Fundamentos matemáticos del aprendizaje reforzado: Q-learning

Miguel Ángel Luquín Guerrero

Trabajo de fin de grado de Matemáticas
Universidad de Zaragoza

Junio de 2025

Abstract

Reinforcement learning (RL) stands as one of the most mathematically rich and practically impactful branches of machine learning. Unlike *supervised* or *unsupervised learning*, RL is characterized by the interaction of an agent with an environment, where learning is driven by the pursuit of long-term rewards through trial and error. This Final Degree Project is motivated by the desire to rigorously understand the mathematical foundations that underpin RL algorithms, with a particular emphasis on the *Q-learning* paradigm and its extensions.

The work begins by introducing the reader to the essential components of RL through accessible examples, such as the "Frozen Lake" environment, which serves as an intuitive gateway to the formalization of *Markov Decision Processes (MDPs)*. The formal framework is developed step by step: *states*, *actions*, *transition probabilities*, *rewards*, and *policies* are defined with mathematical precision, and their interplay is analyzed through the lens of stochastic processes and dynamic programming.

Central to the document is the study of *value functions* and the *Bellman equations*, which provide the backbone for both theoretical analysis and algorithmic design in RL. The existence and uniqueness of optimal policies are established, and the recursive structure of the Bellman equations is leveraged to motivate iterative solution methods. The Q-learning algorithm is then introduced as a *model-free*, *off-policy* approach to learning optimal action-value functions. Its convergence properties are discussed in detail, including the influence of *learning rates*, *discount factors*, and exploration strategies such as ε*-greedy policies*.

A significant portion of the project is devoted to the practical aspects of Q-learning. The interaction between hyperparameters is explored through systematic experimentation in environments like "Frozen Lake", highlighting the delicate balance required for efficient learning.

The work extends its scope to more complex environments, such as "Tic-Tac-Toe", where the state and action spaces grow combinatorially, and classical tabular methods encounter scalability challenges. Here, the discussion naturally transitions to *Deep Q-learning*, where function approximation via *neural networks* is introduced. The mathematical challenges of stability, *overfitting*, and generalization in deep RL are discussed, connecting classical theory to modern deep learning techniques.

Throughout the work, definitions and theorems are presented with clear intuition, followed by formal statements and illustrative examples. Where appropriate, proofs are included to solidify understanding, and connections to recent literature are highlighted to situate the project within the broader context of RL research. It aims to provide not only a solid theoretical foundation but also practical insights that are directly applicable to the design and analysis of RL algorithms in contemporary research and applications.

Índice general

Capítulo 1

Introducción

El aprendizaje por refuerzo (Reinforcement Learning, RL) es una rama del aprendizaje automático que ha cobrado gran protagonismo en la última década debido a su capacidad para resolver problemas donde la toma de decisiones secuenciales es clave. A diferencia del aprendizaje supervisado, en el que se aprende a partir de datos etiquetados, el aprendizaje por refuerzo se basa en la interacción de un agente con un entorno, guiado por un sistema de recompensas, con el objetivo de maximizar una ganancia acumulada a largo plazo.

Uno de los primeros ejemplos físicos de aprendizaje por refuerzo es **MENACE** (*Matchbox Educable Noughts and Crosses Engine*) [12], un dispositivo construido en 1961 por Donald Michie para jugar al tres en raya. MENACE utilizaba 304 cajas de cerillas para representar diferentes estados del juego, y bolas de colores en su interior para seleccionar jugadas. Aprendía reforzando las jugadas exitosas (añadiendo bolas tras una victoria) y penalizando las fallidas (eliminándolas tras una derrota). Aunque rudimentario, este sistema anticipaba los principios básicos del aprendizaje por refuerzo moderno, demostrando cómo un agente puede mejorar su estrategia a partir de la interacción con el entorno y la retroalimentación basada en recompensas.

En esta memoria se estudian los fundamentos matemáticos que sustentan este tipo de aprendizaje, con especial atención al algoritmo Q-learning, uno de los métodos más conocidos y utilizados dentro del RL. El trabajo se estructura en torno al desarrollo formal del marco de los Procesos de Decisión de Markov (MDP), la formulación de las ecuaciones de Bellman, y la derivación del algoritmo de Q-learning como una técnica off-policy y modelo-libre. A través de ejemplos progresivamente más complejos, desde el entorno simplificado de "Frozen Lake" hasta juegos como el tres en raya y problemas clásicos como "CartPole", se analiza el funcionamiento, las ventajas y las limitaciones de Q-learning, incluyendo su extensión a entornos de alta dimensión mediante redes neuronales, lo que se conoce como Deep Q-learning. El código de todos estos programas está disponible en mi repositorio público de GitHub en [10].

Aunque el Q-learning se popularizó en entornos de videojuegos, como por ejemplo en el entrenamiento de agentes que aprenden a jugar a Atari o a resolver puzzles lúdicos, su aplicabilidad va mucho más allá. En robótica, por ejemplo, se utiliza para que robots móviles aprendan a navegar en entornos dinámicos. En logística, puede optimizar rutas de reparto bajo incertidumbre. En finanzas, se emplea para diseñar estrategias de inversión adaptativas. También ha sido usado en áreas tan diversas como el control de tráfico en redes de comunicaciones, la gestión de energía en redes inteligentes (smart grids), o la planificación de tratamientos personalizados en medicina.

A lo largo de este trabajo se busca mantener un equilibrio entre el rigor matemático y la aplicabilidad práctica, introduciendo conceptos clave con claridad, y sustentando los algoritmos con análisis formales y simulaciones computacionales. La meta final es ofrecer una comprensión profunda y estructurada de cómo y por qué Q-learning funciona.

Finalmente, quiero expresar mi más sincero agradecimiento a mis tutores Álvaro Lozano y Rubén Vigara por su dedicación y orientación a lo largo del desarrollo de este trabajo. Su apoyo ha sido fundamental tanto en el aspecto técnico como en la organización general del proyecto.

Capítulo 2

Bases del Aprendizaje Reforzado, Marco Formal y Ecuaciones de Bellman

El aprendizaje reforzado, o aprendizaje por refuerzo, es la rama del aprendizaje automático (Machine Learning) en la que se quiere entrenar a un agente/programa para que realice una tarea concreta, con la particularidad de que no se le enseña explícitamente cómo hacer dicha tarea. Para entender los componentes fundamentales que actúan en el aprendizaje reforzado vamos a introducir el ejemplo más popular del aprendizaje por refuerzo y despúes formalizaremos algunas definiciones.

2.1. Lago helado, parte I

Partimos de un lago de hielo, discretizado como un tablero de tamaño 3×3, este será nuestro **entorno**. Tenemos un pingüino que será nuestro **agente**. La posición del animal en el tablero es el **estado** (s) en el que se encuentra. De esta forma, nuestro **conjunto de estados** son todas las casillas del tablero. En nuestro modelo, el tiempo no fluye de manera continua, sino que está dividido en **pasos** ($t = 0, 1, 2 \ldots$). En cada paso, el pingüino debe moverse una casilla, es decir, debe realizar una **acción** (a) para llegar al **estado siguiente** (s'). En particular, puede moverse arriba, abajo, a la izquierda o a la derecha (**conjunto de acciones**). Como el tablero está helado, no siempre que quiera moverse en una dirección lo hará, sino que el $10\,\%$ de las veces, se moverá en una de las otras direcciones ($\mathbb{P}(error) = 0.1$). El pingüino comienza siempre en la casilla $(0,0)$, el **estado inicial** (s_0). Su objetivo es tan simple como llegar a la otra esquina del tablero, la casilla $(2,2)$. Si lo logra, le daremos 1 punto como **recompensa**. Si se sale del tablero, la recompensa será de -1 puntos (Figura 1.1).

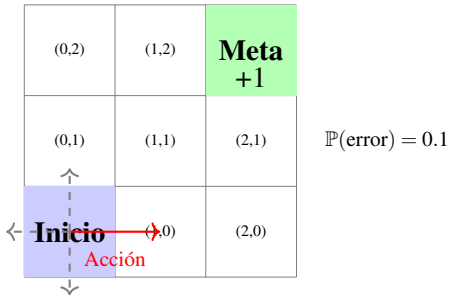


Figura 2.1: Tablero de hielo 3×3 con estado inicial en $(0,0)$ y objetivo en $(2,2)$. Las flechas muestran posibles acciones. Existe una probabilidad del $10\,\%$ de no ir en la dirección deseada.

Abordando este problema con un algoritmo de aprendizaje reforzado, la figura inicialmente no sabe lo que tiene que hacer. El **entrenamiento** consiste en ir probando acciones aleatoriamente (**explorar**) hasta encontrar una sucesión de acciones con recompensa positiva. En ese momento, comienza un balance entre probar nuevas acciones para encontrar una mejor solución (**maximizar la recompensa futura**)

y usar los conocimientos adquiridos para verificar que la solución ofrece, en efecto, una recompensa positiva (**explotar**). Una vez terminado el entrenamiento, obtenemos una **política** (π) que dado un estado, elige una acción para pasar al siguiente estado.

Ahora que hemos introducido los términos que vamos a manejar, vamos a presentar el marco formal que se emplea en el aprendizaje reforzado. Este está fundamentado en los conceptos y enfoques desarrollados en [2]. Este marco incluye la definición de los elementos esenciales, como estados, acciones, recompensas y políticas, y sus interacciones dentro de los Procesos de Decisión de Markov. Las ideas principales se adaptarán y ampliarán para contextualizarlas en el ámbito de este trabajo.

2.2. Marco Formal

Matemáticamente, los algoritmos de aprendizaje reforzado suelen modelarse mediante *Procesos de Decisión de Markov*. En este estudio nos restringiremos al caso finito.

Definición 2.1. Un Proceso de Decisión de Markov (MDP, por sus siglas en inglés) es una tupla (S,A,T,R) donde S y A son conjuntos finitos cuyos elementos son los estados y acciones del proceso respectivamente. Además,

- $T : S \times A \times S \to [0,1]$ con $T(s,a,s') = \mathbb{P}(s' \mid s,a)$, la función de mide la probabilidad de que la acción a en el estado s conduzca al estado s', y
- $R : S \times A \times S \to \mathbb{R}$ la función de recompensa por pasar del estado s a s' mediante la acción a.

Definición 2.2. Dado un MDP (S,A,T,R), un *episodio* es una sucesión de pasos temporales:

$$(s_0, a_0, s_1, r_1, a_1, s_2, r_2, \ldots, s_{F-1}, r_{F-1}, a_{F-1}, s_F, r_F),$$

donde:

- s_0 es el estado inicial,
- a_t es la acción tomada en el paso t,
- $r_{t+1} = R(s_t, a_t, s_{t+1})$ es la recompensa inmediata,
- F es el tiempo de finalización del episodio.

El episodio termina cuando el agente alcanza un estado terminal o se cumple una condición de parada (por ejemplo, un número máximo de pasos). Si existe una sucesión de acciones que se pueden repetir indefinidamente sin alcanzar un estado final, se denomina modelo de *horizonte infinito*.

Un MDP cumple la *propiedad de Markov*, es decir, la probabilidad de transición y las recompensas a un estado futuro $s' \in S$ dependen únicamente del estado y acción actuales, y no del historial previo. En símbolos, para todo $t \in \mathbb{N}$, $s_i \in S$ y $a_i \in A$,

$$\begin{aligned}
&\mathbb{P}(s_{t+1} \mid s_t, a_t, s_{t-1}, a_{t-1}, \ldots) = \mathbb{P}(s_{t+1} \mid s_t, a_t) = T(s_t, a_t, s_{t+1}) \\
&R(s_t, a_t, s_{t+1} \mid s_{t-1}, a_{t-1}, \ldots) = R(s_t, a_t, s_{t+1}).
\end{aligned}$$

Definición 2.3. Dado un MDP (S,A,T,R), una *política determinista* es una función $\pi : S \to A$ que a cada estado $s \in S$ le asigna una acción $a \in A$. Denotamos Π al conjunto de todas las políticas posibles.

Una política (en teoría de juegos se suele denominar *estrategia*) indica cómo proceder en cada estado. Desde un estado inicial $s_0 \in S$, el agente realiza la acción $\pi(s_0) = a_0 \in A$. Según la función de transición T, el agente llega al estado s_1 con probabilidad $T(s_0, a_0, s_1)$ y obtiene una recompensa $r_0 = R(s_0, a_0, s_1)$. Este proceso se repite hasta que el agente alcanza uno de los estados finales s_{goal} si lo hay, o en su defecto, continúa indefinidamente.

Ejemplo 2.4. Una política en el ejemplo del pingüino del principio del capítulo sería moverse siempre a la derecha: $\pi(s) = derecha\ \forall s \in S$. Es fácil comprobar que esta política no es la mejor si queremos llegar a la otra esquina del tablero helado.

Definición 2.5. Se define *política óptima* en un MDP (S,A,T,R) como la función $\pi^* \in \Pi$ que maximice el valor esperado de la recompensa acumulada futura[1].

Si el agente solo se preocupara por la recompensa inmediata, un criterio simple de optimalidad sería optimizar la esperanza $\mathbb{E}[r_t]$ siendo $r_t = R(s_t, a_t, s_{t+1})$. Este criterio no se usa puesto que ignora el impacto de las acciones en estados futuros. Queremos maximizar la recompensa acumulada y esto puede implicar en muchos casos tomar acciones con menor recompensa inmediata pero que a largo plazo sean mayores. En su defecto, el criterio de optimalidad que definiremos, y el usado en algoritmos de Q-learning, es el siguiente:

Definición 2.6. Dado un MDP (S,A,T,R) y un factor de descuento $\gamma \in (0,1)$, el criterio de optimalidad usado para modelos con horizonte infinito es

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right], \tag{2.1}$$

siendo $r_t = R(s_t, a_t, s_{t+1})$ la recompensa obtenida en el paso $t \in \mathbb{N}$. Además, dada una política π, se define $\mathbb{E}_\pi$ como la esperanza condicionada a seguir la política π para todas las acciones de un episodio, es decir,

$$\mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \middle| a_t = \pi(s_t)\right].$$

El *factor de descuento* γ es un parámetro que controla cuánto valoramos las recompensas futuras en comparación con las recompensas inmediatas.

Además, si $R : S \times A \times S \to [-M, M]$ con $M < \infty$ (las recompensas están acotadas), garantizamos que la esperanza de la suma en (2.1) converge y podemos así implementar algoritmos para encontrar ese máximo, que veremos más adelante.

Teorema 2.7. *El criterio de optimalidad* (2.1) *converge* $\forall \gamma \in (0,1)$ *si las recompensas del MDP asociado están acotadas.*

Demostración. Basta con demostrar que el sumatorio converge:

$$\sum_{t=0}^{\infty} \gamma^t r_t \leq \sum_{t=0}^{\infty} \gamma^t |r_t| \leq \sum_{t=0}^{\infty} \gamma^t M = M \sum_{t=0}^{\infty} \gamma^t = M \frac{1}{1-\gamma} < \infty. \qquad \square$$

2.3. Funciones de Valor y las Ecuaciones de Bellman

La mayoría de algoritmos de aprendizaje reforzado calculan las políticas óptimas mediante el aprendizaje de *funciones de valor*. Estas dan un valor real a cada estado, de modo que cuanto mayor sea este valor, mejor es estar en ese estado comparado con el resto.

Definición 2.8. Dado un MDP (S,A,T,R) y una política cualquiera $\pi \in \Pi$, se define el valor de un estado con el criterio de optimalidad (2.1) como la función $V^\pi : S \to \mathbb{R}$ tal que:

$$V^\pi(s) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_k \middle| s_0 = s\right]. \tag{2.2}$$

[1] Es decir, queremos encontrar las acciones que debe realizar el agente para que el valor de la suma de todas las recompensas que obtiene en el proceso sea máxima. Dicho de manera coloquial, queremos que el agente sea lo mejor posible en la tarea para la que lo estamos entrenando.

Lo que se queremos es encontrar la política óptima π^* que maximice la función de valor para cada estado $s \in S$. Una función que asigna un valor a cada par (estado, acción) y que es similar a la anterior, es la siguiente:

Definición 2.9. Sea (S,A,T,R) un MDP, se define el retorno esperado a partir del estado $s \in S$ tomando la acción $a \in A$ y después siguiendo la política cualquiera $\pi \in \Pi$ con el criterio de optimalidad (2.1) como la función $Q^\pi : S \times A \to \mathbb{R}$ tal que:

$$Q^\pi(s,a) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_k \middle| s_0 = s, a_0 = a \right].$$

Además, con esta definición $Q^\pi(s,\pi(s)) = V^\pi(s)$.

En los algoritmos de Q-learning, es esta la función que tratamos de maximizar, esto se verá más en detalle en la siguiente sección. Por el momento, vamos a redefinir la ecuación (2.2) de forma recursiva. Dado un MDP (S,A,T,R) y una política cualquiera $\pi \in \Pi$. Tenemos que

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi \left[r_0 + \gamma r_1 + \gamma^2 r_2 + \dots \middle| s_0 = s \right] \\ &= \mathbb{E}_\pi \left[r_0 + \gamma V^\pi(s_1) \middle| s_0 = s \right] \\ &= \sum_{s' \in S} T(s,\pi(s),s') \Big(R(s,\pi(s),s') + \gamma V^\pi(s') \Big). \end{aligned} \tag{2.3}$$

Operando de manera similar para $Q^\pi(s,a)$ tenemos que

$$\begin{aligned} Q^\pi(s,a) &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_k \middle| s_0 = s, a_0 = a \right] \\ &= \sum_{s' \in S} T(s,a,s') \Big(R(s,a,s') + \gamma V^\pi(s') \Big). \end{aligned} \tag{2.4}$$

Definición 2.10. La expresión (2.3) se conoce como *Ecuación de Bellman* y la expresión (2.4) como *Ecuación de Bellman para valores de estado-acción*. Esto denota que el valor esperado de cada (estado, acción) se define en términos de la recompensa inmediata dada la función de recompensa R, los valores de los posibles estados siguientes ponderados por su función de transición T y el factor de descuento γ.

Ahora bien, para encontrar la política óptima π^*, queremos maximizar el valor de la ecuación (2.2) para cada estado $s \in S$. Es decir, encontrar la función $\pi^* \in \Pi$ tal que $V^{\pi^*}(s) \geq V^\pi(s)\ \forall s \in S, \forall \pi \in \Pi$.

Observación. Cambiaremos la notación de V^{π^*} para la política óptima π^* a la más usada V^*.

Teorema 2.11 (Existencia de política óptima)**.** *Dado un MDP (S,A,T,R) con S y A finitos, recompensas acotadas y un factor de descuento $\gamma \in (0,1)$, existe una política óptima $\pi^* : S \to A$ que para cualquier estado inicial $s_0 \in S$ maximiza la recompensa esperada*

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \right],$$

y cuya función de valor V^* verifica la ecuación de optimalidad de Bellman:

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} T(s,a,s') \left(R(s,a,s') + \gamma V^*(s') \right), \quad \forall s \in S. \tag{2.5}$$

Además, la política óptima puede definirse como:

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in S} T(s,a,s') \left(R(s,a,s') + \gamma V^*(s') \right).$$

Una demostración puede consultarse en [15, Teorema 8.4.5, Página 361].

Observación. En la literatura [1], es también habitual denotar a π^* por $\pi_{greedy}(V)$ siendo V la función de valor que maximiza.

Podemos hacer algo similar para definir la política óptima de la *Ecuación de Bellman para valores de estado-acción* y cuya demostración reside en usar el teorema anterior y (2.4).

$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left(R(s,a,s') + \gamma \max_{a'} Q^*(s',a') \right).$$

A Q^* se la conoce como función Q, y como cabría esperar, es el pilar fundamental de los algoritmos de Q-learning. Para elegir la política óptima, basta con encontrar la función $\pi^* : S \to A$ tal que

$$\pi^*(s) = \arg\max_a Q^*(s,a),$$

para cada estado $s \in S$.

2.4. Lago helado, parte II

La figura 1.2 muestra las 4 estrategias óptimas para el juego del pingüino y el tablero helado. Se calculan numéricamente en Anexo I y también lo resolveremos en el siguiente capítulo usando Q-learning.

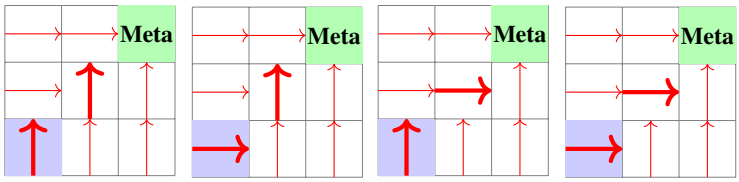
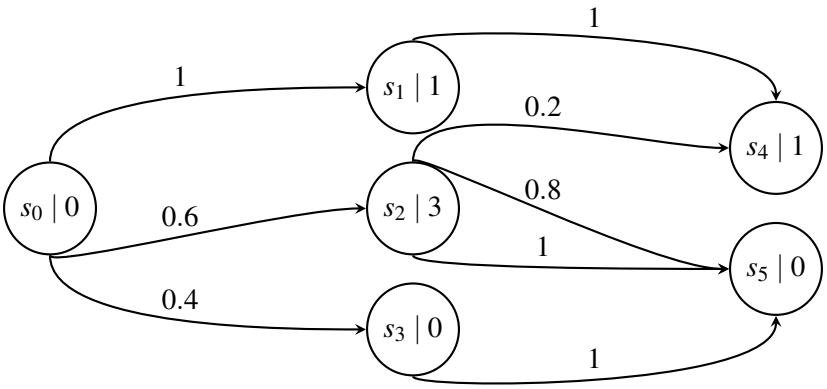


Figura 2.2: Cuatro tableros representando las 4 estrategias óptimas. Las flechas rojas indican el movimiento desde cada estado. Se remarcan las flechas que son diferentes en cada política.

Para cerrar el capítulo y entender mejor los conceptos de estado, funciones de transición, funciones de recompensa y funciones de valor vamos con otro ejemplo sencillo.

Ejemplo 2.12. Este grafo dirigido muestra seis estados s_i, uno en cada nodo. Las recompensas se obtienen al entrar a un nuevo nodo, $r_i = R(s,a,s') = R(s')$. Las acciones para cada estado se representan con *arriba* y *abajo* (de donde sale la arista), y $T(s,a,s')$ es la probabilidad de ir al estado s' tras tomar la acción a en el estado s y su valor está situado encima de cada arista. Partiendo de s_0, y con un factor de descuento de $\gamma = 0.9$ queremos hallar la política óptima $\pi^*(s)$ mediante el cálculo recursivo de $V^*(s)$.



Solución. Usando la ecucación (2.5):

$$V^*(s_4) = 0, V^*(s_5) = 0.$$

Puesto que para esos estados no hay acciones posibles. Ahora calculemos $V^*(s_1)$ y $V^*(s_3)$:

$$\begin{aligned}
V^*(s_1) &= \max_{a\in A} \sum_{s'\in S} T(s_1,a,s') \left(R(s_1,a,s') + \gamma V^*(s')\right) \\
&= \max \left\{ T(s_1, arriba, s_4) \left(R(s_1, arriba, s_4) + \gamma V^*(s_4)\right) \right\} \\
&= \max \left\{ 1 \cdot (1 + 0.9 \cdot 0) \right\} = 1, \\
V^*(s_3) &= \max \left\{ T(s_3, abajo, s_5) \left(R(s_3, abajo, s_5) + \gamma V^*(s_5)\right) \right\} = 0.
\end{aligned}$$

Como solo hay una acción posible para cada estado, esa es la óptima. Ahora calculemos $V^*(s_2)$:

$$\begin{aligned}
V^*(s_2) &= \max_{a\in A} \sum_{s'\in S} T(s_2,a,s') \left(R(s_2,a,s') + \gamma V^*(s')\right) \\
&= \max \left\{ \left(T(s_2, arriba, s_4) \left(R(s_2, arriba, s_4) + \gamma V^*(s_4)\right) + \right.\right. \\
&\qquad \left. T(s_2, arriba, s_5) \left(R(s_2, arriba, s_5) + \gamma V^*(s_5)\right) \right), \\
&\qquad \left. \left(T(s_2, abajo, s_5) \left(R(s_2, abajo, s_5) + \gamma V^*(s_5)\right) \right) \right\} \\
&= \max \left\{ \left(0.2 \cdot (1 + 0.9 \cdot 0) + 0.8 \cdot (0 + 0.9 \cdot 0) \right), \left(1 \cdot (0 + 0.9 \cdot 0) \right) \right\} = 0.2.
\end{aligned}$$

Donde el máximo se alcanza tomando $a = arriba$. Por último en s_0:

$$\begin{aligned}
V^*(s_0) &= \max_{a\in A} \sum_{s'\in S} T(s_0,a,s') \left(R(s_0,a,s') + \gamma V^*(s')\right) \\
&= \max \left\{ \left(0.6 \cdot (3 + 0.9 \cdot 0.2) + 0.4 \cdot (0 + 0.9 \cdot 0) \right), \left(1 \cdot (1 + 0.9 \cdot 1) \right) \right\} = 1.908.
\end{aligned}$$

Habiéndose obtenido el máximo tomando la acción $a = abajo$.

Juntando todos los resultados, tenemos que la política óptima $\pi^*(s)$ para los estados con más de una acción posible es la siguiente:

$$\begin{aligned}
\pi^*(s_0) &= abajo, \\
\pi^*(s_2) &= arriba.
\end{aligned}$$

Observación. Aunque en este ejemplo no se haya visto la importancia del factor de descuento γ, este cobra importancia cuando el grafo de estados es más complejo y no está definido en capas. Con el ejemplo se buscaba mostrar el funcionamiento de elegir la política óptima según los valores de estado.

Capítulo 3

Algoritmo de Q-learning

Ahora que tenemos una noción general de los elementos que sostienen la teoría del aprendizaje reforzado, vamos a describir el algoritmo que concierne a este trabajo: el Q-learning.

Observación. En la práctica vamos a modelar la función Q con una *matriz Q* donde la entrada i, j corresponde con el estado i y la acción j. Esto es posible ya que S y A son finitos.

Definición 3.1. Sea (S,A,T,R) un MDP. Un algoritmo de aprendizaje por refuerzo se denomina *modelo-libre* si aprende una función de valor $Q(s,a)$ sin estimar $T(s'|s,a)$ o $R(s,a)$ para planificar o predecir el resultado de las acciones. El aprendizaje se realiza únicamente a partir de secuencias observadas de estados, acciones y recompensas, mediante interacción directa con el entorno.

Definición 3.2. Dado un MDP (S,A,T,R), sea π una política cualquiera y π^* la política óptima. Un algoritmo de aprendizaje por refuerzo es *off-policy* si estima la función de valor $Q^*(s,a)$ utilizando muestras generadas bajo π, es decir, **sin requerir que la política de comportamiento coincida con la política objetivo** ($\pi \neq \pi^*$ en general). Esto permite aprender sobre una política mientras se explora el entorno con otra distinta.

El Q-learning es un algoritmo de aprendizaje por refuerzo, de tipo off-policy y modelo-libre, diseñado para aprender la política óptima en un MDP mediante la estimación iterativa de los valores de acción-estado $Q(s,a)$. Su objetivo es encontrar una política que maximice la recompensa acumulada esperada, sin requerir conocimiento previo de la dinámica del entorno. Más formalmente:

Definición 3.3. Sea (S,A,T,R) un MDP y $\gamma \in (0,1)$ un factor de descuento. El algoritmo Q-learning actualiza iterativamente la función de valor acción-estado $Q : S \times A \to \mathbb{R}$ mediante

$$Q(s,a) \leftarrow Q(s,a) + \alpha \Big[r + \gamma \max_{a' \in A} Q(s',a') - Q(s,a) \Big], \tag{3.1}$$

donde:

- $\alpha \in (0,1]$ es la tasa de aprendizaje,
- $r = R(s,a,s')$ es la recompensa inmediata obtenida,
- $s' \sim T(\cdot|s,a)$ es el siguiente estado.

Observación. Se suele usar una tasa de aprendizaje que varía en función de los episodios y que se denota α_n para el episodio n.

3.1. Entrenamiento del agente mediante Q-learning

En Q-learning, el entrenamiento del agente se puede describir con el siguiente algoritmo:

Algoritmo 1: Entrenamiento de un agente mediante Q-learning

Data: Espacio de estados S, espacio de acciones A, número de episodios N, factor de descuento $\gamma \in (0,1)$
Result: Función de valor estado-acción $Q_N : S \times A \to \mathbb{R}$

Inicializar $Q_0(s,a)$;
for $n \leftarrow 1$ **to** N **do**
  Inicializar estado s_t;
  Obtener factor de aprendizaje $\alpha_n \in [0,1)$;
  while *el episodio no ha terminado* **do**
    Observar estado actual s_t;
    Elegir y ejecutar acción a_t;
    Observar nuevo estado s_{t+1} y recompensa r_t;
    $V_{n-1}(s_{t+1}) \leftarrow \max_{a \in A} Q_{n-1}(s_{t+1}, a)$;
    Actualizar $Q_n(s_t, a_t)$ mediante:

$$Q_n(s_t,a_t) \leftarrow Q_{n-1}(s_t,a_t) + \alpha_n \left[r_t + \gamma V_{n-1}(s_{t+1}) - Q_{n-1}(s_t,a_t) \right]$$

    $s_t \leftarrow s_{t+1}$;
  end
  foreach $(s,a) \in S \times A$ *no visitado en el episodio* **do**
    $Q_n(s,a) \leftarrow Q_{n-1}(s,a)$;
  end
end

Observación. Veremos diferentes metodologías para inicializar Q_0.

El siguiente teorema define un conjunto de condiciones bajo las cuales $Q_n(s,a) \to Q^*(s,a)$ cuando $n \to \infty$. Definimos B_n como el conjunto de pares (s,a) que ocurren en el episodio n. Y sea

$$\chi_{B_n}(s,a) = \begin{cases} 1, & \text{si } (s,a) \in B_n, \\ 0, & \text{si } (s,a) \notin B_n. \end{cases}$$

Teorema 3.4. *Si las recompensas están acotadas y las tasas de aprendizaje* $0 \leq \alpha_n < 1$ *cumplen*

$$\sum_{i=1}^{\infty} \alpha_i \cdot \chi_{B_i}(s,a) = \infty, \quad \sum_{i=1}^{\infty} \left[\alpha_i \cdot \chi_{B_i}(s,a) \right]^2 < \infty, \quad \forall s \in S, a \in A,$$

entonces $Q_n(s,a) \to Q^*(s,a)$ *cuando* $n \to \infty$, $\forall s,a$, *con probabilidad* 1.

No daremos la demostración puesto que utiliza herramientas matemáticas avanzadas de procesos estocásticos y teoría de la medida que exceden el alcance de este trabajo. Una demostración puede consultarse en [4].

3.2. Exploración vs. Explotación y Estrategia ε-greedy

En aprendizaje reforzado, el agente enfrenta un conflicto fundamental entre:

- **Exploración**: Probar nuevas acciones para mejorar el conocimiento del entorno.
- **Explotación**: Utilizar el conocimiento actual para maximizar la recompensa inmediata.

Aunque en la ecuación de actualización de la función (3.1) se utiliza el valor máximo de las posibles siguientes acciones para actualizar el estado pasado, esto no implica que esa sea la acción que va a tomar el agente. De ser así, el agente estaría haciendo siempre la misma sucesión de acciones y no buscaría unas mejores. Es por esto que en la práctica se implementa la estrategia ε-greedy.

3.2.1. Estrategia ε-greedy

La política ε-greedy con $\varepsilon \in [0,1]$ resuelve este dilema mediante:

$$\pi(s) = \begin{cases} \text{argmax}_a Q(s,a), & \text{con probabilidad } 1-\varepsilon. \\ \text{acción aleatoria}, & \text{con probabilidad } \varepsilon. \end{cases}$$

Análisis de convergencia. Para garantizar la convergencia a Q^*, la secuencia $\{\varepsilon_n\}$ debe satisfacer:

$$\lim_{n\to\infty} \varepsilon_n = 0 \quad \text{y} \quad \sum_{n=1}^{\infty} \varepsilon_n = \infty. \tag{3.2}$$

Recomendaciones prácticas. Aunque existen muchas técnicas para escoger los valores de ε, uno sencillo que se explica en [7] es el siguiente:

- Inicializar ε a 1 para exploración temprana.
- Usar decaimiento exponencial: $\varepsilon_{n+1} = \varepsilon_n \cdot \eta$ con $\eta \in (0.995, 0.999)$. Que aunque no garantice la convergencia por (3.2) puesto que el sumatorio está acotado, funciona muy bien en la práctica como veremos más adelante.

Observación. Hay que tener en cuenta que esto es solo una recomendación y que dependiendo del problema pueden verse beneficiadas otras estrategias. Además también depende del factor de aprendizaje α y el factor de descuento γ así como de la inicialización de Q_0.

3.3. Inicialización de la matriz Q_0

La elección de la matriz inicial $Q_0(s,a)$ es un aspecto fundamental en el aprendizaje por refuerzo basado en Q-learning, pues puede afectar tanto a la velocidad de convergencia como al comportamiento exploratorio del agente en los primeros episodios. Formalmente, recordemos que $Q_0 : S \times A \to \mathbb{R}$ es la función de valor estado-acción con la que el agente comienza en el episodio $n = 1$:

$$Q_1(s,a) = Q_0(s,a) + \alpha_n \left[r + \gamma \max_{a'} Q_0(s',a') - Q_0(s,a) \right].$$

A continuación, se presentan las principales estrategias de inicialización, junto con su fundamentación matemática y sus implicaciones en la dinámica de aprendizaje [8].

3.3.1. Inicialización cero

La opción más habitual y sencilla es fijar $Q_0(s,a) = 0$ para todo $(s,a) \in S \times A$. Esta estrategia es neutra y garantiza que, bajo las condiciones de convergencia del Teorema 2.5, el algoritmo converge a Q^* independientemente de la inicialización. Sin embargo, puede inducir un sesgo temporal hacia la explotación temprana de acciones que, por azar, hayan dado recompensas positivas en los primeros pasos, ralentizando la exploración de alternativas subóptimas que podrían ser mejores a largo plazo.

3.3.2. Inicialización optimista

Consiste en asignar a todos los pares (s,a) un valor inicial alto, típicamente $Q_0(s,a) = Q_{\text{máx}}$, donde

$$Q_{\text{máx}} \geq \frac{R_{\text{máx}}}{1-\gamma} \text{ y } R_{\text{máx}} = \max_{s,a,s'} |R(s,a,s')|.$$

El fundamento teórico es que, al sobrestimar el valor de todas las acciones, el agente tenderá a probar todas ellas, ya que solo tras obtener recompensas reales (menores que $Q_{\text{máx}}$) los valores $Q_n(s,a)$ comenzarán a decrecer hacia el óptimo. Esto garantiza, en la práctica, que cada par (s,a) será visitado múltiples veces, satisfaciendo la condición de visitas infinitas del teorema de convergencia de Q-learning:

$$\sum_{i=1}^{\infty} \alpha_i \cdot \chi_{N_i}(s,a) = \infty.$$

Esta estrategia acelera la exploración en los primeros episodios y puede reducir el tiempo hasta alcanzar una política cercana a la óptima, aunque puede introducir una sobre-exploración inicial si $Q_{\text{máx}}$ es excesivamente grande.

3.3.3. Inicialización aleatoria acotada

Otra posibilidad es inicializar $Q_0(s,a)$ con valores aleatorios extraídos de una distribución uniforme acotada, por ejemplo $Q_0(s,a) \sim \mathscr{U}(-\varepsilon,\varepsilon)$ con ε pequeño. Esta técnica rompe simetrías y puede ser útil en entornos con múltiples acciones igualmente plausibles, o cuando se emplean aproximadores de la función Q (como redes neuronales[1]), donde la simetría en los pesos iniciales puede ser perjudicial para el aprendizaje. El valor de ε debe ser suficientemente pequeño para evitar sesgos extremos, pero suficiente para inducir diversidad inicial.

3.3.4. Comparativa y recomendaciones

Método	Convergencia	Exploración inicial	Requiere conocimiento
Cero	Lenta	Baja	No
Optimista	Rápida	Alta	Sí ($R_{\text{máx}}$)
Aleatoria	Moderada	Media	No

Cuadro 3.1: Comparativa de estrategias de inicialización de Q_0.

En resumen, la inicialización de Q_0 debe seleccionarse en función de la información disponible y el entorno. En problemas sin conocimiento previo, la inicialización cero es recomendable, pero para compensar la poca exploración se debe aplicar por ejemplo una estrategia de ε-greedy. En todos los casos, la convergencia a Q^* está garantizada bajo las condiciones ya descritas, pero la velocidad y la eficiencia del aprendizaje pueden variar significativamente.

3.4. Hiperparámetros en Q-learning

3.4.1. Factor de Descuento γ

El factor de descuento $\gamma \in (0,1)$ en la ecuación de Bellman modificada para Q-learning,

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right],$$

[1] Hace referencia a las Deep Q-Networks (DQN), se verá en el capítulo 5.

controla el peso relativo de las recompensas futuras frente a las inmediatas. Matemáticamente, γ determina el radio de convergencia de la serie geométrica que representa la recompensa acumulada:

$$G = \sum_{k=0}^{\infty} \gamma^k r_{k+1}.$$

Análisis de Convergencia.

- $\gamma \to 1$**:** La serie G se aproxima al horizonte infinito, maximizando la recompensa a largo plazo. Requiere satisfacer las condiciones del Teorema 2.5.
- $\gamma \to 0$**:** Comportamiento miope, optimizando solo recompensas inmediatas. La ecuación de actualización se reduce a:
$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha r.$$

3.4.2. Tasa de Aprendizaje α

La tasa de aprendizaje $\alpha \in [0,1)$ controla el peso relativo que se otorga a la nueva información obtenida en el paso actual frente al valor previamente aprendido. La diferencia de un episodio con respecto del anterior se puede ver como

$$\Delta Q(s,a) = \alpha \underbrace{\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]}_{\text{Error Temporal}},$$

donde el error temporal cuantifica cómo de disparejo es el valor actual $Q(s,a)$ respecto a la nueva estimación.

En la práctica el entrenamiento suele consistir de centenas o miles de episodios dependiendo de la magnitud del problema a optimizar. Es por esto que se diseñan estrategias de decaimiento de la tasa de aprendizaje. La idea que subyace es que al principio del algoritmo, el agente aún no tiene una idea de cómo es la política óptima, luego se intenta que α sea generosamente grande para que se acerque relativamente rápido a una buena política. Conforme pasan los episodios, el agente ya ha aprendido bastante sobre el sistema y es beneficioso reducir la tasa de aprendizaje para suavizar[2] la aproximación a esta política óptima.

Definición 3.5. Sea $\alpha : \{0,1,2,\dots N\} \to [0,1)$ y $\alpha(0) = \alpha_0 \in [0,1)$. Podemos definir las siguientes estrategias de decaimiento:

$$\begin{aligned}
\text{Constante:} \quad & \alpha(n) = \alpha_0. \\
\text{Lineal:} \quad & \alpha(n) = \alpha_0 \cdot \left(1 - \frac{n}{N_{\text{episodios}}}\right). \\
\text{Exponencial:} \quad & \alpha(n) = \alpha_0 \cdot e^{-\lambda n}, \text{ para } \lambda > 0. \\
\text{Inversa Temporal:} \quad & \alpha(n) = \frac{\alpha_0}{1+\kappa n}, \text{ para } \kappa > 0.
\end{aligned}$$

3.4.3. Interacción γ-α y estrategias conjuntas de optimización

Como en la práctica no podemos hacer infinitas iteraciones del algoritmo de Q-learning para garantizar la convergencia, es imprescindible tener un buen equilibrio entre γ y α que nos acerquen a la solución óptima en un número de episodios razonable.

Apoyándonos en [9] tenemos que usualmente $\gamma \in (0.9, 0.99)$ y $\alpha \in (0.05, 0.25)$, pero esto depende de las características del problema, así como de las recompensas y sobretodo el número de episodios. Es por esto que vamos a ver mejor un ejemplo práctico. Vamos a modelar el problema del lago helado descrito en la sección "Lago helado, parte I".

[2] Para más teoría sobre ajuste de modelos de aprendizaje reforzado ver [1].

3.5. Lago helado, parte III

El objetivo es estudiar la convergencia según los valores de γ y α en el algoritmo de Q-learning (3.1). Por ello, vamos a fijar las siguientes condiciones del problema:

- Inicialización cero de la matriz Q_0.
- Estrategia ε-greedy con decaimiento exponencial: $\varepsilon_{n+1} = \varepsilon_n \cdot 0.999$ con $\varepsilon_0 = 1$.
- El conjunto de estados S, acciones A, función de transición T y de recompensa R son las mismas descritas al comienzo del capítulo 2.
- El número de episodios de cada entrenamiento es $N_{episodios} = 100, 200, \ldots 1000$.
- Los diferentes agentes que se entrenarán son:
  - **Agente 1:** $\gamma = 0.25$, $\alpha = 0.9$ — Valores no recomendados
  - **Agente 2:** $\gamma = 0.95$, $\alpha = 0.1$ — Valores recomendados fijos
  - **Agente 3:** $\gamma = 0.25$, $\alpha_n = 0.1 \cdot \left(1 - \frac{n}{N_{\text{episodios}}}\right)$ — Mal γ y decaimiento lineal de α
  - **Agente 4:** $\gamma = 0.95$, $\alpha_n = 0.1 \cdot \left(1 - \frac{n}{N_{\text{episodios}}}\right)$ — Buen γ y decaimiento lineal de α
- Cada agente se entrenará 100 veces para cada $N_{episodios}$ y como conocemos las 4 políticas óptimas del problema (ver Anexo I) veremos cuántas veces han convergido a una de ellas.

Resultados.

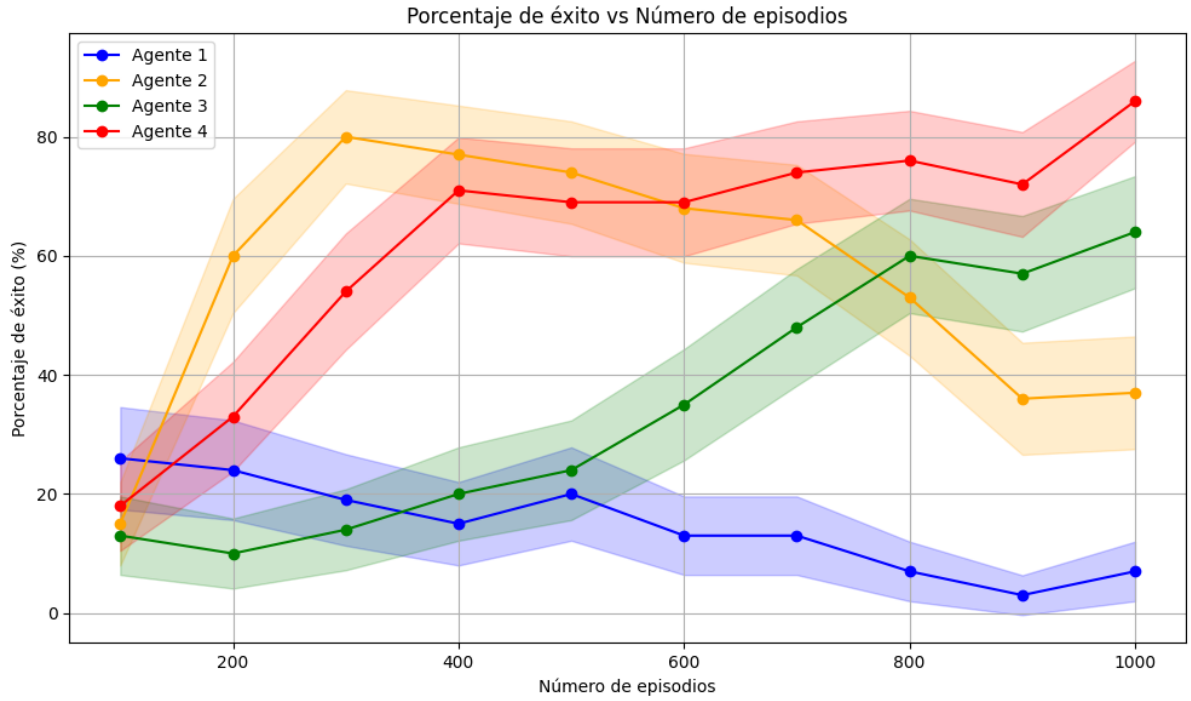


Figura 3.1: Porcentaje de éxito de los agentes en función de los episodios. La zona sombreada son intervalos de confianza del 95 %.

Hemos supuesto del porcentaje de éxito del agente como una distribución Binomial en función del número de episodios. Esto nos permite usar el *intervalo de confianza de Wald* para proporciones binomiales y reflejar la variabilidad estadística de los resultados.

Conclusiones. Como se puede observar, para distinto número de episodios obtenemos que diferentes agentes convergen mejor. Esto se debe a que son muy sensibles a cambios en los hiperparámetros así como al carácter estocástico del problema. Recordemos que para cada movimiento que hace el pingüino, tiene un 10 % de probabilidad de ir en otra dirección.

Dicho esto, los Agentes 1 y 3 que usan un factor de descuento γ no recomendado, convergen relativamente mal, en especial el Agente 1 que también tiene una tasa de aprendizaje no recomendada. En cuanto a los dos modelos con hiperparámetros competentes, el Agente 4 tarda más episodios en converger, pero tiene la característica de mantenerse en esos porcentajes a lo largo de más episodios mientras que la convergencia del Agente 2 crece más rápido pero luego decrece. Esto se debe a que al experimentar con un número tan alto de episodios, la tasa de aprendizaje α es demasiado elevada y actúa en contra del aprendizaje. Este fenómeno conocido como *sobreajuste* o *sobreentrenamiento* se explica más en detalle en [1].

A la hora de elegir los hiperparámetros en un problema desconocido, se puede ver cómo es preferible usar un agente parecido al Agente 4 que es poco volátil y la franja de episodios entre *subajustar* y *sobreajustar* es mucho mayor que por ejemplo el Agente 2.

3.5.1. Estudiar la convergencia del ejemplo haciendo variar α

Vamos a estudiar la importancia del factor de aprendizaje empíricamente. Las condiciones del experimento son las mismas que antes pero con los siguientes cambios:

- Fijamos $\gamma = 0.95$ para todos los agentes.
- El Agente i tiene $\alpha_i = 0.01, 0.02, \ldots, 0.3$.
- Cada agente se entrena 25 veces con $N_{\text{episodios}} = 50, 75, \ldots$ hasta que el ratio de convergencia sea superior a 70 % o en su defecto $N_{\text{episodios}}$ sobrepase los 1000.

Resultados.

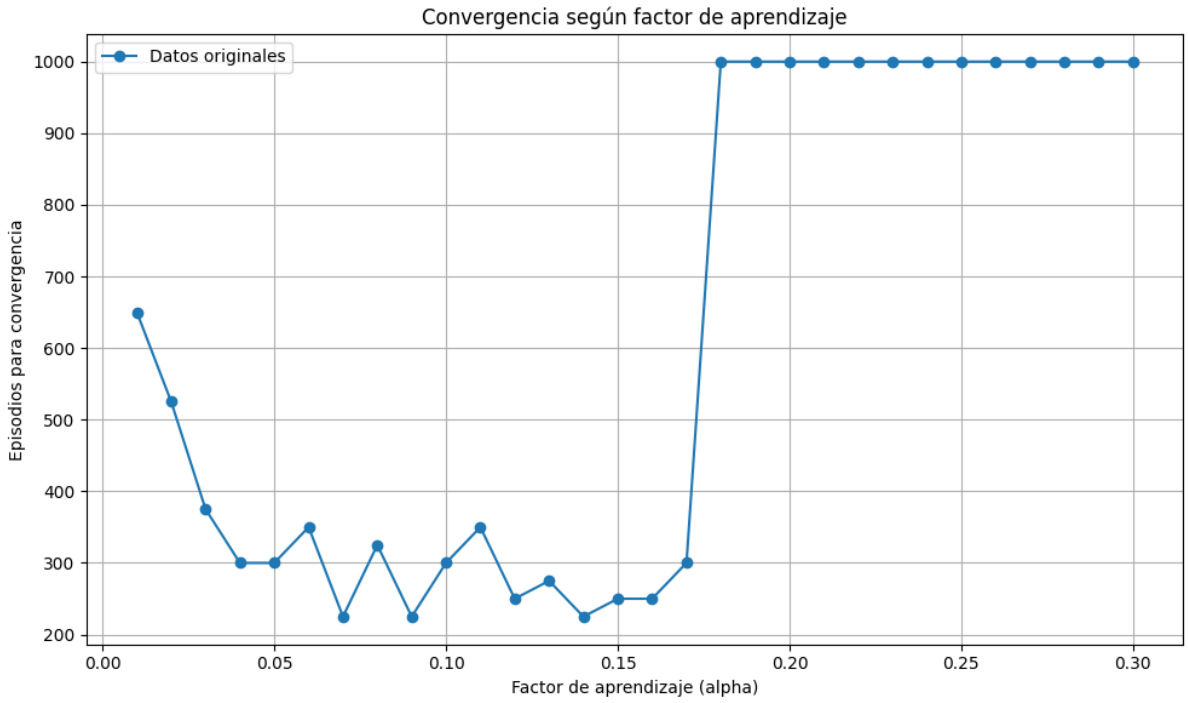


Figura 3.2: Número de episodios hasta converger para factor de aprendizaje $\alpha \in (0, 0.3)$.

Conclusiones. Antes de sacar conclusiones cabe destacar que aunque el criterio utilizado para valorar la convergencia de cada agente no es completamente riguroso, nos permite hacernos una idea global de cómo afecta el factor de aprendizaje en los entrenamientos de modelos. Analizando la gráfica anterior:

- **Valores bajos de α ($\alpha < 0.03$):** El número de episodios necesarios para la convergencia es elevado (más de 400 episodios), lo que indica un aprendizaje lento debido a la escasa actualización de los valores de Q.
- **Rango intermedio ($0.03 < \alpha < 0.18$):** Se observa una convergencia más rápida, con mínimos locales de episodios para convergencia entre 200 y 300. Para estos valores es probable que se dé un sobreajuste si se entrenan a los agentes durante más episodios.
- **Valores altos de α ($\alpha > 0.18$):** La segunda parte de la gráfica muestra un comportamiento inestable propio de un aprendizaje excesivo en episodios tempranos.

En resumen, hemos apoyado empíricamente la teoría descrita en las secciones anteriores con un ejemplo que aunque simple, es posible extrapolar a otros más complejos.

Observación. Los detalles técnicos de la implementación se dejan especificados en el Anexo II.

Capítulo 4

Tres en raya y Q-learning

En este capítulo vamos a modificar el algoritmo básico de Q-learning del capítulo 3 para que aprenda a jugar al tres en raya. Se juega sobre un tablero 3×3 donde 2 jugadores: X y O deben colocar su símbolo una vez por turno y no debe ser sobre una casilla ya jugada. Se debe conseguir realizar una línea recta o diagonal por símbolo.

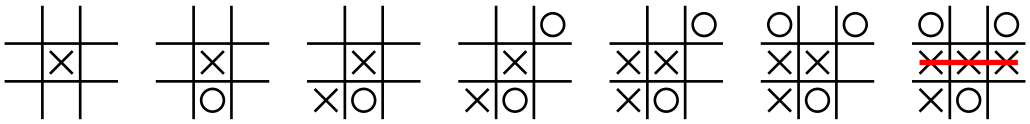


Figura 4.1: Ejemplo de partida de 3 en raya, empieza X.

4.1. Algoritmo de Minimax

Antes de introducir el algoritmo de Q-learning modificado, vamos a destacar el algoritmo más usado y conocido: Minimax.

En primer lugar podemos acotar el número posible de partidas, ya que hay 9 casillas y podemos ver el conjunto U de partidas como las permutaciones de $(1,2,\dots,9)$, donde $u = (u_1,u_2,\dots) \in U$ es la partida que primero se juega una X en la posición u_1, luego una O en la posición u_2 y así sucesivamente. De esta forma $|U| = 9! = 362880$, pero esto no tiene en cuenta que hay muchas partidas de U que no pueden darse ya que terminan antes. Así obtenemos que el número de partidas es $< 9!$, en particular, son 31896 partidas y que se pueden reducir a 14 si eliminamos simetrías [11]. Con esto en mente, este algoritmo es determinista[1] y consiste en elegir siempre el siguiente movimiento en base al siguiente orden:

- Todas las partidas posibles después de ese movimiento acaban en victoria o empate.
- Todas las partidas posibles después de ese movimiento acaban en empate.

Esto es posible siempre y cuando el algoritmo haga el primer o segundo movimiento y hace que si el algoritmo juega contra sí mismo siempre habrá un empate. Ver el algoritmo completo en Anexo III.

4.2. Algoritmo de Q-learning modificado para el tres en raya

Vamos a entrenar a nuestro agente contra sí mismo para que aprenda a jugar en ambos X y O.

[1] Un algoritmo determinista es un procedimiento computacional que no involucra ningún tipo de aleatoriedad; en cada paso, el curso de acción está completamente determinado por el estado actual.

Algoritmo 2: Entrenamiento Q-learning para el Tres en Raya (versión simplificada)

Data: Número de episodios N, tasa de aprendizaje α_n, factor de descuento γ
Result: Tabla Q aprendida $Q(s,a)$

Inicializar $Q(s,a) \leftarrow 0$ para todos los estados s y acciones a;
for *episodio* $\leftarrow 1$ **to** N **do**
  Inicializar estado $s \leftarrow$ tablero vacío, conjunto de acciones disponibles $A \leftarrow \{1,\dots,9\}$;
  while *la partida no ha terminado* **do**
    //Jugador 1;
    Decidir y ejecutar acción a_t en s y eliminar a_t de A;
    Observar nuevo estado s' y recompensa r_t;
    if $A = \emptyset$ **then**
      $Q(s,a_t) \leftarrow Q(s,a_t) + \alpha(r_t - Q(s,a_t))$;
    else
      $Q(s,a_t) \leftarrow Q(s,a_t) + \alpha\,(r_t + \gamma \text{mín}_{a' \in A}\, Q(s',a') - Q(s,a_t))$;
    end
    if $r_t > 0$ *o se ha llenado el tablero* **then**
      break
    end
    $s \leftarrow s'$;
    //Jugador 2;
    Decidir y ejecutar acción a_t en s y eliminar a_t de A;
    Observar nuevo estado s' y recompensa r_t;
    $Q(s,a_t) \leftarrow Q(s,a_t) + \alpha\,(r_t + \gamma \text{máx}_{a' \in A}\, Q(s',a') - Q(s,a_t))$;
    if $r_t < 0$ **then**
      break
    end
    $s \leftarrow s'$;
  end
end

4.2.1. Idea fundamental del algoritmo

La modificación realizada al algoritmo básico permite que la tabla Q se comparta entre dos jugadores. El jugador 1, actualiza la tabla en las jugadas impares y el jugador 2 actualiza las pares. De esta forma entregamos al agente una recompensa de $+1$ si gana el jugador 1 y -1 si gana el jugador 2. Para actualizar las tablas, el jugador 1 toma el **menor** valor esperado de las acciones posibles por su oponente en el siguiente estado, es decir, tras haber escogido dónde va a colocar la X, mira dónde va a colocar el jugador 2 la O y si este valor es muy pequeño, por ejemplo -1, significa que esa X que ha puesto es "mala" porque ha hecho que el oponente gane. Por otro lado el jugador 2 toma el **mayor** valor esperado de las acciones posibles por su oponente en el siguiente estado por el mismo razonamiento.

4.2.2. Hiperparámetros y configuración del entrenamiento

Vamos a justificar por qué hemos elegido cada hiperparámetro de nuestro algoritmo. En primer lugar, nuestro conjunto de estados S representa el todos los posibles estados del tablero (un total de $3^9 = 19683$ estados) y A es el conjunto de acciones posibles, correspondiente a las 9 casillas del tablero. Por tanto, la tabla Q tiene dimensiones 19683×9. Aunque como se ha comentado antes, muchos de estos estados no se pueden alcanzar. Dicho esto hemos establecido lo siguiente:

- **Número de episodios:** 10^5. Para poder visitar 19683 estados varias veces, y aunque la mayoría no son posibles, establecemos un número de episodios alto para garantizar una convergencia.

- **Tasa de aprendizaje**: $\alpha_n = 0.1 \cdot \left(1 - \frac{n}{N_{\text{episodios}}}\right)$. Al tener un número tan alto de episodios, como hemos visto en la sección 2.5.1, es conveniente tomar un α bajo y disminuirlo con los episodios para evitar un sobreajuste en episodios tardíos.

- **Factor de descuento**: $\gamma = 0.95$. Elegimos un valor en el rango aceptable teniendo en cuenta que los episodios son de longitud máxima 9.

- **Factor de exploración**: $\varepsilon_{n+1} = \varepsilon_n \cdot 0.9999$ con $\varepsilon_0 = 1$. Queremos que inicie explorando en abundancia y aprenda qué jugadas hacen que pierda cada jugador. Conforme pasan los episodios queremos que explote los conocimientos que tiene, con esto no solo conseguimos que cada jugador juegue mejor cuando tiene una jugada ganadora, sino que también nos sirve para que los oponentes se hagan mejor defendiéndose de posibles jugadas ganadoras, aquí es donde brilla la tabla Q compartida en el autoaprendizaje. Acerca de haber tomado como tasa de decaimiento 0.9999, la idea reside en intentar ajustarse a las condiciones de convergencia (3.2) para un número de episodios finito. Para ello y con $N = 100000$:

$$\lim_{n \to N} \varepsilon_n = 0.9999^{100000} = 0.00004537723 \approx 0,$$

$$\sum_{n=1}^{N} \varepsilon_n = \sum_{n=1}^{N} (0.9999)^n = 0.9999 \times \frac{1 - (0.9999)^N}{1 - 0.9999} = 9998.55 \quad \text{(suficientemente grande)}.$$

- **Tabla inicial**: $Q(s,a) = 0 \quad \forall (s,a) \in S \times A$. Como la tabla se aprovecha en dos direcciones dependiendo del jugador, es preferible no inicializar la tabla con valores aleatorios ya que puede sugestionar algún comportamiento del agente a la hora de actualizar los valores q. Por otro lado, de cara a comprobar el entrenamiento, podemos ver los estados no visitados, estos serán como máximo aquellos $s \in S$ tal que $Q(s,a) = 0, \forall a \in A$.

4.2.3. Resultados obtenidos y comprobación del algoritmo

Vamos a comentar algunos resultados de la ejecución del programa *tictactoe_train.py* que se puede ver en el Anexo III. En primer lugar, el programa lleva la cuenta del número de partidas que gana cada jugador en los 10^5 episodios. Obtenemos entorno al 82 % de empates, un 12 % de victorias del jugador 1 y un 6 % de victorias del jugador 2. Esto viene fuertemente influenciado por el elevado número de partidas afectadas por la exploración usada en ε-greedy. Aun así, el alto porcentaje de empates sugiere una convergencia del algoritmo a una partida donde ambos jugadores juegan sin errores.

Otro de los valores que nos muestra la ejecución del programa es el número de estados visitados. Como hemos comentado antes, esto se hace contando los estados $s \in S$ tales que $\exists a \in A$ con $Q(s,a) \neq 0$. El argumento es que si ha modificado la tabla Q asociada a ese estado, obligatoriamente ha tenido que pasar por él. Podría darse que se modifique dicho valor con un valor positivo y luego se le reste el complementario, pero teniendo en cuenta el factor de descuento y la tasa de aprendizaje, así como los redondeos propios de los programas informáticos esto es altamente improbable. Sabiendo esto, obtenemos que la máquina se enfrenta a una media de 3250 estados. A priori este valor es muy bajo, si mantenemos por ejemplo el factor de exploración $\varepsilon = 1$, obtenemos para el mismo número de episodios entorno a 4300 estados visitados. Esto no es un problema, el algoritmo al entrenarse contra sí mismo es capaz de evitar muchos estados que no se podrían llegar jugando de manera óptima, por ejemplo, estados que para llegar a ellos ha tenido que saltarse una victoria previa. Para justificar empíricamente que 3250 estados son suficientes vamos con la comprobación del algoritmo.

Se ha diseñado el programa *play_tictactoe.py* [10] para probar al algoritmo mediante dos métodos, prueba humana y fuerza bruta. El primero consiste en que el usuario elija si quiere ser X o O y juegue usando las teclas 1-9 del teclado para indicar en qué casilla colocar su pieza, el algoritmo responderá. Tras una experimentación manual, no solo es imposible ganar sino que siempre que exista una secuencia de jugadas que le aseguren la victoria, el algoritmo las ejecuta.

La fuerza bruta por otro lado es todavía más determinante. Simula 10^6 de partidas donde el algoritmo elige aleatoriamente si ser X o O y juega contra un algoritmo aleatorio. Los resultados tras la ejecución rondan el 90 % de victoria para el algoritmo de Q-learning, un 10 % de empates y un 0 % libre de redondeos de victorias para el algoritmo aleatorio. Este número garantiza que la convergencia aun explorando solo 3250 estados es óptima. Nuestro algoritmo de Q-learning para el tres en raya ha encontrado una política óptima.

Capítulo 5

Deep Q-learning

En este apartado trataremos un modelo de aprendizaje reforzado que se puede emplear para tareas más complejas. En particular, dado un MDP (S,A,T,R), nos interesa el tamaño de la tabla Q que se usa para almacenar los valores Q y que viene dado por $|S| \cdot |A|$. Cuando este tamaño supera las 10^5 combinaciones[1] de estado-acción se sugiere aproximar la función Q y no tener así que almacenar tantos valores.

Observación. Explicar todas las bases y el marco formal que rodean a las redes neuronales y que son necesarias para comprender en detalle el Deep Q-learning se escapa, por espacio disponible, del alcance de este trabajo. Para más detalle, ver [1].

5.1. Fitted Q-learning y redes neuronales

En problemas de Aprendizaje Reforzado con espacios de estado y acción de gran dimensión o continuos, la representación tabular de la función de valor-acción $Q(s,a)$ resulta inviable. El *Fitted Q-learning* aborda este reto empleando un modelo paramétrico $Q(s,a;\theta)$, donde θ denota los parámetros de un aproximador funcional, típicamente una *red neuronal*. El objetivo es ajustar θ para que $Q(s,a;\theta)$ se aproxime a la función $Q^*(s,a)$ óptima.

Definición 5.1. Una *red neuronal feed-forward completamente conectada*, también conocida como *Perceptrón Multicapa* (del inglés *Multilayer Perceptron*, MLP), con $L \in \mathbb{N}, L > 1$ capas (*profundidad* L) y arquitectura $(n_0, n_1, \ldots, n_L)$, donde $n_i \in \mathbb{N}$ representa el número de nodos en la capa i, $n_0 = d$ es la dimensión de la entrada y $n_L = m$ la dimensión de la salida, consiste en una composición de funciones de la forma:

$$f(\mathbf{x};\theta) = \phi_L\left(\mathbf{W}_L\,\phi_{L-1}\left(\mathbf{W}_{L-1}\cdots\phi_1(\mathbf{W}_1\mathbf{x}+\mathbf{b}_1)\cdots+\mathbf{b}_{L-1}\right)+\mathbf{b}_L\right),$$

donde:

- $\mathbf{x} \in \mathbb{R}^d$ es la entrada,
- $\mathbf{W}_l \in \mathbb{R}^{n_l \times n_{l-1}}$ y $\mathbf{b}_l \in \mathbb{R}^{n_l}$ son los parámetros (pesos y sesgos) de la capa l,
- $\phi_l : \mathbb{R}^{n_l} \to \mathbb{R}^{n_l}$ es una función de activación no lineal (por ejemplo, la función ReLU: $\phi(z) = \text{máx}\{0,z\}$).

Las *activaciones intermedias* de la red se definen recursivamente como:

$$\begin{aligned} &\mathbf{h}^{[0]} := \mathbf{x}, \\ &\mathbf{h}^{[l]} := \phi_l(\mathbf{W}_l\mathbf{h}^{[l-1]} + \mathbf{b}_l), \qquad \text{para } l = 1,\ldots,L, \end{aligned}$$

[1] Este límite no es fijo y depende de la capacidad computacional y del objetivo del estudio. Se ha establecido así en este trabajo para marcar una barrera numérica.

donde $\mathbf{h}^{[l]} \in \mathbb{R}^{n_l}$ representa el vector de *activaciones* de la capa l.

El componente j-ésimo de la capa l se denota por $h_j^{[l]}$ y se calcula como:

$$h_j^{[l]} = \phi_l \left(\sum_{k=1}^{n_{l-1}} W_{jk}^{[l]} h_k^{[l-1]} + b_j^{[l]} \right),$$

donde:

- $W_{jk}^{[l]}$ es el peso que conecta la neurona k de la capa $l-1$ con la neurona j de la capa l,
- $b_j^{[l]}$ es el sesgo de la neurona j en la capa l.

Así, $h_j^{[l]}$ representa la salida (o *activación*) de la neurona j en la capa l, tras aplicar la función de activación a la combinación lineal de las salidas de la capa anterior.

Definición 5.2. Se le llama *red neuronal profunda* a una red neuronal con más de 2 capas ($L \geq 3$), o lo que es lo mismo, con más de una capa oculta:

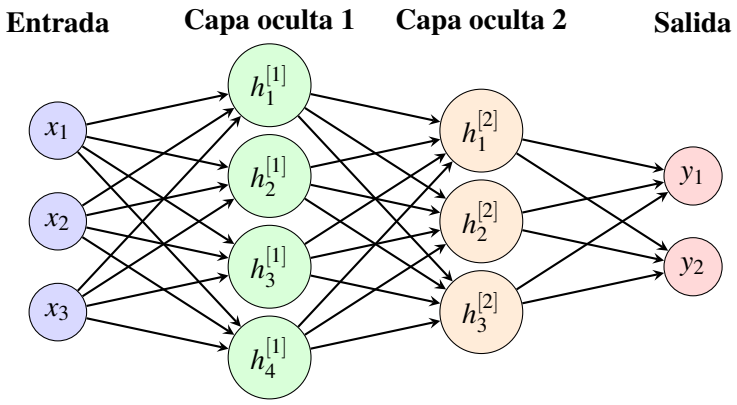


Figura 5.1: Red neuronal profunda (L=3).

Definición 5.3. Dado un MDP (S, A, T, R), un *buffer de experiencia* $\mathscr{D}$ con tamaño $N \in \mathbb{N}$ es un conjunto finito de tuplas de la forma

$$\mathscr{D} = \{(s_t, a_t, r_{t+1}, s_{t+1})\}_{t=1}^{N}.$$

A las tuplas de la forma (s, a, r, s') se las conoce habitualmente como *transiciones*. El buffer $\mathscr{D}$ funciona como almacén de las últimas experiencias que ha vivido el agente de modo que, cuando se alcanza la capacidad máxima, las transiciones más antiguas se sobrescriben con las nuevas. El propósito fundamental del buffer de experiencia es *romper las correlaciones temporales* presentes en las secuencias de datos generadas por el agente, permitiendo que las actualizaciones de los parámetros de la red neuronal se realicen sobre mini-batches[2] de transiciones muestreadas aleatoriamente de $\mathscr{D}$.

Definición 5.4. En el contexto de redes neuronales, una *red objetivo* es una copia desacoplada de la red neuronal principal utilizada para aproximar la función de valor acción-estado $Q(s, a; \theta)$. Denotada como $Q(s, a; \theta^-)$, mantiene sus parámetros θ^- fijos durante C pasos del entrenamiento.

[2]Un *mini-batch* es un pequeño subconjunto aleatorio del conjunto de datos de entrenamiento que se utiliza en cada iteración del proceso de entrenamiento de un modelo de machine learning. En lugar de procesar todo el conjunto de datos a la vez (full batch) o una sola muestra por vez (stochastic), el mini-batch permite calcular el gradiente y actualizar los parámetros del modelo usando solo ese subconjunto, logrando un equilibrio entre eficiencia computacional y estabilidad en la convergencia.

Actualización de parámetros. Dado un conjunto de transiciones (s,a,r,s') almacenadas en un buffer de experiencia $\mathscr{D}$, el aprendizaje se realiza minimizando el error cuadrático medio entre la predicción actual y el objetivo de Bellman:

$$\mathscr{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathscr{D}}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta^-) - Q(s,a;\theta)\right)^2\right],$$

donde $\gamma \in (0,1)$ es el factor de descuento y θ^- son los parámetros de una *red objetivo* que se actualiza cada C pasos para estabilizar el entrenamiento.

La actualización de los parámetros θ se realiza mediante descenso estocástico de gradiente [13] sobre $\mathscr{L}(\theta)$. Este procedimiento, conocido como fitted Q-iteration, generaliza el Q-learning tabular al caso de aproximadores funcionales.

5.2. Deep Q-Networks (DQN)

El *Deep Q-Network* (DQN) es el algoritmo que integra fitted Q-learning con redes neuronales profundas, logrando resolver tareas de control en entornos de alta dimensión. Un algoritmo que se emplea para su entrenamiento es el Algoritmo 3.

5.2.1. Aproximación universal de funciones con redes neuronales

El *Teorema de aproximación universal* con redes neuronales [14], nos asegura que para cualquier función continua, existe una red neuronal de una capa oculta (L = 2) que puede aproximar dicha función en un subconjunto de $\mathbb{R}^n$ con precisión arbitraria.

En el contexto de DQN, esto significa que existe una red neuronal $Q(s,a;\theta)$ capaz de aproximar la verdadera función de valor óptima $Q^*(s,a)$ para cualquier entorno de Markov con recompensas y transiciones continuas.

5.3. CartPole en DQN

Tras introducir unas nociones breves de DQN, vamos a mostrar ahora un ejemplo de entrenamiento de un agente usando DQN que no sería posible realizarse con Q-learning tabular (Usando una tabla Q). En concreto, vamos a entrenar a un agente a balancear un palo moviendo un carrito.

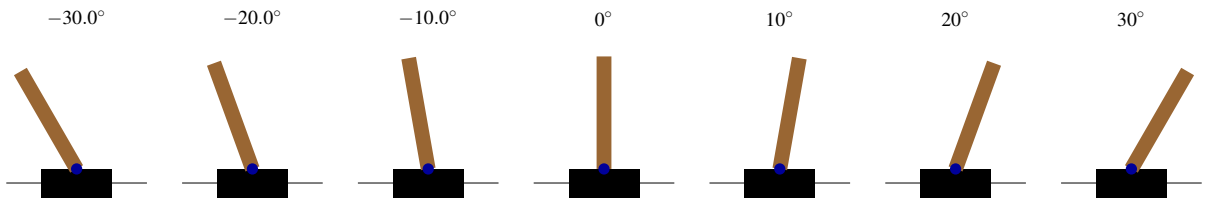


Figura 5.2: Siete CartPoles con diferentes ángulos de inclinación del péndulo.

5.3.1. Descripción del Problema de CartPole

Para modelar el entorno, resultaba más cómodo usar una biblioteca de Python en la que ya estuvieran hechas las físicas del carrito y el palo. Con esto nos podemos centrar solo en la red neuronal y el algoritmo de entrenamiento. En particular hemos usado el siguiente sistema dinámico implementado en Gymnasium[3]:

[3]Gymnasium es una biblioteca open source para entornos de aprendizaje por refuerzo mantenida y desarrollada principalmente por la Farama Foundation, creado para estandarizar y mantener a largo plazo las interfaces y entornos de Aprendizaje Reforzado. En la página `https://gymnasium.farama.org/index.html` se puede encontrar información al respecto.

Algoritmo 3: Procedimiento de entrenamiento DQN con experiencia repetida y red objetivo

Data: Red $Q(s,a;\theta)$, red objetivo $Q(s,a;\theta^-)$, memoria de repetición $\mathscr{D}$, tasa de aprendizaje α, factor de descuento γ, tasa de exploración ε, frecuencia de actualización C

Result: Parámetros entrenados θ

Inicializar $Q(s,a;\theta)$ y $Q(s,a;\theta^-)$ con los mismos valores;
Inicializar memoria de repetición $\mathscr{D}$;
for *cada episodio hasta* $N_{episodios}$ **do**
  Inicializar estado s;
  while *no termina el episodio* **do**
    Seleccionar acción a mediante política ε-greedy:;

$$a = \begin{cases} \text{aleatoria} & \text{con probabilidad } \varepsilon \\ \arg\max\limits_{a} Q(s,a;\theta) & \text{en otro caso} \end{cases}$$

    Ejecutar acción a, observar recompensa r y nuevo estado s';
    Almacenar transición (s,a,r,s') en $\mathscr{D}$;
    $s \leftarrow s'$;
    Muestrear un mini-batch de transiciones (s_j,a_j,r_j,s'_j) de $\mathscr{D}$ de tamaño M;
    for *cada transición en el mini-batch* **do**
      Calcular el objetivo de Bellman:

$$y_j = r_j + \gamma \max_{a'} Q(s'_j,a';\theta^-)$$

      Calcular la pérdida cuadrática:

$$\mathscr{L}_j(\theta) = (y_j - Q(s_j,a_j;\theta))^2$$

    end
    Actualizar los parámetros θ minimizando la pérdida promedio sobre el mini-batch:

$$\mathscr{L}(\theta) = \frac{1}{M}\sum_j L_j(\theta)$$

    if *número de pasos* mód $C = 0$ **then**
      Actualizar red objetivo: $\theta^- \leftarrow \theta$;
    end
  end
end

- **Espacio de estados**: $s = [x, \dot{x}, \theta, \dot{\theta}] \in \mathbb{R}^4$ que incluye:
  - Posición $x \in [-4.8, 4.8]$ del carro.
  - Velocidad $\dot{x}$.
  - Ángulo $\theta \in [-0.418, 0.418]$ radianes del péndulo.
  - Velocidad angular $\dot{\theta}$.
- **Espacio de acciones**: Discreto $a \in \{0, 1\}$ (izquierda/derecha)

- **Dinámica**: Sistema no lineal gobernado por:

$$\ddot{\theta} = \frac{g\sin\theta - \cos\theta\left(\frac{F+m_p l\dot{\theta}^2\sin\theta}{m_c+m_p}\right)}{l\left(\frac{4}{3} - \frac{m_p\cos^2\theta}{m_c+m_p}\right)},$$

  donde $m_c = 1.0$ kg (masa carro), $m_p = 0.1$ kg (masa péndulo), $l = 0.5$ m.

El episodio termina cuando $|\theta| > 0.209$ rad o $|x| > 4.8$, con recompensa $r_t = 1$ por paso exitoso. Además, para variar las condiciones inicales y simular una situación más "real" tenemos que $x_0, \dot{x}_0, \theta_0, \dot{\theta}_0 \in (-0.05, 0.05)$ de forma aleatoria.

5.3.2. Entrenamiento del Agente DQN

La arquitectura implementada y que se puede ver con más detalle en el Anexo IV incluye:

- **Red Q**:

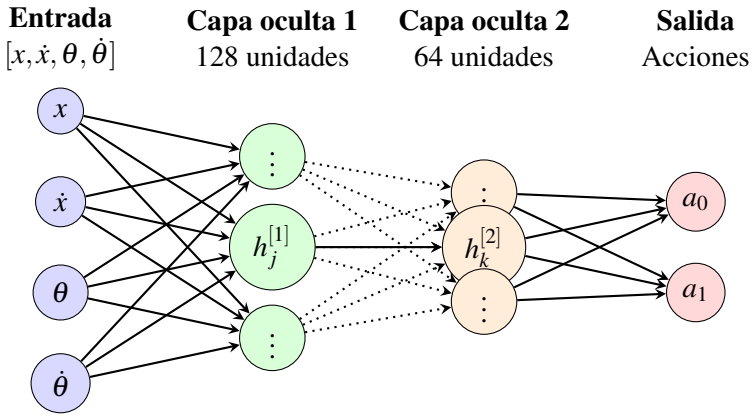


Figura 5.3: Arquitectura de la red Q para CartPole (DQN) con capas completamente conectadas. Las conexiones punteadas representan múltiples conexiones paralelas.

- **Hiperparámetros**:

Hiperparámetro	**Valor**
Factor de descuento (γ)	0.99
Tasa de aprendizaje (α)	2.5×10^{-4}
Tamaño del replay buffer ($\lvert\mathscr{D}\rvert$)	2000
mini-batch size	64
Actualización red objetivo	Al finalizar el episodio
ε_{start}	1.0
ε_{end}	0.1
Decaimiento de ε	500
Función de pérdida	Huber Loss
Episodios de entrenamiento	Máximo de 300
Umbral de aprendizaje	350
Tiempo máximo de episodio	500

- La función de decrecimiento de ε usada para el episodio t es:

$$\varepsilon_t = \varepsilon_{\text{end}} + (\varepsilon_{\text{start}} - \varepsilon_{\text{end}}) \cdot e^{-t/500}.$$

- Función de pérdida (Huber loss), es una función de pérdida que se comporta como la pérdida L2 cuando el error es pequeño y como la pérdida L1 cuando el error es grande. Esto ayuda a mantener la estabilidad del entrenamiento:

$$\mathscr{L} = \frac{1}{N}\sum_{i=1}^{N} \begin{cases} \frac{1}{2}(y_i - Q(s_i,a_i))^2 & \text{si } |y_i - Q| \leq 1, \\ |y_i - Q(s_i,a_i)| - \frac{1}{2} & \text{en otro caso,} \end{cases}$$

  donde $y_i = r_i + \gamma \text{máx}_{a'} Q_{\text{objetivo}}(s_i', a')$.

- Para evitar un sobreajuste se ha implementado un sistema de *parada temprana* que garantice un comportamiento óptimo de media en los últimos 100 episodios. En particular, el agente almacena sus últimas 100 recompensas obtenidas en cada episodio y hace una media de esos valores. Si supera el umbral de aprendizaje establecido (350), se para el entrenamiento.

A continucación se muetran unas gráficas obtenidas en el entrenamiento del agente. Al ser un entorno altemente sensible, las recompensas por episodio tienen un comportamiento oscilante, pero se puede ver como la recompensa media tiene un carácter creciente en general.

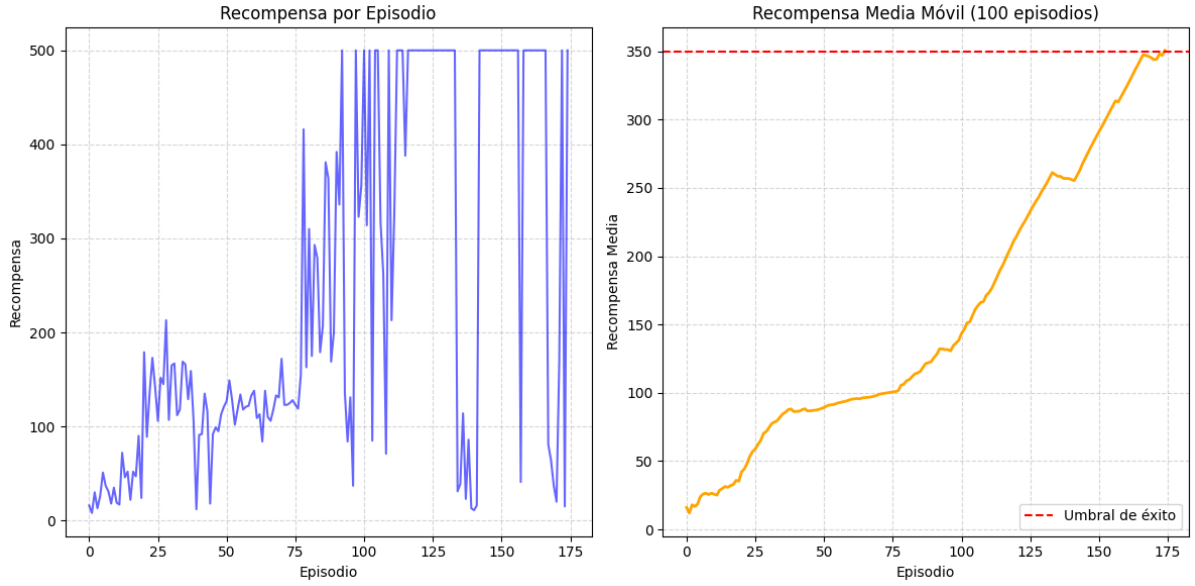


Figura 5.4: Recompensa por cada episodio de entrenamiento y recompensa media de los últimos 100 episodios.

5.3.3. Prueba del agente

Una vez tenemos nuestra red neuronal entrenada, o lo que es lo mismo, unos pesos y sesgos asociados a cada nodo, queremos comprobar que la red neuronal se aproxima a la política óptima. Para ello, y con el fin de hacerlo más visual, se prueba al algoritmo en un entorno donde podemos ver en tiempo real cómo se balancea el palo sobre el carrito. Una vez más, esto se logra usando la librería de Gymnasium. Además, para poner realmente a prueba el agente, aumentamos el tiempo máximo de episodio de 500 a 10000. En todas las pruebas que se han realizado se obtiene que el agente alcanza el tiempo máximo, es decir, podemos afirmar que se ha llegado a una política óptima.

Bibliografía

[1] PRINCE, SIMON JD. *Understanding deep learning. MIT press, 2023.*

[2] WIERING, MARCO A.; VAN OTTERLO, MARTIJN. *Reinforcement learning. Adaptation, learning, and optimization. 2012, vol. 12, no 3, p. 729.*

[3] BELLMAN, RICHARD. *Dynamic programming. science, 1966, vol. 153, no 3731, p. 34-37.*

[4] WATKINS, CHRISTOPHER JCH, AND PETER DAYAN. *Q-learning. Machine learning 8 (1992): 279-292.*

[5] FERNÁNDEZ, SANTIAGO BABÍO, AND ENRIQUE ZUAZUA IRIONDO. *Procesos de decisión de Markov y Q-learning. (2021).*

[6] SUTTON, RICHARD S., AND ANDREW G. BARTO. *Reinforcement learning: An introduction. Vol. 1. No. 1. Cambridge: MIT press, 1998.*

[7] MAROTI, AAKASH. *Rbed: Reward based epsilon decay. arXiv preprint arXiv:1910.13701 (2019).*

[8] SHTEINGART, HANAN, TAL NEIMAN, AND YONATAN LOEWENSTEIN. *The role of first impression in operant learning. Journal of Experimental Psychology: General 142.2 (2013).*

[9] EVEN-DAR, EYAL, AND YISHAY MANSOUR. *Learning rates for Q-learning. Journal of machine learning Research 5.Dec (2003).*

[10] MIGUEL ÁNGEL LUQUÍN GUERRERO. *Repositorio "Fundamentos_Matematicos_AR_TFG"*. `https://github.com/MiguelAngelLuquin/Fundamentos_Matematicos_AR_TFG`.

[11] MARC EVANSTEIN. *There are Exactly 14 Different Games of Tic-Tac-Toe*. `https://www.youtube.com/watch?v=QNFQvX-MQgI`.

[12] DONALD MICHIE. *Matchbox Educable Noughts and Crosses Engine*. `https://en.wikipedia.org/wiki/Matchbox_Educable_Noughts_and_Crosses_Engine`.

[13] STEPHAN, MANDT, MATTHEW D. HOFFMAN, AND DAVID M. BLEI. *Stochastic gradient descent as approximate bayesian inference. Journal of Machine Learning Research 18.134 (2017).*

[14] AUGUSTINE, MIDHUN T. *A survey on universal approximation theorems. arXiv preprint arXiv:2407.12895 (2024).*

[15] MARTIN L PUTERMAN. *Markov decision processes: discrete stochastic dynamic programming. John Wiley & Sons, 2014.*

Anexo I. Resolución numérica del lago helado

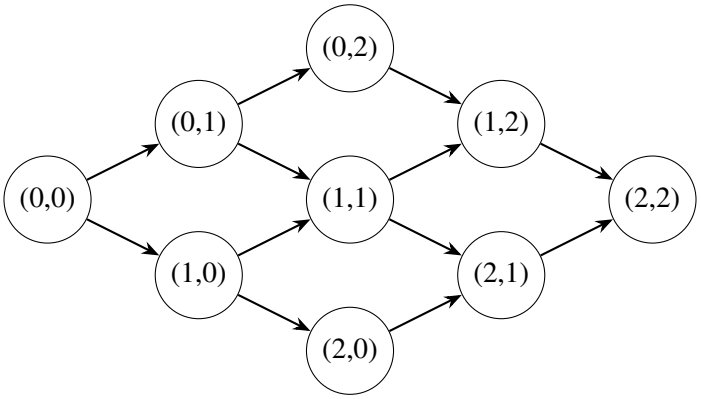
Nuestro objetivo es maximizar

$$V^{\pi}(s) = E_{\pi}\left\{ \sum_{k=0}^{\infty} \gamma^k r_k \middle| s_0 = s \right\}$$

para las condiciones de Lago helado, parte I. Vamos a empezar acotando la magnitud del problema con unas acotaciones previas:

- Ninguna política óptima hará que la acción del pingüino en una casilla del borde sea salirse del tablero.
- Ninguna acción de la política óptima será ir a la izquierda o abajo. Esto se debe a que nos estaríamos alejando de la meta, y el factor de descuento $\gamma \in (0,1)$ que hay en el sumatorio hace que la recompensa futura sea menor cuantos más pasos demos antes de llegar a la meta.

Con esto podemos reducir el problema mediante un grafo similar al del ejemplo Ejemplo 1.12.



Para cada nodo, la flecha superior equivale a "moverse arriba" y la inferior a "moverse abajo". Notar que todos los nodos menos el central tienen la probabilidad de que al tomar una acción nos salgamos del tablero. En particular,

s_i	Casilla	P(salirse del tablero)	P(retroceder sin salirse)	P(fallar pero avanzar)
s_0	(0,0)	2/30	0	1/30
s_1	(0,1)	1/30	1/30	1/30
s_2	(1,0)	1/30	1/30	1/30
s_3	(0,2)	2/30	1/30	0
s_4	(1,1)	0	2/30	1/30
s_5	(2,0)	2/30	1/30	0
s_6	(1,2)	1/30	2/30	0
s_7	(2,1)	1/30	2/30	0

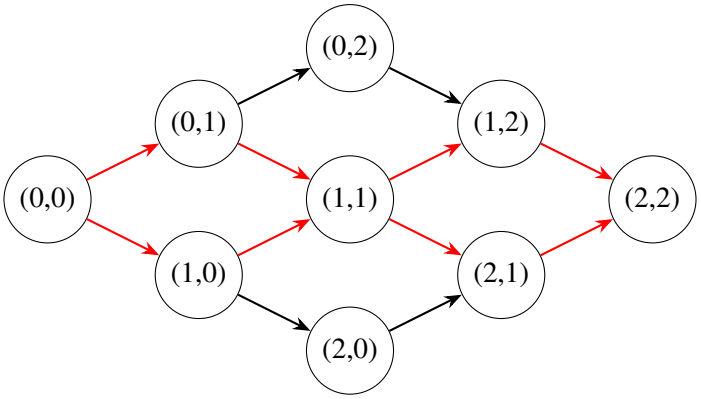
Recordemos que salirse del tablero implica obtener una recompensa de -1. Ya podemos intuir cuáles van a ser las 4 políticas óptimas, pero vamos a dar una última acotación antes de los cálculos. Consideramos que si no se hace la acción deseada y retrocedemos abajo o a la izquierda pero no nos salimos del tablero, obtenemos una recompensa de 0. El argumento es que la política óptima obliga a llegar a meta en 4 acciones y si retrocedemos pero no caemos al lago, podemos seguir llegando a la meta pero no en 4 pasos. Ahora ya podemos calcular todo usando (2.5) y un $\gamma = 0.95$:

$$\begin{aligned}
V^*(s_7) &= T(s_8, arriba, meta)R(s_8, arriba, meta) \\
&\quad + T(s_8, arriba, fuera)R(s_8, arriba, fuera) \\
&\quad + T(s_8, arriba, retroceder)R(s_8, arriba, retroceder) \\
&\quad = 0.867 \\
V^*(s_6) &= 0.867 \\
V^*(s_5) &= 0.867 \cdot 0.9 \cdot 0.95 - 2/30 = 0.674 \\
V^*(s_4) &= 0.867 \cdot 0.933 \cdot 0.95 = 0.768 \\
V^*(s_3) &= 0.867 \cdot 0.9 \cdot 0.95 - 2/30 = 0.674 \\
V^*(s_2) &= 0.9 \cdot 0.95 \cdot 0.768 + 0.95 \cdot 0.033 \cdot 0.674 - 0.066 = 0.612 \\
V^*(s_1) &= 0.9 \cdot 0.95 \cdot 0.768 + 0.95 \cdot 0.033 \cdot 0.674 - 0.066 = 0.612 \\
V^*(s_0) &= 0.933 \cdot 0.95 \cdot 0.612 - 0.066 = 0.476
\end{aligned}$$

y por tanto las 4 políticas óptimas se obtienen de combinar:

s_i	$\pi(s_i)$
s_0	arriba o derecha
s_1	derecha
s_2	arriba
s_3	derecha
s_4	arriba o derecha
s_5	arriba
s_6	derecha
s_7	arriba

Que para verlo visualmente:



Anexo II. Programa para la resolución del lago helado

A continuación se muestra un fragmento representativo del código usado en el primer experimento.

```
def train(self, episodes=100):
    for _ in range(episodes):
        state = (0, 0)
        done = False
        while not done:
            action = self.choose_action(state, self.epsilon)
            action_idx = self.actions.index(action)
            next_state, reward, done = self.step(state, action)
            x, y = state
            nx, ny = next_state
            max_next_q = 0 if done else np.max(self.q_table[nx, ny])
            self.q_table[x, y, action_idx] = (
                (1 - self.alpha) * self.q_table[x, y, action_idx] +
                self.alpha * (reward + self.gamma * max_next_q))
            state = next_state
        self.epsilon *= self.epsilon_decay
        if(self.agent == 3 or self.agent == 4):
                self.alpha = self.alphastart * (1 - i / episodes)

def get_optimal_path(self):
    if (np.argmax(self.q_table[0, 0]) == 0 or np.argmax(self.q_table[0, 0]) ==
        3):
        if (np.argmax(self.q_table[0, 1]) == 3):
            if (np.argmax(self.q_table[1, 0]) == 0):
                if (np.argmax(self.q_table[0, 2]) == 3):
                    if (np.argmax(self.q_table[1, 1]) == 0 or np.argmax(self.
                        q_table[1, 1]) == 3):
                        if (np.argmax(self.q_table[2, 0]) == 0):
                            if (np.argmax(self.q_table[1, 2]) == 3):
                                if (np.argmax(self.q_table[2, 1]) == 0):
                                    return 1
    return 0

if __name__ == "__main__":
    entrenamientos = 100
    episodios = list(range(100, 1001, 100))
    resultados = {j: [] for j in range(1, 5)}
    for j in range(1, 5):
        for k in episodios:
            count = 0
            for i in range(entrenamientos):
                lake = FrozenLakeQLearning(agent=j)
                lake.train(episodes=k)
                count += lake.get_optimal_path()
            porcentaje = 100 * count / entrenamientos
            resultados[j].append(porcentaje)
```

En primer lugar, para cada uno de los 2 experimentos he programado un código similar pero con ligeras modificaciones. Debido a la longitud de ambos, aquí solo comentaremos los detalles más importantes de la implementación común. Para ver el código completo, ver *FrozenLake* en [10].

En primer lugar notar que el algoritmo de actualización de la tabla Q está en las líneas 12-15 y es el mismo que describimos en la Sección 2.1. En las líneas 18-19 se ve como se implementa el alpha decay para los agentes 3 y 4 en cada episodio. Aunque en los parámetros de la función se establece episodes=100 cuando llamamos a la función *train* en *"__main__"*, es cuando ponemos el número de episodios que queremos. En este caso, primero 100, luego 200, y así sucesivamente hasta llegar a 1000.

Para validar si ha encontrado una política óptima, comprobamos que la tabla q coincida con una de las 4 políticas óptimas halladas en Anexo I.

La parte principal consiste de 3 bucles de iteración anidados. El primero indica el agente a entrenar, el segundo el número de episodios que lo vamos a entrenar y el último repite este entrenamiento 100 veces para poder sacar los porcentajes de convergencia.

Por último vamos a ver la implementación en código de fallar en la acción tomada:

```
def step(self, state, action, test=0):
    x, y = state
    new_state = state
    rand = np.random.uniform(0, 1)
    action2 = action
    while(rand < 0.1 and action2 == action and test == 0):
        action2 = np.random.choice(self.actions)
    if action2 == 'up':
        new_state = (x, y+1)
    elif action2 == 'down':
        new_state = (x, y-1)
    elif action2 == 'left':
        new_state = (x-1, y)
    elif action2 == 'right':
        new_state = (x+1, y)

    reward = self.get_reward(new_state)
    done = (reward == 1) or (reward == -0.5)

    if reward == -0.5:
        new_state = state  # No se mueve

    return new_state, reward, done
```

Una vez sabemos la acción que vamos a tomar, si un número pseudoaleatório que se genera es menor que 0.1, no se sale de un bucle hasta que no se elija aleatoriamente con la función de numPy random.choice() otra acción que no sea la que habíamos elegido y así logramos el comportamiento del problema.

Anexo III. Programas para la resolución del tres en raya

Este es el algoritmo de *minimax* en python para tres en raya:

```
def minimax(board, is_maximizing):
    if game_over(board):
        return score(board)  # +1, -1, or 0

    if is_maximizing:
        best_score = -infinity
        for move in possible_moves(board):
            apply_move(board, move, 'X')
            score = minimax(board, False)
            undo_move(board, move)
            best_score = max(score, best_score)
        return best_score
    else:
        best_score = +infinity
        for move in possible_moves(board):
            apply_move(board, move, 'O')
            score = minimax(board, True)
            undo_move(board, move)
            best_score = min(score, best_score)
        return best_score
```

Este es el programa *tictactoe_train.py* usado para entrenar a un agente a jugar al 3 en raya y guardar los resultados en un .csv para su posterior comprobación manual:

```
import numpy as np
import random
import csv

# Parametros del entorno
ROWS = 3
COLS = 3
STATES = 3**(ROWS*COLS)
EPISODES = 100000 #100 000 iteraciones
INI = 0

# Crear el tablero inicial
def tablero_inicial():
    tab = np.zeros((ROWS, COLS))
    return tab

# Obtener la siguiente jugada dados el estado y la accion
def next_tab(tab, action, player):
    tab[((action-1)//COLS)][(action-1)%COLS] = player
    return tab

# Definimos el siguiente estado
def next_states(state, action, player):
```

```
24      state += player*3**(9-action)
25      return state
26
27  #Comprobamos si alguien ha ganado o si la partida ha terminado en empate
28  def checkresult(tab):
29      ganador = 0
30      for i in range(3):
31          if tab[i][0] == tab[i][1] == tab[i][2] != 0:
32              ganador = tab[i][0]
33          if tab[0][i] == tab[1][i] == tab[2][i] != 0:
34              ganador = tab[0][i]
35      if tab[0][0] == tab[1][1] == tab[2][2] != 0:
36          ganador = tab[0][0]
37      if tab[0][2] == tab[1][1] == tab[2][0] != 0:
38          ganador = tab[0][2]
39      return ganador
40
41  # Algoritmo Q-learning para entrenar a la maquina a jugar en ambos roles
42  def q_learning(episodes=EPISODES, alpha=0.1, gamma=0.95):
43      q_table = np.zeros((STATES, ROWS*COLS))  # Inicializar tabla Q
44
45      #Contadores de victorias, empates y derrotas (con respecto del jugador 1)
46      wins = 0
47      draws = 0
48      losses = 0
49      alpha_ini = alpha
50      epsilon_decay = 0.9999
51      epsilon = 1.0
52      #numero de episodios
53      for episode in range(episodes):
54          if episode%5000 == 0:
55              print(episode*100/episodes, "%")
56
57          #inicializamos el estado, el tablero y el turno
58          state = INI
59          tablero = tablero_inicial()
60          turn = 0
61          reward = 0
62
63          # Factor de exploracion que va decreciendo con los episodios
64          epsilon *= epsilon_decay
65          alpha = alpha_ini * (1 - episode / episodes)
66
67          # En el principio de cada episodio, el agente comienza con todas las
                posibilidades de accion
68          ACTIONS = [1,2,3,4,5,6,7,8,9]
69
70          # Esto representa una jugada de cada uno
71          while True:
72
73              turn += 1
74
75              ##########Jugador 1##########
76
77              # Decidir la accion (explorar/explotar)
78              if random.uniform(0, 1) < epsilon:
79                  action = random.choice(ACTIONS)
80              else:
81                  maxim = q_table[state][ACTIONS[0]-1]
82                  action = ACTIONS[0]
83                  for ac in range(len(ACTIONS)-1):
84                      if q_table[state][ACTIONS[ac+1]-1] > maxim:
85                          maxim = q_table[state][ACTIONS[ac+1]-1]
```

```
                        action = ACTIONS[ac+1]

            # Actualizar el estado, el tablero y quitamos la accion de las
                posibles
            next_state = next_states(state, action, 1)
            tablero = next_tab(tablero, action, 1)
            ACTIONS.remove(action)

            # Comprobamos si hay ganador
            reward = checkresult(tablero)

            # Actualizar la tabla Q de la jugada anterior
            if len(ACTIONS) == 0:
                best_next_action = 0
            else:
                valid_q_values = [q_table[next_state][a-1] for a in ACTIONS]
                best_next_action = np.min(valid_q_values)
            q_table[state][action-1] += alpha * (
                reward + gamma * best_next_action - q_table[state][action-1]
            )

            if reward > 0:
                wins += 1
                break
            elif turn == 9:
                draws += 1
                break

            # Fin turno Jugador 1
            state = next_state
            turn += 1

            ##########Jugador 2##########

            # Decidir la accion (explorar/explotar)
            if random.uniform(0, 1) < epsilon:
                action = random.choice(ACTIONS)
            else:
                minim = q_table[state][ACTIONS[0]-1]
                action = ACTIONS[0]
                for ac in range(len(ACTIONS)-1):
                    if q_table[state][ACTIONS[ac+1]-1] < minim:
                        minim = q_table[state][ACTIONS[ac+1]-1]
                        action = ACTIONS[ac+1]

            # Actualizar el estado, el tablero y quitamos la accion de las
                posibles
            next_state = next_states(state, action, 2)
            tablero = next_tab(tablero, action, -1)
            ACTIONS.remove(action) # Quitamos la casilla ya ocupada

            # Comprobamos si hay ganador
            reward = checkresult(tablero) # Comprobamos si hay ganador

            # Actualizar la tabla Q de la jugada anterior
            valid_q_values = [q_table[next_state][a-1] for a in ACTIONS]
            best_next_action = np.max(valid_q_values)
            q_table[state][action-1] += alpha * (
                reward + gamma * best_next_action - q_table[state][action-1]
            )

            if reward < 0:
                losses += 1
```

```
147                 break
148
149             state = next_state
150
151     return q_table, wins, draws, losses
152
153 # Funcion que muestra el ratio de victorias de Jug 1 y 2
154 def resultados(wins, draws, loses):
155     print("Player 1: ", wins*100/EPISODES, "%")
156     print("Draws: ", draws*100/EPISODES, "%")
157     print("Player 2: ", loses*100/EPISODES, "%")
158
159 #Funcion que muestra cuantos estados ha enfrentado la maquina
160 def mirar_estados(tabla):
161     count = 0
162     for state in range(STATES):
163         if not all(x == 0 for x in tabla[state]):
164             count += 1
165     print("Se ha enfrentado la maquina a ", count, " estados.")
166
167 #Funcion que guarda la tabla Q en un .csv
168 def guardar_tabla(tabla):
169     with open('tictactoe.csv', mode='w', newline='') as archivo:
170         escritor = csv.writer(archivo)
171         escritor.writerows(tabla)
172
173
174 #---INICIO DE PROGRAMA PRINCIPAL---#
175
176 #Ejecucion del algoritmo Q-learning
177 q_table, wins, draws, loses = q_learning()
178
179 #Mostrar resultados
180 resultados(wins, draws, loses)
181 mirar_estados(q_table)
182 #Guardar tabla Q en un .csv
183 guardar_tabla(q_table)
```

Anexo IV. Programa para la resolución de CartPole mediante DQN

Código de la función de inicialización del agente para el entrenamiento de CartPole:

```
    def __init__(self, state_dim, action_dim, replay_buffer_capacity=2000,
                 gamma=0.99, lr=25e-4, batch_size=64, target_update_freq=1):
        """
        Inicializa el agente DQN.

        Args:
            state_dim (int): Dimension del espacio de estados.
            action_dim (int): Numero de acciones posibles.
            replay_buffer_capacity (int): Capacidad maxima del Replay Buffer.
            gamma (float): Factor de descuento.
            lr (float): Tasa de aprendizaje.
            batch_size (int): Tamano del mini-batch para entrenamiento.
            target_update_freq (int): Frecuencia para actualizar la red objetivo
                .
        """
        self.state_dim = state_dim
        self.action_dim = action_dim
        self.gamma = gamma
        self.batch_size = batch_size
        self.target_update_freq = target_update_freq

        # Redes neuronales, son 2, una para la politica y otra para el objetivo
        # la red objetivo se actualiza cada "target_update_freq" pasos mientras
            que la politica
        # se actualiza en cada paso
        self.policy_net = DQN(state_dim, action_dim)
        self.target_net = DQN(state_dim, action_dim)
        self.target_net.load_state_dict(self.policy_net.state_dict())
        self.target_net.eval()

        # Optimizador
        self.optimizer = optim.Adam(self.policy_net.parameters(), lr=lr)

        # Replay Buffer
        self.replay_buffer = ReplayBuffer(replay_buffer_capacity)

        # Parametros para epsilon-greedy
        self.epsilon_start = 1.0
        self.epsilon_end = 0.1
        self.epsilon_decay = 500  # Decaimiento exponencial

        self.steps_done = 0
```

Código de la función usada para entrenar al agente:

```
def train(self):
        """
        Entrena la red neuronal utilizando un mini-batch del Replay Buffer.
        """
        if self.replay_buffer.size() < self.batch_size:
            return  # No hay suficientes muestras para entrenar

        # Muestreo aleatorio del Replay Buffer
        batch = self.replay_buffer.sample(self.batch_size)
        states, actions, rewards, next_states, dones = zip(*batch)

        # Convertir listas de numpy.ndarrays a numpy.ndarray antes de crear
            tensores para mejorar rendimiento
        states = torch.FloatTensor(np.array(states))
        actions = torch.LongTensor(np.array(actions)).unsqueeze(1)
        rewards = torch.FloatTensor(np.array(rewards)).unsqueeze(1)
        next_states = torch.FloatTensor(np.array(next_states))
        dones = torch.FloatTensor(np.array(dones)).unsqueeze(1)

        # Calcular los valores Q actuales
        current_q_values = self.policy_net(states).gather(1, actions) # Esta
            funcion invoca por detras a la funcion forward de la clase DQN

        # Calcular los valores Q objetivos usando la red objetivo
        with torch.no_grad(): # No se calculan gradientes para mejorar
            rendimiento ya que no se usan
            max_next_q_values = self.target_net(next_states).max(1)[0].unsqueeze
                (1)
            target_q_values = rewards + (self.gamma * max_next_q_values * (1 -
                dones))  # Funcion de Bellman

        # Perdida Huber entre valores actuales y objetivos
        loss = nn.SmoothL1Loss()(current_q_values, target_q_values)

        # Optimizacion
        self.optimizer.zero_grad()
        loss.backward()
        for param in self.policy_net.parameters():
            param.grad.data.clamp_(-1, 1)  # Clipping para estabilidad numerica
        self.optimizer.step()
```

Para ver el resto de código [10].