KUMARAGURU
college of technology
character is life

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

LABORATORY RECORD

U18AII5203L-REINFORCEMENT LABORATORY

REGULATION: R18

YEAR/SEMESTER: III/V

PROGEAMME: B. TECH -ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

COURSE INCHARGE: Ms.Rupashini P R

Certificate

This is to certify that it is a Bonafide record of practical work done by Sri/Smt. _____ bearing the Roll No _____ of year Artificial Intelligence and Data science branch in the **Reinforcement Learning - U18AII5203** Laboratory during the academic year **2023-2024 (ODD)** under mysupervision.

Faculty in charge

KUMARAGURU college of technology
character is life

INTERNAL ASSESSMENT

Sl. No	Rubrics	Data Visualization (10)	Observation (10)	Total (20)	Viva (10)
1	Implementation of Markov Decision Process Using Value Iteration and Policy Iteration				
2	Implementation of Temporal difference Method Using TD (0) and TD(λ)				
3	Implementation of Q-Learning on Taxi Environment				
4	Implementation of Balanced Cartpole Model Using Deep Q-Network				
5	Implementation of Attari Games Using Dueling Deep Q Learning				
6	Implementation of PAC-MAN game Using Duel Deep Q Network				
7	Implementation of Model-Based Approach (DYNA) on Maze Environment				
8	Implementation of Actor-Critic Reinforcement Algorithm on Cartpole environment				
9	Implementation of PPO in Taxi Environment				
		Total			

KUMARAGURU
college of technology
character is life

Laboratory Mark

Internal Assessment	Viva (10)	Record (20)	Model Exam (30)	Total (60)

Staff in Charge

KUMARAGURU college of technology
character is life

IMPLEMENTATION OF MARKOV DECISION PROCESS

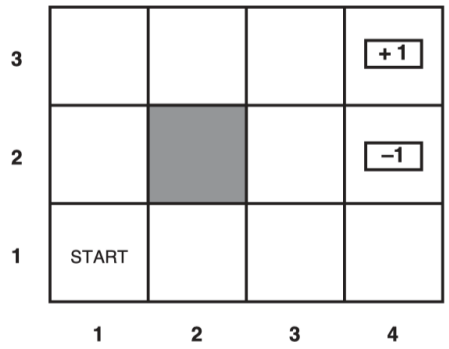
USING VALUE ITERATION AND POLICY ITERATION

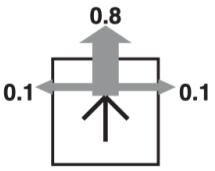
Aim :

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

Introduction :

Environment Design - Grid World





1. This is a **simple 4 x 3 environment**, and each block represents a state.
2. The agent can **move left, right, up, or down from a state.**
3. The "intended" outcome occurs with probability 0.8, but with probability 0.2 the agent moves at right angles to the intended direction.
4. **A collision with the wall or boundary results in no movement.** The two terminal states have reward +1 and -1, respectively, and all other states have a constant reward (e.g. of -0.01).
5. Also, a MDP usually has a discount factor $\gamma = 0.99$ that describes the preference of an agent for current rewards over future rewards.

Model Design - Markov Decision Process

A Markov decision process (MDP), by definition, is a sequential decision problem for a **fully observable, stochastic environment** with a **Markovian transition model** and additive rewards. It consists of a set of states, a set of actions, a transition model, and a reward function.

Value Iteration

Value iteration is an algorithm that gives an optimal policy for a MDP. It calculates the utility of each state, which is defined as the expected sum of discounted rewards from that state onward.

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U(s')$$

Bellman equation.

To solve this system of equations, value iteration uses an iterative approach that repeatedly updates the utility of each state (starting from zero) until an equilibrium is reached (converge). The iteration step, called a Bellman update, looks like this:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U_i(s')$$

Then, after the utilities of states are calculated, we can use them to select an optimal action for each state.

$$\pi^*(s) = \operatorname*{argmax}_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U(s')$$

Algorithm

1. Set up the initial environment.
2. Get the utility of the state reached by performing the given action from the given state.
3. Calculate the utility of a state given an action.
4. Get the optimal policy from U.
5. Choose the action that maximizes the utility.
6. Print the initial environment.
7. Get the optimal policy from U and print it.

Program

```
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

```
The initial U is:

| 0      | 0      | 0      | +1     |
| 0      | WALL   | 0      | -1     |
| 0      | 0      | 0      | 0      |

During the value iteration:

| -0.01 | -0.01 | 0.782 | +1    |
| -0.01 | WALL  | -0.01 | -1    |
| -0.01 | -0.01 | -0.01 | -0.01 |

| -0.01 | 0.607 | 0.858 | +1    |
| -0.01 | WALL  | 0.509 | -1    |
| -0.01 | -0.01 | -0.01 | -0.01 |

| 0.467 | 0.790 | 0.917 | +1    |
| -0.02 | WALL  | 0.621 | -1    |
| -0.02 | -0.02 | 0.389 | -0.02 |

| 0.659 | 0.873 | 0.934 | +1    |
| 0.354 | WALL  | 0.679 | -1    |
| -0.03 | 0.292 | 0.476 | 0.196 |

| 0.781 | 0.902 | 0.941 | +1    |
| 0.582 | WALL  | 0.698 | -1    |
| 0.295 | 0.425 | 0.576 | 0.287 |

| 0.840 | 0.914 | 0.944 | +1    |
| 0.724 | WALL  | 0.705 | -1    |
| 0.522 | 0.530 | 0.613 | 0.375 |

| 0.869 | 0.919 | 0.945 | +1    |
| 0.798 | WALL  | 0.708 | -1    |
| 0.667 | 0.580 | 0.638 | 0.414 |

| 0.883 | 0.920 | 0.945 | +1    |
| 0.836 | WALL  | 0.709 | -1    |
| 0.746 | 0.634 | 0.649 | 0.437 |

| 0.889 | 0.921 | 0.945 | +1    |
| 0.854 | WALL  | 0.710 | -1    |
| 0.789 | 0.706 | 0.658 | 0.449 |

| 0.892 | 0.921 | 0.945 | +1    |
| 0.863 | WALL  | 0.711 | -1    |
| 0.815 | 0.754 | 0.685 | 0.456 |

| 0.893 | 0.921 | 0.946 | +1    |
| 0.867 | WALL  | 0.714 | -1    |
| 0.829 | 0.785 | 0.726 | 0.478 |

| 0.894 | 0.921 | 0.946 | +1    |
| 0.869 | WALL  | 0.721 | -1    |
| 0.837 | 0.802 | 0.754 | 0.513 |

| 0.894 | 0.922 | 0.947 | +1    |
| 0.870 | WALL  | 0.730 | -1    |
| 0.841 | 0.811 | 0.771 | 0.539 |

| 0.895 | 0.922 | 0.948 | +1    |
| 0.870 | WALL  | 0.738 | -1    |
| 0.843 | 0.816 | 0.781 | 0.555 |

| 0.895 | 0.923 | 0.948 | +1    |
| 0.871 | WALL  | 0.746 | -1    |
| 0.844 | 0.819 | 0.787 | 0.565 |

| 0.896 | 0.924 | 0.949 | +1    |
| 0.871 | WALL  | 0.752 | -1    |
| 0.844 | 0.820 | 0.790 | 0.571 |

| 0.897 | 0.925 | 0.950 | +1    |
| 0.872 | WALL  | 0.758 | -1    |
| 0.845 | 0.821 | 0.792 | 0.577 |

| 0.898 | 0.926 | 0.951 | +1    |
| 0.873 | WALL  | 0.763 | -1    |
| 0.846 | 0.822 | 0.794 | 0.583 |

The optimal policy is:

| Right | Right | Right | +1    |
| Up    | WALL  | Left  | -1    |
| Up    | Left  | Left  | Down  |
```

Policy Iteration

Policy iteration is another algorithm that solves MDPs. It starts with a random policy and alternates the following two steps until the policy improvement step yields no change:
(1) Policy evaluation: given a policy, calculate the utility U(s) of each state s if the policy is executed;
(2) Policy improvement: update the policy based on U(s).

$$\pi^*(s) = \underset{a \in A(s)}{\operatorname{argmax}} \sum_{s'} P(s' \mid s, a) U(s')$$

For the policy evaluation step, we use a simplified version of the Bellman equation to calculate the utility of each state.

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s)) U_i(s')$$

To make the algorithm more efficient, we can perform a number of simplified Bellman updates to get an approximation of the utilities instead of calculating the exact solutions.

$$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s)) U_i(s')$$

Algorithm:

1. Set up the initial environment.
2. Construct a random policy.
3. Get the utility of the state reached by performing the given action from the given state.
4. Calculate the utility of a state given an action.
5. Perform some simplified value iteration steps to get an approximation of the utilities.
6. Get the action that maximizes the utility.
7. Print the initial environment.
8. Print the optimal policy.

Program

```
import random
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]]
policy = [[random.randint(0, 3) for j in range(NUM_COL)] for i in range(NUM_ROW)]
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:
                val = ["Down", "Left", "Up", "Right"][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 policyEvaluation(policy, U):
    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] = calculateU(U, r, c, policy[r][c])
                error = max(error, abs(nextU[r][c]-U[r][c]))
        U = nextU
        if error < MAX_ERROR * (1-DISCOUNT) / DISCOUNT:
            break
    return U
def policyIteration(policy, U):
    print("During the policy iteration:\n")
    while True:
        U = policyEvaluation(policy, U)
        unchanged = True
        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
                if maxU > calculateU(U, r, c, policy[r][c]):
                    policy[r][c] = maxAction
                    unchanged = False
        if unchanged:

            break
        printEnvironment(policy)
    return policy
print("The initial random policy is:\n")
printEnvironment(policy)
policy = policyIteration(policy, U)
print("The optimal policy is:\n")
printEnvironment(policy)
```

Output:

```
The initial random policy is:

| Left  | Up    | Left  | +1    |
| Left  | WALL  | Right | -1    |
| Right | Down  | Right | Down  |

During the policy iteration:

| Up    | Left  | Right | +1    |
| Up    | WALL  | Up    | -1    |
| Up    | Left  | Left  | Down  |

| Up    | Right | Right | +1    |
| Up    | WALL  | Up    | -1    |
| Up    | Right | Up    | Down  |

| Right | Right | Right | +1    |
| Up    | WALL  | Left  | -1    |
| Up    | Left  | Up    | Down  |

| Right | Right | Right | +1    |
| Up    | WALL  | Left  | -1    |
| Up    | Left  | Left  | Down  |

The optimal policy is:

| Right | Right | Right | +1    |
| Up    | WALL  | Left  | -1    |
| Up    | Left  | Left  | Down  |
```

Result:

Thus, the Markov Decision Process using Value Iteration and Policy Iteration on gridworld environment was implemented successfully.

KUMARAGURU college of technology
character is life

IMPLEMENTATION OF TEMPORAL DIFFERENCE METHOD USING TD (0) AND TD (λ)

Aim :

To solve Random walk using Temporal Difference algorithms TD (0) and TD (λ)

Introduction:

Environment Design - Random walk

Random Walk involves walking on the random states to reach the termination state. The agent may always move in the intended direction due to the nature of the random walk and tries to reach the termination states with maximum states.

- Total No. of States = 21 including termination states
- Agent now starts from 9th state try to reach anyone of the termination state 1st or 21st.
- All states have zero reward, except the rightmost that has reward +1, the leftmost that hasrewards -1
- Random walk with equal probability to each side.
- Each episode starts at 9th state and discount factor = 1

Model Design - Temporal Difference

- TD learning is a technique to predict a variable's expected value in a sequence of states.
- TD methods learn directly from episodes of experience.
- TD is model-free: no knowledge of MDP transitions / rewards.
- TD learns from incomplete episodes, by bootstrapping.
- TD updates a guess towards a guess.
- TD tries to predict the combination of immediate reward and its own reward prediction at the next moment in time.
- The **TD Error** is the difference between the ultimate correct reward (V*t) and our current prediction (Vt). The current value will be updated by its value + learning rate * error:

$$V_t \leftarrow V_t + \alpha * E_t = V_t + \alpha * (r_{t+1} + \gamma * V_{t+1} - V_t)$$

- Temporal Difference algorithms: TD (0), TD(λ).
- Like the Monte Carlo method, it doesn't require model dynamics.
- Like dynamic programming, it doesn't need to wait until the end of the episode to make an estimate of the value function.

Hyperparameters:

Gamma (γ): the discount rate. A value between 0 and 1. The higher the value the less you are discounting.
Lambda (λ): the credit assignment variable. A value between 0 and 1. The higher the value the more credit you can assign to further back states and actions.
Alpha (α): t**he learning rate**. How much of the error should we accept and therefore adjust our estimates towards. A value between 0 and 1. A higher value adjusts aggressively, accepting more of the error while a smaller one adjusts conservatively but may make more conservative moves towards the actual values.
Delta (δ): a change or difference in value.
***e* ("e-greedy" policy):** the ratio reflective of exploration vs. exploitation. We explore new options with probability *e* and stay at the current max with probability *1-e*. The larger *e* implies more exploration while training.

TD(0) Algorithm

Instead of using the accumulated sum of discounted rewards (Gt) we will only look at the immediate reward (Rt+1), plus the discount of the estimated value of **only 1 step ahead** (V(St+1)) TD(0) can learn environments that do not have terminal states where TD(1) cannot.

$$V(S_t) \leftarrow V(S_t) + \alpha \left(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right)$$

TD(0) Update Value toward Estimated Return

TD(λ) Algorithm

TD(λ) updates before the end of an episode (TD(1)) and use more than a 1 step ahead (TD(0)) for our estimation.

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

$$V(s) \leftarrow V(s) + \alpha \delta_t E_t(s)$$

Update with Respect to TD Error (δt) and Eligibility (Et(s))

Algorithm

1. Set up the initial environment.
2. Create the Random Walk environment using NumPy.
3. Define the random policy, which returns the random action by sampling from the action space.
4. Define the dictionary for storing the value of states, and we initialize the value of all the states to 0.
5. Initialize the discount factor γ and the learning rate α.
6. Set the number of episodes and the number of time steps in each episode.
7. Compute the values of the states.
   * Initialize the utility of the state arbitrarily and keep an eligibility trace for every state as 0.
   * Loop the below steps over episodes
   * For each step of episode
   * Get the utility of the state reached by performing the given action from the given state.
   * Update value V(s) for every state until the terminal state is reached.
8. Print the value for each state.

Program:

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

NUM_STATES = 19
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 + 2) self.z =
      np.zeros(NUM_STATES + 2)
      self.alpha = alpha
      self.gamma =
      gammaself.lmbda =
      lmbda
   def value(self, state):
      v =
      self.weights[state]
      return v
   def updateZ(self,
      state):dev = 1
      self.z *= self.gamma *
      self.lmbdaself.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.zclass RWTD:
```

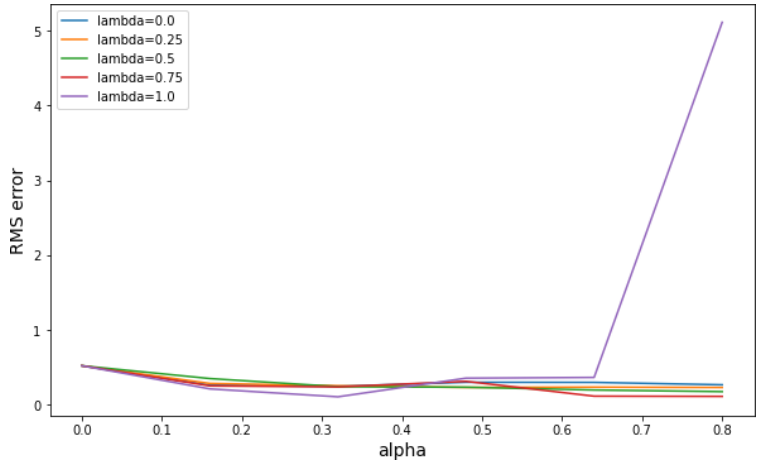
```
    def __init_(self, start=START, end=False,
        debug=False):self.actions = ["left", "right"]
        self.state = start
        self.end = end
        self.reward = 0
        self.debug = debug
    def chooseAction(self):
        action = np.random.choice(self.actions)
        return action
    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 =
        STARTself.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))

if __name_  == "_main__":
    actual_state_values = np.arange(-20, 22, 2) /
    20.0 actual_state_values[0] =
    actual_state_values[-1] = 0actual_state_values
```

```
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_erros =  []
   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_erros.append(rmse)
   plt.plot(alphas, alpha_erros, label="lambda={}".format(lamb))

plt.xlabel("alpha", size=14)
plt.ylabel("RMS error",
size=14)plt.legend()
```

Output:

lambda 0.0 alpha 0.0 rmse 0.5209880722517277
lambda 0.0 alpha 0.16 rmse 0.2630192653781778
lambda 0.0 alpha 0.32 rmse 0.24618118217371632
lambda 0.0 alpha 0.48 rmse 0.30052518175872645
lambda 0.0 alpha 0.64 rmse 0.2987712054664889
lambda 0.0 alpha 0.8 rmse 0.26692169147017286
lambda 0.25 alpha 0.0 rmse 0.5209880722517277
lambda 0.25 alpha 0.16 rmse 0.2849885991423671
lambda 0.25 alpha 0.32 rmse 0.25368028236015905
lambda 0.25 alpha 0.48 rmse 0.23153718622222796
lambda 0.25 alpha 0.64 rmse 0.23577623890236016
lambda 0.25 alpha 0.8 rmse 0.23064186442539267
lambda 0.5 alpha 0.0 rmse 0.5209880722517277
lambda 0.5 alpha 0.16 rmse 0.34990596261018525
lambda 0.5 alpha 0.32 rmse 0.24192049974029942
lambda 0.5 alpha 0.48 rmse 0.23442812033251875
lambda 0.5 alpha 0.64 rmse 0.1971682872612908
lambda 0.5 alpha 0.8 rmse 0.17534845528001267
lambda 0.75 alpha 0.0 rmse 0.5209880722517277
lambda 0.75 alpha 0.16 rmse 0.2521672191876034
lambda 0.75 alpha 0.32 rmse 0.2387612568659104
lambda 0.75 alpha 0.48 rmse 0.31260949752507755
lambda 0.75 alpha 0.64 rmse 0.11474594047202488
lambda 0.75 alpha 0.8 rmse 0.11168839302459821
lambda 1.0 alpha 0.0 rmse 0.5209880722517277
lambda 1.0 alpha 0.16 rmse 0.2115593262983959
lambda 1.0 alpha 0.32 rmse 0.10612222921063669
lambda 1.0 alpha 0.48 rmse 0.355730688852782
lambda 1.0 alpha 0.64 rmse 0.36507027252215335
lambda 1.0 alpha 0.8 rmse 5.1116828511554955

Output Screenshot



Result:

Thus, the Temporal Difference on Random Walk Environment using TD (0) and TD(λ) has been implemented successfully.

KUMARAGURU college of technology
character is life

IMPLEMENTATION OF Q-LEARNING ON TAXI ENVIRONMENT

Aim:

To implement Q-learning on Taxi - v3 Environment

Introduction:

Environment Design: Taxi Environment

Description

There are four designated locations in the grid world indicated by R(ed), G(reen), Y(ellow), and B(lue). When the episode starts, the taxi starts off at a random square and the passenger is at a random location. The taxi drives to the passenger's location, picks up the passenger, drives to the passenger's destination (another one of the four specified locations), and then drops off the passenger. Once the passenger is dropped off, the episode ends.

Actions Space

There are 6 discrete deterministic actions:

- 0: move south
- 1: move north
- 2: move east
- 3: move west
- 4: pickup passenger
- 5: drop off passenger

Action Space	Discrete(6)
Observation Space	Discrete(500)
Import	`gym.make("Taxi-v3")`

Observations Space

There are 500 discrete states since there are 25 taxi positions, 5 possible locations of the passenger (including the case when the passenger is in the taxi), and 4 destination locations.
Passenger locations:

- 0: R(ed)
- 1: G(reen)
- 2: Y(ellow)
- 3: B(lue)
- 4: in taxi

Destinations:

- 0: R(ed)
- 1: G(reen)
- 2: Y(ellow)
- 3: B(lue)

Rewards

- -1 per step unless other reward is triggered.
- +20 delivering passenger.
- -10 executing "pickup" and "drop-off" actions illegally.

Model Design: Q Learning

Q-learning is an **off-policy reinforcement learning algorithm** that seeks to **find the best action to take given the current state.** It's considered off-policy because the q-learning function learns from actions that are outside the current policy, like taking random actions, and therefore a policy isn't needed. More specifically, q-learning seeks to learn a policy that maximizes the total reward.

- Q-Learning is a value-based reinforcement learning algorithm which is used to find the optimal action-selection policy using a Q function.
- Our goal is to maximize the value function Q.
- The Q table helps us to find the best action for each state.
- It helps to maximize the expected reward by selecting the best of all possible actions.
- Q (state, action) returns the expected future reward of that action at that state.
- This function can be estimated using Q-Learning, which iteratively updates Q(s,a) using the **Bellman equation.**
- Initially we explore the environment and update the Q-Table. When the Q-Table is ready, the agent will start to exploit the environment and start taking better actions.

Q-values are initialized to an arbitrary value, and as the agent exposes itself to the environment and receives different rewards by executing different actions, the Q-values are updated using the equation:

$$Q(state, action) \leftarrow (1-\alpha)Q(state, action) + \alpha\left(reward + \gamma \max_a Q(next\ state, all\ actions)\right)$$

Where:

- α (alpha) is the learning rate ($0 < \alpha \leq 1$) - Just like in supervised learning settings, α is the extent to which our Q-values are being updated in every iteration.

- γ (gamma) is the discount factor ($0 \leq \gamma \leq 1$) - determines how much importance we want to give to future rewards. A high value for the discount factor (close to **1**) captures the long-term effective award, whereas, a discount factor of **0** makes our agent consider only immediate reward, hence making it greedy.

Algorithm

1. Set up the initial environment.
2. Create the Taxi-V3 environment using Gym.
3. Initialize the discount factor γ and the learning rate α, epsilon.
4. Compute the values of the states.
5. Q-learning Steps to be followed.
   * Initialize the Q-table by all zeros.
   * Start exploring actions: For each state, select any one among all possible actions for thecurrent state (S).
   * Travel to the next state (S') because of that action (a).
   * For all possible actions from the state (S') select the one with the highest Q-value.
   * Update Q-table values using the equation.
   * Set the next state as the current state.
     If the goal state is reached, then end and repeat the process.
6. Print the value for each state.

Program:

```
import gym
env = gym.make("Taxi-v3").env
import numpy as np
q_table = np.zeros([env.observation_space.n, env.action_space.n])
import random
from IPython.display import clear_output
alpha = 0.1
gamma = 0.6
epsilon = 0.1
all_epochs = []
all_penalties = []
for i in range(1, 100001):
    state = env.reset()
    epochs, penalties, reward, = 0, 0, 0
    done = False
    while not done:
        if random.uniform(0, 1) < epsilon:
            action = env.action_space.sample()
        else:
            action =  np.argmax(q_table[state])
        next_state, reward, done, info = env.step(action)

        old_value = q_table[state, action]
        next_max = np.max(q_table[next_state])
```

```
        new_value = (1 - alpha) * old_value + alpha * (reward + gamma *
        next_max)q_table[state, action] = new_value
        if reward == -
           10:penalties
           += 1 state =
             next_state
        epochs += 1
    if i % 100 == 0:
        clear_output(wait=Tr
        ue)print(f"Episode:
        {i}")

print("Training
finished.\n")q_table[328]
total_epochs, total_penalties = 0, 0
episodes = 100
for _ in
    range(episodes):
    state = env.reset()
    epochs, penalties, reward = 0,
    0, 0done = False
    while not done:
        action = np.argmax(q_table[state])
        state, reward, done, info =
        env.step(action)if reward == -10:
            penalties += 1
        epochs += 1
    total_penalties +=
    penaltiestotal_epochs
    += epochs

print(f"Results after {episodes} episodes:")
print(f"Average timesteps per episode: {total_epochs / episodes}")
print(f"Average penalties per episode: {total_penalties / episodes}")
```

Output:

Results after 100 episodes:
Average timesteps per episode: 12.3
Average penalties per episode: 0.0

Result:

Thus, the Q-learning on Taxi Environment is implemented and executed successfully.

KUMARAGURU college of technology
character is life

IMPLEMENTATION OF BALANCED CARTPOLE MODEL USING DEEP Q-NETWORK

Aim:

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

Introduction:

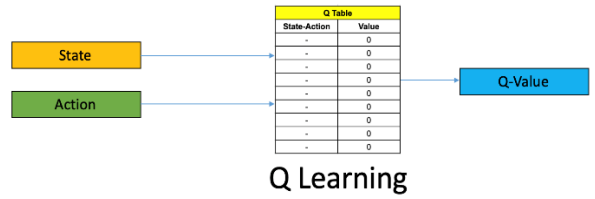
The steps involved in reinforcement learning using deep Q-learning networks (DQNs):

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

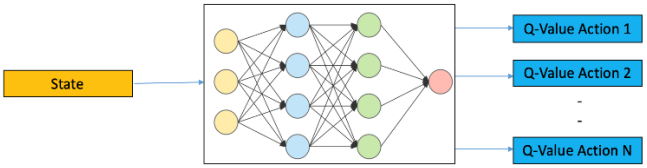
Step-2: The next action is determined by the maximum output of the Q-network.

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:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$



Q Learning



Deep Q Learning

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.
- Keep 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 learns 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.

Program:

```
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:
```

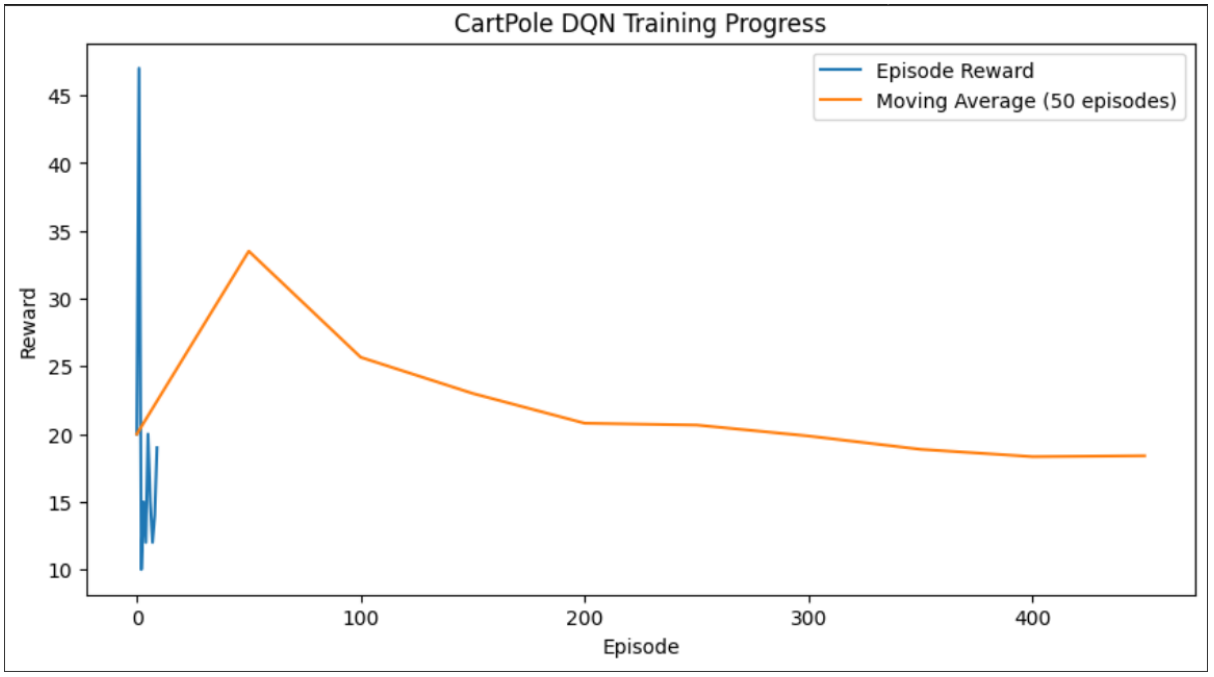
```
        q_values = model.predict(state.reshape(1, -1))
        return np.argmax(q_values)
# Function to update the target Q-network
def update_target_model():
    target_model.set_weights(model.get_weights())
# Training loop
episodes = 10
rewards = []
# Lists to store episode rewards and their moving 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:

```
1/1 [==============================] - 0s 54ms/step
1/1 [==============================] - 0s 84ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 30ms/step
1/1 [==============================] - 0s 30ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 33ms/step
1/1 [==============================] - 0s 36ms/step
1/1 [==============================] - 0s 30ms/step
1/1 [==============================] - 0s 30ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 30ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 38ms/step
1/1 [==============================] - 0s 37ms/step
```



Result:

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

KUMARAGURU college of technology
character is life

IMPLEMENTATION OF ATTARI GAMES USING DUEL DEEP Q-LEARNING

Aim:

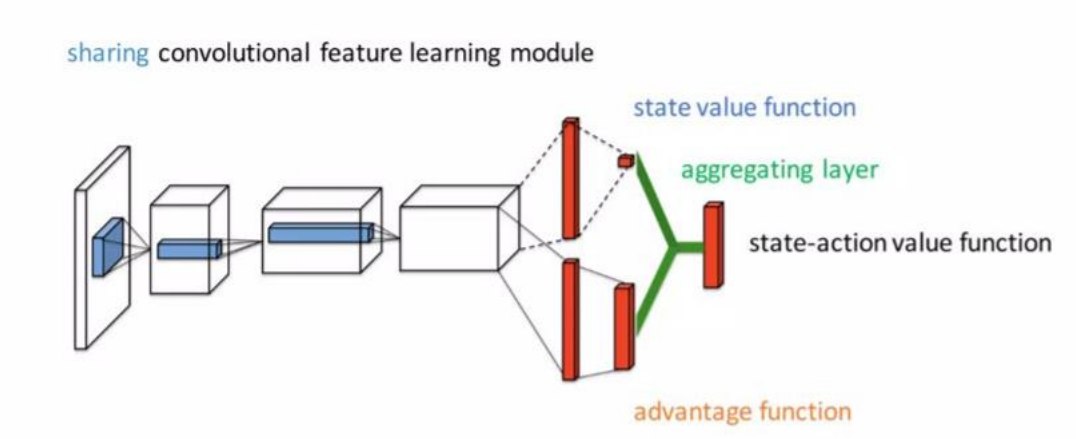
To Implement the Duel Deep Q-learning (Attari Games) using Open Gym module in python.

Introduction:

Dueling Deep Q-Learning (Dueling DQN) is an extension of the Deep Q-Network (DQN) algorithm that enhances the efficiency of Q-learning by separating the value and advantage components of the Q-function.

Dueling DQN employs a neural network architecture designed to separately estimate the value and advantage functions. The network has two streams: one for the value and one for the advantage. These streams share the initial layers of the network and then diverge into separate output layers for V and A.

$$Q(s,a) = V(s) + (A(s,a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a))$$



Algorithm:

Step 1: Create the game environment using OpenAI Gym

Step 2: Design a neural network with separate streams for value and advantage functions.

Step 3: Create a buffer to store experiences for training stability.

Step 4: Interact with the environment, sample experiences, and update the Q-network.

Step 5: Periodically update a target network to stabilize training.

Step 6: Test the trained agent's performance and save the model for future use.

Program:

```
 !pip install gym tensorflow numpy
import gym
import random
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Subtract, Lambda
from tensorflow.keras.optimizers import Adam
# Environment
env = gym.make('CartPole-v1')
state_size = env.observation_space.shape[0]
action_size = env.action_space.n
# Hyperparameters
learning_rate = 0.001
gamma = 0.99
epsilon = 1.0
epsilon_min = 0.01
epsilon_decay = 0.995
memory_size = 10000
batch_size = 64

# Replay memory
memory = []
def build_model():
    input_layer = Input(shape=(state_size,))
    fc1 = Dense(24, activation='relu')(input_layer)
    fc2 = Dense(24, activation='relu')(fc1)
        # Value stream
    value_stream = Dense(1)(fc2)
        # Advantage stream
    advantage_stream = Dense(action_size)(fc2)
        # Combine value and advantage streams
    combined = Subtract()([advantage_stream, Lambda(lambda x: tf.reduce_mean(x, axis=1,
keepdims=True))(advantage_stream)])
        # Q-value output
    q_values = Lambda(lambda x: x[0] + x[1])([value_stream, combined])

    model = Model(inputs=input_layer, outputs=q_values)
```

```
    model.compile(loss='mse', optimizer=Adam(learning_rate=learning_rate))
    return model
model = build_model()
for episode in range(1000):
    state = env.reset()
    state = np.reshape(state, [1, state_size])
    total_reward = 0
    for time_step in range(500):
        if np.random.rand() <= epsilon:
            action = env.action_space.sample()
        else:
            q_values = model.predict(state)
            action = np.argmax(q_values[0])
        next_state, reward, done, _ = env.step(action)
        next_state = np.reshape(next_state, [1, state_size])
        memory.append((state, action, reward, next_state, done))
        state = next_state
        if done:
            print(f"Episode: {episode + 1}, Score: {time_step + 1}")
            break

        if len(memory) > batch_size:
            minibatch = random.sample(memory, batch_size)
            for state, action, reward, next_state, done in minibatch:
                target = reward
                if not done:
                    target = reward + gamma * np.amax(model.predict(next_state)[0])
                target_f = model.predict(state)
                target_f[0][action] = target
                model.fit(state, target_f, epochs=1, verbose=0)

            if epsilon > epsilon_min:
                epsilon *= epsilon_decay
```

Output:

```
Episode: 1, Score: 15
Episode: 2, Score: 50
1/1 [==============================] - 0s 208ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 37ms/step
1/1 [==============================] - 0s 94ms/step
1/1 [==============================] - 0s 88ms/step
1/1 [==============================] - 0s 58ms/step
1/1 [==============================] - 0s 44ms/step
1/1 [==============================] - 0s 94ms/step
1/1 [==============================] - 0s 37ms/step
1/1 [==============================] - 0s 36ms/step
1/1 [==============================] - 0s 83ms/step
1/1 [==============================] - 0s 68ms/step
1/1 [==============================] - 0s 54ms/step
1/1 [==============================] - 0s 51ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 55ms/step
1/1 [==============================] - 0s 38ms/step
1/1 [==============================] - 0s 39ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 42ms/step
1/1 [==============================] - 0s 39ms/step
1/1 [==============================] - 0s 39ms/step
```

Result:

Thus, the Dual Deep Q-learning using Open GYM environment for Attari Game is implemented successfully.

KUMARAGURU college of technology
character is life

IMPLEMENTATION OF PAC-MAN USING DUEL DEEP Q-NETWORK

Aim:

Implementation of PAC-MAN game using Duel Deep Q- Network in python.

Introduction:

Pac-Man using Duel DQN" is an advanced AI technique that trains Pac-Man to play strategically through deep reinforcement learning. It excels in complex game environments, enhancing decision-making using a unique architecture and double Q-learning. Implementing this algorithm requires expertise in reinforcement learning and deep learning, showcasing AI's role in classic gaming.

Algorithm:

Step-1: Initialization:

- Set up Pac-Man environment.
- Create online and target Duel DQN networks.
- Initialize a replay buffer.

Step-2: Training Loop:

Repeat:

- Choose an action with epsilon-greedy strategy.
- Execute the action, observe next state, and collect reward.
- Store the experience in the replay buffer.
- Sample a mini batch from the buffer.
- Compute target Q-values using the target network (Double DQN).
- Update the online network using backpropagation.
- Periodically update the target network's weights.
- Decrease exploration epsilon over time.

Step-3: Evaluation: Periodically assess agent's performance without exploration.

Step-4: Repeat: Continue training and evaluation until satisfactory performance or convergence.

Program:

```
import random
class PacManEnvironment:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.pacman_x = 1
        self.pacman_y = 1
        self.ghost_x = width - 2
        self.ghost_y = height - 2
        self.food_x = random.randint(1, width - 2)
        self.food_y = random.randint(1, height - 2)
        self.score = 0
        self.done = False

    def reset(self):
        self.pacman_x = 1
        self.pacman_y = 1
        self.ghost_x = self.width - 2
        self.ghost_y = self.height - 2
        self.food_x = random.randint(1, self.width - 2)
        self.food_y = random.randint(1, self.height - 2)
        self.score = 0
        self.done = False

    def get_state(self):
        return (self.pacman_x, self.pacman_y, self.ghost_x, self.ghost_y, self.food_x,
self.food_y)

    def take_action(self, action):
        if action == 0:  # Move up
            self.pacman_y -= 1
        elif action == 1:  # Move down
            self.pacman_y += 1
        elif action == 2:  # Move left
            self.pacman_x -= 1
        elif action == 3:  # Move right
            self.pacman_x += 1

        self.pacman_x = max(1, min(self.pacman_x, self.width - 2))
        self.pacman_y = max(1, min(self.pacman_y, self.height - 2))

        if self.pacman_x == self.ghost_x and self.pacman_y == self.ghost_y:
            self.score -= 10
            self.done = True

        if self.pacman_x == self.food_x and self.pacman_y == self.food_y:
            self.score += 10
            self.food_x = random.randint(1, self.width - 2)
            self.food_y = random.randint(1, self.height - 2)

    def is_done(self):
```

```
        return self.done
class DuelingDQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.epsilon = 0.1
        self.q_table = {}

    def select_action(self, state):
        if random.uniform(0, 1) < self.epsilon:
            return random.randint(0, self.action_size - 1)
        return max(range(self.action_size), key=lambda x: self.q_table.get((state, x), 0))

    def update_q_table(self, state, action, reward, next_state):
        best_next_action = max(range(self.action_size), key=lambda x:
self.q_table.get((next_state, x), 0))
        old_value = self.q_table.get((state, action), 0)
        next_value = self.q_table.get((next_state, best_next_action), 0)
        new_value = (1 - 0.1) * old_value + 0.1 * (reward + next_value)
        self.q_table[(state, action)] = new_value

# Main training loop
if __name__ == "__main__":
    width = 5
    height = 5
    state_size = 6
    action_size = 4
    env = PacManEnvironment(width, height)
    agent = DuelingDQNAgent(state_size, action_size)

    episodes = 1000
    for episode in range(episodes):
        env.reset()
        total_reward = 0
        while not env.is_done():
            state = env.get_state()
            action = agent.select_action(state)
            env.take_action(action)
            next_state = env.get_state()
            reward = env.score
            agent.update_q_table(state, action, reward, next_state)
            total_reward += reward

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

# Dueling DQN Agent
class DuelingDQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.epsilon = 0.1
        self.q_table = {}
```

```
    def select_action(self, state):
        if random.uniform(0, 1) < self.epsilon:
            return random.randint(0, self.action_size - 1)
        return max(range(self.action_size), key=lambda x: self.q_table.get((state, x), 0))

    def update_q_table(self, state, action, reward, next_state):
        best_next_action = max(range(self.action_size), key=lambda x:
self.q_table.get((next_state, x), 0))
        old_value = self.q_table.get((state, action), 0)
        next_value = self.q_table.get((next_state, best_next_action), 0)
        new_value = (1 - 0.1) * old_value + 0.1 * (reward + next_value)
        self.q_table[(state, action)] = new_value

# Main training loop
if __name__ == "__main__":
    width = 5
    height = 5
    state_size = 6
    action_size = 4
    env = PacManEnvironment(width, height)
    agent = DuelingDQNAgent(state_size, action_size)

    episodes = 1000
    for episode in range(episodes):
        env.reset()
        total_reward = 0
        while not env.is_done():
            state = env.get_state()
            action = agent.select_action(state)
            env.take_action(action)
            next_state = env.get_state()
            reward = env.score
            agent.update_q_table(state, action, reward, next_state)
            total_reward += reward

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

Output:

```
Episode: 1, Total Reward: 3702470
Episode: 2, Total Reward: 970
Episode: 3, Total Reward: 1060
Episode: 4, Total Reward: 34610
Episode: 5, Total Reward: 0
Episode: 6, Total Reward: -10
Episode: 7, Total Reward: -10
Episode: 8, Total Reward: 337250
Episode: 9, Total Reward: -10
Episode: 10, Total Reward: 109700
Episode: 11, Total Reward: 303950
Episode: 12, Total Reward: 28110
Episode: 13, Total Reward: 2200
Episode: 14, Total Reward: 179940
Episode: 15, Total Reward: 195400
Episode: 16, Total Reward: 6490
Episode: 17, Total Reward: 2944430
Episode: 18, Total Reward: 531940
Episode: 19, Total Reward: 2113930
Episode: 20, Total Reward: 270
Episode: 21, Total Reward: 10
Episode: 22, Total Reward: 122810
Episode: 23, Total Reward: 10
Episode: 24, Total Reward: 1510680
Episode: 25, Total Reward: 3002940
Episode: 26, Total Reward: 1051250
Episode: 27, Total Reward: 1711390
Episode: 28, Total Reward: 0
Episode: 29, Total Reward: 177950
```

Result:

Thus, the PAC-MAN game has been successfully implemented using the Duel Deep Q-network.

KUMARAGURU college of technology
character is life

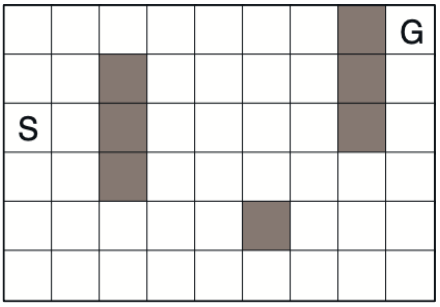
IMPLEMENTATION OF MODEL-BASED APPROACH (DYNA) ON MAZE ENVIRONMENT

Aim:

To implement the Model Based approach (DYNA - Q) on Maze environment.

Introduction:

Design: Simple Maze Environment

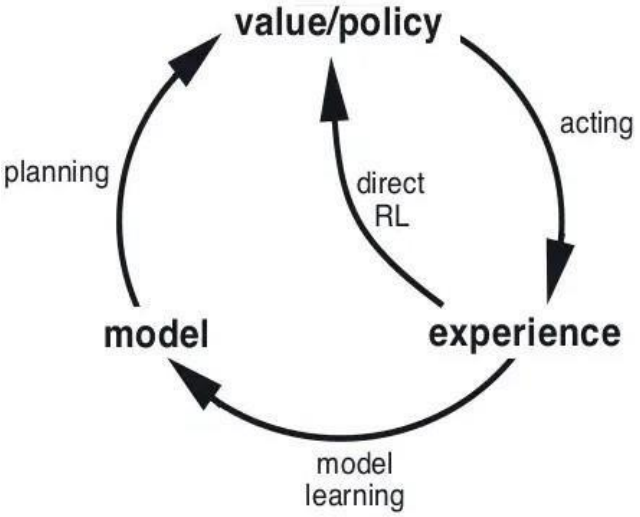


- In this maze environment, the goal is to reach the goal state (G) as fast as possible from the starting state (S).
- There are four actions – up, down, right, left – which take the agent deterministically from a state to the corresponding neighboring states 47 states, 4 actions, deterministic dynamics Obstacles and walls.
- Rewards are 0 except +1 for transition into goal state γ= 0.95, discounted episodic task
- Agent parameters:
- $\alpha = 0.1$, $\varepsilon = 0.1$
- Initial action-values were all zero.

Model Design - DYNA - Q

Dyna-Q involves four basic steps:

1. Action selection: given an observation, select an action to be performed (here, using the ϵ -greedy method).
2. Direct RL: using the observed next state and reward, update the action values (here, using one-step tabular Q-learning).
3. Model learning: using the observed next state and reward, update the model (here, updating a table as the environment is assumed to be deterministic).
4. Planning: update the action values by generating n simulated experiences using certain starting states and actions (here, using the random-sample one-step tabular Q-planning method). This is also known as the 'Indirect RL' step.
5. The process of choosing the state and action to simulate an experience is known as 'search control'.



Algorithm:

1. Set up the initial environment.
2. Create the maze environment using NumPy.
3. Initialize the discount factor γ and the learning rate α, epsilon.
4. Initialize the Q values and model(s,a) for all states and their actions.
5. Compute the values of the states.
   * Initialize the Q-table by all zeros.
   * Start exploring actions using epsilon greedy: For each state, select any one among all possible actions for the current state (S).
   * Travel to the next state (S') because of that action (a)
   * Update Q values using the equation.
   *Now predict the next state and action using the model

$$Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$$

6. If the goal state is reached, then end and repeat the process.
7. Print the value for each state.

$Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
Repeat n times:
  $S \leftarrow$ random previously observed state
  $A \leftarrow$ random action previously taken in S
  $R, S' \leftarrow Model(S,A)$
  $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$

Program

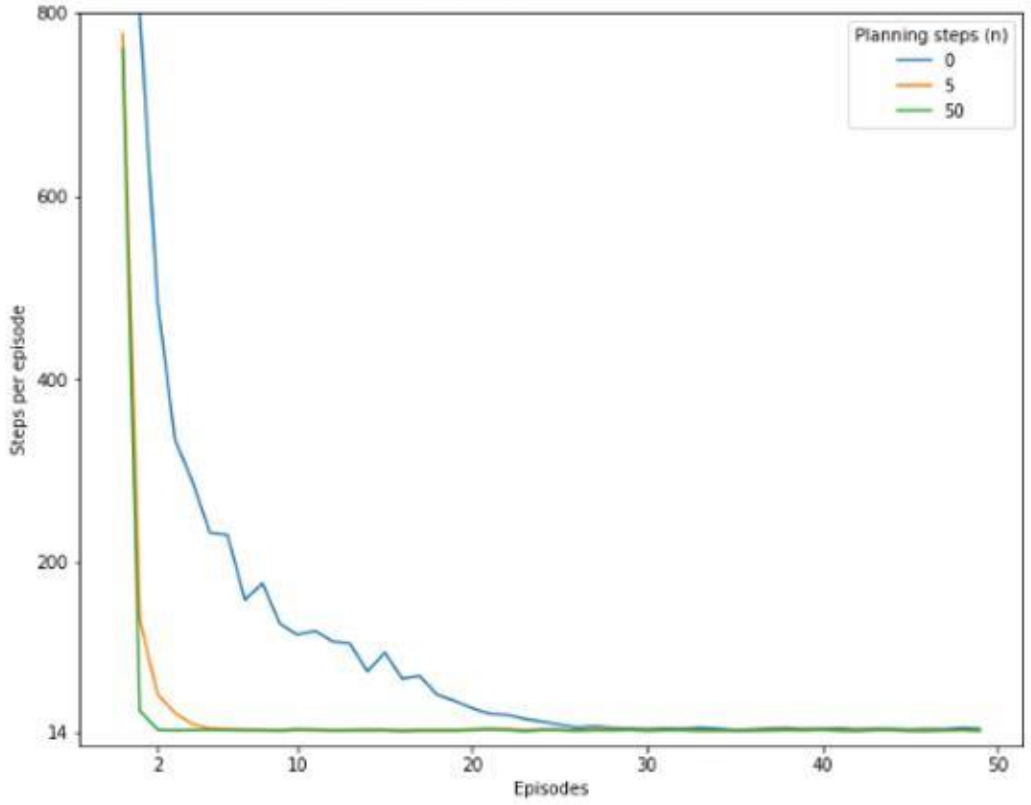
```
import numpy as np
import matplotlib.pyplot as plt

def choose_action(state):
    choice_values = Q_values[state[0],state[1],:]
    e = np.random.random()
    if e<EPSILON:
        action = np.random.choice(ACTIONS)
    else:
        action = np.random.choice(np.flatnonzero(choice_values == np.max(choice_values)))

    return action

def determine_transitions(state,action):
    terminated = False
```

```
    reward = 0
    new_state = state + np.array(ACTION_MOVE[action])
    if new_state[0] == 5 and new_state[1] == 8:
        reward = 1
        terminated = True
    elif new_state[0]==-1 or new_state[0]==6:
        new_state = state
    elif new_state[1] == -1 or new_state[1] == 9:
        new_state = state

    for I in OBSTACLES:
        if new_state[0] == I[0] and new_state[1] == I[1]:
            new_state = state

    return new_state,reward,terminated
def update_model_list(state_list,state,action):
    if len(state_list)==0:
        state_list.append([state,[action]])
    else:
        state_exists = False
        for index,I in enumerate(state_list):
            if I[0][0] == state[0] and I[0][1] == state[1]:
                state_exists = True
                if action not in I[1]:
                    state_list[index][1].append(action)
        if not state_exists:
            state_list.append([state, [action]])

    return state_list

def update_Qvalues(state1,action,reward,state2):
    choice_values = Q_values[state2[0], state2[1], :]
    max_action = np.random.choice(np.flatnonzero(choice_values == np.max(choice_values)))
    Q_values[state1[0],state1[1],action]
+=ALFA*(reward+GAMMA*Q_values[state2[0],state2[1],max_action] -
Q_values[state1[0],state1[1],action])
def run_experiments():
    experiment_steps = np.zeros(EPISODES)

    for experiment in range(EXPERIMENTS):
        if experiment in np.arange(0,EXPERIMENTS,EXPERIMENTS/6):
            print('Running experiment {}'.format(experiment))
        steps_episode = np.zeros(EPISODES)
        global Q_values
        Q_values = np.zeros((6,9,4))
        visited_states_list = []
        for episode in range(EPISODES):
```

```
        current_state = np.array([3,0])
        terminated = False
        step = 0
        while not terminated:
            action = choose_action(current_state)
            new_state,reward,terminated = determine_transitions(current_state,action)
            visited_states_list = update_model_list(visited_states_list,current_state,action)
            update_Qvalues(current_state,action,reward,new_state)
            current_state = new_state
            for i in range(PLANNING_STEPS):
                random_choice = np.random.choice(np.arange(len(visited_states_list)))
                random_state = visited_states_list[random_choice][0]
                random_action = np.random.choice(visited_states_list[random_choice][1])
                _new_state,_reward,_ = determine_transitions(random_state,random_action)
                update_Qvalues(random_state,random_action,_reward,_new_state)
            step+=1
        steps_episode[episode] = step

    experiment_steps+=steps_episode

  experiment_steps/=EXPERIMENTS
  return
EPSILON = 0.1
ALFA = 0.1
GAMMA = 0.95
ACTIONS = np.array([0,1,2,3])
PLANNING_STEPS_CHOICE = [0,5,50]
ACTION_MOVE = [[0, 1], [-1, 0], [0, -1], [1, 0]]

OBSTACLES = [[2,2],[3,2],[4,2],[1,5],[3,7],[4,7],[5,7]]
EXPERIMENTS = 30
EPISODES = 50
experiment_results = np.zeros((3,EPISODES))
for i,PLANNING_STEPS in enumerate(PLANNING_STEPS_CHOICE):
  print('Runnin planning steps = {}'.format(PLANNING_STEPS))
  experiment_results[i,:] = run_experiments()
plt.figure(figsize=(10,8))
Steps_episode_Plot = plt.subplot()
for i,n in enumerate(PLANNING_STEPS_CHOICE):
  Steps_episode_Plot.plot(np.arange(EPISODES),experiment_results[i,:],label = '{}'.format(n))
Steps_episode_Plot.set_xlabel('Episodes')
Steps_episode_Plot.set_ylabel('Steps per episode')
Steps_episode_Plot.set_ylim((0,800))
Steps_episode_Plot.set_yticks([14,200,400,600,800])
Steps_episode_Plot.set_xticks([2,10,20,30,40,50])
Steps_episode_Plot.legend(title = 'Planning steps (n)')
```

Output:

```
Runnin planning steps = 0
Running experiment 0
Running experiment 5
Running experiment 10
Running experiment 15
Running experiment 20
Running experiment 25
Runnin planning steps = 5
Running experiment 0
Running experiment 5
Running experiment 10
Running experiment 15
Running experiment 20
Running experiment 25
Runnin planning steps = 50
Running experiment 0
Running experiment 5
Running experiment 10
Running experiment 15
Running experiment 20
Running experiment 25
<matplotlib.legend.Legend at 0x7fab1273a6d0>
```



Result:

Thus, the Model-Based Approach (Dyna-Q) on Maze Environment is implemented successfully.

KUMARAGURU college of technology
character is life

IMPLEMENTATION OF ACTOR-CRITIC REINFORCEMENT ALGORITHM ON CARTPOLE ENVIRONMENT

Aim:

To implement the Actor Critic method on CartPole-V1 environment.

Introduction:

Environment Design - CartPole-V1

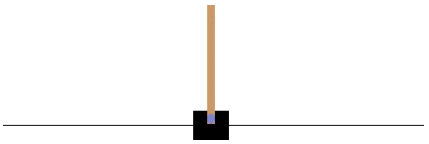
A pole is attached to a cart placed on a frictionless track. The agent must apply force to move the cart. It is rewarded for every time the pole remains upright. The agent, therefore, must learn to keep the pole from falling over.

Action Space	Discrete(2)
Observation Shape	(4,)
Observation High	[4.8 inf 0.42 inf]
Observation Low	[-4.8 -inf -0.42 -inf]
Import	`gym.make("CartPole-v1")`

Action Space

The action is a ndarray with shape (1,) which can take values {0, 1} indicating the direction of the fixed force the cart is pushed with.

Num	Action
0	Push cart to the left
1	Push cart to the right



The velocity that is reduced or increased by the applied force is not fixed and it depends on the angle the pole is pointing. The center of gravity of the pole varies the amount of energy needed to move the cart underneath it.

Observation Space

The observation is a ndarray with shape (4,) with the values corresponding to the following positions and velocities:

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -0.418 rad (-24°)	~ 0.418 rad (24°)
3	Pole Angular Velocity	-Inf	Inf

While the ranges above denote the possible values for observation space of each element, it is not reflective of the allowed values of the state space in an unterminated episode. Particularly:

- The cart x-position (index 0) can be taking values between (-4.8, 4.8), but the episode terminates if the cart leaves the (-2.4, 2.4) range.
- The pole angle can be observed between (-.418, .418) radians (or **±24°**), but the episode terminates if the pole angle is not in the range (-.2095, .2095) (or **±12°**)

Rewards

A reward of +1 is given for every time the pole remains upright.An episode ends when:

1) the pole is more than 15 degrees from vertical.
2) the cart moves more than 2.4 units from the center.

Starting State

All observations are assigned a uniformly random value in (-0.05, 0.05)

Episode End

The episode ends if any one of the following occurs:

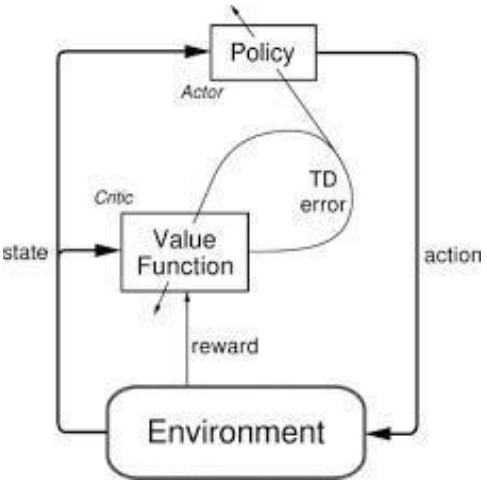
1. Termination: Pole Angle is greater than ±12°
2. Termination: Cart Position is greater than ±2.4 (center of the cart reaches the edge of the display)
3. Truncation: Episode length is greater than 500 (200 for v0)

Model Design: Actor Critic Method

As an agent takes actions and moves through an environment, it learns to map the observed state of the environment to two possible outputs:

1. Recommended action: A probability value for each action in the action space. The part of the agent responsible for this output is called the **actor**.
2. Estimated rewards in the future: Sum of all rewards it expects to receive in the future. The part of the agent responsible for this output is the **critic**.

Agent and Critic learn to perform their tasks, such that the recommended actions from the actor maximize the rewards.



Actor-Critic is a Temporal Difference (TD) version of Policy gradient. It has two networks: Actor and Critic. The actor decided which action should be taken and the critic inform the actor how good was the action and how it should adjust. The learning of the actor is based on policy gradient approach. In comparison, critics evaluate the action produced by the actor by computing the value function.

Algorithm

- Import necessary packages and configure global settings.
- Create the CartPole-V1 environment using Gym.
- Setup the actor and critic neural network
- Initialize the discount factor γ and the learning rate α, epsilon

1. Run the agent on the environment to collect training data per episode.
2. Sample {s_t, a_t} using the policy πθ from the actor-network.
3. Evaluate the advantage function A_t.
4. Compute expected return at each time step.

$$A_{\pi_\theta}(s_t, a_t) = r(s_t, a_t) + V_{\pi_\theta}(s_{t+1}) - V_{\pi_\theta}(s_t)$$

5. Compute the loss for the combined Actor-Critic model.
6. Evaluate the gradient using the below expression:

$$\nabla J(\theta) \approx \sum_{t=0}^{T-1} \nabla_\theta log \pi_\theta(a_t, s_t) A_{\pi_\theta}(s_t, a_t)$$

7. Update the policy parameters, θ of the critic-based value-based RL(Q-learning). δt is equivalent to advantage function.

$$\theta = \theta + \alpha \nabla J(\theta)$$

8. Repeat 1-8 until either success criterion or max episodes has been reached or until we find the optimal policy πθ.

$$w = w + \alpha \delta_t$$

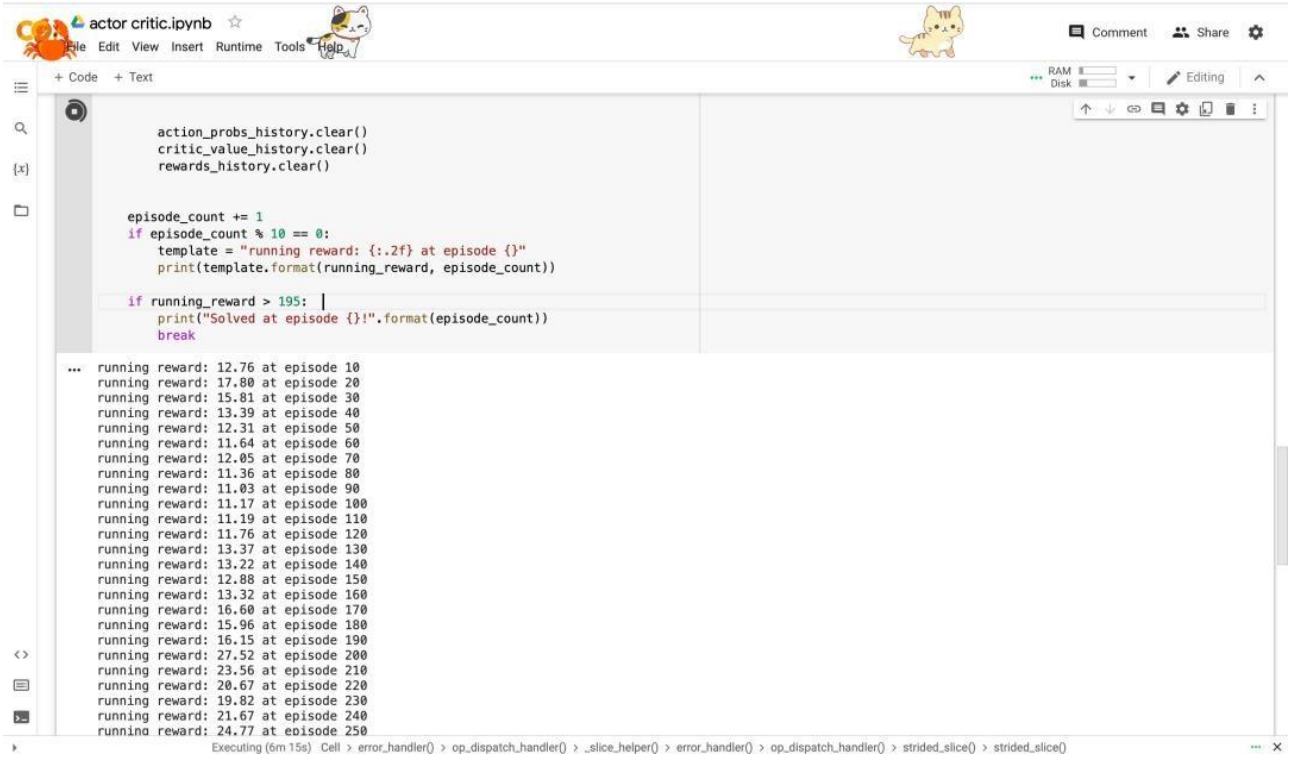
Program:

```
import gym
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
seed = 42
gamma = 0.99
max_steps_per_episode = 10000
env = gym.make("CartPole-v0")
env.seed(seed)
eps = np.finfo(np.float32).eps.item()
num_inputs = 4
num_actions = 2
num_hidden = 128

inputs = layers.Input(shape=(num_inputs,))
common = layers.Dense(num_hidden, activation="relu")(inputs)
action = layers.Dense(num_actions, activation="softmax")(common)
critic = layers.Dense(1)(common)

model = keras.Model(inputs=inputs, outputs=[action, critic])
optimizer = keras.optimizers.Adam(learning_rate=0.01)
huber_loss = keras.losses.Huber()
action_probs_history = []
critic_value_history = []
rewards_history = []
running_reward = 0
episode_count = 0

while True:
  state = env.reset()
   episode_reward = 0
   with tf.GradientTape() as tape:
      for timestep in range(1, max_steps_per_episode):
         state = tf.convert_to_tensor(state)
         state = tf.expand_dims(state, 0)
         action_probs, critic_value = model(state)
         critic_value_history.append(critic_value[0, 0])
         action = np.random.choice(num_actions, p=np.squeeze(action_probs))
         action_probs_history.append(tf.math.log(action_probs[0, action]))
         state, reward, done, _ = env.step(action)
         rewards_history.append(reward)
         episode_reward += reward
```

```
        if done:
            break
    running_reward = 0.05 * episode_reward + (1 - 0.05) * running_reward
    returns = []
    discounted_sum = 0
    for r in rewards_history[::-1]:
        discounted_sum = r + gamma * discounted_sum
        returns.insert(0, discounted_sum)

    returns = np.array(returns)
    returns = (returns - np.mean(returns)) / (np.std(returns) + eps)
    returns = returns.tolist()

    history = zip(action_probs_history, critic_value_history, returns)
    actor_losses = []
    critic_losses = []
    for log_prob, value, ret in history:
            diff = ret - value
        actor_losses.append(-log_prob * diff) # actor loss
       critic_losses.append(
            huber_loss(tf.expand_dims(value, 0), tf.expand_dims(ret, 0))
        )

    loss_value = sum(actor_losses) + sum(critic_losses)
    grads = tape.gradient(loss_value, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
    action_probs_history.clear()
    critic_value_history.clear()
    rewards_history.clear()


episode_count += 1
if episode_count % 10 == 0:
    template = "running reward: {:.2f} at episode {}"
    print(template.format(running_reward, episode_count))

if running_reward > 195:
   print("Solved at episode {}!".format(episode_count))
    break
```

Output:

Solved episode 713!



Result:

Thus, the Actor Critic method on CartPole-V1 environment implemented successfully.

KUMARAGURU college of technology
character is life

IMPLEMENTATION OF PPO IN TAXI ENVIRONMENT

Aim:

The objective is to implement the PPO in Taxi Environment.

Algorithm:

Step: 1 - Import required libraries, including Gym and TensorFlow and define the Policy Network class for the policy network.

Step: 2 - Create the Gym Taxi-v3 environment and initialize the policy network and optimizer.

Step: 3 - Define the PPO loss function (ppo_loss) that calculates the surrogate objective for policy optimization.

Step: 4 - For each episode:

- Collect states, actions, rewards, and old action probabilities during the episode.
- Compute the discounted rewards and advantages for each time step.
- Perform PPO updates using the collected data and advantages.

Step: 5 – Train and test the policy in the environment, calculate and print the average reward over the testing episodes.

Program:

```
 import gym
env = gym.make("Taxi-v3")
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.optimizers import Adam
class PolicyNetwork(tf.keras.Model):
    def __init__(self, num_actions):
        super(PolicyNetwork, self).__init__()
        self.dense1 = layers.Dense(64, activation='relu')
        self.dense2 = layers.Dense(num_actions, activation='softmax')

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

# Initialize policy network
num_actions = env.action_space.n
policy_network = PolicyNetwork(num_actions)

optimizer = Adam(learning_rate=0.001)
def ppo_loss(old_probs, new_probs, advantages, epsilon=0.2):
    ratio = new_probs / (old_probs + 1e-5)
    surrogate1 = ratio * advantages
```

```
    surrogate2 = tf.clip_by_value(ratio, 1 - epsilon, 1 + epsilon) * advantages
    return -tf.reduce_mean(tf.minimum(surrogate1, surrogate2))


def train_ppo(policy_network, optimizer, states, actions, old_probs, advantages):
    with tf.GradientTape() as tape:
        new_probs = policy_network(states)
        loss = ppo_loss(old_probs, new_probs, advantages)
    grads = tape.gradient(loss, policy_network.trainable_variables)
    optimizer.apply_gradients(zip(grads, policy_network.trainable_variables))
num_episodes = 1000
max_steps = 200
discount_factor = 0.99
observation = env.reset()
tf.debugging.enable_check_numerics()
for episode in range(num_episodes):
    states = []
    actions = []
    rewards = []
    old_probs = []

    for step in range(max_steps):
        state = np.reshape(observation, [1, -1])
        states.append(state)

        action_probs = policy_network(state)
        action = np.random.choice(num_actions, p=np.squeeze(action_probs))
        actions.append(action)
        old_probs.append(action_probs[0][action])

        observation, reward, done, _ = env.step(action)
        rewards.append(reward)

        if done:
            break

    discounted_rewards = []
    advantage = 0
    for r in rewards[::-1]:
        advantage = r + discount_factor * advantage
        discounted_rewards.append(advantage)
    discounted_rewards.reverse()
    advantages = discounted_rewards

    states = tf.concat(states, axis=0)
    actions = np.array(actions)
    old_probs = tf.convert_to_tensor(old_probs, dtype=tf.float32)
    advantages = tf.convert_to_tensor(advantages, dtype=tf.float32)

    train_ppo(policy_network, optimizer, states, actions, old_probs, advantages)
```

```
  if episode % 10 == 0:
      print(f"Episode: {episode}, Total Reward: {np.sum(rewards)}")
total_rewards = []
for episode in range(100):
    observation = env.reset()
    episode_reward = 0

    for step in range(max_steps):
        state = np.reshape(observation, [1, -1])
        action_probs = policy_network(state)
        action = np.argmax(np.squeeze(action_probs))
        observation, reward, done, _ = env.step(action)
        episode_reward += reward
        if done:
            break
    total_rewards.append(episode_reward)

print("Average reward over 100 episodes:", np.mean(total_rewards))
```

Output:

```
Episode: 0, Total Reward: -1
Episode: 10, Total Reward: -1
Episode: 20, Total Reward: -10
Episode: 30, Total Reward: -1
Episode: 40, Total Reward: -1
Episode: 50, Total Reward: -1
Episode: 60, Total Reward: -1
Episode: 70, Total Reward: -1
Episode: 80, Total Reward: -1
Episode: 90, Total Reward: -1
Episode: 100, Total Reward: -1
Episode: 110, Total Reward: -1
Episode: 120, Total Reward: -1
Episode: 130, Total Reward: -1
Episode: 140, Total Reward: -1
Episode: 150, Total Reward: -1
Episode: 160, Total Reward: -1
Episode: 170, Total Reward: -1
Episode: 180, Total Reward: -1
Episode: 190, Total Reward: -1
Episode: 200, Total Reward: -1
Episode: 210, Total Reward: -1
Episode: 220, Total Reward: -1
Episode: 230, Total Reward: -1
Episode: 240, Total Reward: -1
Episode: 250, Total Reward: -1
```

```
Average reward over 100 episodes: -200.0
```

Result:

Thus, the Proximal Policy Optimization (PPO) in taxi environment is successfully implemented.