Q-learning para principiantes

Andrea Manuel Simón y Ana López Palomo

December 2023

1 Introducción

En este proyecto vamos a enseñar a una IA a cómo resolver el entorno de Frozen Lake mediante el aprendizaje por refuerzo. Realizaremos el problema desde cero, intentando recrear el algortimo Q-learning nosotros mismos.

Primero comenzaremos instalando el entorno de Frozen Lake e importaremos las bibliotecas necesarias. Gym para el juego, numpy para hacer cálculos matemáticos.

1.1 Explicación del juego Frozen Lake

Frozen Lake es un entorno simple compuesto por mosaicos, donde la IA debe pasar del mosaico inicial al objetivo esquivando los obstáculos y haciéndolo en el menor número de movimientos posible. Los mosaicos pueden ser: lago congelado (seguro) o un agujero (no seguro y te atrapa para siempre). La IA (agente) tiene solo 4 acciones: arriba, abajo, derecha e izquierda.
El entorno siempre tiene la misma estructura, con letras que representan los distintos mosaicos.

Un ejemplo:
S F F F (S: starting point, safe)
F H F H (F: frozen surface, safe)
F F F H (H: hole, stuck forever)
H F F G (G: goal, safe)

Para asegurarnos de que has entendido el juego y lo que debe hacer la IA, vamos a realizarlo de forma manual.
Empezamos en S, despues realizamos las siguientes acciones: derecha –¿ derecha –¿ abajo –¿ abajo –¿ abajo –¿ derecha. Esta opción es correcta, por supuesto hay más, pero debemos de tener en cuenta la condición de realizarlo en el menor número de pasos posibles. En este entorno el número de pasos es 6, que coincide con nuestra solución.

Continuamos con inicializar el entorno con gym. Otra regla del juego es que el hielo puede ser resbaladizo, donde las acciones seleccionadas tienen una posibilidad aleatoria de ser ignoradas por el agente (IA), o antideslizante, donde no se ignoran las acciones. Comenzaremos con el segundo debido a su sencillez.

Obtendremos el siguiente entorno:
SFFF
FHFH
FFFH
HFFG
Es exactamente igual que nuestro ejemplo.

1.2 Tabla Q

En nuestro entorno tenemos 16 mosaicos en total, en otras palabras 16 posiciones diferentes o estados. En cada estado tenemos la posibilidad de realizar 4 acciones (arriba, abajo, derecha o izquierda), donde deberemos de elegir la mejor.
Una manera de representar todas las posibilidades es con una tabla Q, donde las filas enumeran todos los estados (s) y las columnas las acciones (a). En cada celda pondremos un medidor de calidad, Q(s, a), donde el estado será 0 si la elección es mala y 1 si es buena. Gracias a esto guiaremos de una manera eficiente al agente.

Estado ◀ IZQUIERDA	▼ ABAJO	▶ DERECHO	▲ ARRIBA
S=0 Q(0, ◀)	Q(0, ▼)	P(0, ▶)	Q(0, ▲)
1 P(1, ◀)	P(1, ▼)	P(1, ▶)	P(1, ▲)
2 Q(2, ◀)	Q(2, ▼)	P(2, ▶)	Q(2, ▲)
... ...	...	...	...
14 P(14, ◀)	P(14, ▼)	P(14, ▶)	P(14, ▲)
G=15 P(15, ◀)	P(15, ▼)	P(15, ▶)	P(15, ▲)

Figure 1: Enter Caption

Rellenaremos nuestra tabla con ceros, dado que aún no tenemos claro el valor de cada acción en cada estado.

Estableceremos que el agente realice una acción aleatoria mediante el random.choice de las 4 acciones, aunque en dos de ellas (arriba e izquierda) no hará nada al encontrarse en la salida.

Otra opción es utilizar la biblioteca gym para elegir el método de forma aleatoria. Este sacará un número, pero nosotros queremos la acción, por ello le asociaremos a cada acción un número.

Podemos implementar también reset() para que reinicie el juego si caemos en un hoyo o si llega a la meta y render() para visulizar el mapa.

1.3 Interacción con el entorno gym

En el aprendizaje por refuerzo los agentes son recompensados una vez que alcanzan el objetivo, en nuestro caso cuando alcanza G. Nos dará un 1 si llega a G y un cero en el caso contrario.

Tras realizar el código vemos que nos da un 0, por tanto concluimos que solo un estado puede darnos la recompensa positiva. Pero la tabla Q permanecerá igual, llena de ceros hasta que alcancemos G. Una opción sería poder darle

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

Figure 2: Enter Caption

recompensas intermedias, pero el método gym solo deja al final. Resolveremos este problema más adelante.

1.4 Q-apendizaje

Supongamos que hemos alcanzado la meta, ¿cómo reprogramamos la información al estado inicial?
Para ello necesitaremos actualizar el valor de cada celda en la tabla Q, donde consideraremos 1/ la recompensa por alcanzar el siguiente estado y 2/ el valor más alto posible en el siguiente estado.
Sabemos que obtenemos 1 como recompensa cuando llegamos a G, esto lo llamaremos G-1, donde la acción incrementará al llegar a la recompensa. Es decir, hemos ganado el juego y reiniciamos, la próxima vez que el agente esté en un estado próximo a G-1 aumentará el valor del estado a G-2 y así sucesivamente. Con todo esto, habremos actualizado todos los estados.
La lógica de esto sería:

$$Q_{new}(s_t, a_t) = Q(s_t, a_t) + r_t + max_a Q(s_{t+1}, a)$$

at número de columna, st número de fila en cada celda Q. rt es la recompensa para el siguiente estado (1/) y maxaQ(s(t+1), a) es el valor máximo posible en el siguiente estado (2/). Podemos actualizar el valor correspondiente a la acción ganadora en este estado G-1.
La próxima vez que el agente esté en un estado próximo a este (G-2) lo actualizaremos usando la fórmula y obtendremos lo mismo pero para G-2. A pesar

$Q_{new}(G-1,a_t) = Q(G-1,a_t) + r_t + max_a Q(G,a)$, dónde $Q(G-1,a_t) = 0$ y $max_a Q(G,a) = 0$ porque la tabla Q está vacía, y rt=1 porque obtenemos la única recompensa en este entorno. Obtenemos $Q_{new}(G-1,a_t) = 1$

de todo, esta no es la veradera fórmula de actualización de Q-learning. Esta es Entonces, entrenar a nuestro agente en código es:

$$Q_{new}(s_t,a_t) = Q(s_t,a_t) + \alpha \cdot (r_t + \gamma \cdot max_a Q(s_{t+1},a) - Q(s_t,a_t))$$

1. Elegir una acción aleatoria (usando action-space.sample()) si los valores en el estado actual son solo ceros. De lo contrario, tomamos la acción con el valor más alto en el estado actual con la función np.argmax();
2. Implementar esta acción moviéndose en la dirección deseada con step(action);
3. Actualizar el valor del estado original con la acción que realizamos, utilizando información sobre el nuevo estado y la recompensa otorgada por step(action)

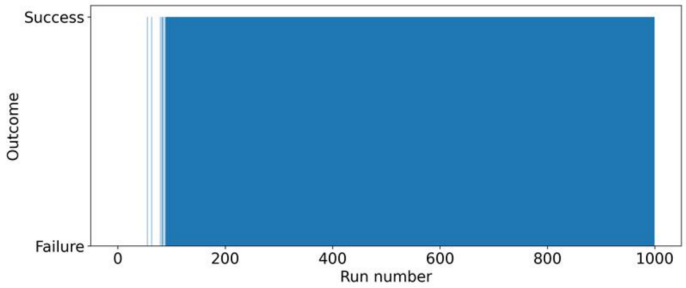
Repetimos estos pasos hasta que el agente llega a G o a un agujero, cuando llega lo reseteamos y lo volvemos a hacer, así 1000 veces.

```
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.]]
```

Figure 3: Enter Caption

Pintándolo conseguimos:

```
Q-table after training:
[[0.        0.59049   0.        0.      ]
 [0.        0.        0.        0.      ]
 [0.        0.0455625 0.        0.      ]
 [0.        0.        0.        0.      ]
 [0.        0.6561    0.        0.      ]
 [0.        0.        0.        0.      ]
 [0.        0.3796875 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.      ]]
```



Cada barra azul corresponde a una victoria, por lo que podemos ver que al agente le costó encontrar el final al comienzo del entrenamiento. Pero una vez que lo encontró varias veces seguidas, comenzó a ganar consistentemente.

Ahora veamos cómo funciona evaluándolo 100 veces. Consideramos que el entrenamiento ha terminado, por lo que ya no necesitamos actualizar la tabla Q. Para ver cómo se desempeña el agente, podemos calcular el porcentaje de veces que logró alcanzar la meta (tasa de éxito).
Conseguos que acierte todas las veces.

1.5 Algoritmo Epsilon-Greedy

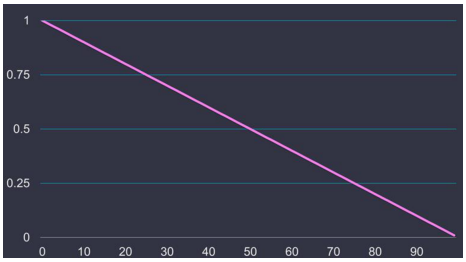
A pesar de conseguir entrenarlo con éxito hay algo que se puede mejorar. El agente siempre elige la acción con el valor más alto, entonces, siempre que un par estado-acción comience a tener un valor distinto de cero, el agente siempre lo elegirá. Las otras acciones nunca se realizarán, lo que significa que nunca actualizaremos su valor. Pero qué pasaría si alguna de estas acciones es mejor que la que realiza.
En otras palabras queremos que nuestro agente tome la acción con el valor más alto y que elija de forma aleatoria la acción para encontrar nuevas rutas. Para conseguir ambas sin que se centre en solo una haremos primero que se centre en la exploración y luego elija la mejor ruta por el valor más alto. Esta técnica se llama algoritmo ávido de épsilon. Épsilon es nuestro parámetro que cambiará dependiendo de si estamos explorando o eligiendo la mejor opción.

Cada vez que el agente tiene que realizar una acción, tiene una probabilidad ϵ de elegir uno al azar, y una probabilidad $1 - \epsilon$ de elegir la de mayor valor.

Podemos disminuir el valor de ϵ al final de cada episodio en una cantidad fija (decaimiento lineal) o en función del valor actual de ϵ (decaimiento exponencial).

Implementemos un decaimiento lineal, pero antes vamos a ver cómo se ve la curva con parámetros arbitrarios. Empezaremos con $\epsilon = 1$ para estar en modo de exploración completa, y disminuir este valor por 0.001 después de cada episodio.

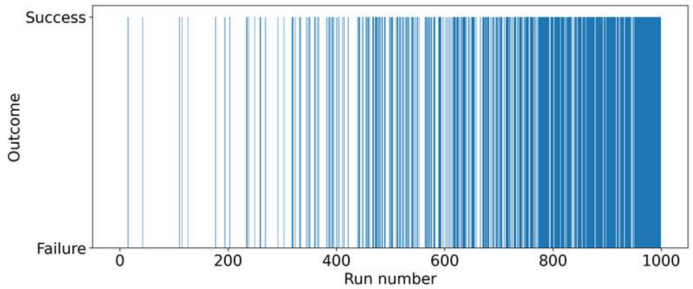
Nos sale:



Lo volvemos a entrenar pero con este nuevo 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.]]
```

```
Q-table after training:
[[0.531441   0.59049    0.59049    0.531441  ]
 [0.531441   0.         0.6561     0.59049   ]
 [0.59049    0.729      0.59049    0.6561    ]
 [0.6561     0.         0.59048879 0.59047491]
 [0.59049    0.6561     0.         0.531441  ]
 [0.         0.         0.         0.        ]
 [0.         0.81       0.         0.6561    ]
 [0.         0.         0.         0.        ]
 [0.6561     0.         0.729      0.59049   ]
 [0.65609944 0.80999966 0.81       0.        ]
 [0.729      0.9        0.         0.729     ]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.         0.60548384 0.9        0.72407524]
 [0.81       0.9        1.         0.81      ]
 [0.         0.         0.         0.        ]]
```



El agente tarda más en ganar de forma constante y la tabla tiene más valores que la anterior.
Lo ponemos en modo evaluación, ya que no queremos explorar, y vemos su rendimiento.
Conseguimos que acierte todas las veces.

Con este nuevo enfoque conseguimos que nuestro modelo sea más flexible y menos dinámico, de manera que aprende diferentes caminos en lugar de solo uno.