UNIVERSIDAD DE MURCIA

FACULTAD DE INGENIERÍA INFORMÁTICA

MASTER EN INTELIGENCIA ARTIFICIAL

UNIVERSIDAD DE MURCIA

Extensiones de Machine Learning

Práctica 2

Profesor:

Luis Daniel Hernández Molinero

Estudiantes:

Antonio Marín Ortega (antonio.marino@um.es)

Darío Escudero de Paco (dario.escuderop@um.es)

Francisco Javier Pérez Pujalte (fj.perezpujalte@um.es)

Índice

1. Introducción

Los algoritmos tabulares de aprendizaje por refuerzo permiten resolver problemas modelados como **Procesos de Decisión de Markov** (MDP) de tamaño relativamente pequeño. En la práctica anterior analizamos el caso simplificado de un bandido k-brazos, equivalente a un MDP trivial de un solo estado. Sin embargo, muchos problemas de decisión se desarrollan en múltiples estados interconectados: juegos, navegación de robots, control de vehículos, etc. En tales casos, el agente debe aprender no solo qué acción es mejor en general, sino qué acción tomar *en cada estado* para maximizar recompensas acumuladas futuras. Esto incrementa notablemente la complejidad, pues ahora el agente debe equilibrar exploración-explotación en cada estado y estimar valores de estados o de estado-acción.

El aprendizaje por refuerzo ofrece varias estrategias para este fin. Entre los métodos **tabulares** (que mantienen una tabla de valores para cada estado o par estado-acción), destacan:

- Algoritmos basados en **Monte Carlo** (MC), que estiman valores a partir de promediar recompensas obtenidas en episodios completos.
- Algoritmos **Temporal-Difference** (TD), como **SARSA** y **Q-Learning**, que actualizan estimaciones en cada paso de tiempo usando aproximaciones bootstrap de las ecuaciones de Bellman.

Estos métodos pueden ser on-policy (aprenden la misma política que utilizan para actuar) u off-policy (aprenden la política óptima mientras actúan posiblemente de forma distinta). Monte Carlo tiene versiones on-policy (por ejemplo, First-Visit MC control con ε-greedy) y off-policy (como MC con *importance sampling* para evaluar una política objetivo mientras se sigue otra comportamiento). SARSA es un método on-policy TD, mientras que Q-Learning es off-policy por definición (aprende la política óptima Q^* mientras posiblemente sigue una política exploratoria distinta).

Un desafío surge cuando el espacio de estados es muy grande o continuo: las tablas de valores se vuelven inviables. Para afrontar entornos complejos, se emplean **funciones de aproximación**. Estas aproximaciones, ya sean lineales (combinaciones de *features*) o no lineales (redes neuronales), permiten generalizar valores a estados no vistos pero requieren técnicas especializadas para aprender de manera estable. En esta práctica exploramos dos enfoques:

- **Aproximación lineal semigradiente:** utilizaremos SARSA con distintas bases de funciones (Fourier, radial RBF, y codificación de baldosas *tile coding*) para resolver un clásico entorno continuo.
- **Deep Q-Learning (DQN):** utilizaremos una red neuronal profunda para aproximar la función $Q(s, a)$, junto con técnicas como *experience replay* y una red objetivo fija, como propusieron , para estabilizar el entrenamiento [1].

Los entornos de prueba escogidos son:

- `FrozenLake` (versión 4x4 y 8x8) – un gridworld donde el agente debe aprender a caminar sobre un lago helado sin caer en agujeros, llegando a una meta. Es un problema discreto con estocasticidad (si el terreno es resbaladizo) y episodios que terminan en éxito o fallo. Su espacio de estados es pequeño (16 o 64 estados) y acciones discretas (4 direcciones). Esto permite aplicar métodos tabulares y comparar su eficacia.

- `MountainCar` – un problema continuo clásico: un coche en un valle debe tomar impulso para subir una colina. El espacio de estados es continuo bidimensional (posición, velocidad) y las acciones discretas (acelerar izquierda, derecha o no acelerar). Es un entorno donde métodos tabulares fracasan (infinitos estados) y se necesita aproximación de funciones para generalizar entre estados similares. Además, es un problema episódico donde cada episodio termina al salir de la colina (éxito) o tras un límite de pasos (fracaso).

El resto del documento se organiza así: en la sección de **Desarrollo** se revisa el marco teórico de estos algoritmos (retomando ecuaciones de Bellman, técnicas on-policy vs off-policy, etc., y describiendo los enfoques de aproximación mencionados). Luego, en **Algoritmos** se dan pseudocódigos simplificados de MC, SARSA, Q-Learning, SARSA semigradiente y DQN, junto a comentarios de implementación (por ejemplo, manejo de exploración ε-greedy, decaimiento de ε, estructuras de datos utilizadas, etc.). En **Evaluación** se detallan los experimentos: primero los entornos tabulares (FrozenLake) comparando MC, SARSA y Q-Learning, y luego los entornos continuos (MountainCar) comparando SARSA con distintas bases y DQN. Se incluyen gráficos de aprendizaje por episodios: tasa de éxito, recompensas obtenidas y longitudes de episodio a través del tiempo de entrenamiento. Por último, se presentan las conclusiones destacando qué algoritmos tuvieron mejor desempeño en cada caso y por qué, así como reflexiones sobre la estabilidad de los métodos con aproximación vs tabulares.

2. Desarrollo: Algoritmos de RL en entornos MDP

2.1. Formulación del problema y notación

En un proceso de decisión de Markov, tenemos un conjunto de estados $\mathcal{S}$ y acciones $\mathcal{A}$. En cada paso t, el agente observa un estado $S_t \in \mathcal{S}$, ejecuta una acción $A_t \in \mathcal{A}$, y el entorno provee una recompensa R_{t+1} y un nuevo estado S_{t+1}. El objetivo es aprender una **política** $\pi(a|s)$ que maximice la **recompensa acumulada esperada** a largo plazo (por ejemplo, retorno descontado $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ con factor de descuento $0 \leq \gamma < 1$).

Las estrategias de solución pasan generalmente por estimar la **función de valor** $V^\pi(s)$ de estados, o la **función de acción-valor** $Q^\pi(s,a)$, que satisfacen las ecuaciones de Bellman. Para la política óptima π^* existe $Q^*(s,a) = \max_\pi Q^\pi(s,a)$ que cumple $Q^*(s,a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} Q^*(S_{t+1}, a') \mid S_t = s, A_t = a]$. Muchos algoritmos de RL intentan aproximar Q^* sin modelo del entorno (i.e., sin conocer las probabilidades de transición), a través de interacciones simuladas.

2.2. Métodos Monte Carlo on-policy y off-policy

Los métodos de Monte Carlo aprenden por *trial-and-error* episodódico, promediando resultados. En control Monte Carlo on-policy (por ejemplo, **Every-Visit MC**):

- Inicializamos $Q(s,a)$ arbitrariamente (ej. cero) y una política exploratoria, típicamente ε-greedy respecto a Q.

- Para cada episodio completo obtenido siguiendo la política, actualizamos para cada par (s,a) visitado la estimación $Q(s,a)$ hacia el promedio de los retornos G observados a partir de esa visita. Es decir, $Q(s,a) \leftarrow Q(s,a) + \alpha(G_{\text{primera visita}(s,a)} - Q(s,a))$ con α puede ser $\frac{1}{N(s,a)}$ o un valor pequeño constante.

- Luego se mejora la política π haciendo ε-greedy con respecto al nuevo Q (esto asegura convergencia al óptimo con probabilidad 1 en el límite, según la aproximación generalizada de policy iteration).

El pseudocódigo básico de MC on-policy (todas las visitas) podría escribirse, para N episodios:

for *episodio = 1* **to** N **do**
  Generar un episodio siguiendo la política π (exploratoria) y almacenar las transiciones $(S_0, A_0, R_1, S_1, \ldots, S_T)$ hasta terminar ;
  for *cada par estado-acción* (s, a) *visitado en el episodio* **do**
    Calcular G = retorno desde la primera vez que se visitó (s, a) ;
    Actualizar promedio: $Q(s, a) \leftarrow Q(s, a) + \alpha(G - Q(s, a))$;
  end
  Actualizar π en cada estado visitado: $\pi(s) \leftarrow \arg\max_a Q(s, a)$ (con probabilidad $1 - \varepsilon$, caso ε-greedy) ;
end

En nuestro caso práctico, aplicamos MC on-policy al entorno FrozenLake. Elegimos ε-greedy con ε decaído (por ejemplo, iniciando en 0.4 y disminuyendo a lo largo de episodios). MC eventualmente converge a una política óptima (bajo ciertas condiciones), pero su velocidad puede ser lenta: requiere muchos episodios porque solo aprende de las evaluaciones completas. Además, en FrozenLake 8x8 (más grande), MC puede tardar en recibir recompensas no nulas (llegar a la meta) lo suficiente como para actualizar valores útiles.

Monte Carlo Off-Policy se basa en el principio de corrección de discrepancias entre distribuciones de comportamiento y objetivo mediante **Importance Sampling (IS)**. Cuando los datos de entrenamiento se generan siguiendo una política de comportamiento μ, pero el objetivo es estimar los valores de una política objetivo π, se emplean pesos de importancia acumulados de la forma:

$$W_{1:t} = \prod_{k=1}^{t} \frac{\pi(A_k|S_k)}{\mu(A_k|S_k)}$$

Este producto pondera el retorno observado G_t para corregir las diferencias entre ambas políticas.

Existen dos variantes principales de IS en Monte Carlo Off-Policy:

- **Ordinary Importance Sampling (OIS):** pondera cada retorno directamente por el peso $W_{1:t}$. Es insesgado, pero con varianza potencialmente muy alta.
- **Weighted Importance Sampling (WIS):** normaliza los pesos acumulados entre episodios, reduciendo la varianza a costa de introducir cierto sesgo.

En nuestro caso, implementamos una versión simplificada de **Monte Carlo Off-Policy Control con Weighted Importance Sampling**. Concretamente:

- La política de comportamiento μ fue moderadamente exploratoria: ε-greedy con $\varepsilon \approx 0{,}3$.

- La política objetivo π fue la greedy pura sobre los valores estimados $Q(s, a)$:

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

- Tras generar cada episodio, el retorno se procesó hacia atrás aplicando el esquema de corte habitual: si en algún paso $A_k \neq \arg\max_a Q(S_k, a)$, el peso acumulado $W_{1:t}$ se detiene (truncamiento), al no coincidir ya con la política objetivo.

La actualización incremental de Q sigue la forma:

$$Q(s, a) \leftarrow Q(s, a) + \frac{W_{1:t}}{C(s, a)} (G_t - Q(s, a))$$

donde $C(s, a)$ es el acumulador de pesos observado para el par (s, a).

Observaciones prácticas: Debido al elevado nivel de exploración de μ, muchas de las trayectorias generadas en entornos como *FrozenLake 8x8* raramente alcanzan el estado objetivo, provocando:

- Muchos episodios con retornos bajos.
- Pesos de importancia $W_{1:t}$ altamente variables (explosivos o casi nulos).
- Actualizaciones de Q ruidosas y de pequeño impacto.

Como consecuencia, la convergencia fue lenta y con alta varianza, fenómeno característico de Monte Carlo Off-Policy con Importance Sampling. Este comportamiento ilustra por qué, en tareas de control, los métodos TD Off-Policy como **Q-Learning** son habitualmente preferidos debido a su mayor estabilidad y eficiencia.

En resumen, Monte Carlo provee estimaciones consistentes y conceptualmente simples, pero su **limitación** es la necesidad de completar episodios para cada actualización significativa. En entornos con probabilidad alta de fracaso (ej. FrozenLake resbaladizo, o 8x8 con muchos agujeros), MC puede requerir muchísimos episodios para obtener suficientes éxitos que guíen el aprendizaje.

2.3. Métodos TD: SARSA y Q-Learning

Los algoritmos TD actualizan valores a partir de estimaciones actuales, permitiendo aprender *durante* el episodio, paso a paso.

SARSA (Estado-Acción-Recompensa-Estado-Acción) es un método on-policy que actualiza $Q(S_t, A_t)$ hacia la dirección de la recompensa obtenida más lo que *la propia política* estima que obtendrá desde el nuevo estado:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma\, Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \Big].$$

Aquí A_{t+1} se elige según la política actual (por ejemplo ε-greedy). Este algoritmo garantiza convergencia a Q^π óptimo si ε se reduce apropiadamente y α disminuye en el tiempo (o se satisface cierta condición de Robbins-Monro). SARSA es *seguro* en el sentido de que aprende la calidad de la política que ejecuta; por tanto, si la política es ε-greedy, tenderá a valores para la política

ε-greedy óptima (no necesariamente la óptima determinista, a menos que $\varepsilon \to 0$ al final).

Q-Learning es off-policy: actualiza $Q(S_t, A_t)$ hacia la estimación suponiendo que en el siguiente estado se seguirá la mejor acción *posible* (no necesariamente la realmente seguida):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big].$$

En la implementación, uno puede seguir una política exploratoria (p.ej. ε-greedy) para obtener datos, pero las actualizaciones siempre consideran la acción *greedy* para estimar el futuro. Esto permite a Q-Learning converger a Q^* incluso bajo exploración persistente (de hecho, Q-Learning es demostrablemente convergente bajo ciertas condiciones de visita y decrecimiento de α). Q-Learning tiende a ser más *arriesgado* que SARSA en entornos estocásticos: al aprender off-policy, puede sobrevalorar estados peligrosos que la política ε-greedy realmente evitaría. En FrozenLake, por ejemplo, Q-Learning puede sobreestimar la utilidad de caminar cerca de agujeros si las transiciones probabilísticas no se muestrean suficientemente; SARSA, en cambio, aprendería el valor real de la política que realmente "patina" evitando riesgos.

2.4. Métodos con aproximación de funciones

Para MountainCar, y en general entornos continuos, no es factible llenar una tabla $Q(s, a)$ para cada estado (infinitos). Se introduce entonces una representación paramétrica $Q(s, a; \boldsymbol{\theta})$. En métodos lineales, $\boldsymbol{\theta}$ es un vector de pesos y $Q(s, a) \approx \boldsymbol{\theta}^\top \mathbf{x}(s, a)$, donde $\mathbf{x}(s, a)$ es un vector de características derivado del estado (y a veces de la acción).

Los algoritmos TD deben adaptarse a este contexto. En lugar de actualizar una entrada de tabla, se ajustan los pesos en dirección del gradiente del error TD. Por ejemplo, el algoritmo **SARSA semigradiente** actualiza:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\Big(r + \gamma Q(s', a'; \boldsymbol{\theta}) - Q(s, a; \boldsymbol{\theta})\Big)\nabla_{\boldsymbol{\theta}} Q(s, a; \boldsymbol{\theta}).$$

Como $Q(s, a; \theta) = \theta^\top x(s, a)$, su gradiente es $x(s, a)$ mismo. Entonces la actualización de θ se convierte en:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\delta\, \mathbf{x}(s, a),$$

donde $\delta = r + \gamma Q(s', a'; \theta) - Q(s, a; \theta)$ es el error TD. Esta es una forma de **Gradient Descent TD** (también conocida como *semi-gradient* porque usamos la propia aproximación para el objetivo).

La clave del éxito está en seleccionar representaciones adecuadas de características $\boldsymbol{\phi}(s)$. En nuestros experimentos sobre *MountainCar*, exploramos tres esquemas de aproximación:

- **Tile Coding (TC):** el espacio continuo de estados se discretiza mediante múltiples rejillas superpuestas (tilings), cada una ligeramente desplazada respecto a las demás. Cada tiling produce una partición del espacio (baldosas o *tiles*), donde exactamente un tile está activo por tiling para cada estado observado. La representación resultante es un vector binario muy disperso (sparse), con un único 1 por tiling y el resto ceros. Esta superposición permite aumentar la resolución efectiva y lograr una mejor generalización local. En nuestra configuración, utilizamos 8 tilings de 8×8 tiles cada uno, lo que da lugar a un total de $8 \times 64 = 512$ características.

- **Funciones de Fourier:** se definen features $x_{i,j}(s) = \cos(\pi(is_1 + js_2))$ hasta cierto orden para estado (s_1, s_2). Esto genera una base de funciones senoidales de frecuencia creciente que pueden aproximar la función de valor con cierta suavidad. Elegimos orden 1 para cada dimensión (lo que da $(1+1)^2 = 4$ features por acción).
- **Funciones de Base Radial (RBF):** colocamos centros Gaussianos en el espacio de estado (en nuestro caso, una retícula de 9x9 centros) y definimos features $x_k(s) = \exp(-\frac{||s-c_k||^2}{2\sigma^2})$ para cada centro c_k. Las RBF capturan localmente la vecindad del estado. Configuramos σ de modo que las Gaussians tengan radio de influencia que cubra aproximadamente hasta vecinos más cercanos.

Implementamos agentes SARSA con cada tipo de feature. Todos comparten la misma estructura pseudocódigo (algoritmo SARSA) pero difieren en cómo calculan Q:

Inicializar pesos $\theta \leftarrow 0$ (dimensión = num. de features × num. de acciones) ;
for *episodio = 1* **to** N **do**
  Obtener estado inicial s ;
  Seleccionar acción a según política ε-greedy usando $Q(s, \cdot; \theta)$;
  for *cada paso en el episodio* **do**
    Tomar acción a, observar recompensa r y estado siguiente s' ;
    Seleccionar a' desde s' por ε-greedy(Q) ;
    Computar $\delta = r + \gamma Q(s', a'; \theta) - Q(s, a; \theta)$;
    Actualizar pesos: $\theta \leftarrow \theta + \alpha\, \delta\, x(s, a)$; `//` $x(s, a)$ `= vector de features de` (s, a)
    $s \leftarrow s'$,;
    $a \leftarrow a'$;
  end
end

Ajustar α fue crítico: con tile coding usamos $\alpha \approx 0{,}01$, con Fourier $\alpha \approx 0{,}05$, y con RBF $\alpha \approx 0{,}01$. Todas con descuento $\gamma = 1$.

Por otro lado, el **Deep Q-Network (DQN)** utiliza una red neuronal para aproximar $Q(s, a; \theta)$. En nuestro caso, empleamos una red sencilla de 2 capas ocultas (por ejemplo, 2 capas de 64 neuronas cada una, con activaciones ReLU). DQN introduce dos técnicas clave:

- **Experience Replay:** en lugar de aprender en línea en cada paso, se almacenan las transiciones (s, a, r, s') en un buffer y se entrenan minibatches muestreados aleatoriamente de dicho buffer. Esto rompe la correlación temporal de los datos y aprovecha mejor cada experiencia.
- **Target Network:** se usa una copia de la red $\hat{Q}(s, a; \theta^-)$ congelada por intervalos de tiempo como objetivo para las actualizaciones. Esto significa que el cálculo $y = r + \gamma \max_{a'} \hat{Q}(s', a'; \theta^-)$ es más estable porque θ^- cambia con menos frecuencia (por ejemplo, se actualiza a θ cada 1000 iteraciones).

Nuestro DQN (basado en Mnih [1]) se entrena así:
Implementamos el DQN en `PyTorch` (ver notebook) con parámetros: buffer de 10000 experiencias, batch size 64, $\gamma = 0{,}99$, $C = 200$ (target update frequency), α de optimizador Adam = 0,001. ε se realizó decaimiento de 1.0 hasta 0.01.

Inicializar pesos θ aleatoriamente, $\theta^- = \theta$;
Inicializar buffer de replay vacío ;
for *episodio = 1* **to** N **do**
  for *cada paso en el episodio* **do**
    Seleccionar a según política ε-greedy (en valores $Q(s, \cdot; \theta)$) ;
    Ejecutar a, observar r, s' ;
    Almacenar $(s, a, r, s', \text{terminal})$ en buffer ;
    Sample aleatorio de minibatch de transiciones (s_j, a_j, r_j, s'_j) del buffer ;
    for *cada muestra en minibatch* **do**
      $y_j = r_j + \gamma \max_{a'} Q(s'_j, a'; \theta^-)$ (ó r_j si s'_j es terminal) ;
    end
    Minimizar pérdida $L = \frac{1}{m} \sum_j [y_j - Q(s_j, a_j; \theta)]^2$ actualizando θ vía retropropagación ;
    Cada C pasos, sincronizar $\theta^- \leftarrow \theta$;
    $s \leftarrow s'$;
  end
end

3. Evaluación y Resultados Experimentales

Repositorio Github [2].

3.1. Entorno `FrozenLake` (tabular)

Para comparar MC, SARSA y Q-Learning empleamos **FrozenLake** en dos tamaños: 4×4 y 8×8, sin resbalamiento (`is_slippery=False`)

Métricas

- *Recompensa media por episodio*: equivale a la **tasa de éxito** (1 si se alcanza la meta, 0 en caso contrario).
- *Longitud media del episodio*: número de pasos hasta terminar (caer o llegar a la meta). Este valor puede descender porque el agente encuentra rutas más cortas *o* porque cae antes; interpretamos siempre junto a la tasa de éxito.
- Mostramos al final la política $\pi(s) = \arg\max_a Q(s, a)$.

FrozenLake 4×4 (máx. 100 pasos). :

- *MC on-policy*: necesita $\sim$75.000 episodios para superar el 90 % de éxito; la curva es ruidosa por la exploración por episodios completos.
- *MC off-policy*: necesita $\sim$200.000 episodios para superar el 60 % de éxito, no siendo bueno para este problema.
- *SARSA*: sobrepasa el 90 % tras $\sim$50.000 episodios.
- *Q-Learning*: sobrepasa el 90 % tras $\sim$50.000 episodios al igual que SARSA.

FrozenLake 8×8 (máx. 100 pasos).

- *MC On-policy*: necesita ∼100 000 episodios para superar el 95 % de éxito.
- *MC off-policy*: en este caso Monte Carlo Off-Policy no llega aprender absolutamente nada ya que se queda cerca del 0 % con una longitud máxima por episodio hasta el 200 000.
- *SARSA*: alcanza 95 % en ∼75 000 episodios.
- *Q-Learning*: al igual que sarsa, alcanza el 95 % en ∼75.000 episodios.

Conclusión. Los métodos TD superan a MC en eficiencia; Q-Learning converge antes, mientras que SARSA ofrece mayor estabilidad en entornos estocásticos. Con exploración decreciente ambos alcanzan la política óptima determinista; la política final aprendida coincide (longitud 6 pasos en 4×4 y 14 en 8×8), validando los valores de Q.

3.2. Entorno MountainCar (aproximación continua)

Para MountainCar, evaluamos:

- SARSA semigradiente con **Tile Coding** (8 tilings, 8x8 cada una, total 512 features por acción).
- SARSA semigradiente con **RBF** (81 Gaussians, 9x9 grid as centers).
- SARSA semigradiente con **Fourier** (orden 1 en cada dimensión, 4 features).
- **DQN** red neuronal profunda ($input\_dim * 64|64 * 128|128 * 128|128 * output\_dim$).

Todas las variantes lograron eventualmente resolver el problema (coche sale de la montaña). Sin embargo, hubo diferencias en velocidad y estabilidad:

- **Sarsa semigradiente con bases de Fourier.** Este agente presentó una convergencia lenta debido a su alta exploración inicial. Durante aproximadamente los primeros ∼1800 episodios de entrenamiento no consiguió ningún episodio exitoso (la recompensa se mantuvo cerca de -200 en promedio, indicando fallos constantes). A partir de entonces, al empezar a reducirse ϵ, el rendimiento mejoró gradualmente. Alcanzó una política que siempre logra la meta tras unos 10.000 episodios de entrenamiento. La política final aprendida muestra cierta variabilidad en su desempeño: la duración media de los episodios (en ejecución *greedy*) es de unos 139 pasos, con una desviación estándar de ±30 pasos. Incluso se observaron casos puntuales cercanos al máximo de 200 pasos (ej. hasta 199 pasos) antes de alcanzar la cima, evidencia de que la solución obtenida no es totalmente óptima ni consistente.
- **Sarsa semigradiente con funciones RBF.** La convergencia del agente con bases radiales fue también relativamente lenta. Comenzó a obtener sus primeros éxitos tras unos ∼2000 episodios, avanzando de forma más suave que el método Fourier. Con suficiente entrenamiento (10.000 episodios) alcanzó finalmente un nivel de desempeño con 100 % de éxito en los intentos. Su política final resultante tiene una duración media de 150 pasos por episodio (±16), ligeramente peor que los agentes de Fourier y Tile Coding en términos de eficiencia. No obstante, presentó menor variabilidad que Fourier: prácticamente no hubo episodios cercanos al límite de 200 pasos (el peor caso observado fue de 191 pasos). Esto indica que, si bien

el agente RBF aprende con lentitud, termina produciendo una política algo más consistente (aunque algo menos eficiente) comparada con Fourier.

- **Sarsa semigradiente con Tile Coding.** El agente basado en Tile Coding demostró un aprendizaje mucho más rápido. Obtuvo sus primeros episodios exitosos dentro de las primeras decenas de episodios (en torno al episodio 50–100 ya había alcanzado la cima en algún intento). Para aproximadamente ∼500 episodios de entrenamiento, ya resolvía el entorno de forma consistente en la mayoría de intentos. Este buen desempeño temprano se debe en parte a que utilizó una exploración moderada desde el inicio ($\epsilon = 0{,}1$ fijo), permitiendo explotar lo aprendido más pronto. La política final aprendida es similar en eficacia a la del agente RBF: alrededor de 146 pasos por episodio en promedio (±16). También aquí la variabilidad fue moderada y todos los intentos evaluados alcanzaron la meta (100 % éxito, con duraciones típicamente entre 130 y 150 pasos, un mejor caso de 88 pasos y peor de 189). Cabe destacar que el agente con Tile Coding había prácticamente convergido mucho antes que los agentes Fourier y RBF; sin embargo, debido a la exploración ϵ-greedy persistente durante el entrenamiento, su recompensa media acumulada se estabilizó en torno a un valor algo más bajo (≈ -175) de lo que logra su política *greedy* final.

- **Deep Q-Learning (DQN).** El agente DQN destacó por lograr el mejor rendimiento tanto en eficiencia como en estabilidad. Comenzó a resolver episodios con éxito tras unos ∼500 episodios, y para alrededor de ∼1000–1500 episodios de entrenamiento ya había aprendido una política de alta calidad. De hecho, alcanzó rápidamente el régimen de 100 % de éxito y continuó mejorando hasta obtener la solución más óptima entre los métodos probados. Su política final asciende la colina en un promedio de 106 pasos por episodio, con una desviación estándar de apenas ±11 pasos. Esto significa que el DQN no solo necesita menos de la mitad de pasos que los demás agentes en promedio, sino que también exhibe la menor variabilidad (comportamiento más consistente). En nuestras evaluaciones, nunca tardó más de 130 pasos en completar el episodio (muy por debajo del tope de 200), cumpliendo holgadamente con el criterio típico de "entorno resuelto" en MountainCar (recompensa media > -110). En términos cualitativos, el DQN aprendió a balancear el carro de forma óptima para tomar el máximo impulso lo antes posible, alcanzando la cima con margen.

En la Tabla siguiente se resumen cuantitativamente las métricas clave de desempeño de cada método, lo que permite comparar su velocidad de aprendizaje y la calidad de las políticas obtenidas:

Método	**Episodios (aprox)**	**Duración media episodio**
Fourier (Sarsa)	∼2000	$139, \pm, 30$ pasos
RBF (Sarsa)	∼2500	$150, \pm, 16$ pasos
Tile Coding (Sarsa)	∼500	$146, \pm, 16$ pasos
Deep Q-Learning (DQN)	∼1000	$106, \pm, 11$ pasos

Tabla 1: Comparativa de rendimiento de los distintos métodos aproximados en `MountainCar-v0`. Se muestra el número de episodios de entrenamiento necesarios (aproximados) para aprender a resolver consistentemente la tarea, junto con la duración media (y desviación) de los episodios alcanzada por la política final de cada agente (evaluada sin exploración). Todos los métodos lograron eventualmente una tasa de éxito del 100 %, pero DQN destaca por requerir menos episodios y menos pasos por episodio para completar la tarea.

4. Conclusiones

A lo largo de esta práctica hemos explorado de manera sistemática distintos algoritmos de aprendizaje por refuerzo, tanto en su versión tabular como con aproximación de funciones, evaluándolos en dos entornos representativos: `FrozenLake` (discreto, pequeño) y `MountainCar` (continuo, complejo). Los resultados experimentales permiten extraer varias conclusiones relevantes:

- **Eficiencia de métodos TD frente a Monte Carlo en entornos tabulares:** En `FrozenLake`, los métodos basados en diferencia temporal (SARSA y Q-Learning) han demostrado una convergencia mucho más rápida y estable que los métodos Monte Carlo, especialmente frente a la variante Off-Policy con *importance sampling*, que mostró gran varianza y dificultad para aprender en problemas con episodios fallidos frecuentes. Esto confirma la ventaja de los métodos TD para tareas de control donde es posible aprender a partir de pasos intermedios, sin necesidad de esperar a finalizar episodios completos.

- **Importancia de la representación de características en entornos continuos:** En `MountainCar`, donde los métodos tabulares son inviables, las distintas técnicas de aproximación funcional han mostrado rendimientos diferenciados. Tile Coding ha ofrecido una excelente capacidad de aprendizaje temprano, incluso con exploración persistente, gracias a su representación densa local; RBF logra políticas algo más estables aunque con aprendizaje más lento; Fourier, pese a su menor dimensionalidad, ha mostrado mayor variabilidad y requiere más episodios para estabilizarse. La correcta elección de características resulta clave para la eficacia de los métodos basados en funciones lineales.

- **Ventajas claras de los métodos Deep RL (DQN):** El uso de redes neuronales profundas ha permitido a DQN superar ampliamente a los métodos clásicos en términos de rapidez, estabilidad y eficiencia de las políticas finales. Gracias a técnicas como *experience replay* y *target network*, DQN es capaz de aprovechar mejor las experiencias y estabilizar el entrenamiento incluso en entornos complejos. En nuestro caso, DQN alcanzó el mejor desempeño tanto en número de episodios necesarios como en calidad de la política aprendida (menos pasos por episodio y menor variabilidad).

- **Limitaciones observadas:** Pese al éxito de DQN, cabe destacar que su implementación requiere mayor complejidad computacional, ajuste de hiperparámetros delicados y un pipeline de entrenamiento más sofisticado, frente a la simplicidad conceptual de los métodos tabulares o lineales. Además, la alta exploración inicial sigue siendo crucial para todos los métodos, para evitar quedar atrapados en políticas subóptimas prematuramente.

En conjunto, estos resultados ilustran de manera práctica el papel que juegan las distintas familias de algoritmos de aprendizaje por refuerzo según la naturaleza del entorno, el tamaño del espacio de estados y las capacidades de representación disponibles. Mientras los métodos tabulares siguen siendo valiosos para problemas discretos pequeños, la generalización mediante aproximación funcional (ya sea lineal o profunda) resulta imprescindible para abordar entornos continuos o de gran dimensionalidad.

Referencias

[1] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". En: *Nature* 518.7540 (2015). DQN Atari paper, págs. 529-533.

[2] Repositorio Github. *k_brazos_AM_DE_JP*. Repositorio en GitHub. 2025. URL: `https://github.com/ermaury/k_brazos_AM_DE_JP`.