ADDIS ABABA UNIVERSITY

ADDIS ABABA INSTITUTE OF TECHNOLOGY

SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING – SITE

Department of Artificial Intelligence

Course: Reinforcement Learning

Report on Reinforcement Learning first assignment.

Submitted by: - Migbar Abera

ID: GSR 3053/15

Section: Regular

Submitted to: Dr. Natnael Argaw (PhD)

1. Here is the line by line explanation for the code

```
import numpy as np
# Define the grid world environment
grid_size = 5
states = grid_size * grid_size  # Total number of states
actions = ["up", "down", "left", "right"]  # Possible actions
```

The grid size is set to 5x5, and the total number of states is calculated. The possible actions the agent can take are defined as "up", "down", "left", and "right".

```
rewards = np.full((grid_size, grid_size), -1)  # Define rewards for each cell
rewards[0, 4] = 10  # Goal state
rewards[1, 1] = -100  # Obstacle or blocked state
```

The rewards matrix is initialized with -1 for all cells. The goal state at (0, 4) is given a reward of +10. - The obstacle state at (1, 1) is given a reward of -100.

```
# Define the transition probabilities for each action in each state
transitions = []
for _ in range(states):
    action_probs = []
    for _ in range(len(actions)):
        action_probs.append({"prob": 0.25})  # All actions have equal probability
    transitions.append(action_probs)
```

An empty array called **transitions** is created to store the transition probabilities. The transition probabilities for each action in each state are set to 0.25, indicating that all actions have an equal probability of occurring.

```
# Define the initial value function
V = np.zeros((grid_size, grid_size))
```

The value function **V** is initialized as a 5x5 matrix filled with zeros.

```
# Value iteration
gamma = 0.9  # Discount factor
epsilon = 0.01  # Convergence threshold
delta = epsilon + 1  # Initialize delta to enter the loop
while delta > epsilon:
    delta = 0
```

The discount factor (**gamma**) is set to 0.9. The convergence threshold (**epsilon**) is set to 0.01. The change in values (**delta**) is initialized to be greater than the convergence threshold to enter the value iteration loop. The loop will continue until **delta** becomes less than or equal to **epsilon**.

```
for i in range(grid_size):
    for j in range(grid_size):
        if (i == 0 and j == 4) or (i == 1 and j == 1):
            continue  # Skip goal state and blocked state
        v = V[i, j]
        state_values = []
```

The code iterates over each state in the grid world. The goal state and the blocked state are skipped. The current value of the state (**v**) is stored, and an empty array called **state_values** is created to store the expected values for each action.

```
for action in actions:
    new_i, new_j = i, j
    if action == "up":
        new_i -= 1
    elif action == "down":
        new_i += 1
    elif action == "left":
        new_j -= 1
    elif action == "right":
        new_j += 1
```

The code iterates over each action for the current state. The new state after applying the action is calculated by adjusting the **new_i** and **new_j** variables based on the chosen action.

```
    new_j += 1

# Check if the new state is valid
if new_i < 0 or new_i >= grid_size or new_j < 0 or new_j >= grid_size or (i == 1 and j == 1):
    new_i, new_j = i, j  # Stay in the current state
```

Boundary conditions and the blocked state at (1, 1) are checked. - If the new state is invalid (outside the grid or blocked), the agent stays in the current state.

```
# Calculate the expected value
value = 0
for prob in transitions[i * grid_size + j][actions.index(action)]["prob"]:
    reward = rewards[new_i, new_j]
    value += prob * (reward + gamma * V[new_i, new_j])
state_values.append(value)
```

The expected value for the current action is calculated by multiplying the transition probability with the reward and discounted value of the new state. The calculated value is added to the **state_values** array.

```
# Update the value function
V[i, j] = max(state_values)

# Update the delta
delta = max(delta, abs(v - V[i, j]))
```

The value function **V** for the current state is updated with the maximum value from the **state_values** array. The change in values (**delta**) is updated as the maximum difference between the old and new values.

```
# Print the optimal policy
policy = np.empty((grid_size, grid_size), dtype=object)
for i in range(grid_size):
    for j in range(grid_size):
        if (i == 0 and j == 4) or (i == 1 and j == 1):
            policy[i, j] = "Goal/Blocked"
```

An empty matrix called **policy** is created to store the optimal policy for each state. The goal state and the blocked state are assigned the value "Goal/Blocked".

```
state_values = []
for action in actions:
    new_i, new_j = i, j
    if action == "up":
        new_i -= 1
    elif action == "down":
        new_i += 1
    elif action == "left":
        new_j -= 1
    elif action == "right":
        new_j += 1

    if new_i < 0 or new_i >= grid_size or new_j < 0 or new_j >= grid_size or (i == 1 and j == 1):
        new_i, new_j = i, j  # Stay in the current state

    state_values.append(V[new_i, new_j])

max_value = max(state_values)
max_indices = [index for index, value in enumerate(state_values) if value == max_value]
optimal_actions = [actions[index] for index in max_indices]
policy[i, j] = optimal_actions
```

For each remaining state, the expected values for each action are calculated based on the updated value function. The maximum value is identified, and the corresponding action(s) are stored as the optimal actions in the **policy** matrix.

```
print("Optimal Policy:")
for row in policy:
    print(row)
```

The optimal policy for each state is printed as follows.

```
Optimal Policy:
[list(['up', 'down', 'left', 'right'])
 list(['up', 'down', 'left', 'right'])
 list(['up', 'down', 'left', 'right'])
 list(['up', 'down', 'left', 'right']) 'Goal/Blocked']
[list(['up', 'down', 'left', 'right']) 'Goal/Blocked'
 list(['up', 'down', 'left', 'right'])
 list(['up', 'down', 'left', 'right'])
 list(['up', 'down', 'left', 'right'])]
[list(['up', 'down', 'left', 'right'])
 list(['up', 'down', 'left', 'right'])
 list(['up', 'down', 'left', 'right'])
 list(['up', 'down', 'left', 'right'])
 list(['up', 'down', 'left', 'right'])]
[list(['up', 'down', 'left', 'right'])
 list(['up', 'down', 'left', 'right'])
 list(['up', 'down', 'left', 'right'])
 list(['up', 'down', 'left', 'right'])
 list(['up', 'down', 'left', 'right'])]
[list(['up', 'down', 'left', 'right'])
 list(['up', 'down', 'left', 'right'])
 list(['up', 'down', 'left', 'right'])
 list(['up', 'down', 'left', 'right'])
 list(['up', 'down', 'left', 'right'])]
```

+ Code + Markdown

Here are the main points.

- The grid world environment is defined as a 5x5 grid where the agent can move in four directions: up, down, left, and right.
- Rewards are defined for each cell in the grid: a reward of +10 is given for reaching the goal state located at (0, 4), and a reward of -1 is given for every other step taken.
- An empty array called **transitions** is initialized to store the transition probabilities for each action in each state. In this case, all actions have an equal probability of 0.25. 4. The value function **V** is

initialized as a 5x5 matrix with zeros. This matrix will contain the expected cumulative rewards for each state.

- To start with, gamma is set at to 0.9, and epsilon is set at 0.01. to. To enter a value iteration loop, the variable delta is initialized with an initial value greater than epsilon..
- The value iteration loop starts and continues until the change in values (delta) becomes less than the convergence threshold (epsilon).
- For each state in the grid world, the current value is stored in the variable **v**, and an array **state_values** is created to store the expected values for each action.
- The algorithm then iterates over each possible action and calculates the new state by applying the action to the current state. Boundary conditions are checked to ensure the agent does not move outside the grid, and the obstacle state at (1, 1) is accounted for.
- The transition probabilities, the rewards associated with the new state, as well as the discounted values of the new state are taken into account in order to find the expected value for the current action. Finally, the state_values array is updated with the expected value.
- The value function **V** for the current state is updated by selecting the maximum value from the **state_values** array. This value represents the expected cumulative reward that can be obtained from the current state.
- Then, we calculate delta that is the biggest change between the old and updated values in all states. Should delta be greater than epsilon then the value iteration cycle continues.
- After the value iteration process has reached its convergence points, the policies with optimal values to each state are obtained. The expected values are computed for every state and the associated actions. The action(s) with the highest expected value are then considered as the optimal actions. It then prints the optimal policy. It executes the value iteration scheme towards identifying an appropriate policy in a specified grid world setting. It proceeds with value iteration and it includes initializing the value function, iteratively updating the value function, checking for convergence and deriving the optimal policy.

2. The code provided below is the simple Q – learning algorithm in a grid world environment. The environment consists of a 5x5 grid where an agent can move in four directions: up, down, left, and right. The aim of the agent is to arrive at the end state where it gets +10 while getting -1 for every other action.

   num_episodes specify how many times the Q-learning algorithm is run for in total. Alpha is set to 0.5, acting as the weighting for the Q-values' update of new information. The discount factor (gamma) is 0.9 and determines the weight placed on future rewards in the updating of the Q value. Zeros are initially used in Q-table to represent the learned Q-values.

In every episode, the agent begins in the starting state (0, 0), and proceeds with actions until it ends up in the goal state. A greedy action selection with an epsilon probability of exploration and exploitation of 0.1 and 0.9 respectively. Using the Q learning update rule, the Q value for the selected action is updated by adding the reward for the current state-action pair and subtracting the maximum Q value for the next state.

Finally, the Q-values are printed out for each state-action pair after the training process in the grid world. The output shows the Q-values for each state in the format "State (row, column): Action: Q-value". Positive Q-values mean better actions while negative Q-values indicate low reward actions. The Q-values should be indicative of how much the agent has learned about the environment and will help guide it in making the best possible decisions.

The learned Q-values represent the expected future rewards for each state-action pair in the grid world. From the printed output, we can observe the following:

- Each state is represented by its coordinates (row, column).
- The Q-values for each state-action pair are shown for the actions Up, Down, Left, and Right. Based on the learned Q-values
    - The agent has now discovered that it can go some specific ways to obtain extra benefits. Specifically, in state (0, 0), the agent knows that it would receive more rewards by going DOWN or RIGHT rather than UP or LEFT.
    - The q-values in the goal state (4, 4) are all zero, showing that the agent has found out that to get more rewards will not be possible from the goal state.
    - Some of the Q-values are higher, meaning that the agent understands which of the optimal actions are suitable for a given state. Take for instance state (3, 3) where Q-values for the Up and Right is higher than all other actions.
    - In some states, there exist some negative Q-values, suggesting that the agent knows that some actions can generate rewards of lower value. For instance, in state (0, 4) the agent understands that taking Down and Left actions leads to negative payoffs as compared to others. However, it is important to note that the Q-learning algorithm is an iteration process, and the learned Q-value could increase with more training sessions. Also, the exploration-exploitation trade-off can be fine-tuned via modifying the epsilon-greedy policy parameter.