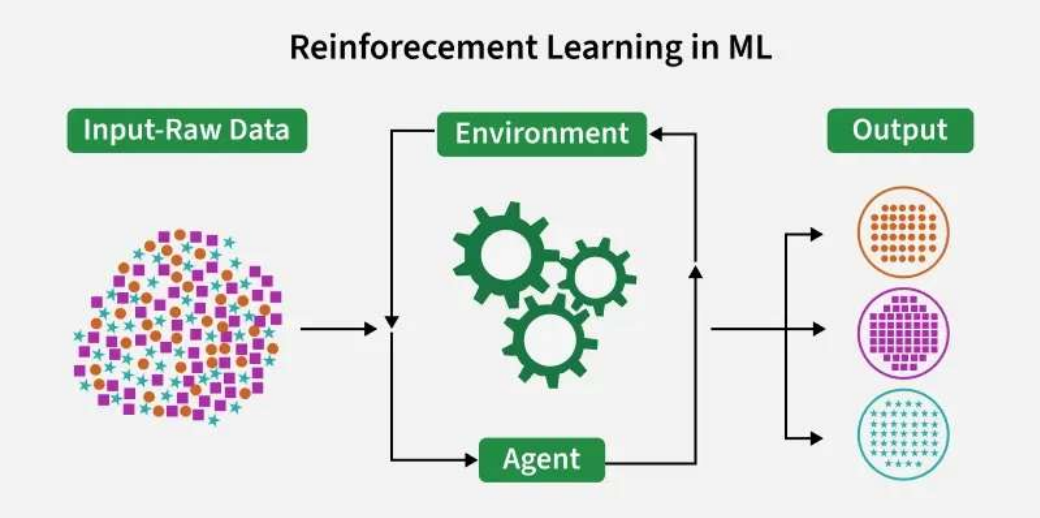
Reinforcement Learning

Reinforcement Learning (RL) is a branch of machine learning that focuses on how agents can learn to make decisions through trial and error to maximize cumulative rewards. RL allows machines to learn by interacting with an environment and receiving feedback based on their actions. This feedback comes in the form of rewards or penalties.



Reinforcement Learning revolves around the idea that an agent (the learner or decision-maker) interacts with an environment to achieve a goal. The agent performs actions and receives feedback to optimize its decision-making over time.

- **Agent**: The decision-maker that performs actions.
- **Environment**: The world or system in which the agent operates.
- **State**: The situation or condition the agent is currently in.
- **Action**: The possible moves or decisions the agent can make.
- **Reward**: The feedback or result from the environment based on the agent's action.

Core Components

Let's see the core components of Reinforcement Learning

1. Policy

- Defines the agent's behavior i.e maps states for actions.
- Can be simple rules or complex computations.
- **Example**: An autonomous car maps pedestrian detection to make necessary stops.

2. Reward Signal

- Represents the goal of the RL problem.
- Guides the agent by providing feedback (positive/negative rewards).

- **Example**: For self-driving cars rewards can be fewer collisions, shorter travel time, lane discipline.

3. Value Function

- Evaluates long-term benefits, not just immediate rewards.
- Measures desirability of a state considering future outcomes.
- **Example**: A vehicle may avoid reckless maneuvers (short-term gain) to maximize overall safety and efficiency.

4. Model

- Simulates the environment to predict outcomes of actions.
- Enables planning and foresight.
- **Example**: Predicting other vehicles' movements to plan safer routes.

Working of Reinforcement Learning

The agent interacts iteratively with its environment in a feedback loop:

- The agent observes the current state of the environment.
- It chooses and performs an action based on its policy.
- The environment responds by transitioning to a new state and providing a reward (or penalty).
- The agent updates its knowledge (policy, value function) based on the reward received and the new state.
- This cycle repeats with the agent balancing exploration (trying new actions) and exploitation (using known good actions) to maximize the cumulative reward over time.

This process is mathematically framed as a Markov Decision Process (MDP) where future states depend only on the current state and action, not on the prior sequence of events.

Implementing Reinforcement Learning

Let's see the working of reinforcement learning with a maze example:

Step 1: Import libraries and Define Maze, Start and Goal

We will import the required libraries such as numpy and matplotlib.

- The maze is represented as a 2D NumPy array.
- Zero values are safe paths; ones are obstacles the agent must avoid.
- Start and goal define the positions where the agent begins and where it aims to reach.

```
import numpy as np

import matplotlib.pyplot as plt
```

```
from matplotlib.colors import ListedColormap


maze = np.array([
    [0, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 1],
    [1, 1, 1, 0, 1, 0, 1, 1, 0, 1],
    [1, 0, 0, 0, 0, 0, 1, 0, 0, 1],
    [1, 0, 1, 1, 1, 1, 1, 0, 1, 1],
    [1, 0, 1, 0, 0, 0, 0, 0, 1, 1],
    [1, 0, 1, 0, 1, 1, 1, 0, 1, 1],
    [1, 0, 1, 0, 1, 0, 0, 0, 1, 1],
    [1, 0, 1, 0, 1, 0, 1, 0, 0, 1],
    [1, 1, 1, 0, 1, 1, 1, 1, 0, 0]
])


start = (0, 0)
goal = (9, 9)
```

Step 2: Define RL Parameters and Initialize Q-Table

We will define RL parameters;

- **num_episodes**: Number of times the agent will attempt to navigate the maze.
- **alpha**: Learning rate that controls how much new information overrides old information.
- **gamma**: Discount factor giving more weight to immediate rewards.
- **epsilon**: Probability of exploration vs exploitation; starts higher to explore more.
- Rewards are set to penalize hitting obstacles, reward reaching the goal and slightly penalize each step to find shortest paths.
- **actions define possible moves:** left, right, up, down.
- **Q** is the Q-Table initialized to zero; it stores expected rewards for each state-action pair.

```
num_episodes = 5000
alpha = 0.1
gamma = 0.9
```

```
epsilon = 0.5


reward_fire = -10
reward_goal = 50
reward_step = -1


actions = [(0, -1), (0, 1), (-1, 0), (1, 0)]


Q = np.zeros(maze.shape + (len(actions),))
```

Step 3: Helper Function for Maze Validity and Action Selection

We will define helper function,

- **is_valid** ensures the agent can only move inside the maze and avoids obstacles.
- **choose_action** implements exploration (random action) vs exploitation (best learned action) strategy.

```
def is_valid(pos):
    r, c = pos
    if r < 0 or r >= maze.shape[0]:
        return False
    if c < 0 or c >= maze.shape[1]:
        return False
    if maze[r, c] == 1:
        return False
    return True


def choose_action(state):
    if np.random.random() < epsilon:
        return np.random.randint(len(actions))
    else:
        return np.argmax(Q[state])
```

Step 4: Train the Agent with Q-Learning Algorithm

We will train the agent by running multiple episodes for the agent to learn. During each episode, the agent selects actions and updates its Q-Table using the Q-learning update rule:

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

Where:

- s: current state (agent's position in the maze)
- a: action taken at state s (e.g., move left, right, up, down)
- r: reward received after taking action a
- s': next state after performing action a
- $\alpha(alpha)$: learning rate controlling how much new information overrides old
- $\gamma(gamma)$ discount factor for future rewards

This equation updates the Q-value based on the current reward and the best future Q-value possible from the next state.

```
rewards_all_episodes = []


for episode in range(num_episodes):
    state = start
    total_rewards = 0
    done = False


    while not done:
        action_index = choose_action(state)
        action = actions[action_index]


        next_state = (state[0] + action[0], state[1] + action[1])


        if not is_valid(next_state):
            reward = reward_fire
            done = True
        elif next_state == goal:
            reward = reward_goal
            done = True
```

```
        else:
            reward = reward_step

        old_value = Q[state][action_index]
        next_max = np.max(Q[next_state]) if is_valid(next_state) else 0

        Q[state][action_index] = old_value + alpha * \
            (reward + gamma * next_max - old_value)

        state = next_state
        total_rewards += reward

    global epsilon
    epsilon = max(0.01, epsilon * 0.995)
    rewards_all_episodes.append(total_rewards)
```

Step 5: Extract the Optimal Path after Training

- This function follows the highest Q-values at each state to extract the best path.
- It stops when the goal is reached or no valid next moves are available.
- The visited set prevents cycles.

```
def get_optimal_path(Q, start, goal, actions, maze, max_steps=200):
    path = [start]
    state = start
    visited = set()

    for _ in range(max_steps):
        if state == goal:
            break
        visited.add(state)

        best_action = None
```

```
        best_value = -float('inf')

        for idx, move in enumerate(actions):
            next_state = (state[0] + move[0], state[1] + move[1])

            if (0 <= next_state[0] < maze.shape[0] and
                0 <= next_state[1] < maze.shape[1] and
                maze[next_state] == 0 and
                    next_state not in visited):

                if Q[state][idx] > best_value:
                    best_value = Q[state][idx]
                    best_action = idx

        if best_action is None:
            break

        move = actions[best_action]
        state = (state[0] + move[0], state[1] + move[1])
        path.append(state)

    return path



optimal_path = get_optimal_path(Q, start, goal, actions, maze)
```

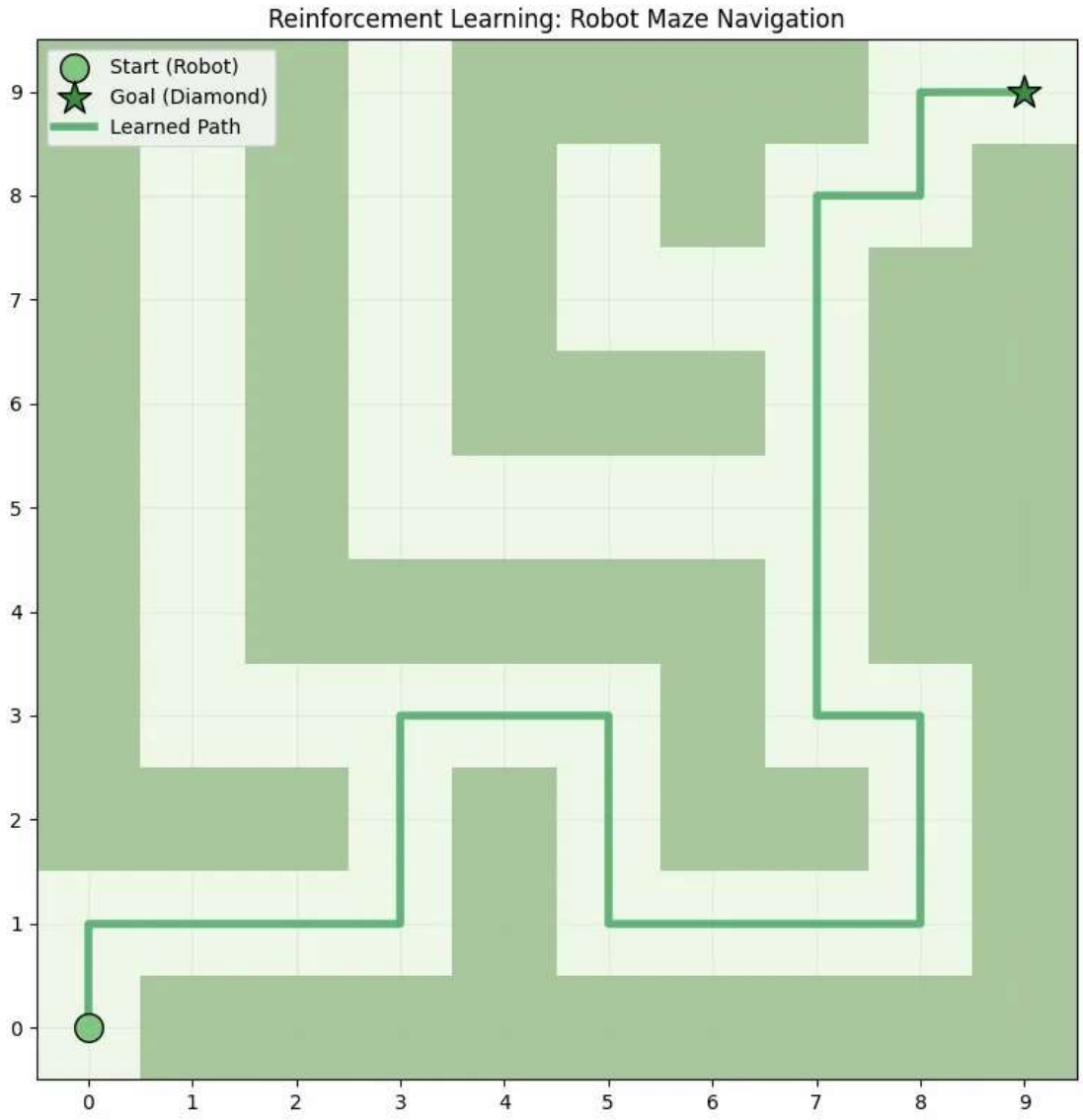
Step 6: Visualize the Maze, Robot Path, Start and Goal

- The maze and path are visualized using a calming green color palette.
- The start and goal positions are visually highlighted.
- The learned path is drawn clearly to demonstrate the agent's solution.

```
def plot_maze_with_path(path):
```

```
    cmap = ListedColormap(['#eef8ea', '#a8c79c'])

    plt.figure(figsize=(8, 8))
    plt.imshow(maze, cmap=cmap)

    plt.scatter(start[1], start[0], marker='o', color='#81c784', edgecolors='black',
                s=200, label='Start (Robot)', zorder=5)
    plt.scatter(goal[1], goal[0], marker='*', color='#388e3c', edgecolors='black',
                s=300, label='Goal (Diamond)', zorder=5)

    rows, cols = zip(*path)
    plt.plot(cols, rows, color='#60b37a', linewidth=4,
             label='Learned Path', zorder=4)

    plt.title('Reinforcement Learning: Robot Maze Navigation')
    plt.gca().invert_yaxis()
    plt.xticks(range(maze.shape[1]))
    plt.yticks(range(maze.shape[0]))
    plt.grid(True, alpha=0.2)
    plt.legend()
    plt.tight_layout()
    plt.show()
plot_maze_with_path(optimal_path)
```

Output:



Maze

As we can see that the model successfully reached the destination by finding the right path.

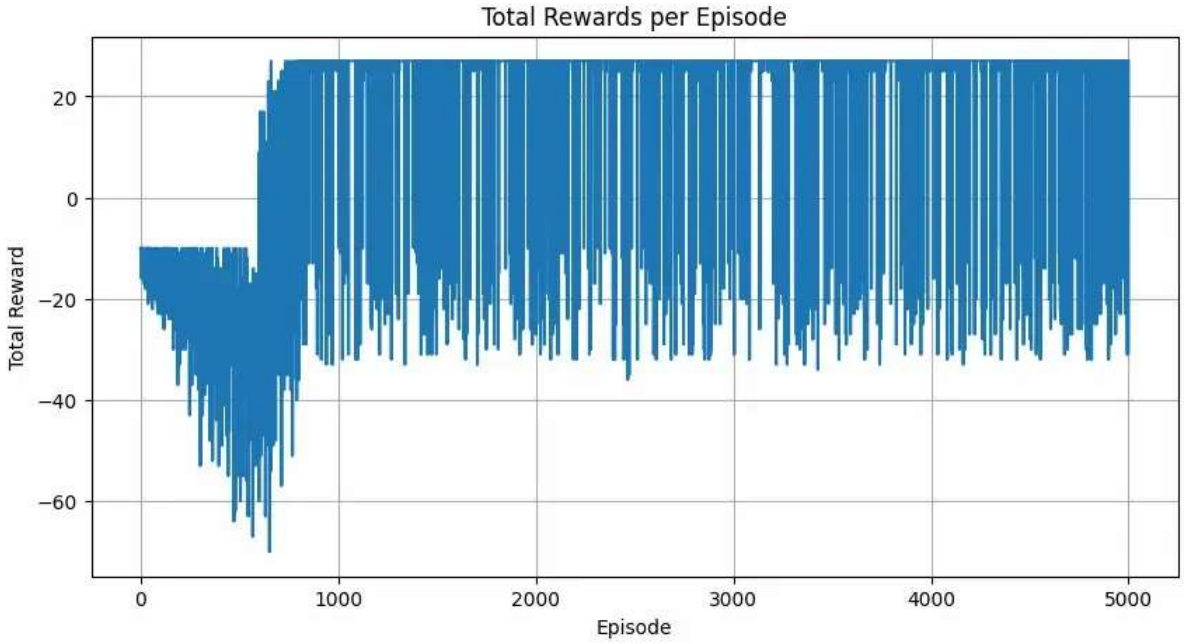
Step 7: Plot Rewards per Training

- This plot shows how the agent's overall performance improves across training episodes.
- We can observe the total reward trend increasing as the agent learns over time.

```
def plot_rewards(rewards):
    plt.figure(figsize=(10, 5))
    plt.plot(rewards)
    plt.title('Total Rewards per Episode')
    plt.xlabel('Episode')
```

```
    plt.ylabel('Total Reward')
    plt.grid(True)
    plt.show()
plot_rewards(rewards_all_episodes)
```

Output:



Rewards

Types of Reinforcements

1. Positive Reinforcement: Positive Reinforcement is defined as when an event, occurs due to a particular behavior, increases the strength and the frequency of the behavior. In other words, it has a positive effect on behavior.

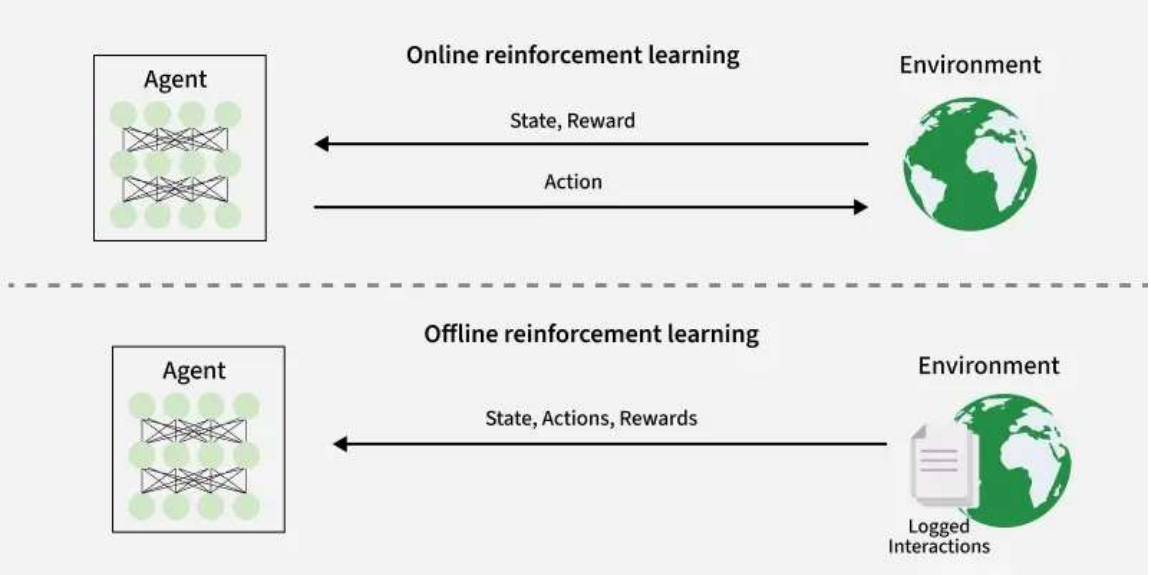
- **Advantages**: Maximizes performance, helps sustain change over time.
- **Disadvantages**: Overuse can lead to excess states that may reduce effectiveness.

2. Negative Reinforcement: Negative Reinforcement is defined as strengthening of behavior because a negative condition is stopped or avoided.

- **Advantages**: Increases behavior frequency, ensures a minimum performance standard.
- **Disadvantages**: It may only encourage just enough action to avoid penalties.

Online vs. Offline Learning

Reinforcement Learning can be categorized based on how and when the learning agent acquires data from its environment, dividing the methods into online RL and offline RL (also known as batch RL).



Online vs Offline RL

- In online RL, the agent learns by actively interacting with the environment in real-time. It collects fresh data during training by executing actions and observing immediate feedback as it learns.
- Offline RL trains the agent exclusively on a pre-collected static dataset of interactions generated by other agents, human demonstrations or historical logs. The agent does not interact with the environment during learning.

Aspect	**Online RL**	**Offline RL**
Data Acquisition	Direct, real-time interaction with environment	Static, pre-collected dataset
Adaptivity	High, continuously adapts	Limited, depends on dataset coverage
Suitability	When environment access or simulation is feasible	When environment interaction is costly or risky
Challenges	Resource-intensive, potentially unsafe	Distributional shift, counterfactual inference issues

Application

- **Robotics**: RL is used to automate tasks in structured environments such as manufacturing, where robots learn to optimize movements and improve efficiency.
- **Games**: Advanced RL algorithms have been used to develop strategies for complex games like chess, Go and video games, outperforming human players in many instances.
- **Industrial Control**: RL helps in real-time adjustments and optimization of industrial operations, such as refining processes in the oil and gas industry.
- **Personalized Training Systems**: RL enables the customization of instructional content based on an individual's learning patterns, improving engagement and effectiveness.

Advantages

- Solves complex sequential decision problems where other approaches fail.
- Learns from real-time interaction, enabling adaptation to changing environments.
- Does not require labeled data, unlike supervised learning.
- Can innovate by discovering new strategies beyond human intuition.
- Handles uncertainty and stochastic environments effectively.

Disadvantages

- Computationally intensive, requiring large amounts of data and processing power.
- Reward function design is critical; poor design leads to unintended behaviors.
- Not suitable for simple problems where traditional methods are more efficient.
- Challenging to debug and interpret, making it hard to explain decisions.
- Exploration-exploitation trade-off requires careful balancing to optimize learning.