APRENDIZAJE AUTOMATICO

INFORME TÉCNICO UNIDAD IV – APRENDIZAJE POR REFORZAMIENTO

PRESENTADO POR

Daniel José Rueda Lobato

Diego Andrés Valderrama Laverde

Catalina Piedrahita Jaramillo

PROFESOR:

José Lisandro Aguilar Castro

UNIVERSIDAD EAFIT

MEDELLÍN

MAESTRÍA EN CIENCIAS DE LOS DATOS Y ANALÍTICA

SEPTIEMBRE 2020

1. OBJETIVO DE LA ITERACIÓN

El objetivo de este trabajo es afianzar los conocimientos vistos en clase acerca del aprendizaje reforzado y aplicarlos en un ejemplo de manera didáctica.

2. CONTEXTUALIZACIÓN DEL PROBLEMA

Todos los seres vivos exhiben algún tipo de comportamiento, en el sentido que realizan alguna acción como respuesta a las señales que reciben del entorno en el que viven. Algunos de ellos, además, modifican su comportamiento a lo largo del tiempo, de forma que ante señales equivalentes se comportan de forma distinta con el paso del tiempo.

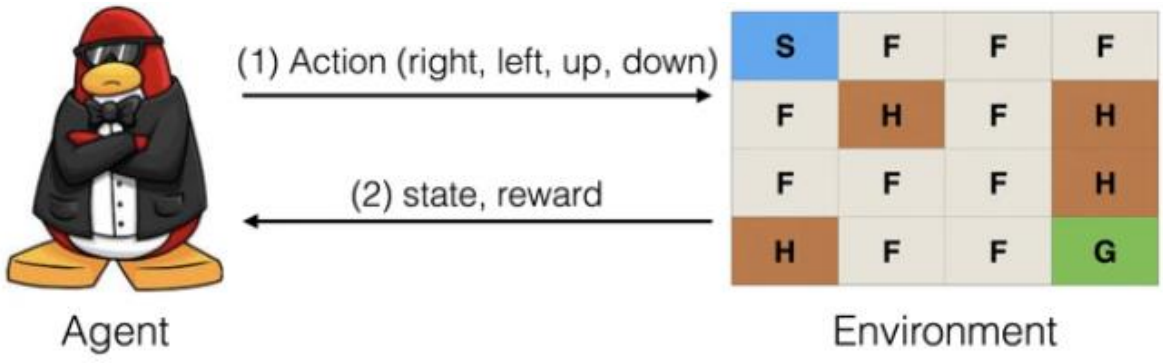
En algunos casos decimos que estos seres vivos han aprendido de su entorno, independientemente de si esa variación en la respuesta ha generado alguna ventaja o no en el individuo. De hecho, esta variación es la que aprovechamos cuando entrenamos un perro para que se siente cuando se lo pidamos. El campo del aprendizaje automático estudia la generación de algoritmos que sean capaces de aprender de su entorno, en este caso, el entorno es el conjunto de datos que el algoritmo recibe en una etapa particular que se llama entrenamiento.

Específicamente en el aprendizaje por reforzamiento, el aprendiz recibe información acerca de lo que es apropiado, y en caso contrario, el entorno le informa acerca de que el comportamiento ha sido inapropiado y normalmente cuánto error se ha cometido.

El objetivo del aprendizaje por refuerzo es extraer qué acciones deben ser elegidas en los diferentes estados para maximizar la recompensa. En cierta forma, buscamos que el agente aprenda lo que se llama una política, que formalmente podemos verla como una aplicación que dice en cada estado, qué acción tomar.

El problema seleccionado para afianzar los conocimientos de aprendizaje por reforzamiento es Frozen Lake. El entorno de Frozen Lake para este ejemplo será una cuadrícula de 4 × 4 que contiene cuatro áreas posibles: Safe (S), Frozen (F), Hole (H) y Goal (G). El agente se mueve alrededor de la cuadrícula hasta que llega a la meta o al hoyo. Si cae en el agujero, tiene que empezar desde el principio y se le recompensa con el valor 0. El proceso continúa hasta que aprende de cada error y finalmente alcanza la meta. Aquí hay una descripción visual de la cuadrícula de Frozen Lake (4 × 4):

Gráfica 1 Descripción visual cuadrícula de Frozen Lake



Implementaremos una de las técnicas de aprendizaje por refuerzo, Q-Learning. Este entorno permitirá que el agente se mueva en consecuencia. Podría ocurrir una acción aleatoria una vez cada

pocos episodios; digamos que el agente se desliza en diferentes direcciones porque es difícil caminar sobre una superficie congelada. Teniendo en cuenta esta situación, debemos permitir algún movimiento aleatorio al principio, pero eventualmente intentaremos reducir su probabilidad. De esta forma podemos corregir el error causado minimizando la pérdida.

Veamos con un poco más de detalle el algoritmo de Q-Learning:

Todo lo que necesitamos almacenar en memoria durante el aprendizaje es una tabla con las recompensas para estados y acciones. En nuestro caso es una tabla de 16 filas por 4 columnas. Esta tabla va a contener la recompensa mixta para cada celda. A medida que el agente va aprendiendo de distintas experiencias, se va volviendo más y más preciso. El algoritmo necesita dos parámetros que debemos ajustar en función del problema que estemos resolviendo:

- Velocidad de aprendizaje (*learning rate*). Es un valor entre 0 y 1 que indica cuánto podemos aprender de cada experiencia. 0 significa que no aprendemos nada de una nueva experiencia, y 1 significa que olvidamos todo lo que sabíamos hasta ahora y nos fiamos completamente de la nueva experiencia.
- Factor de descuento (*discount factor*). Es también un valor entre 0 y 1 que indica cuán importante es el largo plazo. 0 significa que sólo nos importan los refuerzos inmediatos, y 1 significa que los refuerzos inmediatos no importan, sólo importa el largo plazo.

En ambos parámetros los extremos son poco útiles. La velocidad de aprendizaje se puede ajustar en función de la incertidumbre respecto a los estados siguientes en las experiencias. Por ejemplo, en el 3 en raya, la incertidumbre viene de las posibles jugadas del rival, que puede realizar hasta 9 acciones diferentes. Por tanto, una velocidad de aprendizaje superior a 1/9 da un peso excesivo a las nuevas experiencias y olvida demasiado rápido las anteriores.

El factor de descuento establece un balance entre el refuerzo inmediato y el refuerzo a largo plazo. En el caso del 3 en raya, sólo recibimos refuerzos a corto plazo cuando se acaba la partida, así que este factor no importa mucho, cualquier valor entre 0.1 y 0.9 funcionará muy bien. En otros problemas donde se puedan recibir refuerzos intermedios, debemos decidir dónde poner la balanza.

El algoritmo, tiene una función que calcula la calidad (Q) de una combinación estado-acción:

$$Q : S \times A \to \mathbb{R}.$$

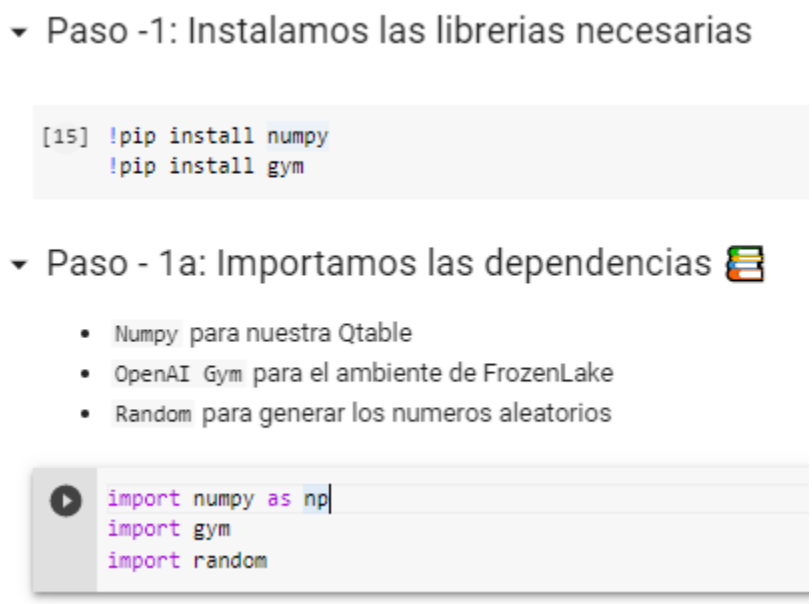
Antes que comience el aprendizaje, Q se inicializa a un valor arbitrario constante (escogido por el programador). Después, en cada tiempo t el agente selecciona una acción a_t, observa una recompensa r_t, introduce un estado nuevo S_{t+1} (Que depende del estado anterior S_t y de la acción seleccionada), y Q se actualiza. El núcleo del algoritmo es una actualización del valor de la iteración simple, haciendo la media ponderada del valor antiguo y la información nueva:

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

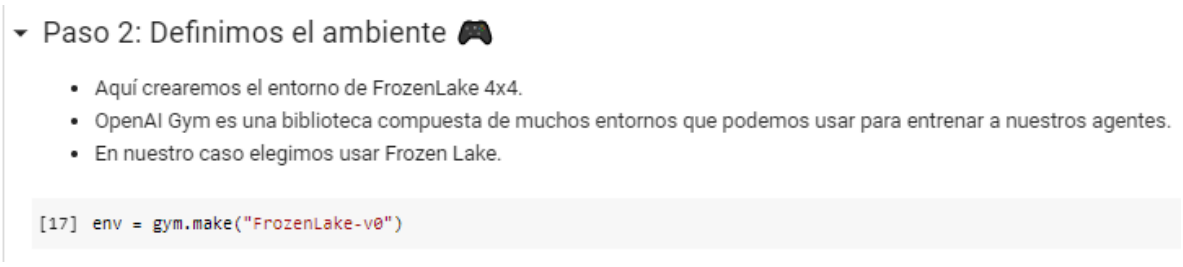
Donde r_t es la recompensa recibida al pasar del estado S_t al estado S_{t+1}, y α es el índice de aprendizaje ($0 < \alpha < 1$). Un episodio del algoritmo termina cuando el estado S_{t+1} es un estado final o terminal. Aun así, Q-learning también puede aprender en tareas sin episodios. Si el factor de descuento es menor que 1, los valores de acción son finitos incluso si el problema puede contener bucles infinitos.

3. DISEÑO DEL MODELO

Inicialmente instalamos e instanciamos las librerías necesarias para abordar el problema, que para este caso fueron numpy y gym.



Como paso siguiente, definimos el ambiente, el cual consiste en invocar el entorno de Frozen Lake.



Posteriormente definimos la tabla Q que almacenará las acciones y estados. Inicialmente se instancian con ceros.

Paso 3: Creamos la Q-tabla y la inicializamos

- Crearemos nuestra tabla Q, para saber cuántas filas (estados) y columnas (acciones) necesitamos, necesitamos calcular el tamaño_de_acción y el tamaño_de_estado
- OpenAI Gym nos proporciona una forma de hacerlo: `env.action_space.n` y `env.observation_space.n`

```
[18] action_size = env.action_space.n
     state_size = env.observation_space.n
```

```
[19] qtable = np.zeros((state_size, action_size))
     print(qtable)
```

A continuación definimos los valores de nuestros hiperparametros, los cuales determinaran las condiciones iniciales de nuestro algoritmo:

Paso 4: Creamos los hiperparametros

```
[20] total_episodes = 1000       # Total episodios. 20000 iniciales
     learning_rate = 0.7          # Tasa de aprendizaje
     max_steps = 99               # Número máximo de pasos por episodio
     gamma = 0.95                 # Tasa de descuento

     # Parámetros de exploración
     epsilon = 1.0                 # Tasa de exploración
     max_epsilon = 1.0             # Probabilidad de exploración al inicio
     min_epsilon = 0.01            # Probabilidad de exploración al inicio
     decay_rate = 0.005            # Tasa de decrecimiento exponencial

     steps_total = []
     rewards_total = []
     egreedy_total = []
```

Nuestro siguiente paso, consiste en la implementación del algoritmo, a grandes rasgos tenemos un ciclo externo que itera sobre el numero total de episodios, y un ciclo interno que itera sobre el máximo numero de pasos.

Se genera un numero aleatorio con distribución uniforme entre 0 y 1, y si el numero es mayor que el épsilon inicial definido en los hiperparametros, realiza una explotación (tomando el mayor valor Q para este estado), de lo contrario realiza una elección al azar (exploración).

Posteriormente toma la acción (a) y observa los estados de resultado y la recompensa (r) y actualiza el valor de Q (tal cual como indica la explicación del algoritmo en el punto anterior).

Fuera del ciclo se reduce el valor del épsilon, puesto que cada que se avance necesitamos menos exploración.

A continuación veamos la implementación realizada:

Paso 5: El algoritmo Q learning 🧠

- Ahora implementamos el algoritmo Q learning :

```
# Lista de recompensas
rewards = []
# Para siempre o hasta que el aprendizaje se detenga
for episode in range(total_episodes):
    # Resetiamos el ambiente
    state = env.reset()
    step = 0
    done = False
    total_rewards = 0
    for step in range(max_steps):
        # 3.Elige una acción en su mundo actual
        ## Inicialmente se genera un numero aleatorio
        exp_exp_tradeoff = random.uniform(0, 1)
        ## Si el numero es mayor al epsilon --> Realiza explotación (tomando el mayor valor Q para este estado)
        if exp_exp_tradeoff > epsilon:
            action = np.argmax(qtable[state,:])
            #print(exp_exp_tradeoff, "action", action)
        # De lo contrario hace una elección al azar --> Exploración
        else:
            action = env.action_space.sample()
            #print("action random", action)
        # Toma la acción (a) y observa los estados de resultado y la recompensa (r)
        new_state, reward, done, info = env.step(action)
        # Actualiza Q(s,a):= Q(s,a) + lr [R(s,a) + gamma * max Q(s',a') - Q(s,a)]
        # qtable[new_state,:] : todas las acciones que podemos tomar desde el nuevo estado
        qtable[state, action] = qtable[state, action] + learning_rate * (reward + gamma * np.max(qtable[new_state, :]) - qtable[state, action])
        total_rewards += reward
        # Nuestro nuevo estado es State
        state = new_state
        # terminar el episodio
        if done == True:
            steps_total.append(step)
            rewards_total.append(reward)
            egreedy_total.append(epsilon)
            break
    # Se reduce el épsilon (porque necesitamos cada vez menos exploración)
    epsilon = min_epsilon + (max_epsilon - min_epsilon)*np.exp(-decay_rate*episode)
    rewards.append(total_rewards)
```

Luego de nuestra implementación del algoritmo Q, usamos el resultado que se encuentra almacenado en nuestra matriz qtable y procedemos a jugar Frozen Lake:

▾ Paso 6: Usamos nuestra Q-table para Jugar FrozenLake !

- Despues de 10000 repeticiones, Nuestra Q-table puede ser usada como "truco" para jugar FrozenLake"
- Al ejecutar estas lineas se puede ver a nuestro agente jugando a FrozenLake.

```
env.reset()

for episode in range(5):
    state = env.reset()
    step = 0
    done = False
    print("************************************************")
    print("EPISODIO ", episode)

    for step in range(max_steps):

        # Toma la acción (índice) que tiene la máxima recompensa futura esperada
        action = np.argmax(qtable[state,:])

        new_state, reward, done, info = env.step(action)

        if done:

            env.render()
            if new_state == 15:
                print("Alcanzamos nuestro objetivo 🏆")
            else:
                print("Caímos en un agujero ☠")

            # Se imprime el numero de pasos en que finalizó.
            print("Numero de pasos", step)

            break
        state = new_state
env.close()
```

4. PRUEBAS Y ANÁLISIS DE RESULTADOS

A continuación se presenta una imagen que contiene algunas pruebas generadas con el resultado de la implementación del algoritmo Q. En todas el agente alcanzó el objetivo con un numero de pasos diferentes.

```
************************************************
EPISODIO  0
  (Down)
SFFF
FHFH
FFFH
HFF
Alcanzamos nuestro objetivo 🏆
Numero de pasos 21
************************************************
EPISODIO  1
  (Down)
SFFF
FHFH
FFFH
HFF
Alcanzamos nuestro objetivo 🏆
Numero de pasos 56
************************************************
EPISODIO  2
  (Down)
SFFF
FHFH
FFFH
HFF
Alcanzamos nuestro objetivo 🏆
Numero de pasos 38
```

Ahora bien, en términos de porcentajes, tenemos los siguientes resultados.

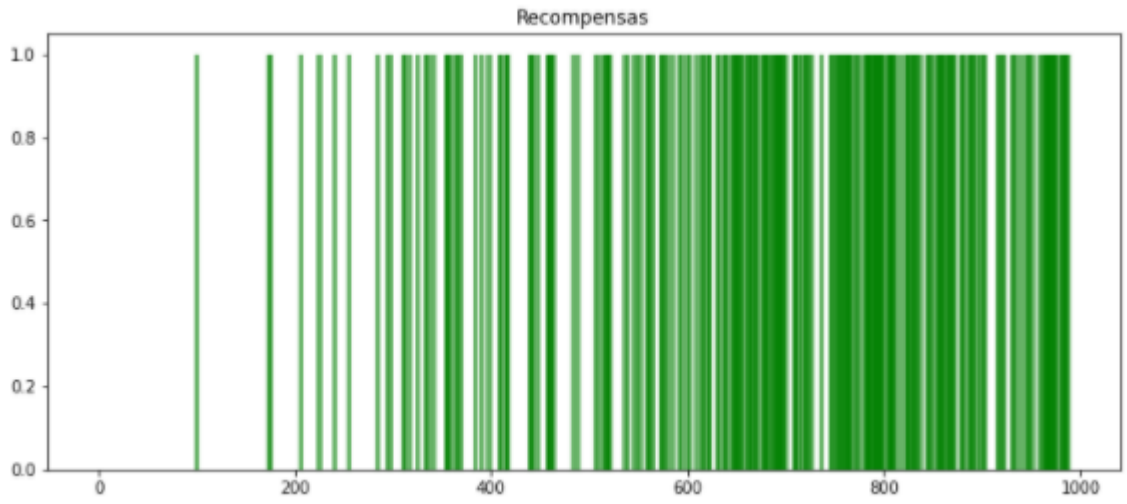
Tabla 1 Resultados Frozen Lake

Indicador	Resultado
Porcentaje de episodios finalizados satisfactoriamente	0.226
Porcentaje de episodios finalizados satisfactoriamente (Últimos 100 episodios)	0.37
Promedio de pasos	19.49
Promedio de pasos (Últimos 100 episodios)	26.75

Basados en un entrenamiento relativamente corto de 1000 episodios el porcentaje de los finalizados satisfactoriamente es del 22.6%, aclarando que en los últimos 100 subió considerablemente al 37%. Esto quiere decir que el algoritmo si aprende y que conforme se incrementa su entrenamiento (basado en aumentar el total de episodios) los resultados mejoran. Cabe aclarar que no necesariamente emplea menos pasos para lograr su objetivo, como se puede ver también incrementaron en los últimos 100 episodios.

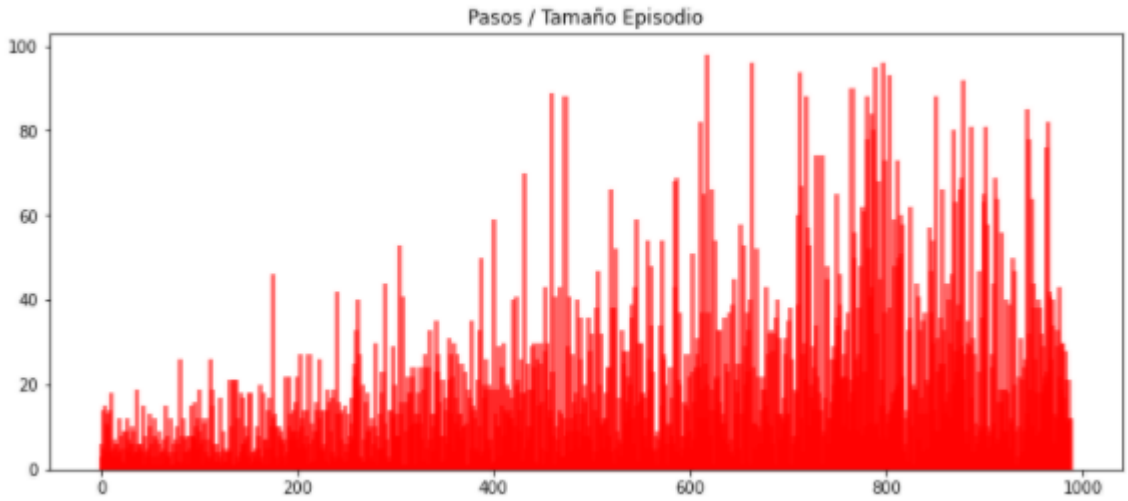
Veamos a continuación en la Gráfica 2, una relación de recompensas recibidas por el agente a medida que se incrementan los episodios.

Gráfica 2 Relación recompensa y número de episodios



Esta es otra clara muestra que conforme se aumentan los episodios, las recompensas también. Aunque el algoritmo emplea más pasos conforme aumentan los episodios como se puede apreciar en el siguiente gráfico.

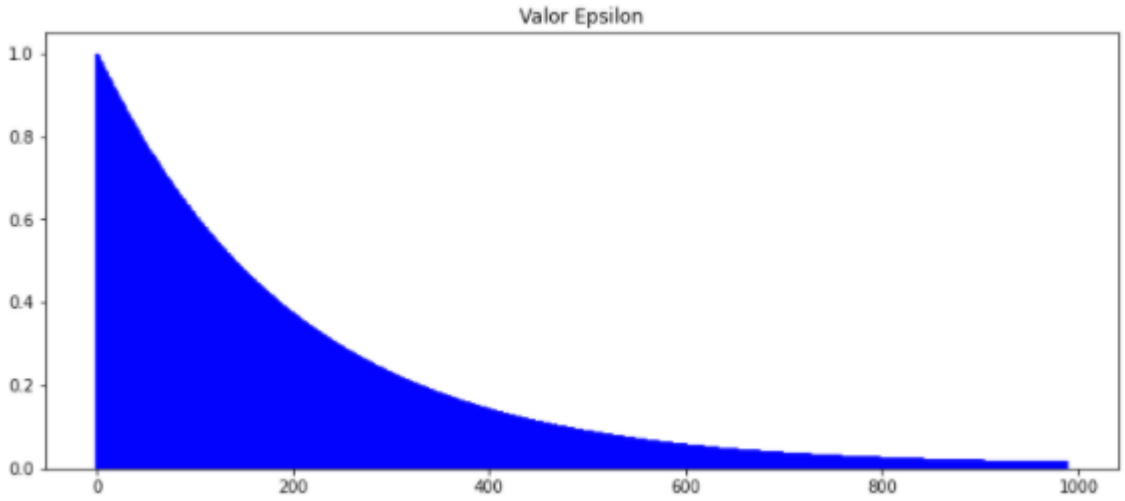
Gráfica 3 Número de pasos por episodio



Esto muestra que, si bien la precisión aumenta, toma más tiempo (más pasos) en llegar a la ruta satisfactoria, esto debido a la alta acción aleatoria del algoritmo.

Por último, adjuntamos un gráfico en el que se puede ver claramente la reducción del épsilon conforme aumentan los episodios. Esto demuestra la reducción de la exploración

Gráfica 4 Valor Épsilon



5. CONCLUSIÓN

Se ha ejecutado de forma exitosa un algoritmo de Q-Learning creando un agente y un entorno. Además, al realizar diferentes pruebas ha sido posible evidenciar como el agente aprendía, ya que a medida que se realizaban mas entrenamientos, este lograba ganar con más frecuencia el juego. Esto evidencia, como efectivamente a mayor nivel de entrenamiento, es mas contante el lograr la recompensa

REFERENCIAS

Maladkar, K. (19 de 03 de 2018). *analyticsindiamag*. Obtenido de Frozen Lake: Beginners Guide To Reinforcement Learning With OpenAI Gym: https://analyticsindiamag.com/openai-gym-frozen-lake-beginners-guide-reinforcement-learning/

Seo, J. D. (07 de 04 de 2018). *Towards Data Science*. Obtenido de My Journey to Reinforcement Learning — Part 1: Q-Learning with Table: https://towardsdatascience.com/my-journey-to-reinforcement-learning-part-1-q-learning-with-table-35540020bcf9

Sharma, A. (19 de 11 de 2019). *Kaggle*. Obtenido de Frozen Lake with OpenAI GYM: https://www.kaggle.com/sarjit07/reinforcement-learning-using-q-table-frozenlake

Thomas, S. (s.f.). Obtenido de https://colab.research.google.com/github/simoninithomas/Deep_reinforcement_learning_Course/blob/master/Q_Learning_with_FrozenLakev2.ipynb#scrollTo=Bt8UsREaBNkJ