

**AUTONOMOUS AGENT NAVIGATION USING REINFORCEMENT LEARNING**

***Department of Computer Science***

***Christ (Deemed to be University), Bengaluru***

**Team members,**

Kripa Mary Jose (2348526)

Melvin Infant A (2348533)

**Problem Statement**

The goal of the project is to address the problem of robotic navigation in a grid-based environment. In other words, the task is to train an agent-a simulated robot-to find the shortest path from a starting point to a goal while avoiding obstacles. This mirrors real-world applications in which autonomous agents, such as robots, must navigate unknown environments efficiently to achieve a goal. The challenge lies in the design of an algorithm that can learn an optimal policy to reach the goal using minimal steps without causing a collision with obstacles.

**Implemented Idea**

The project applies the Q-Learning concept, a model-free reinforcement learning algorithm, to allow the agent to learn a policy for optimal navigation. Q-Learning allows the agent to learn the value of actions in a given state through interactions with the environment, thereby improving its decision-making over time. The environment can be represented as a grid with random obstacles, a starting point, and a goal. The agent uses a Q-table to store state-action values, which it updates according to rewards received upon taking certain actions in different states.

**Rules for the Reward System and Agent Movement**

1. **State and Actions**:
   * **States**: The agent’s state is defined by its coordinates on the grid.
   * **Actions**: The agent can move in four possible directions: up, down, left, or right.
2. **Movement**:
   * The agent changes its position from the current one to a new one according to the chosen action. If the new position is valid (within the grid boundaries and not an obstacle), the move is accepted.
   * If the move reached the goal state, it gets a reward of +1, which means the move was successful. If it hits an obstacle or goes out of bounds, then it gets a small negative reward of -0.1 to penalize the action.
3. **Q-Learning Updates**:
   * **Choosing Actions**: The agent decides on its next action based on an exploration-exploitation strategy, balancing between exploring new actions and exploiting known best actions to maximize immediate rewards.
   * **Q-value Updates**: The Q-value for a state-action pair is updated using the formula below,

Q(state, action) = Q(state, action) + a × : (reward +y × max Q(next\_state, a') –Q(state,action))a'

where a is the learning rate, y is the discount factor, and maxɑ' Q(next\_state, a') is the maximum Q-value for all possible next actions from the next state.

**Overall System and Components**

* **Grid Environment**: Represents the simulated world in which the agent navigates. It has the capability to generate obstacles, to manage the state of the agent, and to determine outcomes resulting from the actions of the agent.
* **Q-Learning Agent**: This module embodies the Q-Learning algorithm. It is responsible for keeping the Q-table, selecting the action from the current state, and updating the Q-values for every taken action.
* **Rendering**: ‘Pygame’ library is used for visualizing the environment. The rendering updates the screen with the agent's position, obstacles, and current rewards to provide a visual feedback loop during training.
* **Training Loop**: Repeatedly resets the environment, forces the agent to take actions, and updates the Q-values for a fixed number of episodes. Each episode represents a complete navigation task from start to goal, collecting rewards along the way to improve the agent's learning.

**Conclusion**

The project represents an example of real applicative Q-Learning in reinforcement learning to solve real-world problems of navigation, thereby contributing to both robotics and artificial intelligence fields.

**Reference**

1. Sutton, R. S., & Barto, A. G. (1998). Reinforcement Learning: An Introduction. MIT Press.
2. Watkins, C. J. C. H., & Dayan, P. (1992). "Q-learning." Machine Learning, 8(3-4), 279-292.
3. Silver, D., & Geist, M. (2007). "The Generalization of Q-learning to General State-Action Spaces." *Machine Learning, 68*(1), 287-316.

**Source Code**

import numpy as np

import pygame

import time

# Environment setup

class GridEnvironment:

    def \_\_init\_\_(self, grid\_size=(5, 5), start=(0, 0), goal=(4, 4), obstacle\_percentage=0.2):

        self.grid\_size = grid\_size

        self.start = start

        self.goal = goal

        self.state = start

        self.obstacle\_percentage = obstacle\_percentage

        self.obstacles = self.generate\_obstacles()  # Fixed random obstacles

        self.reset()

    def generate\_obstacles(self):

        np.random.seed(42)  # Ensures consistent random obstacle placement

        num\_obstacles = int(self.grid\_size[0] \* self.grid\_size[1] \* self.obstacle\_percentage)

        obstacles = set()

        while len(obstacles) < num\_obstacles:

            obstacle = (np.random.randint(self.grid\_size[0]), np.random.randint(self.grid\_size[1]))

            if obstacle != self.start and obstacle != self.goal:

                obstacles.add(obstacle)

        return obstacles

    def reset(self):

        self.state = self.start

        return self.state

    def step(self, action):

        x, y = self.state

        # Define action effects: up, down, left, right

        actions = {

            0: (-1, 0),  # up

            1: (1, 0),   # down

            2: (0, -1),  # left

            3: (0, 1)    # right

        }

        dx, dy = actions[action]

        new\_state = (x + dx, y + dy)

        # Check for boundaries and obstacles

        if (0 <= new\_state[0] < self.grid\_size[0] and

            0 <= new\_state[1] < self.grid\_size[1] and

            new\_state not in self.obstacles):

            self.state = new\_state

        reward = 1 if self.state == self.goal else -0.1

        done = self.state == self.goal

        return self.state, reward, done

    def render(self, screen, cell\_size, current\_reward):

        screen.fill((255, 255, 255))

        for i in range(self.grid\_size[0]):

            for j in range(self.grid\_size[1]):

                rect = pygame.Rect(j \* cell\_size, i \* cell\_size, cell\_size, cell\_size)

                if (i, j) in self.obstacles:

                    pygame.draw.rect(screen, (0, 0, 0), rect)  # Obstacles

                elif (i, j) == self.goal:

                    pygame.draw.rect(screen, (0, 255, 0), rect)  # Goal

                elif (i, j) == self.state:

                    pygame.draw.rect(screen, (0, 0, 255), rect)  # Agent

                else:

                    pygame.draw.rect(screen, (200, 200, 200), rect, 1)  # Grid

        # Display the current reward for this step

        font = pygame.font.Font(None, 36)

        reward\_text = font.render(f"Step Reward: {current\_reward:.2f}", True, (0, 0, 0))

        screen.blit(reward\_text, (10, 10))

        pygame.display.flip()

# Q-Learning implementation

class QLearningAgent:

    def \_\_init\_\_(self, env, learning\_rate=0.1, discount\_factor=0.9, epsilon=0.9):

        self.env = env

        self.lr = learning\_rate

        self.gamma = discount\_factor

        self.epsilon = epsilon

        self.q\_table = {}

    def choose\_action(self, state):

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

            # Exploitation

            if state in self.q\_table:

                return np.argmax(self.q\_table[state])

        # Exploration

        return np.random.choice([0, 1, 2, 3])

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

        if state not in self.q\_table:

            self.q\_table[state] = [0, 0, 0, 0]

        max\_next\_q = max(self.q\_table.get(next\_state, [0, 0, 0, 0]))

        current\_q = self.q\_table[state][action]

        self.q\_table[state][action] += self.lr \* (reward + self.gamma \* max\_next\_q - current\_q)

# Main training loop

def train\_agent():

    pygame.init()

    grid\_size = (5, 5)  # Smaller grid size

    cell\_size = 100     # Larger cells for better visibility

    screen = pygame.display.set\_mode((grid\_size[1] \* cell\_size, grid\_size[0] \* cell\_size))

    pygame.display.set\_caption("Simulated Robot Navigation")

    env = GridEnvironment(grid\_size=grid\_size)

    agent = QLearningAgent(env)

    episodes = 500

    for episode in range(episodes):

        state = env.reset()

        done = False

        total\_reward = 0

        while not done:

            for event in pygame.event.get():

                if event.type == pygame.QUIT:

                    pygame.quit()

                    return

            # Render the environment with the current step's reward

            env.render(screen, cell\_size, total\_reward)

            action = agent.choose\_action(state)

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

            agent.update\_q\_value(state, action, reward, next\_state)

            state = next\_state

            total\_reward += reward

            time.sleep(0.1)

        print(f"Episode {episode + 1}/{episodes} completed. Total Reward: {total\_reward:.2f}")

    print("Training complete!")

    env.render(screen, cell\_size, total\_reward)

    time.sleep(3)

    pygame.quit()

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

    train\_agent()

**Example Output**



