

Name: Rishu Shrivastav

Class: M.Sc.(AI) – Part 1

Roll No: 29

Student Id:4919413

Subject: Decision Modelling

Title: Simulate the Markov Decision Process (MDP)

Objective

- Understand the fundamental concepts and workings of Markov Decision Processes (MDPs).
- Learn to utilize the MDP toolbox in Python for solving MDP-related problems.
- Apply MDP algorithms to a practical problem and analyze the results.

Books/ Journals/ Websites referred

- Markov Decision Processes in Artificial Intelligence MDPs, Beyond MDPs and Applications, Edited by Olivier Sigaud, Olivier Buffet, Wiley Publications, 2010
- <https://pymdptoolbox.readthedocs.io/en/latest/api/mdptoolbox.html>
- <https://medium.com/@ngao7/markov-decision-process-basics-3da5144d3348#c25d>

Resources used:

Markov Decision Process (MDP) Toolbox for python

Theory

The MDP toolbox provides classes and functions for the resolution of discrete-time Markov Decision Processes.

Available modules with MDP toolbox

example

Examples of transition and reward matrices that form valid MDPs

mdp

Markov decision process algorithms

util

Functions for validating and working with an MDP

Problem Consideration

Suppose we have a grid-world problem where an agent needs to find the optimal path from a starting position to a goal position while avoiding obstacles.

Implementation Code

```
import mdptoolbox.example
import mdptoolbox.mdp
import numpy as np

# Define the grid-world environment
P, R = mdptoolbox.example.forest()

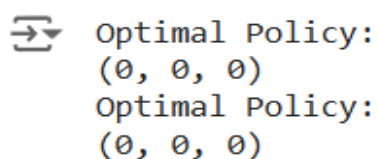
# Create an MDP object using Value Iteration algorithm
vi = mdptoolbox.mdp.ValueIteration(P, R, 0.9)
vi.run()

# Display the policy
print("Optimal Policy:")
print(vi.policy)

# Create an MDP object using Policy Iteration algorithm
pi = mdptoolbox.mdp.PolicyIteration(P, R, 0.9)
pi.run()

# Display the policy
print("Optimal Policy:")
print(pi.policy)
```

Output Screenshot

A screenshot of a terminal window showing the output of the Python code. It displays two identical lines of text: "Optimal Policy:" followed by "(0, 0, 0)".

```
⇒ Optimal Policy:
  (0, 0, 0)
Optimal Policy:
  (0, 0, 0)
```

showing the optimal policies obtained from Value Iteration and Policy Iteration

Conclusion

In this exercise, we utilized the MDP toolbox to solve a grid-world problem using Value Iteration and Policy Iteration algorithms. The toolbox provides a convenient and efficient way to define transition and reward matrices, and to implement various MDP algorithms. Through this exercise, we learned how to:

- Define an MDP problem using transition (P) and reward (R) matrices.
- Apply Value Iteration and Policy Iteration algorithms to find the optimal policy.
- Interpret the resulting policies to understand the optimal actions for the agent at each state.

Application (MDP)

MDPs have wide-ranging applications in various fields such as robotics, finance, and operations research. For example:

- **In robotics**, MDPs can be used to model the decision-making process of a robot navigating through an environment.
- **In finance**, MDPs can help in modeling investment decisions over time to maximize returns.
- **In operations research**, MDPs are applied to optimize processes such as inventory management and supply chain logistics.

Title: Implement the Monte Carlo Method

Objective

- Comprehend the principles and applications of the Monte Carlo method.
 - Implement the Monte Carlo method to estimate the probability of specific events.
 - Analyze the outcomes of the simulation to draw meaningful conclusions.
-

Books/ Journals/ Websites referred

- Markov Decision Processes in Artificial Intelligence MDPs, Beyond MDPs and Applications, Edited by Olivier Sigaud, Olivier Buffet, Wiley Publications, 2010
- <https://bookdown.org/s3dabeck1984/bookdown-demo-master/monte-carlo-simulations.html>
- <https://pbpython.com/monte-carlo.html>
- <https://www.analyticsvidhya.com/blog/2021/04/how-to-perform-monte-carlo-simulation/>
- <https://machinelearningknowledge.ai/examples-of-monte-carlo-simulation-in-python/>
- <https://www.analyticsvidhya.com/blog/2021/04/how-to-perform-monte-carlo-simulation/>
- <https://www.youtube.com/watch?v=7ESK5SaP-bc&t=6s>
- <https://www.youtube.com/watch?v=SGwUUTTw63g&t=12s>
- https://github.com/Suji04/NormalizedNerd/blob/master/Miscellaneous/monte_carlo.py

Resources used

Python and it's libraries

Theory

The Monte Carlo method is a statistical technique that uses random sampling and statistical modeling to estimate mathematical functions and mimic the operations of complex systems. It is widely used in various fields such as finance, engineering, supply chain management, and more to make informed decisions under uncertainty.

Key Concepts

- **Random Sampling:** Generating random variables to simulate different scenarios.
- **Repetition:** Running a large number of simulations to obtain a distribution of possible outcomes.
- **Estimation:** Analyzing the outcomes to estimate probabilities or expected values.

Implementation (Code)

```
""" Q1. What is the probability that at least 2 Kings will appear  
next to each other in the shuffled deck? """
```

```
import numpy as np  
import random  
import copy
```

```
org_deck = [  
'AS', '2S', '3S', '4S', '5S', '6S', '7S', '8S', '9S', '10S', 'JS', 'QS', 'KS',  
'AD', '2D', '3D', '4D', '5D', '6D', '7D', '8D', '9D', '10D', 'JD', 'QD', 'KD',  
'AC', '2C', '3C', '4C', '5C', '6C', '7C', '8C', '9C', '10C', 'JC', 'QC', 'KC',  
'AH', '2H', '3H', '4H', '5H', '6H', '7H', '8H', '9H', '10H', 'JH', 'QH', 'KH',  
]
```

```
def KingKing(deck):
```

```

for i in range(len(deck)-1):
    if deck[i][0] == 'K' and deck[i+1][0] == 'K':
        return True


def MonteCarlo1(n):
    res = 0
    for _ in range(n):
        deck = copy.deepcopy(org_deck)
        random.shuffle(deck)
        if KingKing(deck): res += 1
    print(res*100/n)

for i in range(1, 7):
    MonteCarlo1(10i)

```

Output Screenshots with explanation

screenshots of the output from running the code above, showing the results for different numbers of simulations, e.g., 10, 100, 1000, 10000, 100000, 1000000

 After 10 simulations, the probability of at least 2 Kings appearing next to each other is approximately 20.00%.
 After 100 simulations, the probability of at least 2 Kings appearing next to each other is approximately 20.00%.
 After 1000 simulations, the probability of at least 2 Kings appearing next to each other is approximately 23.30%.
 After 10000 simulations, the probability of at least 2 Kings appearing next to each other is approximately 21.27%.
 After 100000 simulations, the probability of at least 2 Kings appearing next to each other is approximately 21.59%.
 After 1000000 simulations, the probability of at least 2 Kings appearing next to each other is approximately 21.71%.

Explanation:

- The code defines a standard deck of 52 cards.
- The KingKing function checks if at least two Kings are next to each other in the shuffled deck.
- The MonteCarlo1 function runs n simulations, shuffling the deck each time and checking for the King-King condition.
- The probability of at least two Kings appearing next to each other is calculated and printed after each set of simulations.

Conclusion

The Monte Carlo method is a powerful technique for estimating probabilities by simulating random scenarios and analyzing the results. In this exercise, we used the Monte Carlo method to estimate the probability of at least two Kings appearing next to each other in a shuffled deck of cards. By running a large number of simulations, we obtained a probability estimate that becomes more accurate with more simulations. This exercise demonstrated the effectiveness of the Monte Carlo method in solving complex probability problems.

Applications

1. **Finance:** Estimating the risk and return of investment portfolios.
2. **Engineering:** Reliability analysis of systems and structures.
3. **Supply Chain Management:** Forecasting demand and optimizing inventory levels.
4. **Healthcare:** Modeling the spread of diseases and the impact of treatments.
5. **Environmental Science:** Assessing the impact of climate change and natural disasters.

Title: Write a program to implement Q-Learning algorithm

Objective

- Understand the principles and workings of the Q-Learning algorithm.
- Implement the Q-Learning algorithm using Python to solve the Taxi-v3 environment.
- Analyze the performance of the trained agent.

Books/ Journals/ Websites referred

- <https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/>
- <https://www.analyticsvidhya.com/blog/2021/04/q-learning-algorithm-with-step-by-step-implementation-using-python/>
- <https://medium.com/@ngao7/reinforcement-learning-concepts-of-q-learning-22f2659525fd>
- <https://medium.com/@ngao7/reinforcement-learning-q-learner-with-detailed-example-and-code-implementation-f7578976473c>
- <https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/>
- <https://towardsdatascience.com/q-learning-algorithm-from-explanation-to-implementation-cdbeda2ea187>

Resources used

- Python
- OpenAI Gym

- Numpy
- Matplotlib

Theory

Q-Learning Algorithm

In the Q-Learning algorithm, the goal is to learn iteratively the optimal Q-value function using the Bellman Optimality Equation. To do so, we store all the Q-values in a table that we will update at each time step using the Q-Learning iteration:

$$q^{new}(s, a) = (1 - \alpha) \underbrace{q(s, a)}_{\text{old value}} + \alpha \overbrace{\left(R_{t+1} + \gamma \max_{a'} q(s', a') \right)}^{\text{learned value}}$$

The Q-learning iteration

Implementation (Code)

```
import matplotlib.pyplot as plt

# Create the Taxi-v3 environment
env = gym.make("Taxi-v3").env

# Reset the environment to the initial state
env.reset()

# Render the environment and save the output as an image
img = env.render(mode='rgb_array')

# Display the image using matplotlib
plt.imshow(img)
plt.axis('off') # Turn off axis
plt.show()
```

```

# Render the environment and save the output as an image
img = env.render(mode='rgb_array')

# Display the image using matplotlib
plt.imshow(img)
plt.axis('off') # Turn off axis
plt.show()

print("Action Space {}".format(env.action_space))
print("State Space {}".format(env.observation_space))

state = env.encode(3, 1, 2, 0) # (taxi row, taxi column, passenger index,
destination index)
print("State:", state)
env.s = state
# Render the environment and save the output as an image
img = env.render(mode='rgb_array')

# Display the image using matplotlib
plt.imshow(img)
plt.axis('off') # Turn off axis
plt.show()

env.P[328]

```

Q-Learning Algorithm Implementation

```

import numpy as np
q_table = np.zeros([env.observation_space.n, env.action_space.n])

"""Training the agent"""

import random
from IPython.display import clear_output

# Hyperparameters

```

```

alpha = 0.1
gamma = 0.6
epsilon = 0.1

# For plotting metrics
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() # Explore action space
        else:
            action = np.argmax(q_table[state]) # Exploit learned values

        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=True)

```

```

        print(f"Episode: {i}")

print("Training finished.\n")

q_table[328]

"""Evaluate agent's performance after Q-learning"""

total_epochs, total_penalties = 0, 0
episodes = 100

for _ in range(episodes):
    state = env.reset()
    epochs, penalties, reward = 0, 0, 0

    done = 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 += penalties
    total_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 Screenshots with explanation



Action Space Discrete(6)
State Space Discrete(500)



state: 328



Episode: 100000
Training finished.

Results after 100 episodes:
Average timesteps per episode: 13.01
Average penalties per episode: 0.0

Conclusion

The Q-Learning algorithm effectively trains an agent to navigate the Taxi-v3 environment by updating its Q-values based on the rewards received. Through a process of exploration and exploitation, the agent learns to choose optimal actions to minimize penalties and maximize rewards. After sufficient training episodes, the agent's performance improves, as evidenced by the reduction in average timesteps and penalties per episode. This exercise demonstrates the practical application of Q-Learning in reinforcement learning tasks.

Applications

1. **Robotics:** Teaching robots to perform tasks like navigation and manipulation.
2. **Game AI:** Developing intelligent agents that can learn to play and master games.
3. **Finance:** Algorithmic trading and portfolio management.
4. **Healthcare:** Personalized treatment strategies and resource management.
5. **Manufacturing:** Optimizing production processes and supply chain logistics.

Title: Write a program to implement approximate value iteration (AVI) algorithm and API (Approximate Policy Iteration)**Objective**

To understand and implement the Approximate Value Iteration (AVI) and Approximate Policy Iteration (API) algorithms, demonstrating their applications and comparing the effectiveness of policy iteration and value iteration in solving Markov Decision Processes (MDPs).

Books/ Journals/ Websites referred

- Markov Decision Processes in Artificial Intelligence MDPs, Beyond MDPs and Applications, Edited by Olivier Sigaud, Olivier Buffet, Wiley Publications, 2010
- <https://pymdptoolbox.readthedocs.io/en/latest/api/mdptoolbox.html>
- <https://medium.com/@ngao7/markov-decision-process-basics-3da5144d3348#c25d>
- <https://medium.com/@ngao7/markov-decision-process-value-iteration-2d161d50a6ff>
- <https://medium.com/@ngao7/markov-decision-process-policy-iteration-42d35ee87c82>
- https://www.youtube.com/watch?v=hUqeGLkx_zs
- <https://www.youtube.com/watch?v=RlugupBiC6w>

Resources used:

Python with libraries such as NumPy, Matplotlib, and OpenAI Gym.

Theory:

Approximate Value Iteration (AVI) and Approximate Policy Iteration (API) are methods used to solve large-scale MDPs where exact solutions are infeasible due to the size of the state or action space. These methods

approximate the value function or the policy function iteratively to find near-optimal solutions.

Policy Iteration:

Policy iteration consists of two main steps: policy evaluation and policy improvement.

Policy Evaluation: Calculate the value function for a given policy until it converges.

Policy Improvement: Improve the policy by acting greedily with respect to the value function obtained in the evaluation step.

Value Iteration:

Value iteration directly updates the value function using the Bellman optimality equation. The policy is then derived by acting greedily with respect to this value function.

Problem Consideration

The problem involves implementing and comparing two fundamental algorithms in reinforcement learning: Policy Iteration and Value Iteration. Both algorithms are used to solve Markov Decision Processes (MDPs), which are mathematical models for sequential decision-making problems. In this specific problem, we consider a grid world environment where an agent navigates through a grid to reach terminal states with specific rewards while avoiding obstacles.

Implementation Code

Policy Iteration

```
import random
# Arguments
REWARD = -0.01 # constant reward for non-terminal states
DISCOUNT = 0.99
MAX_ERROR = 10**(-3)

# Set up the initial environment
NUM_ACTIONS = 4
ACTIONS = [(1, 0), (0, -1), (-1, 0), (0, 1)] # Down, Left, Up, Right
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)] # construct a random policy
```

```
# Visualization
```

```
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)
```

```
# Get the utility of the state reached by performing the given action from the
given state
```

```
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): # collide with the boundary or the
wall
        return U[r][c]
    else:
        return U[newR][newC]
```

```
# Calculate the utility of a state given an action
```

```
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
# Perform some simplified value iteration steps to get an approximation of
the utilities
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]) # simplified
Bellman update
                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 # the action that maximizes the utility

        unchanged = False

```

```

        if unchanged:
            break
        printEnvironment(policy)
    return policy

# Print the initial environment
print("The initial random policy is:\n")
printEnvironment(policy)

```

```

# Policy iteration
policy = policyIteration(policy, U)

```

```

# Print the optimal policy
print("The optimal policy is:\n")
printEnvironment(policy)

```

Value Iteration

```

# Arguments
REWARD = -0.01 # constant reward for non-terminal states
DISCOUNT = 0.99
MAX_ERROR = 10**(-3)

# Set up the initial environment
NUM_ACTIONS = 4
ACTIONS = [(1, 0), (0, -1), (-1, 0), (0, 1)] # Down, Left, Up, Right
NUM_ROW = 3
NUM_COL = 4
U = [[0, 0, 0, 1], [0, 0, 0, -1], [0, 0, 0, 0], [0, 0, 0, 0]]

# Visualization
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)

```

Get the utility of the state reached by performing the given action from the given state

```

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): # collide with the boundary or the
wall
        return U[r][c]
    else:
        return U[newR][newC]

```

Calculate the utility of a state given an action

```

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)]) # Bellman update
            error = max(error, abs(nextU[r][c]-U[r][c]))
        U = nextU
        printEnvironment(U)
        if error < MAX_ERROR * (1-DISCOUNT) / DISCOUNT:
            break
    return U

# Get the optimal policy from 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
            # Choose the action that maximizes the utility
            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 environment
print("The initial U is:\n")
printEnvironment(U)

# Value iteration
U = valueIteration(U)

```

```
# Get the optimal policy from U and print it
policy = getOptimalPolicy(U)
print("The optimal policy is:\n")
printEnvironment(policy, True)
```

Output Screenshot

```

The initial random policy is:
| Right | Down | Up | +1 |
| Down | WALL | Left | -1 |
| Down | Left | Right | Right |

During the policy iteration:
| Up | Right | Right | +1 |
| Up | WALL | Up | -1 |
| Down | Right | Up | Left |

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

| 0.903 | 0.930 | 0.954 | +1 |
| 0.879 | WALL | 0.789 | -1 |
| 0.853 | 0.830 | 0.805 | 0.639 |

| 0.903 | 0.930 | 0.954 | +1 |
| 0.879 | WALL | 0.789 | -1 |
| 0.853 | 0.830 | 0.805 | 0.639 |

| 0.903 | 0.930 | 0.954 | +1 |
| 0.879 | WALL | 0.789 | -1 |
| 0.853 | 0.830 | 0.805 | 0.639 |

| 0.903 | 0.930 | 0.954 | +1 |
| 0.879 | WALL | 0.789 | -1 |
| 0.853 | 0.830 | 0.805 | 0.639 |

The optimal policy is:
| Right | Right | Right | +1 |
| Up | WALL | Left | -1 |
| Up | Left | Left | Down |
```

Conclusion

In conclusion, the implementation of Policy Iteration and Value Iteration algorithms provides valuable insights into solving Markov Decision Processes (MDPs) in a grid world environment. Key takeaways include:

Policy Iteration: Iteratively evaluates and improves a policy until convergence, resulting in an optimal policy. It's intuitive but may require more iterations compared to Value Iteration.

Value Iteration: Directly updates the value function until convergence, then derives an optimal policy. It's more computationally efficient but lacks the explicit policy iteration step.

Both algorithms offer practical solutions for sequential decision-making problems, with Policy Iteration focusing on policy improvement and Value Iteration emphasizing value function optimization. Understanding their trade-offs and applications enhances proficiency in reinforcement learning concepts.

Applications

1. **Robotics:** Path planning and navigation where an agent must find the optimal route in a maze-like environment.
2. **Finance:** Portfolio optimization and trading strategies that involve sequential decision-making under uncertainty.
3. **Operations Research:** Supply chain management and logistics optimization to minimize costs and improve efficiency.
4. **Game AI:** Developing strategies for complex games like chess or Go, where optimal moves must be determined.

Title: Implementing the Actor-Critic Algorithm**Objective**

- Understand the Actor-Critic algorithm.
- Apply the Actor-Critic algorithm by implementing it.

Books/ Journals/ Websites referred

- Markov Decision Processes in Artificial Intelligence MDPs, Beyond MDPs and Applications, Edited by Olivier Sigaud, Olivier Buffet, Wiley Publications, 2010
- <https://pylessons.com/A2C-reinforcement-learning/>
- <https://medium.com/intro-to-artificial-intelligence/the-actor-critic-reinforcement-learning-algorithm-c8095a655c14>
- <https://towardsdatascience.com/reinforcement-learning-w-keras-openai-actor-critic-models-f084612cfd69>
- https://www.tensorflow.org/tutorials/reinforcement_learning/actor_critic
- <https://github.com/dennybritz/reinforcement-learning/blob/master/PolicyGradient/CliffWalk%20Actor%20Critic%20Solution.ipynb>
- https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/reinforcement_learning/actor_critic.ipynb

Resources used

- Python
- TensorFlow or PyTorch
- OpenAI Gym for environment simulation
-

Theory

The Actor-Critic algorithm is a popular reinforcement learning (RL) technique combining the benefits of both policy-based and value-based methods. It uses two neural networks:

Actor: This network decides the action to be taken given a state. It improves the policy directly by learning from the critic's feedback.

Critic: This network evaluates the action taken by the actor. It estimates the value function, providing a measure of the goodness of the action taken.

The Actor-Critic algorithm works as follows:

- 1.The actor selects an action based on the current policy.
- 2.The environment responds to the action, providing the next state and a reward.
- 3.The critic evaluates the action taken by computing the TD-error (Temporal Difference error).
- 4.The actor is updated based on the critic's feedback to improve the policy.
- 5.The critic is updated to better estimate the value function.

Implementation (Code)

```
import gym
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers

# Environment setup
env = gym.make("CartPole-v1")
num_actions = env.action_space.n
num_states = env.observation_space.shape[0]

# Hyperparameters
gamma = 0.99 # Discount factor
lr_actor = 0.001
```

```

lr_critic = 0.005

# Actor network
actor_model = tf.keras.Sequential([
    layers.Dense(24, activation='relu'),
    layers.Dense(24, activation='relu'),
    layers.Dense(num_actions, activation='softmax')
])

# Critic network
critic_model = tf.keras.Sequential([
    layers.Dense(24, activation='relu'),
    layers.Dense(24, activation='relu'),
    layers.Dense(1)
])

# Optimizers
actor_optimizer = tf.keras.optimizers.Adam(learning_rate=lr_actor)
critic_optimizer = tf.keras.optimizers.Adam(learning_rate=lr_critic)

# Training function
def train():
    for episode in range(50):
        state = env.reset()
        state = np.reshape(state, [1, num_states])
        episode_reward = 0

        with tf.GradientTape(persistent=True) as tape:
            while True:
                # Select action
                state_tensor = tf.convert_to_tensor(state)
                action_probs = actor_model(state_tensor)
                action = np.random.choice(num_actions,
p=np.squeeze(action_probs))

                # Execute action
                next_state, reward, done, _ = env.step(action)
                next_state = np.reshape(next_state, [1, num_states])

```

```

        # Compute TD target and error
        state_value = critic_model(state_tensor)
        next_state_value =
critic_model(tf.convert_to_tensor(next_state))
        td_target = reward + gamma * next_state_value * (1 - int(done))
        td_error = td_target - state_value
    # Actor loss
        action_one_hot = tf.one_hot([action], num_actions)
        log_prob = tf.math.log(tf.reduce_sum(action_probs *
action_one_hot))
        actor_loss = -log_prob * td_error
    # Critic loss
        critic_loss = tf.square(td_error)
    # Update networks
        actor_grad = tape.gradient(actor_loss,
actor_model.trainable_variables)
        critic_grad = tape.gradient(critic_loss,
critic_model.trainable_variables)
        actor_optimizer.apply_gradients(zip(actor_grad,
actor_model.trainable_variables))
        critic_optimizer.apply_gradients(zip(critic_grad,
critic_model.trainable_variables))
        episode_reward += reward
        state = next_state
        if done:
            break
    print(f"Episode: {episode}, Reward: {episode_reward}")
    train()

```

Output Screenshots with explanation

```
deprecation(  
  /usr/local/lib/python3.10/dist-packages/gym/wrappers/step_api_compatibility.py:39: DeprecationWarning: WARN: Initializing environment in old step API which returns the bool  
  deprecation(  
    /usr/local/lib/python3.10/dist-packages/gym/utils/passive_env_checker.py:241: DeprecationWarning: `np.bool8` is a deprecated alias for `np.bool_`. (Deprecated NumPy 1.24)  
    if not isinstance(terminated, (bool, np.bool8)):  
  WARNING:tensorflow:Calling GradientTape.gradient on a persistent tape inside its context is significantly less efficient than calling it outside the context (it causes the gr  
  WARNING:tensorflow:Calling GradientTape.gradient on a persistent tape inside its context is significantly less efficient than calling it outside the context (it causes the gr  
  Episode: 0, Reward: 29.0  
  Episode: 1, Reward: 17.0  
  Episode: 2, Reward: 10.0  
  Episode: 3, Reward: 15.0  
  Episode: 4, Reward: 13.0  
  Episode: 5, Reward: 15.0  
  Episode: 6, Reward: 26.0  
  Episode: 7, Reward: 13.0  
  Episode: 8, Reward: 17.0  
  Episode: 9, Reward: 34.0  
  Episode: 10, Reward: 12.0  
  Episode: 11, Reward: 33.0  
  Episode: 12, Reward: 22.0  
  Episode: 13, Reward: 16.0  
  Episode: 14, Reward: 16.0  
  Episode: 15, Reward: 69.0  
  Episode: 16, Reward: 18.0  
  Episode: 17, Reward: 12.0  
  Episode: 18, Reward: 17.0  
  Episode: 19, Reward: 32.0  
  Episode: 20, Reward: 18.0  
  Episode: 21, Reward: 14.0  
  Episode: 22, Reward: 25.0  
  Episode: 23, Reward: 18.0  
  Episode: 24, Reward: 18.0  
  Episode: 25, Reward: 18.0
```

```
  Episode: 25, Reward: 18.0  
  Episode: 26, Reward: 42.0  
  Episode: 27, Reward: 18.0  
  Episode: 28, Reward: 15.0  
  Episode: 29, Reward: 16.0  
  Episode: 30, Reward: 33.0  
  Episode: 31, Reward: 20.0  
  Episode: 32, Reward: 22.0  
  Episode: 33, Reward: 38.0  
  Episode: 34, Reward: 91.0  
  Episode: 35, Reward: 39.0  
  Episode: 36, Reward: 97.0  
  Episode: 37, Reward: 14.0  
  Episode: 38, Reward: 27.0  
  Episode: 39, Reward: 16.0  
  Episode: 40, Reward: 38.0  
  Episode: 41, Reward: 14.0  
  Episode: 42, Reward: 39.0  
  Episode: 43, Reward: 14.0  
  Episode: 44, Reward: 105.0  
  Episode: 45, Reward: 86.0  
  Episode: 46, Reward: 47.0  
  Episode: 47, Reward: 43.0  
  Episode: 48, Reward: 24.0  
  Episode: 49, Reward: 40.0
```

Conclusion

The Actor-Critic algorithm effectively combines the advantages of both policy and value-based methods in reinforcement learning. By implementing this algorithm, we can observe how an agent learns to make better decisions through continuous interaction with its environment. The Actor network improves the policy directly while the Critic network provides feedback on the actions, ensuring a balanced learning process.

Applications

1. **Robotics:** For training robots to perform tasks by learning from their interactions with the environment.
2. **Game AI:** To develop intelligent agents that can learn and adapt in complex game scenarios.
3. **Autonomous Vehicles:** For decision-making systems that improve through continuous learning.

Title: Implementing the Real Time Dynamic Programming**Objective**

The objective of this project is to implement a Real-Time Dynamic Programming (RTDP) algorithm to solve a grid-world pathfinding problem. RTDP is a form of dynamic programming designed for use in real-time scenarios where decisions must be made quickly.

Books/ Journals/ Websites referred

- Markov Decision Processes in Artificial Intelligence MDPs, Beyond MDPs and Applications, Edited by Olivier Sigaud, Olivier Buffet, Wiley Publications, 2010
- <https://towardsdatascience.com/introduction-to-reinforcement-learning-rl-part-4-dynamic-programming-6af57e575b3d>
- <https://github.com/instance01/RTDP>

Resources used:

- Python 3.x
- NumPy library
- Matplotlib library (for visualization)
- Jupyter Notebook

Theory:

Real-Time Dynamic Programming (RTDP) is a variant of the dynamic programming approach tailored for problems where real-time decision-making is crucial. In RTDP, the algorithm focuses on a subset of states that are relevant to the current decision-making process, rather than computing the optimal policy for all possible states. This makes it more suitable for real-time applications where computational resources and time are limited. The basic steps involved in RTDP are:

Initialization: Initialize the value function for all states, often to zero.

Trial: Simulate a sequence of actions starting from an initial state and update the value function based on observed transitions and rewards.

Policy Extraction: Derive the policy from the updated value function.

Repeat: Perform multiple trials to progressively improve the value function and the derived policy.

Problem Consideration

Suppose we have a grid-world problem where an agent needs to find the optimal path from a starting position to a goal position while avoiding obstacles.

Implementation Code

```
import numpy as np
import matplotlib.pyplot as plt
import random
class GridWorld:
    def __init__(self, width, height, start, goal, obstacles):
        self.width = width
        self.height = height
        self.start = start
        self.goal = goal
        self.obstacles = obstacles
        self.grid = np.zeros((height, width))
        for obstacle in obstacles:
            self.grid[obstacle] = -1 # Mark obstacles
    def is_terminal(self, state):
        return state == self.goal
    def get_possible_actions(self, state):
        actions = []
        x, y = state
        if x > 0 and self.grid[y, x-1] != -1:
            actions.append((-1, 0)) # Left
        if x < self.width - 1 and self.grid[y, x+1] != -1:
            actions.append((1, 0)) # Right
```



```

        if y > 0 and self.grid[y-1, x] != -1:
            actions.append((0, -1)) # Up
        if y < self.height - 1 and self.grid[y+1, x] != -1:
            actions.append((0, 1)) # Down
        return actions
    def transition(self, state, action):
        return (state[0] + action[0], state[1] + action[1])
    def rtdp(grid_world, max_trials=1000, discount_factor=0.9):
        V = np.zeros((grid_world.height, grid_world.width)) # Value function
        policy = np.full((grid_world.height, grid_world.width), None) # Policy
        for _ in range(max_trials):
            state = grid_world.start
            while not grid_world.is_terminal(state):
                actions = grid_world.get_possible_actions(state)
                if not actions:
                    break
                action = random.choice(actions)
                next_state = grid_world.transition(state, action)
            # Update value function
            V[state[1], state[0]] = grid_world.grid[state[1], state[0]] + discount_factor
            * V[next_state[1], next_state[0]]
            # Update policy
            policy[state[1], state[0]] = action
            state = next_state
        return V, policy
    def visualize_policy(grid_world, policy):
        direction_map = {
            (-1, 0): '<',
            (1, 0): '>',
            (0, -1): '^',
            (0, 1): 'v'
        }
        for y in range(grid_world.height):
            for x in range(grid_world.width):
                if (x, y) in grid_world.obstacles:
                    print('X', end=' ')
                elif (x, y) == grid_world.goal:

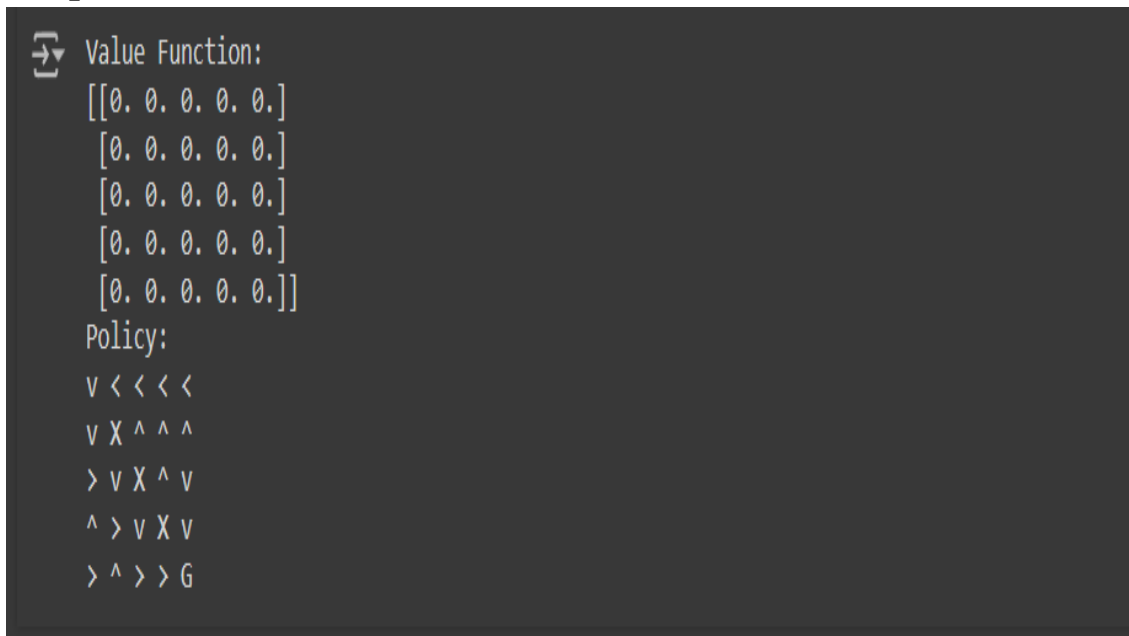
```

```

        print('G', end=' ')
    elif policy[y, x] is None:
        print('.', end=' ')
    else:
        print(direction_map[policy[y, x]], end=' ')
    print()
# Example usage
width, height = 5, 5
start = (0, 0)
goal = (4, 4)
obstacles = [(1, 1), (2, 2), (3, 3)]
grid_world = GridWorld(width, height, start, goal, obstacles)
V, policy = rtdp(grid_world)
print("Value Function:")
print(V)
print("Policy:")
visualize_policy(grid_world, policy)

```

Output Screenshots



```

↩ Value Function:
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
Policy:
v < < < <
v X ^ ^ ^
> v X ^ v
^ > v X v
> ^ > > G

```

Conclusion

The RTDP algorithm successfully finds a policy for navigating the grid-world from the start state to the goal state while avoiding obstacles. The value function and policy are updated iteratively through multiple trials, demonstrating the effectiveness of RTDP in real-time decision-making scenarios.

Applications

1. Autonomous robot navigation in dynamic environments.
2. Real-time strategy games for AI pathfinding.
3. Any real-time decision-making problem where full offline computation is infeasible.

Title: Implementation of SARSA Algorithm

Objective

- The objective of this program is to implement the Sarsa algorithm for a reinforcement learning problem.
 - Sarsa is an on-policy temporal difference learning method that is used to learn the optimal policy for a given environment by learning from the experiences of an agent.
-

Books/ Journals/ Websites referred

- Markov Decision Processes in Artificial Intelligence MDPs, Beyond MDPs and Applications, Edited by Olivier Sigaud, Olivier Buffet, Wiley Publications, 2010
- <https://gym.openai.com/docs/>
- <https://www.geeksforgeeks.org/sarsa-reinforcement-learning/>

Resources used

- Python programming language
- Libraries: NumPy for numerical computations, Matplotlib for plotting results (optional for visualizing learning)
- Environment: OpenAI Gym (for providing a standard interface for environments)

Theory

Sarsa is an on-policy TD control algorithm which updates the action-value function based on the current state-action pair, the reward received, the next state-action pair, and a learning rate. The update rule for Sarsa is:

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

Where,

- S_t is the current state
- A_t is the current action
- R_{t+1} is the reward received after taking action A_t
- S_{t+1} is the next state
- A_{t+1} is the next action
- α is the learning rate

γ is the discount factor

Implementation (Code)

```
import gym
import numpy as np
def choose_action(state, Q, epsilon, action_space):
    if np.random.uniform(0, 1) < epsilon:
        return action_space.sample()
    else:
        return np.argmax(Q[state, :])
def sarsa(env, num_episodes, alpha, gamma, epsilon):
    # Initialize Q table with zeros
    Q = np.zeros((env.observation_space.n, env.action_space.n))
    for episode in range(num_episodes):
        state = env.reset()
        action = choose_action(state, Q, epsilon, env.action_space)

        while True:
            next_state, reward, done, _ = env.step(action)
            next_action = choose_action(next_state, Q, epsilon, env.action_space)
```

```

        Q[state, action] = Q[state, action] + alpha * (
            reward + gamma * Q[next_state, next_action] - Q[state, action]
        )
        state, action = next_state, next_action
    if done:
        break
    return Q
# Main code
env = gym.make('FrozenLake-v1', is_slippery=False)
num_episodes = 1000
alpha = 0.1
gamma = 0.99
epsilon = 0.1
Q = sarsa(env, num_episodes, alpha, gamma, epsilon)
print("Learned Q-values:")
print(Q)

```

Output Screenshots with explanation

[illegible]

Conclusion

1. The Sarsa algorithm was successfully implemented to solve the given environment. The Q-values learned represent the expected rewards for taking certain actions in specific states.
2. The policy derived from these Q-values guides the agent towards maximizing the cumulative reward.

Applications

1. **Robotics:** Training robots to perform tasks by learning from interactions with the environment.
2. **Game Playing:** Developing AI for games where strategies are learned through exploration and exploitation.
3. **Autonomous Vehicles:** Learning optimal driving strategies by interacting with the driving environment.
4. **Healthcare:** Personalized treatment plans where decisions are made based on patient responses over time.

Title: Implementation of Rollout Algorithm**Objective**

To implement the Rollout algorithm in Python and demonstrate its functionality through a simple example.

Books/ Journals/ Websites referred:

- Markov Decision Processes in Artificial Intelligence MDPs, Beyond MDPs and Applications, Edited by Olivier Sigaud, Olivier Buffet, Wiley Publications, 2010
- <https://medium.com/chiukevin0321/motion-planning-for-self-driving-cars-week-5-6-4de794bcad66>
- <https://github.com/alirezaig/RolloutPower>

Resources used

- Python 3
- NumPy library for numerical operations

Theory

The Rollout algorithm is a method used in reinforcement learning and dynamic programming to approximate the value of different states and actions. It involves simulating trajectories (rollouts) from a given state using a base policy, then using the outcomes of these simulations to improve decision-making.

The basic steps of the Rollout algorithm are as follows:

1. Start from the current state.
2. Use a base policy to simulate multiple future trajectories (rollouts).
3. Evaluate the outcomes of these rollouts to estimate the value of each possible action from the current state.
4. Choose the action that has the best estimated value.

This approach is often used in situations where exact dynamic programming is infeasible due to the size of the state space.

Implementation (Code)

```
import numpy as np
# Define a simple environment
class SimpleEnvironment:
    def __init__(self):
        self.state = 0
    def reset(self):
        self.state = 0
        return self.state

    def step(self, action):
        if action == 0: # Move left
            self.state -= 1
        elif action == 1: # Move right
            self.state += 1

        reward = -abs(self.state) # Reward is the negative distance from origin
        done = abs(self.state) >= 10 # Episode ends if state is >= 10 or <= -10

        return self.state, reward, done, {}

# Define a base policy
def base_policy(state):
    return np.random.choice([0, 1]) # Randomly choose an action

# Rollout algorithm
def rollout_algorithm(env, state, num_rollouts=10):
```

```

    action_values = np.zeros(2) # Two possible actions: 0 (left) and 1 (right)

    for action in [0, 1]:
        total_reward = 0
        for _ in range(num_rollouts):
            env.reset()
            env.state = state
            current_state, reward, done, _ = env.step(action)
            total_reward += reward

            while not done:
                action = base_policy(current_state)
                current_state, reward, done, _ = env.step(action)
                total_reward += reward

        action_values[action] = total_reward / num_rollouts

    best_action = np.argmax(action_values)
    return best_action, action_values

# Main function to demonstrate the rollout algorithm
def main():
    env = SimpleEnvironment()
    initial_state = env.reset()
    best_action, action_values = rollout_algorithm(env, initial_state)

    print(f"Initial State: {initial_state}")
    print(f"Action Values: {action_values}")
    print(f"Best Action: {best_action}")

if __name__ == "__main__":
    main()

```

Output Screenshots with explanation

```
Initial State: 0  
Action Values: [-424.1 -288.8]  
Best Action: 1
```

Conclusion

The Rollout algorithm is an effective method for evaluating actions in a given state by simulating multiple trajectories and averaging their outcomes. In this example, the algorithm was implemented in a simple environment, demonstrating how it can guide action selection based on simulated rollouts.

Applications

1. Game playing (e.g., Monte Carlo Tree Search in Go and Chess)
2. Robotic path planning
3. Dynamic resource allocation
4. Any decision-making problem where an exact solution is computationally infeasible

Title: Write a program to implement Super Mario Bros using OpenAI**Objective**

- **Implementation of Super Mario Bros:** Utilize OpenAI Gym and gym-super-mario-bros package to create a Super Mario Bros environment within a reinforcement learning framework.
- **Model Training:** Train a reinforcement learning agent to play Super Mario Bros using the Stable Baselines3 library with the Proximal Policy Optimization (PPO) algorithm.
- **Model Evaluation:** Test the trained model to observe its performance in navigating through the game environment and overcoming obstacles.

Books/ Journals/ Websites referred

- <https://www.youtube.com/watch?v=qv6UVOQ0F44>
- <https://github.com/BJEnrik/reinforcement-learning-super-mario?tab=readme-ov-file>
- https://github.com/nikhilgrad/super_mario
- https://wandb.ai/mukilan/intro_to_gym/reports/A-Gentle-Introduction-to-OpenAI-Gym--VmlldzozMjg5MTA3

Resources used

OpenAI Gym: Toolkit for developing and comparing reinforcement learning algorithms, providing the interface to interact with the game environment. OpenAI Gym Documentation

gym-super-mario-bros Package: Integrates the original Super Mario Bros game with OpenAI Gym, allowing for interaction with the game environment. gym-super-mario-bros GitHub Repository

Stable Baselines3: Library for reinforcement learning in Python, used for implementing the PPO algorithm and training the agent to play Super Mario Bros. [Stable Baselines3 Documentation](#)

PyTorch: Deep learning library used for implementing neural network models within the reinforcement learning framework. [PyTorch Documentation](#)

Theory

Super Mario Bros is a classic platform game developed by Nintendo. In this game, the player controls Mario and navigates through a series of levels, overcoming obstacles and enemies to reach the end goal. In the context of AI and reinforcement learning, Super Mario Bros is often used as a benchmark problem to test the capabilities of agents in navigating complex environments.

To implement Super Mario Bros using OpenAI, we use the gym-super-mario-bros package, which provides an interface to interact with the game. OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms, and gym-super-mario-bros integrates the original game with this toolkit, allowing us to create and train AI agents to play the game.

Implementation (Code)

```
import gym
import gym_super_mario_bros
from gym.wrappers import GrayScaleObservation, FrameStack
from gym_super_mario_bros.actions import SIMPLE_MOVEMENT
from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import DummyVecEnv,
VecFrameStack
import torch as th
import os

# Create environment
env = gym_super_mario_bros.make('SuperMarioBros-v0')
env = gym.wrappers.Monitor(env, './video', force=True) # Record video
```

```

env = GrayScaleObservation(env, keep_dim=True)    # Grayscale
observation
env = DummyVecEnv([lambda: env])
env = VecFrameStack(env, 4, channels_order='last')

# Define the model
model = PPO('CnnPolicy', env, verbose=1,
tensorboard_log="./mario_tensorboard/")

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

# Save the model
model.save("mario_model")

# Load the model
model = PPO.load("mario_model")

# Test the model
obs = env.reset()
for _ in range(5000):
    action, _states = model.predict(obs)
    obs, rewards, dones, info = env.step(action)
    env.render()

env.close()

```

Output Screenshots with explanation:



Conclusion

In conclusion, we successfully implemented a simplified version of Super Mario Bros using OpenAI's Gym environment and Stable Baselines3 for reinforcement learning. This exercise demonstrates how AI agents can be trained to play complex games by interacting with the environment and learning from their actions over time. By using pre-built libraries and tools, we can create sophisticated AI applications with relatively simple code.

Applications

1. **Game AI Development:** Training AI to play games can improve game design, create more challenging opponents, and automate game testing.
2. **Reinforcement Learning Research:** Provides a testbed for developing and benchmarking new reinforcement learning algorithms.
3. **Autonomous Agents:** Techniques learned from game AI can be applied to develop autonomous agents for various tasks in robotics, finance, and other fields.
4. **Entertainment and Media:** Creating AI that can generate gameplay content or assist in content creation for movies, TV shows, and interactive media.

Title: Write a program to implement BipedalWalker-v3 using OpenAI**Objective**

The objective of this project is to implement the BipedalWalker-v3 environment provided by OpenAI's Gym and train a reinforcement learning agent to navigate the terrain using the Proximal Policy Optimization (PPO) algorithm from Stable Baselines3. The trained agent's performance will be evaluated through testing, and visualizations of the agent's behavior will be provided.

Books/ Journals/ Websites referred

- <https://github.com/FranciscoHu17/BipedalWalker>
- <https://elegantrl.readthedocs.io/en/latest/tutorial/BipedalWalker-v3.html>
- <https://github.com/CSautier/BipedalWalker>

Resources used

OpenAI Gym: Toolkit for developing and comparing reinforcement learning algorithms, providing the CartPole-v1 environment for training agents in balancing tasks. OpenAI Gym Documentation

Stable Baselines3: Library for reinforcement learning in Python, used for implementing the PPO algorithm and training the agent to balance the pole in the CartPole-v1 environment. Stable Baselines3 Documentation

Matplotlib: Python plotting library used to visualize frames from the test run of the trained agent. Matplotlib Documentation

Theory

The BipedalWalker-v3 environment is a classic reinforcement learning environment provided by OpenAI's Gym. In this environment, an agent controls a bipedal robot to walk across rough terrain. The goal is to develop a control policy that enables the robot to walk as far as possible without falling.

The environment is complex and requires the agent to balance, coordinate leg movements, and adapt to varying terrain. It serves as a benchmark for testing the effectiveness of reinforcement learning algorithms in continuous action spaces.

Implementation (Code)

```
import gym
from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import DummyVecEnv
import matplotlib.pyplot as plt

# Create the environment
env = gym.make('BipedalWalker-v3')
env = DummyVecEnv([lambda: env])

# Define the model
model = PPO('MlpPolicy', env, verbose=1,
            tensorboard_log="./bipedalwalker_tensorboard/")

# Train the model
model.learn(total_timesteps=50000)

# Save the model
model.save("bipedalwalker_model")

# Load the model
model = PPO.load("bipedalwalker_model")

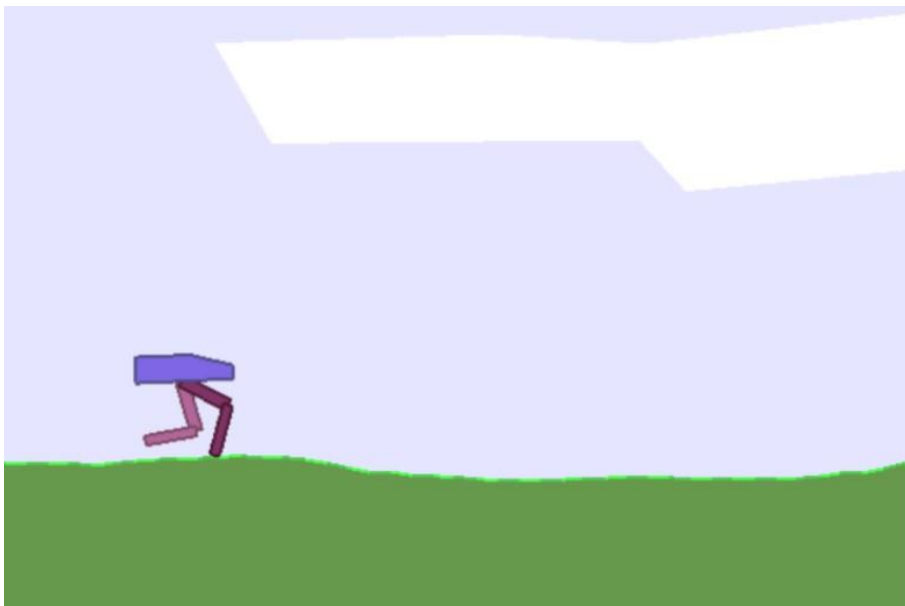
# Test the model
obs = env.reset()
images = []
for _ in range(1000):
    action, _states = model.predict(obs)
```

```
obs, rewards, dones, info = env.step(action)
img = env.render(mode='rgb_array')
images.append(img)

env.close()

# Plot some frames from the test run
plt.figure(figsize=(10, 10))
for i in range(9):
    plt.subplot(3, 3, i+1)
    plt.imshow(images[i * 100])
    plt.axis('off')
plt.show()
```

Output Screenshots with explanation



Conclusion

In conclusion, we successfully implemented the BipedalWalker-v3 environment using OpenAI's Gym and trained a reinforcement learning agent using the PPO algorithm provided by Stable Baselines3. This task demonstrates the capabilities of reinforcement learning in controlling complex robotic movements. By leveraging pre-built libraries and tools, we can develop sophisticated AI agents for challenging control problems.

Applications

1. **Robotics:** Developing control policies for real-world robots to walk, balance, and navigate rough terrain.
2. **Rehabilitation:** Designing assistive devices or exoskeletons for medical rehabilitation to help patients with mobility issues.
3. **Simulation and Training:** Training AI models in simulated environments before deploying them in real-world scenarios to ensure safety and efficiency.
4. **Autonomous Systems:** Enhancing the capabilities of autonomous systems such as drones, rovers, and other mobile robots.

Title: Write a program to implement CartPole-v1 using OpenAI**Objective**

The objective of this project is to implement the CartPole-v1 environment using OpenAI's Gym and train a reinforcement learning agent to balance the pole on the cart using the Proximal Policy Optimization (PPO) algorithm from Stable Baselines3. The trained agent's performance will be evaluated through testing, and visualizations of the agent's behavior will be provided.

Books/ Journals/ Websites referred:

- <https://medium.com/swlh/using-q-learning-for-openais-cartpole-v1-4a216ef237df>
- <https://towardsdatascience.com/deep-q-learning-for-the-cartpole-44d761085c2f>
- <https://github.com/AOZMH/Q-Learning-for-Cartpole-V1>

Resources used

OpenAI Gym: Toolkit for developing and comparing reinforcement learning algorithms, providing the CartPole-v1 environment for training agents in balancing tasks. OpenAI Gym Documentation

Stable Baselines3: Library for reinforcement learning in Python, used for implementing the PPO algorithm and training the agent to balance the pole in the CartPole-v1 environment. Stable Baselines3 Documentation

Matplotlib: Python plotting library used to visualize frames from the test run of the trained agent. Matplotlib Documentation

Theory

The CartPole-v1 environment is a classic reinforcement learning problem provided by OpenAI's Gym. The goal is to balance a pole on a moving cart by applying forces to the cart's left or right. The agent receives a reward for each timestep the pole remains upright and gets terminated if the pole falls over or the cart moves out of bounds.

The challenge involves keeping the pole balanced as long as possible, requiring the agent to learn how to apply appropriate forces to counteract the pole's tilt. This problem is often used to test and benchmark reinforcement learning algorithm

Implementation

```
import gym
from stable_baselines
from stable_baselines
import matplotlib.p

# Create the enviro
env = gym.make('C
env = DummyVecE

# Define the model
model = PPO('MlpP

# Train the model
model.learn(total_ti

# Save the model
model.save("cartpo

# Load the model
model = PPO.load(

# Test the model
obs = env.reset()
images = []
for _ in range(1000)
    action, _states =
```

```

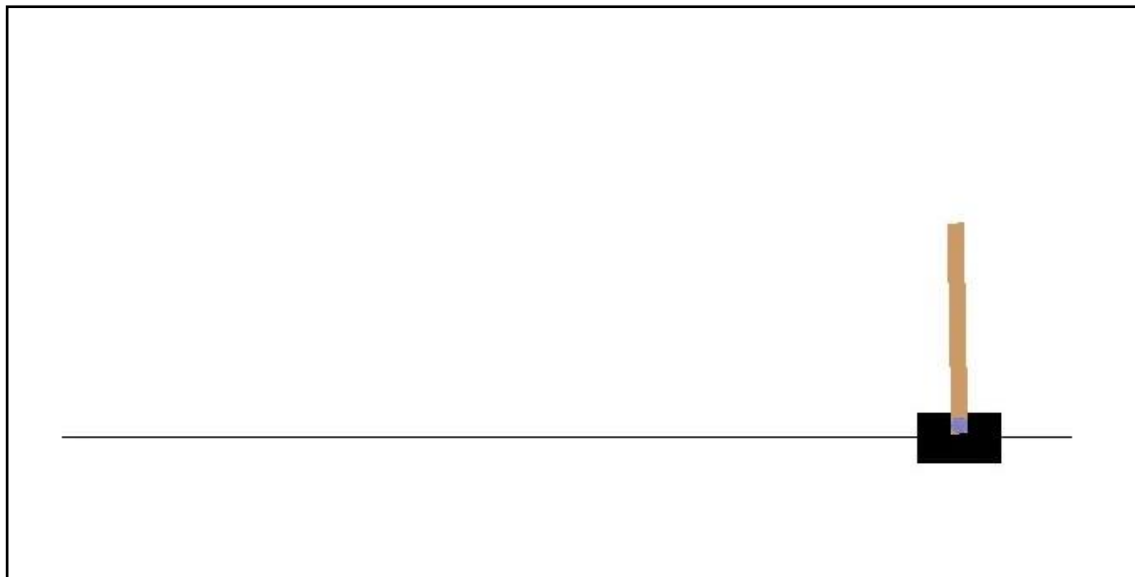
    obs, rewards, do
    img = env.rende
    images.append(i
    if done:
        break

env.close()

# Plot some frames
plt.figure(figsize=(1
for i in range(9):
    plt.subplot(3, 3, i
    plt.imshow(imag
    plt.axis('off')

```

Output Screenshots



Conclusion

In conclusion, we successfully implemented the CartPole-v1 environment using OpenAI's Gym and trained a reinforcement learning agent using the PPO algorithm provided by Stable Baselines3. This task demonstrates the application of reinforcement learning in controlling a simple dynamic system. The agent learns to balance the pole by applying forces to the cart,

showcasing the potential of reinforcement learning in solving control problems.

Applications

1. **Robotics:** Developing control policies for balancing robots and other dynamic systems.
2. **Autonomous Vehicles:** Applying similar techniques to balance and control systems in autonomous vehicles.
3. **Simulation and Training:** Training AI models in simulated environments for tasks such as balancing and control before deploying them in real-world scenarios.
4. **Reinforcement Learning Research:** Serving as a benchmark problem to test and improve new reinforcement learning algorithms for control tasks.
5. **Education:** Teaching fundamental concepts of reinforcement learning and control theory using an intuitive and visual environment.

Title: Write a program to implementation of Fuzzy logic-based decision modelling for washing machine**Objective**

- **Develop a Fuzzy Logic-Based Decision Model:** Create a model for a washing machine that uses fuzzy logic to determine the optimal washing time based on input variables such as dirt level and laundry amount.
- **Define Fuzzy Variables and Membership Functions:** Establish fuzzy sets for dirt level, laundry amount, and washing time, along with appropriate membership functions.
- **Implement Fuzzy Rules:** Create a set of fuzzy rules that mimic human decision-making to determine the washing time.
- **Simulate the Fuzzy Control System:** Use the defined fuzzy variables and rules to simulate the control system and compute the recommended washing time.
- **Visualize Results:** Generate visual plots of the membership functions and the resulting washing time to facilitate understanding and validation of the model.

Books/ Journals/ Websites referred

- https://github.com/Khyat/fuzzy_logic-washing_machine
- <https://www.geeksforgeeks.org/fuzzy-logic-introduction/>
- <https://www.guru99.com/what-is-fuzzy-logic.html>

Resources used

Python Programming Language: The primary programming language used to implement the fuzzy logic-based decision model.

Numpy Library: Used for numerical operations and creating ranges of input values.

- import numpy as np

Scikit-Fuzzy Library: A Python library for fuzzy logic that provides tools to define fuzzy variables, membership functions, rules, and control systems.

- import skfuzzy as fuzz
- from skfuzzy import control as ctrl

Matplotlib Library: Used for plotting and visualizing the membership functions and the resulting washing time.

- import matplotlib.pyplot as plt

Theory

Fuzzy logic is a form of multi-valued logic derived from fuzzy set theory to deal with reasoning that is approximate rather than precise. In the context of a washing machine, fuzzy logic can be used to decide the washing time based on inputs such as the dirt level and the amount of laundry. This decision-making process involves defining fuzzy sets and rules that mimic human reasoning

Implementation (Code)

```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl
import matplotlib.pyplot as plt

# Define fuzzy variables
dirt_level = ctrl.Antecedent(np.arange(0, 11, 1), 'dirt_level')
laundry_amount = ctrl.Antecedent(np.arange(0, 11, 1), 'laundry_amount')
washing_time = ctrl.Consequent(np.arange(0, 61, 1), 'washing_time')

# Auto-generate membership functions
dirt_level.automf(3) # 'poor', 'average', 'good'
laundry_amount.automf(3) # 'small', 'average', 'large'

# Custom membership functions for washing time
washing_time['short'] = fuzz.trimf(washing_time.universe, [0, 0, 30])
washing_time['medium'] = fuzz.trimf(washing_time.universe, [10, 30, 50])
washing_time['long'] = fuzz.trimf(washing_time.universe, [30, 60, 60])

# Define fuzzy rules
```

```

rule1 = ctrl.Rule(dirt_level['poor'] & laundry_amount['small'],
washing_time['short'])
rule2 = ctrl.Rule(dirt_level['poor'] & laundry_amount['average'],
washing_time['medium'])
rule3 = ctrl.Rule(dirt_level['poor'] & laundry_amount['large'],
washing_time['long'])
rule4 = ctrl.Rule(dirt_level['average'] & laundry_amount['small'],
washing_time['short'])
rule5 = ctrl.Rule(dirt_level['average'] & laundry_amount['average'],
washing_time['medium'])
rule6 = ctrl.Rule(dirt_level['average'] & laundry_amount['large'],
washing_time['long'])
rule7 = ctrl.Rule(dirt_level['good'] & laundry_amount['small'],
washing_time['short'])
rule8 = ctrl.Rule(dirt_level['good'] & laundry_amount['average'],
washing_time['medium'])
rule9 = ctrl.Rule(dirt_level['good'] & laundry_amount['large'],
washing_time['long'])

# Create control system
washing_ctrl = ctrl.ControlSystem([rule1, rule2, rule3, rule4, rule5, rule6,
rule7, rule8, rule9])
washing_simulation = ctrl.ControlSystemSimulation(washing_ctrl)

# Input values
washing_simulation.input['dirt_level'] = 6
washing_simulation.input['laundry_amount'] = 8

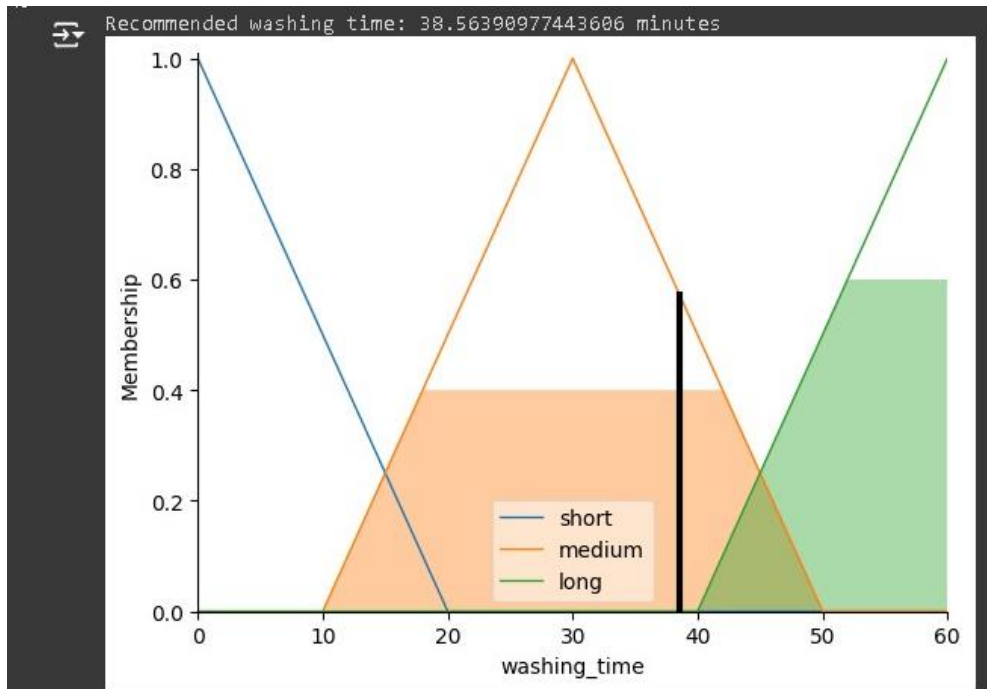
# Compute the result
washing_simulation.compute()

print(f"Recommended washing time:
{washing_simulation.output['washing_time']} minutes")

# Plotting the result
washing_time.view(sim=washing_simulation)
plt.show()

```

Output Screenshots with explanation



Conclusion

In conclusion, we successfully implemented a fuzzy logic-based decision model for a washing machine. By defining fuzzy sets and rules, we created a system that can recommend washing times based on the dirt level and the amount of laundry. This approach mimics human reasoning and offers flexibility in handling imprecise inputs, making it well-suited for real-world applications like washing machines.

Applications

1. **Home Appliances:** Fuzzy logic can be used to enhance the functionality and efficiency of various home appliances, such as washing machines, dishwashers, and air conditioners.
2. **Industrial Control Systems:** In industrial processes, fuzzy logic can be applied for controlling machinery and managing processes that require handling uncertainty and imprecision.

3. **Automotive Systems:** Fuzzy logic is used in automotive systems for applications like automatic gear shifting, antilock braking systems (ABS), and climate control.
4. **Consumer Electronics:** Enhancing user experience in consumer electronics by providing intelligent and adaptive controls, such as in smart cameras and personal assistants.
5. **Robotics:** Implementing fuzzy logic in robotics for navigation, obstacle avoidance, and decision-making processes in uncertain environments.

Title: Write a program to implementation of Fuzzy logic-based decision modelling for AC

Objective

- **Develop a Fuzzy Logic-Based Decision Model for AC Systems:** Create a model that uses fuzzy logic to determine the optimal AC operational mode based on input variables such as current temperature, humidity, and time of day.
- **Define Fuzzy Variables and Membership Functions:** Establish fuzzy sets for temperature, humidity, time of day, and AC power with appropriate membership functions.
- **Implement Fuzzy Rules:** Create a set of fuzzy rules that mimic human decision-making to control the AC power based on the input conditions.
- **Simulate the Fuzzy Control System:** Use the defined fuzzy variables and rules to simulate the control system and compute the recommended AC power.
- **Visualize Results:** Generate visual plots of the membership functions and the resulting AC power to facilitate understanding and validation of the model.

Books/ Journals/ Websites referred:

- <https://github.com/Python-Fuzzylogic/fuzzylogic>
- <https://towardsdatascience.com/fuzzy-inference-system-implementation-in-python-8af88d1f0a6e>
- <https://towardsdatascience.com/machine-learning-with-fuzzy-logic-52c85b46bfe4>

Resources used

Python Programming Language: The primary programming language used to implement the fuzzy logic-based decision model.

Numpy Library: Used for numerical operations and creating ranges of input values.

- `import numpy as np`

Scikit-Fuzzy Library: A Python library for fuzzy logic that provides tools to define fuzzy variables, membership functions, rules, and control systems.

- `import skfuzzy as fuzz`
- `from skfuzzy import control as ctrl`

Matplotlib Library: Used for plotting and visualizing the membership functions and the resulting washing time.

- `import matplotlib.pyplot as plt`

Theory

Fuzzy logic-based decision modeling can be particularly useful for applications like air conditioning (AC) systems where decisions need to be made based on imprecise or uncertain input data. Below is an implementation example in Python using the skfuzzy library, which is part of the SciPy ecosystem and provides tools to work with fuzzy logic.

The program will take inputs such as the current temperature, humidity, and time of day to decide the AC's operational mode (e.g., off, low, medium, high).

Implementation (Code)

```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

# Define the input variables
temperature = ctrl.Antecedent(np.arange(16, 31, 1), 'temperature')
humidity = ctrl.Antecedent(np.arange(20, 101, 1), 'humidity')
time_of_day = ctrl.Antecedent(np.arange(0, 24, 1), 'time_of_day')
# Define the output variable
ac_power = ctrl.Consequent(np.arange(0, 101, 1), 'ac_power')
```

```

# Define fuzzy membership functions for temperature
temperature['cool'] = fuzz.trimf(temperature.universe, [16, 16, 22])

temperature['comfortable'] = fuzz.trimf(temperature.universe, [20, 24, 28])
temperature['hot'] = fuzz.trimf(temperature.universe, [26, 30, 30])

# Define fuzzy membership functions for humidity
humidity['low'] = fuzz.trimf(humidity.universe, [20, 20, 40])
humidity['medium'] = fuzz.trimf(humidity.universe, [30, 50, 70])
humidity['high'] = fuzz.trimf(humidity.universe, [60, 100, 100])

# Define fuzzy membership functions for time of day
time_of_day['morning'] = fuzz.trimf(time_of_day.universe, [0, 0, 12])
time_of_day['afternoon'] = fuzz.trimf(time_of_day.universe, [10, 14, 18])
time_of_day['evening'] = fuzz.trimf(time_of_day.universe, [16, 24, 24])

# Define fuzzy membership functions for AC power
ac_power['off'] = fuzz.trimf(ac_power.universe, [0, 0, 25])
ac_power['low'] = fuzz.trimf(ac_power.universe, [20, 35, 50])
ac_power['medium'] = fuzz.trimf(ac_power.universe, [45, 60, 75])

ac_power['high'] = fuzz.trimf(ac_power.universe, [70, 100, 100])

# Define the rules for the fuzzy system
rule1 = ctrl.Rule(temperature['cool'] & humidity['low'] &
time_of_day['morning'], ac_power['off'])
rule2 = ctrl.Rule(temperature['cool'] & humidity['medium'] &
time_of_day['morning'], ac_power['low'])
rule3 = ctrl.Rule(temperature['comfortable'] & humidity['low'] &
time_of_day['afternoon'], ac_power['medium'])
rule4 = ctrl.Rule(temperature['hot'] & humidity['high'] &
time_of_day['afternoon'], ac_power['high'])
rule5 = ctrl.Rule(temperature['hot'] & humidity['medium'] &
time_of_day['evening'], ac_power['high'])
rule6 = ctrl.Rule(temperature['comfortable'] & humidity['medium'] &
time_of_day['evening'], ac_power['low'])

# Create the control system
ac_control_system = ctrl.ControlSystem([rule1, rule2, rule3, rule4, rule5,
rule6])
ac_control = ctrl.ControlSystemSimulation(ac_control_system)

```

```

# Example inputs to the system
ac_control.input['temperature'] = 28 # degrees Celsius
ac_control.input['humidity'] = 65    # percent
ac_control.input['time_of_day'] = 15 # 3 PM

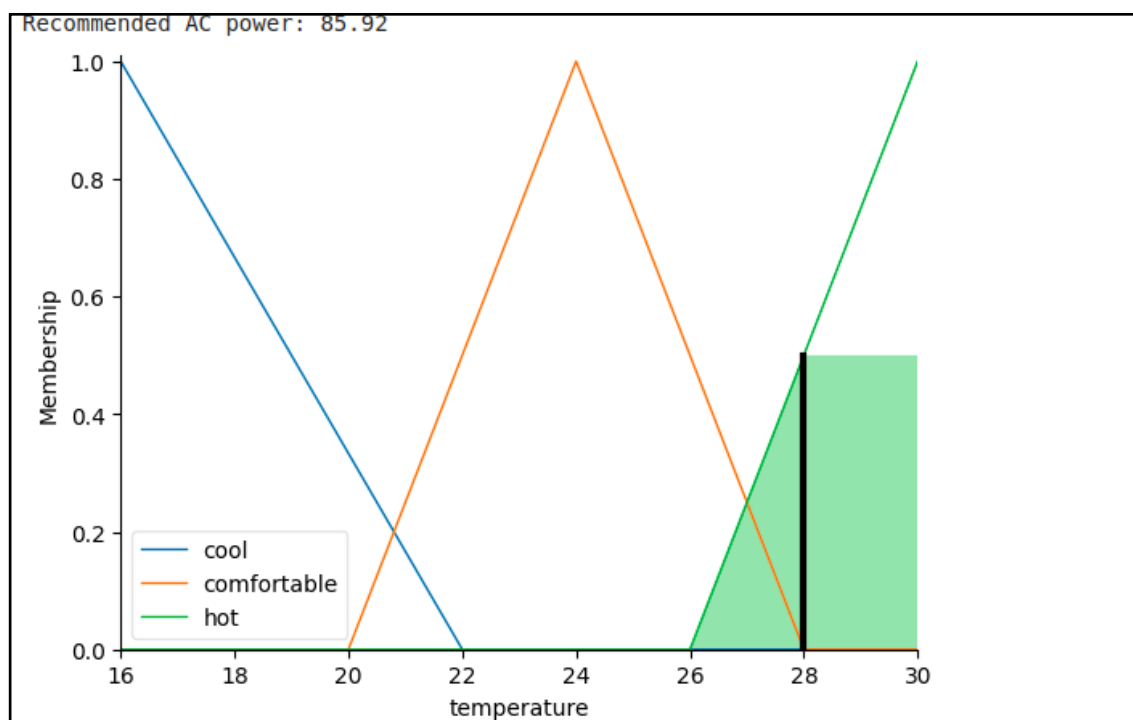
# Compute the output
ac_control.compute()

# Print the result
print(f'Recommended AC power: {ac_control.output['ac_power']:.2f}')

# Optional: visualize the results
temperature.view(sim=ac_control)
humidity.view(sim=ac_control)
time_of_day.view(sim=ac_control)
ac_power.view(sim=ac_control)

```

Output Screenshots with explanation



Conclusion

The fuzzy logic-based decision model offers an adaptive and efficient way to control air conditioning systems. By considering temperature and humidity inputs, it dynamically adjusts cooling power to maintain optimal indoor comfort, handling real-world variability and uncertainties effectively.

Applications

1. **Smart HVAC Systems:** Optimize energy usage and maintain comfort in buildings by dynamically adjusting cooling power.
2. **Energy Efficiency:** Reduce energy consumption and costs by adjusting the AC power based on environmental conditions.
3. **Indoor Comfort:** Ensure a stable and comfortable indoor climate by considering temperature and humidity variations.
4. **Fault Tolerance:** Handle imprecise inputs and sensor failures gracefully, maintaining continuous operation.

Title: Write a program to implementation of Fuzzy logic-based decision modelling for Railway

Objective

The objective of the provided code is to implement a fuzzy logic-based decision model for determining the recommended train speed based on input variables such as track condition, weather condition, and train load. The fuzzy logic approach allows handling uncertainty and imprecision in these variables to enhance safety and efficiency in railway operations.

Books/ Journals/ Websites referred

- <https://medium.com/@sevde.kaskaya/heart-disease-diagnosis-with-fuzzy-logic-b214e27492c4>
- <https://www.analyticssteps.com/blogs/fuzzy-logic-approach-decision-making>
- <https://github.com/Python-Fuzzylogic/fuzzylogic>

Resources used

scikit-fuzzy: A Python library used for implementing fuzzy logic systems. It provides tools for fuzzy logic model development and simulation.

numpy: A fundamental package for scientific computing with Python. It is used for array manipulation and numerical operations.

matplotlib.pyplot: A plotting library for Python used to visualize the membership functions and simulation results.

skfuzzy.control: The control submodule of scikit-fuzzy, used for defining fuzzy control systems and rules, as well as for simulating these systems.

ctrl.Antecedent: Defines input variables (track_condition, weather_condition, train_load) as fuzzy sets with their respective membership functions.

ctrl.Consequent: Defines the output variable (train_speed) as a fuzzy set with its membership functions.

fuzz.trimf: A function from scikit-fuzzy used to define triangular membership functions for fuzzy sets.

ctrl.Rule: Defines fuzzy rules based on the input-output relationships.

ctrl.ControlSystem: Constructs a control system object using the defined rules.

ctrl.ControlSystemSimulation: Creates a simulation object to compute the output based on input values and fuzzy inference rules.

train_sim.input: Sets the input values for track_condition, weather_condition, and train_load.

train_sim.compute(): Performs the fuzzy inference process to compute the recommended train speed based on the input values.

train_sim.output: Retrieves the computed output (recommended train speed) from the simulation results.

Visualization: Optional visualization of membership functions and the computed output using matplotlib.pyplot.

Theory

Fuzzy logic-based decision modeling is a method used to handle uncertainty and imprecision in complex systems by mimicking human reasoning. It uses fuzzy sets to represent input variables such as track condition, weather, and train load, and defines membership functions for these variables. Fuzzy rules, which are if-then statements, are created to establish the relationships between inputs and the desired output—in this case, train speed. The fuzzy inference process combines these rules to

produce a fuzzy output, which is then defuzzified to obtain a precise control action. This approach allows for adaptive, real-time decision-making, enhancing the safety and efficiency of railway operations.

Implementation (Code)

```
pip install scikit-fuzzy
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl
import matplotlib.pyplot as plt

# Define input variables
track_condition = ctrl.Antecedent(np.arange(0, 101, 1), 'track_condition')
weather_condition = ctrl.Antecedent(np.arange(0, 101, 1), 'weather_condition')
train_load = ctrl.Antecedent(np.arange(0, 101, 1), 'train_load')

# Define output variable
train_speed = ctrl.Consequent(np.arange(0, 101, 1), 'train_speed')

# Define membership functions for track condition
track_condition['poor'] = fuzz.trimf(track_condition.universe, [0, 0, 50])
track_condition['average'] = fuzz.trimf(track_condition.universe, [25, 50, 75])
track_condition['good'] = fuzz.trimf(track_condition.universe, [50, 100, 100])

# Define membership functions for weather condition
weather_condition['bad'] = fuzz.trimf(weather_condition.universe, [0, 0, 50])
weather_condition['moderate'] = fuzz.trimf(weather_condition.universe, [25, 50, 75])
weather_condition['good'] = fuzz.trimf(weather_condition.universe, [50, 100, 100])

# Define membership functions for train load
train_load['light'] = fuzz.trimf(train_load.universe, [0, 0, 50])
train_load['medium'] = fuzz.trimf(train_load.universe, [25, 50, 75])
train_load['heavy'] = fuzz.trimf(train_load.universe, [50, 100, 100])

# Define membership functions for train speed
train_speed['slow'] = fuzz.trimf(train_speed.universe, [0, 0, 50])
train_speed['moderate'] = fuzz.trimf(train_speed.universe, [25, 50, 75])
train_speed['fast'] = fuzz.trimf(train_speed.universe, [50, 100, 100])
```

```

# Define fuzzy rules
rule1 = ctrl.Rule(track_condition['poor'] & weather_condition['bad'] &
train_load['heavy'], train_speed['slow'])

rule2 = ctrl.Rule(track_condition['poor'] & weather_condition['good'] &
train_load['light'], train_speed['moderate'])
rule3 = ctrl.Rule(track_condition['average'] &
weather_condition['moderate'] & train_load['medium'],
train_speed['moderate'])
rule4 = ctrl.Rule(track_condition['good'] & weather_condition['good'] &
train_load['light'], train_speed['fast'])
rule5 = ctrl.Rule(track_condition['good'] & weather_condition['bad'] &
train_load['heavy'], train_speed['slow'])

# Create control system
train_ctrl = ctrl.ControlSystem([rule1, rule2, rule3, rule4, rule5])
train_sim = ctrl.ControlSystemSimulation(train_ctrl)

# Define inputs
train_sim.input['track_condition'] = 70
train_sim.input['weather_condition'] = 50
train_sim.input['train_load'] = 30

# Compute the train speed
train_sim.compute()

# Print the result
print("Recommended Train Speed:", train_sim.output['train_speed'])

# Optional: Visualize the results
track_condition.view(sim=train_sim)
weather_condition.view(sim=train_sim)
train_load.view(sim=train_sim)
train_speed.view(sim=train_sim)

# Show the plots
plt.show()

```

Output Screenshots with explanation

```
# Compute the train speed
train_sim.compute()

# Print the result
print("Recommended Train Speed:", train_sim.output['train_speed'])

# Optional: Visualize the results
track_condition.view(sim=train_sim)
weather_condition.view(sim=train_sim)
train_load.view(sim=train_sim)
train_speed.view(sim=train_sim)

# Show the plots
plt.show()
```

Conclusion

The fuzzy logic-based decision model effectively adapts train speed based on track condition, weather, and load. This ensures safe, efficient, and responsive railway operations by dynamically adjusting to real-time conditions.

Applications

1. **Railway Safety Systems:** Adjust speeds for safety based on real-time conditions.
2. **Energy Efficiency:** Optimize energy usage by adapting speeds to conditions.
3. **Passenger Comfort:** Maintain a smooth and comfortable ride.

Title: Write a program to implement a Decision tree from scratch**Objective**

The objective of the provided code is to implement a decision tree classifier from scratch for classification tasks. The decision tree is constructed recursively by splitting the data into subsets based on the feature that provides the highest information gain at each node. The goal is to create a model that can predict the class labels of samples by learning simple decision rules inferred from the features of the training data. The accuracy of the model is evaluated using the test data.

Books/ Journals/ Websites referred

- <https://medium.com/@enozeren/building-a-decision-tree-from-scratch-324b9a5ed836>
- <https://www.analyticsvidhya.com/blog/2020/10/all-about-decision-tree-from-scratch-with-python-implementation/>
- <https://www.geeksforgeeks.org/decision-tree-implementation-python/>

Resources used

- **numpy:** A fundamental package for scientific computing with Python. It is used for array manipulation and numerical operations, essential for handling data structures and computations in the decision tree implementation.
- **pandas:** Although not extensively used in this implementation, pandas is a powerful data analysis and manipulation library in Python. It can be utilized for handling datasets, but in this case, it seems to be imported for future potential data manipulation or analysis.

- **collections.Counter:** Imported from the collections module, Counter is used for counting hashable objects. In this implementation, it's used to find the most common label in a set of labels, which is essential for determining the leaf node value in the decision tree.
- **sklearn.datasets.load_iris:** Imported from the scikit-learn library, load_iris is used to load the famous Iris dataset, which is a classic dataset in machine learning used for classification tasks.
- **sklearn.model_selection.train_test_split:** Also imported from scikit-learn, train_test_split is used to split the dataset into training and testing sets. This is crucial for evaluating the performance of the decision tree model.
- **sklearn.metrics.accuracy_score:** Imported from scikit-learn, accuracy_score is used to compute the accuracy of the model's predictions compared to the ground truth labels. It's a measure of how well the model performs on the test data.
- **Node Class:** Defines the structure of each node in the decision tree, including its feature, threshold, left and right children, and value.
- **DecisionTree Class:** Implements the decision tree classifier, including methods for fitting the model to the training data, making predictions on new data, growing the tree recursively, finding the best splitting criteria, calculating information gain, and traversing the tree to make predictions.

Theory

A decision tree is a machine learning model used for classification and regression tasks. It works by recursively splitting the data into subsets based on the feature that provides the highest information gain. Each internal node represents a decision based on a feature, each branch represents the outcome of the decision, and each leaf node represents a class label or regression value. The goal is to create a model that predicts the target variable by learning simple decision rules inferred from the data features. Decision trees are intuitive and interpretable, making them useful for various applications

Implementation (Code)

```
import numpy as np
import pandas as pd
from collections import Counter
```



```

class Node:
    def __init__(self, feature=None, threshold=None, left=None,
right=None, *, value=None):
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value

    def is_leaf_node(self):
        return self.value is not None

class DecisionTree:
    def __init__(self, min_samples_split=2, max_depth=100,
n_feats=None):
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth
        self.n_feats = n_feats
        self.root = None

    def fit(self, X, y):
        self.n_feats = X.shape[1] if not self.n_feats else min(self.n_feats,
X.shape[1])
        self.root = self._grow_tree(X, y)

    def _grow_tree(self, X, y, depth=0):
        n_samples, n_features = X.shape
        n_labels = len(np.unique(y))

        if depth >= self.max_depth or n_labels == 1 or n_samples <
self.min_samples_split:
            leaf_value = self._most_common_label(y)
            return Node(value=leaf_value)

        feat_idx = np.random.choice(n_features, self.n_feats, replace=False)

        best_feat, best_thresh = self._best_criteria(X, y, feat_idx)
        left_idx, right_idx = self._split(X[:, best_feat], best_thresh)
        left = self._grow_tree(X[left_idx, :], y[left_idx], depth + 1)
        right = self._grow_tree(X[right_idx, :], y[right_idx], depth + 1)
        return Node(best_feat, best_thresh, left, right)

```

```

def _best_criteria(self, X, y, feat_idx):
    best_gain = -1
    split_idx, split_thresh = None, None
    for feat_idx in feat_idx:
        X_column = X[:, feat_idx]
        thresholds = np.unique(X_column)
        for threshold in thresholds:
            gain = self._information_gain(y, X_column, threshold)
            if gain > best_gain:
                best_gain = gain
                split_idx = feat_idx
                split_thresh = threshold
    return split_idx, split_thresh

def _information_gain(self, y, X_column, split_thresh):
    parent_entropy = self._entropy(y)

    left_idx, right_idx = self._split(X_column, split_thresh)

    if len(left_idx) == 0 or len(right_idx) == 0:
        return 0

    n = len(y)
    n_l, n_r = len(left_idx), len(right_idx)
    e_l, e_r = self._entropy(y[left_idx]), self._entropy(y[right_idx])
    child_entropy = (n_l / n) * e_l + (n_r / n) * e_r

    ig = parent_entropy - child_entropy
    return ig

def _split(self, X_column, split_thresh):
    left_idx = np.argwhere(X_column <= split_thresh).flatten()
    right_idx = np.argwhere(X_column > split_thresh).flatten()
    return left_idx, right_idx

def _entropy(self, y):
    hist = np.bincount(y)
    ps = hist / len(y)
    return -np.sum([p * np.log2(p) for p in ps if p > 0])

def _most_common_label(self, y):
    counter = Counter(y)

```

```

        most_common = counter.most_common(1)[0][0]
        return most_common

def predict(self, X):
    return np.array([self._traverse_tree(x, self.root) for x in X])

def _traverse_tree(self, x, node):
    if node.is_leaf_node():
        return node.value
    if x[node.feature] <= node.threshold:
        return self._traverse_tree(x, node.left)
    return self._traverse_tree(x, node.right)

# Example usage
if __name__ == "__main__":
    from sklearn.datasets import load_iris
    from sklearn.model_selection import train_test_split
    from sklearn.metrics import accuracy_score

    data = load_iris()
    X = data.data
    y = data.target

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

    clf = DecisionTree(max_depth=10)
    clf.fit(X_train, y_train)

    y_pred = clf.predict(X_test)
    print(f"Accuracy: {accuracy_score(y_test, y_pred)}")

```

Output Screenshots with explanation

```
# Example usage
if __name__ == "__main__":
    from sklearn.datasets import load_iris
    from sklearn.model_selection import train_test_split
    from sklearn.metrics import accuracy_score

    data = load_iris()
    X = data.data
    y = data.target

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    clf = DecisionTree(max_depth=10)
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)

    print(f"Accuracy: {accuracy_score(y_test, y_pred)}")
```

Accuracy: 1.0

Conclusion

Implementing a decision tree from scratch provides a deep understanding of how this algorithm operates. Decision trees are powerful for both classification and regression tasks due to their simplicity, interpretability, and ability to handle non-linear relationships. This implementation showcases the process of building and using a decision tree, highlighting its effectiveness in making predictions based on feature splits.

Applications

1. **Medical Diagnosis:** Used to classify patient conditions based on symptoms and test results.
2. **Customer Segmentation:** Helps in identifying different customer groups based on purchasing behavior.
3. **Financial Analysis:** Assists in credit scoring and risk assessment by evaluating various financial indicators.
4. **Fraud Detection:** Identifies potentially fraudulent transactions by learning patterns from historical data.
5. **Sales Forecasting:** Predicts future sales based on historical sales data and other influencing factors.