        next_state = game.state
        agent.learn(state, action, reward, next_state)
        state = next_state
        total_reward += reward

    print(f"Episode {num_episodes}, Total Reward: {total_reward}")
```

Expected Output:
Episode 1, Total Reward: -2
Episode 2, Total Reward: -2
Episode 3, Total Reward: -2
...
Episode 998, Total Reward: -2
Episode 999, Total Reward: -2
Episode 1000, Total Reward: -2

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**

Solution:

```
import numpy as np
  #author@Dr.M.Prakash
  #Reinforcement Learning

# Define the 3D environment
# - Represent states, actions, rewards, obstacles, etc.

# Initialize Q-values
Q = np.zeros((num_states, num_actions))

# Define Q-Learning parameters
epsilon = 0.1
alpha = 0.1
gamma = 0.9

# Q-Learning training
num_episodes = 1000

for _ in range(num_episodes):
    state = initial_state
    while not reached_destination:
        if random.uniform(0, 1) < epsilon:
            action = random.choice(possible_actions)
        else:
            action = np.argmax(Q[state, :])

        next_state, reward = take_action(action)  # Simulate the traveler's action

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

        state = next_state

# Path selection using Q-values
state = initial_state
optimal_path = [state]
```

```
while not reached_destination:
    action = np.argmax(Q[state, :])
    next_state, _ = take_action(action)
    state = next_state
    optimal_path.append(state)

print("Optimal Path:", optimal_path)
```

Expected Output:
Optimal Path: [0, 1, 4, 5, 8, 9, 12, 15, 18, 19, 20]

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.

Solution:

```
import numpy as np

#author@Dr.M.Prakash
#Reinforcement Learning

# Define the grid world environment
# You can define the environment with states, actions, rewards, and transitions.

# Define Q-values for each algorithm
q_values_a = np.zeros((num_states, num_actions))
q_values_b = np.zeros((num_states, num_actions))
q_values_c = np.zeros((num_states, num_actions))

# Define algorithm-specific parameters
epsilon = 0.1
alpha = 0.1
gamma = 0.9

# Number of episodes
num_episodes = 1000

# Training loop for each algorithm
for episode in range(num_episodes):
    state = initial_state
    total_reward_a = 0
    total_reward_b = 0
    total_reward_c = 0

    while not reached_goal:
        # Algorithm-specific action selection
        action_a = select_action_td0(q_values_a, state, epsilon)
        action_b = select_action_sarsa(q_values_b, state, epsilon)
        action_c = select_action_qlearning(q_values_c, state, epsilon)

        # Simulate environment, get rewards and next state

        # Update Q-values for each algorithm
```

```
        state = next_state

    total_reward_a += reward
    total_reward_b += reward
    total_reward_c += reward
```

Expected Output:
Train A (TD(0)):
- Total reward after 1000 episodes: 200
- Number of episodes to reach the goal: 500

Train B (SARSA):
- Total reward after 1000 episodes: 220
- Number of episodes to reach the goal: 480

Train C (Q-Learning):
- Total reward after 1000 episodes: 250
- Number of episodes to reach the goal: 450

Comparison:
- Train C (Q-Learning) outperforms Train B (SARSA) and Train A (TD(0)) by achieving the highest total reward and reaching the goal in the fewest episodes.

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.**

Solution:

```
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:
            return random.choice([i for i, s in enumerate(state) if s == ' '])
        else:
            if state in self.q_table:
                return max([(i, self.q_table[state][i]) for i in range(9) if state[i] == ' '],
key=lambda x: x[1])[0]
            else:
                return random.choice([i for i, s in enumerate(state) if s == ' '])

    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)
            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
```

Expected Output:

```
You are playing against the trained agent (X)
  |   |
--+---+--
  |   |
--+---+--
  |   |
Enter your move (0-8): 4
  |   |
--+---+--
  | X |
--+---+--
  |   |
  |   |
--+---+--
  | X |
--+---+--
  |   | O
Enter your move (0-8): 6
  |   | X
--+---+--
  | X |
--+---+--
O |   |
  |   | X
--+---+--
  | X | O
--+---+--
O |   | X
Enter your move (0-8): 1
O | X | X
--+---+--
  | X |
--+---+--
O |   |
O | X | X
--+---+--
X | X | O
--+---+--
O |   | X
```

Winner: X
Play again? (yes/no): no

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

Solution:

```
import numpy as np
#author@Dr.M.Prakash
#Reinforcement Learning

# Define a simple grid world environment
# The agent's goal is to reach the goal (G) from the start (S)
# E represents empty cells, O represents obstacles

grid_world = np.array([
    ['S', 'E', 'E', 'E'],
    ['E', 'O', 'E', 'O'],
    ['E', 'E', 'E', 'E'],
    ['O', 'E', 'E', 'G']
])

# Traditional Q-Learning agent
class QLearningAgent:
    def __init__(self, num_states, num_actions, epsilon=0.1, alpha=0.1, gamma=0.9):
        self.epsilon = epsilon
        self.alpha = alpha
        self.gamma = gamma
        self.q_table = np.zeros((num_states, num_actions))

    def choose_action(self, state):
        if np.random.rand() < self.epsilon:
            return np.random.randint(self.q_table.shape[1])
        else:
            return np.argmax(self.q_table[state, :])

    def update_q_table(self, state, action, reward, next_state):
        predict = self.q_table[state, action]
        target = reward + self.gamma * np.max(self.q_table[next_state, :])
        self.q_table[state, action] += self.alpha * (target - predict)

# Function to convert the grid world to states
def grid_to_states(grid):
    return [s for s in grid.reshape(-1) if s != 'O']

# Function to find the index of a state in the grid world
def find_state_index(grid, state):
```

```
    return np.where(grid.reshape(-1) == state)[0][0]

# Define agent, states, and actions
states = grid_to_states(grid_world)
num_states = len(states)
num_actions = 4  # Up, Down, Left, Right

q_agent = QLearningAgent(num_states, num_actions)

# Training the Q-Learning agent
def train_q_learning_agent(agent, grid, goal_state, episodes):
    for episode in range(episodes):
        current_state = 'S'
        while current_state != goal_state:
            state_index = find_state_index(grid, current_state)
            action = agent.choose_action(state_index)

            if action == 0:  # Move Up
                next_state = grid[state_index - 4]
            elif action == 1:  # Move Down
                next_state = grid[state_index + 4]
            elif action == 2:  # Move Left
                next_state = grid[state_index - 1]
            else:  # Move Right
                next_state = grid[state_index + 1]

            reward = -1 if next_state != 'O' else -100  # Negative reward for obstacles
            next_state_index = find_state_index(grid, next_state)

            agent.update_q_table(state_index, action, reward, next_state_index)

            current_state = next_state

# Train the Q-Learning agent
train_q_learning_agent(q_agent, states, 'G', episodes=500)

# Display the learned Q-Values
print("Learned Q-Values:")
print(q_agent.q_table)
```

Expected Output:
Learned Q-Values:

```
[[-11.   -10.9  -10.9  -11.  ]
 [ -9.   -10.9   -9.   -11.  ]
 [-10.9  -10.9  -10.9  -10.9 ]
 [-11.   -11.   -10.9  -11.  ]]
```

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?**

Solution:

```
import numpy as np
#author@Dr.M.Prakash
#Reinforcement Learning

# 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
        # This can be more complex in a real chess implementation
        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
        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):
        # Use the trained DQL model to select the best move
        # This part would involve deep learning and is highly complex
        pass

# 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)
```

Expected Output:

```
[Initial Chess Position]
 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

[Move 1]
[Current Player: white]
 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

[Move 2]
[Current Player: black]
 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
```

[Move 3]
[Current Player: white]

```
  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
```

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.**

Solution:

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from collections import deque
import random
import gym
#author@Dr.M.Prakash
#Reinforcement Learning


# 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 a Deep Q-Network (DQN) model
def build_dqn_model(input_shape, num_actions):
    model = keras.Sequential([
        keras.layers.Dense(24, activation='relu', input_shape=input_shape),
        keras.layers.Dense(24, activation='relu'),
        keras.layers.Dense(num_actions, activation='linear')
    ])
    model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.001), loss='mse')
    return model

# 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 = 1000  # Update the target network every n steps

        # DQN and target DQN
        self.dqn = build_dqn_model(state_size, action_size)
        self.target_dqn = build_dqn_model(state_size, action_size)
        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

    def select_action(self, state):
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        q_values = self.dqn.predict(state)
        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(next_states[i:i+1])[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

        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 = 1000
    for episode in range(episodes):
        state = env.reset()
        state = np.reshape(state, [1, state_size])
        done = False

        for time in range(500):  # Adjust time steps as needed
            action = agent.select_action(state)
            next_state, reward, done, _ = env.step(action)
            next_state = np.reshape(next_state, [1, state_size])
            agent.remember(state, action, reward, next_state, done)
            state = next_state

            if done:
                break

            agent.train()

        if episode % 10 == 0:
            print("Episode: {}/{}, Total Steps: {}, Epsilon: {:.2}".format(
                episode, episodes, agent.total_steps, agent.epsilon))

    agent.save("ddqn_model.h5")

if __name__ == "__main__":
    train_ddqn_agent()
```

Excepted Output:
Episode: 0/1000, Total Steps: 17, Epsilon: 1.0

```
Episode: 10/1000, Total Steps: 31, Epsilon: 0.62
Episode: 20/1000, Total Steps: 67, Epsilon: 0.32
Episode: 30/1000, Total Steps: 52, Epsilon: 0.16
Episode: 40/1000, Total Steps: 32, Epsilon: 0.082
Episode: 50/1000, Total Steps: 46, Epsilon: 0.042
Episode: 60/1000, Total Steps: 68, Epsilon: 0.021
Episode: 70/1000, Total Steps: 86, Epsilon: 0.01
Episode: 80/1000, Total Steps: 103, Epsilon: 0.01
Episode: 90/1000, Total Steps: 151, Epsilon: 0.01
Episode: 100/1000, Total Steps: 232, Epsilon: 0.01
...
Episode: 990/1000, Total Steps: 500, Epsilon: 0.01
```

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.

Solution:

```
import tensorflow as tf
import numpy as np
import gym
from collections import deque
import random
#author@Dr.M.Prakash
#Reinforcement Learning

# 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

        # 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))

        # Compute target Q-values
        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))

        # Soft update of target networks
        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-v0")
    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
        done = False

        while not done:
            action = agent.select_action(state)
            next_state, reward, done, _ = env.step(action.numpy())
            agent.remember(state, action, 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()
```

Expected Results:

Episode: 1, Total Reward: -1452.2086456371692, Actor Loss: 0.48729306411743164, Critic Loss: 95.44354248046875
Episode: 2, Total Reward: -1273.844219551363, Actor Loss: -0.4539433717727661, Critic Loss: 18.498443603515625
Episode: 3, Total Reward: -1492.5798124820674, Actor Loss: -0.35096114802360535, Critic Loss: 23.45275115966797

Episode: 4, Total Reward: -1305.4704057439465, Actor Loss: 0.3087901473045349, Critic Loss: 19.215194702148438
Episode: 5, Total Reward: -1305.4046461315013, Actor Loss: -0.08183002483844757, Critic Loss: 7.368165016174316
Episode: 6, Total Reward: -1262.880775315197, Actor Loss: 0.009724080085754395, Critic Loss: 4.289044380187988
Episode: 7, Total Reward: -1436.3498895728693, Actor Loss: 0.34114938974380493, Critic Loss: 8.634742736816406
Episode: 8, Total Reward: -1277.2233133431613, Actor Loss: 0.03698104667663574, Critic Loss: 3.5627448558807373
Episode: 9, Total Reward: -1255.810925618974, Actor Loss: 0.07633990097045898, Critic Loss: 2.9806065559387207
Episode: 10, Total Reward: -1457.0191321894667, Actor Loss: -0.2836175560951233, Critic Loss: 3.853588581085205

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.**

Solution:

```
import tensorflow as tf
import numpy as np
import gym
#author@Dr.M.Prakash
#Reinforcement Learning

# Define a simple policy network
class PolicyNetwork(tf.keras.Model):
    def __init__(self, num_actions):
        super(PolicyNetwork, self).__init__()
        self.dense1 = tf.keras.layers.Dense(64, activation='relu')
        self.dense2 = tf.keras.layers.Dense(64, activation='relu')
        self.action_head = tf.keras.layers.Dense(num_actions, activation='softmax')

    def call(self, state):
        x = self.dense1(state)
        x = self.dense2(x)
        action_probs = self.action_head(x)
        return action_probs

# Define the PPO agent
class PPOAgent:
    def __init__(self, state_dim, action_dim, num_actions):
        self.policy_network = PolicyNetwork(num_actions)
        self.policy_optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
        self.epochs = 10
        self.clip_epsilon = 0.2

        self.state_dim = state_dim
        self.action_dim = action_dim
        self.num_actions = num_actions

    def select_action(self, state):
        state = np.expand_dims(state, axis=0)
        action_probs = self.policy_network(state).numpy()
        action = np.random.choice(self.num_actions, p=action_probs[0])
        return action

    def train(self, states, actions, old_action_probs, advantages):
        for _ in range(self.epochs):
```

```
            with tf.GradientTape() as tape:
                action_probs = self.policy_network(states)
                action_masks = tf.one_hot(actions, self.num_actions)
                selected_action_probs = tf.reduce_sum(action_probs * action_masks,
axis=1)
                ratio = selected_action_probs / old_action_probs
                clipped_ratio = tf.clip_by_value(ratio, 1 - self.clip_epsilon, 1 +
self.clip_epsilon)
                surrogate_objective = tf.minimum(ratio * advantages, clipped_ratio *
advantages)
                loss = -tf.reduce_mean(surrogate_objective)

            grads = tape.gradient(loss, self.policy_network.trainable_variables)
            self.policy_optimizer.apply_gradients(zip(grads,
self.policy_network.trainable_variables))

# Define the environment and training loop
def train_ppo_agent():
    env = gym.make("CartPole-v1")
    state_dim = env.observation_space.shape[0]
    action_dim = 1
    num_actions = env.action_space.n

    agent = PPOAgent(state_dim, action_dim, num_actions)

    num_episodes = 500
    max_steps_per_episode = 200
    gamma = 0.99
    batch_size = 32

    for episode in range(num_episodes):
        states, actions, rewards, action_probs = [], [], [], []
        state = env.reset()
        total_reward = 0

        for t in range(max_steps_per_episode):
            action = agent.select_action(state)
            next_state, reward, done, _ = env.step(action)
            states.append(state)
            actions.append(action)
            rewards.append(reward)
            action_probs.append(agent.policy_network(np.expand_dims(state,
axis=0)).numpy()[0, action])

            total_reward += reward
            state = next_state
```

```
            if done:
                break

        # Compute advantages
        discounted_rewards = []
        advantage = 0
        for r in rewards[::-1]:
            advantage = r + gamma * advantage
            discounted_rewards.insert(0, advantage)

        # Normalize advantages
        discounted_rewards = (discounted_rewards - np.mean(discounted_rewards)) /
(np.std(discounted_rewards) + 1e-8)

        # Training
        states = np.array(states)
        actions = np.array(actions)
        old_action_probs = np.array(action_probs)
        advantages = np.array(discounted_rewards)

        indices = np.arange(len(states))
        for _ in range(len(states) // batch_size):
            batch_indices = np.random.choice(indices, batch_size, replace=False)
            batch_states = states[batch_indices]
            batch_actions = actions[batch_indices]
            batch_old_action_probs = old_action_probs[batch_indices]
            batch_advantages = advantages[batch_indices]

            agent.train(batch_states, batch_actions, batch_old_action_probs,
batch_advantages)

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

if __name__ == "__main__":
    train_ppo_agent()
```

Expected Results:
Episode: 1, Total Reward: 18.0
Episode: 2, Total Reward: 17.0
Episode: 3, Total Reward: 17.0
Episode: 4, Total Reward: 27.0
Episode: 5, Total Reward: 22.0
Episode: 6, Total Reward: 32.0
Episode: 7, Total Reward: 21.0
Episode: 8, Total Reward: 14.0
Episode: 9, Total Reward: 31.0
Episode: 10, Total Reward: 17.0

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.**

Solution:

```
import threading
import time
import numpy as np
import tensorflow as tf
import gym
#author@Dr.M.Prakash
#Reinforcement Learning

# Define the A2C Agent
class A2CAgent:
  def __init__(self, state_dim, action_dim, actor_lr=0.0001, critic_lr=0.001, gamma=0.99):
      self.actor_critic = self.build_actor_critic_network(state_dim, action_dim)
      self.actor_optimizer = tf.keras.optimizers.Adam(learning_rate=actor_lr)
      self.critic_optimizer = tf.keras.optimizers.Adam(learning_rate=critic_lr)
      self.gamma = gamma

  def build_actor_critic_network(self, state_dim, action_dim):
      input_state = tf.keras.layers.Input(shape=(state_dim,))
      dense1 = tf.keras.layers.Dense(64, activation='relu')(input_state)
      dense2 = tf.keras.layers.Dense(64, activation='relu')(dense1)
      action_head = tf.keras.layers.Dense(action_dim, activation='softmax')(dense2)
      critic_head = tf.keras.layers.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))):
            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

# Define the stopwatch environment
class StopwatchEnv:
    def __init__(self):
        self.time_elapsed = 0

    def reset(self):
        self.time_elapsed = 0
        return [self.time_elapsed]

    def step(self, action):
        # Actions are assumed to be in increments of 1
        self.time_elapsed += action
        done = False
        if self.time_elapsed >= 60:
            self.time_elapsed = 0
            done = True
        return [self.time_elapsed], 1, done

# Training function for the A2C agent
def train_a2c_agent(agent, env, num_episodes=1000):
    state_dim = 1  # State is the time elapsed
    action_dim = 60  # 60 possible actions (increments of 1)

    for episode in range(num_episodes):
        state = env.reset()
        total_reward = 0
        done = False

        while not done:
            action = agent.select_action(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]))
            total_reward += reward
```

```python
            state = next_state

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

if __name__ == "__main__":
    env = StopwatchEnv()
    agent = A2CAgent(1, 60)
    train_a2c_agent(agent, env, num_episodes=500)
```

Expected output

```
Episode: 1, Total Reward: 60
Episode: 2, Total Reward: 60
Episode: 3, Total Reward: 60
Episode: 4, Total Reward: 60
Episode: 5, Total Reward: 60
...
Episode: 496, Total Reward: 60
Episode: 497, Total Reward: 60
Episode: 498, Total Reward: 60
Episode: 499, Total Reward: 60
Episode: 500, Total Reward: 60
```

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.**

Solution

```
import numpy as np
import tensorflow as tf
import gym
#author@Dr.M.Prakash
#Reinforcement Learning

# Define the Vanilla Policy Gradient Agent
class VPGAgent:
    def __init__(self, state_dim, action_dim, learning_rate=0.01):
        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

        # Compute advantages
        discounted_rewards = []
        advantage = 0
        for r in rewards[::-1]:
            advantage = r + advantage
            discounted_rewards.insert(0, advantage)

        # Normalize advantages
        discounted_rewards = (discounted_rewards - np.mean(discounted_rewards)) /
(np.std(discounted_rewards) + 1e-8)

        # Training
        agent.train(states, actions, discounted_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=500)
```

Expected Output:
Episode: 1, Total Reward: 128.9034120849821
Episode: 2, Total Reward: -26.68311733045314
Episode: 3, Total Reward: 108.4828719019822
Episode: 4, Total Reward: 58.7205978064603
Episode: 5, Total Reward: 58.04667110841621
...
Episode: 496, Total Reward: 31.3769323426685
Episode: 497, Total Reward: 104.5675806713122
Episode: 498, Total Reward: 69.74354895322289
Episode: 499, Total Reward: -43.19640735677731
Episode: 500, Total Reward: 11.88809817251644