ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Graduado en Ingeniería de Software

Comparación de algoritmos de aprendizaje por refuerzo basados en Q-Learning

Comparison of Q-Learning-based reinforcement learning algorithms

Realizado por

Romolo Rosario Caponera De Cobellis

Tutorizado por

José Luis Pérez de la Cruz

Departamento

Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, JUNIO DE 2021

Fecha defensa: Julio de 2021

Resumen

El presente Trabajo de Fin de Grado se centra en el Aprendizaje por Refuerzo, y más concretamente en el algoritmo QLearning[1], comparando tres de sus variantes más populares (Épsilon-greedy [2], SoftMax [3] y Upper Confidence Bound [4]), en el entorno FrozenLake [5] ofertado por el framework OpenAI Gym [6].
La finalidad del proyecto no es únicamente la de mostrar las diferencias, tanto en implementación como en rendimiento, que se puedan evidenciar entre los algoritmos, sino ofrecer una aplicación que permita al usuario final inspeccionar los procesos de entrenamiento y resolución del problema, tomando sus propias mediciones, e incluso experimentar variando los parámetros de los algoritmos, mediante una interfaz intuitiva y visual.

Se ha desarrollado un software que pretende suplir estas necesidades, ofreciendo la posibilidad de observar el proceso de entrenamiento de manera visual e intuitiva, y la posibilidad de ver, a través de varios gráficos, métricas sobre el entrenamiento que permitirán discernir la efectividad y calidad del mismo. La implementación del software se ha realizado siguiendo *buenas prácticas* y patrones de diseño que permiten a cualquier usuario, con los conocimientos técnicos necesarios, añadir sus propias variantes del algoritmo con cierta flexibilidad, y compararlas con las que ya están incluidas en la aplicación.

El software se ha implementado en el lenguaje de programación Python [7], utilizando el Entorno de Desarrollo Integrado (IDE de ahora en adelante) Pycharms [8], y las plataformas OpenAI Gym para la obtención del entorno, y PyQt5 [9] para la Interfaz Gráfica de Usuario.

Todo el contenido de este proyecto se ha publicado en la plataforma GitHub [10], que permite el acceso y explotación colaborativos al público.

Palabras clave: QLearning, Aprendizaje por Refuerzo, Inteligencia Artificial, Python

Abstract

This work focuses on Reinforcement Learning, and more specifically in the QLearning [1] algorithm, by comparing three amongst its most popular variants (e.g. Epsilon-greedy [2], SoftMax [3], and Upper Confidence Bound [4]), in the FrozenLake [5] environment provided by OpenAI Gym [6].This project does not only aim to illustrate the differences between the algorithms, both as implementation and performance, but also to offer an end-user application that allows the inspection of the different algorithms during the training and resolution tasks in a much more Intuitive and visual interface. Moreover, it offers the chance of "playing" around with algorithms' hyperparameters, to see and really experience how they impact agent's behavior, and why they are so important.

A computer program that aims to fulfill these needs has been developed, offering a chance to observe the training process in a visual, intuitive way, and to see, through multiple graphs, training metrics that will make clear the effectiveness and quality of it. The software has been implemented according to Good Practices and design patterns that allow any user, as long as he hold programming knowledge, to implement and add to the application his own variants of the algorithm, thus comparing them to the current ones.

This software has been implemented in Python [7] programming language, using the PyCharms Integrated Development Environment (IDE) [8], the OpenAI Gym framwork for the environments, and the PyQt5 [9] framework for the Graphical User Interface development.

This project is also available to download on GitHub [10] platform, allowing collaborative access and exploitation worldwide.

Keywords: QLearning, Reinforcement Learning, Artificial Intelligence, Python

Índice

Índice **1**

Introducción **3**

1.1 Motivación **3**

1.2 Objetivos **4**

1.3 Estructura de la memoria **5**

Nociones básicas sobre Inteligencia Artificial **7**

2.1 Contexto histórico de la Inteligencia Artificial **7**

2.2 Tipos de Inteligencia Artificial **9**

2.2.1 Según su funcionamiento 9

2.2.2 Según su capacidad 11

2.3 Machine Learning **14**

2.3.1 Tipos de Machine Learning 15

Frozen Lake y Proceso de Decisión de Markov **21**

3.1 El entorno Frozen Lake **21**

3.2 Proceso de Decisión de Markov **23**

Q-Learning **25**

4.1 Aprendizaje por Refuerzo **25**

4.2 Q-Learning **26**

4.2.1. Tasa de aprendizaje (α) 28

4.2.2. Tasa de descuento (γ) 31

Políticas implementadas **35**

5.1 Explotación vs exploración **35**

5.2 Ɛ-greedy **36**

5.3 SoftMax **37**

5.4 Upper Confidence Bound **39**

Software Desarrollado ... **43**
6.1 Diagrama de clases ... **43**
6.2 Patrones de diseño ... **46**
6.2.1 Modelo-Vista-Controlador ... 46
6.2.2 Envoltorio ... 47
6.2.3 Estrategia ... 48
6.2.4 Delegación ... 49
6.2.5 Multithreading ... 49
6.2.6 Callbacks ... 50
6.3 Interfaz de usuario ... **51**
6.3.1 Ventana principal ... 51
6.3.2 Menú superior ... 54
6.3.3 Evolución del entrenamiento ... 55
6.3.4 Banco de pruebas ... 56
Resultados obtenidos y conclusiones ... **61**
Conclusiones ... **65**
Referencias ... **68**
Manual de Instalación ... **73**
Requerimientos: ... **73**
Instalación y ejecución ... **73**

1

Introducción

1.1 Motivación

Nos encontramos en una época en la que la Inteligencia Artificial ha conquistado por completo nuestras vidas, estando cada vez más presente en todos nuestros dispositivos y todos los ámbitos de nuestra sociedad.

Cada día que pasa son mayores los avances e innovaciones en este ámbito, así como la popularidad del mismo: son pocos los que aún no han oído hablar, aun sin llegar a entenderlo, de conceptos como la Inteligencia Artificial, el Deep Learning o las redes neuronales.

Sin embargo, el conocimiento real que tiene el usuario medio sobre estos conceptos puede llegar a ser mínimo, tratándose además de un tema que puede alcanzar una gran complejidad: hay muchos tipos de Inteligencia Artificial, muchos algoritmos para cada tipo, y muchas variantes de cada algoritmo.

No solo puede ser complicado comprender el funcionamiento de los mismos, sino que además, con frecuencia, resulta difícil discernir cuál es mejor que otro

en la resolución de un determinado problema. No es inusual que cierto algoritmo sepa resolver perfectamente un problema concreto, y presente un rendimiento muy inferior en otro.

Es por ello que el presente proyecto está orientado a desarrollar un software que permita visualizar de manera sencilla y didáctica el comportamiento de algunos algoritmos de aprendizaje, e integrar en el mismo herramientas para la comparativa de rendimiento entre ellas.

Más concretamente, se ha escogido el campo del Aprendizaje por Refuerzo, el algoritmo Q-Learning y tres de las variantes más importantes del mismo.

1.2 Objetivos

La principal finalidad de este proyecto es facilitar la comprensión y el conocimiento del usuario respecto al aprendizaje por refuerzo en general, y a Q-Learning y sus variantes en particular.

Los principales objetivos del proyecto son:

1. Ofrecer una comparación significativa del rendimiento y la eficiencia de las variantes seleccionadas en el entorno FrozenLake, determinando qué tipo de algoritmo funciona mejor en esta clase de entornos.
2. Desarrollar un sistema que permita a aquellos usuarios sin amplios conocimientos en el ámbito de la inteligencia artificial o el aprendizaje por refuerzo, visualizar de una manera intuitiva el funcionamiento de este algoritmo, y de cada una de sus variantes.

El sistema permitirá al usuario modificar los hiperparámetros de las variantes (que, como se detallará más adelante, son parámetros que permiten regular la eficiencia del entrenamiento) para visualizar cómo afectan al rendimiento final del algoritmo. Mostrará también gráficos y métricas en tiempo real sobre el rendimiento y eficiencia de las variantes, para que el propio usuario experimente en primera persona qué variante funciona mejor y traiga sus propias conclusiones.

Además, para usuarios más expertos, el sistema permitirá la implementación de nuevas variantes del algoritmo que se integrarán completamente con la aplicación, dando así la posibilidad de compararlas de manera sencilla con las ya implementadas.

1.3 Estructura de la memoria

Se ha optado por estructurar esta memoria según la especificidad de los temas, partiendo de aquellos más básicos hasta los matices más importantes.

Se inicia el trabajo con una contextualización de la Inteligencia Artificial, tanto en términos históricos como conceptuales, para poco a poco ir adentrándose cada vez más en el aprendizaje por refuerzo, en el Q-Learning y sus variantes.

Tras el resumen del fundamento teórico, se pasa a la descripción del software diseñado, mostrando la estructura interna del mismo y la motivación de las decisiones de diseño tomadas. Una vez definido el diseño interno del sistema, se describen sus funcionalidades a nivel de usuario, con la ayuda de capturas de pantalla.

Finalmente, se presentan los resultados obtenidos de las mediciones, aclarando qué algoritmo se corona como el mejor en este ámbito, y terminando con conclusiones finales.

2

Nociones básicas sobre Inteligencia Artificial

Antes de comenzar a ilustrar los temas que se tratarán en este trabajo, se presenta el marco contextual en el que nos movemos, aclarando para ello unos conceptos básicos que resultan esenciales para la comprensión de esta memoria.

2.1 Contexto histórico de la Inteligencia Artificial

El de la Inteligencia Artificial (IA de ahora en adelante) es sin duda un concepto fundamental para el desarrollo de este tema. Al considerar la Inteligencia Artificial, el público general pensará, probablemente, en algo moderno, novedoso y en auge, quizás relacionado con la robótica y, por qué no, visualizará en su mente alguno de los robots humanoides con los que las películas futuristas tanto nos han deleitado.

Sin embargo, esta idea no podría ser más errónea: aunque la IA se encuentre en auge en este momento, es en realidad un campo de estudio mucho más antiguo que la propia informática.
El concepto de razonamiento artificial se remonta al siglo XIV, si bien no es hasta mediados del siglo XX[11] que se obtiene la matemática y la maquinaria suficientemente potente como para materializar esos conceptos.
Es en los años 50 de este siglo cuando el gran matemático Alan Turing, considerado uno de los padres de la informática y de la IA, se pregunta, en uno de sus artículos más famosos [12], si las máquinas podrían llegar algún día a pensar como los humanos.
Es aquí donde nace el famoso Test De Turing, donde el matemático afirma que podremos considerar que **una máquina es capaz de pensar, y por tanto inteligente, si es capaz de engañar a un humano para hacerle creer que ella es también humana**.
Con el paso de los años, varios programas de ordenador han sido capaces de superar el test de Turing, como ELIZA [13], PARRY [14], o A.L.I.C.E. [15], habiendo sido está última galardonada con el premio Loebner como el bot conversacional más humano hasta en 3 ediciones distintas.
Sin embargo, la definición propuesta por Turing no convenció a todos. Algunos matemáticos y filósofos pensaban que el hecho de que una máquina pudiera hablar e interactuar con una persona de manera verosímil, no implicaba que pensara o que entendiera lo que estaba haciendo.
Esta idea culminó en 1980 cuando el filósofo americano John Searle publicó en un artículo el conocido como *argumento de la habitación china* [16], en el que justificaba que el hecho de que una persona que no hable chino, a través del uso de diccionarios y manuales, pueda interactuar con un hablante chino de manera natural y verosímil, no implica que esté entendiendo ni una sola palabra de lo que escribe: se limita únicamente a seguir las instrucciones del manual.
Durante la historia, la IA ha recibido varias definiciones para tratar de concretar su interpretación unívoca. Hoy en día, según Oxford Languages, la IA se define como un *"Programa de computación diseñado para realizar determinadas operaciones que se consideran propias de la inteligencia humana, como el*

autoaprendizaje ". A pesar de ello, el test de Turing sigue siendo un punto de referencia cuando se habla de IA.
En la historia más reciente, una de las mayores hazañas de la IA fue AlphaGO[17], un programa que fue capaz de derrotar al tres veces campeón europeo de Go[18], el que se considera el juego de mesa más complicado del mundo.

2.2 Tipos de Inteligencia Artificial

A la hora de clasificar la Inteligencia Artificial, generalmente, encontramos dos posibles clasificaciones, que dependerán de los atributos de la misma en los que nos centremos: su funcionamiento o su capacidad.

2.2.1 Según su funcionamiento

Al centrarnos en el funcionamiento de la IA, nos fijaremos únicamente en el diseño de esta, generando una clasificación más "teórica" que la alternativa clasificación por capacidad.
Dividiendo las IAs según este criterio, encontraremos los siguientes cuatro tipos[19]:

1. **IA Reactiva**. Este modelo de IA no tiene ningún tipo de memoria, sino que se dedica únicamente a responder a los estímulos que recibe, es decir, producir una salida para una entrada, pero sin que importe ninguna de las entradas o salidas que haya podido recibir anteriormente, ni que afecte a las salidas sucesivas. Dentro de este tipo podemos encontrar algoritmos muy importantes como Deep Blue[20], la máquina que consiguió derrotar a Kasparov al ajedrez, o incluso algunos modelos de redes neuronales como el de AlphaGo[17].

   Si algo tienen en común estas máquinas es que no tienen en cuenta los movimientos pasados ni futuros, simplemente captan la posición actual del tablero y deciden, según el algoritmo que tengan implementado (o aprendizaje, en el caso de la red neuronal), qué movimiento será más eficiente.

   A pesar de ser el tipo más básico de IA, se trata de una tecnología muy potente que permitirá aún muchos avances en el futuro.

2. **IA de Memoria Limitada.** Si bien las capacidades de las IAs reactivas pueden resultar impresionantes, hay tareas que este tipo de tecnología todavía no puede llegar a realizar. Si lo pensamos detenidamente, todo lo que implique movimiento requerirá de un mínimo de memoria. Pensando, por ejemplo, en un coche inteligente, veremos como a la hora de conducir no podremos comportarnos de igual manera si los coches de nuestro alrededor están circulando a 120km/h por la autovía, o están estacionados en el aparcamiento de un supermercado. La velocidad de los vehículos no es algo que se pueda intuir basándose en una percepción instantánea (una fotografía). Necesitamos cierta noción del tiempo, del pasado, que nos permita medir la velocidad de los vehículos y comportarnos en consecuencia.

   Esto es precisamente lo que hace una IA de Memoria Limitada: retiene ciertos datos de instantes anteriores para poder realizar mejor su tarea, aunque no utiliza esa información para aprender ni mejorar su funcionamiento.

   La IA tiene memoria, pero está *limitada* a obtener unos pocos datos de instantes anteriores para mejorar la toma de decisiones actual, y luego "olvidarlos".

3. **Teoría de la mente**. Viendo cómo la IA Reactiva no tiene un conocimiento real del mundo, sino tan solo una percepción del instante actual, y cómo la IA de Memoria Limitada sí que tiene cierto conocimiento respecto al mundo que le rodea, el paso siguiente es tener consciencia no sólo del entorno sino también del resto de entidades o agentes que lo habitan.

   En el campo de la psicología, se llama "teoría de la mente" [21] a la percepción, por parte de humanos o animales, de que las demás entidades que los rodean pueden tener pensamientos y sentimientos que modifiquen su conducta, es decir, ser consciente de que no se es el único ser "inteligente" del entorno.

   En la evolución del ser humano, esta percepción ha jugado un rol fundamental, ayudándonos a formar comunidades o sociedades que nos han permitido evolucionar tal y como lo hemos hecho.

Para que distintos individuos puedan colaborar para alcanzar objetivos comunes, conocer y entender los pensamientos de los demás, cumple un papel fundamental. Y esto es algo que se aplica perfectamente a las máquinas también: si nuestra expectativa es que, como en las películas de ciencia ficción, las IAs estén a nuestro alrededor y colaboren con nosotros y entre ellas, es imprescindible que entiendan nuestros pensamientos y sentimientos. Este nuevo nivel de IA es el que marca la línea divisoria entre lo que hacemos hoy y lo que podremos hacer en el futuro. Actualmente aún no se ha conseguido desarrollar ninguna IA que cumpla con la teoría de la mente.

4. **IA autoconsciente.** Podemos ver este último nivel de IA como una extensión de la IA de nivel 3. Sin embargo, aquí se busca algo más que la simple comprensión de los sentimientos de agentes externos. Se busca la autoconsciencia: conocerse y conocer nuestros propios sentimientos.

   Conceptualmente, hay una diferencia radical entre las afirmaciones "Quiero esto" y "Sé que quiero esto": la primera muestra un simple impulso que cualquier animal podría tener, pero la segunda demuestra autoconocimiento y consciencia: "quiero esto, y soy consciente de ello".

   Cuando somos capaces de conocernos a nosotros mismos, también somos capaces de predecir nuestros sentimientos, y los de los demás. Cuando vemos a alguien comer muy deprisa, podemos inferir que tiene hambre, porque nosotros mismos, cuando tenemos hambre, comemos muy deprisa.

   Este último salto evolutivo en la IA es lo que nos permitiría tener máquinas tan inteligentes, o más, que los propios seres humanos, ubicándonos en un entorno que, de nuevo, parece mucho más de ciencia ficción que una situación real.

   Sin embargo, aún quedan décadas, si no siglos, para que podamos ver una IA de este tipo en funcionamiento y accesible al público.

2.2.2 Según su capacidad

La clasificación de las IAs según su capacidad resulta mucho más sencilla y pragmática que la anterior. Las clasificaremos, según el nivel de "inteligencia" que demuestran: IA débil, fuerte y SuperIA[22].

1. **Inteligencia Artificial Débil.** Actualmente, este es el único tipo de Inteligencia Artificial que hemos sido capaces de desarrollar. Se define como débil a la IA que es capaz de resolver de manera eficiente la tarea para la que ha sido entrenada, pero que no es capaz de realizar otras tareas. Podemos pensar en IAs de reconocimiento y clasificación de imágenes que han sido diseñadas y entrenadas con ese propósito específico, y que lo hacen sorprendentemente bien. Sin embargo, si a esta IA le "pidiéramos" que conversara con nosotros, que resolviera un puzle, o incluso una sencilla operación matemática, ésta sería incapaz de hacerlo. Por otro lado, cabe destacar que el nivel de precisión y exactitud en la realización de una tarea por parte de una IA Débil es generalmente muy elevado, incluso mejor que el que alcanzaría un humano.

   Concretamente, resulta interesante mencionar el caso de AutoML[23], una red neuronal diseñada con el propósito de generar sus propias arquitecturas de redes neuronales destinadas a la detección y reconocimiento de imágenes.

   El rendimiento fue tal que en 2017 NASNet, una de las redes neuronales generadas por AutoML, obtuvo mejor puntuación que cualquier otra red hasta la fecha en el dataset de clasificación de imágenes ImageNet y el de reconocimiento de objetos COCO [24], [25], [26], [27].

   En este subconjunto, la mayor parte de IAs son de Memoria Limitada (tipo 2 según la clasificación anterior).

2. **Inteligencia Artificial Fuerte.** Como cabe suponer, si la IA Débil es aquella que solamente es capaz de realizar una tarea, la IA general o Fuerte será aquella capaz de resolver todo tipo de problemas.

   Más concretamente, la IA fuerte es aquella que puede comportarse de manera indistinguible a un humano en cualquier campo, desde reconocimiento de imágenes u objetos, procesamiento del lenguaje natural, hasta la resolución de problemas lógicos. Esta IA es, a todos los efectos, igual de capaz que cualquier humano.

   Aún no se ha desarrollado ninguna IA de este tipo, ni está previsto que se desarrolle en un futuro cercano, aunque hay algunos proyectos realmente prometedores. Concretamente un ejemplo que ha suscitado mucho

interés en la comunidad científica, es el de GPT-3[28], una red neuronal orientada al Procesamiento del Lenguaje Natural. Más específicamente, esta red está entrenada para predecir cuál será la siguiente palabra dado un texto de entrada: puede usarse como bot conversacional, es capaz de resumir o traducir textos, de traducir fragmentos de código de un lenguaje a otro, o incluso generar código informático simplemente especificando, en lenguaje natural, qué es lo que se quiere que haga [29][30]. La red es tan potente y está tan bien entrenada, que puede parecer que sea capaz de resolver varias tareas y de interactuar con un humano de forma bastante natural. Sin embargo, estrictamente hablando, la máquina está realizando una única tarea, la única para la que ha sido programada: predecir cuál será la siguiente palabra.

Basados en la tecnología de GPT-3, también hay muchos otros proyectos de gran interés, de entre los que destaca CLIP[31], que une el procesamiento de lenguaje natural con la generación y procesamiento de imágenes. Además Fujitsu construyó en 2011 el superordenador K[32], el primero en romper la barrera de los 10 PetaFLOPS[1], y que en 2013 fue capaz de simular un segundo de actividad cerebral, aunque esta simulación le llevó 40 minutos[33]. Este experimento nos muestra cómo, si bien la simulación de la actividad cerebral humana parece algo prometedor, aún está muy lejos de ser viable en términos de tiempo y potencia de cálculo.

En referencia a la clasificación anterior, las IAs de tipo fuerte se corresponderán, generalmente, con las IAs que cumplan con la teoría de la mente (tipo 3).

3. **Super IA**. La Superinteligencia artificial es una idea teórica de la que aún no conocemos con certeza la viabilidad real. Cuando hablamos de

---

[1]El FLOPS (Floating Point Operations Per Second, u *operaciones en coma flotante por segundo* en español) es una unidad que mide la cantidad de operaciones que un chip es capaz de realizar por segundo (1FLOPS = 1 operación/segundo). Un PetaFLOPS son 10^{15}FLOPS.

superIA nos referimos a un tipo de IA que no solo comprende e imita el comportamiento humano de una manera totalmente indistinguible de un humano real, sino que además adquiere consciencia propia (convirtiéndose en una IA de tipo 4 en la clasificación por funcionamiento) y supera las habilidades intelectuales de un humano.

Esta es la idea con la que todos los directores de cine han estado fantaseando tanto, y es el tipo de IA que siempre acaba rebelándose contra la humanidad. No existen casos reales de este modelo de IA, aunque Skynet podría ser un buen ejemplo. [34]

2.3 Machine Learning

El Machine Learning (ML de ahora en adelante) es el campo de la Inteligencia Artificial cuyo objetivo es desarrollar máquinas que sean capaces de realizar tareas sin ser específicamente programadas para ello, sino que poseen la habilidad de aprender con la experiencia[35]. En muchas circunstancias, la implementación de un algoritmo concreto puede resultar extremadamente compleja o impráctica, e incluso es posible que el propio programador no sepa cómo implementarlo. En este caso, puede ser conveniente que sea la propia máquina la que aprenda, que detecte cuáles son los patrones que sigue el entorno, cómo cada parámetro de entrada influye en la salida, y finalmente aprenda cómo resolver el problema.

El clásico ejemplo es el de reconocimiento de objetos. Algo tan sencillo como pedirle a una máquina que reconozca mesas es, en realidad, una tarea muy ardua de implementar. En el fondo, ¿qué es exactamente una mesa? "Un tablero horizontal con cuatro patas" parece una definición inicial razonable, sin embargo, un taburete encaja en esa descripción sin ser una mesa, y una mesa con menos de cuatro patas no lo hace.

Según la RAE, una mesa es un "Mueble compuesto de un tablero horizontal liso y sostenido a la altura conveniente, generalmente por una o varias patas, para diferentes usos, como escribir, comer, etc."[36]. Sin embargo, podemos encontrar mesas que están sujetas a la pared (sin patas). Además, ¿cuál se considera una altura conveniente? ¿La altura a la que una persona media está cómoda para escribir, comer, etc? ¿Acaso las mesas para niños no son mesas también?

La realidad es que no es así como funciona nuestro cerebro: la población general no conoce la definición exacta de una mesa. Sin embargo, todos sabemos diferenciar qué es una mesa y qué no, porque hemos ido aprendiendo, a base de experiencias, qué es una mesa y qué no lo es.

Esto es precisamente lo que pretende el ML: que no sea necesaria una definición formal del objeto que queremos identificar, o una implementación exacta de la solución del problema que buscamos resolver, sino que sea la propia máquina que, a través de la experiencia, aprenda a resolverlo.

El ejemplo que hemos visto anteriormente, el reconocimiento de objetos, es un problema que comúnmente se resuelve a través del Deep Learning, uno de los subtipos del ML que está teniendo mucho éxito en los últimos tiempos. Más allá de ése, se contemplan otros subtipos de ML.

2.3.1 Tipos de Machine Learning

Aunque alguno autores pueden llegar a considerar hasta seis tipo de ML[37], generalmente suelen considerarse tan solo los tres más importantes[38]:

1. **Aprendizaje Supervisado.** El aprendizaje supervisado es el campo al que pertenece el anteriormente mencionado Deep Learning. La manera más sencilla de visualizar este tipo de aprendizaje[39] es pensar en él como si estuviéramos enseñando a un niño a través de flashcards (o - *tarjetas didácticas* en español) [40]. En la fase de entrenamiento suministramos a la máquina una gran cantidad de datos *etiquetados* (es decir, entradas junto a su salida esperada) y dejamos que la máquina "aprenda" los patrones que relacionan la entrada con la salida. Una vez que la IA obtenga buenos resultados con los datos de entrenamiento, se

le muestran entradas diferentes a las que vio durante el entrenamiento y se comprueba la eficacia del mismo con datos reales (Figura 1).

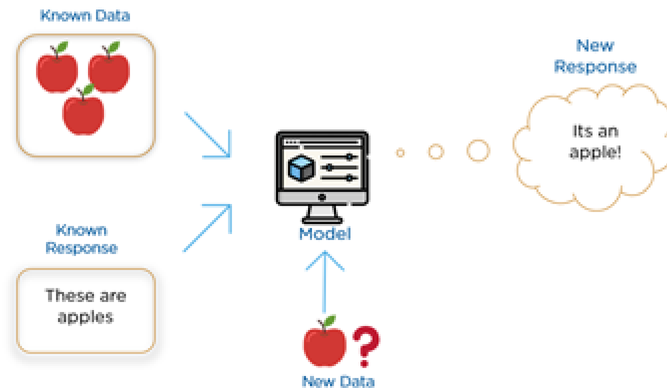

Figura 1 - Representación gráfica del proceso de aprendizaje supervisado. Extraído de [39]

En referencia al ejemplo anterior de detección de mesas, las entradas podrían ser fotografías con y sin mesas, y las salidas *True* o *False,* dependiendo de si en esa imagen hay o no una mesa. Cuando la máquina haya sido entrenada con una serie de fotografías, se le proporcionarán fotografías nuevas para comprobar que el entrenamiento es efectivo con imágenes no pertenecientes al dataset de entrenamiento.

Esto último se hace porque es posible que, por las características o el tamaño del dataset de entrada, la máquina acabe *sobreentrenada* o *infraentrenada*[41].

Por ejemplo, si usamos un dataset de tan solo diez imágenes es muy probable que el modelo acabe "memorizando" esas diez imágenes y no reconociendo como mesa ninguna otra. Por otro lado, si solo utilizamos imágenes de mesas marrones, es probable que cuando se le muestre al modelo una mesa negra o blanca, no la reconozca como mesa. Por esto resulta muy importante la elección del dataset en este tipo de entrenamiento.

2. **Aprendizaje No Supervisado.** A pesar de que el aprendizaje supervisado consigue muy buenos resultados en muchos ámbitos, tiene una importante inconveniente: requiere que todos los datos estén

etiquetados (es decir, que cada entrada tenga una salida correspondiente) en un mundo en el que la mayoría de la información no lo está.

Si bien una posible solución a esto son las llamadas *labeling factories*[42] (o *fábricas de etiquetado* en español), en las que los trabajadores se dedican exclusivamente a etiquetar datos, otra opción mucho más versátil es la de utilizar algoritmos que no requieran de datos etiquetados. Aquí es donde entra en juego el aprendizaje *no supervisado*. En el aprendizaje no supervisado, se le suministra a la máquina una gran cantidad de datos sin etiquetar, y será el propio algoritmo el que detecte patrones en los datos, y los separe en consecuencia.

En este tipo de aprendizaje, generalmente, las entradas serán características de los objetos que estamos intentando clasificar. Por ejemplo, el tamaño, forma y color de una fruta podrían ser parámetros entradas para un algoritmo clasificador de frutas. En ningún caso este algoritmo podrá decirnos "esto es una manzana" o "esto es una pera" como salida porque en él no existen los conceptos de "manzana" o "pera" (a diferencia del aprendizaje supervisado, que sí nos decía qué eran mesas y qué no), pero sí agrupará todas las manzanas por un lado y las peras por otro, en base a las características suministradas como entrada: buscará patrones en los datos de entrada y los agrupará en función de los mismos (es decir, agrupará las frutas según su tamaño, su forma y su color, Figura 2).

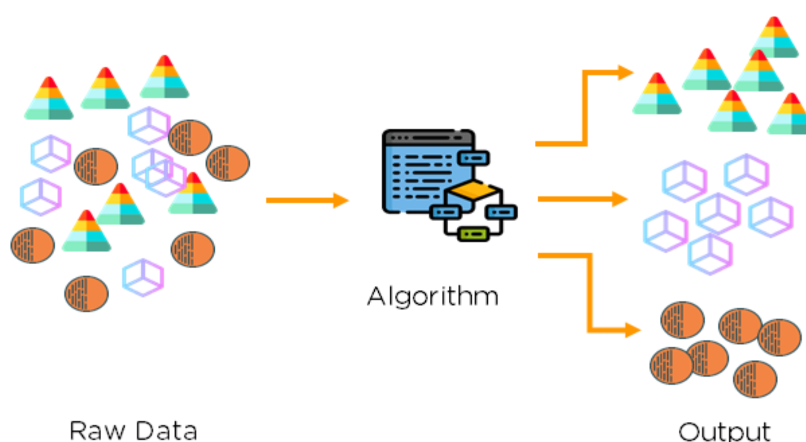

Figura 2 - Representación gráfica del aprendizaje no supervisado. Extraído de [38].

Este tipo de aprendizaje es utilizado especialmente en recomendadores (por ejemplo en Netflix[43] o Spotify[44]), o en los servicios de publicidad

a través de las cookies[45]: si un usuario ha indicado que le gusta una serie o canción, se le recomendará contenido similar.

3. **Aprendizaje por Refuerzo.** La relación entre el aprendizaje supervisado y el no supervisado parece obvia: el etiquetado (o no) de los datos. Sin embargo, el aprendizaje por refuerzo no tiene apenas relación con estos dos tipos de enfoques, ni siquiera existe un dataset (o conjunto de datos) del que extraer conclusiones. La manera más intuitiva de entender el aprendizaje por refuerzo es la de "aprender de tus errores". Como podemos observar en la Figura 3, el agente realiza acciones (que cambian su estado), y recibe del entorno una recompensa.

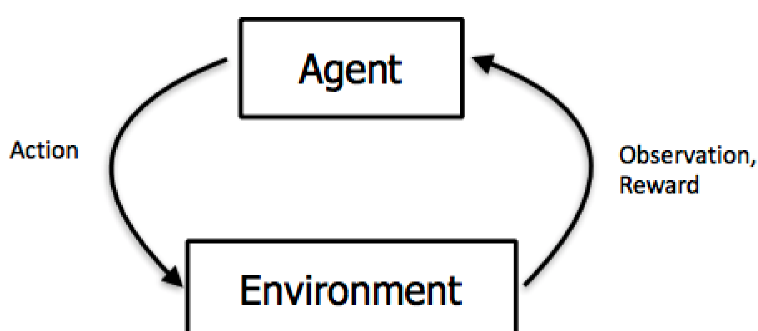

Figura 3 - Esquema básico del aprendizaje por refuerzo. Extraído de [46].

Para entender mejor este algoritmo, consideramos el siguiente ejemplo: un ratón que es entrenado para recorrer un laberinto y llegar hasta el queso (Figura 4). En este caso, las acciones que podrá realizar el agente (nuestro ratón) serán moverse arriba, abajo, a la derecha o a la izquierda. Cada vez que el ratón realice una acción, cambiará su estado (en este caso, su posición en el laberinto) y obtendrá una recompensa. Generalmente, esta recompensa será 0, ya que no gana nada por moverse a una casilla cualquiera. Sin embargo, existen algunas casillas en las que el ratón recibe una descarga: esas son las casillas con recompensa negativa, o castigo. También tenemos casillas con agua que sin duda agradarán a nuestro sediento ratoncito: cuando el ratón alcance una de estas casillas, obtiene una recompensa positiva. Por último, está

la casilla final con el queso, que es nuestro objetivo, y que por tanto será la recompensa más grande de todas.

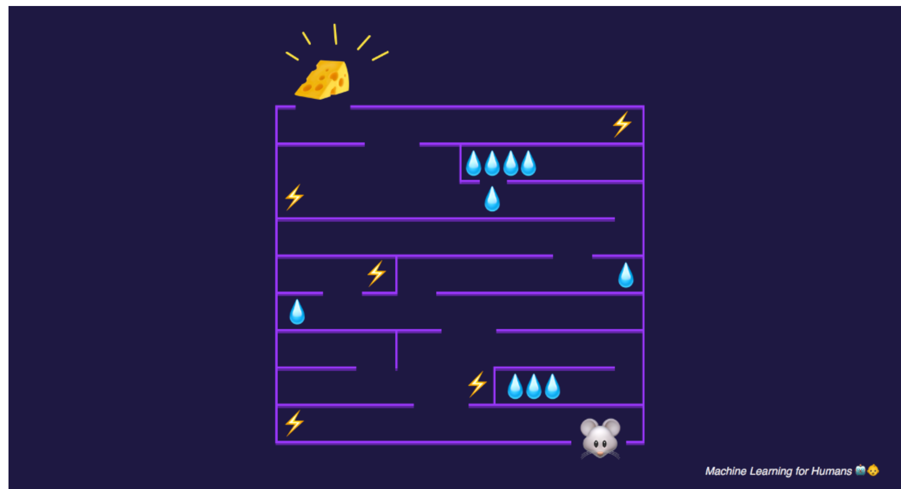

Figura 4 - Ejemplo de un caso de aprendizaje por refuerzo. Extraído de [38].

Lo que hará nuestro agente, representado por el ratón, será intentar maximizar la recompensa: alcanzar el queso sin recibir ninguna descarga y, de paso, beber todo lo que pueda. Para ello deberá aprender cuáles son las casillas con recompensa positiva, para alcanzarlas, y cuáles son las que tienen recompensa negativa, para evitarlas.

Uno de los usos más interesantes del aprendizaje por refuerzo son los videojuegos: se implementa este tipo de IA para jugar videojuegos con el objetivo de obtener puntuaciones muy altas. Uno de los primeros, y más importantes, casos fue el de Atari Breakout[47] en 2015, que obtuvo una puntuación extremadamente alta, e incluso llego a "darse cuenta" de que la estrategia más eficiente es la de cavar un túnel entre los bloques para que la bola se quede atrapada entre los bloques y el techo[48].

Es precisamente en este tipo de aprendizaje donde encontramos algoritmos como el **Q-Learning**, en el que nos centraremos durante el resto del presente trabajo.

3

Frozen Lake y Proceso de Decisión de Markov

Una vez introducidos los conceptos básicos sobre Inteligencia Artificial y Aprendizaje por Refuerzo (aunque profundizaremos más en este tema en la sección correspondiente) conviene hacer más explícito el contexto en el que nos encontramos.

3.1 El entorno Frozen Lake

El entorno Frozen Lake nos sitúa, como el nombre sugiere, en un lago helado y con agujeros en su interior. Desafortunadamente, nuestro frisbee (o disco volador) ha acabado en la otra esquina del lago y deberemos ir a recogerlo. Por supuesto, por el camino debemos evitar caer al agua helada por alguno de esos agujeros.

En la práctica, este entorno se ve representado por un tablero (véase la Figura 5) en el que cada casilla puede:

- Estar congelada (F)
- Ser un agujero (H)
- Ser la casilla de salida (S)
- O bien ser la casilla objetivo (G)

0.0 S	0.0 F	0.0 F	0.0 F
0.0 F	0.0 H	0.0 F	0.0 H
0.0 F	0.0 F	0.0 F	0.0 H
0.0 H	0.0 F	0.0 F	0.0 G

Figura 5 - Representación gráfica del entorno Frozen Lake

Con la información que tenemos hasta el momento, usar Machine Learning para resolver este problema puede parecer excesivo, como *matar moscas a cañonazos*: ¿Por qué no utilizar algún algoritmo de búsqueda de caminos, como A*[49], que son reconocidos por su eficiencia? Un motivo sería que el agente no podría ejecutar estos algoritmos porque no conoce el entorno pero, aun conociéndolo, hay algo más que caracteriza a Frozen Lake: su *no determinismo*. Que un entorno sea no determinista significa que, aunque realicemos la misma acción desde el mismo estado, no siempre obtendremos el mismo resultado. Esto se traduce en que el lago, como sabemos, está helado, por lo que es posible que el agente quiera ir hacia adelante, pero se resbale y acabe en la casilla derecha o izquierda (pero nunca en la de detrás). Esto complica mucho las cosas porque, aunque planeemos un camino óptimo utilizando A* o similares, es muy probable que no seamos capaces de ejecutar fielmente ese camino, porque antes o después nos resbalaremos. Por suerte, tenemos otra herramienta matemática para modelar y resolver esta clase de problemas: los Procesos de Decisión de Markov[50], o MDP por sus siglas en inglés.

3.2 Proceso de Decisión de Markov

A pesar de que los MDP pueden alcanzar una complejidad matemática considerable, esencialmente nos permiten modelar problemas o situaciones en las que los resultados son parcialmente dependientes del agente, pero también parcialmente aleatorios.

Volviendo a Frozen Lake, el resultado de nuestras acciones será en parte aleatorio, porque podremos acabar en una casilla distinta a la que queríamos dirigirnos, y en parte dependiente de nuestra acción porque nunca iremos en la dirección opuesta a la que inicialmente intentamos tomar (es decir, si vamos hacia adelante, nunca acabaremos en la casilla de atrás).

Lo más interesante de un MDP, es que define que las probabilidades de acabar en uno u otro estado no dependen en absoluto de los estados por los que se haya pasado anteriormente, sino únicamente del estado actual y de la acción que se ejecuta.

Esto, que se muestra en la Ecuación 1 (o, conceptualmente, en la Figura 6), simplifica bastante el proceso, porque implica que no es necesario conocer (o almacenar) todos los estados que se han visitado anteriormente para decidir qué acción tomar en el presente.

$$P(X_{t+1} = S_{t+1} | X_t = S_t, X_{t-1} = S_{t-1}, \dots, X_0 = S_0) \\ = P(X_{t+1} = S_{t+1} | X_t = S_t)$$

Ecuación 1 - Propiedad de Markov. Define que la probabilidad de que en el instante $t+1$ acabemos en el estado S_{t+1} es independiente de todos los estados anteriores.

Generalmente, ligado a un MDP, encontraremos una *función de transición*, que es la que determina la probabilidad, partiendo de un estado S y realizando una acción A, de acabar en un estado S'.

P(future | present, past) = P(future | present,)

Markov property

Figura 6 - Simplificación de la propiedad de Markov. Extraído de [51].

Como veremos más adelante, esta función puede ser conocida o desconocida por el agente, lo que cambiará su manera de actuar. Además, esta función de transición suele venir acompañada de una *función de recompensa*, que es la que le devolverá a nuestro agente la recompensa correspondiente a su acción.

4

Q-Learning

4.1 Aprendizaje por Refuerzo

El aprendizaje por refuerzo es el algoritmo ideal para resolver problemas que estén basados o modelados a través de MDP. Como idea intuitiva, se ha dicho que el aprendizaje por refuerzo consiste en "equivocarse y aprender de ello". Concretando un poco más, podemos desglosar el aprendizaje por refuerzo en seis fases principales [52]:

1. **Observación del entorno**. Obtendremos información sobre el entorno, pero, sobre todo, sobre nuestro estado respecto al mismo.
2. **Toma de decisión.** Una vez conocemos la información del entorno y de nuestro estado, debemos elegir alguna de las acciones posibles, siguiendo algún tipo de estrategia. Esta estrategia, a la que denominaremos *política*, es una parte crucial del algoritmo, y es precisamente la elección de diferentes estrategias lo que diferencia las distintas variantes del Q-Learning que analizaremos a continuación.
3. **Ejecución de la decisión tomada.** Una vez que sabemos cuál es la acción que debemos tomar, deberemos ejecutarla para ver cómo ésta afecta al entorno y a nosotros mismos.

4. **Asignación de la recompensa.** Ya vimos cómo cada acción tiene una recompensa (o castigo, si esta es negativa). En esta fase, será el entorno el que nos dé nuestra recompensa por la acción realizada.
5. **Mejora de nuestra estrategia.** Según la recompensa que recibamos, podremos deducir qué tan buena (o mala) fue la acción escogida. Por supuesto, si sabemos que la acción ha dado resultados positivos, querremos repetirla, mientras que querremos evitarla si nos ha generado un castigo: deberemos "actualizar" nuestra política en función de la recompensa obtenida.
6. **Repetición.** Por cada acción que realizamos, y cada recompensa que recibimos, estamos un paso más cerca de hallar la estrategia óptima para resolver el problema. Deberemos repetir todos los pasos anteriores y seguir mejorando nuestra estrategia hasta que tengamos una óptima. Más adelante se definirá qué se entiende por una "buena" estrategia en el entorno FrozenLake.

En términos generales, el aprendizaje por refuerzo puede ejecutarse de dos maneras, que difieren en su esencia: con algoritmos *model-based* (basado en el modelo) o *model-free* (sin modelo). La diferencia radica en que los algoritmos basados en el modelo conocen el funcionamiento del entorno (por ejemplo, un MDP), es decir, conocen su función de transición y de recompensa, y las utilizan para estimar la política óptima, mientras que los algoritmos sin modelo no conocen ni estiman estas funciones, sino que infieren la política basándose únicamente en la recompensa que van recibiendo al realizar acciones. El algoritmo del que hablaremos, Q-Learning, es un algoritmo *model-free.*

4.2 Q-Learning

Q-Learning es un algoritmo *off-policy*, es decir, que para estimar la política óptima utiliza una política distinta (en el caso de Q-Learning veremos cómo utiliza, por ejemplo, ε-greedy para estimar la política), al contrario que los algoritmos *on-policy*, que utilizan la misma política que están infiriendo para continuar con la estimación [53].

Esto significa que, cuando hablemos de Q-Learning (o cualquier otro algoritmo *off-policy*), tendremos que diferenciar entre la política que sigue el agente para aprender y la que está aprendiendo.

Más adelante veremos cómo las diferentes implementaciones de Q-Learning que examinaremos se diferencian en la política que utilizan para aprender, pero no necesariamente en la aprendida: la diferencia residirá principalmente en la eficiencia del aprendizaje.

Una vez aclarado esto, podemos adentrarnos más en el funcionamiento del algoritmo, resaltando los siguientes puntos clave:

- Q-Learning se basa en el uso de una matriz llamada Q para la estimación de la política óptima. La Q de Q-Learning viene de esta matriz, que, a su vez, se llama así por el término "Quality" ("calidad" en inglés).
- La matriz Q relaciona todos los estados y acciones con su ***valor esperado***, y almacena los primeros en las filas y las segundas en las columnas, asignando su *valor esperado* a cada combinación de estado y acción.

Q	Acción1	Acción2	Acción3	Acción4
Estado1				
Estado2				
Estado3				
Estado4				

Tabla 1 - Ejemplo de matriz Q, donde cada fila se corresponde con un estado y cada columna con una acción.

- El *valor esperado* es la recompensa que esperamos obtener al realizar una determinada acción desde un determinado estado. Es decir, es la recompensa que (creemos que) recibiríamos si, después de realizar esta acción, todas las decisiones que tomamos a continuación son acertadas, y se corresponde con el máximo de los valores de Q para ese estado.

Es decir, el algoritmo almacena en la matriz Q lo "bueno" que puede llegar a ser realizar cada acción desde cada estado. Recordemos que el objetivo es maximizar la recompensa y, por tanto, cuanto mayor sea el valor esperado, mejor creemos será ejecutar esa acción.

La matriz se inicializa con valores arbitrarios, que generalmente son nulos:

$$Q_{t=0}(s,a) = 0 \qquad \forall s \in [1, \dots, m];\ \forall a \in [1, \dots, n]$$

Para m estados y n acciones. Cuando se realiza una acción, se actualiza el valor esperado del par *estado/acción* seleccionado (estado en el que nos encontrábamos y acción que hemos realizado). Esta actualización se hace siguiendo la fórmula que podemos ver en la Figura 7.

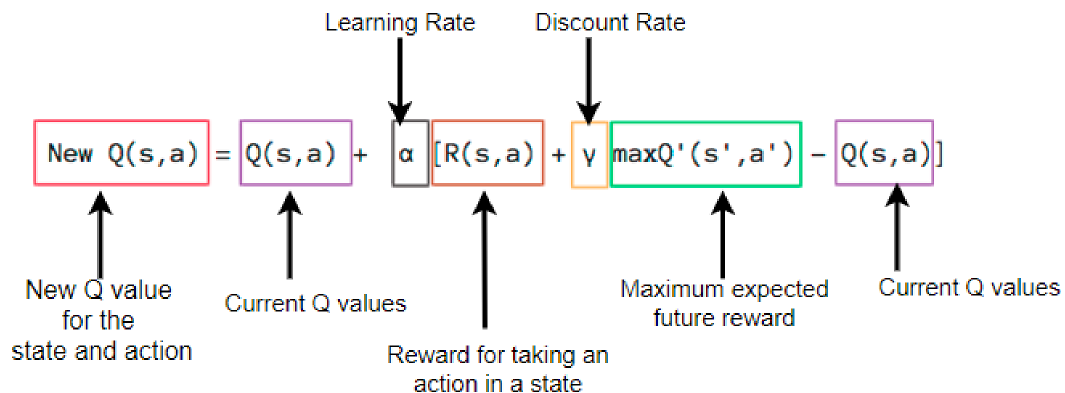

Figura 7 - Explicación de la fórmula del Q-Learning. Extraído de [52].

Observando la imagen, veremos que aparecen dos factores que no hemos mencionado aún: el ***Learning Rate*** (o tasa de aprendizaje) **α**, y el ***Discount Rate*** (o tasa de descuento) **γ.** Estos dos factores se conocen como *hiperparámetros*.

Los hiperparámetros son parámetros que no son aprendidos por la máquina, sino que son configurados por el diseñador, y que regulan el aprendizaje[54]. Ambos hiperparámetros tomarán siempre valores entre 0 y 1. Veamos qué representa cada uno de estos hiperparámetros y por qué son importantes.

4.2.1. Tasa de aprendizaje (α)

Si observamos la fórmula de la Figura 7, vemos que para actualizar el valor de Q utilizamos dos sumandos: el primero es el valor actual de Q para el estado s y la acción a, $Q(s,a)$, mientras que el segundo es un producto en el que interviene la tasa de aprendizaje.

Esto puede ayudarnos a deducir que la tasa de aprendizaje afecta de manera directa a la magnitud de la variación del valor de Q, es decir, cuanto más pequeña sea la tasa de aprendizaje, más lentamente cambiará el valor de Q, y viceversa. De ahí que se denomine tasa de aprendizaje, pues regula con qué velocidad

aprenderá la máquina. Aunque a primera vista podría parecer que cuanto más alta sea la tasa de aprendizaje más rápido aprenderá la máquina, y por tanto siempre convendrá elegir un valor alto de α, en realidad esto no es así.

Para entenderlo, pensemos en la función de recompensa: esta tendrá un máximo, que es el valor al que intentamos llegar. Con el fin de simplificar, representaremos esta función como una sencilla parábola, aunque una función de recompensa real difícilmente tendrá esta forma. Vamos a representar entonces la función que relaciona el valor de un par estado/acción (es decir, una casilla de Q) en el eje X con la recompensa final obtenida (en el eje Y). Recordemos que, aunque nosotros la estemos representando aquí, el agente **no** conoce esta función de recompensa.

Mirando entonces la Figura 8, podemos observar cómo una tasa de aprendizaje demasiado alta haría que el agente pasara de un extremo a otro, dando "saltos" demasiado grandes, sin alcanzar el objetivo (o haciéndolo tras muchas iteraciones) porque éste se encuentre a una distancia menor de la que la tasa de aprendizaje nos hace recorrer: el agente se "mueve" en dirección del objetivo, pero da pasos demasiado grandes.

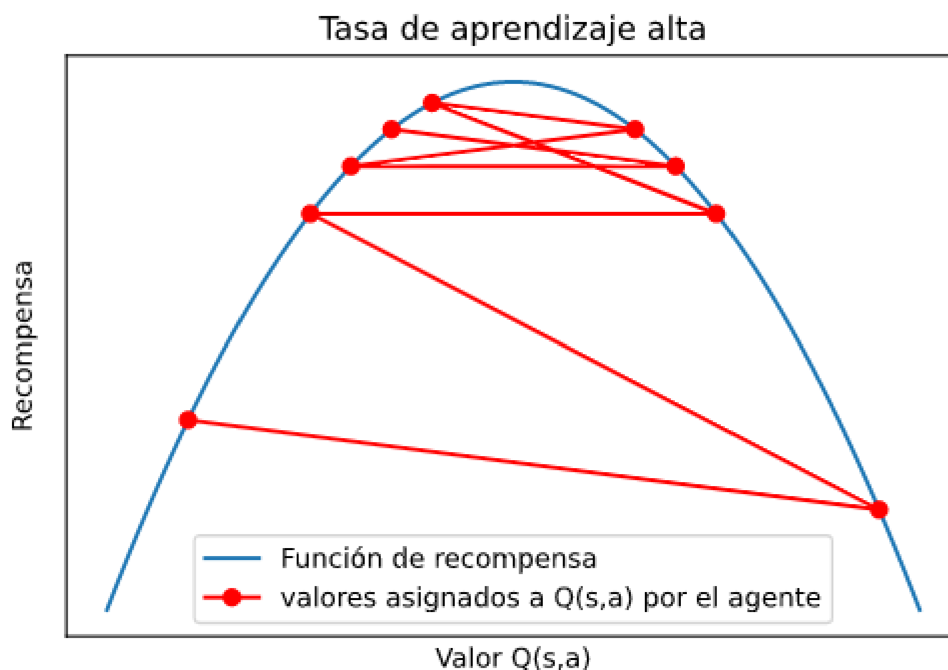

Figura 8 - Representación del proceso de aprendizaje con una tasa de aprendizaje muy alta.

Podríamos entonces pensar que resulta más eficiente utilizar una tasa de aprendizaje baja, ya que así siempre alcanzaríamos el objetivo sin sobrepasarlo.

Si bien esto es cierto, la Figura 9 nos muestra como una tasa de aprendizaje demasiado baja tampoco será óptima, puesto que requerirá de una altísima cantidad de pasos para alcanzar el objetivo.

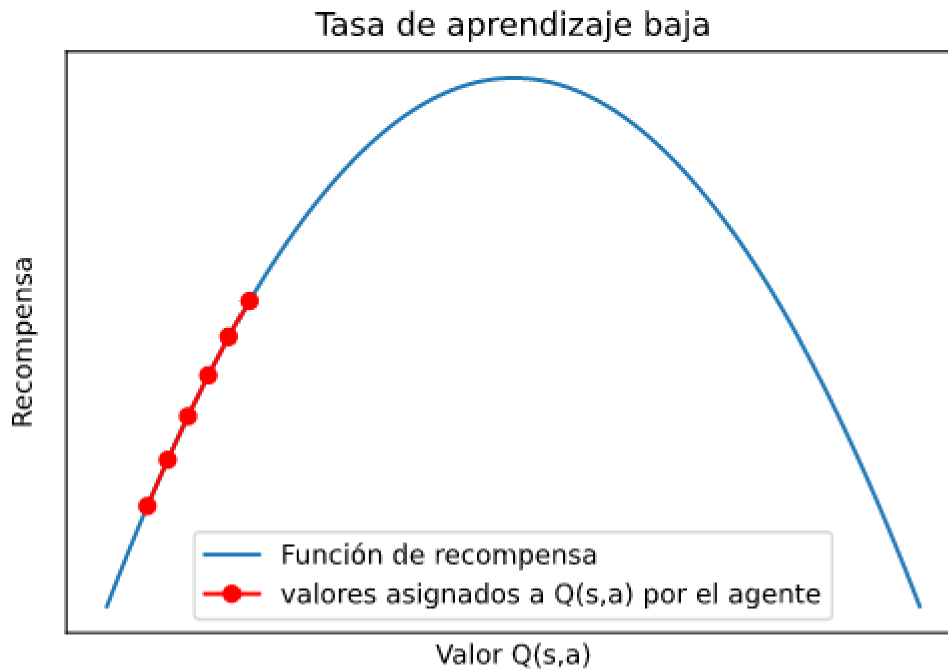

Figura 9 - Representación del proceso de aprendizaje con una tasa de aprendizaje muy baja.

Como ocurre en muchos ámbitos, la solución está en un punto intermedio: una tasa de aprendizaje óptima, como la de la Figura 10, implica utilizar un valor suficientemente pequeño como para no sobrepasar el objetivo, pero a la vez lo suficientemente grande como para no necesitar una cantidad excesiva de iteraciones.

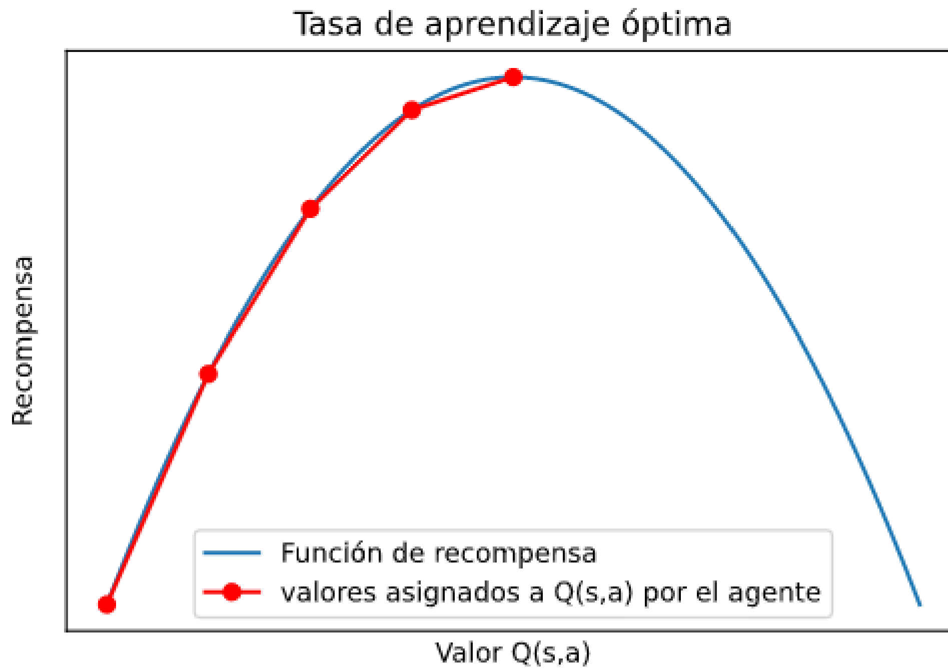

Figura 10 - Representación del proceso de aprendizaje con una tasa de aprendizaje óptima.

4.2.2. Tasa de descuento (γ)

Si nos centramos en el factor que multiplica a la tasa de descuento, veremos que es a su vez una expresión de varios términos. Intuitivamente, lo que se está haciendo es "corregir" la recompensa que pensábamos que obtendríamos ($Q(s,a)$) con la recompensa que realmente hemos obtenido ($R(s,a)$). Además, actualizaremos el valor de Q teniendo también en cuenta cómo de bueno es el estado al que hemos llegado, añadiendo a $Q(s,a)$ el máximo valor esperado de todas las acciones que se pueden realizar desde el nuevo estado ($\max Q'(s',a')$). Esto significa que cuando consigamos una recompensa positiva, todas las casillas cercanas se verán afectadas, y de este modo la recompensa se "propagará" a través de las casillas vecinas.

¿Qué representa entonces el factor γ que multiplica a este valor? La importancia que se le da a los estados futuros o, visto de otra manera, la velocidad con la que se propaga la recompensa. Un agente con γ=0 se centrará únicamente en la recompensa actual, ignorando la posible recompensa futura que pueda darle cierta acción. Otro con γ=1, por el contrario, se centrará en maximizar las recompensas futuras ignorando casi por completo la recompensa actual.

Podría entonces parecer razonable usar siempre tasas de descuento altas, para que el agente se centre en maximizar la recompensa total y no la inmediata. Sin embargo, esto

no es tan simple: cuanto más alta sea la tasa de descuento, mayor influencia tendrán los estados futuros sobre el estado actual, pero no en todos los entornos las acciones tienen tanta repercusión, y una tasa de descuento demasiado alta haría que cierta información irrelevante influya sobre el valor del estado actual.

Para visualizar mejor este concepto, pongamos un ejemplo más cotidiano[55]: imaginemos que al principio de cada mes nos tomamos un helado, y queremos decidir cuál es el que más nos gusta. Para ello, elegimos un helado y juzgamos la calidad de la elección en función de las recompensas que éste nos dé. Que el helado nos guste o no será una recompensa inmediata, pero también podríamos tener en cuenta si durante el resto del día nos encontramos mal, nuestro estado de ánimo al día siguiente, etc. Podríamos tener en cuenta tantos días como quisiéramos, con el fin de maximizar la recompensa total en lugar de la inmediata.

El problema es que, si tuviéramos en cuenta todo el mes, el hecho de que a las dos semanas nos encontráramos mal por, por ejemplo, comer comida en mal estado, repercutiría sobre la valoración que estamos haciendo del helado, a pesar de que los dos hechos no tengan relación. Del mismo modo, una tasa de descuento demasiado alta añadiría "ruido" a nuestra valoración de la acción, mezclándola con información innecesaria e incluso perjudicial.

Como vemos, en ambos casos es de extrema importancia ajustar los parámetros con cuidado y adecuarlos al problema y entorno con el que estemos trabajando. Ahora que conocemos cómo se actualiza la matriz en base a la realización de acciones y la obtención de la correspondiente recompensa, veamos cómo se seleccionan las acciones a ejecutar ya que, recordemos, Q-Learning es un algoritmo *off-policy* y por tanto trata de forma distinta la política que estima y la que utiliza para estimar. Podemos entonces formalizar Q-Learning como se muestra en el Algoritmo 1.

Algoritmo 1: Q-Learning

```
1   Inicializar Q arbitrariamente (generalmente a 0)
2   Por cada episodio repetir:
3       Inicializar el estado s
4       Por cada paso en el episodio repetir:
5           Según la política de selección (por ejemplo,
            Ɛ-greedy) seleccionar una acción a

6           Ejecutar la acción a, observar la recompensa r
            y el nuevo estado s'
7           Actualizar Q:
            Q(s,a) ← Q(s,a) + α[r + γmax_a(Q(s',a) − Q(s,a)
8           Actualizar s←s'
9       Hasta que s sea un estado final
```

Algoritmo 1 - Q-Learning.

Vemos que el algoritmo hace referencia a *episodios* y *pasos*. Un paso no es más que el ejecutar una acción con todo lo que esto conlleva (obtención de recompensa, actualización de Q, etc.).

Un episodio es un ciclo completo de pasos, es decir, un episodio termina cuando el agente alcanza alguno de los estados finales (en nuestro caso, bien la casilla objetivo o bien uno de los agujeros).

Tras un episodio, se reinicia el estado del agente y el entorno y se ejecuta otro episodio desde el principio. Esto se hace para aprender del resultado del episodio anterior, ya que un solo episodio no es suficiente para aprender una buena política. También se hace referencia a la política de selección (aquella que utilizamos para decidir qué acciones tomar durante el entrenamiento), y a Ɛ-greedy. Exploraremos estos conceptos en la siguiente sección.

5

Políticas implementadas

5.1 Explotación vs exploración

La política estimada por Q-Learning es sencilla de seguir: para cada estado, realizar la acción con mayor valor esperado. A esto lo llamaremos *explotación*, porque *explotamos* el conocimiento que tenemos.

Esto sin embargo resulta de poca utilidad cuando tenemos una política que aún no está formada y la matriz Q está compuesta únicamente de 0.

Además, cuando la política empiece a estar formada y tengamos una opción favorable, podríamos caer en un máximo local, es decir, elegir siempre la acción que nos proporciona resultados, cuando es posible que haya otra que nos resulte aún más ventajosa y que aún no hayamos visitado.

Por ello resulta interesante, de vez en cuando, tomar acciones a pesar de que según nuestro conocimiento no parezcan óptimas: porque podrían serlo, pero aún no las hayamos ejecutado.

Esto es lo que llamaremos *exploración*: tomar acciones distintas a las que nos recomienda nuestro conocimiento para *explorar* cómo de buenas (o malas) son. Por tanto, es fundamental encontrar un equilibrio entre *exploración* y *explotación.* La manera de conseguirlo varía según las distintas políticas y variantes de Q-Learning, aunque la más básica es ε-greedy.

5.2 Ɛ-greedy

Ɛ-greedy es la más sencilla de las variantes que se han implementado, aunque es también la más conocida por ser la "original", y se asocia comúnmente con el algoritmo Q-Learning.
En esta política se añade un nuevo hiperparámetro: Ɛ, que determinará la probabilidad de exploración del algoritmo. Esto significa que, a la hora de seleccionar una acción (paso 5 del Algoritmo 1), tendremos una probabilidad Ɛ de elegir una acción aleatoria, y una probabilidad (1−Ɛ) de elegir la acción que Q nos dice que es más ventajosa (Algoritmo 2).
Este sencillo enfoque nos permite distanciarnos de los máximos locales a través de la exploración, alcanzando el máximo absoluto.
Al igual que con los demás hiperparámetros, en este caso también resulta complicada la elección del valor adecuado para Ɛ: un valor muy alto haría que demasiadas acciones se tomaran de manera aleatoria, empeorando la eficiencia. Un valor demasiado bajo, en cambio, no sería suficientemente efectivo para evitar caer en máximos locales.
En esta ocasión, la práctica más común es la de no utilizar un valor estático de Ɛ para todo el entrenamiento, sino irlo reduciendo con el tiempo.
Durante el primer episodio, los estados que se han visitado son muy pocos, y por tanto la explotación es poco significativa, ya que los máximos en nuestra matriz Q están basados en una cantidad de datos insuficiente. Por eso, en esta fase resulta conveniente tener una alta tasa de exploración, para conocer realmente qué es lo que nos rodea. Según va avanzando el entrenamiento, nuestra Q está cada vez mejor informada, la cantidad de estados y acciones exploradas será cada vez mayor y, por tanto, es más sensato reducir la tasa de exploración. Esto se denomina *Ɛ decay* o decaimiento Ɛ.

Aunque existen muchas maneras de implementar esta característica, en este proyecto se ha optado por utilizar un decaimiento exponencial a través de un parámetro adicional llamado *decaimiento épsilon*.

Este será un número entre 0 y 1, y determina cuánto disminuye Ɛ en cada episodio.

Más concretamente, en cada episodio se actualiza Ɛ como el producto de Ɛ y el decaimiento Ɛ. De esta forma, un valor de 1 implicará que Ɛ no disminuye con el tiempo, mientras que un valor de 0.9 significará que en cada episodio Ɛ será un 90% de lo que fue en el episodio anterior.

Algoritmo 2: Selección de acciones con Ɛ-greedy

```
1   Seleccionar un número aleatorio r entre 0 y 1
2   Si r<Ɛ:
3       Seleccionar una acción aleatoria a
4   En otro caso:
5       Seleccionar la acción a con mayor valor esperado
        a = arg max_a Q(s,a)
6   Devolver a
```

Algoritmo 2 - Selección de acciones a través de Ɛ-greedy.

5.3 SoftMax

Aunque Ɛ-greedy funciona bastante bien, tiene un fallo importante, y es que hay una probabilidad Ɛ de tomar una acción de manera *completamente aleatoria*.

Esto significa que en Ɛ-greedy tenemos las mismas probabilidades de elegir la peor acción que la segunda mejor, cuando en muchos casos no nos convendrá elegir la que (pensamos que) es peor.

Por ello surge la idea de utilizar la función SoftMax.

La función SoftMax, definida como en la Ecuación 2, es una función matemática que se utiliza para *normalizar* las probabilidades de un conjunto de elementos.

Esta función toma como entrada un conjunto de elementos y asigna a cada uno de ellos una probabilidad entre 0 y 1, de manera que la suma de todos sea 1.

Este valor será mayor cuanto mayor sea el valor original de ese elemento (probabilidad "ponderada"). La probabilidad de la i-ésima acción a, se define según:

$$Softmax(a_i) = \frac{e^{a_i}}{\sum_{k=1}^{dim(a)} e^{a_k}}$$

Ecuación 2 - Función Softmax.

En el campo del aprendizaje por refuerzo, además, se utiliza con frecuencia [3] una variante de Softmax que incluye el nuevo parámetro τ, denominado temperatura (Ecuación 3).

La temperatura determina la proporcionalidad en las probabilidades generadas. Esto significa que cuanto menor sea τ, más acentuada será la diferencia en la probabilidad entre las acciones con mejores y peores valores esperados, y al contrario, cuanto mayor sea τ, menor será esta diferencia y más equitativo será el reparto de probabilidades.

De manera análoga que con Ɛ-greedy, al principio conviene explorar, y según van pasando los episodios, progresar hacia una explotación cada vez más influyente.

Por ello, se utiliza el mismo enfoque que con Ɛ, y se inicia el algoritmo con una temperatura alta (haciendo que todas las acciones tengan probabilidades muy similares y que, por tanto, las acciones se tomen de manera casi aleatoria), y poco a poco se va menguando de manera que la acción con mayor valor esperado vaya cobrando más importancia.

$$Softmax(a_i) = \frac{e^{\frac{Q(a_i)}{\tau}}}{\sum_{k=1}^{dim(Q)} e^{\frac{Q(k)}{\tau}}}$$

Ecuación 3 - Función Softmax con temperatura

Si observamos esta fórmula veremos que, además de en la notación, no encontramos grandes diferencias: tan solo se ha añadido τ como divisor en los exponentes de e.

Una vez que calculamos las probabilidades de cada una de las acciones, únicamente deberemos elegir una acción acorde a ellas, lanzando un número aleatorio y seleccionando la acción correspondiente.

Si, por ejemplo, como resultado de la función Softmax, obtuviéramos las probabilidades [0.6, 0.2, 0.1, 0.1], significaría que al lanzar el número aleatorio:

- Elegiríamos la primera acción si este se encuentra entre 0 y 0.6
- Elegiríamos la segunda acción si este se encuentra entre 0.6 y 0.8
- Elegiríamos la tercera acción si este se encuentra entre 0.8 y 0.9
- Elegiríamos la última acción si este se encuentra entre 0.9 y 1.

De este modo, seguimos manteniendo cierta aleatoriedad, aunque las acciones con mejores valores esperados tendrán más probabilidades de ser elegidas.

5.4 Upper Confidence Bound

En cualquier entorno no determinista, como lo es FrozenLake, existe incertidumbre sobre cuán bueno es realmente ese estado, pues no siempre nos ofrecerá los mismos resultados. Los algoritmos explorados hasta ahora afrontan esta incertidumbre de un modo más bien conservador, abriendo la puerta a las acciones con peores valores de Q, ya sea de manera totalmente aleatoria (como en ε-greedy) o ponderada (como en SoftMax). UCB (Upper Confidence Bound, o *límite superior de confianza*), en cambio opta por un enfoque más *optimista*.
Este algoritmo define un *intervalo de confianza*, que representa la del valor *estimado* de la acción (contenido en la matriz Q), dentro de la cual podemos asegurar que se encuentra el valor *real* de esta (Figura 11).

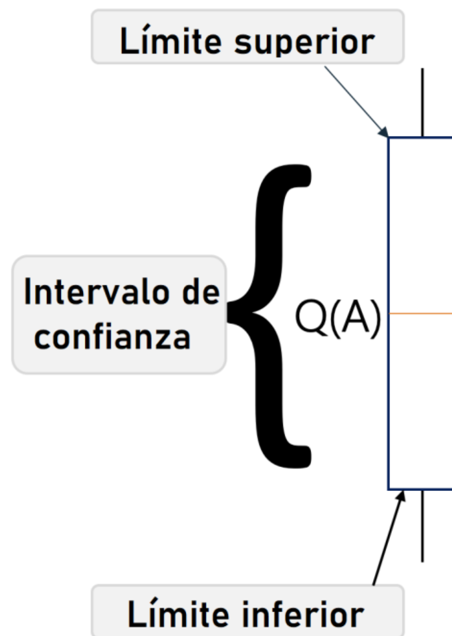

Figura 11 - Representación del intervalo de confianza en UCB. Adaptado de [56] .

Puesto que el valor real puede encontrarse en cualquier punto de ese intervalo, el algoritmo UCB, dentro de su optimismo, asumirá que se encuentra en el *límite superior* del mismo, de modo que, al seleccionar una acción, elegirá aquella cuyo límite superior sea mayor. Esta es la diferencia fundamental con los algoritmos vistos hasta ahora: no seleccionamos la acción únicamente en función del valor estimado de Q, sino que también tendremos en cuenta de manera explícita la incertidumbre del mismo. Llamaremos *bonus* β a la diferencia que hay entre el valor estimado de Q y el límite superior de su intervalo de incertidumbre, de manera que podremos formalizar la selección de acciones como en la Ecuación 4:

$$\mathrm{A} = \arg\max_{\mathrm{a}}\big(Q(s,a) + \beta(s,a)\big)$$

Ecuación 4 - Selección de acciones con UCB.

En la que podemos ver que la acción seleccionada A no es más que aquella que cuenta con el límite superior más alto.

Este bonus se calcula como se define en la Ecuación 5:

$$\beta(s,a) = C\sqrt{2 \times \frac{logN(s)}{\sum_{i=0}^{dim(a)} N(s,a_i)}}$$

Ecuación 5 - Función *bonus* en UCB

En esta función se aprecia la presencia de un nuevo hiperparámetro C, llamado también *valor de confianza*, y que regula el nivel de exploración del algoritmo: valores altos de C favorecerán una mayor exploración, mientras que $C = 0$ implicará la total ausencia de ella [57]. Aunque generalmente C suele definirse como una constante, en este trabajo se ha añadido otro hiperparámetro *decaimiento* C que, de manera análoga a los casos anteriores, reduce el valor de C con el tiempo, debido a que se ha comprobado cómo, al menos en este entorno, el decaimiento de C favorece un mayor rendimiento del algoritmo.

A diferencia de otros algoritmos, UCB, además de la matriz Q, también almacena la matriz N, que contiene el número de veces que se ha tomado una determinada acción, de manera que $N(s, a)$ representa cuántas veces se ha tomado la acción a desde el estado s. Este dato resulta indispensable para el cálculo de la incertidumbre, ya que cuantas más veces hayamos tomado una acción, menor será la incertidumbre sobre la misma.

En la función β vemos cómo se calcula cuántas veces se ha tomado una acción a respecto al número total de veces que se ha alcanzado el estado s. Lo que conseguimos calculando la proporción de estos dos factores es que la incertidumbre sobre un par estado/acción decrezca cada vez que lo exploramos, pues tenemos más información *real* sobre la que basar nuestras decisiones.

6

Software Desarrollado

Tras conocer todo el fundamento teórico, veamos ahora el software que se ha diseñado para visualizar este algoritmo y sus variantes, así como sus métricas de rendimiento.

6.1 Diagrama de clases

Este proyecto se ha desarrollado siguiendo el paradigma de Programación Orientada a Objetos[58], aunque no todo el funcionamiento está incluido dentro de alguna clase: el propio Q-Learning, por ejemplo, es un módulo independiente que no forma parte de ningún objeto.

En el ámbito del diseño del software, se utiliza UML (*Unified Modelling Language* o *Lenguaje Unificado de Modelado)* para ilustrarla arquitectura de un proyecto, marcando así los elementos que intervienen y las relaciones que hay entre ellos[59].

Vemos en la Figura 12 un diagrama UML de todas las clases del proyecto, así como los métodos públicos más importantes.

Nótese que, para reducir la complejidad del diagrama, no todos los atributos ni métodos se muestran, sino tan solo aquellos que resulten de mayor importancia para la comprensión del diseño.

Es importante destacar que, a pesar de que muchos métodos y parámetros se marcan como privados, puesto que es lo que dictan las buenas prácticas de programación, en la realidad Python, lenguaje en el que se ha desarrollado este software, no permite modificar la visibilidad de sus atributos, sino que utiliza una nomenclatura específica para los elementos privados o protegidos, y confía en la "buena fe" del programador, que deberá acatar la convención de nomenclatura y no acceder a esos campos o métodos [60].

Figura 12 - Diagrama de clases del proyecto.
Este diagrama se ha desarrollado utilizando la herramienta PlantUML[67] para el IDE Pycharms [8] y la hoja de estilos desarrollada por el usuario Drakemor en GitHub [68].

Ya que se trata de un esquema bastante complejo, desglosémoslo remarcando los puntos clave.

6.2 Patrones de diseño

El diseño se basa principalmente en el patrón MVC [61](Modelo-Vista-Controlador), uno de los patrones de diseño más extendidos en el campo del desarrollo de software que cuentan con una interfaz gráfica.

6.2.1 Modelo-Vista-Controlador

Como el nombre sugiere, el patrón MVC se basa en tres componentes para definir el flujo de trabajo:

- **El modelo** es el componente que contiene toda la lógica del programa.
  Modela el escenario que se desea simular, contiene los datos, etc.
  En nuestro caso, el modelo se corresponde con la clase *Agente* y el módulo *qlearning*.
- **La vista** es el componente visual, la Interfaz Gráfica de Usuario, o GUI en inglés. Este componente no contiene ninguna lógica, tan solo muestra los elementos visuales al usuario, como botones, textos, etc. de manera que el usuario pueda interactuar con la aplicación. En nuestro caso, esto está representado por el módulo *ventanas* y todas las clases en él contenidas (*VentanaPrincipal, VentanaMetricas, VentanaBenchmark y VentanaAjustesBenchmark*)
- **El controlador** es la pieza faltante entre estos dos componentes: el que recoge las acciones del usuario desde la vista, las envía al modelo para que las procese, y envía de nuevo el resultado a la vista para que esta lo muestre. Hace de intermediario, de manera que la vista y el modelo no deban interactuar directamente en ningún momento, lo que garantiza un alto grado de modularidad en la aplicación.
  En nuestra aplicación, la función de controlador la realiza la clase *Controlador* del módulo *Controlador.py.*
  Cabe destacar que, en este caso concreto, sí se ha introducido cierta lógica en el controlador, a pesar de que esto no respete completamente el patrón, para diferenciar cuáles acciones pertenecían a la lógica del algoritmo (contenidas en *Agente* y *qlearning*) y cuáles a la de la GUI (contenidas, de manera excepcional, en el controlador).

6.2.2 Envoltorio

Si bien esta aplicación está diseñada específicamente para el entorno FrozenLake, uno de sus objetivos es la modularidad y la sencillez a la hora de cambiar o modificar componentes y comportamientos. Por eso, ningún componente utiliza directamente el entorno de OpenAIGym de FrozenLake, sino un *wrapper* (envoltorio) del mismo que se encuentra en la clase *FrozenLake*.

El patrón wrapper es una simplificación del patrón *decorator* [62], que se limita a "envolver" un objeto, ofreciendo la misma interfaz (es decir, los mismos métodos) e interactuando con el objeto envuelto, pero sin que este interactúe con el exterior ni viceversa. Lo que esto permite es que sea muy sencillo cambiar el objeto envuelto por otro, ya que el resto de clases y métodos se verán inalterados (puesto que interactuaban con el wrapper y no con el objeto en sí).

Trasladado a nuestro caso concreto, esto implica que cambiar de FrozenLake a cualquier otro entorno tan solo requeriría modificar la clase *FrozenLake* (aunque sería necesario crear un nuevo widget para representar gráficamente el entorno ya que, como veremos más adelante, el este específicamente diseñado para *FrozenLake*).

Podemos ver este comportamiento en la Figura 13, donde se muestra cómo FrozenLake mantiene una relación de *agregación* con la clase *Env* de OpenAIGym.

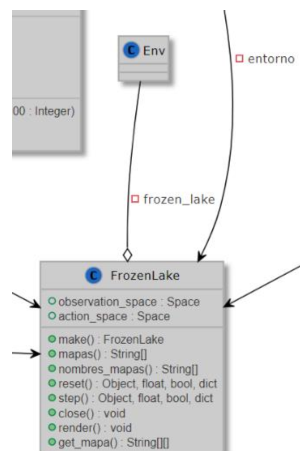

Figura 13 - Patrón wrapper en FrozenLake.

6.2.3 Estrategia

La característica principal de esta aplicación es la posibilidad de entrenar agentes en un entorno con varios algoritmos distintos y compararlos.

Realizar esto en una sola clase resultaría extremadamente complicado e ineficiente, sobre todo a la hora de modificar, añadir o eliminar políticas.

Si lo pensamos detenidamente, lo que estamos intentando hacer con todos los algoritmos es lo mismo: aprender a resolver el problema, aunque cada algoritmo utiliza un enfoque, o *estrategia*, distinto.

El patrón *Strategy* [63] (o estrategia), ofrece la solución a este tipo de problemas. Este patrón consiste en la creación de una interfaz (o clase abstracta) que agrupa a toda la familia de algoritmos, y de la que hereda cada uno de ellos en clases independientes. De este modo, añadir una nueva política será tan sencillo como crear una nueva clase que herede de la clase abstracta *Política* e implemente los métodos necesarios. Así, la lógica del módulo *qlearning* es capaz de cambiar de un algoritmo a otro sin ninguna dificultad, puesto que todos son en el fondo instancias de la clase *Política*.

En este caso concreto, podríamos considerar también que se tratara del patrón *template method* [64] ya que, para casos en los que los algoritmos no intercambian objetos entre sí (como es el caso) ambos patrones son equivalentes.

En la Figura 14 podemos ver cómo está estructurado este patrón de diseño, con la clase abstracta *Política* y las distintas políticas implementadas heredando de ella.

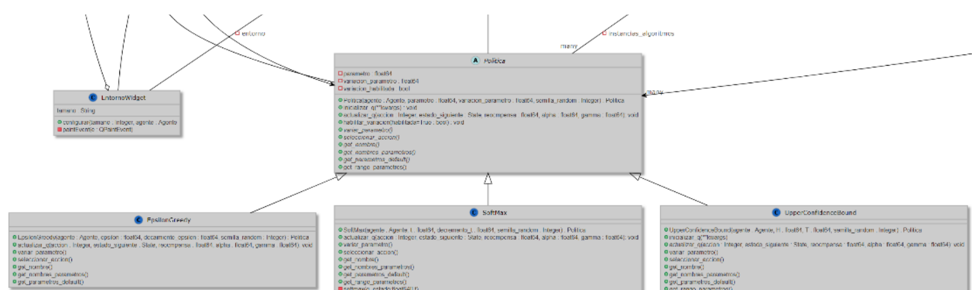

Figura 14 - Patrón estrategia aplicado a las políticas

Gracias a la flexibilidad que este patrón ofrece, la aplicación está completamente diseñada para mantener un número variable de algoritmos en todas sus ventanas. Así, añadir un nuevo algoritmo consiste simplemente en implementarlo

(heredando de *Política*) e invocar el método *registrar_algoritmo*() de la clase *Controlador* proporcionando, como parámetro, la clase del algoritmo que se desea añadir.

En la Figura 15 se muestra, como ejemplificación de ello, el código completo de la función *main*() para crear y añadir todos los algoritmos al controlador. Vemos como, con solo añadir una línea, nuestro algoritmo estará completamente disponible para su ejecución.

```
c = Controlador()
c.registrar_algoritmo(SoftMax)
c.registrar_algoritmo(UpperConfidenceBound)
c.start()
```

Figura 15 - Proceso de creación del controlador.

6.2.4 Delegación

Durante todo el capítulo, cuando se ha hablado del modelo, se ha hablado de la clase *Agente* y de módulo *qlearning* de manera conjunta, casi como si de una única entidad se tratase.

Esto, lejos de ser fruto del azar, se debe a la aplicación del patrón *delegación*. Este patrón busca, como muchos otros, la máxima abstracción posible y, para ello, delega sus responsabilidades sobre otras clases o entidades. En este caso el *Agente*, que es el encargado de aprender y resolver el problema, delega esta tarea sobre el módulo *qlearning*, que es el que contiene toda la lógica para el entrenamiento y ejecución de la política.

En cierto modo, podríamos decir que esto convierte al *Agente* en un "contenedor" de la matriz Q que sirve como soporte para *qlearning*, quien se encargará de la lógica del programa. Este mecanismo se ha visto modificado durante el desarrollo del proyecto por la inclusión del *multithreading*, que ha alterado ligeramente la estructura original del programa en algunos casos.

6.2.5 Multithreading

Aunque no se trata de un patrón de diseño en sí mismo, es sabido que cualquier aplicación que haga un uso intensivo de la CPU debe realizar estas operaciones en una hebra independiente de la de la GUI para evitar ralentizaciones y fallos de respuesta. Para solventar esta limitación, la aplicación cuenta con tres clases

que se pueden ver en la Figura 16 (*ThreadEntrenamiento, ThreadEjecucion y ThreadBenchmark)* que son las que se encargan de realizar todos los cálculos y operaciones no relacionadas con la GUI y que, por tanto, se ejecutan en hebras independientes.

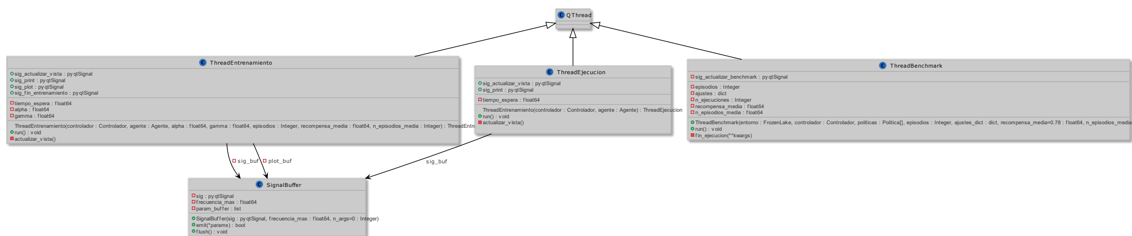

Figura 16 - Implementación del multithreading.

En cierto modo, ya que son estas clases las que manejan la lógica del agente y de *qlearning*, podría verse como otro patrón de delegación: la clase *ThreadEntrenamiento* por ejemplo, cuya función es entrenar al agente, delega esta labor sobre la clase *qlearning*. Estas clases, al fin y al cabo, no son más que "contenedores", que permiten ejecutar ciertos métodos en hebras diferentes.

Por supuesto, la comunicación entre la hebra de la GUI y las hebras de cálculo intensivo, jamás se realiza de manera directa, ya que esto podría dar lugar a una *race condition* [65], esto es, cuando dos o más hebras modifican una misma variable simultáneamente produciendo valores incoherentes. Al contrario, esto siempre se realiza a través de las señales que contiene cada hebra, y que pertenecen a la clase pyqtSignal.

6.2.6 Callbacks

Aunque esto tampoco puede considerarse un patrón de diseño como tal, es una parte destacable de la arquitectura del sistema que influye enormemente en el resto del diseño.

Buscando una vez más maximizar la modularización del sistema, el módulo *qlearning* se ha diseñado de tal manera que pueda modificarse parte de su funcionalidad sin alterar el código del script.

Esto se ha diseñado en base a un sistema de *callbacks*, aprovechando la versatilidad de Python que permite asignar funciones a variables.

El módulo ofrece por tanto una serie de variables que se utilizan como callbacks, y que se ejecutan en determinadas situaciones (véase Figura 17).

```python
    qlearning.callback_entrenamiento_fin_entrenamiento = self.fin_ejecucion

def fin_ejecucion(self, **kwargs):
    bundle = kwargs.get('bundle')
    n_pasos = bundle.n_episodio
    self.sig_actualizar_benchmark.emit(self.politica_actual, n_pasos)
```

Figura 17 - utilización de los callbacks de *qlearning* en la clase *ThreadBenchma*

Los nombres de los callbacks son descriptivos, de manera que el callback "*calback_entrenamiento_fin_paso*" se ejecutará al final de cada paso. De este modo, si quisiéramos que al final de cada paso se imprimiese el estado actual, tan solo deberíamos asignar a ese callback la función que imprima el estado. Las funciones que se asignen al callback deberán aceptar el parámetro **kwargs, a través del cual se enviará un objeto de la clase QLearningBundle, que contendrá información de interés respecto al estado actual del módulo qlearning (la recompensa media hasta ese momento, el número de pasos, etc).

De esta manera se ha diseñado el sistema de tiempo de espera entre pasos: el módulo qlearning es totalmente ajeno a esta funcionalidad, pero uno de sus callbacks (al fin de cada paso) apunta a una función que contiene una espera.

Esto permite a los usuarios más avanzados, poder alterar la funcionalidad básica sin necesidad de acceder ni comprender el código intrínseco a este módulo.

6.3 Interfaz de usuario

Ahora que conocemos el diseño interno de la aplicación y su arquitectura, es momento de presentar cómo es visualmente el software. Se detallarán a continuación las principales funcionalidades de la aplicación de una manera visual, tal y como lo vería el usuario final.

6.3.1 Ventana principal

Nada más iniciar la aplicación, veremos la ventana principal que se muestra en la Figura 18. El elemento central de la interfaz es el componente *EntornoWidget*. Este ha sido diseñado específicamente para el entorno FrozenLake, y muestra el lago helado de nuestro problema, representando las casillas con la nomenclatura ya mencionada:

- S- casilla de salida
- G - casilla objetivo
- F - casilla congelada (transitable)
- H - agujero (estado terminal con castigo)

Además, encontramos varios botones, cuyo nombre es generalmente indicativo de la funcionalidad, como el de *play/pause*, que pausa o reanuda la ejecución o entrenamiento del agente, o el de *reset*, que restaura el estado del agente, del entorno y todos los valores de los parámetros configurables.

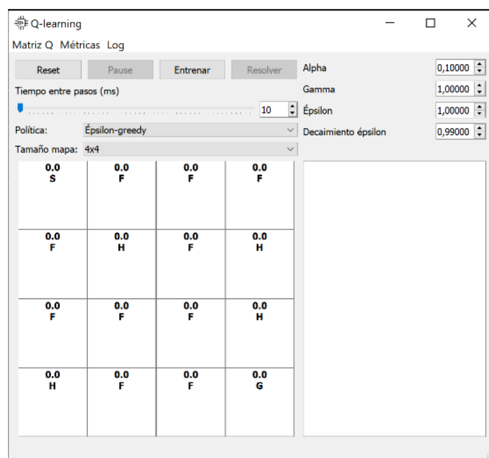

Figura 18 - Ventana principal al iniciar la aplicación

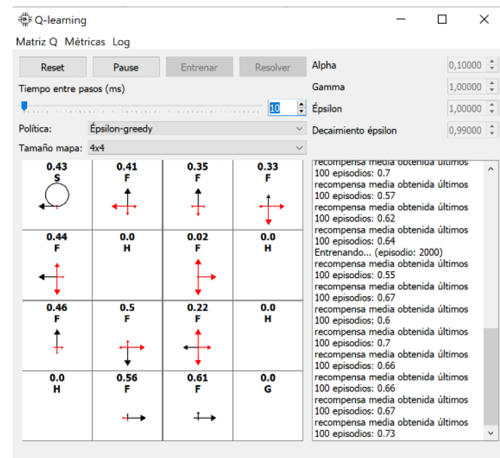

Figura 19 - Ventana principal durante el entrenamiento

Bajo los botones encontraremos varios parámetros de configuración:

- Por un lado, un control deslizante nos permitirá elegir el tiempo, en milisegundos, que se esperará tras ejecutar un paso y antes de ejecutar el siguiente. Esto resulta útil para regular la velocidad de ejecución, reduciéndola para observar con detalle qué ocurre en cada momento o, al contrario, maximizándola para finalizar el entrenamiento lo antes posible. Este parámetro puede configurarse también a través de la casilla de texto que se encuentra a la derecha, y que está sincronizada con el control deslizante.
- Debajo encontraremos los dos desplegables que caracterizan la aplicación: la selección de la política, y la selección del tamaño del mapa. Como ya se ha comentado, las políticas que se encuentran implementadas actualmente son *ε-greedy, SoftMax y Upper Confidence*

Bound, aunque otras políticas que sean añadidas por el usuario se mostrarían de igual modo en el desplegable.

El mapa, por otro lado, está disponible en los dos tamaños que ofrece el framework de OpenAIGym: 4x4 y 8x8.

A la derecha encontramos los distintos hiperparámetros que se han mencionado anteriormente, junto a una casilla de texto para poder configurarlos.

Nótese que dos de estos hiperparámetros, α y γ, serán fijos, mientras que los otros dos cambiarán en función del algoritmo que seleccionemos.

En el ejemplo en Figura 18 el algoritmo seleccionado es Ɛ-greedy, y por tanto los parámetros que aparecen son Ɛ y $decaimiento$ Ɛ.Si seleccionáramos el algoritmo SoftMax, encontraríamos en su lugar los parámetros τ y *decaimiento* τ.

Una vez configuradas las variables descritas hasta ahora, se inciaría el entrenamiento del agente a través del botón *entrenar*.

Durante el entrenamiento no será posible modificar los hiperparámetros, ya que esto produciría un efecto disruptivo y, por ello, las distintas casillas de configuración se deshabilitan hasta que se haga un *reset*.

Mientras el agente se esté entrenando, se mostrará una representación en tiempo real de su matriz Q en el componente *EntornoWidget* sobre el que se dibujarán unas flechas y se mostrarán los siguientes valores (Figura 19).

En primer lugar, no se está mostrando la matriz Q tal y como se definió anteriormente (una matriz de tamaño *número de estados* x *acciones desde cada estado*), sino que se ha distribuido esta matriz según la misma disposición que en el FrozenLake. Esto corresponde a una matriz 4x4 para el mapa 4x4, y una 8x8 para el mapa 8x8, de manera que sobre cada casilla del lago se muestra su estado correspondiente en la matriz Q.

Las flechas representan las acciones disponibles desde cada estado: moverse hacia arriba, abajo, izquierda o derecha.

El tamaño y color de las flechas tampoco son azarosos: la longitud de la flecha representa la magnitud del valor esperado de esa acción, mientras que el color codifica su signo (negro para los valores positivos y rojo para los negativos). Esto significa que una flecha larga y negra es muy positiva, mientras que una flecha larga y roja es muy negativa.

Sobre cada casilla, además, se muestra un número: este es el máximo valor esperado de ese estado, es decir, el máximo de entre todos los valores esperados de sus acciones, y mide lo bueno o malo que es alcanzarlo. Finalmente, con el progreso del algoritmo, observaremos un círculo que se desplazará por las varias casillas, representando al agente y su posición en la matriz, lo que nos permitirá seguir sus pasos durante el entrenamiento.

A la derecha de nuestro *EntornoWidget* encontraremos una caja de *log*, donde se irán mostrando diferentes mensajes sobre el estado del agente y del entrenamiento.

Como podemos ver en la Figura 19, cada cierto tiempo se muestra la *tasa de éxito*, es decir, la recompensa media obtenida durante los últimos 100 episodios. Esto se debe a que, según la documentación oficial de OpenAIGym, este problema se considera resuelto cuando esta tasa de éxito alcance el valor 0.78, es decir, se ha alcanzado el estado objetivo un 78% de las veces.

Tras finalizar el entrenamiento, se activará el botón *resolver.* Esta opción nos permitirá ejecutar la política **fuera** del entrenamiento, es decir, aplicar lo aprendido para resolver el problema.

Esto significa que no se aprenderá nada nuevo de este tipo de ejecución, sino que simplemente se seguirá al pie de la letra la política calculada durante el entrenamiento.

6.3.2 Menú superior

La interfaz muestra un menú en la barra superior de la aplicación, en la que se distinguen tres secciones. La más sencilla de las tres secciones es la de *Log*, que contiene una única opción llamada *limpiar log* que borrará todo el contenido de la caja de log (Figura 20).

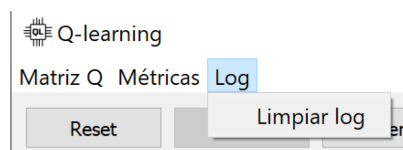

Figura 20 - Sección *Log* del menú superior

A continuación, encontramos el menú *Matriz Q* mostrado en la Figura 21, que contiene las opciones para *importar* y *exportar* la matriz Q. De este modo, tras el entrenamiento, podremos guardar el resultado en un fichero para utilizarlo

posteriormente. Nótese que los mapas de tamaño 4x4 y 8x8 utilizan formatos de fichero distintos (.pol4 y .pol8 respectivamente).

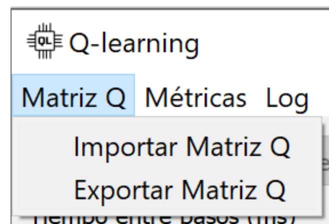

Figura 21 - Sección *Matriz Q* del menú superior

Por último, y tal vez más importante, encontramos la sección de métricas que podemos observar en la Figura 22.

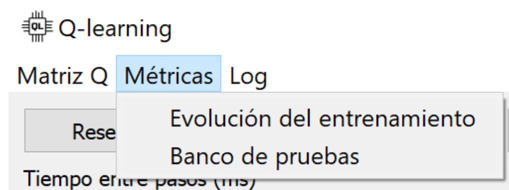

Figura 22 - Sección *Métricas* del menú superior

Esta sección ofrece dos opciones que se detallarán a continuación.

6.3.3 Evolución del entrenamiento

Si, como se ha mencionado anteriormente, la eficacia del entrenamiento se mide por el número de veces que el agente alcanza el objetivo en las 100 iteraciones anteriores, dándose por resuelto el problema cuando este alcanza el 78%, resulta de gran interés poder tener mayor información respecto a cómo va evolucionando esta cifra con el paso de los episodios.

Es por ello que la opción *evolución del entrenamiento* nos mostrará una nueva ventana en la que se mostrará en tiempo real cómo varía este valor (Figura 23). El gráfico se actualiza durante el entrenamiento, y mantiene los datos de otros algoritmos que se hayan entrenado anteriormente, permitiendo una comparación de la eficiencia muy rápida y visual.

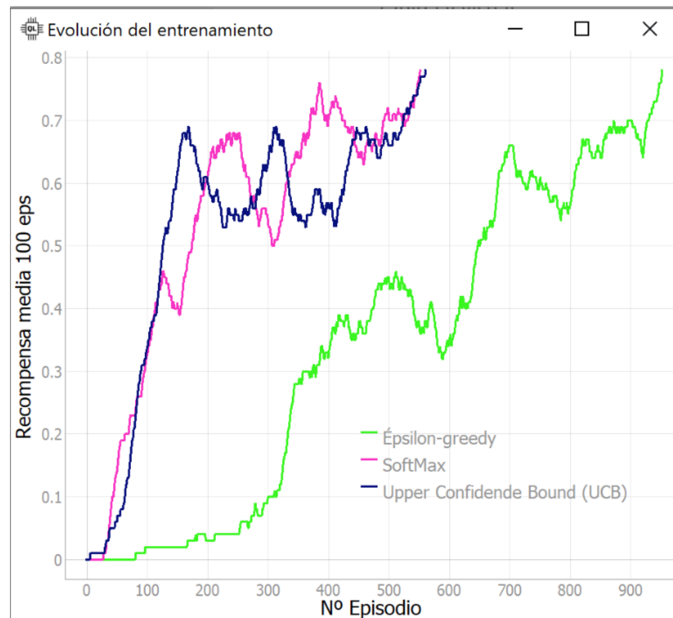

Figura 23 - Ventana de *evolución del entrenamiento*

Es interesante destacar que, pulsando con el botón derecho del ratón sobre el gráfico, podremos desplegar un menú que nos permitirá exportar este gráfico a una gran variedad de formatos para guardarlo en nuestro equipo.

6.3.4 Banco de pruebas

Aunque la inspección de la evolución del entrenamiento puede resultar muy útil, cada ejecución del algoritmo está inevitablemente sujeta a ciertos niveles de aleatoriedad, sobre todo teniendo en cuenta que algunos algoritmos están basados en el uso de números aleatorios, y que el propio entorno es no-determinista. Una métrica más significativa es el número de pasos que el algoritmo ha ejecutado hasta la resolución del problema. Este parámetro es, en definitiva, el que define la eficiencia de un algoritmo. Por ello, se ha implementado una herramienta adicional, denominada *banco de pruebas* (Figura 24), que nos permite probar todos los algoritmos, ejecutando un cierto número de veces cada uno de ellos (10 veces por defecto), y comparando tres métricas específicas:

1. La mejor ejecución, es decir, aquella que ha alcanzado el 0.78 con el menor número de entrenamientos.
2. La peor ejecución, es decir, la que ha necesitado más episodios para alcanzar esta cifra.

3. La media de todas las ejecuciones que, a pesar de estar condicionada a la presencia de valores muy altos o muy bajos (*outlayers*), representa posiblemente la métrica más significativa de las tres.

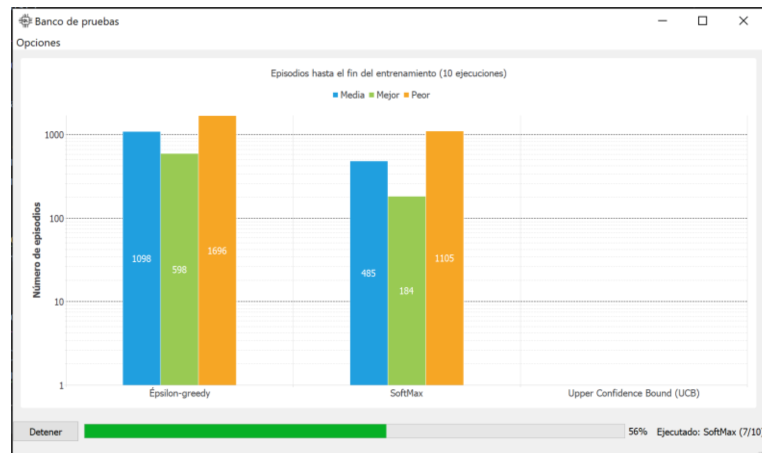

Figura 24 - Ventana de *banco de pruebas*

Este gráfico se irá actualizando en tiempo real según se ejecuten los algoritmos, de modo que la aplicación resulte aún más interactiva en esta fase.

La distribución de esta ventana es sencilla:

encontraremos el gráfico en el centro de la pantalla, y justo debajo de él el panel de controles, compuesto por:

- Un botón de Iniciar/Detener que, como el nombre sugiere, nos permitirá iniciar o detener las pruebas que se realicen.
- Una barra de progreso, que mostrará el progreso total de las pruebas
- Una descripción textual que nos indicará en qué punto de las pruebas nos encontramos (qué algoritmo y qué número de ejecución de este se está ejecutando)

Esta ventana también cuenta con un menú superior, donde encontramos una sección de *opciones* que permite configurar ciertos parámetros, como el número de ejecuciones por algoritmo que se desea ejecutar, o los hiperparámetros de cada variante (- Ventana de *configuración del banco de pruebas*, Figura 25). Esto evita que se comparen algoritmos en condiciones de configuración óptima con algoritmos con una configuración pobre.

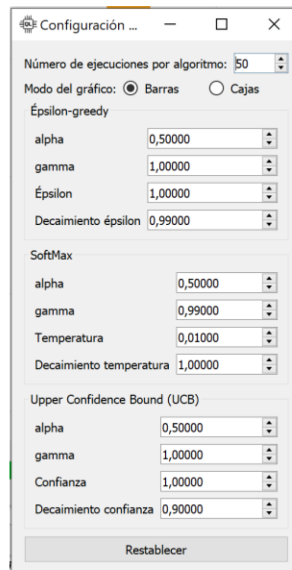

Figura 25 - Ventana de *configuración del banco de pruebas*

Observando la ventana de *configuración del banco de pruebas* se puede apreciar la presencia de dos botones radiales que nos permitirán elegir el modo del gráfico que más nos convenga. Por defecto se mostrará un gráfico de barras como el de la Figura 24, aunque también tenemos la opción de mostrar un gráfico de cajas (llamado también *box and whiskers* o *cajas y bigotes)* como en la Figura 26.

Figura 26 - Ventana de benchmark con gráfico de cajas.

Este tipo de gráfico es más complejo que el de barras, pero nos ofrece mayor información. Se basa en el uso de los *cuartiles,* una medida estadística para determinar la probabilidad de que los datos estén por encima o por debajo de un determinado valor, de modo que hay una probabilidad del 25% de que un dato

sea menor que el primer cuartil, un 50% de que sea menor que el segundo (también llamado *mediana*) y un 75% de que sea menor que el tercero. Las cajas ocupan el espacio entre el primer y el tercer cuartil (llamado *espacio intercuartílico)*, y la barra central representa la mediana. Los "*bigotes*" representan el valor máximo y el mínimo, descartando los valores anormalmente altos o bajos (*outliers*[2]). Esto nos permite, por ejemplo, deducir cuál de los algoritmos tiene menor dispersión en sus medidas (es más "estable"), o visualizar el rango en el que se encuentran la mayoría de datos.

Para mejorar la visualización, ambos gráficos utilizan una escala logarítmica en base 10.

---

[2] Se consideran *outliers* aquellos valores que estén 1.5 veces el rango intercuartílico por encima del tercer cuartil o por debajo del primero ($outlier \geq Q3 + 1.5RIC$ o $outlier \leq Q1 - 1.5RIC$, donde $Q1$ y $Q3$ representan el primer y tercer cuartil, y RIC el Rango InterCuartílico).

7

Resultados obtenidos y conclusiones

El programa detallado en las secciones anteriores se ha utilizado para llevar a cabo una evaluación del rendimiento de los algoritmos analizados, midiendo las métricas principales en términos de número de episodios necesarios para alcanzar la convergencia del algoritmo (fin del entrenamiento), y la evolución del mismo.

En la Figura 27 observamos la evolución del entrenamiento para los tres algoritmos. En esta ocasión, para eliminar la aleatoriedad intrínseca a cada una de las ejecuciones, se han realizado 30 mediciones con cada algoritmo, de manera que lo mostrado en la figura corresponde a la media de las mismas, y se ha fijado un límite superior de 5000 episodios.

Se puede apreciar cómo la curva de UCB es la que crece más rápidamente, seguida por SoftMax y finalmente por $\varepsilon - greedy$, cuya pendiente inicial es considerablemente menor a las otras dos. Encontramos dos datos de interés en esta gráfica:

1. El primero es que tanto $\varepsilon - greedy$ como SoftMax parecen no alcanzar el 0.78, quedándose visiblemente por debajo de UCB, que sí alcanza esta

cifra. Esto se debe a que el valor representado se corresponde con la media de un cierto número de ejecuciones (30 en este caso), y por tanto no se alcanza el valor 0.78 hasta que todas las ejecuciones lo hayan alcanzado (recordemos que, puesto que el entrenamiento finaliza en una tasa de éxito de 0.78, ninguna ejecución puede alcanzar un valor superior). Deducimos por tanto que, si alguna de las ejecuciones no termina antes de los 5000 episodios, esa curva no alcanzará el 0.78 de media. Como corolario, cuanto mayor sea el número de ejecuciones que no consiguen terminar dentro del umbral de las 5000 ejecuciones, menor será el valor alcancazado por la curva: vemos por tanto que en el caso de $\mathcal{E} - greedy$ son menos las ejecuciones que se han completado en comparación con SoftMax o UCB.

2. La curva del algoritmo UCB finaliza antes que las demás (entorno a los 2500 episodios). Esto, al contrario que en el caso anterior, significa que todas las ejecuciones completaron el entrenamiento en menos de aproximadamente 2500 episodios, siendo esta la cantidad empleada por la más lenta de ellas.

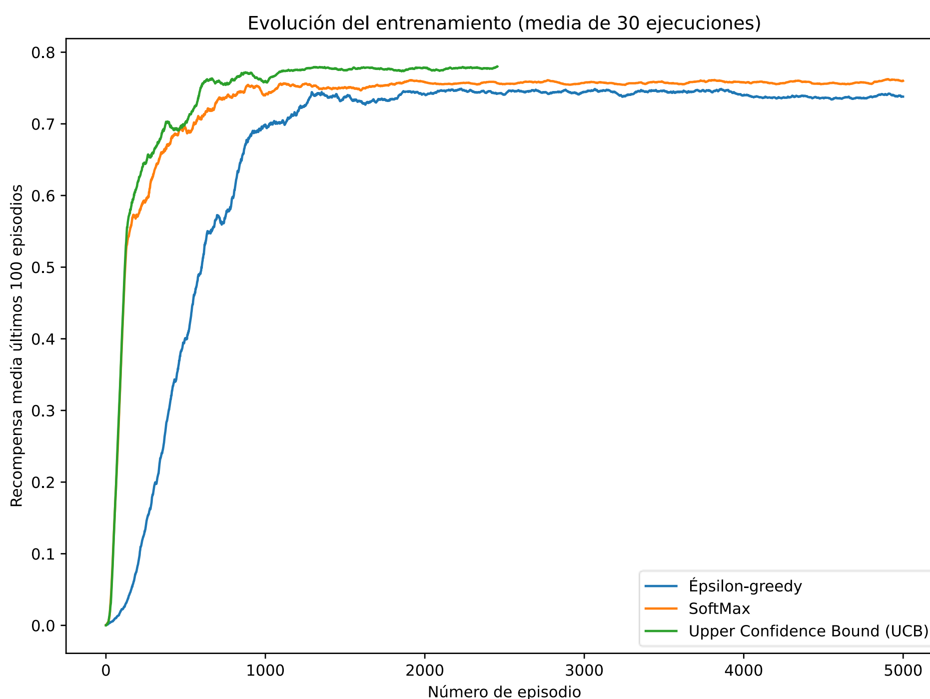

Figura 27 - Evolución del entrenamiento. (30 ejecuciones).

A la vista de estos datos, es seguro afirmar que el algoritmo UCB es el más eficiente de los comparados, siendo el que permite al agente el entrenamiento más rápido. Aunque no alcanza su rendimiento, SoftMax también ofrece resultados prometedores, y netamente superiores a los de $\mathcal{E} - greedy$, que obtiene el peor rendimiento entre los algoritmos seleccionados.

Para investigar ulteriormente la eficiencia de los algoritmos analizados, se ha considerado también el número de episodios que cada uno de los algoritmos ha necesitado para obtener una política óptima (Figura 28). Para limitar aún más el factor de aleatoriedad de los datos, se han utilizado 50 ejecuciones para cada algoritmo.

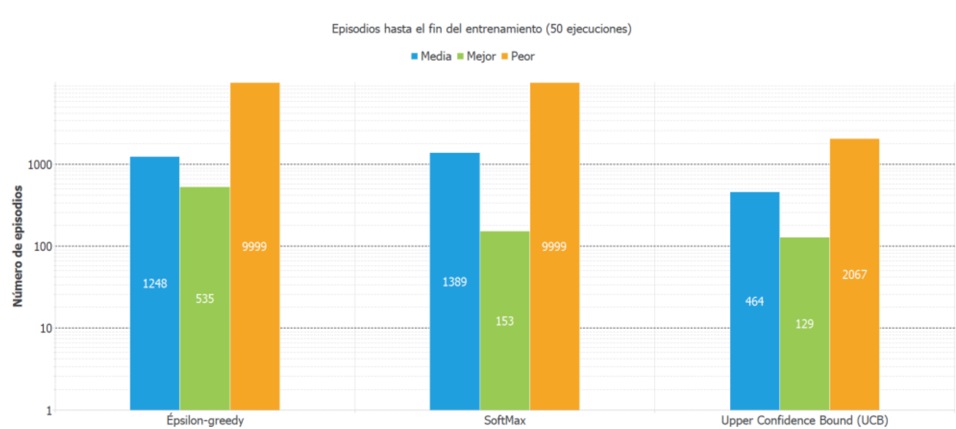

Figura 28 - Pasos hasta el fin del entrenamiento con 50 ejecuciones.

Si bien la información que nos aporta el dato de la *peor ejecución* es poco significativo ya que, en al menos una de las cincuenta ejecuciones, todos los algoritmos han alcanzado el techo superior de las 9999 iteraciones, los datos de la *media* y la *mejor ejecución* se muestran más esclarecedores. Observamos que ambos datos son coherentes en cuanto al orden de rendimiento de los algoritmos:

1. UCB es el que tiene tanto una mejor ejecución (129 episodios) como una media (464 episodios) menores, por lo que podemos concluir que se trata del algoritmo más eficiente entre los seleccionados en el entorno FrozenLake. En comparación con SoftMax ha necesitado un 15.7% menos de episodios en su mejor ejecución, y un **66.6**% menos de media. Respecto a $\mathcal{E} - greedy$, ha recorrido un **75.9% menos** de pasos en su mejor ejecución y un 63.9% menos de media. Vemos como en esta última

comparativa, sobre todo en el caso de la mejor ejecución, la mejora es muy significativa.

2. SoftMax consigue un rendimiento peor que UCB, necesitando incluso más pasos de media que $\varepsilon - greedy$, aunque consigue una mejor ejecución que no se aleja tanto de la de UCB, con una diferencia de aproximadamente 25 episodios (153 frente a 129).
3. Finalmente, $\varepsilon - greedy$ muestra un rendimiento similar, aunque algo mejor, que el de SoftMax, en cuanto a la media. Sin embargo, respecto a la mejor ejecución el rendimiento es notablemente inferior, necesitando más del triple de episodios. Esto puede denotar que, aunque de media los dos algoritmos puedan alcanzar prestaciones similares, SoftMax tiene la capacidad, si se dan las condiciones idóneas, de aprender más rápidamente que $\varepsilon - greedy$.

Podemos obtener más información observando el gráfico de cajas de la Figura 29, donde se muestran datos estadísticos de bastante interés.

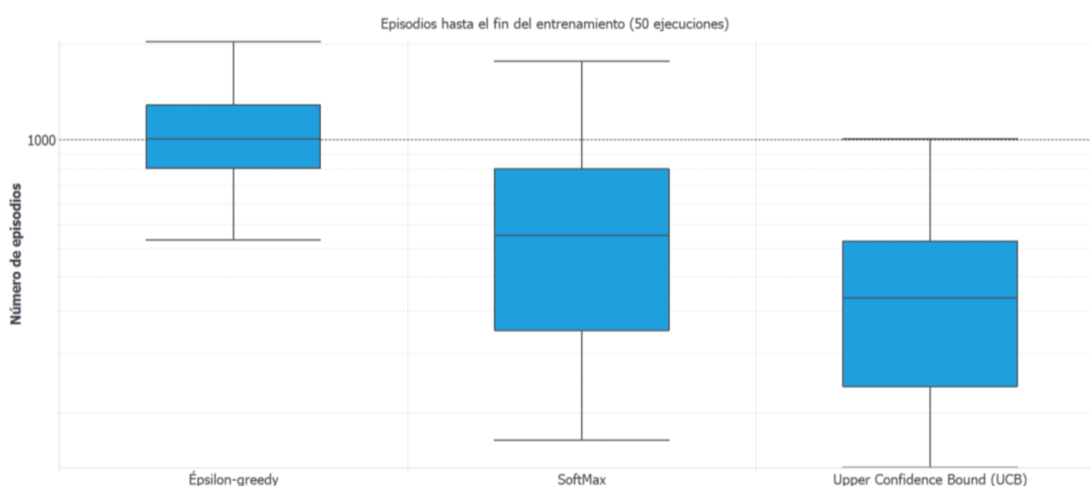

Figura 29 - Número de episodios hasta la convergencia con 100 ejecuciones (gráfico de cajas)

Todos los descriptores coinciden en la superioridad de UCB. La mediana de UCB es notablemente más baja que $\varepsilon - greedy$ y SoftMax, y su dispersión (rango intercuartil – ancho vertical de la caja) es un orden de magnitud menor que la de $\varepsilon - greedy$. Esto significa, no solamente que UCB es más eficiente en general

(la mediana es más baja), sino que la variabilidad de los resultados obtenidos es mucho más baja. En otras palabras, el UCB es un algoritmos más eficiente y más estable. En cuanto a los extremos inferiores, en línea con los resultados anteriores, UCB presenta mínimos mucho más bajos que $\varepsilon - greedy$, y ligeramente más que los de SoftMax. Los extremos superiores reflejan una clara superioridad de UCB frente a ambos antagonistas, presentando valores algo superiores a la altura del 3er cuartil en el caso de SoftMax e incluso a la altura de la mediana en el caso de EG. En términos estadísticos, la probabilidad de que una ejecución necesite un número superior a 1000 episodios para resolver el problema, en UCB es prácticamente nula (coincide con su valor extremo superior), en SoftMax está entorno al 20% (entre el cuarto y el tercer cuartil, más próximo a este último), y en $\varepsilon - greedy$ es de aproximadamente el 50% (mediana). Vemos también que, a pesar de que la media de SoftMax fuera más alta que la de $\varepsilon - greedy$, su mediana es considerablemente más baja, mientras que su límite superior está ligeramente por debajo. Esto muestra cómo la presencia de *outliers* puede influir en la media dando una sensación de peor rendimiento que, atendiendo a mediciones más precisas como la mediana y el límite superior, podemos descartar.

Es importante destacar que la clara superioridad de UCB en este entorno podría verse invertida en otro tipo de entornos cuyas funciones de transición y recompensa o dimensiones varíen significativamente respecto a FrozenLake, por lo que la transferencia de estos resultados es posible, aunque no está garantizada.

Conclusiones

El presente trabajo se ha centrado en tres de las variantes más populares del algoritmo Q-Learning ($\varepsilon - greedy$, SoftMax y UCB), buscando discernir cuál o cuáles ofrecen el mejor rendimiento en el entorno FrozenLake de OpenAIGym, un entorno no determinista en el que el objetivo es recorrer un lago virtual helado de una esquina a otra y sin sufrir ningún percance. Este lago contiene agujeros, que son estados terminales y que proporcionan un castigo, y una casilla objetivo que proporciona una recompensa. El hecho de que el lago esté helado representa el no determinismo de este entorno, de modo que al seleccionar una

acción es posible el agente se "resbale" y acabe en una posición distinta a su objetivo. Por estas características, FrozenLake representa un entorno ideal para experimentar y comparar algoritmos de aprendizaje por refuerzo basados en Q-Learning.

Se ha visto que la variante más eficiente ha resultado ser UCB, que puede necesitar hasta un 76% menos de episodios que otras variantes para completar su entrenamiento, un resultado muy prometedor que reafirma la superioridad de UCB en este tipo de entornos respecto al resto de políticas implementadas.

La estrategia SoftMax, si bien no alcanza el rendimiento de UCB, también ofrece muy buenos resultados, que superan ampliamente a los de $\varepsilon - greedy$ que, a pesar de ser una de las políticas más populares, ofrece el rendimiento más bajo de estas variantes.

Más allá de los resultados experimentales expuestos, este trabajo ofrece una herramienta que permite al usuario explorar de manera sencilla y didáctica el funcionamiento de este algoritmo y realizar en tiempo real las mediciones aquí mostradas, a la par que permite la inclusión de políticas propias para la comparación con las que ya están implementadas.

El software pretende ser una herramienta versátil y potente que, más allá del ámbito investigativo, se extiende también al didáctico, donde estudiantes y profesores se verán beneficiados, facilitando tanto la comprensión de los conceptos teóricos por parte del alumno, gracias a interfaz gráfica del programa, como la transferencia de estos contenidos por parte del docente. Además, la distribución de este programa a través de un canal público, gratuito y de fácil acceso como lo es GitHub, favorece el desarrollo colaborativo, de modo que estudiantes u otros usuarios puedan proporcionar sus propias aportaciones en forma de nuevas políticas o funcionalidades, que conviertan al sistema en una herramienta aún más potente para todos sus usuarios.

Partiendo de este trabajo existen posibles expansiones cuya realización va más allá del alcance del presente proyecto, entre las que se incluyen: el desarrollo e inclusión de nuevas políticas, la posibilidad de ejecutarlas en otro tipo de entornos más allá de FrozenLake o la inclusión de secciones teóricas donde se detalle con mayor precisión qué ocurre durante el entrenamiento.

Referencias

[1] "Q-learning." https://es.wikipedia.org/wiki/Q-learning (accessed May 20, 2021).
[2] "Epsilon-Greedy Algorithm in Reinforcement Learning ." https://www.geeksforgeeks.org/epsilon-greedy-algorithm-in-reinforcement-learning/ (accessed May 20, 2021).
[3] "Softmax function." https://en.wikipedia.org/wiki/Softmax_function (accessed May 20, 2021).
[4] A. D. Tijsma, M. M. Drugan, and M. A. Wiering, "Comparing exploration strategies for Q-learning in random stochastic mazes," Feb. 2017, doi: 10.1109/SSCI.2016.7849366.
[5] "FrozenLake." https://gym.openai.com/envs/FrozenLake8x8-v0/ (accessed May 20, 2021).
[6] "Gym: A toolkit for developing and comparing reinforcement learning algorithms." https://gym.openai.com/ (accessed May 20, 2021).
[7] Guido van Rossum, "Python." 1991, Accessed: May 20, 2021. [Online]. Available: https://www.python.org/.
[8] JetBrains, "Pycharm Community Edition." 2020, Accessed: May 20, 2021. [Online]. Available: https://www.jetbrains.com/es-es/pycharm/.
[9] Riverbank Computing Limited, "PyQt5." 2021.
[10] R. R. Caponera De Cobellis, "elKuston/QLearning: Mi TFG sobre aprendizaje por refuerzo." 2021, Accessed: May 24, 2021. [Online]. Available: https://github.com/elKuston/QLearning.
[11] "Historia de la inteligencia artificial." https://es.wikipedia.org/wiki/Historia_de_la_inteligencia_artificial (accessed May 20, 2021).
[12] A. M. TURING, "I.—COMPUTING MACHINERY AND INTELLIGENCE," *Mind*, vol. LIX, no. 236, pp. 433–460, Oct. 1950, doi: 10.1093/mind/LIX.236.433.
[13] "ELIZA - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/ELIZA (accessed May 20, 2021).
[14] K. M. Colby, F. D. Hilf, S. Weber, and H. C. Kraemer, "Turing-like indistinguishability tests for the validation of a computer simulation of paranoid processes," *Artif. Intell.*, vol. 3, no. C, pp. 199–221, Jan. 1972, doi: 10.1016/0004-3702(72)90049-5.
[15] B. A. Shawar and E. Atwell, "ALICE chatbot: Trials and outputs," *Comput. y Sist.*, vol. 19, no. 4, pp. 625–632, 2015, doi: 10.13053/CyS-19-4-2326.
[16] "The Chinese Room Argument (Stanford Encyclopedia of Philosophy)." https://plato.stanford.edu/entries/chinese-room/#3 (accessed May 20, 2021).
[17] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016, doi: 10.1038/nature16961.

[18] “Go (game) - Wikipedia.” https://en.wikipedia.org/wiki/Go_(game) (accessed May 20, 2021).
[19] “Understanding the Four Types of Artificial Intelligence.” https://www.govtech.com/computing/understanding-the-four-types-of-artificial-intelligence.html (accessed May 26, 2021).
[20] M. Campbell, A. J. Hoane, and F. H. Hsu, “Deep Blue,” *Artif. Intell.*, vol. 134, no. 1–2, pp. 57–83, Jan. 2002, doi: 10.1016/S0004-3702(01)00129-1.
[21] D. Premack and G. Woodruff, “Does the chimpanzee have a theory of mind?,” *Behav. Brain Sci.*, vol. 1, no. 4, pp. 515–526, 1978, doi: 10.1017/S0140525X00076512.
[22] “What are the 3 types of AI? A guide to narrow, general, and super artificial intelligence | Codebots.” https://codebots.com/artificial-intelligence/the-3-types-of-ai-is-the-third-even-possible (accessed May 27, 2021).
[23] “Google AI Blog: Using Machine Learning to Explore Neural Network Architecture.” https://ai.googleblog.com/2017/05/using-machine-learning-to-explore.html (accessed May 27, 2021).
[24] “Google’s AI Built Another AI That Outperforms Man-Made Models.” https://futurism.com/google-artificial-intelligence-built-ai (accessed May 27, 2021).
[25] “Conjunto de datos - Wikipedia, la enciclopedia libre.” https://es.wikipedia.org/wiki/Conjunto_de_datos (accessed May 29, 2021).
[26] “ImageNet.” https://image-net.org/ (accessed May 27, 2021).
[27] “COCO - Common Objects in Context.” https://cocodataset.org/#home (accessed May 27, 2021).
[28] T. B. Brown *et al.*, “Language Models are Few-Shot Learners,” May 2020, Accessed: May 27, 2021. [Online]. Available: http://arxiv.org/abs/2005.14165.
[29] C. Santana Vega, “ PROBANDO a GPT-3 - ¡El CHATBOT más avanzado! - YouTube.” https://www.youtube.com/watch?v=otvqkWFvUZU (accessed May 27, 2021).
[30] C. Santana Vega, “La Siguiente Gran Revolución: NLP (Procesamiento del Lenguaje Natural) - YouTube.” https://www.youtube.com/watch?v=cTQiN9dewIg&t=476s (accessed May 27, 2021).
[31] A. Radford *et al.*, “Learning Transferable Visual Models From Natural Language Supervision,” Feb. 2021, Accessed: May 27, 2021. [Online]. Available: http://arxiv.org/abs/2103.00020.
[32] “Japanese supercomputer is fastest in the world - CNET.” https://www.cnet.com/news/japanese-supercomputer-is-fastest-in-the-world/ (accessed May 28, 2021).
[33] “Fujitsu supercomputer simulates 1 second of brain activity - CNET.” https://www.cnet.com/news/fujitsu-supercomputer-simulates-1-second-of-brain-activity/ (accessed May 27, 2021).
[34] “Skynet (Terminator) - Wikipedia, la enciclopedia libre.” https://es.wikipedia.org/wiki/Skynet_(Terminator) (accessed May 28, 2021).
[35] “Machine Learning (ML) vs. Artificial Intelligence (AI) — Crucial Differences | by Roberto Iriondo | Towards AI.” https://pub.towardsai.net/differences-between-ai-and-machine-learning-and-why-it-matters-1255b182fc6

(accessed May 27, 2021).
[36] "mesa | Definición | Diccionario de la lengua española | RAE - ASALE." https://dle.rae.es/mesa (accessed May 28, 2021).
[37] T. Oladipupo Ayodele, "X Types of Machine Learning Algorithms." Accessed: May 28, 2021. [Online]. Available: www.intechopen.com.
[38] "What Is Machine Learning: Definition, Types, Applications and Examples - Potentia Analytics." https://www.potentiaco.com/what-is-machine-learning-definition-types-applications-and-examples/ (accessed May 28, 2021).
[39] "What are the types of machine learning? | by Hunter Heidenreich | Towards Data Science." https://towardsdatascience.com/what-are-the-types-of-machine-learning-e2b9e5d1756f (accessed May 29, 2021).
[40] "Flash cards - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Flash_cards (accessed May 29, 2021).
[41] "Qué es overfitting y underfitting y cómo solucionarlo | Aprende Machine Learning." https://www.aprendemachinelearning.com/que-es-overfitting-y-underfitting-y-como-solucionarlo/ (accessed May 29, 2021).
[42] "China's booming AI data labeling business has roots in rural farming villages - The Washington Post." https://www.washingtonpost.com/business/2019/09/26/hottest-job-chinas-hinterlands-teaching-ai-tell-truck-turtle/ (accessed May 29, 2021).
[43] "Netflix España - Ver series en línea, ver películas en línea." https://www.netflix.com/es/ (accessed May 29, 2021).
[44] "Escuchar lo es todo - Spotify." https://www.spotify.com/es/ (accessed May 29, 2021).
[45] "Using machine learning to understand customers behavior | by Euge Inzaugarat | Towards Data Science." https://towardsdatascience.com/using-machine-learning-to-understand-customers-behavior-f41b567d3a50 (accessed May 29, 2021).
[46] "Explaining Reinforcement Learning to your next-door-neighbor | by Kartik Chaudhary | Towards Data Science." https://towardsdatascience.com/explaining-reinforcement-learning-to-your-next-door-neighbor-256d2e279f8a (accessed May 29, 2021).
[47] "Breakout (videojuego) - Wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Breakout_(videojuego) (accessed May 29, 2021).
[48] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015, doi: 10.1038/nature14236.
[49] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. 4, no. 2, pp. 100–107, 1968, doi: 10.1109/TSSC.1968.300136.
[50] "Understanding Markov Decision Process (MDP) | by Rohan Jagtap | Towards Data Science." https://towardsdatascience.com/understanding-the-markov-decision-process-mdp-8f838510f150 (accessed Jun. 01, 2021).
[51] "Introduction to Markov chains. Definitions, properties and PageRank… | by Joseph Rocca | Towards Data Science." https://towardsdatascience.com/brief-introduction-to-markov-chains-2c8cab9c98ab (accessed Jun. 01, 2021).

[52] “A Beginners Guide to Q-Learning. Model-Free Reinforcement Learning | by Chathurangi Shyalika | Towards Data Science.” https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c (accessed May 31, 2021).
[53] “On-Policy VS Off-Policy Reinforcement Learning: The Differences.” https://analyticsindiamag.com/reinforcement-learning-policy/ (accessed Jun. 03, 2021).
[54] “Hyperparameter (machine learning) - Wikipedia.” https://en.wikipedia.org/wiki/Hyperparameter_(machine_learning) (accessed Jun. 04, 2021).
[55] “Understanding the role of the discount factor in reinforcement learning - Cross Validated.” https://stats.stackexchange.com/questions/221402/understanding-the-role-of-the-discount-factor-in-reinforcement-learning (accessed Jun. 07, 2021).
[56] “Upper Confidence Bound Algorithm in Reinforcement Learning - GeeksforGeeks.” https://www.geeksforgeeks.org/upper-confidence-bound-algorithm-in-reinforcement-learning/ (accessed Jun. 11, 2021).
[57] “The Upper Confidence Bound (UCB) Bandit Algorithm | by Steve Roberts | Towards Data Science.” https://towardsdatascience.com/the-upper-confidence-bound-ucb-bandit-algorithm-c05c2bf4c13f (accessed Jun. 10, 2021).
[58] “Programación orientada a objetos - Wikipedia, la enciclopedia libre.” https://es.wikipedia.org/wiki/Programación_orientada_a_objetos (accessed Jun. 08, 2021).
[59] “What is UML | Unified Modeling Language.” https://www.uml.org/what-is-uml.htm (accessed Jun. 14, 2021).
[60] “Public, Private, and Protected — Access Modifiers in Python | by Sarath Kaul | Better Programming.” https://betterprogramming.pub/public-private-and-protected-access-modifiers-in-python-9024f4c1dd4 (accessed Jun. 09, 2021).
[61] “MVC Design Pattern - GeeksforGeeks.” https://www.geeksforgeeks.org/mvc-design-pattern/ (accessed Jun. 09, 2021).
[62] “Decorator.” https://refactoring.guru/es/design-patterns/decorator (accessed Jun. 09, 2021).
[63] “Strategy.” https://refactoring.guru/es/design-patterns/strategy (accessed Jun. 09, 2021).
[64] “Template Method.” https://refactoring.guru/es/design-patterns/template-method (accessed Jun. 09, 2021).
[65] “multithreading - What is a race condition? - Stack Overflow.” https://stackoverflow.com/questions/34510/what-is-a-race-condition (accessed Jun. 09, 2021).
[66] “GitHub Desktop | Simple collaboration from your desktop.” https://desktop.github.com/ (accessed Jun. 10, 2021).
[67] “herramienta de código abierto que utiliza descripciones textuales simples para dibujar diagramas UML.” https://plantuml.com/es/ (accessed Jun. 09, 2021).
[68] “Drakemor/RedDress-PlantUML: Stylesheets for PlantUML.” https://github.com/Drakemor/RedDress-PlantUML (accessed Jun. 09,

2021).

Apéndice A

Manual de Instalación

Requerimientos:

Puesto que este software está desarrollado utilizando Python 3, será necesario tener esta tecnología instalada en el equipo, que puede ser descargada de manera gratuita a través de su página oficial [7].

Instalación y ejecución

Una vez se dispone de Python 3 en el equipo, la instalación y ejecución del software será muy sencilla. El software se distribuye a través de GitHub [10] de manera gratuita bajo licencia GPLv3. Deberemos pulsar en *code* (Figura 30) y bien en *Download ZIP* para descargarlo, o bien clonarlo a través de GitHub Desktop[66] (o cualquier cliente de Git).

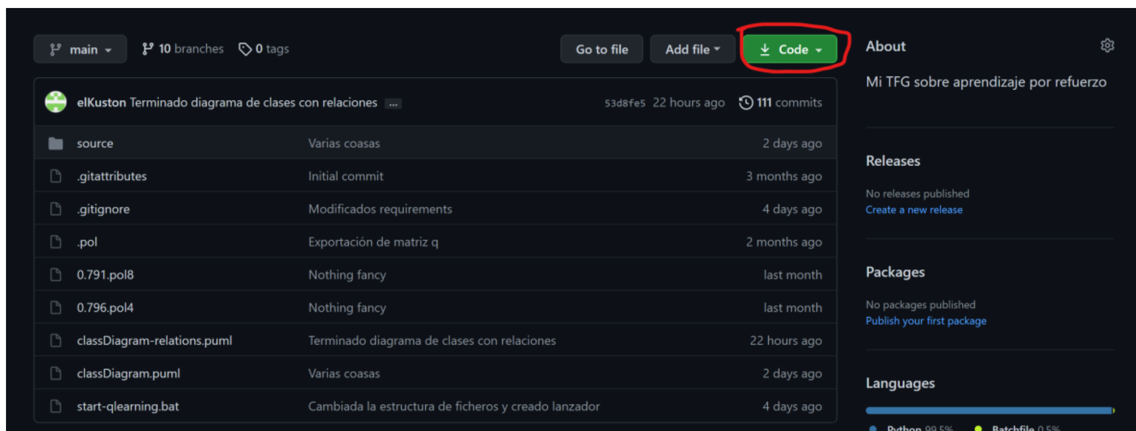

Figura 30 - Vista de la rama *main* del proyecto en GitHub

Una vez tengamos acceso al software en nuestro equipo (ya sea clonando el repositorio o extrayendo el ZIP), nos encontraremos con un directorio *source* y

un archivo *start-qlearning.bat* (Figura 31) junto con otros ficheros que pueden ser ignorados. En el directorio *source* encontraremos todo el código fuente del programa, que podrá ser abierto y modificado por los usuarios que así lo deseen. Por otro lado, para instalar e iniciar el programa, bastará con abrir el fichero *start-qlearning.bat*. En la primera ejecución, se instalarán todas las dependencias necesarias en un entorno virtual de Python llamado *qlearningEnv* en la misma carpeta en la que se encuentra el fichero, y a continuación se iniciará la aplicación. Este es un proceso que podría tardar unos minutos en función del equipo y la conexión a internet, y que no se realizará en ejecuciones posteriores.

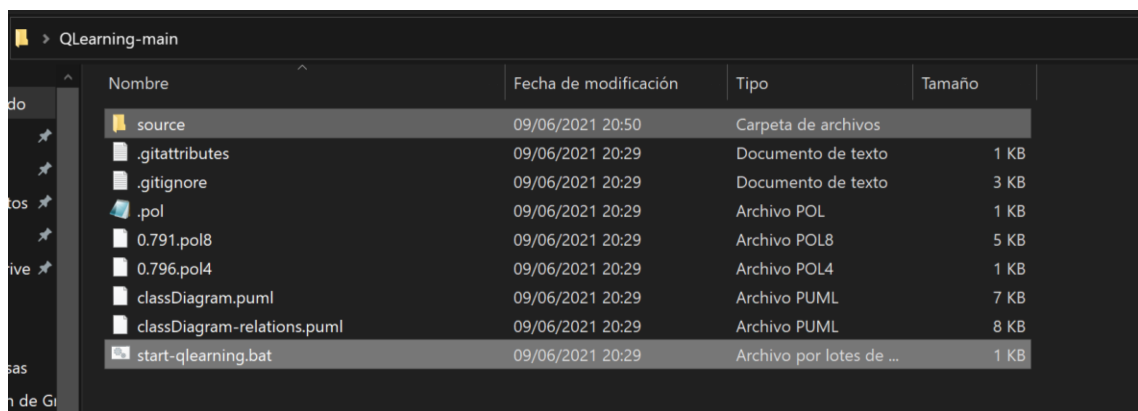

Figura 31 - Directorio con los ficheros de la aplicación

Tras este proceso se iniciará la aplicación y el usuario podrá utilizarla con libertad.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA

E.T.S de Ingeniería Informática

Bulevar Louis Pasteur, 35

Campus de Teatinos

29071 Málaga