Faculty of Computers &
Artificial Intelligence

Benha University

Reinforcement Project
‘Maze Solving Self-driving Car’

⟶ Artificial Intelligence Department ⟵

Presented by :

Haidy Talaat Fadel Elseadawy.

Mariam Mohamed Fathy Aboalata.

Ahmed Ayman Ahmed Thabet.

Under Supervision of :

Eng. Mariam Mahmoud

Table of Contents

Chapter 1

Problem Definition and Environment Setup.

Chapter1

1.Problem Definition and Environment Setup.

1.1 Introduction.

With the rapid advancements in Artificial Intelligence and Machine Learning, algorithms capable of solving complex problems like mazes have become a key area of both educational and research interest. This project aims to design and develop an intelligent system capable of efficiently exploring and solving mazes using Deep Reinforcement Learning techniques.
The maze was entirely built from scratch, without relying on any pre-existing libraries, in order to gain a deeper understanding of the maze generation process and to have full control over the environment's structure and complexity. As a first step, the agent was allowed to move randomly through the maze, enabling it to explore the environment and collect experience before formal training.

To train the agent, we implemented a Deep Q-Learning Network (DQN), a reinforcement learning algorithm that combines Q-Learning with deep neural networks to estimate optimal action-values. By using DQN, the agent was able to learn the most efficient path to solve the maze through trial and error, balancing exploration and exploitation effectively.

This project stands as a practical application of AI concepts, merging programming, algorithm design, and deep learning. It provides a valuable hands-on experience in developing intelligent agents that can learn and adapt within dynamic environments.

1.2 Problem Statement.

Maze solving is a well-known problem in artificial intelligence, used to test how well an agent can make decisions and learn in an unknown environment. While traditional algorithms like DFS or A* can solve mazes, they rely on full knowledge of the maze and do not involve any learning or adaptation.

In this project, the goal is to build an intelligent agent that can learn to solve a maze without knowing its structure in advance. The maze is generated from scratch, and the agent must figure out how to reach the goal through trial and error.

The challenge is to train the agent to find the shortest or most efficient path using only the feedback it receives from the environment and also avoid obstacles that in the way. At first, the agent moves randomly to explore the maze and collect experience. Then, it uses a Deep Q-Learning Network (DQN) to learn from that experience and improve its decisions over time.

This project aims to create a system that can adapt to different mazes and solve them effectively, demonstrating the power of deep reinforcement learning in dynamic environments.

1.3 Objectives.

The main objectives of this project are:

1. **Build a Maze Environment from Scratch**
    - Design and generate maze structures without using external or prebuilt libraries, to better understand the maze generation process and customize the environment.
2. **Simulate Random Agent Behavior Before Training**
    - Allow the agent to take random actions through the maze to try to reach the goal without any learning or memory that stores the experience.
3. **Train the Agent Using Deep Q-Learning (DQN)**
    - Implement a Deep Q-Learning Network to enable the agent to learn the best path to the goal through reinforcement learning and show difference and the role of Reinforcement Learning in making agent learn and take actions based on learning from experience.
4. **Develop an Adaptive Maze Solver**
    - Ensure the agent can adapt to different maze layouts and still find an efficient solution.
5. **Compare Performance Before and After Training**
    - Evaluate the difference in the agent's behavior and efficiency before and after applying DQN.
6. **Demonstrate a Practical Application of Reinforcement Learning**

- Show how reinforcement learning can be applied to real-world problem-solving scenarios in unknown environments.

1.3 *Environment Setup*.

Maze Generation (Built from Scratch):

In this project, the maze environment was built entirely from scratch without relying on any external libraries or pre-built environments. This allowed full control over the maze's structure, logic, and complexity, and provided a deeper understanding of how such environments work internally.

We created a custom environment by extending the gym.Env class, which is compatible with reinforcement learning frameworks. The maze is represented using a 2D NumPy array, where each cell in the grid has a specific meaning:

- 1 represents walls (unwalkable cells)
- 0 represents empty paths (walkable)
- 2 represents obstacles (punishable but passable)
- 3 is the starting point of the agent
- 4 is the goal that the agent needs to reach

The environment was rendered visually using Pygame, with each cell displayed in a specific color:

- Black for walls
- Red for obstacles
- Green for the starting point
- Blue for the goal
- Yellow for the agent

The maze logic handles the following:

- Movement in four directions (up, down, left, right)
- Collision with walls (penalized)
- Reaching the goal (rewarded)
- Interaction or avoidance of obstacles (negative or positive reward accordingly)

Additionally, the environment tracks cumulative rewards and allows the agent to explore the maze. This environment setup was crucial for training the agent using Deep Q-Learning, and it helped simulate real-world scenarios where the agent learns from its surroundings rather than pre-defined maps.

This custom maze provided a dynamic and challenging environment for the agent, supporting both exploration and learning phases of the project.

Implementation for building custom environment:

1. Import necessary Libraries:

In [ ]:
```
!pip install pygame
```

```
Requirement already satisfied: pygame in /usr/local/lib/python3.11/dist-packages (2.6.1)
```

In [ ]:
```
import numpy as np
import gym
from gym import Env
from gym.spaces import Discrete, Box
import pygame
from google.colab import files
from IPython.display import Image, display
import random
import time
import cv2
import glob
import os
```

2.Building Custom Maze Environment:

2.1 **Class Initialization: __init__()**

This is where the environment is set up:

- **Maze Definition**: A 2D NumPy array defines the maze layout using numbers:
    - 1 = Wall
    - 0 = Free path
    - 2 = Obstacle
    - 3 = Starting point
    - 4 = Goal
- **Agent State**: Finds the start and goal positions. The agent begins at the start.
- **Action & Observation Spaces**:
    - action_space = Discrete(4): 4 possible actions → up, down, left, right.

- observation_space = Box(...): Agent's position is a 2D coordinate within the maze bounds.

- **Rendering with Pygame**:
  - Each maze cell is drawn as a square (40x40 pixels).
  - The full window size depends on the maze size.

2. Building Custom Maze Environment

```
[ ]  class CustomMazeEnv(Env):
        def __init__(self, render_mode="rgb_array"):
            super().__init__()
            self.total_reward = 0  # Initialize cumulative reward

            # Maze structure
            self.maze = np.array([
                [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                [1, 3, 0, 2, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1],
                [1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1],
                [1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1],
                [1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1],
                [1, 0, 0, 1, 0, 1, 0, 0, 0, 2, 0, 1, 0, 4],
                [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1],
                [1, 0, 2, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1],
                [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1],
                [1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1],
                [1, 0, 0, 1, 0, 1, 0, 2, 0, 0, 0, 0, 0, 1],
                [1, 0, 0, 1, 1, 1, 0, 2, 0, 0, 2, 2, 0, 1],
                [1, 0, 0, 0, 0, 0, 0, 2, 0, 0, 2, 0, 0, 1],
                [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
            ])

            self.start_pos = np.argwhere(self.maze == 3)[0]
            self.goal_pos = np.argwhere(self.maze == 4)[0]
            self.agent_pos = self.start_pos.copy()
```

```
        self.start_pos = np.argwhere(self.maze == 3)[0]
        self.goal_pos = np.argwhere(self.maze == 4)[0]
        self.agent_pos = self.start_pos.copy()

        self.action_space = Discrete(4)
        self.observation_space = Box(low=np.array([0, 0]), high=np.array([14, 13]), dtype=np.int32)
        self.render_mode = render_mode

        pygame.init()
        # the size of each square of the maze
        self.cell_size = 40
        # total size of the screen =rows*columns*cell_size (14,13)*40
        self.screen = pygame.display.set_mode((self.maze.shape[1] * self.cell_size, self.maze.shape[0] * self.c
        pygame.display.set_caption("Custom Maze Environment")
```

2.2 Reset Method: reset()

This function resets the environment for a new episode.

- Sets the agent back to the starting point.
- Resets the total reward to zero. Returns the initial state (agent position).

```
def reset(self):
    #resets the environment but keeps the total reward
    self.agent_pos = self.start_pos.copy()
    self.total_reward = 0  # Reset cumulative reward
    return np.array(self.agent_pos, dtype=np.int32)
```

2.3 Obstacle Check: _avoided_obstacle()

This helper checks if the agent **moved near an obstacle** without stepping on it (to optionally reward "smart" avoidance).

- Checks 4 directions around the agent's **previous** position.
- If any neighboring tile is an obstacle (2) and wasn't stepped on → return True.

```
def _avoided_obstacle(self, old_pos, new_pos):
    x, y = old_pos
    for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:  # 4 directions
        if (x + dx, y + dy) != new_pos and self.maze[x + dx, y + dy] == 2:
            return True  # found an obstacle nearby but didn't step on it
    return False
```

2.4 Step Function: step(action)

Executes a single action and updates the environment.

- Computes the new position based on action (0-3).
- Checks what's at the new position:
    - **Wall or Obstacle?** → Penalty: -10
    - **Goal?** → Reward: +100 and mark episode as done
    - **Moved near obstacle but avoided it?** → Small reward: +5
    - **Valid empty move?** → No penalty or bonus
- Updates total reward and returns:

- New state.
- Total cumulative reward.
- Done flag.
- Maze value at new location

```
def step(self, action):
    #Takes an action and updates the state with rewards
    x, y = self.agent_pos
    old_pos = (x, y)

    if action == 0:
        new_x, new_y = x - 1, y  # left
    elif action == 1:
        new_x, new_y = x + 1, y  # right
    elif action == 2:
        new_x, new_y = x, y - 1  # up
    elif action == 3:
        new_x, new_y = x, y + 1  # down

    done = False
    expected_maze_value = self.maze[new_x, new_y] if 0 <= new_x < self.maze.shape[0] and 0 <= new_y < self.
    reward = 0
    if expected_maze_value == 1 or expected_maze_value == 2:
        reward = -10  # Hitting wall or obstacle
    else:
        self.agent_pos = [new_x, new_y]
        if (new_x, new_y) == tuple(self.goal_pos):
            reward = 100
            done = True
        elif self._avoided_obstacle(old_pos, (new_x, new_y)):
            reward = 5

    self.total_reward += reward
    return np.array(self.agent_pos, dtype=np.int32), self.total_reward, done, expected_maze_value
```

2.5 Reset Method: reset()

This function resets the environment for a new episode.

- Sets the agent back to the starting point.
- Resets the total reward to zero.
- Returns the initial state (agent position).

```
def render(self, save_as_image=False, image_path="/content/maze_image.png"):
    self.screen.fill((255, 255, 255))
    colors = {1: (0, 0, 0), 2: (255, 0, 0), 3: (0, 255, 0), 4: (0, 0, 255), 0: (255, 255, 255)}

    for r in range(self.maze.shape[0]):
        for c in range(self.maze.shape[1]):
            pygame.draw.rect(self.screen, colors[self.maze[r, c]], (c * self.cell_size, r * self.cell_size,
            pygame.draw.rect(self.screen, (0, 0, 0), (c * self.cell_size, r * self.cell_size, self.cell_si
    # to draw the agent as yellow
    pygame.draw.circle(self.screen, (255, 255, 0), (self.agent_pos[1] * self.cell_size + self.cell_size //
    pygame.display.flip()

    # Save the rendered maze to an image file if required
    if save_as_image:
        pygame.image.save(self.screen, image_path)
        print(f"Maze image saved as {image_path}")
        files.download(image_path)

        # Display the image in Colab
        display(Image(filename=image_path))

def close(self):
    pygame.quit()
```

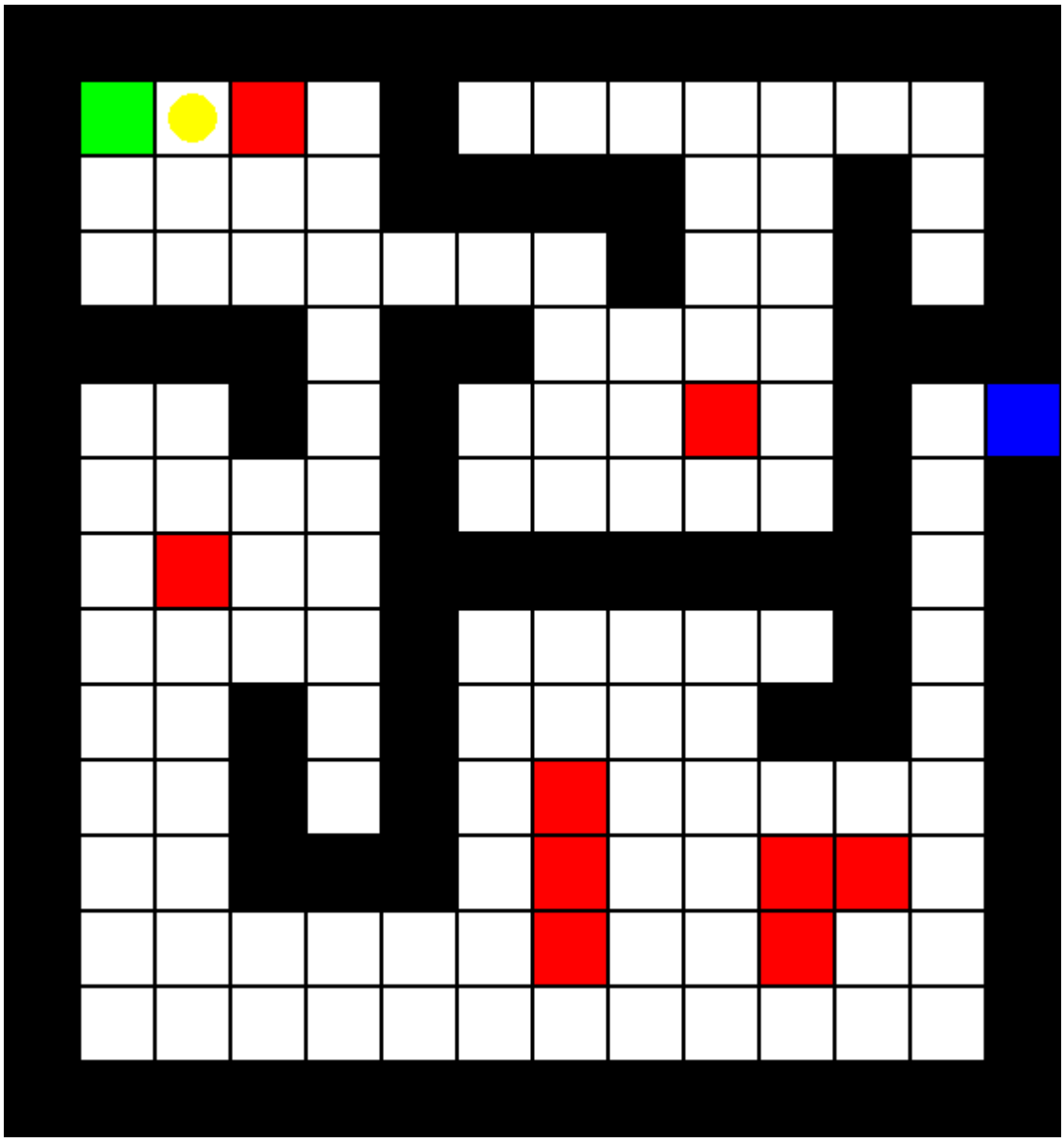
2.6 Render and Save the Maze:

- In a simple loop, the agent repeatedly takes the action to move downward (action = 3) as the main aim of this code is to render the environment using Pygame.
- As an output a snapshot of the maze is saved to an image file (maze_image.png), making it easy to visualize progress frame by frame.

```
env = CustomMazeEnv()
state = env.reset()
done = False

# Simulate a few steps and render/save the maze
while not done:
    action = 3  # Example action (move down)
    next_state, reward, done, maze_value = env.step(action)
    env.render(save_as_image=True, image_path="/content/maze_image.png")
    break
```

The Maze after Building and Rendering:



Chapter 2

Algorithm Selection and Implementation

Chapter2

2. Algorithm Selection and Implementation

2.1 Taking Random Actions.

Before training the agent, we allow it to perform random actions in the maze. This means the agent moves through the environment without any learned strategy or purpose, exploring the maze in an unstructured way. These random movements help the agent to gather initial experiences, such as recognizing obstacles or the general layout of the maze.

The main aim of allowing agent to take random actions is to get a baseline for comparison when using more advanced strategies DQN Algorithm in the next step.

```
[ ] episodes = 20
    for episode in range(1, episodes+1):
        state = env.reset()
        done = False
        total_comulative_reward = 0

        while not done:
            env.render()
            action = np.random.choice([0,1,2,3])
            n_state, reward, done, maze_value = env.step(action)
            total_comulative_reward+=reward
        print('Episode:{} Total Comulative Reward:{}'.format(episode, total_comulative_reward))
```

```
Episode:1 Total Comulative Reward:-11699495
Episode:2 Total Comulative Reward:-4371375
Episode:3 Total Comulative Reward:-324693245
Episode:4 Total Comulative Reward:-448300145
Episode:5 Total Comulative Reward:-80032870
Episode:6 Total Comulative Reward:-32503800
Episode:7 Total Comulative Reward:-24751320
Episode:8 Total Comulative Reward:-8352465
Episode:9 Total Comulative Reward:-2830165
Episode:10 Total Comulative Reward:-697530
Episode:11 Total Comulative Reward:-93778520
Episode:12 Total Comulative Reward:-647145
Episode:13 Total Comulative Reward:-327818275
Episode:14 Total Comulative Reward:-576821510
Episode:15 Total Comulative Reward:-5226450
Episode:16 Total Comulative Reward:-111212635
Episode:17 Total Comulative Reward:-809395
Episode:18 Total Comulative Reward:-41528460
Episode:19 Total Comulative Reward:-617735
Episode:20 Total Comulative Reward:-11519465
```

Try to record a video for the agent while taking these random actions to follow it's progress:

```python
def record_agent(env, agent=None, episodes=1):
    env.record_video = True
    obs = env.reset()
    done = False

    for _ in range(episodes):
        obs = env.reset()
        done = False
        while not done:
            if agent:
                action = agent.select_action(obs)
            else:
                action = env.action_space.sample()
            obs, _, done, _ = env.step(action)
            env.render()
    env.close()
```

```python
[ ] env = CustomMazeEnv(record_video=True, output_video_path="Random_action_demo.mp4")
```

```python
[ ] record_agent(env, episodes=1)
```

```
Video saved to Random_action_demo.mp4
```

The link that contains the video:

https://github.com/HaidyTalaat/Reinforcement-Learning-Project-Maze-Solving-Self-Driving-Car/blob/mariam-random-agent-recording-demo/Random_actions_demo%20.mp4

2.2 Building DQN Algorithm:

A Deep Q-Network (DQN) is a reinforcement learning algorithm that combines Q-learning with deep neural networks. It is designed to solve complex decision-making tasks where the state space is large, and traditional Q-learning methods become inefficient.

In standard Q-learning, a Q-table is used to store the Q-values for each state-action pair. However, as the state space grows (such as in problems involving images or high-dimensional data), a Q-table becomes impractical due to its size. DQN addresses this by approximating the Q-function using a neural network, allowing it to generalize across different states and actions.

DQN Algorithm Steps:

To summarize, building a **Deep Q-Network (DQN)** involves the following key steps:

1. **Defining hyperparameters** that control the learning process.
2. **Building a neural network** that approximates the Q-function.
3. **Storing experiences** in the replay buffer to prevent overfitting.
4. **Using the epsilon-greedy policy** to balance exploration and exploitation.
5. **Implementing the target network** to stabilize learning.
6. **Applying the Q-learning update rule** to adjust Q-values based on the observed rewards.
7. **Training the model** through repeated interaction with the environment.
8. **Evaluating the trained agent** to assess its performance and fine-tune the model.

1.Defining Hyperparameters:

In this section, various hyperparameters are defined. These control the agent's training process:

- **learning_rate_a (α):** Defines how quickly the model adjusts during training. A higher learning rate means larger updates to the model weights.
- **discount_factor_g (γ):** Determines how much the agent values future rewards. A value close to 1 means the agent values future rewards almost as much as immediate rewards.
- **network_sync_rate:** Specifies the frequency of synchronizing the target network with the policy network, improving stability in learning.
- **replay_memory_size:** The maximum size of the replay buffer, which stores past experiences for training.
- **mini_batch_size:** The number of experiences sampled from the replay memory for each training step.
- **episodes:** The total number of training episodes, or iterations, the agent will undergo.
- **epsilon:** The initial exploration rate, which defines the probability of selecting a random action.
- **epsilon_decay:** The rate at which epsilon decays after each episode. This encourages the agent to explore more initially and exploit learned strategies over time.
- **epsilon_min:** The minimum threshold for epsilon, ensuring some level of exploration throughout the training process.

```
[ ] state_size=env.observation_space.shape #represented by a 2D coordinate, [row, column]
    action_size=env.action_space.n
    memory_size = 2000
    gamma = 0.95
    epsilon = 1.0
    epsilon_min = 0.01
    epsilon_decay = 0.995
    learning_rate = 0.001
    batch_size = 16
    episodes = 20
    target_update_freq = 10
    render_interval = 10
    model_filename = "dqn_maze_weights.h5"
    demo_video_filename = "demo_maze.mp4"
```

2.Building the Neural Network Model:

This function defines the architecture of the **neural network** used by the agent to approximate the Q-values for each action in the given state. The network is built using the **Keras** library.

- **Sequential()**: This indicates that the model consists of a linear stack of layers.
- **Dense Layers**: These are fully connected layers where each neuron is connected to every neuron in the previous layer. Here, the model has three **hidden layers**, each with ReLU activation.
- **BatchNormalization**: This layer normalizes the inputs to each layer, improving training stability and accelerating convergence.
- **Dropout**: Dropout is applied with a rate of 0.2 to prevent overfitting by randomly setting some neurons to zero during training.
- **Output Layer**: The final dense layer outputs a value for each action, representing the predicted Q-values. The activation function used here is **linear** since Q-values are continuous numbers.

```
[ ]  def build_model(states, actions):
         model = Sequential()
         model.add(Dense(24, activation='relu', input_shape=states, kernel_initializer='he_normal'))
         model.add(BatchNormalization())
         model.add(Dropout(0.2))
         model.add(Dense(48, activation='relu', kernel_initializer='he_normal'))
         model.add(BatchNormalization())
         model.add(Dropout(0.2))
         model.add(Dense(24, activation='relu', kernel_initializer='he_normal'))
         model.add(Dense(actions, activation='linear'))
         return model
```

```
[ ]  model = build_model(states, actions)
     target_model = build_model(states, actions)
     target_model.set_weights(model.get_weights())

     model.compile(loss='mse', optimizer=Adam(learning_rate=learning_rate_a))
     target_model.compile(loss='mse', optimizer=Adam(learning_rate=learning_rate_a))
```

```
[ ]  model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 24)	72
batch_normalization (BatchNormalization)	(None, 24)	96
dropout (Dropout)	(None, 24)	0
dense_1 (Dense)	(None, 48)	1,200
batch_normalization_1 (BatchNormalization)	(None, 48)	192
dropout_1 (Dropout)	(None, 48)	0
dense_2 (Dense)	(None, 24)	1,176
dense_3 (Dense)	(None, 4)	100

```
Total params: 2,836 (11.08 KB)
Trainable params: 2,692 (10.52 KB)
Non-trainable params: 144 (576.00 B)
```

3.Experience Replay Buffer:

This section implements the **experience replay buffer** using a **deque** (double-ended queue) from Python's collections module.

- **deque(maxlen=max_size)**: The deque is used to store the agent's past experiences. The maxlen parameter ensures that when the buffer reaches its maximum size (replay_memory_size), the oldest experiences are discarded to make space for new ones. This helps prevent the model from overfitting to recent experiences and improves stability by allowing the agent to sample a diverse set of past experiences.

4.Storing Experiences in the Replay Buffer:

This function stores the experience **(state, action, reward, next state, done)** in the replay memory buffer

This function also is called each time the agent interacts with the environment, storing the experience for later training.

```
[ ] from collections import deque
    max_size = replay_memory_size
    memory = deque(maxlen=max_size)
    def store_experience(state, action, reward, next_state, done):
        memory.append((state, action, reward, next_state, done))
```

5. Epsilon-Greedy Policy:

This function implements the **epsilon-greedy policy**, which is used to decide the action that the agent will take.

- **Exploration**: With probability epsilon, the agent chooses a random action (exploring).
- **Exploitation**: With probability 1 - epsilon, the agent selects the action that has the highest Q-value according to the current policy (exploiting what it has learned).

The purpose of this function is to balance between **exploration** (discovering new actions) and **exploitation** (choosing the best-known action).

```
[ ] def epsilon_greedy_policy(q_values, epsilon, action_space):
        if np.random.rand() <= epsilon:
            return random.choice(range(action_space))
        else:
            return np.argmax(q_values)
```

7. Training Loop:

The training loop is the core of the reinforcement learning process in DQN. It enables the agent to interact with the environment, store experiences, and improve its policy over time. The loop runs for a specified number of episodes and involves several key actions at each step.

This part typically includes the following steps:

1. The agent starts by choosing an action based on the epsilon-greedy policy.
2. The environment responds with the new state and reward.
3. The agent stores the experience in the replay buffer.
4. After a certain number of steps, the agent samples a mini-batch of experiences from the replay buffer.
5. The agent updates the Q-network using the Bellman equation and minimizes the loss between predicted and actual Q-values.
6. The target network is periodically synchronized with the policy network.

```
for episode in range(episodes):
    state = env.reset()
    total_reward = 0
    step_count = 0
    done = False

    while not done:
        # Predict Q-values from current state using the policy model
        q_values = model.predict(np.array([state]))[0]

        # Select action using epsilon-greedy strategy
        action = epsilon_greedy_policy(q_values, epsilon, action_space=actions)

        # Take the chosen action in the environment
        next_state, reward, done, _ = env.step(action)

        # Store the experience in memory for replay
        store_experience(state, action, reward, next_state, done)

        # Move to the next state
        state = next_state
        total_reward += reward
        step_count += 1

    # Learn from past experiences if we have enough stored
    if len(memory) >= mini_batch_size:
        mini_batch = random.sample(memory, mini_batch_size)
        states_batch = []
        target_q_values_batch = []
```

```
            # Bellman update
            if done_b:
                q_values_current[action_b] = reward_b  # No future rewards if terminal
            else:
                q_values_current[action_b] = reward_b + discount_factor_g * np.max(q_values_next)

            # Collect data for batch training
            states_batch.append(state_b)
            target_q_values_batch.append(q_values_current)

        # Train the target model on this mini-batch
        model.fit(np.array(states_batch), np.array(target_q_values_batch), epochs=1, verbose=1)

        #  Sync the target model every N steps for stability
        if step_count % network_sync_rate == 0:
            target_model.set_weights(model.get_weights())

    # Check if the current reward is the best
    if total_reward > best_reward_so_far:
        best_reward_so_far = total_reward  # Update best reward
        model.save_weights('best_dqn_weights.weights.h5')  # Save the best model weights
        print(f"Saved best model at Episode {episode + 1} with Reward: {total_reward}")

    # Update epsilon
    epsilon = max(epsilon_min, epsilon * epsilon_decay)

    # Append total reward and steps for analysis
    total_rewards.append(total_reward)
    total_steps.append(step_count)

    # Print progress for the episode
    print(f"Episode {episode + 1}/{episodes} - Reward: {total_reward}, Steps: {step_count}")
```

Chapter 3

Optimization and Evaluation

Chapter3

3. Optimization and Evaluation

3.1 Optimization:

This section outlines the techniques used to optimize and evaluate the performance of a Deep Q-Learning (DQN) agent in a custom maze environment. Optimization was performed in three main areas:

- **Hyperparameter Tuning**
- **Neural Network Architecture Enhancements**
- **Reward Shaping**

Each optimization played a vital role in improving the agent's learning efficiency and generalization capability. Below, we detail each component and how it contributes to the training process.

3.1.1 Hyperparameter Tuning:

Hyperparameters greatly influence the convergence and performance of the agent. Here are the selected values and their purposes:

- **Learning Rate (0.0001):** Slower updates allow the model to learn stably.
- **Discount Factor (γ=0.9):** Encourages long-term rewards over immediate ones.
- **Sync Rate:** Target network is synced every 200 steps to stabilize Q-value updates.
- **Replay Memory:** Allows the agent to reuse past experiences for efficient learning.
- **Epsilon-Greedy Parameters:** Control exploration-exploitation trade-off. The decay prevents premature convergence.

```
learning_rate_a = 0.0001
discount_factor_g = 0.9         # discount rate (gamma)
network_sync_rate = 200        # number of steps the agent takes before syncing the policy and target net
replay_memory_size = 5000       # size of replay memory
mini_batch_size = 32           # size of the training data set sampled from the replay memory
epsilon = 1.0                 # exploration rate
epsilon_decay = 0.957           # exploration rate decay
epsilon_min = 0.1  # Never fully stop exploring
episodes = 25
```

3.1.2 Neural Network Architecture:

The DQN model was enhanced with deeper layers, Leaky ReLU activations, batch normalization, and dropout layers to boost learning capacity and prevent overfitting.

- **Leaky ReLU:** Prevents dying neuron problem and maintains gradient flow.
- **Batch Normalization:** Stabilizes learning by normalizing activations.
- **Dropout:** Prevents overfitting during training.
- **Layer Depth:** Three hidden layers increase the network's capacity to approximate complex Q-value functions.

```
def build_model(state_size, action_size):
    model = Sequential()

    # First hidden layer with Leaky ReLU
    model.add(Dense(256, input_shape=(state_size,)))
    model.add(LeakyReLU(alpha=0.01))
    model.add(BatchNormalization())
    model.add(Dropout(0.2))

    # Second hidden layer with Leaky ReLU
    model.add(Dense(128))
    model.add(LeakyReLU(alpha=0.01))
    model.add(BatchNormalization())

    # Third hidden layer with Leaky ReLU
    model.add(Dense(64))
    model.add(LeakyReLU(alpha=0.01))
    model.add(BatchNormalization())
    model.add(Dropout(0.2))

    # Output layer
    model.add(Dense(action_size, activation='linear'))

    return model
```

3.1.3 Reward Shaping:

Reward shaping is performed inside the environment logic (env.step()), where rewards are calculated based on agent actions, such as avoiding obstacles, reaching the goal, or making progress toward the goal. These shaped rewards are then returned during interaction and used by the agent for learning inside the main training loop."

As it icludes :

- Obstacle penalty.
- Goal reward.
- Progress bonus.
- Obstacle avoidance bonus.

```
if expected_maze_value == 1 or expected_maze_value == 2:
    reward = -2
elif self.avoided_obstacle(old_pos, (new_x, new_y)):
    reward = 5
elif (new_x, new_y) == tuple(self.goal_pos):
    reward = 100
else:
    reward = -0.05

if goal_distance_after < goal_distance_before:
    reward += 1
```

Showing Training Episodes after aplying Optimization Techniques:

```
Episode 1/25 | Composite Score: -256.6| Reward: -427.6 | Steps: 500 | Loss: 2.0403 | ε: 0.957
model saved at Episode 1/25
Episode 2/25 | Composite Score: -286.8| Reward: -478.0 | Steps: 500 | Loss: 1.7588 | ε: 0.916
model saved at Episode 2/25
Episode 3/25 | Composite Score: -307.9| Reward: -513.2 | Steps: 500 | Loss: 2.1426 | ε: 0.876
Episode 4/25 | Composite Score: -103.9| Reward: 100.0 | Steps: 207 | Loss: 1.3621 | ε: 0.839
Episode 5/25 | Composite Score: 155.0| Reward: 220.0 | Steps: 117 | Loss: 1.5533 | ε: 0.803
model saved at Episode 5/25
Episode 6/25 | Composite Score: -107.3| Reward: 92.8 | Steps: 217 | Loss: 1.4544 | ε: 0.768
Episode 7/25 | Composite Score: -498.5| Reward: -834.0 | Steps: 500 | Loss: 1.3461 | ε: 0.735
Episode 8/25 | Composite Score: 41.6| Reward: 53.2 | Steps: 339 | Loss: 1.5096 | ε: 0.704
Episode 9/25 | Composite Score: -144.7| Reward: -241.2 | Steps: 500 | Loss: 1.9147 | ε: 0.673
Episode 10/25 | Composite Score: -127.0| Reward: 69.4 | Steps: 314 | Loss: 1.4105 | ε: 0.644
Episode 11/25 | Composite Score: -94.0| Reward: -157.2 | Steps: 500 | Loss: 1.2701 | ε: 0.617
Episode 12/25 | Composite Score: -50.0| Reward: -89.6 | Steps: 500 | Loss: 1.2386 | ε: 0.590
Episode 13/25 | Composite Score: -62.3| Reward: -104.4 | Steps: 500 | Loss: 1.1913 | ε: 0.565
Episode 14/25 | Composite Score: -94.7| Reward: -155.2 | Steps: 500 | Loss: 1.8729 | ε: 0.540
Episode 15/25 | Composite Score: -30.8| Reward: -51.6 | Steps: 500 | Loss: 1.2454 | ε: 0.517
Episode 16/25 | Composite Score: -37.1| Reward: -62.4 | Steps: 500 | Loss: 671.5385 | ε: 0.495
Episode 17/25 | Composite Score: 7.8| Reward: 15.6 | Steps: 500 | Loss: 1.5813 | ε: 0.474
Episode 18/25 | Composite Score: -30.6| Reward: -48.4 | Steps: 500 | Loss: 2.0859 | ε: 0.453
Episode 19/25 | Composite Score: 83.9| Reward: 139.6 | Steps: 500 | Loss: 2.3069 | ε: 0.434
Episode 20/25 | Composite Score: 131.2| Reward: 215.2 | Steps: 500 | Loss: 1.3862 | ε: 0.415
Episode 21/25 | Composite Score: 280.1| Reward: 456.6 | Steps: 398 | Loss: 1.8606 | ε: 0.397
model saved at Episode 21/25
Episode 22/25 | Composite Score: 192.0| Reward: 320.0 | Steps: 500 | Loss: 1.4285 | ε: 0.380
Episode 23/25 | Composite Score: 187.7| Reward: 312.8 | Steps: 500 | Loss: 2.6620 | ε: 0.364
Episode 24/25 | Composite Score: 226.1| Reward: 376.8 | Steps: 500 | Loss: 4.2099 | ε: 0.348
Episode 25/25 | Composite Score: 165.6| Reward: 276.0 | Steps: 500 | Loss: 5.2232 | ε: 0.333
```

2.3 Evaluation:

Once a reinforcement learning (RL) agent has been trained, it's crucial to evaluate its performance to determine:

- ***Whether learning has occurred effectively.***
- ***How stable and efficient the learning process was.***
- ***If the agent can generalize and solve the problem consistently.***

Evaluation involves analyzing various metrics collected during training, such as:

- ***Total reward per episode: Measures how well the agent is achieving the task.***
- ***Number of steps per episode: Indicates efficiency and policy quality.***
- ***Training loss per episode: Reflects the learning progress and stability.***

By visualizing these metrics, we can identify trends, spot instability, and fine-tune hyperparameters or the environment design further.

3.2.1 Rewards per Episode:

This code shows the total accumulated reward the agent received in each episode. It reflects how well the agent performed over time.

Interpretation:

- An upward trend indicates that the agent is learning to achieve higher rewards (e.g., reaching the goal more often or avoiding penalties).
- A flat or noisy pattern may suggest unstable learning, ineffective policy updates, or inadequate reward signals.

As seen in the plot that shows the results of our agent:

In early episodes, the reward is low due to exploration. Over time, as the policy improves, the rewards increase and stabilize.

```
# Plotting rewards per episode
plt.figure(figsize=(6, 4))
plt.plot(range(episodes), episode_rewards, color='blue', label='Reward')
plt.xlabel('Episodes')
plt.ylabel('Reward')
plt.title('Rewards per Episode')
plt.grid(True)
plt.show()
```

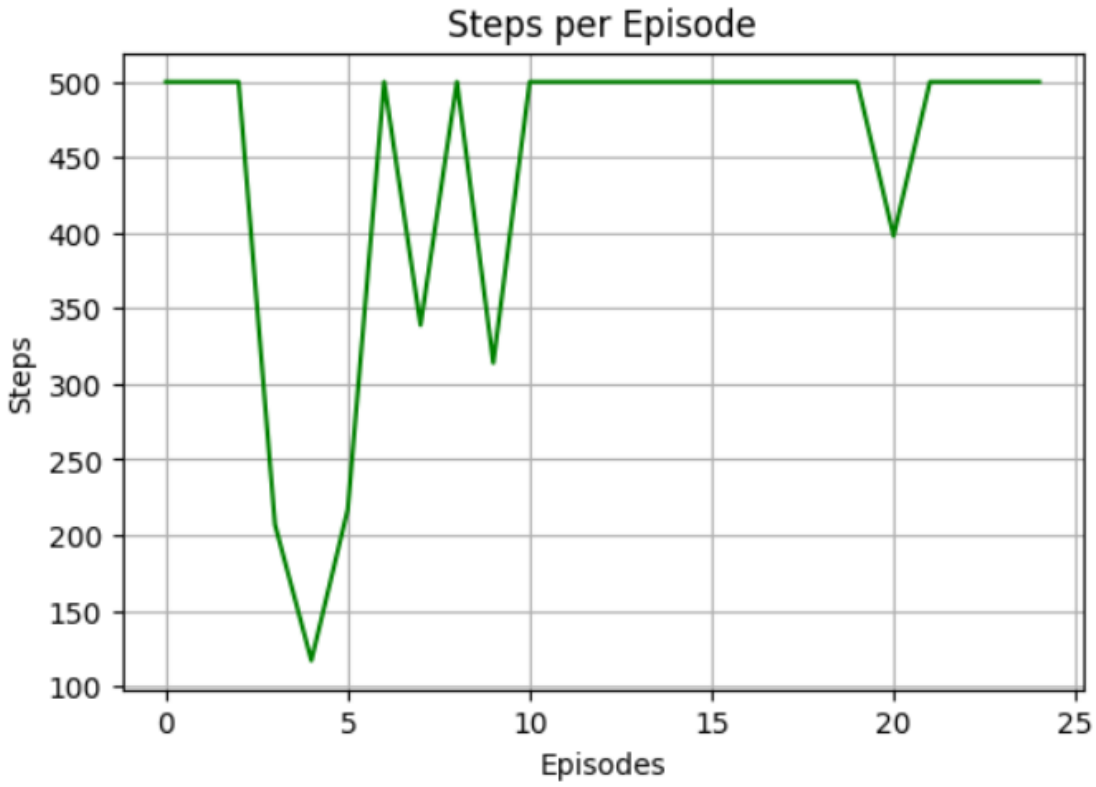
3.2.2 Steps per Episode:

This code visualizes how many **steps** the agent took to complete each episode (either reaching the goal or failing).

Interpretation:

- A **decrease in steps** can indicate improved efficiency, as the agent finds shorter paths to the goal.
- A **spike in steps** may mean the agent is getting stuck or exploring inefficient paths.

```
# Plotting steps per episode
plt.figure(figsize=(6, 4))
plt.plot(range(episodes), episode_steps, color='green', label='Steps')
plt.xlabel('Episodes')
plt.ylabel('Steps')
plt.title('Steps per Episode')
plt.grid(True)
plt.show()
```



3.2.3 Loss per Episode:

This plot tracks the loss computed during training, which quantifies the difference between predicted Q-values and target Q-values (Bellman updates).

Interpretation:

- **A declining loss trend indicates the agent's predictions are becoming more accurate.**
- **High variance or sharp spikes might indicate instability or overfitting.**

```
# Plotting loss per episode
plt.figure(figsize=(6, 4))
plt.plot(range(episodes), episode_losses, color='red', label='Steps')
plt.xlabel('Episodes')
plt.ylabel('losses')
plt.title('losses per Episode')
plt.grid(True)
plt.show()
```