.UBA fiuba
FACULTAD DE INGENIERÍA

APRENDIZAJE POR REFUERZO

Desafío Final

ABSTRACT
En este trabajo se aborda la resolución de un problema simple mediante Aprendizaje por Refuerzo, utilizando el algoritmo Q-Learning.

Charaf, Christopher – Villanueva, Azul
CEIA

1. INTRODUCCIÓN

Se diseña un entorno cuadrado 5x5 en el cual un agente inicia en la posición (0,0) y debe llegar a la posición (4,4). El entorno contiene casillas seguras (F), pozos (H) y una meta (G).

El agente puede moverse en cuatro direcciones y recibe +1 al alcanzar la meta, -1 si cae en un hueco y -0.01 en otros casos para incentivar un camino óptimo.

El episodio termina si cae en un pozo o si llega al objetivo. La política se aprende mediante Q-Learning.

```python
class FrozenLakeCustomEnv(gym.Env):
    def __init__(self):
        super().__init__()
        self.size = 5
        self.action_space = spaces.Discrete(4)  # 0: izquierda, 1: abajo, 2: derecha, 3: arriba
        self.observation_space = spaces.Discrete(self.size * self.size)

        self.map = np.array([
            ['S', 'F', 'F', 'F', 'F'],
            ['F', 'F', 'H', 'F', 'F'],
            ['F', 'H', 'F', 'H', 'F'],
            ['F', 'F', 'F', 'F', 'F'],
            ['H', 'F', 'H', 'F', 'G']
        ])

        self.start_pos = (0, 0)
        self.goal_pos = (4, 4)
        self.hole_pos = {(1, 2), (2, 1), (2, 3), (4, 0), (4, 2)}
        self.state = self.start_pos

    def _to_state_index(self, pos):
        return pos[0] * self.size + pos[1]

    def _from_state_index(self, index):
        return (index // self.size, index % self.size)

    def reset(self, seed=None, options=None):
        super().reset(seed=seed)
        self.state = self.start_pos
        return self._to_state_index(self.state), {}

    def step(self, action):
        row, col = self.state
        new_row, new_col = row, col
        if action == 0: new_col = max(col - 1, 0)
        elif action == 1: new_row = min(row + 1, self.size - 1)
        elif action == 2: new_col = min(col + 1, self.size - 1)
        elif action == 3: new_row = max(row - 1, 0)

        self.state = (new_row, new_col)
        if self.state in self.hole_pos:
            return self._to_state_index(self.state), -1.0, True, False, {}
        elif self.state == self.goal_pos:
            return self._to_state_index(self.state), 1.0, True, False, {}
        else:
            return self._to_state_index(self.state), -0.01, False, False, {}

    def render(self, mode='human', delay=0.3):
        grid = self.map.copy()
        r, c = self.state
        grid[r, c] = 'A'
        clear_output(wait=True)
        print("Estado actual:")
        print("\n".join([" ".join(row) for row in grid]))
        print(f"Posición: ({r}, {c})")
        sleep(delay)
```

✓ 0.0s

2. IMPLEMENTACIÓN DEL ENTORNO Y ALGORITMO

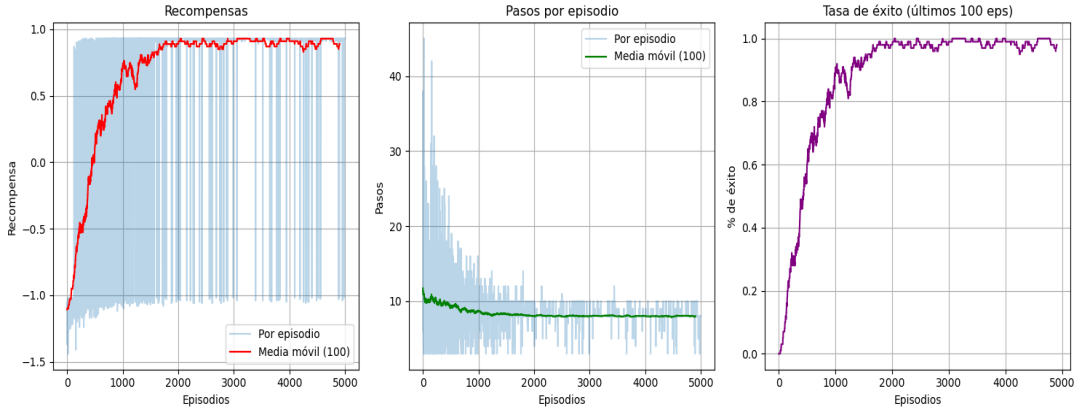
Se desarrolló el entorno herendando de gymnasium.Env, utilizando espacios de observación y acción discretos. La tabla Q se inicializa en ceros y se actualiza mediante la ecuación clásica del algoritmo Q-Learning con política ε-greedy. Los parámetros utilizados fueron: tasa de aprendizaje 0.1, factor de descuento 0.99, ε inicial de 1.0 con decay hasta 0.01.

```python
env = FrozenLakeCustomEnv()
n_states = env.observation_space.n
n_actions = env.action_space.n
q_table = np.zeros((n_states, n_actions))

alpha = 0.1
gamma = 0.99
epsilon = 1.0
epsilon_min = 0.01
epsilon_decay = 0.998
n_episodes = 5000

rewards = []
steps_list = []
successes = []

for episode in range(n_episodes):
    state, _ = env.reset()
    done = False
    total_reward = 0
    steps = 0

    while not done:
        if np.random.random() < epsilon:
            action = env.action_space.sample()
        else:
            action = np.argmax(q_table[state])

        next_state, reward, done, _, _ = env.step(action)
        best_next = np.max(q_table[next_state])
        q_table[state, action] += alpha * (reward + gamma * best_next - q_table[state, action])
        state = next_state
        total_reward += reward
        steps += 1

    rewards.append(total_reward)
    steps_list.append(steps)
    successes.append(1 if reward == 1.0 else 0)
    epsilon = max(epsilon_min, epsilon * epsilon_decay)
    if episode % 500 == 0:
        print(f"Episodio {episode}: Recompensa promedio (últimos 100) = {np.mean(rewards[-100:]):.2f}")
```

✓ 0.3s

3. RESULTADOS Y GRÁFICO DE CONVERGENCIA

Se entrenó el agente durante 5000 episodios y se observó convergencia en la política aprendida, reflejada en un aumento progresivo de la recompensa media. A continuación, se presenta el gráfico de convergencia, pasos por episodio y tasa de éxito por cada 100 episodios, hasta 5000.



4. POLÍTICA APRENDIDA

Una vez entrenado el agente, se imprime la política aprendida representada con flechas en la grilla, indicando la mejor acción en cada estado según la tabla Q. Las casillas con pozos se indican como 'H', la meta como 'G', y el resto con las direcciones ← ↓ → ↑.

```
Política óptima aprendida:
↓ ← ← ↓ ↓
↓ ← H → ↓
↓ H ↓ H ↓
→ → → → ↓
H ↑ H → G

Valores Q máximos:
0.86 0.85 0.25 0.08 0.11
0.88 0.86 0.00 0.18 0.63
0.90 0.00 0.93 0.00 0.98
0.92 0.94 0.96 0.98 1.00
0.00 0.92 0.00 1.00 0.00
```

5. DESAFÍOS ENCONTRADOS

- Inicialmente, el entorno definido contenía demasiados pozos y caminos poco accesibles, lo que impedía que el agente aprendiera una política útil. Esto fue resuelto rediseñando el mapa para que mantuviera obstáculos, pero permitiera rutas viables hacia la meta, en este caso hay una potencialidad, dados los parámetros correctos de entrenamiento e hiperparámetros, de aumentar la complejidad del entorno.
- El gráfico de convergencia se mostraba plano porque el agente no lograba alcanzar la meta en ningún episodio. Este comportamiento fue resuelto tras los cambios en el entorno y una mejora en la configuración de recompensas.
- La recompensa inicial del entorno era de tipo binario (1 si ganaba, 0 en todo otro caso). Se introdujeron penalizaciones pequeñas por paso y penalizaciones más fuertes por caer en pozos, mejorando el aprendizaje.
- Hubo varias oportunidades donde se ajustaron los hiperparámetros para acelerar la convergencia.

6. CONCLUSIONES

Para concluir, el experimento demostró la efectividad del algoritmo Q-Learning en un entorno personalizado. El agente fue capaz de aprender una política óptima y generalizable sin el uso de librerías obsoletas.

El diseño modular permite extender el entorno y probar otros algoritmos como SARSA o Monte Carlo fácilmente.

6. REPOSITORIO DEL CÓDIGO

Github