

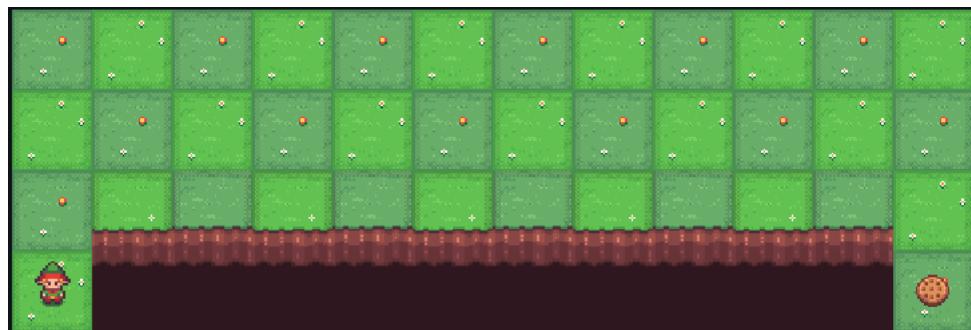
Práctica 2 - Introducción al aprendizaje por refuerzo: Experimentando con el entorno CliffWalking-v0

Sistemas Inteligentes Distribuidos (FIB-UPC)
Cuatrimestre de primavera, 2024-2025

Fernando Guirao
fernando.guirao@estudiantat.upc.edu

Nicolás Llorens
nicolas.llorens@estudiantat.upc.edu

16 de mayo de 2025



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Índice

1. Introducción	3
1.1. Proceso de decisión de Markov (MDP)	3
1.2. Valor de estado y valor de acción	4
1.3. Ecuaciones de Bellman	5
1.4. Entorno Cliff-Walking-v0	7
2. Iteración de valor	8
2.1. Experimentación	9
3. ¿Qué hacer cuando no conocemos la dinámica del MDP?	12
4. Estimación directa	16
4.1. Experimentación	16
5. Q-Learning	18
5.1. Experimentación	20
6. Conclusiones	25

1. Introducción

Esta práctica es una introducción al aprendizaje por refuerzo (RL) que nos ha servido para conocer los fundamentos teóricos de las técnicas más básicas y fundamentales de este campo, así como para ponerlas en práctica y observar de forma empírica los criterios seguidos por cada algoritmo.

En esta memoria presentamos primero cada uno de los algoritmos propuestos, para posteriormente ponerlos a prueba en un mismo entorno. El entorno en cuestión es **Cliff-Walking-v0** de la librería Gymnasium. Esta librería proporciona diversos entornos que representan procesos decisionales de Markov en los que podemos probar agentes de aprendizaje por refuerzo. El entorno que utilizaremos es de los más simples que ofrece la librería, con espacios de estados y acciones pequeños y discretos, lo cual nos permite observar con claridad el comportamiento de los algoritmos y las diferencias entre éstos.

La experimentación con los algoritmos consiste en realizar entrenamientos bajo diferentes combinaciones de parámetros e hiperparámetros (número de episodios, factor de descuento, tasa de exploración...) que posteriormente son evaluados de diferentes maneras (según la recompensa obtenida, optimalidad de la política...).

Antes de presentar los algoritmos y analizar los resultados, incluimos una breve explicación del marco teórico necesario para entender su funcionamiento.

La principal referencia para el desarrollo de esta práctica ha sido el libro *Reinforcement Learning: An Introduction* [Sutton and Barto, 2018].

1.1. Proceso de decisión de Markov (MDP)

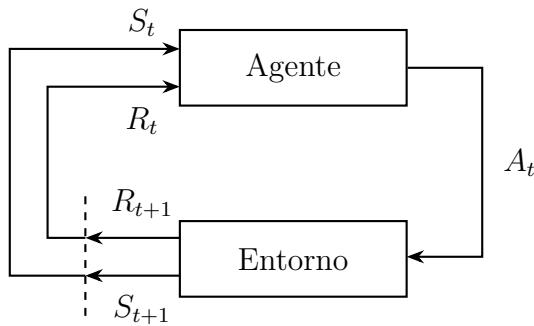


Figura 1: Interacción agente-entorno en un MDP

Un proceso de decisión de Markov (MDP) es el marco matemático que utilizamos para modelar entornos en los que un agente debe tomar decisiones de forma secuencial, afectando al estado futuro del entorno.

Un MDP está definido por la tupla $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p)$, donde:

- \mathcal{S} : Conjunto de **estados**.
- \mathcal{A} : Conjunto de **acciones**.
- $\mathcal{R} \in \mathbb{R}$: Conjunto de **recompensas**.

Además, contamos con la siguiente función de probabilidad que nos dice la probabilidad de obtener la recompensa $r \in \mathcal{R}$ e ir al estado $s' \in \mathcal{S}$ si el agente lleva a cabo la acción $a \in \mathcal{A}(s)$ estando en el estado $s \in \mathcal{S}$.

$$p(s', r | s, a) = \text{Prob}(S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a)$$

Si conocemos esta función de probabilidad, decimos que tenemos conocimiento completo del entorno (del MDP).

El agente observa S_t , y S_t cumple la **propiedad de Markov**, es decir, la probabilidad del próximo estado y recompensa sólo depende del estado actual y la acción que toma el agente (como se puede ver en la función de transición), y no del historial de estados y acciones.

Cuando el agente actúa sobre el entorno, obtenemos **trazas** de estados, acciones y recompensas.

$$s_0^m, a_0^m, r_1^m, s_1^m, a_1^m, r_2^m, s_2^m, a_2^m, \dots, r_T^m, s_T^m$$

Donde T indica el final de un **episodio**, y m indica el número de episodio en el que se generó la traza.

El comportamiento del agente lo define la **política**, que nos dice la probabilidad de tomar una acción a desde un estado s .

$$\pi(a|s)$$

Queremos una política que acumule muchas recompensas. La recompensa acumulada, que llamamos **retorno**, la expresamos como el sumatorio de recompensas futuras que obtendremos. Cada una de estas recompensas se multiplica por un **factor de descuento** $\gamma \in [0, 1]$ que nos permite priorizar las recompensas más inmediatas si es necesario.

$$G_t = \sum_{k=t+1}^T \gamma^{k-t-1} R_k = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{T-t-1} R_T$$

Nuestro objetivo es encontrar una política que maximice el retorno esperado.

$$\max_{\pi} \mathbb{E}_{\pi}[G_t]$$

1.2. Valor de estado y valor de acción

Para que un agente aprenda una política que maximice el retorno esperado, podemos proporcionarle información sobre el valor esperado de cada estado —lo que se conoce como **valor de estado**—, de modo que pueda decidir hacia qué estados le conviene ir.

Otra opción es proporcionarle el **valor de acción**, que estima la recompensa esperada de ejecutar una acción específica en un estado concreto. De este modo, el agente puede elegir directamente la acción más conveniente.

En ambos casos, disponer de estas estimaciones permite al agente aprender y seguir una **política óptima**.

La función de valor de estado se parece mucho a lo que buscamos optimizar (la recompensa esperada) sólo que desglosado por estado.

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

De manera similar, desglosando por estado y acción, tenemos la función de valor de acción.

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

Una política es óptima si consigue la mayor recompensa esperada posible para todos los estados, y también para todos los pares estado-acción.

$$\begin{aligned} v_{\pi_*}(s) &\geq v_\pi(s) && \text{para todo } s \text{ y para cualquier } \pi \\ q_{\pi_*}(s, a) &\geq q_\pi(s, a) && \text{para todo } s, a \text{ y para cualquier } \pi \end{aligned}$$

1.3. Ecuaciones de Bellman

Supongamos un MDP muy sencillo, compuesto por 5 estados.

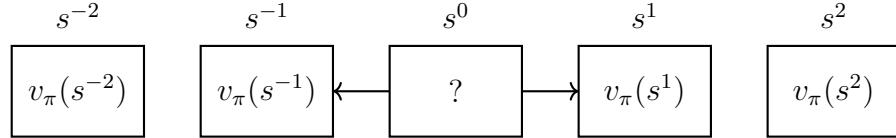


Figura 2: MDP sencillo con 5 estados

Cada casilla representa un estado. Desde cada casilla, el agente puede moverse a la izquierda y a la derecha (a no ser que tope con el final). Conocemos la función de probabilidad, el factor de descuento, y el valor de todos los estados menos el de s^0 . Vamos a ver cómo, con esta información, podemos conocer el valor del estado de s^0 .

Recordamos que la función de valor de estado bajo una política π es:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

Y que el retorno es:

$$G_t = R_{t+1} + \gamma \underbrace{(R_{t+2} + \gamma R_{t+3} + \dots)}_{G_{t+1}}$$

Como desde s^0 hay dos acciones posibles (izquierda y derecha), el valor de estado se puede expresar como una media ponderada de los valores esperados al tomar cada acción:

$$v_\pi(s^0) = \sum_{a \in \{\leftarrow, \rightarrow\}} \pi(a|s^0) (\underbrace{\mathbb{E}_\pi[G_t | s^0, a]}_{q_\pi(s^0, a)})$$

Ahora desarrollamos la esperanza para una acción concreta, por ejemplo, la de ir a la derecha:

$$\begin{aligned} \mathbb{E}_\pi[G_t | s^0, \rightarrow] &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | s^0, \rightarrow] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | s^0, \rightarrow] \\ &= \sum_{s', r} p(s', r | s^0, \rightarrow) [r + \gamma v_\pi(s')] \end{aligned}$$

Nota: Aunque $G_{t+1} \neq v_\pi(S_{t+1})$ como variables aleatorias, sí son iguales en esperanza:

$$\mathbb{E}_\pi[G_{t+1} | s^0, \rightarrow] = \mathbb{E}_\pi[v_\pi(S_{t+1}) | s^0, \rightarrow]$$

Esto se debe a que $v_\pi(s') = \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s']$, y al tomar la esperanza total sobre los posibles S_{t+1} , se obtiene el mismo valor.

Hacemos lo mismo para la acción de ir a la izquierda y luego combinamos los dos resultados según la política:

$$v_\pi(s^0) = \pi(\rightarrow | s^0) \sum_{s', r} p(s', r | s^0, \rightarrow) [r + \gamma v_\pi(s')] + \pi(\leftarrow | s^0) \sum_{s', r} p(s', r | s^0, \leftarrow) [r + \gamma v_\pi(s')]$$

De este ejemplo, podemos extraer las siguientes ecuaciones para cualquier política π , todos los estados $s \in \mathcal{S}$ y todas las acciones $a \in \mathcal{A}(s)$.

Por un lado, el **valor de un estado** es el valor de todas las acciones que se pueden tomar desde este, ponderadas por la política.

$$v_\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(a|s) q_\pi(s, a) \tag{1}$$

Por otro lado, el **valor de una acción** equivale a la suma de los valores de los posibles estados a los que puede ir, más la recompensa inmediata por ir a cada uno de ellos, todo ello ponderado por la probabilidad de ir a cada estado y recibir la respectiva recompensa inmediata.

$$q_\pi(s, a) = \sum_{\substack{s' \in \mathcal{S} \\ r \in \mathcal{R}}} p(s', r | s, a) [r + \gamma v_\pi(s')] \tag{2}$$

Como se puede observar, existe una relación mutua entre el valor de estado y el valor de acción: el primero se define como una combinación ponderada de valores de acción según la política, y el segundo depende de las recompensas inmediatas y del valor del estado siguiente. Al sustituir una definición en la otra, obtenemos una forma recursiva de expresar ambos valores. Estas expresiones recursivas son conocidas como **las ecuaciones de Bellman**, y constituyen la base teórica de muchos algoritmos de aprendizaje por refuerzo.

Ecuación de Bellman para el valor de estado v_π

Sustituyendo (2) en (1) obtenemos:

$$v_\pi(s) = \sum_{a \in A(s)} \pi(a|s) \sum_{\substack{s' \in S \\ r \in R}} p(s', r|s, a) [r + \gamma v_\pi(s')]$$

Ecuación de Bellman para el valor de acción q_π

Sustituyendo (1) en (2) obtenemos:

$$q_\pi(s, a) = \sum_{\substack{s' \in S \\ r \in R}} p(s', r|s, a) \left[r + \gamma \sum_{a' \in A(s')} \pi(a'|s') q_\pi(s', a') \right]$$

Estas ecuaciones son el nexo que conecta todo el espacio de estados, permitiendo que un agente lo atraviese de forma inteligente. Aplicarlas equivale, en esencia, a realizar programación dinámica (técnica introducida por Richard Bellman), ya que permiten actualizar el valor de un estado a partir del valor de otros adyacentes. De este modo, el valor de cada estado puede propagarse progresivamente según una política dada.

Si se está siguiendo la política óptima, las expresiones (1) y (2) pasan a ser:

$$\begin{aligned} v_*(s) &= \max_{a \in A(s)} q_*(s, a) \\ q_*(s, a) &= \sum_{\substack{s' \in S \\ r \in R}} p(s', r|s, a) [r + \gamma v_*(s')] \end{aligned}$$

ya que la política óptima busca maximizar retornos.

1.4. Entorno Cliff-Walking-v0

La figura 3 muestra el entorno sobre el que probamos los algoritmos. Está representado por una rejilla donde cada casilla es un estado. Tenemos un estado inicial S (*Start*) y un estado final G (*Goal*).

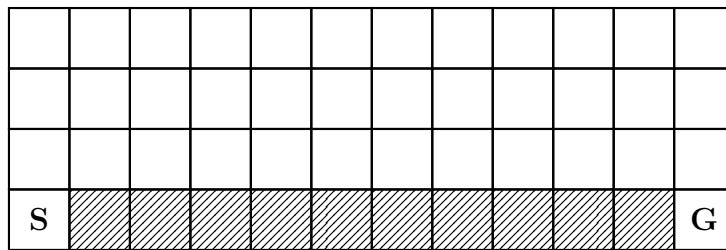


Figura 3: Entorno Cliff-Walking-v0 con zona del acantilado marcada

Las acciones permiten al agente moverse a izquierda, derecha, arriba o abajo (siempre que no tope con un borde). Cada movimiento penaliza al agente con una recompensa de -1 .

Las casillas rayadas en la figura 3 entre S y G son el acantilado. Caer en cualquiera de esas casillas conlleva una recompensa de -100 .

$$R_t = \begin{cases} -100 & \text{si } S_t = \text{rayado} \\ -1 & \text{si } S_t = \square \end{cases}$$

El entorno por defecto es determinista, pero si lo declaramos con el parámetro `is_slippery = True` entonces el agente a veces se moverá en la dirección opuesta a la deseada.

Nota: En el funcionamiento original del entorno, al caer en el acantilado el agente recibe una recompensa de -100 y es transportado de nuevo al estado inicial S. Esto implica que al calcular $q(s, a)$ para una acción hacia el acantilado, se utiliza $v(S)$ como valor del siguiente estado, lo cual puede resultar poco natural. Por ello, decidimos tratar las casillas del acantilado como estados terminales. Esta modificación mantiene el objetivo y la dificultad del entorno sin alterar su naturaleza.

2. Iteración de valor

El algoritmo de iteración de valor forma parte de la familia de algoritmos de **iteración de política generalizada (GPI)**. Según Sutton y Barto «GPI se refiere a la idea general de permitir que los procesos de evaluación de política y mejora de política interactúen, independientemente del nivel de granularidad u otros detalles de ambos procesos. Casi todos los métodos de aprendizaje por refuerzo se pueden describir adecuadamente como GPI»

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$

Figura 4: Ejemplo de Generalized Policy Iteration (GPI)

La **evaluación de política** consiste en calcular todos los valores de la función de valor de estado $v_\pi(s)$ o la función de valor de acción $q_\pi(s, a)$ siguiendo una política fija π en un MDP. Este proceso normalmente se repite hasta que el valor de los estados no cambia o lo hace muy poco, de modo que aproximamos el valor real de cada estado siguiendo esa política.

Algorithm 1 Evaluación de política

- 1: Inicializar $V_0(s) \leftarrow 0$ (o valores aleatorios), $\forall s \in \mathcal{S}$
 - 2: **for** $t = 1$ **hasta** t_{MAX} **do**
 - 3: **for** cada estado $s \in \mathcal{S}$ **do**
 - 4: $V_t(s) \leftarrow \sum_{a \in \mathcal{A}(s)} \pi(a|s) \sum_{\substack{s' \in \mathcal{S} \\ r \in \mathcal{R}}} p(s', r | s, a) [r + \gamma V_{t-1}(s')] \triangleright$ Actualizar el valor de s
 - 5: **end for**
 - 6: **end for**
-

La **mejora de política** es simplemente mejorar una política π dada una función de valor de estado $v_\pi(s)$ acorde con esta política. Definimos entonces una nueva política que escoja siempre la acción con mayor valor (recordemos que podemos calcular el valor de los pares estado-acción en función de los valores de los estados vecinos).

$$\pi'(s) = \operatorname{argmax}_a q_\pi(s, a)$$

Como mencionábamos antes, los algoritmos GPI combinan evaluación y mejora de la política con el objetivo de alcanzar una política óptima. En la práctica, esperar a que los valores de estado converjan por completo durante la evaluación puede resultar ineficiente. El algoritmo de **iteración de valor** soluciona esto mejorando la política en cada iteración de evaluación, sin necesidad de esperar a la convergencia total.

Algorithm 2 Iteración de valor

```

1: Inicializar  $V_0(s) \leftarrow 0$  (o valores aleatorios),  $\forall s \in \mathcal{S}$ 
2: for  $t = 1$  hasta  $t_{\text{MAX}}$  do
3:   for cada estado  $s \in \mathcal{S}$  do
4:      $V_t(s) \leftarrow \max_{a \in \mathcal{A}(s)} \sum_{\substack{s' \in \mathcal{S} \\ r \in \mathcal{R}}} p(s', r | s, a)[r + \gamma V_{t-1}(s')] \triangleright$  Elegir la mejor acción en  $s$ 
5:   end for
6: end for
7: Derivar la política óptima:  $\pi^*(s) \leftarrow \arg \max_{a \in \mathcal{A}(s)} \sum_{s', r} p(s', r | s, a)[r + \gamma V_{t_{\text{MAX}}}(s')]$ 

```

2.1. Experimentación

Vamos a ver el algoritmo de iteración de valor en acción. Aplicar este algoritmo requiere conocer completamente la dinámica del MDP, lo cual nos garantiza que podremos encontrar la política y el valor óptimos.

Como primer experimento, hacemos que el entorno sea determinista. En este caso, veremos que en pocas iteraciones la iteración de valor converge a $v_*(s)$ y $\pi_*(s)$.

Utilizamos un factor de descuento $\gamma = 1$, ya que las recompensas negativas en cada paso ya incentivan al agente a alcanzar el objetivo lo antes posible, penalizando trayectorias largas sin necesidad de desvalorizarlas con el tiempo.

El umbral de convergencia se fija como $\theta = 1 \cdot 10^{-3}$. En la práctica, observaremos que al alcanzarlo, la diferencia entre iteraciones es ya despreciable y el algoritmo ha convergido.

Después de tan solo 15 iteraciones, el algoritmo approxima con precisión los valores óptimos de cada estado, y con ello la política óptima, que al no haber determinismo es bastante evidente.



Figura 5: Valores aprendidos por iteración de valor con determinismo.

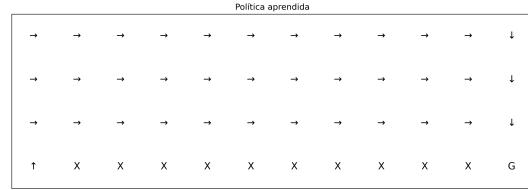


Figura 6: Política aprendida por iteración de valor con determinismo.

En la figura 5, podemos interpretar el valor de cada estado como los pasos que quedan para llegar al objetivo. Por otro lado, la figura 6 refleja claramente el camino más corto. Fácil, ¿no?. Vamos a introducir ahora indeterminismo con el parámetro `is_slippery = True`. Recordemos que, de este modo, si queremos ir por ejemplo hacia la derecha, tendremos $1/3$ de probabilidad de ir a donde queremos, pero también esa misma probabilidad de ir hacia arriba o hacia abajo (las direcciones perpendiculares a ir hacia la derecha). Lo que podemos deducir de esto es que ahora moverse cerca del acantilado conllevará un riesgo y eso debería verse reflejado en los valores de los estados cercanos al acantilado.

Ejecutamos ahora el algoritmo con el mismo factor de descuento y umbral de convergencia. Tras 127 iteraciones, el algoritmo llega al umbral de convergencia, pero si nos fijamos en la figura 7, vemos que a partir de las 50 iteraciones la política ya se mantiene estable.

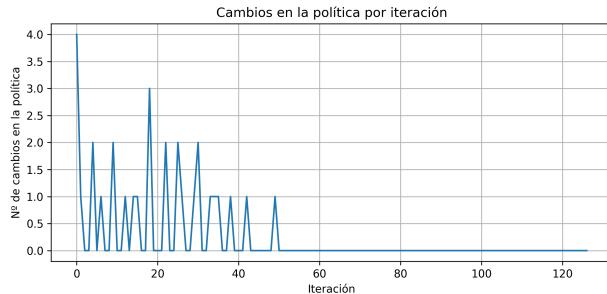


Figura 7: Número de cambios en la política por iteración.

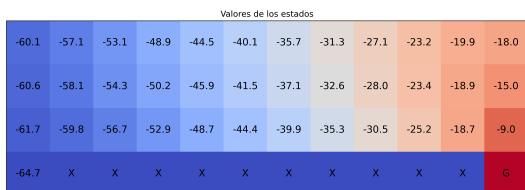


Figura 8: Valores aprendidos por iteración de valor.

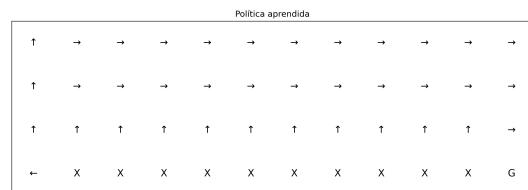


Figura 9: Política aprendida por iteración de valor.

El efecto del indeterminismo es claro. Los valores de los estados al borde del acantilado tienen, en general, peor valor esperado que los de la fila superior. Esto se debe probablemente a que en esos estados, un movimiento a la izquierda o a la derecha conlleva el riesgo de caer al acantilado, por lo que la mejor acción es ir hacia arriba y alejarse del acantilado. Viendo la política aprendida, podemos corroborar esta idea. De hecho, podemos resumir el comportamiento del agente en que cada vez que esté en una casilla colindante al acantilado, ejecutará la acción que conlleve riesgo 0 y, una vez a salvo, tratará de moverse en dirección hacia el objetivo.

Podemos también ver cómo mejora la política con el paso de las iteraciones del algoritmo de iteración de valor. La figura 10 muestra los resultados de probar, por cada iteración (es decir, por cada mejora de política) la política aprendida hasta el momento simulando 200 episodios (cada uno con un máximo de 200 pasos) y calculando el retorno total promedio.

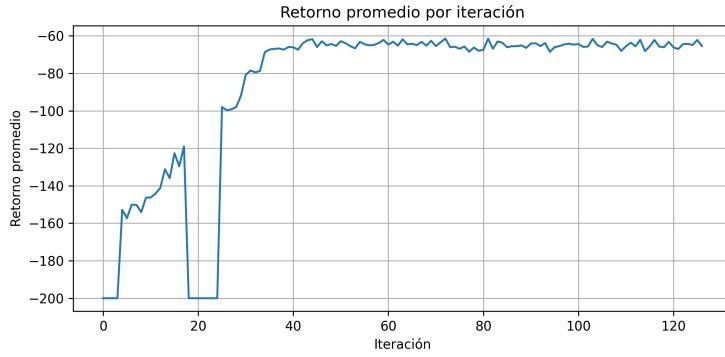


Figura 10: Retorno promedio por iteración ($\gamma = 1$).

Observamos algo interesante, alrededor de la iteración 20, el retorno promedio cae drásticamente, y ampliando el límite de pasos por episodio hemos visto que sigue cayendo hasta el límite. Observando la política adoptada durante esas iteraciones, nos damos cuenta de que la política condena al agente a quedarse atrapado en la primera columna del entorno, es decir, siempre se mueve a la izquierda. Esto sucede probablemente porque la función de valor $V(s)$ aproximada en ese momento está lejos de $v_\pi(s)$.

Probamos a decrementar el factor de descuento ($\gamma = 0,90$), y como vemos en la figura 11, durante esas iteraciones se dejan de adoptar políticas problemáticas.



Figura 11: Retorno promedio por iteración ($\gamma = 0,9$).

La zona de convergencia (del retorno promedio) es la misma que antes, sólo que ahora no se aprenden políticas que estancan al agente (excepto al principio). También vemos que el umbral de convergencia del algoritmo se alcanza antes, lo cual es razonable ya que un factor de descuento más agresivo suaviza los valores esperados de cada estado y es más fácil llegar a tener diferencias pequeñas entre iteraciones.

En cualquier caso, la política alcanzada es la misma, y podemos decir con seguridad que es la óptima. Además, el algoritmo de iteración de valor por sí sólo, sin tener en cuenta las simulaciones de episodios que hacemos en cada iteración para ver cómo rinde la política, se ejecuta muy rápido, ya que como vimos en clase de teoría, tiene una cota temporal polinómica respecto al número de estados y acciones $O(t_{MAX}S^2A)$.

Por lo tanto la conclusión es que conocer la dinámica del MDP, es decir, $p(s', r | s, a)$ es lo mejor que nos puede ocurrir, pero en la mayoría de situaciones es algo utópico.

Resumen: Iteración de valor converge rápidamente en entornos deterministas. Con indeterminismo, la política aprendida toma precauciones adicionales. Reducir γ evita políticas problemáticas en fases tempranas y acelera la convergencia numérica sin cambiar la política final.

3. ¿Qué hacer cuando no conocemos la dinámica del MDP?

Así como en las secciones de la introducción, esta sección nuevamente es teórica. Condensa bastante información, pero es un intento de situar al lector en lo relacionado a aquellas técnicas de RL que empleamos cuando no conocemos la función de probabilidad del MDP y llegar hasta el algoritmo de estimación directa.

Cuando no tenemos acceso a $p(s', r | s, a)$ y no podemos resolver las ecuaciones de Bellman, lo único que nos queda es simular episodios con alguna política arbitraria y, viendo la recompensa que obtenemos cada vez que estamos en un estado concreto y ejecutamos

una determinada acción, realizar un promedio para saber el valor de ese par estado-acción siguiendo esa política. Luego, podemos mejorar nuestra política siguiendo los pares estado-acción que nos han dado más recompensa, o quizás explorar otros nuevos que no hemos probado.

Esta idea intuitiva es la que hay detrás de los **métodos de Monte Carlo** (MC). De manera más formal, si simulamos M episodios, obtenemos las trazas:

$$s_0^m, a_0^m, r_1^m, s_1^m, a_1^m, r_2^m, \dots, s_{T_m}^m \quad \text{para } m = 1, \dots, M$$

Con las trazas podemos usar promedios para calcular $q_\pi(s, a)$. Recordemos que el retorno esperado de un par estado-acción es:

$$\mathbb{E}_\pi[G_t | S_t = s, A_t = a] \approx \frac{1}{C(s, a)} \sum_{m=1}^M \sum_{\tau=0}^{T_m-1} \mathbb{I}[s_\tau^m = s \wedge a_\tau^m = a] \cdot g_\tau^m = Q(s, a)$$

Como vemos, por cada traza, sumamos todos los retornos vistos cada vez que estábamos en el estado deseado y lo dividimos por el número de veces que hemos visto ese estado. Recordemos también el retorno observado desde el paso τ del episodio m :

$$g_\tau^m = r_{\tau+1}^m + \gamma r_{\tau+2}^m + \gamma^2 r_{\tau+3}^m + \dots$$

El problema de esta fórmula es que requiere almacenar todas las trazas para luego procesarlas. Para evitarlo, preferimos actualizar nuestra estimación cada vez que terminamos un episodio. Esto nos lleva a la siguiente regla de actualización incremental:

$$Q(s_t^m, a_t^m) \leftarrow Q(s_t^m, a_t^m) + \frac{1}{C(s_t^m, a_t^m)} (g_t^m - Q(s_t^m, a_t^m))$$

Finalmente, podemos sustituir el término $\frac{1}{C(s_t^m, a_t^m)}$ por una constante $\alpha \in (0, 1]$ que llamamos **step size**. Esto simplifica el análisis teórico, permite usar técnicas online y sigue garantizando la convergencia bajo ciertas condiciones. Nos queda la regla de actualización característica de Monte Carlo Control:

$$Q(s_t^m, a_t^m) \leftarrow Q(s_t^m, a_t^m) + \alpha (g_t^m - Q(s_t^m, a_t^m))$$

Con esta regla de actualización, a continuación presentamos el algoritmo de MC Control, que consiste en simular una traza, actualizar con ella el valor de cada par estado-acción, actualizar la política según los nuevos valores de los pares estado-acción (dando también probabilidad a realizar acciones exploratorias, no sólo siguiendo los valores nuevos).

Algorithm 3 Constant- α Monte Carlo Control

Require: $\epsilon > 0$ ▷ Probabilidad de tomar una acción aleatoria
Require: $\alpha \in (0, 1]$ ▷ Step size
Require: $M \in \mathbb{N}$ ▷ Número total de episodios

- 1: Inicializar $\pi \leftarrow$ una política ϵ -soft arbitraria
- 2: Inicializar $Q(s, a)$ arbitrariamente para todo $s \in \mathcal{S}, a \in \mathcal{A}(s)$
- 3: **for** $m = 1$ **to** M **do**
- 4: Simular episodio bajo π : obtener $s_0^m, a_0^m, r_1^m, \dots, a_{T_m-1}^m, r_{T_m}^m$
- 5: **for** $t = 0$ **to** $T_m - 1$ **do**
- 6: Calcular el retorno: $g_t^m = r_{t+1}^m + \gamma r_{t+2}^m + \dots$
- 7: $Q(s_t^m, a_t^m) \leftarrow Q(s_t^m, a_t^m) + \alpha(g_t^m - Q(s_t^m, a_t^m))$
- 8: **end for**
- 9: $\pi \leftarrow \epsilon$ -greedy(Q)
- 10: **end for**

El algoritmo que actualiza la política de forma ϵ -greedy es:

Algorithm 4 Actualización ϵ -greedy de la política

- 1: **for all** $s \in \mathcal{S}$ **do**
- 2: $a^* \leftarrow \arg \max_a Q(s, a)$ ▷ Desempatar arbitrariamente si hay empate
- 3: **for all** $a \in \mathcal{A}(s)$ **do**
- 4: **if** $a = a^*$ **then**
- 5: $\pi(a | s) \leftarrow 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|}$
- 6: **else**
- 7: $\pi(a | s) \leftarrow \frac{\epsilon}{|\mathcal{A}(s)|}$
- 8: **end if**
- 9: **end for**
- 10: **end for**

Si usamos MC Control, debemos esperar que un episodio termine antes de poder actualizar los valores de los pares estado-acción. Esto puede suponer un problema si los episodios son demasiado largos. El aprendizaje puede ser más lento ya que dentro de un episodio puede haber información útil sobre recompensas y transiciones que posponemos aprender hasta que éste termine. Cambiar esto puede hacer que la política cambie durante el propio episodio, lo que cambia (y seguramente mejora) cómo exploramos ese episodio.

En la regla de actualización de MC Control, el retorno del par estado-acción (cuyo valor queremos estimar) en una determinada traza ((g_t^m)) es lo que guía el aprendizaje, pero conocer ese retorno implica tener acceso a toda la traza (para saber las recompensas). Pero, ¿y si actualizamos $Q(s_t, a_t)$ cuando obtenemos una nueva recompensa y el resto del retorno esperado lo estimamos usando $Q(s_{t+1}, a_{t+1})$ (es decir, el retorno esperado del siguiente par estado-acción)?

En vez de usar g_t^m , usamos $r_{t+1}^m + \gamma Q(s_{t+1}^m, a_{t+1}^m)$. Por lo que la regla de actualización pasa a ser:

$$Q(s_t^m, a_t^m) \leftarrow Q(s_t^m, a_t^m) + \alpha(r_{t+1}^m + \gamma Q(s_{t+1}^m, a_{t+1}^m) - Q(s_t^m, a_t^m))$$

También podemos esperar a obtener más recompensas, en vez de sólo una. Así, podemos generalizar este ajuste:

$$g_{t:t+n}^m = r_{t+1}^m + \cdots + \gamma^{n-1} r_{t+n}^m + \gamma^n Q(s_{t+n}^m, a_{t+n}^m)$$

Luego:

$$Q(s_t^m, a_t^m) \leftarrow Q(s_t^m, a_t^m) + \alpha(g_{t:t+n}^m - Q(s_t^m, a_t^m))$$

Esto es en esencia el **aprendizaje por diferencias temporales** (TD(n)), donde n indica el número de pasos que esperamos para actualizar (o, visto de otro modo, el lag que introducimos al actualizar). Nótese que TD(∞) Control = MC Control ¹.

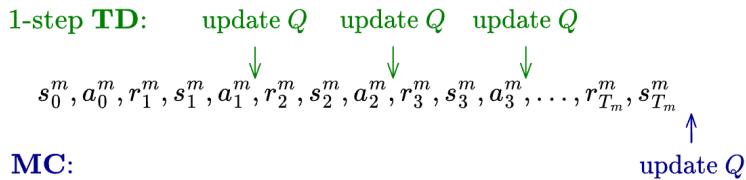


Figura 12: Diferencia de actualización entre TD(1) y MC.

Si recordamos el algoritmo de MC Control 3, vemos que la política que genera los datos es la misma que la política que queremos mejorar. Cuando esto ocurre, decimos que el algoritmo es **on-policy**, y cuando usamos una política distinta para cada cosa decimos que el algoritmo es **off-policy**.

TD(n) Control, en su versión on-policy, es conocido también como **SARSA**. El nombre proviene de la tupla de estados y acciones $((s_t^m, a_t^m, r_{t+1}^m, s_{t+1}^m, a_{t+1}^m))$ que utilizamos en la regla de actualización de TD(1) Control (ó 1-step SARSA).

En el libro de Sutton y Barto, se hace una comparativa entre MC Control TD(1) Control utilizando **batch training**. Esto consiste en que a partir de un conjunto fijo de episodios ya simulados, los procesamos repetidamente actualizando los valores de $Q(s, a)$ hasta que convergen. Cuando utilizamos *batch training* decimos que el aprendizaje es **offline**. En este caso, naturalmente, la política que genera los datos es diferente de la política que aprendemos, por lo que el aprendizaje es off-policy.

Lo interesante es ver que, bajo estas condiciones, MC Control actualiza $Q(s, a)$ para que se ajuste al retorno observado en los datos. Es decir, minimiza el error cuadrático medio respecto a los retornos vistos. Esto equivale a hacer la media de los g_t^m observados cuando

¹Cuando decimos “Control” nos referimos a la versión de optimización del algoritmo, la que busca obtener la política óptima, por lo tanto estimamos Q . En las versiones “Evaluation”, la política es fija, simplemente nos fijamos en los estados y recompensas que vamos obteniendo como si se tratase de un MRP (Markov Reward Process) y estimamos V

se visita (s, a) . Por el otro lado TD Control actualiza $Q(s, a)$ hacia la recompensa inmediata más la estimación del futuro. A nivel batch, esto es equivalente a resolver el MDP implícito que generan las frecuencias empíricas de transición observadas:

$$\hat{p}(s', r|s, a) = \frac{\# \text{ veces que } (s, a) \rightarrow (s', r)}{\# \text{ veces que se tomó } (s, a)} \quad (3)$$

Esto define un modelo empírico, y TD(1) Control offline termina resolviendo la ecuación de Bellman para ese modelo estimado, sin construir explícitamente \hat{p} :

$$Q(s, a) = \sum_{s', r} \hat{p}(s', r|s, a) [r + \gamma Q(s', \pi(s'))]$$

En definitiva TD(1) Control (ó 1-step SARSA) ajusta los valores para ser coherentes con las ecuaciones de Bellman (maximizar el *likelihood* del MDP), lo cual más eficaz si el entorno es verdaderamente un MDP. MC Control ajusta los valores para coincidir con los retornos vistos, lo cual es robusto pero puede desaprovechar estructura, es decir, es una ventaja cuando el entorno es ruidoso o ligeramente no-Markoviano, ya que no se apoya tanto en la estructura del modelo.

4. Estimación directa

Del apartado anterior, deducimos que el método que implementaremos en este apartado converge a la misma solución que 1-step SARSA en su versión *offline*, una técnica *model-free*. En cambio, para realizar **estimación directa** vamos a construir explícitamente el MDP empírico a partir de los datos observados. Una vez estimadas las probabilidades de transición y recompensas, podremos aplicar directamente nuestro algoritmo de iteración de valor, ya que dispondremos de la dinámica completa del entorno y, por tanto, podremos usar las ecuaciones de Bellman.

4.1. Experimentación

Simularemos un número determinado de episodios. En cada episodio el agente recoge experiencia actuando de forma aleatoria. Tenemos un contador por cada transición $(s, a) \rightarrow (s', r)$ observada, que vamos incrementando cada vez que volvamos a observarla. Con eso podemos aproximar la función de probabilidad tal y como muestra la expresión 3. Es importante recalcar que **no usamos la función de probabilidad del MDP**.

Cómo vimos en la sección 2.1, el retorno promedio logrado con la política óptima se situaba en algo menos de -60. Vamos a probar a estimar \hat{p} con diferentes números de episodios, y con cada estimación haremos iteración de valor. Vamos a ver cuánto se acercan las políticas aprendidas en cada caso a ese retorno promedio.

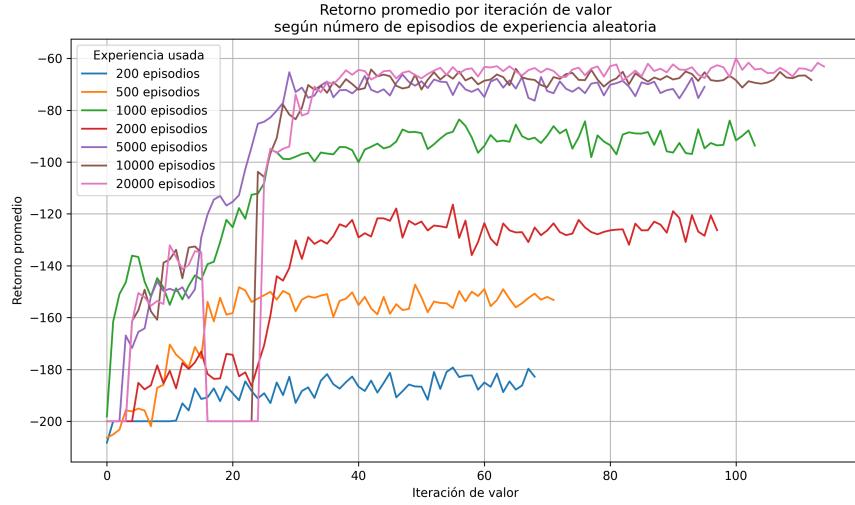


Figura 13: Retorno promedio por iteración de valor según experiencia usada para estimar \hat{p} ($\gamma = 1$).

Vemos que en las ejecuciones hechas con menos de 1000 episodios, la función de probabilidad estimada está todavía lejos de la realidad del MDP y los retornos son menores que -100, lo cual indica que el agente cae a veces al acantilado. Pero llama la atención que este rendimiento pobre también ocurre con 2000 episodios, y es que esto es una consecuencia de la estocasticidad de este experimento, dada por el muestreo aleatorio de las acciones y también la aleatoriedad del desenlace de cada acción. Por lo tanto, hacer más simulaciones aleatorias no siempre garantiza calidad en la experiencia y por ello un mejor aprendizaje de p .

Con 5000 , 10000 y 20000 episodios, sí que el agente logra ser lo suficientemente inteligente para no caer al acantilado, pero si nos fijamos por ejemplo en la política aprendida con 5000 episodios, vemos que no es exactamente la óptima, mientras que con 20000 episodios sí la conseguimos (en este experimento).



Figura 14: Política aprendida estimando \hat{p} con 5000 episodios.

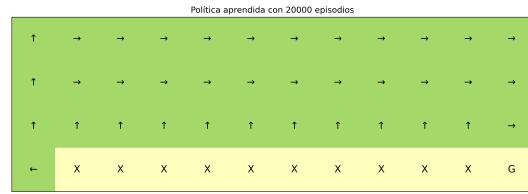


Figura 15: Política aprendida estimando \hat{p} con 20000 episodios.

De nuevo, la estocasticidad antes mencionada no nos garantiza lograr una política idéntica a la óptima, pero sí que es mucho más probable lograrla con valores altos del número

de episodios. Además, aunque la política no sea idéntica, con 10000 o 20000 episodios, ya conseguimos una política que nos da los mismos retornos que con la política óptima, lo cual indica que aquella acción en la que diferimos de la política óptima no tiene tanta relevancia para el objetivo del agente.

Otra forma de ver cómo, en general, un número de episodios mayor nos hace comprender mejor el MDP es viendo el error promedio normalizado del valor de cada estado $V(s)$ respecto a su valor óptimo $v_*(s)$ que consideramos que es el de la figura 8.

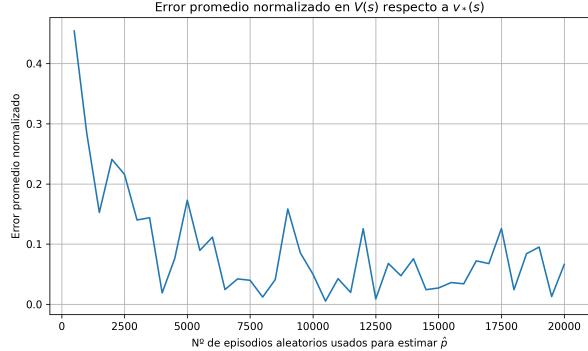


Figura 16: Error promedio normalizado de $V(s)$ vs $v_*(s)$ ($\gamma = 1$).

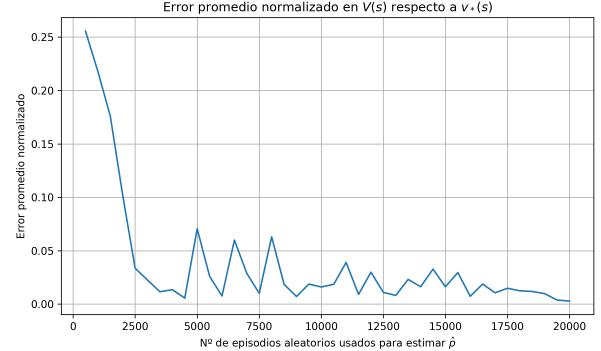


Figura 17: Error promedio normalizado de $V(s)$ vs $v_*(s)$ ($\gamma = 0,9$).

Hemos probado con dos valores distintos para el factor de descuento ya que, pese a que con $\gamma = 1$ nos acercamos a la zona de convergencia donde $V(s) \approx v_*(s)$ hay bastante más ruido que si lo hacemos con $\gamma = 0,9$. Deducimos que esto ocurre porque cuanto mayor sea γ , mayor será la sensibilidad del valor estimado a errores en la estimación del modelo, generando más ruido y varianza. Por eso, aunque $\gamma = 1$ sea teóricamente óptimo, $\gamma < 1$ da estimaciones más estables cuando \hat{p} es aproximado.

Resumen: La estimación directa permite aproximar el modelo $p(s', r|s, a)$ a partir de experiencia. Con suficientes episodios, logramos políticas y valores cercanos a los óptimos, pero la calidad de la estimación depende tanto de la cantidad como de la diversidad de las transiciones observadas. Con $\gamma = 1$ se alcanzan los valores correctos, pero con más ruido; usar $\gamma < 1$ produce estimaciones más estables con modelos aproximados.

5. Q-Learning

Para entender este algoritmo podemos partir de 1-step SARSA. Hacemos un ajuste en el target de la regla de actualización (lo que guía el aprendizaje).

$$\begin{array}{c}
 r_{t+1}^m + \gamma Q(s_{t+1}^m, \color{red}a_{t+1}^m) \\
 \downarrow \\
 r_{t+1}^m + \gamma \max_a Q(s_{t+1}^m, \color{red}a)
 \end{array}$$

Figura 18: Modificación del target cuando pasamos de 1-step SARSA a Q-Learning.

La diferencia es que ahora, en vez de usar el valor del siguiente par estado-acción, usamos el mayor valor de acción para actualizar el valor de un determinado par estado-acción.

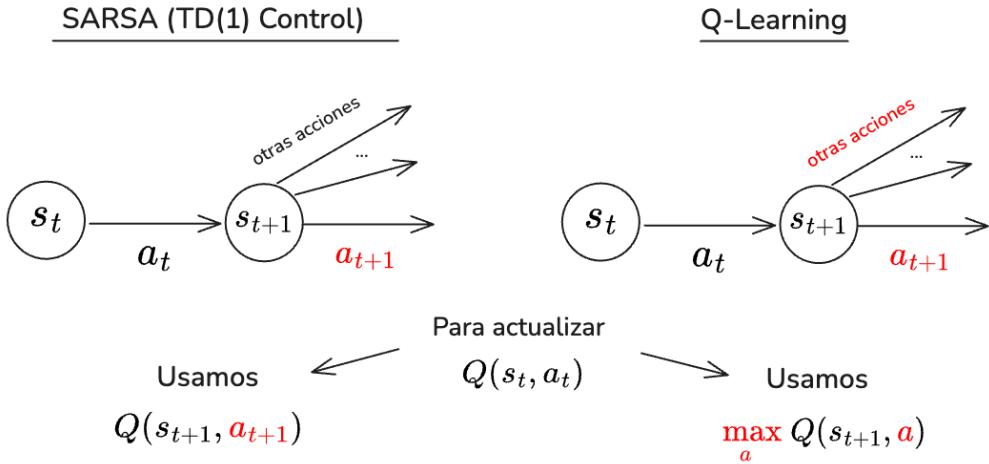


Figura 19: Representación de la diferencia entre 1-step SARSA y Q-Learning.

El uso del operador máximo convierte a Q-Learning en un método off-policy. Aunque los datos se recojan siguiendo una política exploratoria, como una ϵ -greedy, a la hora de actualizar los valores se asume que el agente siempre elige la mejor acción posible. Por eso, el objetivo de Q-Learning es estimar q_* , mientras que en SARSA es q_π , el valor bajo la política actual. Ambos algoritmos pueden converger a q_* , pero SARSA lo hace de forma más conservadora, ajustando los valores en función del comportamiento real del agente, mientras que Q-Learning actualiza como si se hubiera actuado de forma óptima en todo momento, incluso cuando no ha sido así. En el caso de SARSA, la convergencia a q_* se da porque la política ϵ -greedy que sigue se va mejorando progresivamente durante el aprendizaje, acercándose cada vez más a la óptima. En cambio, Q-Learning se acerca a q_* de forma más agresiva, ya que sus actualizaciones están guiadas directamente por el valor máximo estimado, no por las acciones realmente ejecutadas.

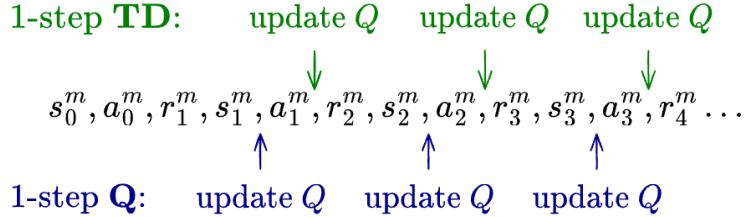


Figura 20: Diferencia de actualización entre TD(1) (1-step SARSA) y Q-Learning.

Las actualizaciones ahora ocurren antes de a_{t+1} porque esa acción no forma parte de la regla de actualización.

5.1. Experimentación

En Q-Learning, usamos un nuevo parámetro, que es α .

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a Q(s_{t+1}, a))$$

Este parámetro pondera el peso que le damos al conocimiento extra que adquirimos en cada paso de cada episodio. Cuanto mayor es α , más peso le damos a la nueva experiencia observada de cara a actualizar nuestra función de valor de acción. Por ello, este parámetro es conocido como **step size**, ya que nos permite regular cuán sensible a las nuevas experiencias es nuestra estimación (si cambia mucho, da pasos más "grandes", si sólo dejamos que cambie poco, da pasos más pequeños).

Para observar esto —y aprovechando que Q-Learning es un método online— podemos graficar la estimación que hace el algoritmo de $V(s)$ frente al valor óptimo de cada estado $v_*(s)$, que nuevamente obtenemos mediante iteración de valor (que conoce el MDP). Mostraremos cómo evoluciona el error promedio absoluto de $V(s)$ episodio a episodio.

Es importante destacar que en Q-Learning cada episodio aporta experiencia y aprendizaje al mismo tiempo, mientras que en el método de estimación directa, los episodios se simulan bajo una política aleatoria y se usan exclusivamente para estimar \hat{p} y luego aplicar las ecuaciones de Bellman.

Aunque los gráficos que presentamos a continuación son similares a los de las figuras 16 y 17, la forma de generarlos es muy distinta. En estimación directa, cada punto requiere ejecutar el algoritmo completo de iteración de valor sobre un modelo distinto (una \hat{p} distinta). Esto produce resultados más discretos. En cambio, en Q-Learning, el aprendizaje es incremental: si ejecutamos el algoritmo durante 5000 episodios, obtenemos una estimación de $V(s)$ en cada uno de los episodios < 5000. Esto da lugar a gráficos más continuos y, además, con menor coste computacional.

Una vez aclarado esto, veamos cómo evoluciona el error de $V(s)$ conforme avanzan los episodios y según valores distintos de α . Usaremos $\gamma = 1$ como factor de descuento y dejaremos que Q-Learning aprenda durante 5000 episodios.

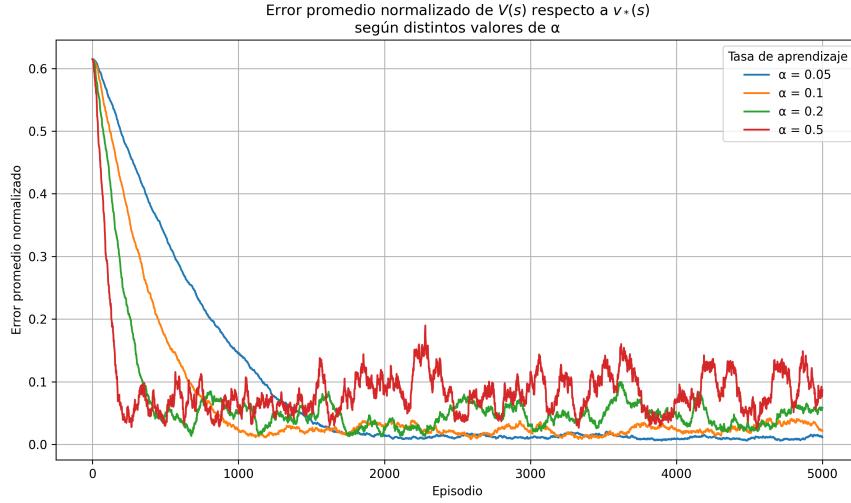


Figura 21: Evolución del error de $V(s)$ respecto a $v_*(s)$ para varios valores de α ($\gamma = 1$).

La figura 21 muestra claramente lo que decíamos. Cuando el step size es más grande, el algoritmo da pasos más grandes hacia el área de convergencia, pero de forma imprecisa y con bastante ruido. Sin embargo, usando el menor valor de α , se tardan más episodios en llegar a la zona de convergencia, pero la estimación es mucho más precisa y cercana a error nulo.

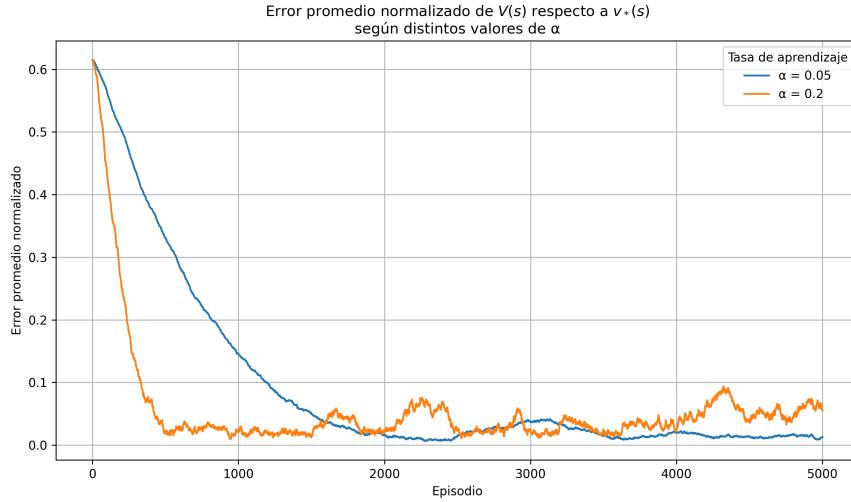


Figura 22: Evolución del error de $V(s)$ respecto a $v_*(s)$ para dos valores de α ($\gamma = 1$).

En la figura 22 mostramos sólo 2 valores de α que representan estos dos comportamientos opuestos para ver lo que decimos con mayor claridad.

Ahora, vamos a probar a repetir el experimento con $\gamma = 0,9$.

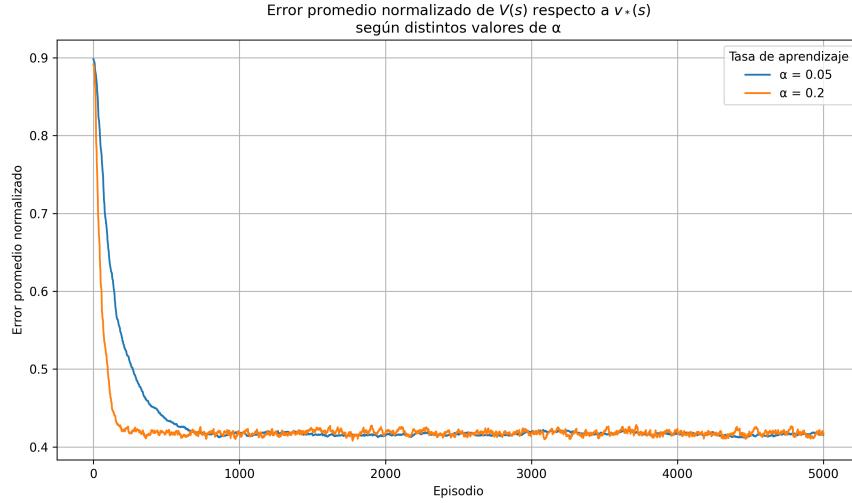


Figura 23: Evolución del error de $V(s)$ respecto a $v_*(s)$ para dos valores de α ($\gamma = 0.9$).

Como nos ocurría en estimación directa, reducir γ quita ruido y ayuda a estimar $v_*(s)$ de forma precisa más temprano. Recordemos que γ , en la regla de actualización pondera aquello del target que no se ha simulado aún en el episodio, así que dar menos pesos a esas estimaciones parece restar ruido al aprendizaje. No obstante, una γ menor puede homogeneizar el valor de los estados, y hay que controlar que eso no llegue al punto de confundir a la política.

Quedémonos con $\alpha = 0.05$ y veamos cuál es el retorno promedio por episodio para los distintos valores de γ que hemos probado.

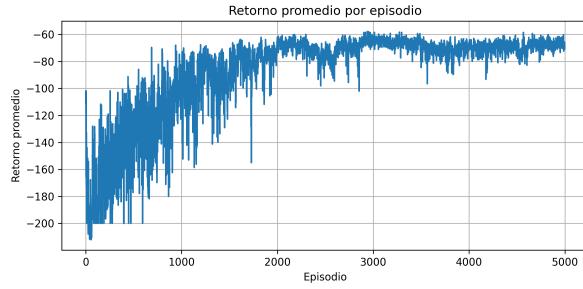


Figura 24: Retorno promedio por episodio de Q-Learning ($\gamma = 1$).

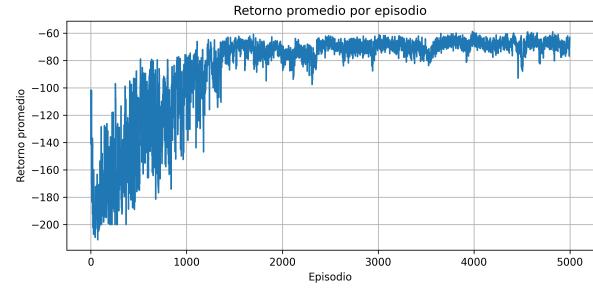


Figura 25: Retorno promedio por episodio de Q-Learning ($\gamma = 0.9$).

Viendo las figuras 24 y 25, podemos decir que alcanzamos el óptimo de retorno promedio prácticamente a la par (según la ejecución puede variar por la estocasticidad). Por lo tanto, pese a que con $\gamma = 1$ la convergencia a $v_*(s)$ es más tardía, la estimación de $V(s)$ es suficiente para que el agente aprenda la política adecuada, ya que como decimos, el retorno promedio óptimo se alcanza a la par con cada γ . Que con $\gamma = 0.9$ minimicemos el error antes indica

lo que decíamos antes: el valor de los estados es menos heterogéneo y por ello estimarlos es más rápido. Esto último podemos verlo en las figuras 26 e 27, que muestran el valor de cada estado tras 5000 iteraciones de QLearning para cada valor de γ .



Figura 26: Valor de cada estado tras 5000 episodios de Q-Learning ($\gamma = 1$).

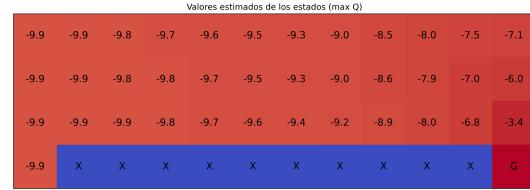


Figura 27: Valor de cada estado tras 5000 episodios de Q-Learning ($\gamma = 0,9$).

También es interesante ver el valor desglosado por cada estado-acción. Las figuras 28 e 29 muestran esto. Para cada estado, tenemos 4 subcasillas que representan los valores de las acciones de ir hacia arriba, derecha, abajo, izquierda, en orden horario empezando por la subcasilla de arriba a la izquierda.



Figura 28: Valor de cada estado-acción tras 5000 episodios de Q-Learning ($\gamma = 1$).



Figura 29: Valor de cada estado-acción tras 5000 episodios de Q-Learning ($\gamma = 0,9$).

De esta manera podemos observar el valor de las acciones, el valor de los estados (máximo valor de acción) y la política (acción que maximiza el valor) a la vez. Lo primero que salta a la vista es cómo el agente tiene muy claro qué acciones no debe tomar en las casillas colindantes al acantilado, y son todas aquellas que dan probabilidad a precipitarse por éste. Por otro lado, se confirma cómo una menor γ hace el valor de los estados más homogéneos, pero suficientemente diferentes como para entender que debe evitar el acantilado y moverse hacia el objetivo.

Vamos a hacer un último experimento. Antes hemos visto que según el step size α tenemos un tradeoff entre llegar rápido a $v_*(s)$ y hacerlo de forma precisa. Una práctica común en RL es implementar un ***decay*** (decaimiento) en la tasa de exploración ϵ que hasta ahora la hemos fijado en 0.1 y la tasa de aprendizaje α o step size. El objetivo es lograr el mejor comportamiento de cada valor, es decir, en el caso de ϵ , hacer más exploración al principio para lograr una mejor cobertura del espacio de estados y después pasar a explorar menos y explotar el conocimiento adquirido. En el caso de α , al principio buscaremos grandes saltos

en el aprendizaje, y al final ajustes más suaves que reduzcan la varianza y estabilicen los valores.

Para probar esto, veremos de nuevo la evolución de $V(s)$ respecto a $v_*(s)$ sin decay, con decay de ϵ , con decay de α y con decay de ambos. En todos los casos usaremos $\gamma = 1$. Falta añadir que cuando no usamos decay, fijamos $\alpha = 0,05$ y $\epsilon = 0,1$, pero cuando usamos decay en un parámetro, fijamos su valor en 0.2.

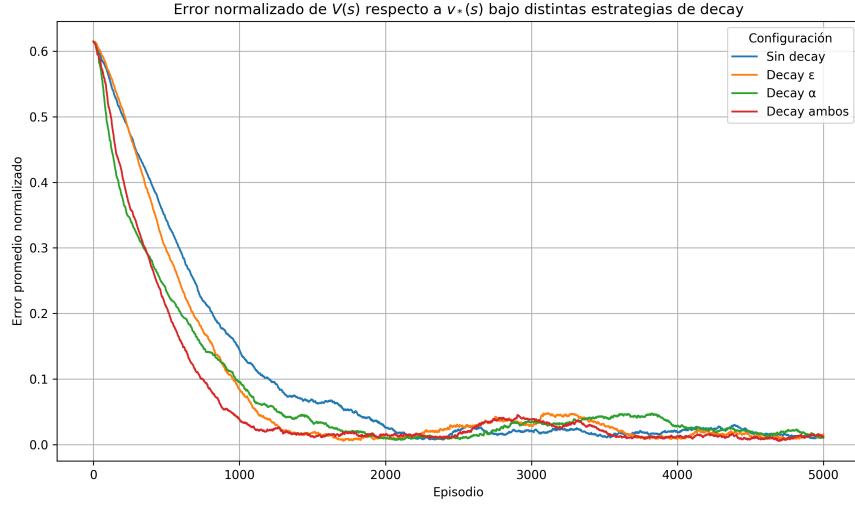


Figura 30: Evolución del error de $V(s)$ respecto a $v_*(s)$ para diferentes estrategias de decay ($\gamma = 1$).

La figura 30 confirma lo que decíamos. La línea roja muestra la evolución del algoritmo usando decay en ambos parámetros y, aunque la diferencia no es mucha, es la que antes llega a la zona de convergencia y además se estabiliza sin problemas.

Por último, repetimos el experimento con $\gamma = 0,95$, ya que con $\gamma < 1$ hemos visto que podemos acercarnos a $v_*(s)$ con menos episodios (aunque ello no implique directamente obtener antes la política con mejores retornos).

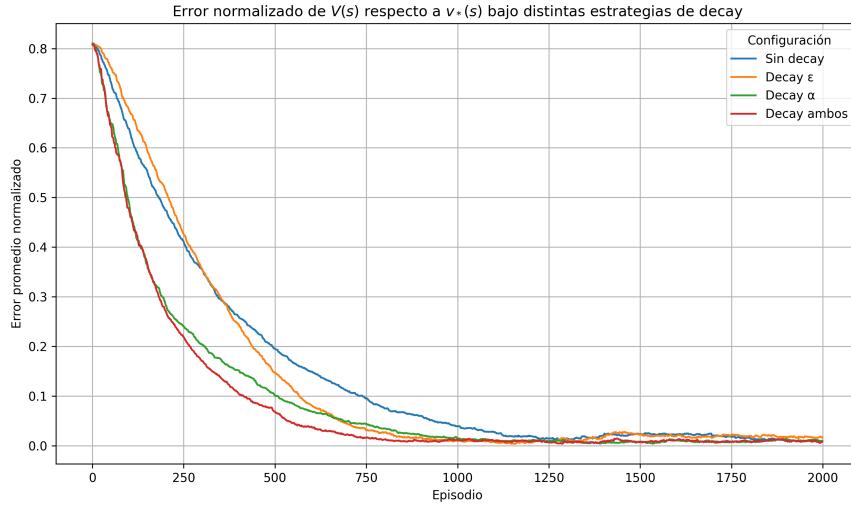


Figura 31: Evolución del error de $V(s)$ respecto a $v_*(s)$ para diferentes estrategias de decay ($\gamma = 0,95$).

El resultado es que gracias al decay de α y ϵ y la ligera reducción de γ logramos aproximar $V(s) \approx v_*(s)$ con alrededor de 1000 episodios.

Resumen: Q-Learning aprende online y converge a q_* estimando los valores de acción. Un valor de α pequeño garantiza precisión, mientras que uno mayor acelera la convergencia pero añade ruido. Reducir γ suaviza el aprendizaje, aunque puede hacer menos distinguibles los valores de los estados. Aplicar *decay* tanto a ϵ como a α permite explotar bien al inicio y estabilizar el aprendizaje al final, lo cual mejora la eficiencia (menos episodios) del algoritmo en este caso.

6. Conclusiones

En esta última sección explicamos las conclusiones generales extraídas tras realizar esta práctica.

En primer lugar, lo más positivo de esta práctica ha sido el conocimiento adquirido sobre el aprendizaje por refuerzo. Hemos entendido el funcionamiento de las 3 técnicas utilizadas, además de los otros algoritmos que están estrechamente relacionados como SARSA, TD(n)... La simplicidad del entorno nos ha permitido observar con mucha claridad cómo los fundamentos teóricos se cumplían en la práctica. Esto es sólo una primera toma de contacto, pero una base sólida para seguir profundizando.

Siguiendo con la simplicidad del entorno, cabe destacar que gracias a ello hemos podido llevar a cabo una gran variedad de experimentos, y cada uno repetidas veces. Los tiempos de aprendizaje, tanto en iteración de valor como en Q-Learning han sido siempre cuestión de

escasos segundos < 10 s. Los únicos experimentos con un coste computacional más significativo son aquellos en los que calculamos promedios de retorno en cada iteración/episodio de aprendizaje, ya que debido a la estocasticidad del entorno llevamos a cabo 200 simulaciones por cada iteración/episodio. Por ello, no hemos considerado el tiempo como una métrica de estudio en los experimentos. Sin embargo, reducir el número de episodios necesarios para un buen aprendizaje gracias a las configuraciones de ϵ , α y γ es algo que, pese a que no se traduzca en diferencias temporales significativas en este entorno, es muy interesante de cara a problemas más complejos.

También entendemos que no hay un valor ideal para cada parámetro, es algo que depende del problema, pero la simplicidad del entorno hace que no sea difícil dar con una configuración que dé buenos resultados.

Hay otras conclusiones que son más complejas de exemplificar visualmente en un informe como este. Nos referimos a la observación de la evolución política mientras se hace el aprendizaje, que podemos ver con la ventana gráfica que ofrece el entorno. Por ejemplo, en el caso de Q-Learning, el agente durante los primeros episodios se mueve de forma aleatoria hasta que, como cae muchas veces por el acantilado, aprende a alejarse cuando tiene la probabilidad de caer. Después, cuando eventualmente topa con el goal en algún episodio ya da valor a las acciones que lo llevan inmediatamente a éste, desde algún estado colindante (porque este algoritmo mantiene la esencia del aprendizaje por diferencias temporales, y en este caso la diferencia temporal es de 1).

En definitiva, esta práctica nos ha permitido entender los fundamentos del aprendizaje por refuerzo en un entorno simple pero expresivo. Más allá del ejercicio académico, este conocimiento conecta con aplicaciones reales: empresas como NVIDIA o Siemens ya emplean RL para optimizar desde circuitos hasta turbinas, y DeepMind lo ha aplicado incluso al control de reactores de fusión. Además, modelos como DeepSeek utilizan Reinforcement Learning from Human Feedback (RLHF), un enfoque que permite ajustar los modelos de lenguaje a preferencias humanas. Todo ello demuestra que, aunque en 2018 se señalara con razón que “Deep Reinforcement Learning Doesn’t Work Yet”, hoy el RL es una herramienta fundamental en algunos de los avances más relevantes de la IA.

Referencias

[Sutton and Barto, 2018] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, Massachusetts, second edition.