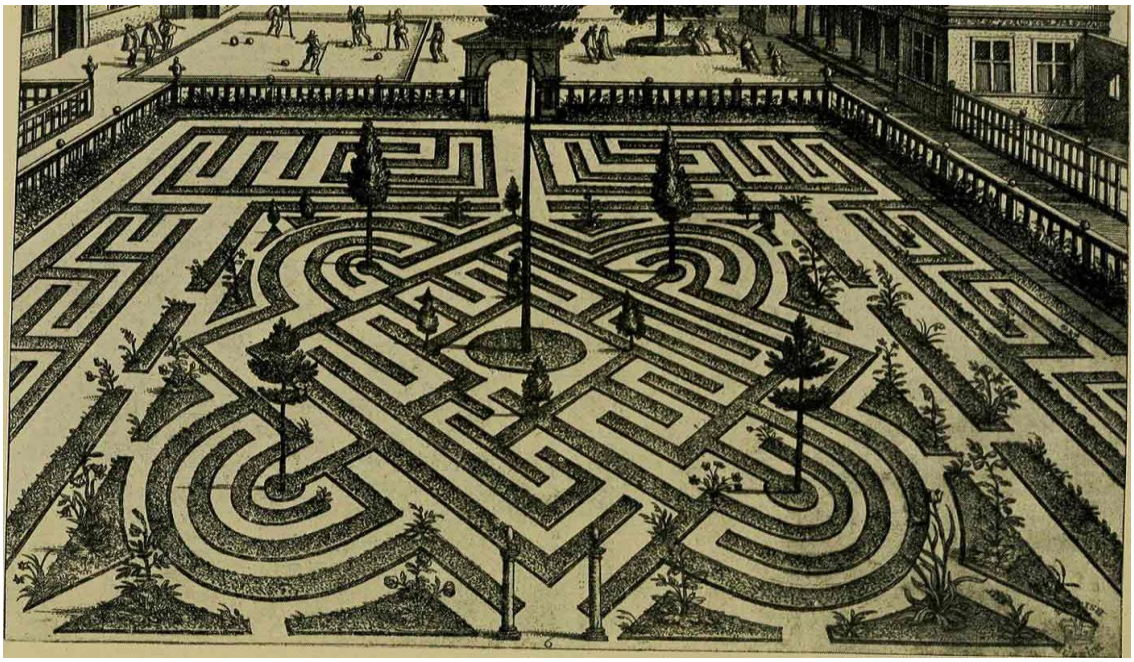
Exploring Reinforcement Learning Techniques for Maze Solving

Kuldeep, Yashwanth, Anupriya, Ishaan Garg & Pranav

ArIES, IIT Roorkee



1. INTRODUCTION: -

Maze Solving is a Classical problem in the field of Machine learning. Mazes are complex environments which require the agent to navigate through the maze and find the optimal policy to reach the goal state from the start state.

Recently, reinforcement learning (RL) has emerged as a promising approach for solving maze problems. RL is a subfield of machine learning that enables agents to learn from experience by trial-and-error. In the context of maze solving, RL agents learn to navigate through the maze environment by receiving rewards or penalties based on their actions. By optimizing their behavior over time, these agents can find optimal solutions to maze problems.

In this paper, we present a simple and easy-to-understand approach for using RL to solve mazes. We focus on small, easy mazes that can be easily generated and provide a beginner-friendly introduction to RL and Q-learning, a popular RL algorithm for solving maze problems. We also demonstrate how to implement these algorithms using Python and popular machine learning libraries such as TensorFlow and Keras.

Overall, this paper offers a simple and effective approach for navigating through easy mazes using RL.

2. WHAT IS REINFORCEMENT LEARNING?

Reinforcement learning (RL) is a type of machine learning that enables agents to learn from experience by trial-and-error. The basic idea behind RL is to train an agent to make a sequence of decisions that maximize a cumulative reward signal over time. The agent interacts with an environment and receives feedback in the form of rewards or penalties based on its actions. By optimizing its behavior over time, the agent can learn to achieve its goals more effectively.

In RL, the agent learns from experience by following a trial-and-error approach. The agent takes actions in the environment and receives feedback in the form of rewards or penalties based on its actions. The goal of the agent is to learn a policy that maps states to actions that maximize the cumulative reward over time. The agent uses this policy to make decisions in the environment and receive feedback. Over time, the agent learns to adjust its policy to maximize its reward.

RL is a powerful tool for solving complex problems that involve decision-making and uncertainty. RL has been successfully applied to a wide range of domains, including game playing, robotics, and finance. One of the key advantages of RL is its ability to learn from experience without requiring a pre-defined set of rules or a specific problem formulation. This makes RL a flexible and adaptable approach that can be used in many different applications.

3. Defining the Environment

First, we need to define the maze environment, because there is no such environment available on “gym”.

Here is the code defining the maze environment -

- Importing necessary modules and making maze environment

```
# import necessary modeules
import random
import numpy as np
import matplotlib.pyplot as plt
import IPython.display as display
import time
from PIL import Image
from gym import spaces

# creating the maze env
class MazeEnv():

    def __init__(self, maze, start=(0, 0)):
        super(MazeEnv, self).__init__()

        self.maze = maze
        self.start = start
        self.end =(self.maze.shape[0]-1, self.maze.shape[1]-1) #end point of maze is bottom right of the maze.
        self.current_pos = self.start
        self.agent_pos = self.start

        self.action_space = [0,1,2,3] # 4 actions , up ,down ,left and right
        self.observation_space = spaces.Box(low=0, high=255, shape=(10, 10, 2))

    def reset(self): #reset the agent position to start
        self.current_pos = self.start
        self.agent_pos = self.start
        obs = self._get_obs()
        return obs
```

```
def step(self, action): #gives new state,reward and if the maze is solved or not, after taking action a in current state
    new_pos = self._get_new_pos(action)
    reward = self._get_reward(new_pos)
    done = self._get_done(new_pos)
    if self.maze[new_pos] == 0:
        self.current_pos = new_pos
        self.agent_pos = new_pos
    obs = self._get_obs()
    return obs, reward, done, {}

def render(self, mode='human'): #renders the maze and the agent
    if mode not in ['rgb_array', 'human']:
        raise ValueError(f"Invalid input '{mode})
    if mode == 'rgb_array':
        img = np.zeros((self.maze.shape[0], self.maze.shape[1], 3))
        img[self.maze == 1] = [1, 1, 1]
        img[self.agent_pos[0], self.agent_pos[1]] = [0, 0, 1]
        if self.start == self.agent_pos:
            pass
        else:
            img[self.start[0], self.start[1]] = [255, 0, 0]
        if self.end==self.agent_pos:
            pass
        else:
            img[self.end[0], self.end[1]] = [0, 255, 0]
        img = np.clip(img, 0, 1)
        return img
    elif mode == 'human':
        img = self.render(mode='rgb_array')
        plt.imshow(img)
        plt.show()
        plt.clf()
```

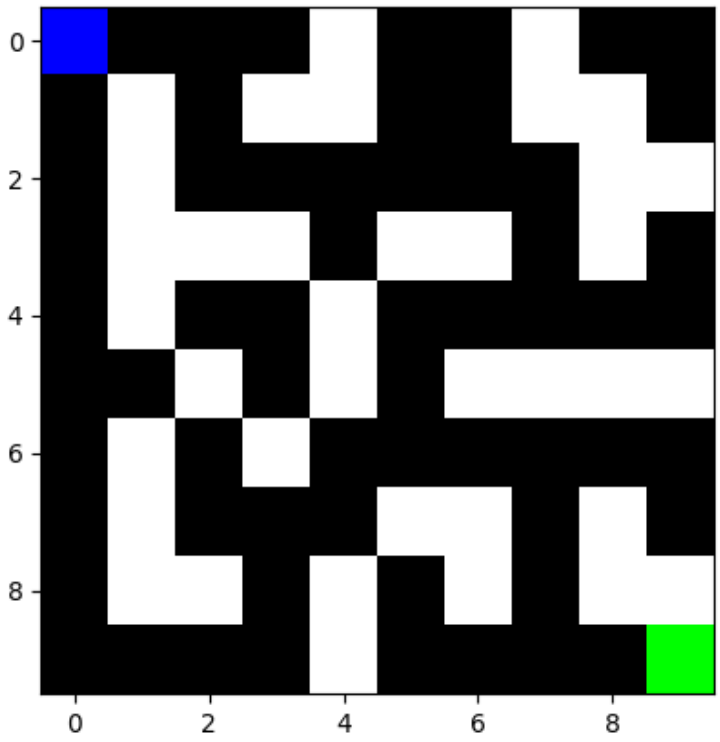
```
def _get_obs(self):  #gives the observation from maze
    obs = np.zeros(self.maze.shape + (2,))
    obs[self.maze == 1] = [0, 0]
    obs[self.maze == 0] = [255, 255]
    obs[self.start] = [0, 255]
    obs[self.end] = [255, 0]
    obs[self.agent_pos] = [255, 255]
    return obs

def _get_new_pos(self, action): #new position of agent after taking action a
    row, col = self.current_pos
    if action == 0: # move up
        row = max(row-1, 0)
    elif action == 1: # move down
        row = min(row+1, self.maze.shape[0]-1)
    elif action == 2: # move left
        col = max(col-1, 0)
    elif action == 3: # move right
        col = min(col+1, self.maze.shape[1]-1)
    return (row, col)

def _get_reward(self, new_pos): #reward associated with states
    if new_pos == self.end:
        return 100
    elif self.maze[new_pos[0], new_pos[1]] == 1:
        return -100
    else:
        return -1


def _get_done(self, new_pos):
    return new_pos == self.end
```

The input to create the environment should be a NumPy array made up using 0 and 1. where 0 defines the free path where the agent can move freely and 1 represents the restricted area I.e., walls.

For example: -

```
maze = np.array([[0, 0, 0, 0, 1, 0, 0, 1, 0, 0],
                 [0, 1, 0, 1, 1, 0, 0, 1, 1, 0],
                 [0, 1, 0, 0, 0, 0, 0, 0, 1, 1],
                 [0, 1, 1, 1, 0, 1, 1, 0, 1, 0],
                 [0, 1, 0, 0, 1, 0, 0, 0, 0, 0],
                 [0, 0, 1, 0, 1, 0, 1, 1, 1, 1],
                 [0, 1, 0, 1, 0, 0, 0, 0, 0, 0],
                 [0, 1, 0, 0, 0, 1, 1, 0, 1, 0],
                 [0, 1, 1, 0, 1, 0, 1, 0, 1, 1],
                 [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]])
env= MazeEnv(maze,start=(0,0))
```

This is how the env will convert the NumPy array to maze with blue agent being at the starting point and representing end point or goal state as green.



4. Early Approach

Earlier we used a CNN model to predict the optimal action for the agent in a particular state.

Here is the code for the earlier approach using CNN model-

```
class MazeAgent:
    def __init__(self, env):
        self.env = env
        self.state_size = env.observation_space.shape
        self.action_size = env.action_space.n
        self.gamma = 0.95
        self.epsilon = 1.0
        self.epsilon_min = 0.05
        self.epsilon_decay = 0.95
        self.learning_rate = 0.1
        self.memory = []
        self.model = self._build_model()

    def _build_model(self):
#         model = Sequential()
#         model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=self.state_size))
#         model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
#         model.add(Flatten())
#         model.add(Dense(256, activation='relu'))
#         model.add(Dense(self.action_size, activation='linear'))
        model = tf.keras.models.load_model("my_model")
        model.compile(loss='mse', optimizer=Adam(learning_rate=self.learning_rate))
        return model

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))

    def act(self, state):
        if np.random.uniform(0,1) <= self.epsilon:
            action = self.env.action_space.sample()
        else:
            act_values = self.model.predict(state)
            action=np.argmax(act_values[0])
        return action
```

```
    def replay(self):

        if len(self.memory) < 1:
            return

        minibatch = self.memory
        states = np.zeros(
            (len(self.memory), self.state_size[0], self.state_size[1], self.state_size[2]))
        next_states = np.zeros(
            (len(self.memory), self.state_size[0], self.state_size[1], self.state_size[2]))
        actions, rewards, dones = [], [], []

        for i in range(len(self.memory)):
            state, action, reward, next_state, done = minibatch[i]
            states[i] = state
            next_states[i] = next_state
            actions.append(action)
            rewards.append(reward)
            dones.append(done)

        actions = np.array(actions)
        rewards = np.array(rewards)
        dones = np.array(dones)

        targets = self.model.predict(states)
        q_values_next = self.model.predict(next_states)
        targets[np.arange(len(self.memory)), actions.astype(
            int)] = rewards + self.gamma * np.amax(q_values_next, axis=1) * (1 - dones)

        self.model.fit(states, targets, epochs=1, verbose=0)
```

```
def train(self, episodes):

    for episode in range(episodes):
        state = self.env.reset()
        state = np.reshape(state, [1, self.state_size[0], self.state_size[1], self.state_size[2]])
        score = 0
        for time in range(200):
            action = self.act(state)
            next_state, reward, done, info = self.env.step(action)
            next_state = np.reshape(next_state, [1, self.state_size[0], self.state_size[1], self.state_size[2]])
            self.remember(state, action, reward, next_state, done)
            score += reward
            state = next_state
            if done:
                print(f"episode: {episode}/{episodes}, score: {score}")
                break
            self.replay()
        if self.epsilon >= self.epsilon_min:
            self.epsilon = self.epsilon*self.epsilon_decay
def play(self, render=True):
    state = self.env.reset()
    state = np.reshape(state, [1, self.state_size[0], self.state_size[1], self.state_size[2]])
    total_reward = 0
    while True:
        action = self.act1(state)
        next_state, reward, done, info = self.env.step(action)
        total_reward += reward
        if render:
            self.env.render()
        if done:
            break
        state = np.reshape(next_state, [1, self.state_size[0], self.state_size[1], self.state_size[2]])
    print(f"Total reward: {total_reward}")

def save_model(self, name):
    self.model.save(name)
```

The code above is an implementation of a Maze Agent class that uses a convolutional neural network (CNN) for reinforcement learning. The objective of the agent is to learn how to navigate a maze environment by finding the optimal path to a goal location while avoiding obstacles.

- The **constructor** of the Maze Agent class initializes the environment, the size of the state and action spaces, as well as the parameters for the reinforcement learning algorithm, such as the discount factor, exploration rate, and learning rate. The memory buffer is also created to store experience tuples, which consist of the current state, the action taken, the reward received, the next state, and a flag indicating if the episode is over.
- The **_build_model** method is responsible for creating the CNN model that will be used to predict the optimal action given a state. However, the original implementation was commented out and replaced by loading a pre-existing model using Keras. This approach can be useful when training a model is time-consuming, but it also limits the ability to modify and fine-tune the architecture for the specific problem.
- The **act method** is used to select an action based on an exploration-exploitation trade-off. If a random number is less than the exploration rate, a random action is selected. Otherwise, the CNN model predicts the Q-values for each action, and the one with the highest value is selected.
- The **replay method** is responsible for updating the Q-values for the states visited during an episode using a mini-batch of experience tuples. First, the states, actions, rewards, and flags are extracted from

the memory buffer. Then, the target Q-values are computed using the Bellman equation, and the CNN model is updated to minimize the mean squared error between the predicted and target Q-values.

- The **train method** is used to run multiple episodes of the maze environment, during which the agent interacts with the environment, updates its memory buffer, and trains the CNN model using the replay method. After each episode, the exploration rate is reduced using a decay factor to shift the agent towards exploitation.
- Overall, this implementation of the MazeAgent class using a CNN model is a **failed approach** since the model did not achieve the desired performance in navigating the maze environment. The lack of knowledge about the maze environment and the reinforcement learning algorithm used might have contributed to this result. Additionally, the pre-existing model used in the implementation limits the ability to modify and fine-tune the architecture for the specific problem, which could be another reason for the failure.

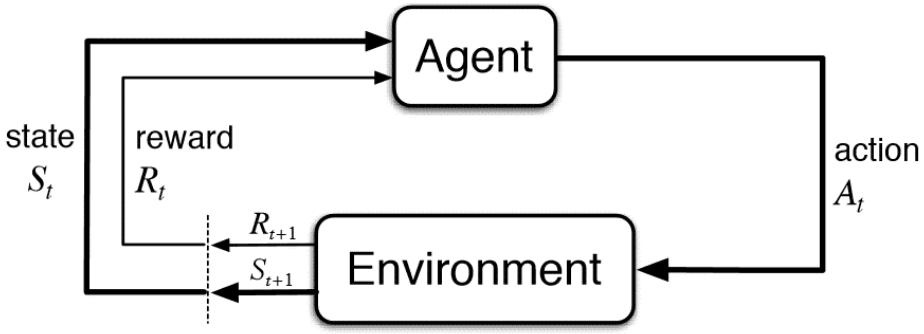
5. Final Model

```
class agent:
    def __init__(self,env):
        self.q_table = np.zeros((env.observation_space.shape[0], env.observation_space.shape[1], len(env.action_space)))
        alpha = 0.1  #learning rat
        gamma = 0.99  #discount factor
        epsilon = 1.0  #for greedy policy
        max_epsilon = 1.0
        min_epsilon = 0.01
        decay = 0.001
        self.env=env
        num_episodes = 1000
        for i in range(num_episodes):
            state = env.reset()
            done = False
            for j in range(200):
                if np.random.uniform() < epsilon:  #exploring the env
                    action = random.choice(env.action_space)
                else:
                    action = np.argmax(self.q_table[env.agent_pos[0], env.agent_pos[1], :])    #exploitation
                b,c =env.agent_pos[0], env.agent_pos[1]
                next_state, reward, done, info = env.step(action)
                if done:
                    break

                # Updating the q table
                old_value = self.q_table[b , c, action]
                next_max = np.max(self.q_table[env.agent_pos[0], env.agent_pos[1], :])
                new_value = (1 - alpha) * old_value + alpha * (reward + gamma * next_max)
                self.q_table[b , c, action] = new_value
                state = next_state  # updating the new_state

            # defining epsilon decay
            epsilon = min_epsilon + (max_epsilon - min_epsilon) * np.exp(-decay *i)
```

```python
# Use the final q table to solve the maze optimally
def solve_maze(self):
        state = env.reset()
        total_reward=0
        done = False
        i=0
        self.plots= []
        while not done:
            prev_pos = self.env.agent_pos
            self.plots.append(plt.gcf())  # collecting the plots of agent position to make a video of it in future
            self.env.render()  # render the current maze and agent position
            action = np.argmax(self.q_table[self.env.agent_pos[0], self.env.agent_pos[1], :]) # choosing the optimal action from the q table for a state
            state, reward, done, info = self.env.step(action) # taking the action
            total_reward+=reward  # cummulating the reward
            mem.append(self.env.agent_pos)
            if max(mem.count(item) for item in mem)==5: # if the agent moves in a repetetive pattern then break the loop and print that maze couldn't be solved
                break
            if self.env.agent_pos == prev_pos: # if agent is stuck inside the maze and no further move is possible then say that maze is not solvable
                i+=1
                if i==10:
                    break
            time.sleep(0.5) #time delay between rendering the plots
            display.clear_output(wait=True) # clearing the display

        if done:
            self.plots.append(plt.gcf())
            self.env.render()
            print(f"Maze Solved\nTotal Reward: {total_reward}")
        else:
            print("The Maze is not solvable")
```

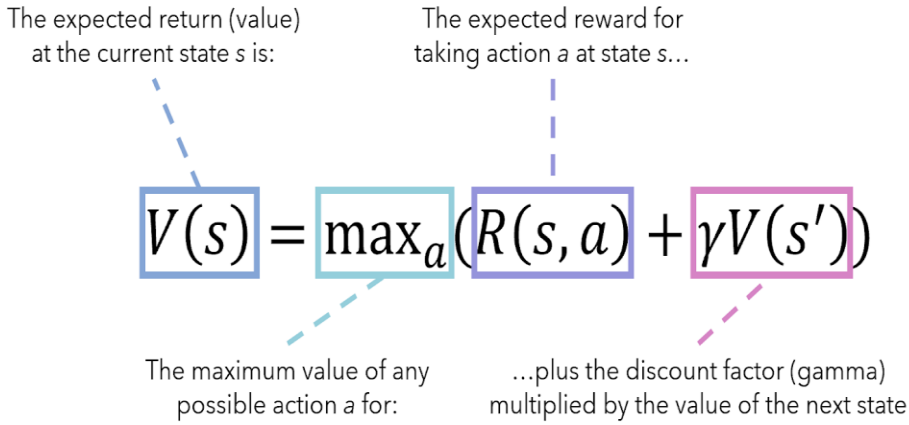
- The agent class presented in the code initializes a Q-table of zeros, which is used to store the expected reward for each action in each state. The agent then sets various hyperparameters such as the learning rate (alpha), discount factor (gamma), and exploration rate (epsilon) that will be used throughout the learning process.
- The agent then iteratively performs a series of episodes in the environment, with each episode consisting of multiple time steps where the agent chooses an action, receives a reward, and updates the Q-table accordingly. During each step, the agent chooses an action using an epsilon-greedy policy, which balances exploration and exploitation. Specifically, the agent selects a random action with probability epsilon, and chooses the action with the highest expected reward from the Q-table with probability 1-epsilon.
- The Q-table is updated at each time step by using the Bellman equation to estimate the expected reward for the current state-action pair. The new Q-value is computed as a weighted average of the current Q-value and the estimated value of the next state-action pair, where the weight is determined by the learning rate and discount factor.
- After the Q-learning process is complete, the agent can use the final Q-table to solve the maze optimally. The agent then runs a function **`solve_maze`** that takes the optimal policy learned during training and uses it to solve the maze. During this process, the agent chooses the optimal action at each time step, accumulates the reward, and updates the plot to show its movement through the maze. If the agent becomes stuck in a repetitive pattern, or if it is not able to reach the goal, the algorithm will stop and output an appropriate message.

5.1 Q-Learning



- Q-learning is a type of machine learning algorithm that is used to teach an agent to make decisions in an environment. In simple terms, Q-learning involves an agent making a series of decisions based on the rewards it receives for each action it takes.
- The agent learns by exploring the environment and trying different actions, and then updating its "Q-table" based on the rewards it receives. The Q-table is a table that stores the expected reward for each possible action in each possible state.
- Over time, as the agent continues to take actions and receive rewards, it becomes more skilled at making decisions that lead to the highest rewards. This is known as "learning by trial and error".
- One of the benefits of Q-learning is that it does not require the agent to have prior knowledge of the environment or any specific rules. The agent learns on its own by exploring the environment and updating its Q-table based on the rewards it receives. This makes Q-learning a popular choice for reinforcement learning tasks in complex and dynamic environments.

5.2 Bellman's Equation



- Bellman equation is a fundamental concept in reinforcement learning, and it is widely used to update the value function of a policy based on the rewards received and the expected value of the next state. The Bellman equation is named after Richard Bellman, who was one of the pioneers of dynamic programming.

- The Bellman equation is a recursive equation that calculates the expected value of the current state based on the expected value of the next state and the immediate reward received. In other words, it expresses the relationship between the current state's value and the value of its successor states.
- In reinforcement learning, the value of a state is defined as the expected cumulative reward received from that state onwards under a particular policy. The Bellman equation can be written in two forms, known as the Bellman expectation equation and the Bellman optimality equation.
- The Bellman expectation equation for the value function is:

  $$V(s) = E [r + \gamma V(s') \mid s, a],$$

  where V(s) is the value of the current state, r is the immediate reward received, γ is the discount factor that represents the importance of future rewards, and s' is the next state that the agent will transition to after acting a.

- The Bellman optimality equation for the Q-function is:

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

  where Q(s,a) is the value of acting a in state s, r is the immediate reward received, γ is the discount factor that represents the importance of future rewards, s' is the next state that the agent will transition to after acting a, and a' is the optimal action in the next state.

- The Bellman equation is essential in reinforcement learning because it provides a way to update the value function based on the observed rewards and expected future rewards. The value function can be updated using a process called value iteration or policy iteration, which involves repeatedly applying the Bellman equation to update the value function until it converges to the optimal value function.

5.3 Greedy Policy

The Epsilon-greedy policy and Boltzmann exploration policy are two of the most widely used exploration strategies in reinforcement learning algorithms.

- The Epsilon-greedy policy is a commonly used strategy in reinforcement learning algorithms that balance the exploration and exploitation in an environment. It is an approach where the agent chooses between a random action and the current best action. This policy consists of two parameters: epsilon (ε) and (1 - ε). Epsilon defines the probability of choosing a random action while (1 - ε) is the probability of selecting the best action. The Epsilon-greedy policy is a simple and efficient way of ensuring that the agent explores the environment while also exploiting the current knowledge.
  We have used epsilon greedy for our final model.

- Boltzmann exploration policy, on the other hand, is another approach to exploration that uses a SoftMax function to select actions. This policy encourages the agent to try all available actions in the environment, and the probability of each action is determined by a temperature parameter. The temperature parameter controls the level of exploration of the agent, and as the temperature

decreases, the policy becomes greedier. In Boltzmann exploration policy, each action is assigned a probability, and the action with the highest probability is chosen.

5.4 Epsilon Decay

While it may seem intuitive to simply decay epsilon by multiplying it with gamma at each episode, this approach may not result in optimal performance. This is because the effect of decay would become smaller and smaller as the number of episodes increases, potentially leading to premature convergence and suboptimal solutions.

Instead, the chosen approach of exponential decay (epsilon = min_epsilon + (max_epsilon - min_epsilon) * np.exp(-decay *i)) ensures that the decay factor remains significant and non-negligible even after many episodes. This allows for a more gradual and controlled exploration-exploitation tradeoff, leading to better convergence to optimal solutions. Additionally, the use of a decay parameter that can be adjusted allows for further fine-tuning of the learning process.

6.Deployement

Deployment is a crucial step in machine learning projects, as it involves making the model available for use in the real world. Deploying a model involves many steps such as integrating the model into a software application, testing it, and deploying it on a server.

We have deployed the model using “Streamlit” which can take the input as text using 0 and 1 and displays each step taken by agent to solve the maze optimally, we have added the image intake as well but due to some error in “Streamlit” server it can’t solve the maze in image but image maze can be solved using the code in Jupyter notebook or similar platform. Example text input -

0, 0, 0, 0, 1, 0, 0, 1, 0, 0

0, 1, 0, 1, 1, 0, 0, 1, 1, 0

0, 1, 0, 0, 0, 0, 0, 0, 1, 1

0, 1, 1, 1, 0, 1, 1, 0, 1, 0

0, 1, 0, 0, 1, 0, 0, 0, 0, 0

0, 0, 1, 0, 1, 0, 1, 1, 1, 1

0, 1, 0, 1, 0, 0, 0, 0, 0, 0

0, 1, 0, 0, 0, 1, 1, 0, 1, 0

0, 1, 1, 0, 1, 0, 1, 0, 1, 1

0, 0, 0, 0, 1, 0, 0, 0, 0, 0

Here is the link to the site feel free to try it -

https://damnkuldeep-mazesolvingusingrl-mazesolver-gajetg.streamlit.app/

7.CONCLUSION

We presented an approach to solving a maze using reinforcement learning. Specifically, we used a Q-learning algorithm to train an agent to navigate through a maze and reach the goal state while avoiding obstacles.

We began by discussing the problem of maze-solving and introduced the concept of reinforcement learning as a potential solution. We then described our approach to implementing the Q-learning algorithm and provided details on the environment and reward functions used in our implementation.

Overall, our results demonstrate that reinforcement learning can be an effective approach to solving maze navigation problems. Future work could focus on improving the performance of the algorithm by exploring different reward functions or using more sophisticated deep reinforcement learning techniques.

You can get the code from the GitHub Repository: -

https://github.com/DamnKuldeep/Maze-Solving/tree/main

8. ACKNOWLEDGEMENT

We would like to express our sincere gratitude to our supervisors, Shreshth Mehrotra and Avishkar, for providing us with guidance and support throughout this project. Their valuable feedback and suggestions were instrumental in shaping the direction of this study.

We would also like to thank “Artificial Intelligence & Electronic Society (ArIES)” for providing us with the necessary resources and facilities to carry out this project.

Finally, we would like to express our appreciation to all the individuals who participated in this project, whose contributions were crucial in helping us obtain the results presented in this paper.