|  |  |
| --- | --- |
| **EX NO:1** | **MARKOV DECISION PROCESS** |

**AIM:**

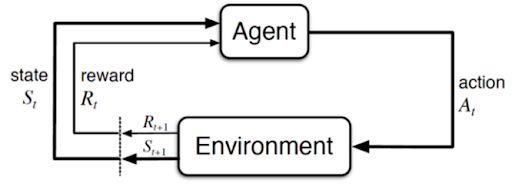
To implement Markov Decision Process.

**INTRODUCTION:**

The MDP framework has the following key components:

* *S*: states (*s ∈ S*)
* *A*: Actions *(a ∈ A)*
* *P (St+1*|*st.at)*: Transition probabilities
* R *(s)*: Reward

The graphical representation of the MDP model:



The Markov Property can be evaluated by using this equation:

P[St+1|St] = P [St+1 |S1,S2,S3……St]

**ALGORITHM:**

**Step-1:** Set Up Grid: Create a 2D grid with specified width and height. The agent starts at the top-left corner and the goal is at the bottom-right corner.

**Step-2:** Move Agent: The agent can move up (North), right (East), down (South), or left (West).

**Step-3:** Update Position: When the agent moves, the code updates its position. If a move would take the agent off the grid, it is kept within the grid's boundaries instead.

**Step-4:** Check Goal: After each move, the code checks if the agent has reached the goal. If it has, the game ends.

**Step-5:** Reward Agent: Each move gets a reward of -1. The goal is to reach the end as quickly as possible because more moves mean a lower total reward.

**Step-6:** Print Grid: If in debug mode, the current state of the grid is printed after each move.

**Step-7:** Run Simulation: The agent is manually moved around the grid in a series of steps, with the state of the grid and the results of each move printed out.

**IMPLEMENTATION:**

from collections import defaultdict, namedtuple

from enum import Enum

from typing import Tuple, List

import random

from IPython.display import clear\_output

Point = namedtuple('Point', ['x', 'y'])

class Direction(Enum):

NORTH = "⬆"

EAST = "⮕"

SOUTH = "⬇"

WEST = "⬅"

@classmethod

def values(self):

return [v for v in self]

class SimpleGridWorld(object):

def \_\_init\_\_(self, width: int = 5, height: int = 5, debug: bool = False):

self.width = width

self.height = height

self.debug = debug

self.action\_space = [d for d in Direction]

self.reset()

class SimpleGridWorld(SimpleGridWorld):

def reset(self):

self.cur\_pos = Point(x=0, y=(self.height - 1))

self.goal = Point(x=(self.width - 1), y=0)

# If debug, print state

if self.debug:

print(self)

return self.cur\_pos, 0, False

class SimpleGridWorld(SimpleGridWorld):

def step(self, action: Direction):

# Depending on the action, mutate the environment state

if action == Direction.NORTH:

self.cur\_pos = Point(self.cur\_pos.x, self.cur\_pos.y + 1)

elif action == Direction.EAST:

self.cur\_pos = Point(self.cur\_pos.x + 1, self.cur\_pos.y)

elif action == Direction.SOUTH:

self.cur\_pos = Point(self.cur\_pos.x, self.cur\_pos.y - 1)

elif action == Direction.WEST:

self.cur\_pos = Point(self.cur\_pos.x - 1, self.cur\_pos.y)

# Check if out of bounds

if self.cur\_pos.x >= self.width:

self.cur\_pos = Point(self.width - 1, self.cur\_pos.y)

if self.cur\_pos.y >= self.height:

self.cur\_pos = Point(self.cur\_pos.x, self.height - 1)

if self.cur\_pos.x < 0:

self.cur\_pos = Point(0, self.cur\_pos.y)

if self.cur\_pos.y < 0:

self.cur\_pos = Point(self.cur\_pos.x, 0)

# If at goal, terminate

is\_terminal = self.cur\_pos == self.goal

# Constant -1 reward to promote speed-to-goal

reward = -1

# If debug, print state

if self.debug:

print(self)

return self.cur\_pos, reward, is\_terminal

class SimpleGridWorld(SimpleGridWorld):

def \_\_repr\_\_(self):

res = ""

for y in reversed(range(self.height)):

for x in range(self.width):

if self.goal.x == x and self.goal.y == y:

if self.cur\_pos.x == x and self.cur\_pos.y == y:

res += "@"

else:

res += "o"

continue

if self.cur\_pos.x == x and self.cur\_pos.y == y:

res += "x"

else:

res += "\_"

res += "\n"

return res

s = SimpleGridWorld(debug=True)

print("☝ This shows a simple visualisation of the environment state.\n")

s.step(Direction.SOUTH)

print(s.step(Direction.SOUTH), "⬅ This displays the state and reward from the environment 𝐀𝐅𝐓𝐄𝐑 moving.\n")

s.step(Direction.SOUTH)

s.step(Direction.SOUTH)

s.step(Direction.EAST)

s.step(Direction.EAST)

s.step(Direction.EAST)

s.step(Direction.EAST)

**OUTPUT:**

A screenshot of a test

Description automatically generated

**RESULT:**

Thus, the Markov Decision Process have been implemented successfully.

|  |  |
| --- | --- |
| **EX NO 2** | **MARKOV DECISION PROCESS ITERATIONS** |

**AIM:**

To implement an integrated approach that combines MDP value iteration and policy iteration methods to enhance convergence and efficiency in solving reinforcement learning problems.

**INTRODUCTION:**

Unlike policy evaluation which has linear equations that can be solved directly, in value iteration, because of the max operation, the equations are not linear anymore. As a result, we must use an iterative procedure to solve them.

Similar as we did in policy iteration, we start from initializing the utility of every state as zero and we set γ as 0.5. What we need to do is to loop through states using the Bellman equation.

A black and white image of a symbol

Description automatically generated

Here, v(s) 🡪 Value of current state

r(s) 🡪 Reward of current state

γ 🡪 Discount Factor

s’ 🡪 Next state

s, a 🡪 State, Action

v(s’) 🡪 Value of next state

**ALGORITHM:**

**STEP 1**: Initialize the value function arbitrarily for all states.

**STEP 2:** Update the value function iteratively using the Bellman equation, considering the immediate reward and the expected value of successor states.

**STEP 3:** For each state, update the policy by selecting the action that maximizes the expected cumulative reward according to the current value function.

**STEP 4:** Repeat steps 2 and 3 until the change in the value function or policy becomes negligible.

**STEP 5:** Once convergence is reached, return the optimal value function and policy for making decisions in the MDP.

**IMPLEMENTATION:**

REWARD = -0.01

DISCOUNT = 0.99

MAX\_ERROR = 10\*\*(-3)

NUM\_ACTIONS = 4

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

NUM\_ROW = 3

NUM\_COL = 4

U = [[0, 0, 0, 1], [0, 0, 0, -1], [0, 0, 0, 0], [0, 0, 0, 0]]

def printEnvironment(arr, policy=False):

res = ""

for r in range(NUM\_ROW):

res += "|"

for c in range(NUM\_COL):

if r == c == 1:

val = "WALL"

elif r <= 1 and c == 3:

val = "+1" if r == 0 else "-1"

else:

if policy:

val = ["Down", "Left", "Up", "Right"][arr[r][c]]

else:

val = str(arr[r][c])

res += " " + val[:5].ljust(5) + " |" # format

res += "\n"

print(res)

def getU(U, r, c, action):

dr, dc = ACTIONS[action]

newR, newC = r+dr, c+dc

if newR < 0 or newC < 0 or newR >= NUM\_ROW or newC >= NUM\_COL or (newR == newC == 1):

return U[r][c]

else:

return U[newR][newC]

def calculateU(U, r, c, action):

u = REWARD

u += 0.1 \* DISCOUNT \* getU(U, r, c, (action-1) % 4)

u += 0.8 \* DISCOUNT \* getU(U, r, c, action)

u += 0.1 \* DISCOUNT \* getU(U, r, c, (action+1) % 4)

return u

def valueIteration(U):

print("During the value iteration:\n")

while True:

nextU = [[0, 0, 0, 1], [0, 0, 0, -1], [0, 0, 0, 0], [0, 0, 0, 0]]

error = 0

for r in range(NUM\_ROW):

for c in range(NUM\_COL):

if (r <= 1 and c == 3) or (r == c == 1):

continue

nextU[r][c] = max([calculateU(U, r, c, action) for action in range(NUM\_ACTIONS)])

error = max(error, abs(nextU[r][c] - U[r][c]))

U = nextU

printEnvironment(U)

if error < MAX\_ERROR \* (1 - DISCOUNT) / DISCOUNT:

break

return U

def getOptimalPolicy(U):

policy = [[-1, -1, -1, -1] for i in range(NUM\_ROW)]

for r in range(NUM\_ROW):

for c in range(NUM\_COL):

if (r <= 1 and c == 3) or (r == c == 1):

continue

maxAction, maxU = None, -float("inf")

for action in range(NUM\_ACTIONS):

u = calculateU(U, r, c, action)

if u > maxU:

maxAction, maxU = action, u

policy[r][c] = maxAction

return policy

print("The initial U is:\n")

printEnvironment(U)

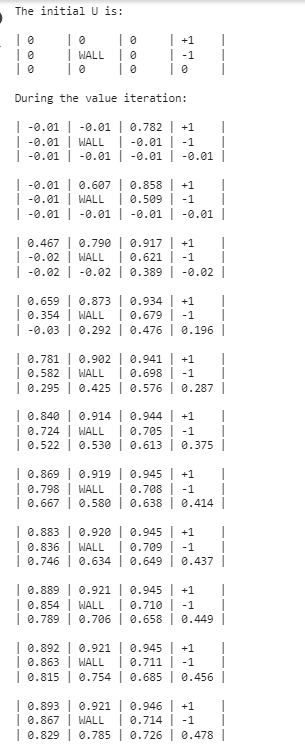
U = valueIteration(U)

policy = getOptimalPolicy(U)

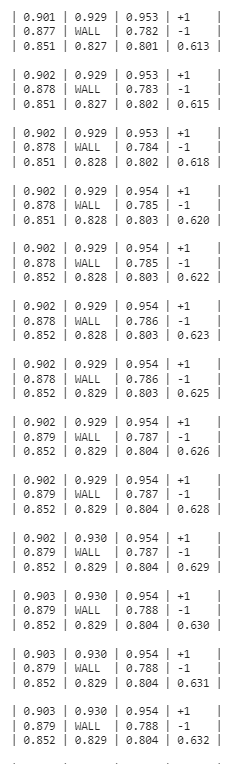
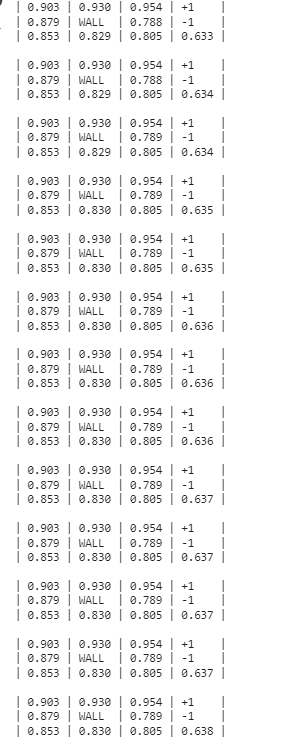
print("The optimal policy is:\n")

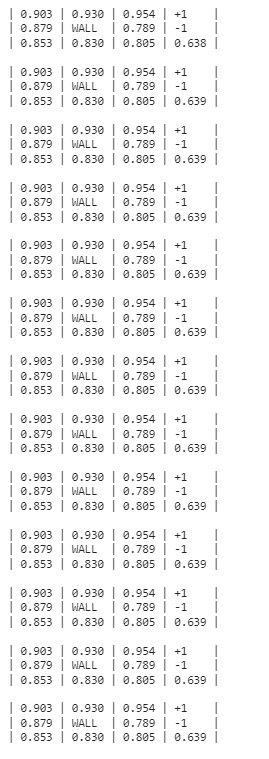
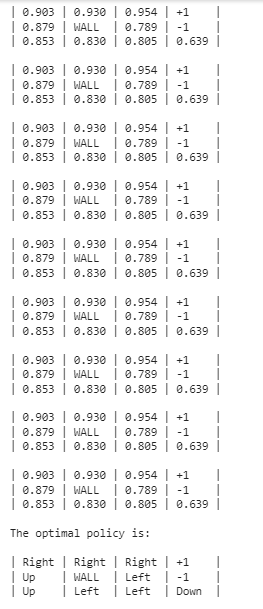
printEnvironment(policy, True)

**OUTPUT:**

 A table of numbers with numbers and symbols

Description automatically generated with medium confidence





**RESULT:**

Thus, the MDP Value Iteration has been implemented successfully.

|  |  |
| --- | --- |
| **EX NO:3** | **TEMPORAL DIFFERENCE LEARNING** |

**AIM:**

The aim of Temporal Difference (TD) Learning is to develop and apply a reinforcement learning algorithm that enables an agent to learn optimal policies for sequential decision-making tasks.

**INTRODUCTION:**

Temporal Difference (TD) learning is used to approximate the values of states (or state-action pairs) and learn an optimal policy in a Markov Decision Process (MDP) or a similar environment. TD learning combines elements of dynamic programming and Monte Carlo methods to update value estimates based on both immediate rewards and predictions about future rewards. It is particularly useful in scenarios where you want to learn from incomplete sequences of experiences.

**A diagram of a policy

Description automatically generated**

δ\_t = R\_t+1 + γ \* V(S\_t+1) - V(S\_t)

**Where:**

* δ\_t is the TD error at time step t for state S\_t and action A\_t.
* R\_t+1 is the immediate reward received after taking action A\_t in state S\_t and transitioning to state S\_t+1.
* γ (gamma) is the discount factor, representing the importance of future rewards. It's a value between 0 and 1.
* V(S\_t+1) is the estimated value of the next state S\_t+1.
* V(S\_t) is the estimated value of the current state S\_t.

**ALGORITHM:**

**STEP1:** Initialize the value function or policy.

**STEP2:** Interaction with the Environment.

**STEP 3:** Update the Value Function(s).

**STEP 4:** TD Learning can update the policy based on the updated value function(s).

**STEP 5:** Repeat steps 4 and 2 iteratively for multiple episodes until the agent's policy and value function(s) converge to an optimal or near optimal solution.

**IMPLEMENTATION:**

import random

class RandomWalkEnvironment:

def \_\_init\_\_(self, num\_states):

self.num\_states = num\_states

self.current\_state = num\_states // 2 # Start in the middle

self.actions = [-1, 1] # Left and right

def reset(self):

self.current\_state = self.num\_states // 2

def step(self, action):

reward = 0

done = False

new\_state = self.current\_state + action

if new\_state == 0:

reward = -1

done = True

elif new\_state == self.num\_states - 1:

reward = 1

done = True

self.current\_state = new\_state

return new\_state, reward, done

def render(self):

env\_str = "|" + " " \* self.num\_states + "|"

env\_str = env\_str[:self.current\_state+1] + "X" + env\_str[self.current\_state+2:]

print(env\_str)

class TDLearner:

def \_\_init\_\_(self, num\_states, alpha, gamma):

self.num\_states = num\_states

self.alpha = alpha # Learning rate

self.gamma = gamma # Discount factor

self.values = [0] \* num\_states

def update(self, state, reward, next\_state):

td\_target = reward + self.gamma \* self.values[next\_state]

td\_error = td\_target - self.values[state]

self.values[state] += self.alpha \* td\_error

def get\_action(self, state):

# Epsilon-greedy policy

if random.random() < 0.5:

return random.choice([-1, 1])

else:

if state == 0:

return 1

elif state == self.num\_states - 1:

return -1

else:

return -1 if self.values[state - 1] > self.values[state + 1] else 1

def main():

num\_states = 7

env = RandomWalkEnvironment(num\_states)

td\_learner = TDLearner(num\_states, alpha=0.1, gamma=1.0)

num\_episodes = 10

for episode in range(num\_episodes):

state = env.current\_state

env.reset()

done = False

print(f"Episode {episode + 1}")

env.render()

while not done:

action = td\_learner.get\_action(state)

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

td\_learner.update(state, reward, next\_state)

state = next\_state

env.render()

print("Learned values:", td\_learner.values)

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

main()

A line of black x

Description automatically generated with medium confidence**OUTPUT SCREENSHOT:**

A piece of paper with black x

Description automatically generated

A white paper with black x and a white background

Description automatically generated with medium confidence A white background with black text

Description automatically generated

**RESULT:**

Thus, the Temporal Difference Learning has been implemented successfully.

|  |  |
| --- | --- |
| **EX NO:4** | **TEMPORAL DIFFERENCE METHOD TD (0) AND TD(LAMBDA)** |

**AIM:**

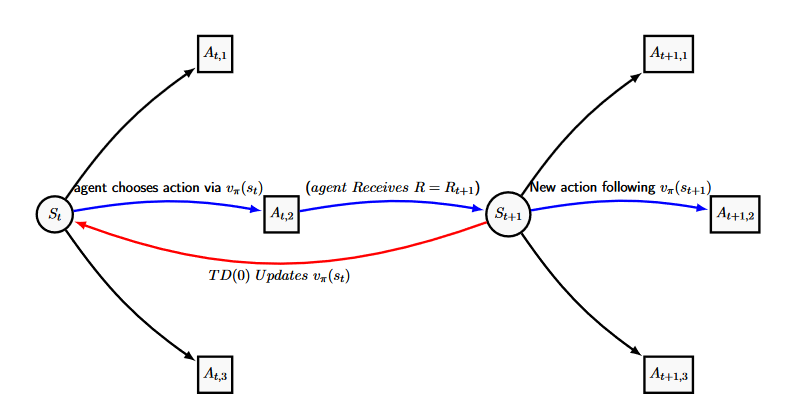
To implement Temporal Difference method TD (0) and TD(Lambda).

**INTRODUCTION:**

**TD (0):**

Temporal-Difference TD (0) learning updates the estimated value of a state V for policy based on the reward the agent received and the value of the state it transitioned to. Specifically, if our agent is in a current state st, takes the action at and receives the reward rt, then we update our estimate of V following.

V(st)←V(st)+α[rt+1+γV(st+1)–V(st)]



**TD(λ):**

(Temporal Difference (TD) methods, including TD(λ), are a class of reinforcement learning algorithms that combine elements of both Monte Carlo methods and dynamic programming. These methods are used to estimate value functions and learn optimal policies in Markov Decision Processes (MDPs) or similar environments. TD(λ) extends basic TD learning (TD (0)) by incorporating eligibility traces to give credit to multiple preceding states and actions. This enables more efficient and flexible learning in scenarios with delayed or sparse rewards.

**ALGORITHM:**

**Step 1:** Define environment constants and value function class for TD (lambda) learning.

**Step 2:** Simulate random walk interactions with action choices, transitions, and rewards.

**Step 3:** Iterate over learning rates and lambdas, simulating TD (lambda) learning and calculating RMSE.

**Step 4:** Use plotting libraries to create RMSE vs. alpha plots for different lambdas.

**Step 5:** Display plots illustrating how alpha and lambda impact TD (lambda) learning.

**IMPLEMENTATION:**

import numpy as np

import matplotlib.pyplot as plt

NUM\_STATES = 21

START = 9

END\_0 = 0

END\_1 = 20

class ValueFunctionTD:

    def \_\_init\_\_(self, alpha=0.1, gamma=0.9, lmbda=0.8):

        self.weights = np.zeros(NUM\_STATES)

        self.z = np.zeros(NUM\_STATES)

        self.alpha = alpha

        self.gamma = gamma

        self.lmbda = lmbda

    def value(self, state):

        v = self.weights[state]

        return v

    def updateZ(self, state):

        dev = 1

        self.z \*= self.gamma \* self.lmbda

        self.z[state] += dev

    def learn(self, state, nxtState, reward):

        delta = reward + self.gamma \* self.value(nxtState) - self.value(state)

        delta \*= self.alpha

        self.weights += delta \* self.z

class RWTD:

    def \_\_init\_\_(self, start=START, end=False, debug=False):

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

        self.state = start

        self.end = end

        self.reward = 0

        self.debug = debug

    def chooseAction(self):

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

        return action

    def takeAction(self, action):

        new\_state = self.state

        if not self.end:

            if action == "left":

                new\_state = self.state - 1

            else:

                new\_state = self.state + 1

            if new\_state in [END\_0, END\_1]:

                self.end = True

        return new\_state

    def giveReward(self, state):

        if state == END\_0:

            return -1

        if state == END\_1:

            return 1

        return 0

    def reset(self):

        self.state = START

        self.end = False

        self.states = []

    def play(self, valueFunc, rounds=100):

        for \_ in range(rounds):

            self.reset()

            action = self.chooseAction()

            while not self.end:

                nxtState = self.takeAction(action)

                self.reward = self.giveReward(nxtState)

                valueFunc.updateZ(self.state)

                valueFunc.learn(self.state, nxtState, self.reward)

                self.state = nxtState

                action = self.chooseAction()

                if self.debug:

                    print("end at {} reward {}".format(self.state, self.reward))

# Main code

# Update the dimensions of actual\_state\_values to match the number of states

actual\_state\_values = np.arange(-20, 22, 2) / 20.0

actual\_state\_values[0] = actual\_state\_values[-1] = 0

alphas = np.linspace(0, 0.8, 6)

lambdas = np.linspace(0, 1, 5)

rounds = 50

plt.figure(figsize=[10, 6])

for lamb in lambdas:

    alpha\_errors = []

    for alpha in alphas:

        valueFunc = ValueFunctionTD(alpha=alpha, lmbda=lamb)

        rw = RWTD(debug=False)

        rw.play(valueFunc, rounds=rounds)

        rmse = np.sqrt(np.mean(np.power(valueFunc.weights - actual\_state\_values, 2)))

        print("lambda {} alpha {} rmse {}".format(lamb, alpha, rmse))

        alpha\_errors.append(rmse)

    plt.plot(alphas, alpha\_errors, label="lambda={}".format(lamb))

plt.xlabel("alpha", size=14)

plt.ylabel("RMS error", size=14)

plt.legend()

plt.show()

**OUTPUT:**

A screenshot of a computer

Description automatically generated

A graph with a line and numbers

Description automatically generated with medium confidence

**RESULT:**

Thus, the temporal difference method TD (0) and TD (lambda) has been implemented successfully.

|  |  |
| --- | --- |
| **EX NO 5** | **IMPLEMENTATION OF BALANCED CARTPOLE MODEL USING DEEP Q-NETWORK** |

**AIM:**

To implement a balanced cartpole model using Deep Q-Network.

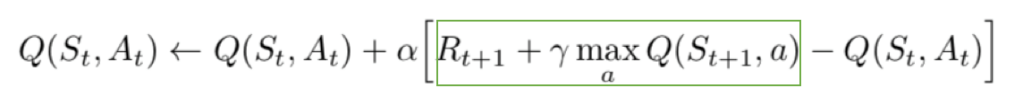
**INTRODUCTION:**

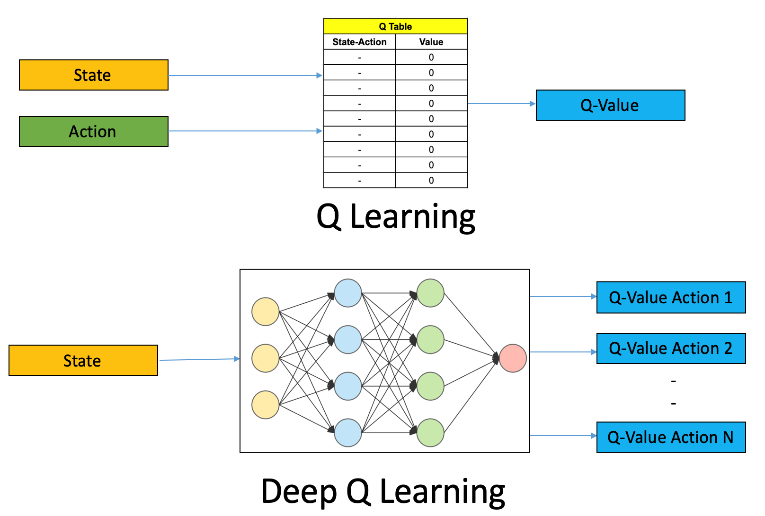
The steps involved in reinforcement learning using deep Q-learning networks (DQNs):

Step-1: All the past experience is stored by the user in memory.

Step-2: The next action is determined by the maximum output of the Q-network.

Step-3: The loss function here is mean squared error of the predicted Q-value and the target Q-value – Q\*. This is basically a regression problem. However, we do not know the target or actual value here as we are dealing with a reinforcement learning problem. Going back to the Q-value update equation derived from the Bellman equation. we have:





**ALGORITHM:**

**Step 1:** Import the necessary libraries for doing math, working with neural networks, simulating environments, and making plots.

**Step 2:** Create a simulated environment called "CartPole-v1" where an agent tries to balance a pole on a moving cart.

**Step 3:** Set up a neural network to help the agent make decisions. This network learns from its experiences.

**Step 4:** Decide the values of hyperparameters like future rewards (gamma), how often it explores new actions versus sticking with what it knows (epsilon), and how fast it learns (learning rate).

**Step 5:** Start a training loop, where the agent interacts with the environment to learn.

* In each episode of training (think of it like a game), the agent:
* Starts in an initial state (cart and pole positions).
* Repeatedly takes actions (move the cart left or right) and observes the results.
* Based on what it observes, it updates its knowledge (the neural network) to make better decisions in the future.
* Keeps track of the total reward it receives during the episode.

**Step 6:** Choose the action. Here the agent uses an "epsilon-greedy" strategy to decide what action to take.

**Step 7:** Update the neural network from the results of the above actions and then adjust the network to be better at estimating the correct values.

**Step 8:** To help with stability we create a target network copy the main network's knowledge. This helps the training process.

**Step 9:** As the agent learn more, it becomes less exploratory (chooses random actions less often) over time. This is controlled by “**epsilon**”.

**Step 10:** Evaluate and visualize the below code.

**IMPLEMENTATION:**

import numpy as np

import tensorflow as tf

import gym

import matplotlib.pyplot as plt

# Create the CartPole environment

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

# Define the Q-network

model = tf.keras.Sequential([

tf.keras.layers.Dense(24, activation='relu', input\_shape=(4,)),

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

tf.keras.layers.Dense(2, activation='linear') # 2 output nodes for left and right actions

])

# Define the optimizer and loss function

optimizer = tf.keras.optimizers.Adam(learning\_rate=0.001)

loss\_fn = tf.keras.losses.MeanSquaredError()

# Training parameters

gamma = 0.99

epsilon = 1.0

epsilon\_min = 0.01

epsilon\_decay = 0.995

batch\_size = 32

target\_update\_frequency = 100

# Initialize the target Q-network with the same weights as the main Q-network

target\_model = tf.keras.models.clone\_model(model)

target\_model.set\_weights(model.get\_weights())

# Function to select an action based on epsilon-greedy policy

def select\_action(state):

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

return env.action\_space.sample() # Explore by taking a random action

else:

q\_values = model.predict(state.reshape(1, -1))

return np.argmax(q\_values)

# Function to update the target Q-network

def update\_target\_model():

target\_model.set\_weights(model.get\_weights())

# Training loop

episodes = 10

rewards = []

# Lists to store episode rewards and their moving average

episode\_rewards = []

moving\_avg\_rewards = []

# Visualization: Initialize a figure for plotting rewards

plt.figure(figsize=(10, 5))

plt.xlabel('Episode')

plt.ylabel('Reward')

plt.title('CartPole DQN Training Progress')

for episode in range(episodes):

state = env.reset()

total\_reward = 0

while True:

action = select\_action(state)

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

target = model.predict(state.reshape(1, -1))

if done:

target[0][action] = reward

else:

target[0][action] = reward + gamma \* np.max(target\_model.predict(next\_state.reshape(1, -1)))

with tf.GradientTape() as tape:

q\_values = model(state.reshape(1, -1))

loss = loss\_fn(target, q\_values)

gradients = tape.gradient(loss, model.trainable\_variables)

optimizer.apply\_gradients(zip(gradients, model.trainable\_variables))

state = next\_state

total\_reward += reward

if done:

rewards.append(total\_reward)

episode\_rewards.append(total\_reward)

if epsilon > epsilon\_min:

epsilon \*= epsilon\_decay

if episode % target\_update\_frequency == 0:

update\_target\_model()

if episode % 10 == 0:

print(f"Episode {episode}, Average Reward: {np.mean(rewards[-10:])}")

# Visualization: Plot the rewards over episodes every 50 episodes

if episode % 50 == 0 and episode > 0:

moving\_avg = np.mean(episode\_rewards[-50:])

moving\_avg\_rewards.append(moving\_avg)

plt.plot(rewards, label='Episode Reward')

plt.plot(np.arange(len(moving\_avg\_rewards)) \* 50, moving\_avg\_rewards, label='Moving Average (50 episodes)')

plt.legend()

plt.pause(0.1)

break

# Final visualization: Plot the rewards and moving average

moving\_avg\_rewards = [np.mean(episode\_rewards[i-50:i+1]) if i >= 50 else np.mean(episode\_rewards[:i+1]) for i in range(len(episode\_rewards))]

plt.plot(rewards, label='Episode Reward')

plt.plot(np.arange(len(moving\_avg\_rewards)) \* 50, moving\_avg\_rewards, label='Moving Average (50 episodes)')

plt.legend()

plt.show()

# Evaluate the trained model

total\_rewards = []

for \_ in range(10):

state = env.reset()

episode\_reward = 0

while True:

action = np.argmax(model.predict(state.reshape(1, -1)))

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

episode\_reward += reward

state = next\_state

if done:

total\_rewards.append(episode\_reward)

break

print("Average Reward (Evaluation):", np.mean(total\_rewards))

**OUTPUT:**

A screenshot of a computer program

Description automatically generated

A graph showing a line

Description automatically generated

**RESULT:**

Thus, the balanced cartpole model using Deep Q-Network has been executed successfully.