Aprendizaje Automático por Refuerzo Q-soAprendizaje

1st Alejandro Barranco Castro,
dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
alebarcas1@alum.us.es //
alex@barranco.ws

2nd Juan Manuel Varela Amaya,
dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
juavarama@alum.us.es //
juadevarama@gmail.com

Resumen: Este documento muestra nuestro proyecto en python sobre q-learning, en este caso lo usamos para seguir un mapa de celdas desde un punto A hasta un punto B, tomando el camino que minimice el costo al máximo.

Tras investigar y finalmente completar el proyecto, hemos obtenido resultados positivos.

Palabras clave: Inteligencia artificial, Tablero, Cuadrícula, q-learning.

1. Introducción

Nuestro trabajo se basa en resolver un problema mediante un tablero.2

Un agente al que llamaremos R (un **ratón**) debe atravesar el tablero, que cuenta con unas celdas que representan distintas zonas y el objetivo es ir desde un punto de inicio A hasta un punto final B al que conoceremos con **Queso** minimizando el coste de recorrer el camino.

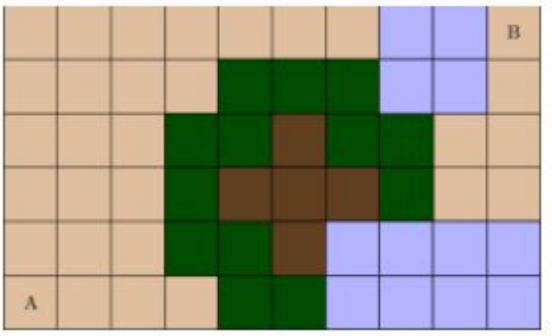

Fig. 1. Tablero ejemplo de nuestro trabajo.

Las celdas marrón claro representan zonas de llanura, las cuales hay que priorizar. Las verdes zonas boscosas, las marrón oscuro zonas montañosas y las azules zonas con agua, las cuales no pueden pisarse. Tras las zonas de llanura, debemos priorizar las zonas boscosas y por último a las zonas montañosas.

Nos encontramos ante un problema de Q-Learning. En Q-Learning se involucra a un agente (en nuestro ejemplo R, el ratón), un conjunto S, y un conjunto C de acciones por estado. Llevando a cabo una acción **a** perteneciente a **C**, el agente R pasa de un estado a otro **S**, a perteneciente a **C**. La

ejecución de una acción **a** concreta le proporciona una recompensa al agente.

2. ¿Cómo funciona el Q-Learning?

El Q-Learning es una técnica de **Aprendizaje por refuerzo,** que se ocupa de determinar qué acciones debe escoger un agente de software en un entorno dado con el fin de maximizar una "recompensa" o premio acumulado.

En aprendizaje de máquina, el medio ambiente es formulado generalmente como un **proceso de decisión de Markov (MDP)**, y muchos algoritmos de aprendizaje por refuerzo se relacionan con técnicas de la programación dinámica, de ahí que en otros campos de investigación donde se estudian los métodos de aprendizaje de refuerzo se le llame programación dinámica aproximada.

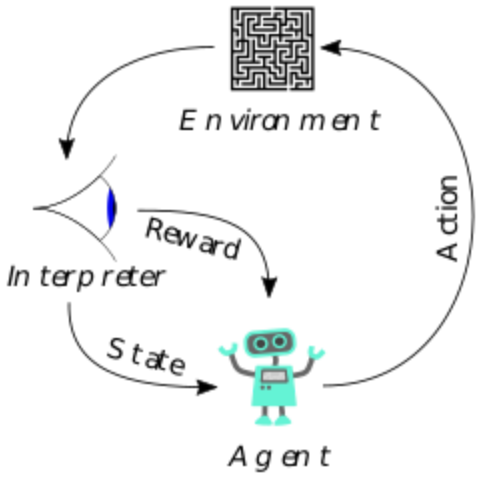

Fig. 2. Encuadre típico de un escenario de aprendizaje por refuerzo.

Un **proceso de decisión de Markov (MDP)** es una extensión de una cadena de Markov [3], en la cual tenemos dos estados **E** y **A**, y las probabilidades de pasar de un estado a otro.
En el **MDP**, se añade las **acciones**, que es la posibilidad de hacer una elección, y las **recompensas**, que provienen de tomar una u otra decisión para llegar a un nuevo estado.

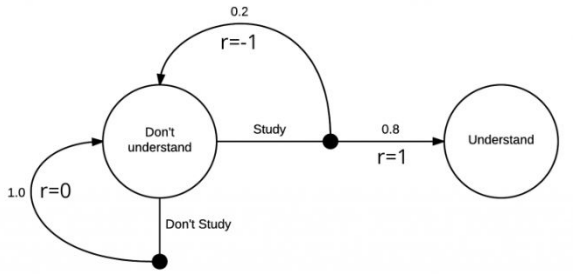

Fig. 3. Proceso de decisión de Marko

Para cualquier proceso de decisión de Markov finito, Q-Learning encuentra una política óptima maximizando la recompensa total. No podemos tomar la recompensa simplemente como el sumatorio de las **recompensas inmediatas** que obtenemos por realizar una acción en el entorno, porque contamos igual las recompensas inmediatas a las recompensas futuras.

Para mejorar esto, usamos un **factor de descuento γ,** que será un número entre 0 y 1 (0 <= γ <= 1), y determina la importancia de recompensas futuras. Un factor de 0 hace que el agente sea corto de miras, considerando solo las recompensas actuales, mientras que un factor que se acerque a 1 hará que luche por una recompensa alta a largo plazo.

$$\begin{aligned} R_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots \\ &= r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \dots) \\ &= r_{t+1} + \gamma R_{t+1} \end{aligned}$$

Fig. 4. Fórmula de la recompensa con factor

Por otro lado, tenemos el **reglamento**. Este es una función que dice que **acciones** deben tomarse en que **estados**. Esta función suele denominarse como **π(s,a)**, y representa la probabilidad de que ocurra la acción **a** en el

estado **s**. Como es una probabilidad de una acción, la suma de todas siempre debe ser 1.

Llegados a este punto vamos a introducir otros dos conceptos, la recompensa inmediata (que ya se ha mencionado antes) y la probabilidad de transición;

- La **recompensa inmediata** es la obtenida por ir desde el estado **s** al estado **s+1** a través de la acción **a**.
- La **probabilidad de transición** es la probabilidad de ir desde el estado **s** al estado **s+1** a través de la acción **a**.

Hablemos ahora de las **funciones valor**. Estas funciones son una forma de medir cómo de bueno es llegar a un estado o cómo de bueno es realizar cierta acción.

Por un lado, tenemos la **función de calidad del estado.** Esta representa la recompensa total esperada que podemos obtener empezando desde ese estado, y depende del **reglamento**.

$$V^{\pi}(s) = \mathbb{E}\big[R_t \quad | \quad s_t = s\big]$$

Fig. 5. Fórmula de la función del valor del estado.

Luego, tenemos la **función de calidad de la combinación estado-acción**, y representa la recompensa total que podemos obtener empezando desde ese estado, tomando esa acción. También depende del **reglamento.**

$$Q^{\pi}(s,a) = \mathbb{E}\big[R_t \quad | \quad s_t = s, a_t = a\big]$$

Fig. 6. Fórmula de la función de calidad estado-acción.

Tras esto, llegamos a la ecuación de **Bellman**. Uno de los principales problemas del Q-Learning es su dependencia en esta ecuación. La nueva Q, dependiente del reglamento viene de la recompensa inmediata y del producto del factor de descuento y la función de calidad del nuevo estado.

$$Q^{\pi}(s_t, a_t) = r(s_t, a_t) + \gamma V^{\pi}(s_{t+1})$$

El problema de dejar esta ecuación así sería que no tiene por qué ser la más óptima. Podemos mejorarla si usamos la recompensa máxima esperada a través de relacionar la función de calidad del estado con la combinación estado-acción.

$$Q(s,a) = r + \gamma \max_{a'} Q(s', a')$$

Un episodio del algoritmo termina cuando el siguiente estado **s+1** es un estado final, aunque este no necesita finalizar un episodio para aprender.

Ahora que hemos explicado cómo funciona el algoritmo, ¿De donde viene?

Como hemos dicho anteriormente, Q-Learning es una técnica de **aprendizaje por refuerzo.** En concreto es una implementación de un **Método de Diferencia Temporal**.[4]

En estos, antes de que comience el aprendizaje, se inicializa **Q** a un valor aleatorio constante. Después de un tiempo **t**, el agente toma una acción **a** observando una recompensa **r** y se toma un nuevo estado $\mathbf{s_{t+1}}$ (que depende de un estado $\mathbf{s_t}$ y de la decisión **a**), y **Q** se actualiza.

Este toma idea de la programación dinámica y de la Técnica Montecarlo. Por un lado, las decisiones se basan en las estimaciones previas, por lo que no es necesario acabar un problema para aprender, por otro lado, no requieren un modelo exacto del entorno, y basta explorar para modificar su comportamiento.

Esta rama del aprendizaje por refuerzo es la más novedosa, y la que tiene actualmente una línea de investigación más fuerte. Se basa en un entorno de rejilla, en el que el agente puede tomar 4 distintas acciones: Arriba, abajo,

derecha e izquierda. De aquí sale el Q-Learning. Su objetivo es aprender una serie de normas para que el agente en cuestión sepa qué acciones tomar en las distintas circunstancias.

I. Metodología

Hemos Introducido las casillas del mapa en python mediante un array de 2 dimensiones de enteros. En este Array las casillas de llanura tenían el valor de 1 paso, las casillas de bosque tenían un valor de 2 pasos, las de montaña valían 3 pasos y las de agua valían 0.
El sistema busca llegar desde el punto A al punto B sumando el número de pasos de cada casilla por la que pasa, y minimizándolo todo lo que pueda.

El mapa de ejemplo del proyecto sería en nuestro código:

```
[[1, 1, 1, 1, 1, 1, 1, 0, 0, 1],
[1, 1, 1, 1, 2, 2, 2, 0, 0, 1],
[1, 1, 1, 2, 2, 4, 2, 2, 1, 1],
[1, 1, 1, 2, 4, 4, 4, 2, 1, 1],
[1, 1, 1, 2, 2, 4, 0, 0, 0, 0],
[1, 1, 1, 1, 2, 2, 0, 0, 0, 0]].
```

El mapa, también guarda el tamaño que tiene en el eje Y y el eje X. y devuelve el valor de la celda en la posición (Y,X) que le pases.

Simplemente modificando dicho array de 2 dimensiones podemos diseñar cualquier mapa que queramos que el algoritmo resuelva.

También tenemos definida la función "**tipoCelda"** que devuelve el valor de una celda a partir de su posición X e Y.

La posición inicial es guardada en las variables **posX** y **posY**. Y la posición final en **endX** y **endY**.

También tenemos definidas la función "**step(self, action)**", que recibe un número aleatorio entre 0 y 3, correspondiendo cada número a una dirección. 0 es moverse hacia la izquierda, 1 es moverse hacia la derecha, 2 es moverse hacia arriba y 3 es moverse hacia abajo. Cuando la función recibe un número, antes comprueba si es posible moverse en esa dirección.

Por ejemplo, si quiere moverse hacia arriba, debe comprobar que en la celda de arriba no hay una casilla de agua, marcadas con el valor "0" y además debe comprobar que su posición en el eje Y no es 0, ya que en el mapa formado por el doble array, la posición 0 en el eje Y es el límite superior del mapa.
Para todas las direcciones se comprueba que no haya una casilla de agua en esa dirección ni que se pueda salir del mapa.

En pseudocódigo, la comprobación de Moverse hacia arriba sería la siguiente:

Si PosEjeYActual > 0 y además ValorCelda(posEjeYActual -1, posEjeXActual) no es 0

La función **randomAction(self)** Es usada para obtener un número aleatorio entre 0 y 3. Ese número obtenido es el que se pasará a la función **step(self, action)** para obtener la acción de cada turno.

La función "**render(self)**" se dedica a imprimir el mapa actual por consola.
Recorre completamente con dos bucles for el mapa actual, pasando por cada casilla e imprimiendo por consola su valor. Si pasa por la casilla donde se encuentra ahora mismo el ratón, imprimirá en su lugar una "**R**".
Si pasa por la posición de destino, imprimirá una "**Q**".

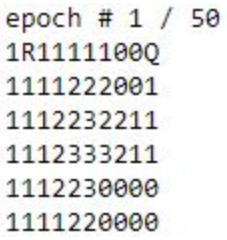

Fig. 7. Output de la consola mostrando el método **render**. Se aprecia la posición final **Q** en (9,0) y la posición actual del ratón **R** en (1,0).

Con todo esto, podemos centrarnos ahora en la programación del algoritmo en sí.

Empezamos creando la **Q-table**, que contiene todos los **Q-valores** para todas las parejas **estado-acción**. Es necesaria para cuando el algoritmo decida ir por la mayor recompensa inmediata en lugar de tomar una acción aleatoria.

Luego, los 4 parámetros principales del algoritmo. **Epoch, Gamma, Epsilon, Decay**.

Epoch se trata del número de iteraciones en las que vamos a entrenar el algoritmo. Una iteración consiste en el recorrido del agente desde el punto inicial hasta el final.

Gamma es el factor de descuento. Se trata de un número entre 0 y 1 que determina la importancia de recompensas futuras. Un valor bajo hará que se centre en las recompensas actuales mientras que un valor alto hará que luche por mejores recompensas a largo plazo.

Epsilon se trata de un número. Para cada paso el algoritmo generará un número aleatorio, si es menor que Epsilon, entonces el siguiente paso será aleatorio, si es mayor que Epsilon, el siguiente paso será el que tenga una mayor recompensa inmediata.

Decay es un valor que tras cada iteración reduce el valor de **Epsilon**. Tras cada iteración el nuevo valor de **Epsilon** es la multiplicación del **decay** por **Epsilon** y el resultado se resta a **Epsilon**. Reduciendo **Epsilon** tras cada iteración conseguimos que conforme se vaya entrenando la **Q-table**, se dejen de usar acciones aleatorias.

Ahora nos centraremos en que hace el algoritmo para cada iteración.

El algoritmo se repite un número de veces igual al valor de **Epoch**, y hace movimientos hasta que el agente llega a la posición final.

Cada uno de estos movimientos comienza imprimiendo por pantalla el **Epoch** actual y el mapa.

Seguidamente cuenta el número de pasos de la casilla actual y lo suma a la variable **steps**.

Tras esto se calcula un número aleatorio para compararlo con **Epsilon** y ver si se toma una acción aleatoria o se sigue la **Q-table**.

Una vez obtenido el resultado, se realiza la acción, se actualiza la **Q-table**, se avanza a la siguiente casilla y se actualiza **Epsilon**.

Si no se ha alcanzado la casilla final se realiza otro movimiento, empezando de nuevo. Si se ha alcanzado, se muestra por pantalla el número de pasos que ha tomado llegar hasta el final y a continuación se empieza la siguiente iteración, con la **Q-table** algo más entrenada.

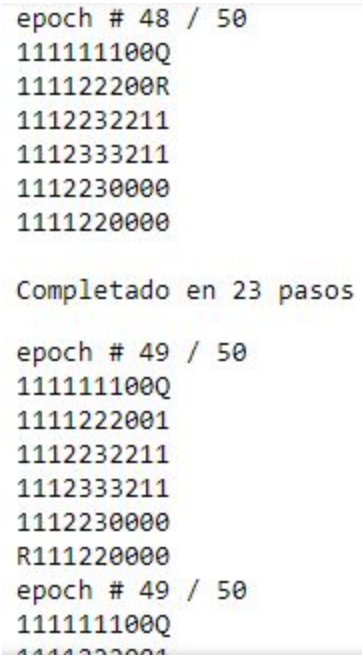

Fig. 8. Output de la consola terminando una iteración, revelando el número de pasos que ha tomado completarse y comenzando una nueva. (Esto no es la mejora gráfica)

También hemos añadido al proyecto una mejora que consiste en una interfaz gráfica usando la librería de python **tkinter**[8][9][10]. Esta interfaz muestra un dibujo del mapa que le pasemos y los movimientos del agente en cada paso hasta llegar a su destino.

II. Resultados

En este apartado mostraremos algunos experimentos y sus resultados.

Recordamos que las casillas de llanura equivalen a 1 paso, las de bosque a 2 pasos y las de montaña a 3 pasos.

Epoch	**Tabla de resultados para mapa de ejemplo**		
	Decay	***Epsilon***	***Gamma***
	0,1	***0,08***	***0,1***
1	196		
2	285		
3	100		
4	86		
5	220		
6	158		
7	298		
8	60		
9	63		
10	54		
11	39		
12	41		
13	59		
14	31		
15	82		
16	40		
17	69		
18	**20**		
19	32		
20	26		
21	54		
22	25		
23	49		
24	34		
25	56		
26	37		
27	25		
28	34		
29	31		
30	26		
31	27		
32	25		
33	28		
34	23		
35	25		
36	**20**		
37	26		
38	27		
39	26		
40	27		
41	23		
42	24		
43	22		
44	22		
45	23		

46	22
47	23
48	23
49	23
50	24
20	**Camino mínimo hallado**

Tabla 1

Epoch	**Tabla de resultados para mapa de ejemplo**		
	Decay	***Epsilon***	***Gamma***
	0,1	***0,08***	***0,8***
1	2239		
2	348		
3	317		
4	877		
5	35		
6	85		
7	62		
8	60		
9	62		
10	28		
11	78		
12	53		
13	81		
14	87		
15	31		
16	60		
17	30		
18	30		
19	36		
20	38		
21	25		
22	28		
23	34		
24	27		
25	32		
26	26		
27	27		
28	**24**		
29	27		
30	**24**		
31	**24**		
32	**24**		
33	**24**		
34	**24**		
35	**24**		
36	**24**		
37	**24**		
38	26		
39	**24**		
40	**24**		
41	**24**		
42	**24**		
43	**24**		
44	**24**		
45	**24**		
46	**24**		
47	**24**		
48	**24**		
49	**24**		
50	**24**		
24	**Camino mínimo hallado**		

Tabla 2

Epoch	**Tabla de resultados para mapa de ejemplo**		
	Decay	***Epsilon***	***Gamma***
	0,9	***0,08***	***0,1***
1	361		
2	218		
3	38		
4	145		
5	146		
6	156		
7	77		
8	53		
9	92		
10	83		
11	79		

12	67
13	35
14	40
15	57
16	48
17	66
18	36
19	33
20	38
21	23
22	44
23	33
24	32
25	34
26	23
27	**22**
28	28
29	23
30	48
31	31
32	**22**
33	48
34	28
35	24
36	43
37	25
38	31
39	24
40	26
41	25
42	**22**
43	28
44	23
45	**22**
46	24
47	24
48	25
49	26
50	24

22	**Camino mínimo hallado**

Tabla 3

Epoch	**Tabla de resultados para mapa de ejemplo**		
	Decay	***Epsilon***	***Gamma***
	0,1	***0,78***	***0,1***
1	731		
2	150		
3	915		
4	365		
5	78		
6	117		
7	57		
8	98		
9	65		
10	60		
11	89		
12	35		
13	52		
14	85		
15	41		
16	55		
17	78		
18	48		
19	69		
20	46		
21	37		
22	27		
23	42		
24	55		
25	29		
26	35		
27	29		
28	27		
29	32		
30	26		
31	25		
32	**20**		
33	29		

34	34
35	30
36	22
37	25
38	**20**
39	32
40	23
41	25
42	27
43	33
44	24
45	26
46	24
47	21
48	25
49	22
50	23
20	**Camino mínimo hallado**

Tabla 4

Intento	**Tabla de resultados para mapa de ejemplo tras Epoch=50**			
	Decay	***Epsilon***	***Gamma***	***Camino mínimo encontrado***
1	***0,1***	***0,08***	***0,1***	***20***
2	***0,1***	***0,08***	***0,1***	***20***
3	***0,1***	***0,08***	***0,1***	***20***
4	***0,1***	***0,08***	***0,1***	***21***
5	***0,1***	***0,08***	***0,1***	***20***
1	***0,1***	***0,08***	***0,8***	***23***
2	***0,1***	***0,08***	***0,8***	***23***
3	***0,1***	***0,08***	***0,8***	***26***
4	***0,1***	***0,08***	***0,8***	***23***
5	***0,1***	***0,08***	***0,8***	***22***
1	***0,9***	***0,08***	***0,1***	***22***
2	***0,9***	***0,08***	***0,1***	***23***
3	***0,9***	***0,08***	***0,1***	***20***
4	***0,9***	***0,08***	***0,1***	***22***
5	***0,9***	***0,08***	***0,1***	***21***
1	***0,1***	***0,78***	***0,1***	***20***
2	***0,1***	***0,78***	***0,1***	***21***
3	***0,1***	***0,78***	***0,1***	***20***
4	***0,1***	***0,78***	***0,1***	***20***
5	***0,1***	***0,78***	***0,1***	***20***

Tabla 5

Epoch	**Tabla de resultados para mapa número 2**		
	Decay	***Epsilon***	***Gamma***
	0,1	***0,08***	***0,1***
1	11		
2	15		
3	5		
4	**4**		
5	16		
6	**4**		
7	**4**		
8	**4**		
9	**4**		
10	**4**		
11	7		
12	**4**		
13	**4**		
14	**4**		
15	**4**		
4	**Camino mínimo hallado**		

Tabla 6

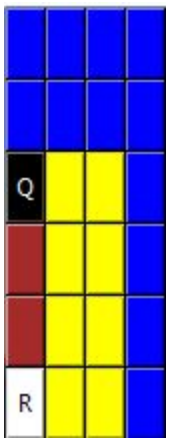

Fig. 9. Mapa número 2, mostrado usando la mejora gráfica, este mapa es usado en la tabla 6, el color rojo son montañas, el azul agua y el amarillo llanuras

Epoch	Tabla de resultados para mapa número 3		
	Decay	*Epsilon*	*Gamma*
	0,1	*0,08*	*0,1*
1	-		

Tabla 7

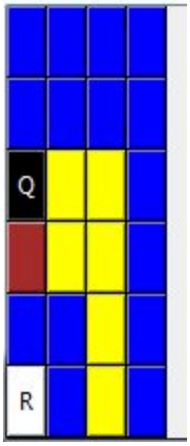

Fig. 10. Mapa número 3, mostrado usando la mejora gráfica, usado en la tabla 7, el color rojo son montañas, el azul agua y el amarillo llanuras

III. Conclusiones

En este apartado vamos a comenzar comentando y explicando los experimentos mostrados en las tablas del apartado anterior.

Hemos comenzado con cuatro casos que hacían uso del mapa del ejemplo y 50 iteraciones. En la primera tabla usamos los valores por defecto que venían en el código del documento que se nos recomendó[5].

En la segunda tabla, pusimos un valor alto del parámetro **Gamma**.

En la tercera tabla, pusimos un valor alto de la variable **decay** y en la cuarta tabla hicimos lo mismo con **Epsilon**.

Con los valores por defecto podemos observar que llegamos al camino mínimo posible del mapa de ejemplo, en este caso, 20 pasos (teniendo en cuenta que caminar por llanura es 1 paso, por bosque son 2 pasos y por montaña 3 pasos). Si nos fijamos en la tabla 5, también podemos apreciar que se alcanza el camino mínimo de forma consistente, así que parecen buenos valores para este mapa.

Cuando subimos mucho el **Gamma** se puede apreciar que las primeras iteraciones suelen tener números muy grandes, y que cuando lleva algo más de la mitad de las iteraciones, suelen estabilizarse y centrarse siempre en el mismo camino, que da la casualidad de que pocas veces es el camino más óptimo posible, como podemos ver en la tabla 5.

Cuando aumentamos el **decay** el sistema deja de tomar decisiones aleatorias antes, dejando de explorar alternativas. Podemos apreciar en las tablas 3 y 5 como el número de pasos por iteración se va reduciendo, quedando cerca del valor óptimo, pero llegando en muy pocas ocasiones.

Si aumentamos los valores de **Epsilon**, le damos más posibilidades de explorar soluciones al algoritmos. Permitiendo que entre alguna de ellas esté la más óptima, suele encontrarla con facilidad, como apreciamos en las tablas 4 y 5.

Tras estas pruebas decidimos probar otros mapas, y usando los mismos valores que en la tabla 1, probamos a resolver el mapa de la figura 9. El algoritmo era capaz de obtener el camino mínimo con facilidad y consistencia en menos de 15 iteraciones.

También hicimos la prueba de dejar el algoritmo en un mapa que no pudiese resolver, por ejemplo rodeando el punto inicial de agua, y como esperábamos, no se obtuvo ningún resultado.

Tras estos experimentos podemos deducir que tanto aumentar el **Epsilon** cómo usar los

valores por defecto son buenas opciones para encontrar la solución más óptima, sin embargo, aumentar mucho el **Gamma** y el **decay** no eran óptimos para ello, con la desventaja de que aumentando el **Gamma**, el algoritmo iba muy lento pues las primeras iteraciones eran muy largas.

Estas fueron nuestras conclusiones tras los experimentos. Como conclusiones del proyecto en general podemos destacar que este proyecto nos ayudó bastante a entender python, entender este algoritmo concreto y a trabajar con interfaces gráficas al añadir una como mejora. Cosas que no habíamos aprendido en toda la carrera.

Referencias

[1] Página web del curso IA de Ingeniería del Software. https://www.cs.us.es/cursos/iais.

[2] https://es.wikipedia.org/wiki/Cadena_de_M%C3%A1rkov

[3] https://es.wikipedia.org/wiki/Aprendizaje_por_refuerzo#M%C3%A9todos_de_diferencias_temporales

[4] https://es.wikipedia.org/wiki/Q-learning

[5] https://medium.com/datadriveninvestor/math-of-q-learning-python-code-5dcbdc49b6f6

[6] Práctica 3 web de la asignatura de inteligencia artificial de la US https://www.cs.us.es/cursos/iais-2018/practicas/Pr%C3%A1ctica_03.zip

[7] https://www.cs.us.es/cursos/iais-2018/trabajos/AprendizajePorRefuerzo.pdf

[8] https://www.daniweb.com/programming/software-development/threads/244488/root-update

[9] https://www.youtube.com/watch?v=5d1CfnYT-KM

[10] https://docs.python.org/3/library/tk.html