

1. Lago Congelado (Frozen Lake)

El juego Frozen Lake consiste en llevar a un agente desde un punto inicial a un objetivo en un entorno de mosaicos que representan un lago con áreas seguras y trampas. El agente puede moverse hacia la izquierda, derecha, arriba o abajo, evitando caer en los agujeros mientras busca llegar al objetivo con la menor cantidad de movimientos.

<i>SFFF</i>	(S: punto de inicio, seguro)
<i>FHFFH</i>	F: superficie congelada, seguro)
<i>FFFFH</i>	H: agujero, atrapado para siempre)
<i>HFFG</i>	G: objetivo, seguro)

Los mosaicos del juego son marcados como superficies congeladas seguras (indicadas con una marca de verificación) o agujeros peligrosos (indicados con una cruz). A pesar de que existen múltiples formas de llegar al objetivo, el desafío es hacerlo en la menor cantidad de acciones posibles. Por ejemplo, aunque se puedan encontrar diferentes rutas para llegar al objetivo, algunas pueden dar vueltas innecesarias antes de llegar al objetivo, lo cual no cumple con el requisito de minimizar los movimientos.

Para empezar, se utiliza la biblioteca 'gym' para inicializar el entorno del juego. Hay dos versiones disponibles: una con hielo resbaladizo, donde las acciones pueden ser ignoradas ocasionalmente por el agente, y otra sin esta característica. Comenzamos con la versión sin hielo resbaladizo, ya que es más sencillo de entender y abordar.

2. Q-Table

En el entorno Frozen Lake, hay 16 estados distintos y 4 acciones posibles para cada uno: ir izquierda, abajo, derecha y arriba. Para tomar decisiones informadas en cada estado, asignamos valores de calidad a las acciones. Esto se logra mediante una tabla Q, que enumera los estados en filas y las acciones en columnas.

Cada celda de la tabla Q contiene un valor $Q(s, a)$, representando la calidad de la acción 'a' en el estado 's'. El valor más alto indica la mejor acción a tomar en un estado dado.

Estado	← IZQUIERDA	↓ ABAJO	→ DERECHA	↑ ARRIBA
0	$Q(0, \leftarrow)$	$Q(0, \downarrow)$	$Q(0, \rightarrow)$	$Q(0, \uparrow)$
1	$Q(1, \leftarrow)$	$Q(1, \downarrow)$	$Q(1, \rightarrow)$	$Q(1, \uparrow)$
2	$Q(2, \leftarrow)$	$Q(2, \downarrow)$	$Q(2, \rightarrow)$	$Q(2, \uparrow)$
...
14	$Q(14, \leftarrow)$	$Q(14, \downarrow)$	$Q(14, \rightarrow)$	$Q(14, \uparrow)$
15	$Q(15, \leftarrow)$	$Q(15, \downarrow)$	$Q(15, \rightarrow)$	$Q(15, \uparrow)$

Esta tabla inicialmente se llena con valores de cero, ya que aún no se han calculado los valores de calidad para cada

acción en cada estado. Es esencial para nuestro agente, ya que al estar en un estado particular, puede consultar esta tabla para tomar la acción con el valor de calidad más alto.

Vamos a crear nuestra tabla Q y llenarla con ceros ya que aún no tenemos idea del valor de cada acción en cada estado.

```
1  import gym
2  import random
3  import numpy as np
4
5  # Initialize the non-slippery Frozen Lake environment
6  environment = gym.make("FrozenLake-v1", is_slippery=False, render_mode="
    rgb_array")
7  environment.reset()
8  environment.render()
9
10 # Initialize Q-table with zeros
11 # Our table has the following dimensions:
12 # (rows x columns) = (states x actions) = (16 x 4)
13 qtable = np.zeros((16, 4))
14
15 # Alternatively, the gym library can also directly
16 # give us the number of states and actions using
17 # "env.observation_space.n" and "env.action_space.n"
18 nb_states = environment.observation_space.n # = 16
19 nb_actions = environment.action_space.n     # = 4
20 qtable = np.zeros((nb_states, nb_actions))
21
22 # Let's see how it looks
23 print('Q-table =')
24 print(qtable)
```

```
(env).reset()
Q-table =
[[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.]]
```

En este momento, el agente se encuentra en el estado inicial S . En esta posición, solo tiene dos opciones de movimiento disponibles: ir hacia la derecha (\rightarrow DERECHA) o hacia abajo (\downarrow ABAJO). Aunque el agente técnicamente puede realizar las acciones de moverse hacia arriba (\uparrow ARRIBA) o hacia la izquierda (\leftarrow IZQUIERDA), estas acciones no resultarán en un cambio de estado. En otras palabras, no hay restricciones impuestas sobre qué acciones puede realizar el agente; sin embargo, naturalmente comprenderá que algunas de estas acciones no tienen efecto en su posición actual y no cambiarán su estado.

La biblioteca gym implementa un metodo para elegir de manera random una acción, por lo que nos ahorramos el tener que usar la librería random, haciendo uso del siguiente comando:

```
1 environment.action_space.sample()
```

```
(env) PS C:\Users\PC\Desktop\UAX\Tercero\Optimizacion y Control\Repos\Toc14_Aprendizaje_Refuerzo> python refuerzo.py
2
(env) PS C:\Users\PC\Desktop\UAX\Tercero\Optimizacion y Control\Repos\Toc14_Aprendizaje_Refuerzo> python refuerzo.py
1
(env) PS C:\Users\PC\Desktop\UAX\Tercero\Optimizacion y Control\Repos\Toc14_Aprendizaje_Refuerzo> python refuerzo.py
2
(env) PS C:\Users\PC\Desktop\UAX\Tercero\Optimizacion y Control\Repos\Toc14_Aprendizaje_Refuerzo> python refuerzo.py
1
(env) PS C:\Users\PC\Desktop\UAX\Tercero\Optimizacion y Control\Repos\Toc14_Aprendizaje_Refuerzo> python refuerzo.py
1
(env) PS C:\Users\PC\Desktop\UAX\Tercero\Optimizacion y Control\Repos\Toc14_Aprendizaje_Refuerzo> python refuerzo.py
2
(env) PS C:\Users\PC\Desktop\UAX\Tercero\Optimizacion y Control\Repos\Toc14_Aprendizaje_Refuerzo>
```

Vemos que la salida varía entre 1 y 2, ¿Por qué puede ser esto? y, ¿Por qué es un número?

\leftarrow LEFT = 0
 \downarrow DOWN = 1
 \rightarrow RIGHT = 2
 \uparrow UP = 3

De esta manera, la biblioteca gym conecta números con las direcciones del agente en el juego. Vamos a intentar mover el agente para que haga un movimiento hacia la derecha:

```
1 environment.step(2)environment.render()
```

Y el resultado es:

Salida:

(Right)

SFFF

FHFH

FFFH

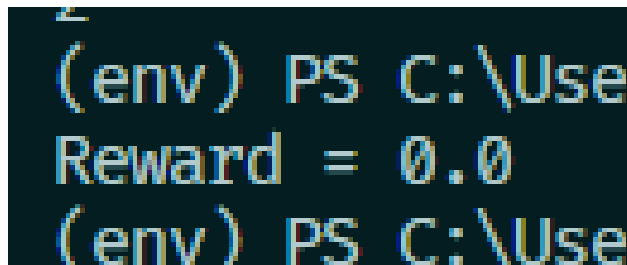
HFFG

Ahora que comprendemos la interacción con nuestro entorno en Gym, regresemos al algoritmo. En el aprendizaje por refuerzo, los agentes reciben recompensas del entorno cuando logran cumplir un objetivo específico. En el contexto de Frozen Lake, el agente solo recibe una recompensa cuando alcanza el estado G (consultar el código fuente). Esta recompensa está predefinida en el entorno y está fuera de nuestro control: proporciona un valor de 1 cuando el agente llega al estado G y un valor de 0 en caso contrario.

Cada vez que implementemos una acción, imprimiremos esta recompensa. El método `step(action)` proporciona esta recompensa.

```

1  # 1. Randomly choose an action using action_space.sample()
2  action = environment.action_space.sample()
3
4  # 2. Implement this action and move the agent in the desired direction
5  new_state, reward, done, info = environment.step(action)
6
7  # Display the results (reward and map)
8  environment.render()
9  print(f'Reward = {reward}')
```



La situación es desafiante: la recompensa actual es 0, y solo un estado específico ofrece una recompensa positiva en todo el juego. ¿Cómo podemos encontrar la ruta correcta si solo sabemos al final si hemos tenido éxito? Para obtener una recompensa, debemos confiar en la casualidad de encontrar la secuencia correcta de acciones. Lamentablemente, esa es la naturaleza del proceso: la tabla Q se llena de ceros hasta que el agente encuentre la meta G por casualidad.

Sería más fácil si hubiera recompensas más pequeñas durante el camino hacia la meta G. Sin embargo, este es un desafío clave del aprendizaje por refuerzo: las recompensas son escasas, lo que dificulta entrenar a los agentes cuando la única recompensa está al final de una larga secuencia de acciones. Se han propuesto técnicas para abordar esto, pero exploraremos estas soluciones en otro momento.

3. Q-Aprendizaje

Volvamos al problema. En este juego, encontrar la meta G es una cuestión de suerte. Pero una vez que la alcanzamos, ¿cómo retrocedemos esa información al estado inicial? El algoritmo Q-learning ofrece una solución inteligente a este dilema. Actualiza los valores de nuestros pares estado-acción (representados en la tabla Q) considerando la recompensa por alcanzar el próximo estado y el máximo valor en ese estado.

Supongamos que obtenemos una recompensa de 1 al llegar a G. El valor del estado junto a G (denominémoslo G-1) se incrementa gracias a esta recompensa. Ahora, cada vez que el agente esté en un estado cercano a G-1, aumentará el valor de G-1, y así sucesivamente hasta alcanzar el estado inicial S.

Intentemos encontrar la fórmula de actualización para propagar los valores desde G hasta S. Los valores representan la calidad de una acción en un estado específico (0 si es pobre, 1 si es la mejor posible). Queremos actualizar el valor de la acción en el estado (por ejemplo, cuando el agente está en el estado inicial S). Esta actualización se realiza con la fórmula:

$$Q_{nuevo}(s, a) = Q_{viejo}(s, a) + \alpha \cdot \left(R + \gamma \cdot \max_{a'} Q_{viejo}(s', a') - Q_{viejo}(s, a) \right)$$

Donde: Q_{nuevo} es el nuevo valor, Q_{viejo} es el valor existente, α es la tasa de aprendizaje (entre 0 y 1), R es la recompensa para el siguiente estado, γ es el factor de descuento (entre 0 y 1).

Con el algoritmo Q-learning, la actualización se realiza así:

$$Q_{nuevo}(s, a) = Q_{viejo}(s, a) + \alpha \cdot \left(R + \gamma \cdot \max_{a'} Q_{viejo}(s', a') - Q_{viejo}(s, a) \right)$$

Es crucial encontrar un equilibrio entre el conocimiento pasado (α) y el peso de las recompensas futuras (γ). Luego, entrenar al agente implica:

1. Elegir una acción aleatoria si los valores en el estado actual son ceros, o tomar la acción con el mayor valor.
2. Implementar la acción y actualizar el valor del estado actual según la fórmula de Q-learning.
3. Repetir hasta que el agente alcance la meta o se atasque.

Este proceso se repite durante varios episodios, reiniciando el entorno después de cada uno. Graficar los resultados puede ayudar a visualizar el progreso del agente.

```

1 plt.rcParams['figure.dpi'] = 300
2 plt.rcParams.update({'font.size': 17})
3
4 # We re-initialize the Q-table
5 qtable = np.zeros((environment.observation_space.n, environment.
6                     action_space.n))
7
8 # Hyperparameters
9 episodes = 1000          # Total number of episodes
10 alpha = 0.5              # Learning rate
11 gamma = 0.9              # Discount factor
12
13 # List of outcomes to plot
14 outcomes = []
15
16 print('Q-table before training:')
17 print(qtable)
18
19 # Training
20 for _ in range(episodes):
21     state = environment.reset()
22     state = int(state[0])

```

```

22     done = False
23
24     # By default, we consider our outcome to be a failure
25     outcomes.append("Failure")
26
27     # Until the agent gets stuck in a hole or reaches the goal, keep
28     # training it
29     while not done:
30         # Choose the action with the highest value in the current state
31         if np.max(qtable[state]) > 0:
32             action = np.argmax(qtable[state])
33
34         # If there's no best action (only zeros), take a random one
35         else:
36             action = environment.action_space.sample()
37
38         # Implement this action and move the agent in the desired
39         # direction
40         new_state, reward, done, truncated, info = environment.step(action)
41
42         # Update Q(s,a)
43         qtable[state, action] = qtable[state, action] + \
44             alpha * (reward + gamma * np.max(qtable[
45                 new_state]) -
46                 qtable[state, action])
47
48         # Update our current state
49         state = new_state
50
51         # If we have a reward, it means that our outcome is a success
52         if reward:
53             outcomes[-1] = "Success"
54
55     print()
56     print('=====')
57     print('Q-table after training:')
58     print(qtable)
59
60     # Plot outcomes
61     plt.figure(figsize=(12, 5))
62     plt.xlabel("Run number")
63     plt.ylabel("Outcome")
64     ax = plt.gca()
65     plt.bar(range(len(outcomes)), outcomes, width=1.0)
66     plt.show()

```

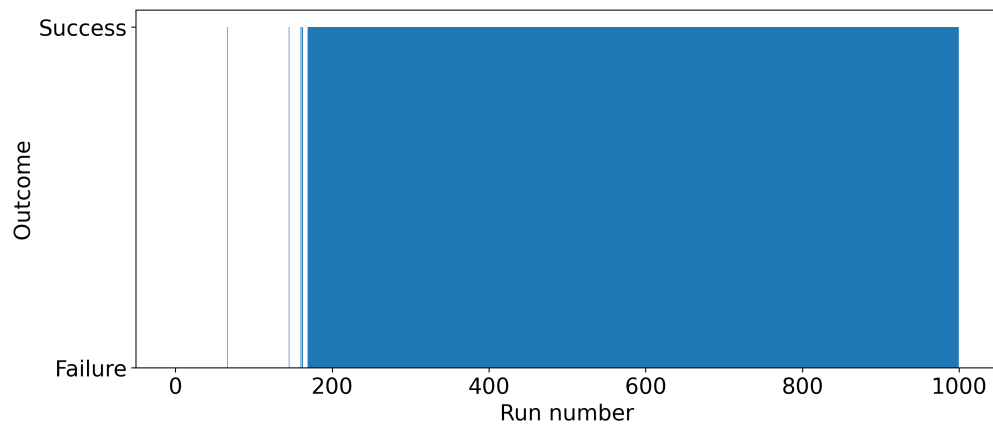


```

=====
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.      0.59049  0.      0.      ]
 [0.      0.      0.      0.      ]
 [0.      0.      0.      0.      ]
 [0.      0.      0.      0.      ]
 [0.      0.6561  0.      0.      ]
 [0.      0.      0.      0.      ]
 [0.      0.2784375 0.      0.      ]
 [0.      0.      0.      0.      ]
 [0.      0.      0.729  0.      ]
 [0.      0.      0.81   0.      ]
 [0.      0.9     0.      0.      ]
 [0.      0.      0.      0.      ]
 [0.      0.      0.      0.      ]
 [0.      0.      0.      0.      ]
 [0.      0.      1.      0.      ]
 [0.      0.      0.      0.      ]

```



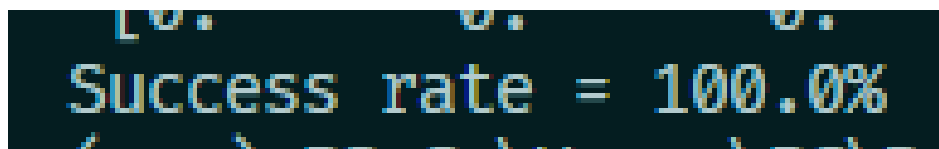
Después de entrenar al agente, observamos un progreso significativo. Al principio, le costó encontrar la meta, pero

con el tiempo, logró alcanzarla de manera más consistente. La tabla Q resultante muestra las acciones aprendidas por el agente para lograr su objetivo. Evaluamos su rendimiento en 100 episodios para medir su tasa de éxito en llegar a la meta.

```

1     episodes = 100
2     nb_success = 0
3
4     # Evaluation
5     for _ in range(episodes):
6         state = environment.reset()
7         state = int(state[0])
8         done = False
9
10        # Until the agent gets stuck or reaches the goal, keep training it
11        while not done:
12            # Choose the action with the highest value in the current state
13            if np.max(qtable[state]) > 0:
14                action = np.argmax(qtable[state])
15
16            # If there's no best action (only zeros), take a random one
17            else:
18                action = environment.action_space.sample()
19
20            # Implement this action and move the agent in the desired
21            # direction
22            new_state, reward, done, truncated, info = environment.step(action)
23
24            # Update our current state
25            state = new_state
26
27            # When we get a reward, it means we solved the game
28            nb_success += reward
29
30        # Let's check our success rate!
31        print (f"Success rate = {nb_success/episodes*100}%")

```



Vemos que hemos conseguido una tasa de éxito del 100 %. Con esto queda resuelto el Frozen Lake no resbaladizo.

Además podemos ver los movimientos del agente y mostrar la secuencia de acciones para saber si se eligió la mejor:

```

1     state = environment.reset()

```

```
29 print(f"Sequence = {sequence}")
```

(env) PS C:\Users\ps\Documents\U

manera óptima.

ABAJO, ABAJO, ABAJO, DERECHA, es precisamente la secuencia que anticipamos al inicio.

4. Algoritmo Epsilon-Greedy

A pesar del éxito obtenido, hay un inconveniente en nuestro enfoque previo: el agente siempre elige la acción con el valor más alto. Esto significa que las otras acciones nunca se exploran ni actualizan en su valor, lo cual podría limitar el aprendizaje del agente. ¿Qué pasaría si alguna de estas acciones no exploradas fuera mejor que la opción que el agente siempre elige? ¿No deberíamos incentivar al agente para que explore nuevas acciones y así mejorar?

En resumen, deseamos que nuestro agente:

- Elija la acción más valiosa (explotación).
- Seleccione ocasionalmente acciones aleatorias para descubrir alternativas potencialmente mejores (exploración).

Queremos equilibrar estos comportamientos: si el agente solo explota, no aprenderá más, pero si solo explora, el entrenamiento será ineficiente. Por ello, ajustaremos un parámetro con el tiempo: al principio, se preferirá la exploración extensa, pero a medida que el agente conozca mejor el entorno, la exploración se reducirá. Este parámetro, denominado ϵ , controla el nivel de aleatoriedad en la selección de acciones.

Este enfoque se conoce como algoritmo epsilon-greedy, donde epsilon es el parámetro clave. Es un método sencillo pero muy efectivo para lograr un equilibrio adecuado. En cada decisión, el agente tiene una probabilidad ϵ de seleccionar aleatoriamente una acción y una probabilidad de 1 menos ϵ de elegir la acción más valiosa. Podemos reducir gradualmente el valor de ϵ al final de cada episodio, ya sea en cantidades fijas (decaimiento lineal) o en función de su valor actual (decaimiento exponencial).

Vamos a ver como hacerlo en código:

```

1      # We re-initialize the Q-table
2      qtable = np.zeros((environment.observation_space.n, environment.
          action_space.n))
3
4      # Hyperparameters
5      episodes = 1000          # Total number of episode
6      salpha = 0.5             # Learning rate
7      gamma = 0.9             # Discount factor
8      epsilon = 1.0           # Amount of randomness in the action selection
9      epsilon_decay = 0.001    # Fixed amount to decrease
10
11     # List of outcomes to plot
12     outcomes = []
13
14     print('Q-table before training:')
15     print(qtable)
16
17     # Training
18     for _ in range(episodes):
19         state = environment.reset()
20         state = int(state[0])
21         done = False

```

```

22
23     # By default, we consider our outcome to be a failure
24     outcomes.append("Failure")
25
26     # Until the agent gets stuck in a hole or reaches the goal, keep
27     # training it
28     while not done:
29         # Generate a random number between 0 and 1
30         rnd = np.random.random()
31         # If random number < epsilon, take a random action
32         if rnd < epsilon:
33             action = environment.action_space.sample()
34         # Else, take the action with the highest value in the current
35         # state
36         else:
37             action = np.argmax(qtable[state])
38
39         # Implement this action and move the agent in the desired
40         # direction
41         new_state, reward, done, truncated, info = environment.step(action
42         )
43
44         # Update Q(s,a)
45         qtable[state, action] = qtable[state, action] + \
46             alpha * (reward + gamma * np.max(qtable[
47                 new_state]) - qtable[state, action])
48
49         # Update our current state
50         state = new_state
51
52         # If we have a reward, it means that our outcome is a success
53         if reward:
54             outcomes[-1] = "Success"
55
56         # Update epsilon
57         epsilon = max(epsilon - epsilon_decay, 0)
58
59     print()
60     print('=====')
61     print('Q-table after training:')
62     print(qtable)
63
64     # Plot outcomes
65     plt.figure(figsize=(12, 5))
66     plt.xlabel("Run number")
67     plt.ylabel("Outcome")
68     ax = plt.gca()
69     plt.bar(range(len(outcomes)), outcomes, width=1.0)
70     plt.show()

```

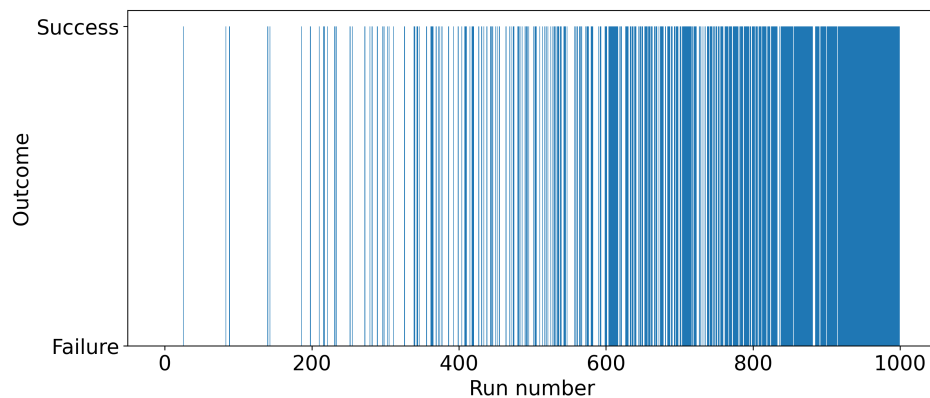
```

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.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]

=====

Q-table after training:
[[0.531441 0.59049 0.59049 0.531441 ]
 [0.53143933 0. 0.6561 0.59048966]
 [0.59048798 0.729 0.59037812 0.65609978]
 [0.65608833 0. 0.50043901 0.53419925]
 [0.59049 0.6561 0. 0.531441 ]
 ...
 [0. 0. 0. 0. ]
 [0. 0.80968849 0.9 0.72897477]
 [0.81 0.9 1. 0.81 ]
 [0. 0. 0. 0. ]]

```



Con estos resultados, vemos que ahora el agente ahora tarda más en ganar de manera constante, lo que indica que ha aprendido múltiples formas de alcanzar el objetivo. La tabla Q tiene más valores diferentes de cero debido a la exploración forzada de nuevos pares estado-acción. Sin embargo, para evaluar su rendimiento, deseamos ver si este agente tiene éxito como el anterior. Durante la evaluación, ya no requerimos exploración, ya que el agente está completamente entrenado. Vamos a implementarlo en código:

```

1 episodes = 100

```

```

2     nb_success = 0
3
4     # Evaluation
5     for _ in range(100):
6         state = environment.reset()
7         state = int(state[0])
8         done = False
9
10        # Until the agent gets stuck or reaches the goal, keep training it
11        while not done:
12            # Choose the action with the highest value in the current state
13            action = np.argmax(qtable[state])
14
15            # Implement this action and move the agent in the desired
16            # direction
17            new_state, reward, done, truncated, info = environment.step(action)
18
19            # Update our current state
20            state = new_state
21
22            # When we get a reward, it means we solved the game
23            nb_success += reward
24
25        # Let's check our success rate!
26        print (f"Success rate = {nb_success/episodes*100}%")

```



Success rate = 100.0%

Vemos que el rendimiento sigue siendo del 100 por ciento, y el modelo es ahora más flexible, ya que el agente ha aprendido varios caminos para llegar a G. Sin embargo, esto puede causar problemas en términos de rendimiento, pero es importante disponer de varias soluciones en lugar de una sola.

5. Desafío: Lago helado resbaladizo

Vamos a ver ahora la variante resbaladiza. Para ello, en el código, ponemos *is_slippery* = *True*, y el agente ahora sólo tendrá un 33 % de probabilidades de realizar el movimiento. Si fallamos, vamos aleatoriamente a una de las otras 3 acciones. Vamos a verlo en nuestro código:

```

1  # Initialize the slippery Frozen Lake
2  environment = gym.make("FrozenLake-v1", is_slippery=True, render_mode="
    rgb_array")
3  environment.reset()
4
5  # We re-initialize the Q-table
6  qtable = np.zeros((environment.observation_space.n, environment.
    action_space.n))
7
8  # Hyperparameters
9  episodes = 1000          # Total number of episodes
10 alpha = 0.5              # Learning rate
11 gamma = 0.9              # Discount factor
12 epsilon = 1.0            # Amount of randomness in the action selection
13 epsilon_decay = 0.001    # Fixed amount to decrease
14
15 # List of outcomes to plot
16 outcomes = []
17
18 print('Q-table before training:')
19 print(qtable)
20
21 # Training
22 for _ in range(episodes):
23     state = environment.reset()
24     state = int(state[0])
25     done = False
26
27     # By default, we consider our outcome to be a failure
28     outcomes.append("Failure")
29
30     # Until the agent gets stuck in a hole or reaches the goal, keep
31     # training it
32     while not done:
33         # Generate a random number between 0 and 1
34         rnd = np.random.random()
35
36         # If random number < epsilon, take a random action
37         if rnd < epsilon:
38             action = environment.action_space.sample()
39
40         # Else, take the action with the highest value in the current
41         # state
42         else:
43             action = np.argmax(qtable[state])
44
45         # Implement this action and move the agent in the desired
46         # direction
47         new_state, reward, done, truncated, info = environment.step(action
48             )
49
50         # Update Q(s,a)

```



```
47     qtable[state, action] = qtable[state, action] + \
48         alpha * (reward + gamma * np.max(qtable[
49             new_state]) - qtable[state, action])
50
51     # Update our current state
52     state = new_state
53
54     # If we have a reward, it means that our outcome is a success
55     if reward:
56         outcomes[-1] = "Success"
57
58     # Update epsilon
59     epsilon = max(epsilon - epsilon_decay, 0)
60
61     print()
62     print('=====')
63     print('Q-table after training:')
64     print(qtable)
65
66     # Plot outcomes
67     plt.figure(figsize=(12, 5))
68     plt.xlabel("Run number")
69     plt.ylabel("Outcome")
70     ax = plt.gca()
71     plt.bar(range(len(outcomes)), outcomes, width=1.0)
72     plt.show()
```

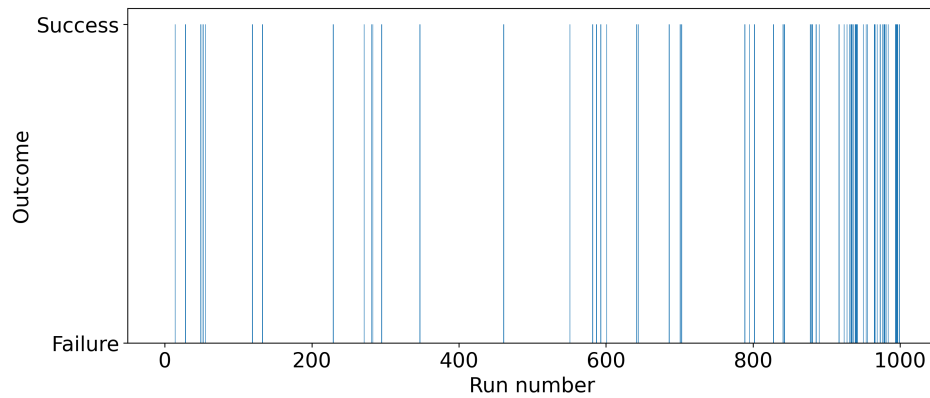
```

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:
[[0.06589316 0.04564106 0.0457194 0.04537534]
 [0.00730303 0.02500573 0.01741667 0.05022884]
 [0.0805941 0.02190637 0.02629862 0.02517283]
 [0.01165542 0.01244329 0.01737831 0.0234233 ]
 [0.1048828 0.0334248 0.04800678 0.03423923]
 ...
 [0. 0. 0. 0. ]
 [0.08397768 0.1446028 0.58595241 0.1521196 ]
 [0.41196023 0.88516932 0.40280423 0.38744793]
 [0. 0. 0. 0. ]]

```



Vamos a ver que ha ocurrido con la tasa de éxito...

```

1 episodes = 100
2 nb_success = 0
3
4 # Evaluation
5 for _ in range(100):

```

```

6     state = environment.reset()
7     state = int(state[0])
8     done = False
9
10    # Until the agent gets stuck or reaches the goal, keep training it
11    while not done:
12        # Choose the action with the highest value in the current state
13        action = np.argmax(qtable[state])
14
15        # Implement this action and move the agent in the desired
16        # direction
17        new_state, reward, done, truncated, info = environment.step(action)
18
19        # Update our current state
20        state = new_state
21
22        # When we get a reward, it means we solved the game
23        nb_success += reward
24
25    # Let's check our success rate!
26    print (f"Success rate = {nb_success/episodes*100} %")

```



Success rate = 72.0%

Podemos ver que la tasa ha empeorado, siendo ahora del 72 %. Sin embargo este porcentaje se puede aumentar ajustando unos diversos parámetros.

6. Conclusión

El algoritmo Q-learning es fundamental en el aprendizaje por refuerzo, un método poderoso y simple. En esta práctica:

- Aprendimos a interactuar con el entorno Gym para tomar decisiones y mover a nuestro agente.
- Presentamos la tabla Q, que representa valores de acciones en estados específicos.
- Experimentamos con la fórmula de actualización del Q-learning para abordar la escasez de recompensas.

- Desarrollamos un proceso de entrenamiento y evaluación que resolvió el desafío Frozen Lake con una tasa de éxito del 100
- Implementamos el conocido algoritmo ϵ -greedy para equilibrar la exploración y explotación de pares de estado-acción.

Si bien el entorno Frozen Lake es simple, en entornos más complejos, el almacenamiento de la tabla Q puede ser imposible debido al gran número de estados y acciones, especialmente en entornos continuos (como Super Mario Bros. o Minecraft). Para abordar esto, una técnica popular involucra el uso de redes neuronales profundas para aproximar la tabla Q. Sin embargo, esto añade complejidad debido a la inestabilidad de las redes neuronales.