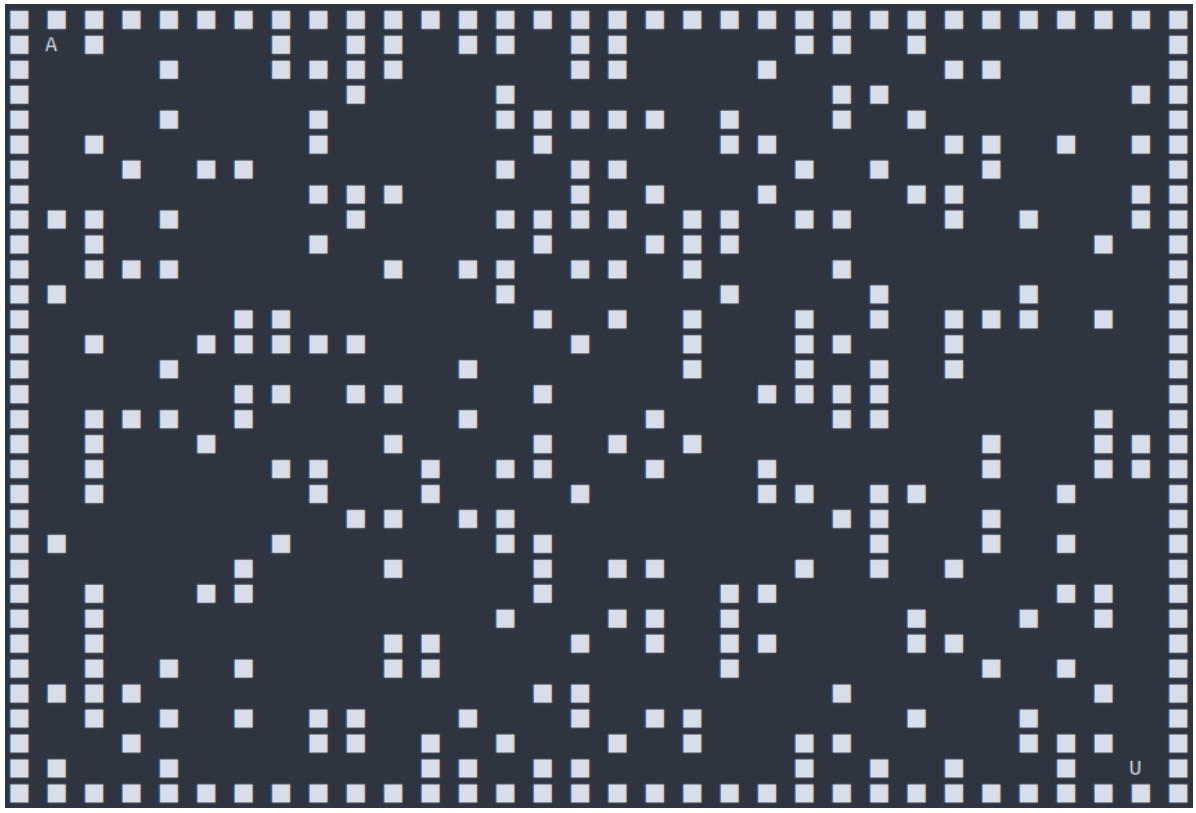
Intelligent Models

Agent in Labyrinth Walls (Q-Learning)

Bocchini Fabio - matricola: 340600 - A.A. 2021/2022



GitHub repository: https://github.com/FabioBocchini/agent_in_labyrinth_walls.git

Obiettivo

Creazione di un environment per Reinforcement Learning rappresentante un labirinto, e di un agente che tramite un algoritmo di Q-Learning impara a muoversi attraverso il labirinto per trovare l'uscita.

Environment

La creazione dell'environment è stata effettuata tramite la libreria Open AI Gym che permette di estendere la classe `Env` per crearne di propri.

La rappresentazione del labirinto sarà una matrice dove ogni elemento può essere 0 = vuoto o 1 = muro

Azioni, Osservazioni e Stati

Le azioni che un agente può eseguire sono 4, ovvero muoversi nelle 4 direzioni cardinali

```
actions = {
    "U": 0, # Up
    "D": 1, # Down
    "L": 2, # Left
    "R": 3  # Right
}
```

L'osservazione di un agente è rappresentata dal vicinato di Moore, ovvero conosce tutto quello che si trova a distanza 1 da lui, ortogonalmente e diagonalmente. Viene rappresentato come una lista con 8 elementi

```
"""
0 1 2
3 A 6
7 8 9
"""
```

Ogni elemento della lista sarà

- 0 se la posizione è vuota
- 1 se nella posizione c'è un muro

L'osservazione può quindi essere vista come un numero binario con 8 cifre

`[1,1,1,1,0,1,0,0] = 11110100 = 244`

Le osservazioni quindi sono rappresentabili da una lista di valori da 0 a 255

```
self.state_space = [i for i in range(256)]
```

Generazione di un labirinto

```
    def generate_labyrinth(self):
        labyrinth_h = self.labyrinth_size["h"]
        labyrinth_w = self.labyrinth_size["w"]
        walls_to_insert = int(((labyrinth_h-2)*(labyrinth_w-
2)self.labyrinth_size["p"])/100)

        labyrinth = np.zeros((labyrinth_h, labyrinth_w), dtype=int)

        for y in range(labyrinth_h):
            for x in range(labyrinth_w):
                if (y == 0) or (y == labyrinth_h - 1) or (x == 0) or (x ==
labyrinth_w - 1):
                    labyrinth[y][x] = 1

        while walls_to_insert:
            x = random.randint(1, labyrinth_w - 2)
            y = random.randint(1, labyrinth_h - 2)
            if labyrinth[y][x] == 0:
                labyrinth[y][x] = 1
                walls_to_insert = walls_to_insert - 1
        return labyrinth
```

Dati in input altezza, larghezza e percentuale di muri presenti viene creata una matrice h x w. Ad entrambe le dimensioni è stato sommato 2 per inserire un bordo di mura al labirinto.

Le mura all'interno del labirinto vengono aggiunte tramite un ciclo che controlla semplicemente che vengano aggiunte mura fino ad arrivare alla percentuale scelta.

Non è un algoritmo ottimo, potrebbe rallentare se la percentuale è troppo alta e potrebbe creare labirinti senza cammini liberi tra entrata ed uscita. Si consiglia quindi di scegliere percentuali relativamente basse e di controllare il labirinto generato prima di addestrare l'agente.

Dopo la creazione del labirinto, viene salvato su un file che poi potrà essere utilizzato per eseguire il programma senza crearne uno nuovo.

Step

La funzione `step(action)` data un'azione, esegue l'azione nell'environment

Per prima cosa, se l'agente vuole muoversi in una posizione libera, lo muove e calcola la prossima osservazione sotto forma di intero

```
state = [
            self.labyrinth[agent_y - 1][agent_x - 1],  # 0
            self.labyrinth[agent_y - 1][agent_x],  # 1
            self.labyrinth[agent_y - 1][agent_x + 1],  # 2

            self.labyrinth[agent_y][agent_x - 1],  # 3
            # A, the agent is here
            self.labyrinth[agent_y][agent_x + 1],  # 4

            self.labyrinth[agent_y + 1][agent_x - 1],  # 5
            self.labyrinth[agent_y + 1][agent_x],  # 6
            self.labyrinth[agent_y + 1][agent_x + 1]  # 7
        ]
```

Oltre all'osservazione ritorna il reward generato dall'azione:

- -1 se si muove in una casella libera
- -5 se sbatte contro un muro
- +10 se si muove sulla casella dell'uscita

Per ultima cosa controlla che l'esecuzione sia finita, ritornando `done=True` se è stata raggiunta l'uscita o se è stato eseguito il numero massimo di azioni

```
    def step(self, action: int):
        bumped_wall, observation = self.next_observation(action)
        reward = -1
        done = False

        if bumped_wall:
            reward = -5

        if self.agent_position["x"] == self.labyrinth_size["w"] - 2 \
                and self.agent_position["y"] == self.labyrinth_size["h"] - 2:
            done = True
            reward = 10

        self.max_actions = self.max_actions - 1
        if self.max_actions == 0:
            done = True

        return observation, reward, done, {}
```

Reset

La funzione reset riporta l'environment nella posizione di partenza, spostando l'agente nella posizione (1,1) e ritorna l'osservazione in questo stato

```
    def reset(self):
        self.agent_position = {
            "x": 1,
            "y": 1
        }
        _, observation = self.next_observation()
        return observation
```

Q-Learning

Il training dell'agente viene effettuato tramite un algoritmo di Q-learning, che sfrutta una matrice `Q(stato, azione) per memorizzare quali sono le migliori azioni da eseguire in un certo stato tramite la formula:

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

dove

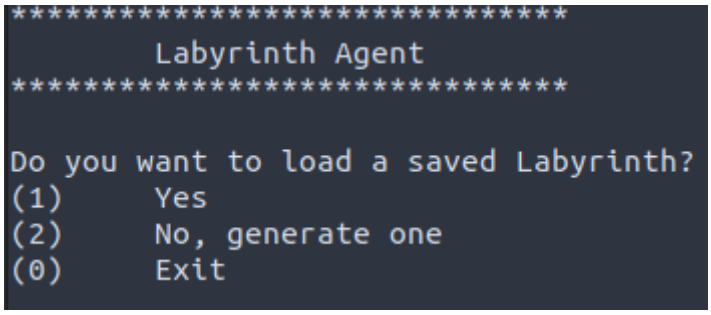
- **α** è il learning rate, ovvero indica quanto l'algoritmo deve scegliere l'esplorazione (exploration) di nuove azioni invece che sfruttare (exploitation) quelle che già conosce.
- **γ** è il discount factor, che abbassa o alza il valore di un reward in base a **t** (numero di epoche passate)
- **maxQ** è una funzione che ritorna l'azione con stima di qualità maggiore per un determinato stato

```
def max_action(self, q, state, actions):
    """
    Returns the actions with highest value in the Q matrix for a given
state
    """
    values = np.array([q[state, a] for a in actions])
    action = np.argmax(values)
    return actions[action]

def training(self, epochs=50000, steps=200, alpha=0.1, gamma=1.0, eps=1.0,
plot=True):  [...]
    for i in range(epochs):
        if i % int(epochs / 10) == 0:
            print('epochs passed: ', i)

        done = False
        ep_rewards = 0
        num_actions = 0
        observation = self.env.reset()
        while not done and num_actions <= steps:
            rand = np.random.random()
            action = \
                self.max_action(q, observation, self.env.possible_actions)
\
                if rand < (1 - eps) \
                else self.env.action_space_sample()
            observation_next, reward, done, info = self.env.step(action)
            num_actions += 1
```

```
            ep_rewards += reward
            action_next = self.max_action(q, observation_next,
self.env.possible_actions)
            q[observation, action] = q[observation, action] + alpha * (
               reward + gamma * q[observation_next, action_next] -
q[observation, action]
            )
            observation = observation_next
        if eps - 2 / epochs > 0:
            eps -= 2 / epochs
        else:
            eps = 0
        total_rewards[i] = ep_rewards
[...]
```

Il programma

```
python3 main.py
```



Avviato il programma possiamo scegliere se caricare un labirinto generato in precedenza (deve trovarsi nella cartella root del programma con il nome `labyrinth`) o generarne uno fornendo le varie dimensioni

Dopo aver caricato o generato il labirinto possiamo scegliere tra:

- **Training**, allenare l'agente tramite l'algoritmo di Q-learning, facendo questo la matrice Q(State, Action) viene salvata su file, così da poter essere salvata e riutilizzata poi. Alla fine

dell'esecuzione viene mostrato il grafo dei reward accumulati durante le varie epoche.
- **Execute Automatically** Mostra come l'agente ha imparato a muoversi nel labirinto utilizzando una matrice Q salvata in precedenza. Ad ogni passo mostra quale azione è stata effettuata, il reward assegnato in quello step, l'osservazione fatta e il reward accumulato dall'inizio dell'esecuzione
- **Execute step by step** come Execute Automatically ma viene richiesto un input ad ogni step per continuare

Risultati

Ho effettuato vari test su labirinti 10x10 e 30x30, sempre con il 20% di celle occupate da mura.

Per ogni test si possono trovare nella cartella `./results` Il plot generato e i file `labyrinth` e `Qmatrix`

	Epochs	Steps	Grid H	Grid W	Wall %	Max actions	Training Time	Solution
1	25000	400	10	10	20	100	66.53 s	Yes
2	25000	400	10	10	20	400	74.53 s	Yes
3	25000	400	30	30	20	400	280.94 s	No
4	50000	400	10	10	20	100	100.10 s	Yes
5	50000	400	10	10	20	400	132.62 s	Yes
6	50000	400	30	30	20	100	477.35 s	No
7	50000	400	30	30	20	400	504.17 s	Yes

Possiamo notare che le dimensioni del labirinto influiscono molto sul tempo d'esecuzione del training. Come possiamo vedere dai risultati 4 e 6 il tempo d'esecuzione sale linearmente rispetto ad una dimensione del labirinto (perché sono uguali).

Anche il numero di epoche influenza linearmente il tempo d'esecuzione, raddoppiando le eopche il tempo infatti raddoppia.

Il numero di step e il numero massimo di azioni eseguibili possono essere considerati come la stessa cosa, dato che influenzano allo stesso modo l'algoritmo. Settando Max Actions a 100, qualsiasi valore di Steps superiore a 100 non sarà mai raggiunto dall'algoritmo per passare alla prossima epoca.

Il numero di azioni massimo eseguibile influenza il tempo d'esecuzione ma in modo minore rispetto alle altre dimensioni.

Possiamo anche notare che quando una soluzione non viene trovata, l'agente si muove in una posizione casuale del labirinto e inizia ad oscillare tra due posizioni, ma non prova mai a muoversi contro un muro.

Ecco alcuni plot dei test effettuati

- **1:** 
- **3:** 
- **7:**