Frozen Lake

Diego Fernández

17 de outubro de 2023

0.1 Introducción

Frozen Lake constituye un ambiente elemental formado por cuadros a los que la inteligencia artificial debe transitar desde una casilla de partida hasta una meta específica. Dichas casillas pueden consistir en un lago congelado seguro o en una trampa inexorable. El agente, o IA, dispone de cuatro opciones de acción, a saber: desplazarse hacia la izquierda, hacia abajo, hacia la derecha o hacia arriba. El agente debe adquirir la habilidad de evitar las trampas a fin de alcanzar el objetivo en el menor número de movimientos posible. Para resolver este ejercicio implementaremos un algoritmo de Q-learning y el algoritmo Epsilon-Greedy.

0.2 Q-Table

Iniciamos una tabla Q con cero en todos los valores. Esta tabla debe tener las msmas dimensiones del entorno de nuestro Frozen Lake.

0.3 Q-learning

Vamos a emplear diversos hiperparámetros que regulan el proceso de entrenamiento, tales como el número total de episodios (episodios), la tasa de aprendizaje (alfa), el factor de descuento (gamma), la tasa inicial de exploración (epsilon) y la velocidad de disminución de epsilon (epsilon-decay). En el proceso de entrenamiento, se llevará a cabo un bucle de longitud igual al número de episodios, en el cual, en cada episodio, el agente inicia desde un estado inicial y realiza acciones con el propósito de navegar por el entorno hasta llegar a un estado final, ya sea con éxito o fracaso, o hasta alcanzar su objetivo.

El agente decide qué acciones tomar siguiendo la estrategia Epsilon-Greedy, que implica que en ocasiones ejecutará una acción de manera aleatoria con una probabilidad igual a epsilon, mientras que en los demás casos, optará por la acción con el valor Q más elevado en el estado actual.

Una vez que se ha elegido la acción, esta se lleva a cabo y, a continuación, se actualizan los valores Q del agente mediante la aplicación de la fórmula de actualización de Q-learning.

0.4 Código

Iniciaremos realizando todas las importaciones necesarias e inicando nuestro entorno Frozen Lake

```
!pip install -q gym matplotlib
import gym
import random
import numpy as np
from IPython.display import clear_output
import time
import matplotlib.pyplot as plt
```

```
environment = gym.make("FrozenLake-v1", is_slippery=False)
environment.reset()
environment.render()
```

Ahora configuramos el entorno de manera aleatoria con environment.action space.sample()

```
# 1. Randomly choose an action using action_space.sample()
action = environment.action_space.sample()

# 2. Implement this action and move the agent in the desired direction
new_state, reward, done, info = environment.step(action)

# Display the results (reward and map)
environment.render()
print(f'Reward = {reward}')
```

```
Reward = 0.0
```

Tras esto, declararemos e inicializaremos nuestros hiper-parámetros y nuestra q-table (a 0 esta última).

```
qtable = np.zeros((environment.observation_space.n, environment.action_space.n))

# Hyperparameters
episodes = 1000        # Total number of episodes
alpha = 0.5            # Learning rate
gamma = 0.9            # Discount factor
epsilon = 1.0          # Amount of randomness in the action selection
epsilon_decay = 0.001  # Fixed amount to decrease

# List of outcomes to plot
outcomes = []

print('Q-table before training:')
print(qtable)
```

Después de esto, procederemos a iniciar el proceso de entrenamiento dentro de un ciclo, en el cual en cada iteración, el agente inicia desde un estado inicial y ejecuta acciones con el fin de desplazarse a través del entorno hasta alcanzar un estado terminal, siguiendo la estrategia Epsilon-Greedy. La acción seleccionada se ejecuta y, seguidamente, los valores Q del agente se actualizan mediante la aplicación de la fórmula de actualización de Q-learning.

Epsilon se modifica en cada episodio mediante la fórmula epsilon = max(epsilon - epsilon decay, 0), lo que conduce a una reducción progresiva de la exploración a medida que el entrenamiento avanza.

```python
# Training
for _ in range(episodes):
    state = environment.reset()
    done = False

    # By default, we consider our outcome to be a failure
    outcomes.append("Failure")

    # Until the agent gets stuck in a hole or reaches the goal, keep training it
    while not done:
        # Generate a random number between 0 and 1
        rnd = np.random.random()

        # If random number < epsilon, take a random action
        if rnd < epsilon:
          action = environment.action_space.sample()
        # Else, take the action with the highest value in the current state
        else:
          action = np.argmax(qtable[state])

        # Implement this action and move the agent in the desired direction
        new_state, reward, done, info = environment.step(action)

        # Update Q(s,a)
        qtable[state, action] = qtable[state, action] + \
                                alpha * (reward + gamma * np.max(qtable[new_state]) - qtable[state, action])

        # Update our current state
        state = new_state

        # If we have a reward, it means that our outcome is a success
        if reward:
          outcomes[-1] = "Success"

    # Update epsilon
    epsilon = max(epsilon - epsilon_decay, 0)
```

Tras el entrenamiento, podemos ver como los valores de la q-table se han ajustado al modelo:

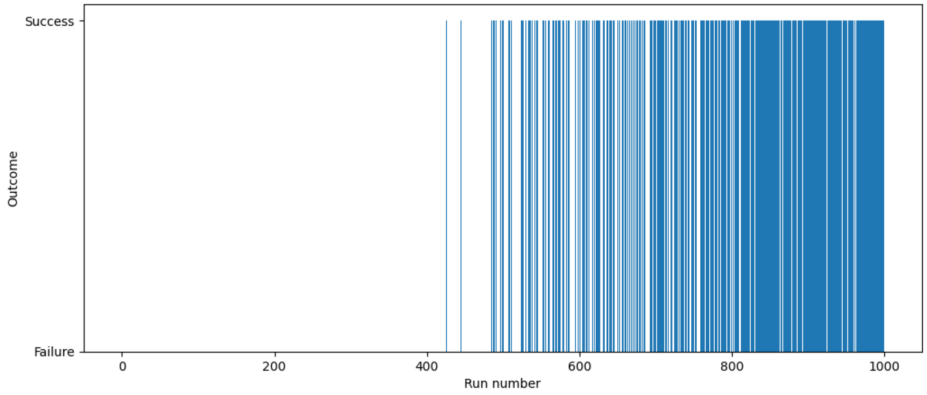
```
Q-table before training:
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]

==========================================
Q-table after training:
[[0.531441   0.59049    0.4782969  0.531441  ]
 [0.531441   0.         0.40213119 0.44814621]
 [0.47427227 0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.59049    0.6561     0.         0.531441  ]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.6561     0.         0.729      0.59049   ]
 [0.65609999 0.81       0.80999999 0.        ]
 [0.67664651 0.9        0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.         0.80990112 0.9        0.72899467]
 [0.80999812 0.9        1.         0.80999996]
 [0.         0.         0.         0.        ]]
```

En la siguiente gráfica, también podemos ver la tasa de éxito con el paso de los "episodios"

```python
episodes = 100
nb_success = 0

# Evaluation
for _ in range(100):
    state = environment.reset()
    done = False

    # Until the agent gets stuck or reaches the goal, keep training it
    while not done:
        # Choose the action with the highest value in the current state
        action = np.argmax(qtable[state])

        # Implement this action and move the agent in the desired direction
        new_state, reward, done, info = environment.step(action)

        # Update our current state
        state = new_state

        # When we get a reward, it means we solved the game
        nb_success += reward

# Let's check our success rate!
print (f"Success rate = {nb_success/episodes*100}%")
```

```
Success rate = 100.0%
```

Una vez que el agente ha sido entrenado, procedemos a evaluar su desempeño y podemos observar que alcanza una tasa del 100 por 100 de éxito.



0.5 Conclusiones

En esta práctica, hemos adquirido un conocimiento valioso acerca del algoritmo Epsilon-Greedy, que representa una estrategia fundamental en el ámbito del aprendizaje por refuerzo. Este algoritmo desempeña un papel crucial al equilibrar la exploración y la explotación de manera controlada y adaptable.

Epsilon-Greedy se distingue por su capacidad de gestionar el dilema entre buscar nuevas opciones y aprovechar al máximo las decisiones ya conocidas. Esto se logra al permitir que el agente, con cierta probabilidad, elija una acción de manera completamente aleatoria en lugar de optar por la acción que se considera la mejor según su conocimiento actual. Esta característica fomenta la exploración continua del entorno, incluso después de que el agente haya aprendido acciones que parecen ser las más óptimas en determinados estados.

El resultado de esta estrategia es que el agente sigue explorando su entorno en busca de nuevas oportunidades, lo que puede ser fundamental para descubrir caminos alternativos o inesperados que podrían llevar a soluciones aún más beneficiosas. En esencia, Epsilon-Greedy proporciona una herramienta poderosa para asegurar que el agente mantenga un enfoque equilibrado entre la explotación de lo que ya sabe y la búsqueda de nuevas experiencias para mejorar su toma de decisiones.