Assignment 4

Name : Karim Mohamed Almahrouky

ID : 8237

Name : Ahmed Maher

ID : 8017

Name : Aly Essam Aly

ID : 8021

Introduction

This Code implements **Markov Decision Processes (MDP)** using two primary algorithms: **Value Iteration** and **Policy Iteration**. These algorithms are commonly used in reinforcement learning to determine optimal policies in grid-based environments. The core concept revolves around determining the best set of actions to take in each state to maximize the total expected reward over time. This grid-based setup includes predefined rewards, actions (up, down, left, right), and state transition probabilities.

The grid is a 3x3 matrix where rewards are assigned, and the algorithms calculate utilities for each cell based on possible actions. The algorithms iterate until a policy (set of actions for each state) is optimized and convergence is achieved.

Code Explanation:

Imports and initialization:

```
      import numpy as np
import random

Rows = 3
Cols = 3
Actions_count = 4
Actions = [(1, 0), (0, -1), (-1, 0), (0, 1)]  # Down, Left, Up, Right
Gamma = 0.99
MAX_ERROR = 1e-4  # Convergence threshold
```

PrintGrid

```
def printGrid(arr, policy=False):
    res = ""
    for r in range(Rows):
        res += "|"
        for c in range(Cols):
            if policy:
                if r == 0 and (c == 0 or c == 2):
                    val = "T.State"
                else:
                    val = ["Down", "Left", "Up", "Right"][arr[r][c]]
            else:
                val = str(arr[r][c])
            res += " " + val[:7].center(7) + " |"
        res += "\n"
    print(res)
```

Explanation:

- **Purpose**: This function is used to print the grid in a neatly formatted manner. Depending on the policy flag, it either prints the utilities (values) for each grid cell or the actions from the policy.
- **How it works:**

It loops through each row and column of the grid.

If policy is set to True, it prints the corresponding action (Down, Left, Up, Right) for each cell. Some cells, like the top-left and top-right corners, are special and labeled as "T.State" (Terminal State).

If policy is False, it prints the utility values for each cell.

getValue:

```
def getValue(U, r, c, action):
    rMove, cMove = Actions[action]
    newR, newC = r + rMove, c + cMove
    if newR < 0 or newC < 0 or newR >= Rows or newC >= Cols or (newR == newC
== 1):  # invalid move
        return U[r][c]
    else:
        return U[newR][newC]
```

Explaination

- **Purpose**: This function retrieves the utility value of a neighboring cell based on the given action (move direction).
- **How it works**:

It uses the Actions array to calculate the new row (newR) and column (newC) based on the current position (r, c) and the chosen action.

It checks if the move is valid (i.e., within the grid and not hitting an obstacle).

If the move is valid, it returns the utility of the new cell. If it's invalid (out of bounds or hitting an obstacle), it returns the current cell's utility.

calcValue:

```
def calcValue(U, r, c, action):
    v = -1
    v += 0.1 * Gamma * getValue(U, r, c, (action - 1) % 4)
    v += 0.8 * Gamma * getValue(U, r, c, action)
    v += 0.1 * Gamma * getValue(U, r, c, (action + 1) % 4)
    return v
```

Explanation:

Purpose: This function calculates the expected value (utility) of taking a certain action from a given state. It considers the uncertainty in actions, i.e., there's a chance the agent might move left or right instead of the intended direction.

How it works:

It calculates the utility of three possible moves: left of the intended direction (10% chance), the intended move (80% chance), and right of the intended direction (10% chance).

It sums these weighted utilities and discounts them by Gamma (which reduces future rewards' impact).

valueIteration:

```
def valueIteration(arr, reward):
    print("During the value iteration:\n")
    while True:
        temp = [[reward, 0, 10], [0, 0, 0], [0, 0, 0]]
        error = 0
        for r in range(Rows):
            for c in range(Cols):
                if r == 0 and (c == 2 or c == 0):
                    continue
                temp[r][c] = max([calcValue(arr, r, c, action) for action in
range(Actions_count)])
                error = max(error, abs(temp[r][c] - arr[r][c]))
        arr = temp
        printGrid(arr)
        if error < MAX_ERROR:
            break
    return arr
```

Explanation:

Purpose: This function performs the **Value Iteration** algorithm to find the optimal utility values for each state in the grid.

How it works:

It starts by initializing a grid with rewards for each state.

It then loops over each state in the grid and computes the maximum utility for that state by considering all possible actions.

This process repeats until the difference between the old and new utility values is less than MAX_ERROR, meaning the utilities have converged.

It prints the grid at each iteration to show the progress of the utility updates.

getOptimalPolicy:

```
def getOptimalPolicy(arr):
    policy = np.zeros((Rows, Cols), dtype=int)
    for r in range(Rows):
        for c in range(Cols):
            if r == 0 and (c == 2 or c == 0):
                continue
            # Choose the action that maximizes the utility
            maxAction, maxv = None, -float("inf")
            for action in range(Actions_count):
                v = calcValue(arr, r, c, action)
                if v > maxv:
                    maxAction, maxv = action, v
            policy[r][c] = maxAction
    return policy
```

Explanation:

Purpose: This function determines the optimal policy by choosing the action that maximizes the utility for each state.

How it works:

For each cell in the grid, it calculates the utility for each action (up, down, left, right) using the calcValue function.

It picks the action with the highest utility and stores it in the policy grid.

Returns the optimal policy, which is a 2D array showing the best action to take from each state.

policyEvaluation:

```
def policyEvaluation(policy, arr, reward):
    while True:
        temp = [[reward, 0, 10], [0, 0, 0], [0, 0, 0]]
        error = 0
        for r in range(Rows):
            for c in range(Cols):
                if r == 0 and (c == 2 or c == 0):
                    continue
                temp[r][c] = calcValue(arr, r, c, policy[r][c])
                error = max(error, abs(temp[r][c] - arr[r][c]))
        arr = temp
        if error < MAX_ERROR:
            break
    return arr
```

Explanation:

Purpose: This function evaluates the utility of a given policy. It computes the utility of each state based on the actions dictated by the current policy.

How it works:

It loops through each cell in the grid and uses the action prescribed by the policy to calculate the utility.

This process continues until the utilities converge (i.e., the difference between old and new utilities is smaller than MAX_ERROR).

Returns the utility grid that corresponds to the current policy.

policyIteration:

```
def policyIteration(policy, arr, reward):
    print("During the policy iteration:\n")
    while True:
        arr = policyEvaluation(policy, arr, reward)
        unchanged = True
        for r in range(Rows):
            for c in range(Cols):
                if r == 0 and (c == 2 or c == 0):
                    continue
                maxAction, maxv = None, -float("inf")
                for action in range(Actions_count):
                    value = calcValue(arr, r, c, action)
                    if value > maxv:
                        maxAction, maxv = action, value
                if maxv > calcValue(arr, r, c, policy[r][c]):
                    policy[r][c] = maxAction
                    unchanged = False
        if unchanged:
            break
        printGrid(policy, True)
    return policy
```

Explanation:

- **Purpose**: This function performs the **Policy Iteration** algorithm. It improves a given policy by alternating between policy evaluation and policy improvement.
- **How it works**:

  First, it evaluates the current policy using policyEvaluation.

  Then, it improves the policy by checking if a better action (with a higher utility) exists for each state. If a better action is found, it updates the policy.

  This process repeats until no changes are made to the policy, indicating convergence.

  The function prints the grid showing the policy at each iteration.

Main Loop - Value Iteration:

```
r_values = [100, 3, 0, -3]
for r in r_values:
    Grid = [
        [r, -1, 10],
        [-1, -1, -1],
        [-1, -1, -1]
    ]
    print("The initial U for ", r, " is:\n")
    printGrid(Grid)
    final = valueIteration(Grid, r)
    policy = getOptimalPolicy(final)
    print("The optimal policy is:\n")
    printGrid(policy, True)
```

Main Loop - Policy Iteration:

```
r_values = [100, 3, 0, -3]
for r in r_values:
    Grid = [
        [r, -1, 10],
        [-1, -1, -1],
        [-1, -1, -1]
    ]
    policy = [[random.randint(0, 3) for j in range(Cols)] for i in
range(Rows)]
    print("The initial random policy for ", r, " is:\n")
    printGrid(policy, True)
    finall = policyIteration(policy, Grid, r)
    print("The optimal policy is:\n")
    printGrid(finall, True)
```

Main Loops Explanation:

Value Iteration Loop:

- For each reward value in r_values, the program initializes a grid and performs the value iteration algorithm.
- After convergence, it extracts and prints the optimal policy for each reward setting.

Policy Iteration Loop:

- For each reward value in r_values, a random policy is generated, and policy iteration is used to improve it.

- After convergence, the program prints the optimal policy.

Results: