# University of Isfahan

## Maze Project

Authors:

Melika Shirian

Kianoosh Vadaei

**Fundamentals and Applications of Artificial Intelligence**

Dr. Hossein Karshenas

2023-11-08

## Introduction

Reinforcement Learning (RL) is a branch of artificial intelligence where agents learn optimal decision-making by interacting with environments and receiving feedback through rewards or penalties. Unlike traditional machine learning, RL is based on trial-and-error learning, making it suitable for complex, dynamic scenarios. This report explores a specific RL project, highlighting the problem, methodology, and outcomes, showcasing RL's application in solving real-world challenges.

SARSA, or State-Action-Reward-State-Action, is an on-policy reinforcement learning algorithm. It adapts to changes in the agent's policy as it learns from real-time experiences. Using a temporal difference learning approach, SARSA updates Q-values, representing expected future rewards. Its distinctive feature is the direct learning from the current policy, making it apt for scenarios where actions influence subsequent states. This report explores SARSA's practical implementation, highlighting its strengths and limitations.

## Code Explanation

In this project, we endeavored to implement learning for an intelligent agent in a maze environment using the SARSA algorithm.

Decision-making in this algorithm relies on the Q-matrix. Subsequently, the agent, over a specified number of game episodes, takes a limited number of steps in each episode. At each step, the agent chooses an action, and the Q-table is updated accordingly.

```python
class Maze:

    @staticmethod
    def choose_action(state):
        action = 0
        epsilon = 0.1

        epsilon_choice = np.random.choice((1,0) , p=[epsilon, 1-epsilon])

        # epsilon_choice = np.random.uniform(0, 1)

        if epsilon_choice:
            action = env.action_space.sample()
        else:
            action = np.argmax(Q[state, :])
        return action

    @staticmethod
    def update(state, state2, reward, action, action2):
        predict = Q[state, action]
        target = reward + 0.9 * Q[state2, action2]
        Q[state, action] = Q[state, action] + 0.1 * (target - predict)
```

Firstly, let's delve into the explanation of the **maze** class. This class comprises two static functions. Let's begin by elucidating the **choose_action(state)** function:

In this function, initially, a default action is considered. Then, utilizing the e-epsilon algorithm, an action is chosen. This occurs as follows: a small value is assigned to epsilon, for example, 0.1. Subsequently, with a probability of epsilon, a random action is chosen, and with a probability of 1-epsilon, the best action in the

current state is selected, defined as argmax(Q[state, :]). The chosen action is then returned at the end of the function.

Following that, the **update(state, state2, reward, action, action2)** function:

In this function, the Q-matrix is updated based on the following equation.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

Here, $\boldsymbol{\alpha}$ represents the learning rate, typically set to 0.1, and $\boldsymbol{\gamma}$ is the discount factor determining the weight given to future rewards. The value of this variable is commonly set to 0.9.

```
# Create an environment
env = gym.make("maze-random-10x10-plus-v0")
observation = env.reset()

NUM_EPISODES = 1000

total_win_during_learning = 0
```

This project has been implemented using the Gym library provided by OpenAI. Here, we begin by creating the game environment.

```
Q = np.zeros((100, 4))
```

Then, we define the Q-matrix with dimensions 100 by 4, representing the number of states and actions, respectively.

```python
for episode in range(NUM_EPISODES):
    #print(f'Episode: {episode}')

    training = 0
    state1 = 0
    action1 = Maze.choose_action(state1)

    while training < 200:   #or not done
        env.render()

        # Getting the next state
        state2, reward, done, truncated = env.step(action1)
        state2 = state2[0] * 10 + state2[1] #converting state from 2d into 1d out of 100

        # Choosing the next action
        action2 = Maze.choose_action(state2)

        # Learning the Q-value
        Maze.update(state1, state2, reward, action1, action2)

        state1 = state2
        action1 = action2

        # Updating the respective vaLues
        training += 1

        # If at the end of learning process
        if state1 == 99:
            reward = 1
        else:
            reward = (-0.5)/state1

        if done or truncated:
            observation = env.reset()
            total_win_during_learning +=1
            # print("WON  ******************************!")
            break
```

Now, we proceed to execute the algorithm. The algorithm runs in two main loops. The first loop iterates over game episodes, and the second loop operates as follows: for each episode, the agent, within a maximum of 200 steps, selects an action based on the described function, applies that action, and transitions to **state2**. Next, **state2** is translated into a number between 0 and 100 using the definitions of **x** and **y**. Subsequently, the second action is chosen from **state2**, and the Q-matrix is updated.

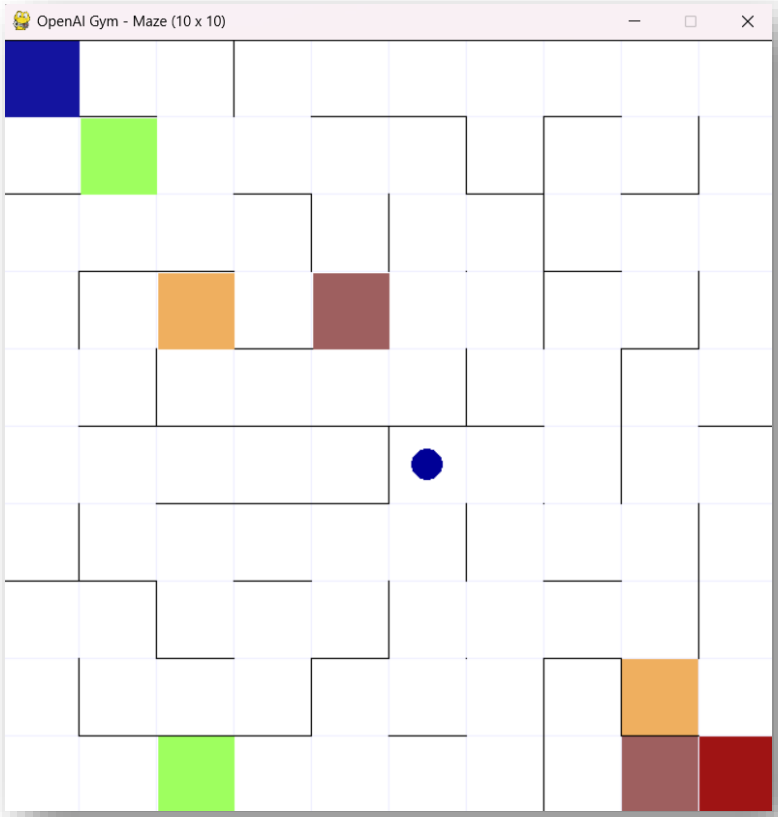
At the end of each step, the current state and action are replaced. Finally, if the current state is the goal state, the reward becomes 1; otherwise, it becomes 0.5 divided by **state1**.

Ultimately, a check is performed to reset the environment if the agent has reached the goal. The second loop ceases if the goal is achieved.

```python
#============================================================
# TESTING THE AGENT AFTER LEARNING USING SARSA ALGORITHN.
#============================================================

next_state = 0
total_win_after_learning = 0
observation = env.reset()
for i in range(10000):
    env.render()
    action = np.argmax(Q[next_state, :])
    # print(f'action is: {action}')

    next_state, reward, done, truncated = env.step(action)
    next_state = next_state[0] * 10 + next_state[1]

    if done or truncated:
        total_win_after_learning +=1
        observation = env.reset()

print(f'total win in 10000 itterarion after learning: {total_win_after_learning}')

print(f'total win during learning is : {total_win_during_learning}')
env.close()
```

In the final section of the code, after the agent has learned, it goes through the environment from scratch for 10,000 steps. The total number of wins achieved by the agent in these 10,000 steps is printed as output. Additionally, the agent's wins during the learning process are also printed.

Finally, the environment is closed in the last line of the code, concluding the program.




## Usage

To use this environment, follow these steps after downloading the package from this link:

**Script Usage:**

1. Navigate to the downloaded package directory using the terminal:

   ```
   cd path/to/downloaded/package
   ```

2. Run the following command to install the package:

   ```
   python setup.py install
   ```

3. Run the Script:

   ```
   Python main.py
   ```

```
python main.py
```

**References**


- GeeksforGeeks. "SARSA Reinforcement Learning." Retrieved from this link.
- OpenAI. "ChatGPT." https://www.openai.com/
- Russell, Stuart, and Norvig, Peter. "Artificial Intelligence: A Modern Approach." (Book)