Universitat Politècnica de Catalunya

Facultat d'Informàtica de Barcelona

Laboratorio - Práctica 2

Sistemes Intel·ligents Distribuïts

Fuster Palà, Llum
Molina Sedano, Òscar
Orteu Aubach, Júlia
Puerta del Valle, Javier

1 Introducción

El aprendizaje por refuerzo (RL) permite entrenar agentes que, a partir de la interacción con un entorno modelado como un proceso de decisión de Markov (MDP), aprenden a maximizar una señal de recompensa. En este trabajo, el objetivo principal es evaluar y comparar distintos algoritmos de RL —Iteración de Valor, Estimación Directa (Model-Based), Q-Learning y métodos de gradiente de política (REINFORCE y Actor-Critic)— en el entorno CliffWalking-v0[1] configurado en modo "*slippery*" `True`.

Para ello, diseñaremos una serie de experimentos que variarán parámetros clave (número de episodios, factor de descuento, tasa de aprendizaje, coeficiente de exploración y función de recompensa) con el fin de medir su impacto en métricas de rendimiento como la recompensa media, la optimalidad de la política resultante, el tiempo de entrenamiento y la eficiencia computacional. Este enfoque nos permitirá identificar fortalezas y limitaciones de cada algoritmo en un entorno con penalizaciones severas y espacios discretos de estados y acciones.

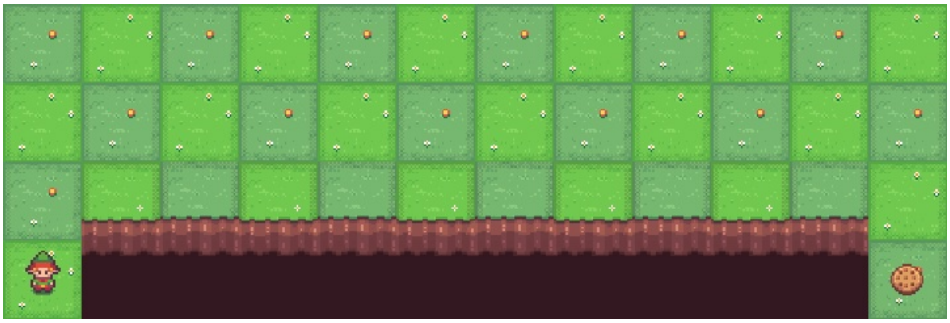


Figure 1: Visualización del entorno CliffWalking-v0 donde el agente debe navegar desde la esquina inferior izquierda hasta la meta (galleta) en la esquina inferior derecha, evitando caer por el acantilado (zona oscura) que abarca 10 estados consecutivos en la parte inferior del grid 4x12. En modo `slippery=True`, el agente tiene probabilidad de deslizarse a estados adyacentes, añadiendo estocasticidad al entorno.

El trabajo se estructura de la siguiente manera: en la Sección 2 se establece el formato de presentación de resultados para cada algoritmo evaluado. Las Secciones 3, 4, 5 y 6 presentan el análisis de Value Iteration, Model-Based, Q-Learning y los métodos de gradiente de política (Actor-Critic y REINFORCE) respectivamente. La Sección 7 ofrece una comparación integral de los resultados obtenidos por todos los algoritmos. Finalmente, la Sección 8 detalla las instrucciones de ejecución y en los Anexos A se adjunta el código fuente de las implementaciones.

[1] https://gymnasium.farama.org/environments/toy_text/cliff_walking/

2 Metodología experimental común

Para garantizar una comparación justa entre algoritmos, hemos aplicado una metodología experimental consistente en todos los casos. A continuación, se detalla el formato de presentación de resultados que se utilizará en las siguientes secciones. Para cada algoritmo, se presentan 4 gráficas con gamma como eje X, mostrando las siguientes métricas:

- (a) **Optimalidad de la solución**: número de acciones diferentes respecto a la política óptima.
- (b) **Reward medio en test**: recompensa obtenida tras 20 ejecuciones post-entrenamiento.
- (c) **Reward de entrenamiento**: recompensa obtenida durante la fase de entrenamiento.
- (d) **Tiempo de ejecución**: tiempo computacional requerido en segundos.

Cada gráfico contiene 6 líneas que corresponden a las siguientes configuraciones:

- (azul) Función de reward base y 500 episodios de entrenamiento.
- (naranja) Función de reward custom y 500 episodios de entrenamiento.
- (verde) Función de reward base y 4000 episodios de entrenamiento.
- (rojo) Función de reward custom y 4000 episodios de entrenamiento.
- (morado) Función de reward base y 10000 episodios de entrenamiento.
- (marrón) Función de reward custom y 10000 episodios de entrenamiento.

3 Value Iteration

3.1 Análisis previo

Value Iteration es un algoritmo **off-policy** que utiliza una versión iterativa de la ecuación de Bellman para estimar los valores óptimos de cada estado.

Las ventajas de este algoritmo se pueden definir como :

- Encuentra la política óptima siempre y cuando se conozcan todas las transiciones entre estados y exista una función de recompensa bien definida.
- Robustez al ruido siempre y cuando se conozcan las probabilidades de transiciones.
- No tiene exploración ni entrenamiento por episodios.
- Inicialización totalmente determinista, ya que se parte de una inicialización de una función de valoración de estados V.

Las desventajas, en cambio, se pueden resumir en:

- Dificultad de convergencia si no hay una función de recompensa adecuada que permita propagar los valores V de los estados.
- Dificultad de elección de método de parada si este no está definido, ya que se puede detener el algoritmo si se llega a un cierto reward óptimo o si la diferencia máxima entre iteraciones es menor que un cierto threshold.
- Poco escalable computacionalmente, ya que es un método iterativo y depende tanto del número de acciones como del número de estados.
- Necesita conocer el entorno de forma completa junto con todas las probabilidades de transición entre estados.

3.2 Implementación

La implementación realizada es la misma que la implementada en los *notebooks* de laboratorio. No se ha incluido ningún contenido adicional más que la aparición de diversos *prints* para verificar la correcta ejecución del mismo. El código completo del algoritmo puede consultarse en el Anexo A.1.

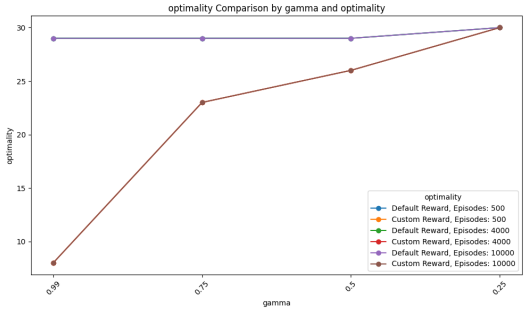
3.3 Experimentación

En cuanto a la experimentación realizada, se han realizado ejecuciones con todas las combinaciones de los valores de los hiperparámetros siguientes:

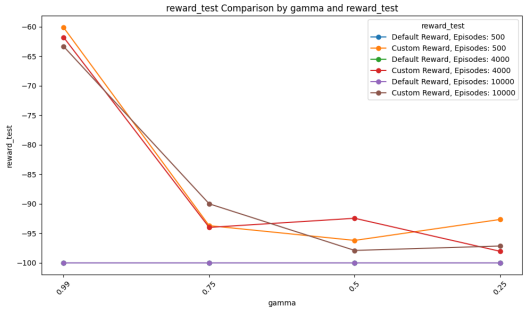
1. Número de episodios: [500, 4000, 10000]
2. Gamma: [0.99, 0.75, 0.5, 0.25]
3. Función de recompensa: [default, custom]
    - **default**: se emplea la función de recompensa por defecto del entorno `CliffWalking-v0`, donde la mayoría de los estados entregan una recompensa de -1, excepto los del acantilado, que penalizan severamente al agente.
    - **custom**: se modifica la función de recompensa para que, al alcanzar el estado terminal (estado 47), la recompensa sea de 0. Esta modificación permite una mejor propagación del valor hacia los estados anteriores y favorece la convergencia hacia una política más eficaz.

3.4 Resultados

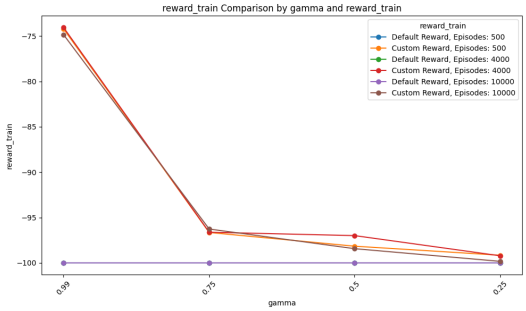
Los resultados se presentan siguiendo el formato estándar establecido en la Sección 2, con 4 gráficas mostrando las métricas de optimalidad, reward de test, reward de entrenamiento y tiempo de ejecución.



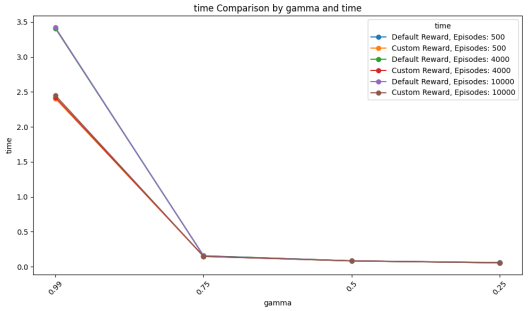
(a) Resultados para valores de gamma según el criterio de optimalidad de la solución.



(b) Resultados para valores de gamma según el criterio de Reward de test.



(c) Resultados para valores de gamma según el criterio de Reward de entrenamiento.



(d) Resultados para valores de gamma según el criterio de tiempo de ejecución.

Figure 2: Resultados de la experimentación del algoritmo Value Iteration.

Como se observa en los gráficos de la Figura 2, existe una diferencia apreciable según la función de reward que se utilice.

Por un lado, con la función de reward predeterminada, el algoritmo no converge hacia ninguna solución debido a que conseguir llegar a la meta se obtiene el mismo reward que una casilla normal i, por tanto, no existe la propagación de los valores de valoración.

Por otro lado, modificando el reward obtenido en el estado final a un 0, creamos esa propagación característica del algoritmo que permite obtener una mejor solución.

Un aspecto importante a destacar y que se denota a simple vista es que para valores de gamma más bajos, se obtienen unas métricas peores, además de una reducción del tiempo de ejecución. Esto es debido a que no se tiene tan en cuenta las penalizaciones immediatas que se obtienen en los primeros estados del acantilado.

4 Model-based

El algoritmo Model-Based es un método que primero construye un modelo del entrono observando como funciona, y luego usa ese modelo para encontrar la mejor política. Es diferente a modelos como Q-learning que aprenden directamente sin crear un modelo.

4.1 Análisis previo

El la assignatura de *Sistemas Inteligentes Distribuidos* hemos visto que existen diferentes implementaciones de algoritmos basados en modelo. En las clases de teoría se presentó el enfoque Model-Based con Monte Carlo, mientras que en las sesiones de laboratorio trabajamos con Estimación Directa. Para esta práctica, hemos decidido implementar el método con Estimación Directa, ya que nos permitía trabajar con el código base proporcionado.

Las ventajas del algoritmo Model-Based son:

- Una vez que tiene el modelo, puede planificar sin necesitar más interacciones con el entorno.
- Es más fácil entender qué está aprendiendo el agente porque podemos ver el modelo.
- Se puede actualizar el modelo sin empezar de cero.

Las desventajas son:

- Es difícil explorar bien para construir un modelo preciso.
- Usa más memoria que otros métodos.
- Si el modelo no es muy bueno, los errores se propagan y afectan la política final.
- La iteración de valor sobre un modelo estimado en espacios de estados grandes resulta costosa computacionalmente.

4.2 Implementación

La implementación está basada en el notebook de laboratorio, pero adaptada para CliffWalking. El método de Estimación Directa genera, en cada iteración, varias trayectorias aleatorias. A partir de las frecuencias observadas estima las probabilidades de transición y las recompensas, y sobre este modelo provisional aplica iteración de valor para actualizar $V(s)$. El proceso se resume en un ciclo que alterna continuamente dos fases: exploración (muestreo de trayectorias) y planificación (actualización de valores).

1. **Exploración**: El agente hace movimientos aleatorios y cuenta cuántas veces pasa de un estado a otro para estimar las probabilidades de transición $P(s'|s, a)$.
2. **Planificación**: Usa el modelo estimado para calcular la mejor política con iteración de valor.

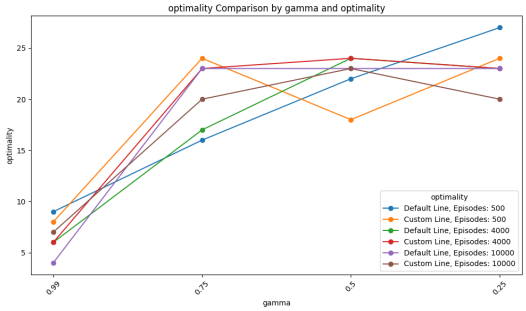
Para CliffWalking hemos hecho algunos cambios como añadir un criterio para parar cuando los valores de los estados dejan de cambiar mucho y evaluamos cómo va la política cada cierto tiempo. El código completo está en el Anexo A.2.

4.3 Experimentación

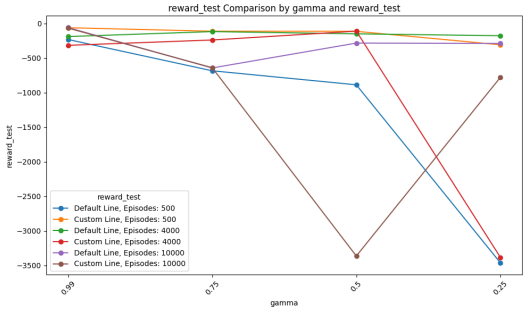
En cuanto a la experimentación , se han realizado ejecuciones con todas las combinaciones de los mismos valores de los hiperparámetros que en el apartado del algoritmo anterior Value Iteration 3.3.

4.4 Resultados

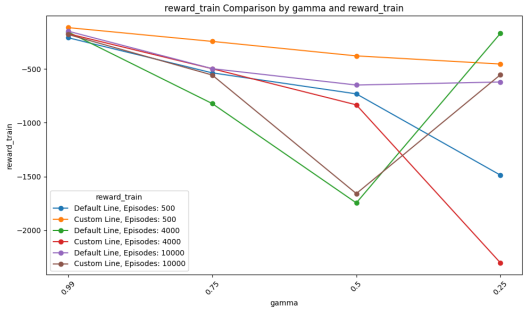
Los resultados obtenidos para el algoritmo Model-Based se presentan en la Figura 3, siguiendo el formato experimental descrito en la Sección 2.



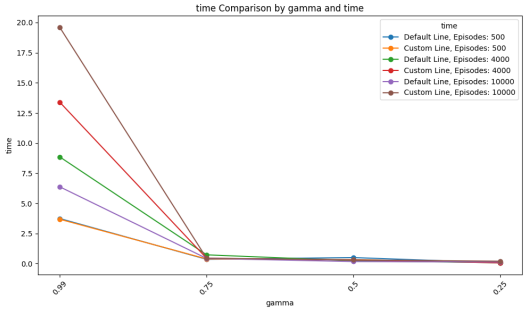
(a) Resultados para valores de gamma según el criterio de optimalidad de la solución.



(b) Resultados para valores de gamma según el criterio de Reward de test.



(c) Resultados para valores de gamma según el criterio de Reward de entrenamiento.



(d) Resultados para valores de gamma según el criterio de tiempo de ejecución.

Figure 3: Resultados de la experimentación del algoritmo Model-Based.

Analizando los resultados obtenidos, observamos que el factor gamma juega un papel importante en la calidad de la política aprendida. Cuando `gamma=0.99`, todas las configuraciones (tanto con la recompensa por defecto como con la recompensa "custom" y usando 500, 4000 o 10000 episodios) consiguen una distancia muy pequeña a la política óptima y reportan recompensas medias de test próximas a cero. Esto refleja que valorar el largo plazo permite al agente planificar rutas seguras y evitar el acantilado con eficacia. Sin embargo, reducir gamma a valores intermedios o bajos (0.75, 0.5, 0.25) produce un aumento "casi" monótono del número de acciones subóptimas y una caída de las recompensas en test (llegando incluso a –3000), lo que demuestra que el agente pasa a ignorar el riesgo acumulado del precipicio y adopta comportamientos miopes y peligrosos.

En la fase de entrenamiento también apreciamos que gamma elevado facilita una convergencia más estable, mientras que al reducir gamma se generan fuertes oscilaciones y caídas bruscas en la recompensa. Además, añadir la función de recompensa custom aporta una pequeña mejora en optimalidad y en recompensa de test cuando $\gamma \geq 0.75$.

En cuanto al tiempo de ejecución, planificar con gamma alto exige un coste computacional notable (hasta 20 segundos con 10.000 episodios y reward custom). Con $\gamma \leq 0.75$ el tiempo se desploma a unos pocos décimos de segundo, pero este coste computacional se justifica por la calidad superior de las políticas obtenidas.

5 Q-learning

5.1 Análisis previo

El Q-learning es un algoritmo *model-free* que permite estimar de forma iterativa la función de valor de acción $Q(s, a)$ sin necesidad de conocer el modelo interno del entorno. Esto resulta especialmente útil en CliffWalking-v1 con modo *slippery*, donde las transiciones son estocásticas y construir un modelo preciso podría ser complicado. Al ser *off-policy*, el agente puede aprender de experiencias generadas por políticas muy exploratorias, lo que favorece un uso más eficiente de los datos recogidos durante el entrenamiento. Además, su convergencia teórica está garantizada en entornos discretos siempre que la política de exploración—por ejemplo, ϵ-greedy con decaimiento adecuado—asegure una cobertura suficiente de estados y acciones.

No obstante, el Q-learning presenta ciertos retos en escenarios con penalizaciones severas como el acantilado de CliffWalking. La propagación lenta de recompensas negativas puede requerir un gran número de episodios para que el agente aprenda a evitar sistemáticamente el borde, particularmente si el factor de descuento γ se sitúa cercano a 1 o la tasa de aprendizaje α es demasiado baja. Asimismo, la sensibilidad a los hiperparámetros obliga a ajustar cuidadosamente α, γ y los parámetros de exploración (ϵ inicial y su decaimiento) para lograr un buen equilibrio entre exploración y explotación. Un ajuste inadecuado puede derivar en agentes excesivamente cautelosos, que eviten explorar rutas potencialmente óptimas, o en agentes que continúen sufriendo caídas frecuentes.

En entornos estocásticos como CliffWalking, las estimaciones de Q pueden variar mucho debido a la aleatoriedad de las transiciones, lo que puede provocar que el aprendizaje sea inestable si los hiperparámetros no están bien ajustados. Por ejemplo, si el valor de ϵ es demasiado alto, el agente explorará en exceso y caerá muchas veces en el acantilado; en cambio, si es demasiado bajo, dejará de explorar y no encontrará buenas soluciones. Además, como las penalizaciones negativas se propagan lentamente, es importante ajustar correctamente la tasa de aprendizaje α y el factor de descuento γ para que el agente aprenda a evitar las zonas peligrosas sin dejar de buscar rutas mejores. En resumen, entrenar con Q-learning en este entorno requiere encontrar un buen equilibrio entre explorar el entorno y evitar errores costosos.

5.2 Implementación

La implementación de Q-learning parte del notebook de laboratorio manteniendo la estructura principal de la clase de agente y el bucle de entrenamiento por episodios. En nuestra implementación la clase ha sido renombrada a `Qlearning` y se han añadido dos nuevos parámetros, `epsilon_decay` y `lr_decay`, que permiten aplicar estrategias de "decay" (hiperbólico o exponencial) a la exploración y a la tasa de aprendizaje. Estos decaimientos se calculan mediante los métodos `epsilon_fn` y `lr_fn`, que reciben el índice de episodio para ajustar dinámicamente ϵ y lr en cada paso. Además, se ha incorporado un método `train(num_episodes)` que centraliza el ciclo de entrenamiento, muestra el promedio de recompensa cada 500 episodios y devuelve el historial completo de recompensas, simplificando los bucles manuales del notebook. El código completo de la implementación se encuentra disponible en el Anexo A.3.

5.3 Experimentación

En cuanto a la experimentación realizada, se han ejecutado todas las combinaciones posibles de los siguientes valores de hiperparámetros:

1. Número de episodios: [500, 4000, 10000]
2. Gamma: [0.99, 0.75, 0.5, 0.25]
3. Epsilon (coeficiente de exploración): [0.1, 0.3, 0.5, 0.8]

4. Estrategia de decaimiento de ϵ: [none, hyperbolic, exponential]
   - **none**: el valor de ϵ permanece constante durante todo el entrenamiento.
   - **hyperbolic**: el valor de ϵ disminuye con la fórmula $\epsilon_t = \frac{\epsilon_0}{t}$, lo que genera una disminución suave y progresiva a lo largo del tiempo.
   - **exponential**: el valor de ϵ decrece siguiendo $\epsilon_t = \frac{\epsilon_0}{e^t}$, produciendo una caída más rápida en las primeras fases del entrenamiento.
5. Tasa de aprendizaje: [0.1, 0.01, 0.001]
6. Estrategia de decaimiento de la tasa de aprendizaje: [none, hyperbolic, exponential]
   - Se aplican los mismos principios que en el caso del decaimiento de ϵ.
7. Función de recompensa: [default, custom, final_100]
   - **default**: se utiliza la función de recompensa original del entorno `CliffWalking-v0`.
   - **custom**: se modifica el entorno para que la recompensa al alcanzar el estado terminal (estado 47) sea 0, en lugar del valor por defecto, facilitando la propagación del valor a lo largo del episodio.
   - **final_100**: se otorga una recompensa de 100 al alcanzar el objetivo final, manteniéndose el resto de transiciones sin cambios. Esto refuerza con mayor claridad las trayectorias exitosas.

5.4 Resultados

5.4.1 Influencia de γ en Q-learning

En las Figuras 4(a)–(d) se recogen los resultados de Q-learning al variar $\gamma \in \{0.99, 0.75, 0.5, 0.25\}$:

Tiempo de ejecución. El coste computacional está prácticamete determinado por el número de episodios, creciendo de forma lineal entre 500, 4000 y 10000 entrenamientos; la variación en γ introduce sólo pequeñas oscilaciones (del orden de décimas de segundo) atribuibles a la distribución de pasos por episodio.

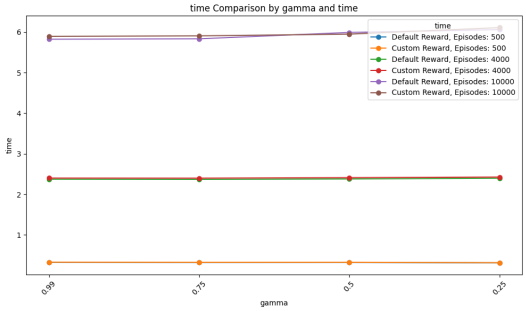
Recompensa durante el entrenamiento. La media de reward durante el aprendizaje mejora claramente al aumentar los episodios pasando de valores alrededor de -280 en 500 episodios a casi -238 con 10000. Respecto a γ, el rendimiento de entrenamiento alcanza un ligero máximo en $\gamma = 0.75$, mientras que $\gamma = 0.25$ provoca la peor propagación de las penalizaciones a largo plazo.

Recompensa en fase de test. Con solo 500 episodios, los valores bajos de γ (0.25, –0.5) producen recompensas de test menos negativas, pues el agente privilegia las recompensas inmediatas y evita caer en el acantilado sin aprender rutas óptimas. Sin embargo, con 10000 episodios, $\gamma = 0.99$ ofrece el mejor rendimiento en test, al permitir planificar trayectorias seguras considerando las penalizaciones futuras.

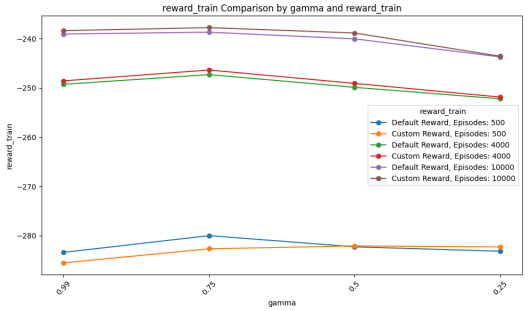
Optimalidad de la política. Con 10000 episodios, $\gamma = 0.99$ produce la política más cercana al óptimo ($\approx$ 18 acciones distintas), mientras que $\gamma = 0.25$ da lugar a políticas muy alejadas. En configuraciones con menores episodios, $\gamma = 0.75$ consigue los mejores resultados dentro de ese rango.

Conclusiones sobre γ:

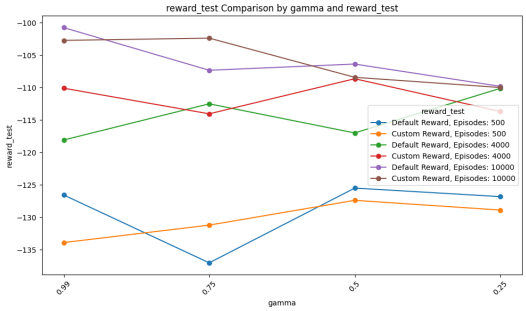
- Valores intermedios ($\gamma \approx 0.75$) favorecen el aprendizaje rápido en entornos ruidosos.



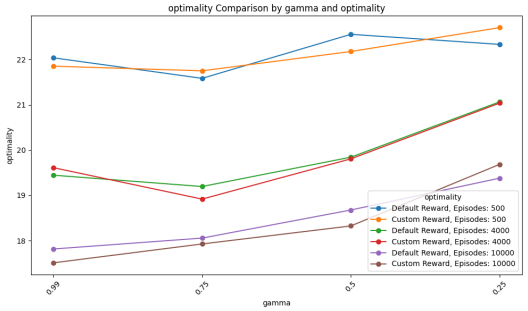
(a) Tiempo de ejecución.



(b) Reward medio durante el entrenamiento.



(c) Reward medio en fase de test.



(d) Optimalidad (número de acciones distintas frente a la política óptima).

Figure 4: Efecto de γ en las métricas clave de Q-learning.

- γ alto (0.99) es imprescindible para acercarse al óptimo cuando se dispone de suficientes episodios.
- Con pocos episodios, γ bajo puede reducir la penalización en test pero genera políticas muy subóptimas.
- Recomendamos $\gamma = 0.99$, $\geq 10\,000$ episodios y, si se desea mejorar la convergencia, la función de recompensa *custom*.

5.4.2 Influencia de la tasa de aprendizaje α

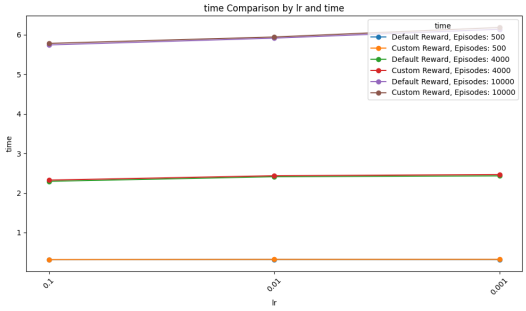
En la Figura 5 se muestran las métricas frente a los valores de $\alpha \in \{0.1, 0.01, 0.001\}$:

Tiempo de ejecución. Prácticamente invariable: el tiempo está dictado por el número de episodios.

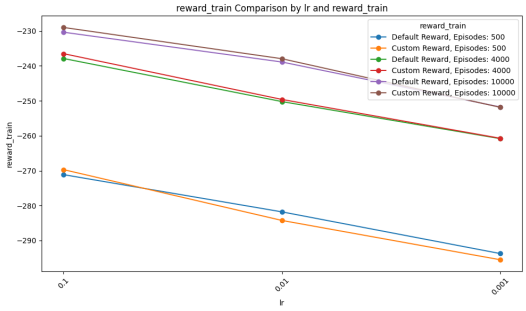
Recompensa en entrenamiento. La línea de reward_train muestra que $\alpha = 0.1$ converge más rápido al obtener el reward medio menos negativo (aprox. –230), mientras que $\alpha = 0.01$ y sobre todo $\alpha = 0.001$ ralentizan la mejora, alcanzando valores más negativos (–248 y –262 respectivamente).

Recompensa en fase de test. De manera consistente, $\alpha = 0.1$ produce el mejor reward_test (menos negativo, ≈ -100), seguido de $\alpha = 0.01$ ($\approx$ –107) y finalmente $\alpha = 0.001$ ($\approx$ –116). Esto indica que una tasa de aprendizaje alta favorece tanto la convergencia como la calidad de la política final.

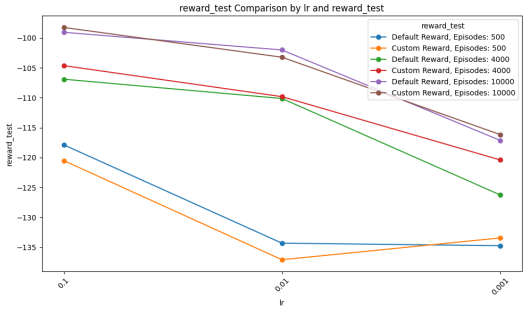
Optimalidad de la política. La métrica de optimalidad confirma el mismo patrón: $\alpha = 0.1$ minimiza las discrepancias frente a la política óptima ($\approx$16 acciones distintas), $\alpha = 0.01$ empeora ligeramente ($\approx$17) y $\alpha = 0.001$ resulta claramente peor ($\approx$22).



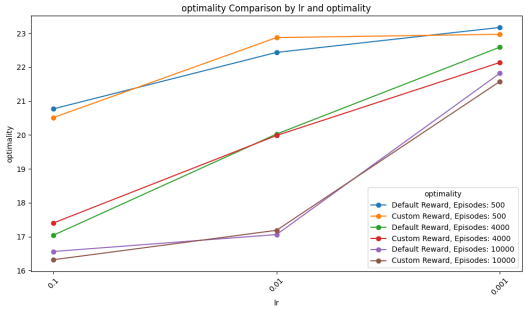
(a) Tiempo de ejecución.



(b) Reward medio durante el entrenamiento.



(c) Reward medio en fase de test.



(d) Optimalidad de la política.

Figure 5: Efecto de la tasa de aprendizaje α en Q-learning.

Conclusiones sobre α:

- $\alpha = 0.1$ es la opción más eficaz: acelera el entrenamiento, ofrece el mejor reward_test y genera la política más cercana al óptimo.
- $\alpha = 0.01$ modera la inestabilidad pero ralentiza la convergencia y empeora ligeramente todas las métricas.
- $\alpha = 0.001$ converge muy lentamente, con reward y optimalidad significativamente peores.

5.4.3 Influencia del decaimiento de la tasa de aprendizaje

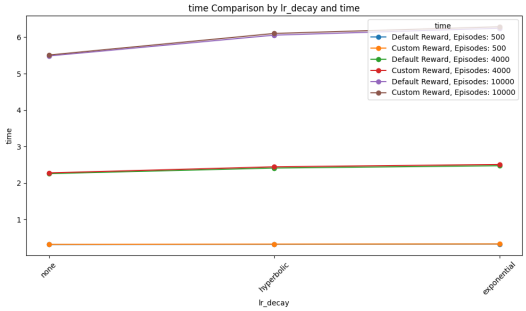
Comparamos tres esquemas de decaimiento (`none`, `hyperbolic`, `exponential`). La Figura 6 presenta los resultados:

Tiempo de ejecución. El coste extra de calcular la tasa adaptativa es despreciable en general, aunque aumenta ligeramente cuando el número de episodios es muy alto.

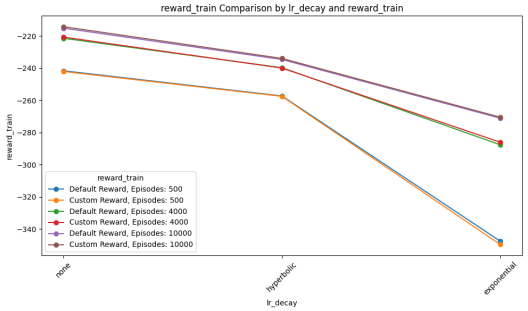
Recompensa en entrenamiento. Sin decaimiento (`none`) alcanza los mejores valores ($\approx$ –230). `hyperbolic` es intermedio ($\approx$ –235). `exponential` degrada sensiblemente el aprendizaje ($\approx$ –350).

Recompensa en fase de test. `none` ofrece el reward-test más alto ($\approx$ –80), `hyperbolic` intermedio ($\approx$ –100) y `exponential` el peor rendimiento ($\approx$ –140).

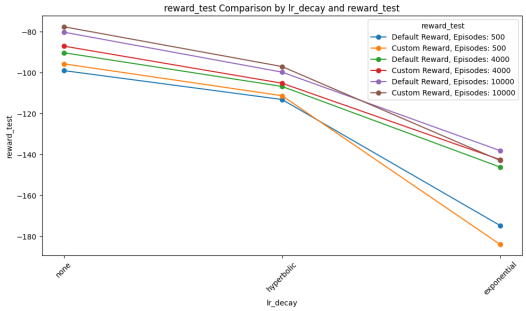
Optimalidad de la política. `none` obtiene $\approx$12 acciones distintas, `hyperbolic` $\approx$20 y `exponential` $\approx$23.



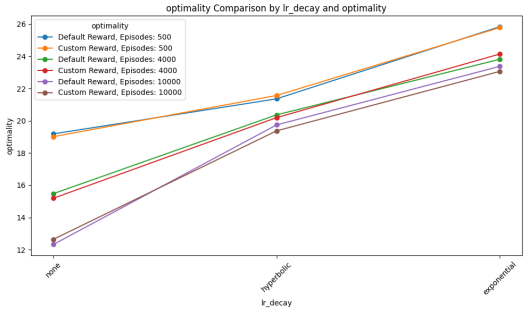
(a) Tiempo de ejecución.



(b) Reward medio durante el entrenamiento.



(c) Reward medio en fase de test.



(d) Optimalidad de la política.

Figure 6: Efecto de los esquemas de decaimiento de α en Q-learning.

Conclusiones sobre el decaimiento de α:

- No aplicar decaimiento (`none`) es la opción más eficaz.
- Un decaimiento suave (`hyperbolic`) funciona razonablemente, pero penaliza ligeramente la calidad final.
- El decaimiento exponencial resulta excesivo, cortando el aprendizaje prematuramente.

Es posible que no se haya elegido bien el valor inicial de la tasa de aprendizaje para este tipo de funciones de decay. Sería interesante en el futuro comparar las diferentes funciones de decay para valores iniciales superiores a 0.1. Se ha hecho un pequeño test para comprobarlo: comparando tres valores iniciales de $\alpha_0 \in \{0.9, 0.5, 0.1\}$ con $\gamma = 0.99$, $\epsilon = 0.1$, reward *default* y 10000 episodios, aplicando esquemas de decaimiento `none`, `hyperbolic` y `exponential`. Los resultados muestran que para $\alpha_0 = 0.9$ y 0.5 el decaimiento `hyperbolic` logra el mejor reward_test (aprox. –64) y mejor optimalidad, mientras que para $\alpha_0 = 0.1$ el mejor rendimiento se obtiene sin decaimiento ($\approx$–58, optimality=7). En todos los casos el decaimiento `exponential` resulta demasiado agresivo y degrada significativamente la calidad final de la política.

6 Actor-Critic y REINFORCE (extra)

Como métodos basados en optimización de política, se ha decidido escoger REINFORCE con la implementación realizada en los *notebooks* del laboratorio. Después de probar con muchísimos hiperparámetros, se ha llegado a la conclusión que no era posible obtener una solución posible. Esto es debido a que en esta implementación se utiliza un modelo lineal, la cual cosa imposibilita la obtención de un resultado óptimo en este entorno debido a su elevada complejidad (mayor número de estados y mayor penalización que en el entorno de **FrozenLake**.

A modo de dar una solución a este problema desde una vertiente de optimización de política, se ha optado por implementar un método **Actor-Critic**.

6.1 Análisis previo

Actor-Critic es un algoritmo **on-policy** que une dos enfoques del aprendizaje por refuerzo.

Por un lado, existe el **Actor**, que se encarga de aprender una política de forma directa a partir de los datos recopilados durante la ejecución de la exploración. Por otro lado, existe el **Critic** que se encarga de aprender una función de valoración de estados (V o Q) para evaluar cuanto de buenas son las decisiones del actor y poder así corregir o no el aprendizaje.

Las ventajas de **Actor-Critic** se pueden resumir en:

- Una mayor estabilidad durante el aprendizaje gracias a la valoración realizada por el crítico sobre el actor
- Una mayor eficiencia gracias al aprendizaje de la función de valor y política lo que permite una mejor exploración
- Una mayor robustez en entornos estocásticos gracias a la función de valoración
- Actualización continua, lo que se traduce en una mayor adaptabilidad a las variaciones encontradas durante el entrenamiento.

Las desventajas por otro lado, se pueden resumir en:

- La convergencia es más lenta en entornos donde hay una gran penalización por realizar una acción incorrecta.
- Sensibilidad a hiperparámetros, donde una mala inicialización del *learning rate* puede conllevar a conseguir soluciones sub-óptimas.
- Complejidad adicional, debido a la existencia de 2 *agentes* (el actor y el crítico) puede conllevar problemas en la convergencia si alguno de los 2 no realiza un correcto aprendizaje.

6.2 Implementación

La implementación del código para este algoritmo se ha basado en el algoritmo 1, donde se define durante el entrenamiento como se actualiza tanto el crítico (relacionado con la función de valoración V) como el actor (relacionado con la ejecución de las acciones).

Algorithm 1 Actor-Critic tabular (con valor estatal como baseline)

Require: Environment $\mathcal{E}$, learning rates $\alpha > 0$ (actor) and $\beta > 0$ (critic), discount factor $\gamma \in [0, 1]$, maximum steps per episode T_{max}, number of episodes $N_{episodes}$

Initialize policy parameters $\theta \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|}$ to zeros
Initialize value function $V(s) \in \mathbb{R}^{|\mathcal{S}|}$ to zeros for all $s \in \mathcal{S}$
Initialize an empty list *Rewards* to store episode rewards
for *episode* $= 1$ to $N_{episodes}$ **do**
    Initialize the environment and observe the initial state s_0
    Initialize the total reward for the episode $R \leftarrow 0$
    for $t = 1$ to T_{max} **do**
        Calculate action probabilities $\pi(a|s_t, \theta)$ using the softmax function:

$$\pi(a|s_t, \theta) = \frac{\exp(\theta(s_t, a) - \max_{a'} \theta(s_t, a'))}{\sum_{a' \in \mathcal{A}} \exp(\theta(s_t, a') - \max_{a''} \theta(s_t, a''))}$$

        Select action a_t according to the probability distribution $\pi(\cdot|s_t, \theta)$
        Execute action a_t in the environment and observe the next state s_{t+1} and reward r_{t+1}
        Set *done* $\leftarrow$ whether the episode terminated at step $t + 1$
        Calculate the TD target: $y_t = r_{t+1} + \gamma V(s_{t+1})(1 - done)$
        Calculate the TD error: $\delta_t = y_t - V(s_t)$
        Update the value function (critic):

$$V(s_t) \leftarrow V(s_t) + \beta\delta_t$$

        Calculate the gradient of the log policy: $\nabla_{\theta(s_t,\cdot)} \log \pi(a_t|s_t, \theta)$
        Initialize $\nabla_{\theta(s_t,\cdot)} \log \pi(a_t|s_t, \theta) = -\pi(\cdot|s_t, \theta)$
        Set $\nabla_{\theta(s_t,a_t)} \log \pi(a_t|s_t, \theta) \leftarrow \nabla_{\theta(s_t,a_t)} \log \pi(a_t|s_t, \theta) + 1$
        Update the policy parameters (actor):

$$\theta(s_t, a) \leftarrow \theta(s_t, a) + \alpha\delta_t \nabla_{\theta(s_t,a)} \log \pi(a_t|s_t, \theta) \quad \forall a \in \mathcal{A}$$

        Update the total reward: $R \leftarrow R + r_{t+1}$
        Update the current state: $s_t \leftarrow s_{t+1}$
        if *done* is true **then**
            Break the inner loop
        end if
    end for
end for
return *Rewards*

6.3 Experimentación

En cuanto a la experimentación realizada, se han realizado ejecuciones con todas las combinaciones de los valores de los hiperparámetros siguientes:

1. Número de episodios : [500, 4000, 10000]
2. Gamma : [0.99, 0.75, 0.5, 0.2]
3. Learning Rate : [0.1, 0.01, 0.001]
4. Función de recompensa : [default, custom]

La función de recompensa custom hace referència a la obtención de un valor de recompensa de 0

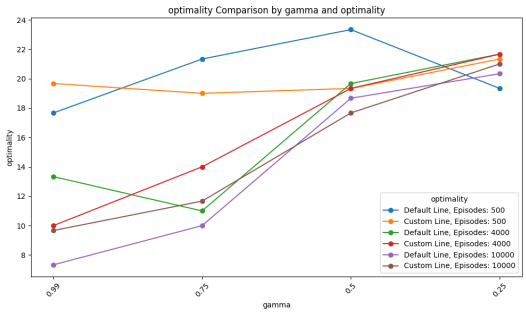
si el próximo estado es el terminal, es decir, el estado 47.

Además a modo de simplificación, dado que los *learning rates* de tanto el crític como el actor deben ser idealmente diferentes, se ha optado por multiplicar el valor elegido por 0.1 y definirlo como el *learning rate* del **Critic**

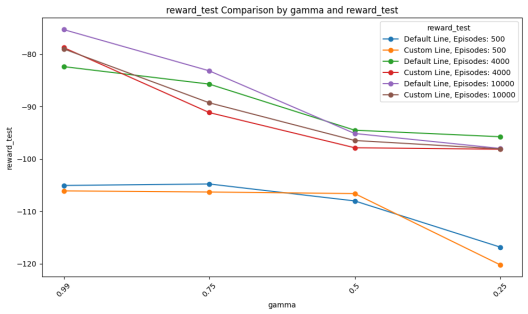
6.4 Resultados

En el caso de Actor-Critic, dado que se le añade el hiperparámetro del *learning rate*, por tal de reducir tanto el tamaño de lineas en los gràficos como el número de gràficas, los valores que aparecen corresponden a la media respecto a los valores obtenidos por las otras métricas. Es decir, en las figuras 7 cada valor de cada gráfico corresponde con la media del resulta obtenido por las 3 iteraciones adicionales de los 3 paràmetros del *learning rate*.

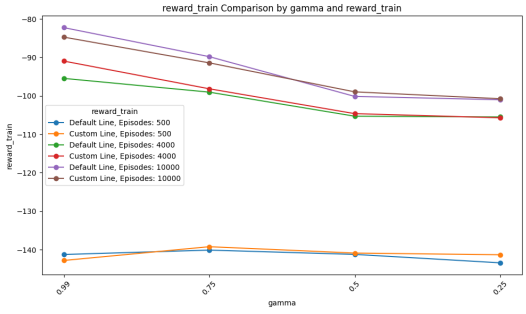
En las siguientes 4 figuras se puede observar para diferentes valores de gamma (Eje X), los resultados obtenidos para las siguientes métricas a estudiar:



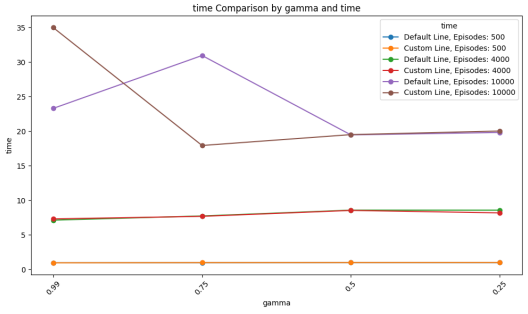
(a) Resultados para valores de gamma según el criterio de optimalidad de la solución.



(b) Resultados para valores de gamma según el criterio de Reward de test.



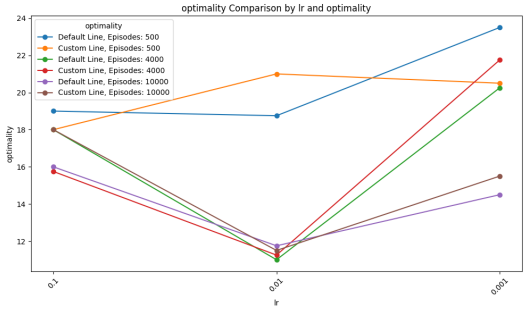
(c) Resultados para valores de gamma según el criterio de Reward de entrenamiento.



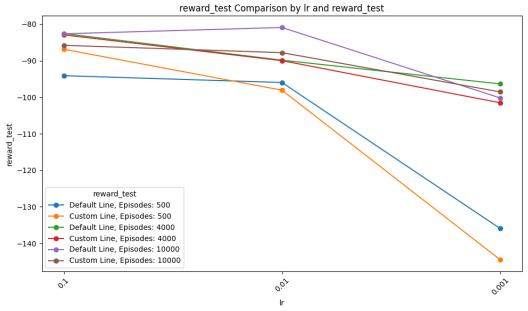
(d) Resultados para valores de gamma según el criterio de tiempo de ejecución.

Figure 7: Resultados de la experimentación del algoritmo Actor-Critic para gamma como eje X.

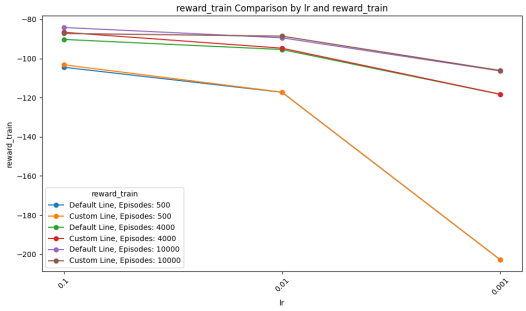
En las siguientes 4 figuras, se realiza el mismo proceso anterior pero esta vez con los diferentes valores para el *learning rate*.



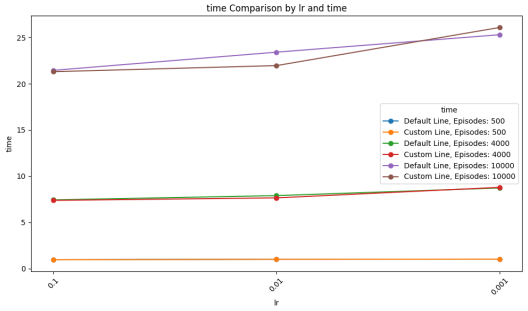
(a) Resultados para valores de *learning rate* según el criterio de optimalidad de la solución.



(b) Resultados para valores de *learning rate* según el criterio de reward de test.



(c) Resultados para valores de *learning rate* según el criterio de reward de entrenamiento.



(d) Resultados para valores de *learning rate* según el criterio de tiempo de ejecución.

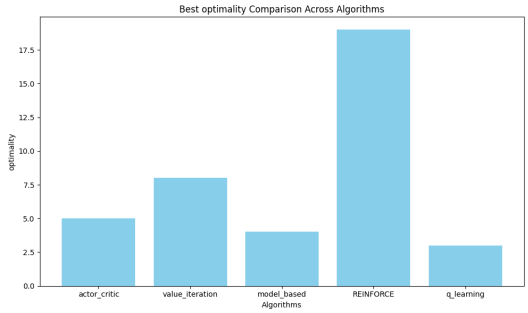
Figure 8: Resultados de la experimentación del algoritmo Actor-Critic para *learning rate* como eje X.

Qué se puede extraer de las gráficas 7, 8?

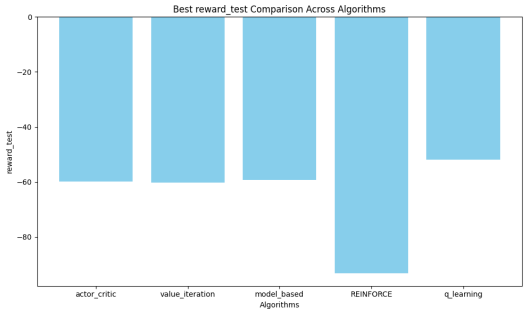
- Para valores de gamma cercanos a 1:
  - Se obtienen mejores rendimientos con un número elevado de iteraciones. Además la función de recompensa base supera ligeramente a la custom.
- Para valores de gamma cercanos a 0:
  - Se obtienen peores valores de las métricas de rendimiento, lo que es una consecuencia común y es que se le da menos importancia a las recompensas immediatas y siendo el entorno propenso a tener una penalización muy grande por la exploración, hace que no se propague esta información.
- Para valores de *learning rate*:
  - A simple vista, el mejor valor és el de 0.01. Esto es debido a un mejora extraordinaria en cuanto a la optimalidad de la solución a diferencia de los valores 0.1 y 0.001. Se le añade además que el número de iteraciones debe ser alto para una mejor convergencia.
  - El tiempo de ejecución es independiente debido a que está directamente relacionado con el número de iteraciones.
  - Los rewards son similares entre los valores de 0.1 y 0.01 pero dependen directamente del número de iteraciones para converger hacia una mejor solución.
  - Cabe recordar que los valores de reward son superiores a otros algoritmos debido a que es la media sobre los valores obtenidos para los diferentes valores de gamma, que, como se ha demostrado, aumentan según disminuye la gamma.

7 Análisis global

Una vez se han realizado todos los experimentos para cada algoritmo, a continuación se observan 2 gráficos donde se observan los mejores resultados obtenidos por los diferentes algoritmos respecto a las métricas de **optimalidad** y **reward test**. Se ha añadido también los resultados del método **REINFORCE** para poder demostrar así su ineficacia en este entorno.



(a) Resultados para valores de *optimality* para los diferentes algoritmos.



(b) Resultados para valores de *reward test* para los diferentes algoritmos.

Figure 9: Mejores métricas obtenidas por los algoritmos.

En términos de optimality, que evalúa cuán cercana es la política obtenida al comportamiento óptimo, el algoritmo que se destaca es q-learning, con un valor de 3, el más bajo entre los analizados. Este valor indica que la política aprendida por q-learning es la que más se aproxima al comportamiento teóricamente ideal. Le siguen model-based (4) y actor-critic (5), mostrando también una proximidad aceptable al óptimo.

En cuanto a los valores de reward en test, Q-Learning también alcanza el mejor resultado, con aproximadamente –52, por delante de Model-Based, Value Iteration y Actor Critic (cerca de –60). Otra vez, REINFORCE obtiene un valor muy por debajo de los otros algoritmos.

Alg	**episodes**	**gamma**	**reward_signal**	**epsilon**	**epsilon_decay**
value_iteration	500	0.99	custom	0.0	none
model_based	10000	0.99	default	0.0	none
REINFORCE	500	0.25	default	0.0	none
actor_critic	4000	0.99	custom	0.0	none
q_learning	10000	0.5	default	0.1	exponential

Table 1: Parte 1: Parámetros de configuración de entrenamiento.

Alg	**lr**	**lr_decay**	**reward_train**	**reward_test**	**time**	**optimality**
value_iteration	0.0	0	-74.23	-60.15	2.41	8
model_based	0.0	0	-147.40	-59.35	6.37	4
REINFORCE	0.1	0.99	-110.61	-96.55	0.71	19
actor_critic	0.01	0	-85.78	-74.25	6.80	5
q_learning	0.01	none	-81.55	-62.80	4.50	3

Table 2: Parte 2: Resultados de rendimiento y otros hiperparámetros.

En las tablas anteriores podemos ver las mejores configuraciones obtenidas comparando los valores de optimalidad de la política final. Así pues, volvemos a ver que la mayor optimalidad se obtiene con

el Q-learning, siguiendo por model-based y actor-ciritic. En la tabla 2 se pueden ver los valores promedio obtenida en fase de prueba (reward test) con las políticas más cercanas a la óptima. El algoritmo model-based se posiciona como el de mejor desempeño con un valor de -59.35, seguido de value iteration (-60.15) y q-learning (-62.80). Estos resultados indican que estos algoritmos son más efectivos en generar políticas que maximizan la recompensa esperada en entornos desconocidos o no vistos durante el entrenamiento. Sin embargo, los valores difieren con los mostrados en la Figura 9b, donde se recogen los mejores resultados absolutos de `reward_test` obtenidos por cada algoritmo en cualquier configuración, lo que pone de manifiesto que la estocasticidad del entorno "slippery" puede hacer que una política muy cercana al óptimo en número de acciones no alcance necesariamente la recompensa máxima en todas las ejecuciones de prueba.

Tomando en conjunto ambas métricas, q-learning se presenta como el algoritmo más balanceado, combinando un excelente valor de optimality con un reward-test competitivo, lo que lo posiciona como una alternativa sólida para entornos similares. Model-based y value-iteration destacan por su alto rendimiento en test, aunque con valores de optimality ligeramente inferiores. actor-critic mantiene un comportamiento consistente, aunque sin sobresalir significativamente en ninguna de las métricas. Finalmente, REINFORCE queda descartado como una opción viable para el entorno seleccionado.

8 Ejecución

Este proyecto permite ejecutar los entrenamientos y experimentaciones de dos formas principales: mediante la ejecución de un conjunto completo de combinaciones de hiperparámetros, o bien lanzando un único experimento con parámetros específicos desde la línea de comandos.

8.1 Ejecución completa de experimentos

Para ejecutar automáticamente una batería de experimentos con múltiples configuraciones de hiperparámetros, es suficiente con ejecutar el script principal `execution.py`:

```
python execution.py
```

Este script se encarga de recorrer todas las combinaciones de parámetros definidas en el archivo `set_params.py`, dentro de la lista `PARAMETRES`. Para cada combinación, se ejecuta `experimentation.py`, que entrena y evalúa el agente correspondiente. Los resultados (recompensas, tiempo de entrenamiento y optimalidad) se almacenan en archivos CSV dentro del directorio `output/`, además de generar las visualizaciones correspondientes en la carpeta `results/<nombre_algoritmo>/`.

Para modificar los experimentos que se van a ejecutar es necesario editar el contenido de la lista `PARAMETRES` en el archivo `set_params.py`, especificando los valores deseados para los siguientes parámetros: `alg`, `episodes`, gamma, `epsilon`, `epsilon_decay`, `reward_signal`, `lr`, y `lr_decay`.

8.2 Ejecución manual de un experimento individual

También es posible ejecutar un único experimento directamente con `experimentation.py`, especificando manualmente los valores deseados de los hiperparámetros mediante argumentos de línea de comandos. Por ejemplo, para lanzar un entrenamiento del algoritmo Q-Learning durante 500 episodios, con $\gamma = 0.9$ y una tasa de aprendizaje de 0.01, se puede utilizar:

```
python experimentation.py --alg q_learning --episodes 500 --gamma 0.9 --lr 0.01
```

El script también acepta otros argumentos como `--exp_id`, `--rew` (tipo de recompensa), `--epsilon`, `--epsilon_decay` y `--lr_decay`. Para consultar todos los argumentos disponibles, puede ejecutarse:

```
python experimentation.py --help
```

Al finalizar la ejecución, el agente entrenado se evalúa automáticamente, y los resultados se guardan en formato CSV dentro del directorio `output/`. También se generan representaciones gráficas de las recompensas y de la política aprendida.

A Código Python

A.1 Código del Value Iteration

```python
import numpy as np
import gymnasium as gym
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import time

class ValueIteration:
    def __init__(self, env: gym.Env, gamma:float=0.9, num_episodes:int=1000, t_max:int=100):
        self.env = env.unwrapped
        self.V = np.zeros(self.env.observation_space.n)
        self.gamma = gamma
        self.num_episodes = num_episodes
        self.t_max = t_max




    def calc_action_value(self, state, action):
        action_value = sum([prob * (reward + self.gamma * self.V[next_state])
                            for prob, next_state, reward, _
                            in self.env.P[state][action]])

        return action_value

    def select_action(self, state):
        best_action = 0
        best_value = -np.inf
        for action in range(self.env.action_space.n):
            action_value = self.calc_action_value(state, action)
            if best_value < action_value:
                best_value = action_value
                best_action = action
        return best_action

    def value_iteration(self):
        max_diff = 0
        for state in range(self.env.observation_space.n):
                state_values = []
                for action in range(self.env.action_space.n):
                    state_values.append(self.calc_action_value(state, action))
                new_V = max(state_values)
                diff = abs(new_V - self.V[state])
                if diff > max_diff:
                    max_diff = diff
                self.V[state] = new_V

        return self.V, max_diff

    def policy(self):
        policy = np.zeros(self.env.observation_space.n)
        for s in range(self.env.observation_space.n):
            Q_values = [self.calc_action_value(s,a) for a in range(self.env.action_space.n)]
            policy[s] = np.argmax(np.array(Q_values))
        return policy

    def check_improvements(self):
        reward_test = 0.0
        for i in range(20):
```

```
            total_reward = 0.0
            state, _ = self.env.reset()
            for i in range(self.t_max):
                action = self.select_action(state)
                new_state, new_reward, is_done, truncated, _ = self.env.step(action)
                total_reward += new_reward
                if is_done:
                    break
                state = new_state
            reward_test += total_reward
        reward_avg = reward_test / 20
        return reward_avg

    def train(self, num_episodes:int=1000):
        rewards = []
        max_diffs = []
        t = 0
        best_reward = -1000
        ready_to_compare = False

        while not ready_to_compare or max_diffs[-1] > 0.05:
            _, max_diff = self.value_iteration()

            max_diffs.append(max_diff)
            t += 1
            reward_test = self.check_improvements()
            print(f"Iteration {t}, reward_test of {reward_test}, max_diff = " + str(
max_diff))
            rewards.append(reward_test)

            if reward_test > best_reward:
                best_reward = reward_test

            if len(max_diffs) > 1:
                ready_to_compare = True


        return rewards

    def test(self, num_episodes:int=1000):
        rewards = []
        for i in range(num_episodes):
            total_reward = 0.0
            state, _ = self.env.reset()
            for i in range(self.t_max):
                action = self.select_action(state)
                new_state, new_reward, is_done, truncated, _ = self.env.step(action)
                total_reward += new_reward
                if is_done:
                    break
                state = new_state
            rewards.append(total_reward)
        return rewards
```

Listing 1: Código fuente del modelo Value Iteration

A.2 Código del Model-Based

```python
import numpy as np
import collections
import time
import gymnasium as gym

class ModelBased:
    def __init__(self, env, gamma=0.99, num_trajectories=10, t_max=100, threshold
    =0.01):
        self.env = env
        self.state = None
        self.rewards = collections.defaultdict(float)
        self.transits = collections.defaultdict(collections.Counter)
        self.V = np.zeros(self.env.observation_space.n)
        self.gamma = gamma
        self.num_trajectories = num_trajectories
        self.t_max = t_max
        self.threshold = threshold
        self.reset()

    def reset(self):
        """Reset the agent state and initialize the environment"""
        self.state, _ = self.env.reset()

    def play_n_random_steps(self, count):
        """Collect transition data by randomly exploring the environment"""
        for _ in range(count):
            action = self.env.action_space.sample()
            new_state, reward, is_done, truncated, _ = self.env.step(action)

            self.rewards[(self.state, action, new_state)] = reward
            self.transits[(self.state, action)][new_state] += 1

            if is_done:
                self.reset()
            else:
                self.state = new_state

    def calc_action_value(self, state, action):
        """Calculate the expected value of taking an action in a state"""
        target_counts = self.transits[(state, action)]
        total = sum(target_counts.values())

        if total == 0:
            return 0.0

        action_value = 0.0
        for next_state, count in target_counts.items():
            prob = count / total  # Estimated transition probability
            reward = self.rewards[(state, action, next_state)]
            action_value += prob * (reward + self.gamma * self.V[next_state])

        return action_value

    def select_action(self, state):
        """Select the best action for a state based on current estimates"""
        best_action, best_value = None, None

        for action in range(self.env.action_space.n):
            action_value = self.calc_action_value(state, action)
            if best_value is None or best_value < action_value:
                best_value = action_value
                best_action = action

        return best_action
```

```
64
65      def value_iteration(self):
66          """Perform one step of value iteration using the estimated model"""
67          # First collect more data about the environment
68          self.play_n_random_steps(self.num_trajectories)
69
70          max_diff = 0
71          for state in range(self.env.observation_space.n):
72              state_values = [
73                  self.calc_action_value(state, action)
74                  for action in range(self.env.action_space.n)
75              ]
76
77              new_V = max(state_values) if state_values else 0
78
79              diff = abs(new_V - self.V[state])
80              if diff > max_diff:
81                  max_diff = diff
82
83              self.V[state] = new_V
84
85          return self.V, max_diff
86
87      def policy(self):
88          """Extract the current greedy policy from the value function"""
89          policy = np.zeros(self.env.observation_space.n, dtype=int)
90
91          for s in range(self.env.observation_space.n):
92              Q_values = [self.calc_action_value(s, a) for a in range(self.env.
    action_space.n)]
93              policy[s] = np.argmax(np.array(Q_values)) if Q_values else 0
94
95          return policy
96
97      def train(self, num_episodes=1000):
98          """Train the agent using model-based value iteration"""
99          rewards = []
100         max_diffs = []
101         best_reward = -np.inf
102         iteration = 0
103
104         while iteration < num_episodes:
105             _, max_diff = self.value_iteration()
106             max_diffs.append(max_diff)
107
108             reward_test = self.check_improvements()
109             rewards.append(reward_test)
110
111             iteration += 1
112
113             if iteration % 50 == 0:
114                 print(f"Iteration {iteration}, reward: {reward_test:.2f}, max diff: {
    max_diff:.6f}")
115
116             if reward_test > best_reward:
117                 best_reward = reward_test
118
119             if max_diff < self.threshold:
120                 print(f"Converged after {iteration} iterations with max diff {max_diff
    :.6f}")
121                 break
122
123         return rewards
124
125     def check_improvements(self):
```

```
        """Evaluate the current policy by running test episodes"""
        reward_test = 0.0
        num_test_episodes = min(10, self.t_max)  # Limit the number of test episodes

        for _ in range(num_test_episodes):
            total_reward = 0.0
            state, _ = self.env.reset()

            for _ in range(self.t_max):
                action = self.select_action(state)
                new_state, reward, is_done, truncated, _ = self.env.step(action)
                total_reward += reward

                if is_done:
                    break

                state = new_state

            reward_test += total_reward

        return reward_test / num_test_episodes

    def test(self, num_episodes=20):
        rewards = []

        for _ in range(num_episodes):
            total_reward = 0.0
            state, _ = self.env.reset()

            for _ in range(self.t_max):
                action = self.select_action(state)
                new_state, reward, is_done, truncated, _ = self.env.step(action)
                total_reward += reward

                if is_done:
                    break

                state = new_state

            rewards.append(total_reward)

        return rewards
```

Listing 2: Código fuente del modelo Model-Based

A.3 Código del Q-Learning

```python
import numpy as np
import random
import gymnasium as gym
from .utils import print_policy, draw_history, draw_rewards
import math

class Qlearning:
    """
    Algoritmo Q-learning
    """
    def __init__(self, env, gamma, learning_rate, epsilon, t_max,
                 epsilon_decay="none", lr_decay="none"):
        self.env = env
        self.Q = np.zeros((env.observation_space.n, env.action_space.n))
        self.gamma = gamma
        self.lr0 = learning_rate
        self.epsilon0 = epsilon
        self.t_max = t_max
        self.epsilon_decay = epsilon_decay
        self.lr_decay = lr_decay

    def lr_fn(self, t):
        """Devuelve learning rate en el episodio t."""
        if self.lr_decay == "none":
            return self.lr0
        elif self.lr_decay == "hyperbolic":
            return self.lr0 / t
        elif self.lr_decay == "exponential":
            return self.lr0 / math.exp(t)
        else:
            raise ValueError(f"lr_decay desconocido: {self.lr_decay}")

    def epsilon_fn(self, t):
        """Devuelve epsilon en el episodio t."""
        if self.epsilon_decay == "none":
            return self.epsilon0
        elif self.epsilon_decay == "hyperbolic":
            return self.epsilon0 / t
        elif self.epsilon_decay == "exponential":
            return self.epsilon0 / math.exp(t)
        else:
            raise ValueError(f"epsilon_decay desconocido: {self.epsilon_decay}")

    def select_action(self, state, t, training=True):
        if training and random.random() <= self.epsilon_fn(t):
            return np.random.choice(self.env.action_space.n)
        else:
            return np.argmax(self.Q[state,])

    def update_Q(self, state, action, reward, next_state, t):
        best_next_action = np.argmax(self.Q[next_state,])
        td_target = reward + self.gamma * self.Q[next_state, best_next_action]
        td_error = td_target - self.Q[state, action]
        self.Q[state, action] += self.lr_fn(t) * td_error

    def learn_from_episode(self):
        state, _ = self.env.reset()
        total_reward = 0
        for i in range(self.t_max):
            action = self.select_action(state, i+1)
            new_state, new_reward, is_done, truncated, _ = self.env.step(action)
            total_reward += new_reward
            self.update_Q(state, action, new_reward, new_state, i+1)
            if is_done:
```

```
                break
            state = new_state
        return total_reward

    def policy(self):
        policy = np.zeros(self.env.observation_space.n)
        for s in range(self.env.observation_space.n):
            policy[s] = np.argmax(np.array(self.Q[s]))
        return policy

    def train(self, num_episodes):
        rewards = []
        for i in range(num_episodes):
            reward = self.learn_from_episode()
            rewards.append(reward)
            if i % 500 == 499:
                print(f"Episode {i+1}: {sum(rewards[-500:-1])/len(rewards[-500:-1])} "
)
        return rewards

    def test(self, num_episodes):
        is_done = False
        rewards = []
        for n_ep in range(num_episodes):
            state, _ = self.env.reset()
            total_reward = 0
            for i in range(self.t_max):
                action = self.select_action(state, 1, training=False)
                state, reward, is_done, truncated, _ = self.env.step(action)
                total_reward = total_reward + reward
                self.env.render()
                if is_done:
                    break
            rewards.append(total_reward)
            # if n_ep % 500 == 499:
            #     print(f"Episode {n_ep}: {sum(rewards[-500:-1])/len(rewards[-500:-1])
} ")
            print(f"Episode {n_ep}: {rewards[-1]} ")
        return rewards
```

Listing 3: Código fuente del modelo Q-Learning

A.4 Código del algoritmo REINFORCE

```python
import numpy as np
import gymnasium as gym
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import time
""" CliffWalking-v1"""

class ReinforceAgent:
    def __init__(self, env, gamma, learning_rate, lr_decay=1, seed=0, t_max =100):
        self.env = env
        self.gamma = gamma
        self.learning_rate = learning_rate
        self.lr_decay = lr_decay
        self.T_MAX = t_max  # Número máximo de pasos por episodio
        # Objeto que representa la política (J(theta)) como una matriz estados X acciones,
        # con una probabilidad inicial para cada par estado accion igual a: pi(a|s) = 1/|A|
        self.policy_table = np.ones((self.env.observation_space.n, self.env.action_space.n)) / self.env.action_space.n
        np.random.seed(seed)

    def select_action(self, state, training=True):
        action_probabilities = self.policy_table[state]
        if training:
            # Escogemos la acción según el vector de policy_table correspondiente a la acción,
            # con una distribución de probabilidad igual a los valores actuales de este vector
            return np.random.choice(np.arange(self.env.action_space.n), p=action_probabilities)
        else:
            return np.argmax(action_probabilities)

    def update_policy(self, episode):
        states, actions, rewards = episode
        discounted_rewards = np.zeros_like(rewards)
        running_add = 0
        for t in reversed(range(len(rewards))):
            running_add = running_add * self.gamma + rewards[t]
            discounted_rewards[t] = running_add

        loss = -np.sum(np.log(self.policy_table[states, actions]) * discounted_rewards) / len(states)
        policy_logits = np.log(self.policy_table)
        for t in range(len(states)):
            G_t = discounted_rewards[t]
            action_probs = np.exp(policy_logits[states[t]])
            action_probs /= np.sum(action_probs)
            policy_gradient = G_t * (1 - action_probs[actions[t]])
            policy_logits[states[t], actions[t]] += self.learning_rate * policy_gradient
            # Alternativa:
            #policy_gradient = 1.0 / action_probs[actions[t]]
            #policy_logits[states[t], actions[t]] += self.learning_rate * G_t * policy_gradient

        exp_logits = np.exp(policy_logits)
        self.policy_table = exp_logits / np.sum(exp_logits, axis=1, keepdims=True)
        return loss

    def learn_from_episode(self):
        state, _ = self.env.reset()
```

```
        episode = []
        done = False
        step = 0
        total_reward = 0

        while not done and step < self.T_MAX:
            action = self.select_action(state)
            next_state, reward, done, terminated, _ = self.env.step(action)
            episode.append((state, action, reward))
            state = next_state
            total_reward = total_reward + reward
            step += 1
            if done:
                break
        loss = self.update_policy(zip(*episode))
        self.learning_rate = self.learning_rate * self.lr_decay
        return total_reward, loss

    def policy(self):
        policy = np.zeros(self.env.observation_space.n)
        for s in range(self.env.observation_space.n):
            action_probabilities = self.policy_table[s]
            policy[s] = np.argmax(action_probabilities)
        return policy

    def train(self, num_episodes:int=1000):
        rewards = []
        for i in range(num_episodes):
            reward, loss = self.learn_from_episode()
            rewards.append(reward)
            print(f"Episode {i+1}/{num_episodes}, Reward: {reward}, Loss: {loss},
    Learning Rate: {self.learning_rate}")
        return rewards

    def test(self, num_episodes:int=1000):
        rewards = []
        for i in range(num_episodes):
            total_reward = 0.0
            state, _ = self.env.reset()
            for i in range(self.T_MAX):
                action = self.select_action(state, training=False)
                new_state, new_reward, is_done, truncated, _ = self.env.step(action)
                total_reward += new_reward
                if is_done:
                    break
                state = new_state
            rewards.append(total_reward)
        return rewards
```

Listing 4: Código fuente del modelo REINFORCE

A.5 Código del algoritmo Actor-Critic

```python
import numpy as np
import gymnasium as gym

class TabularActorCritic:
    def __init__(self, env, alpha=0.01, beta=0.1, gamma=0.99, t_max:int=30):
        self.env = env
        self.n_states = self.env.observation_space.n
        self.n_actions = self.env.action_space.n
        self.theta = np.zeros((self.n_states, self.n_actions))  # Política (preferencias)
        self.V = np.zeros(self.n_states)                    # Valor crítico
        self.alpha = alpha  # Tasa de aprendizaje para actor
        self.beta = beta    # Tasa de aprendizaje para crítico
        self.gamma = gamma  # Factor de descuento
        self.t_max = t_max  # Máximo número de pasos por episodio

    def get_action_probs(self, state):
        preferences = self.theta[state]
        exp_prefs = np.exp(preferences - np.max(preferences))
        return exp_prefs / np.sum(exp_prefs)

    def select_action(self, state):
        probs = self.get_action_probs(state)
        return np.random.choice(self.n_actions, p=probs)

    def update(self, state, action, reward, next_state, done):
        # TD target y error
        target = reward + (0 if done else self.gamma * self.V[next_state])
        td_error = target - self.V[state]

        # Actualiza el crítico (valor del estado)
        self.V[state] += self.beta * td_error

        # Gradiente de log   (a|s)
        probs = self.get_action_probs(state)
        grad_log = -probs
        grad_log[action] += 1  #     log   (a|s)

        # Actualiza el actor
        self.theta[state] += self.alpha * td_error * grad_log

    def train(self, num_episodes=1000):
        rewards = []
        for episode in range(num_episodes):
            state, _ = self.env.reset()
            total_reward = 0
            t = 0

            while t < self.t_max:
                action = self.select_action(state)
                next_state, reward, done, *info = self.env.step(action)
                self.update(state, action, reward, next_state, done)
                state = next_state
                total_reward += reward
                if done:
                    break

            rewards.append(total_reward)
            if (episode + 1) % 100 == 0:
                avg = np.mean(rewards[-100:])
                print(f"Episodio {episode+1}, recompensa promedio (últimos 100): {avg:.2f}")
        return rewards

```

```
    def test(self, num_episodes=100):
        rewards = []
        for i in range(num_episodes):
            total_reward = 0.0
            state, _ = self.env.reset()
            for i in range(self.t_max):
                action = self.select_action(state)
                new_state, new_reward, is_done, truncated, _ = self.env.step(action)
                total_reward += new_reward
                if is_done:
                    break
                state = new_state
            rewards.append(total_reward)
        return rewards


    def policy(self):
        policy = np.zeros(self.n_states, dtype=int)
        for s in range(self.n_states):
            action_probs = self.get_action_probs(s)
            policy[s] = np.argmax(action_probs)
        return policy
```

Listing 5: Código fuente del modelo Actor-Critic

A.6 Funciones auxiliares (`utils.py`)

```python
import matplotlib.pyplot as plt
import gymnasium as gym
import seaborn as sns
import pandas as pd
import numpy as np

def revert_state_to_row_col(state):
    row = state // 12
    col = state % 12
    return row,col

def print_policy(policy):
    # Using ASCII characters instead of Unicode arrows
    visual_help = {0: '^', 1: '>', 2: 'v', 3: '<'}
    actual_policy = np.zeros((4, 12)).tolist()
    for i in range(len(policy)):
        row, col = revert_state_to_row_col(i)
        actual_policy[row][col] = visual_help[policy[i]]

    for row in actual_policy:
        print(" | ".join(row))

def draw_rewards(rewards, show=True, path = ""):
    data = pd.DataFrame({'Episode': range(1, len(rewards) + 1), 'Reward': rewards})
    plt.figure(figsize=(10, 6))
    sns.lineplot(x='Episode', y='Reward', data=data)

    plt.title('Rewards Over Episodes')
    plt.xlabel('Episode')
    plt.ylabel('Reward')
    plt.grid(True)
    plt.tight_layout()

    if show:
        plt.show()
    else:
        plt.savefig(path)

def draw_history(history, title):
    plt.figure(figsize=(10, 5))
    plt.plot(moving_average(history), label=f"{title} promedio (ventana=50)")
    plt.xlabel("Episodio")
    plt.ylabel(title)
    plt.title(title)
    plt.grid()
    plt.legend()
    plt.show()

def moving_average(x, w=50):
    return np.convolve(x, np.ones(w)/w, mode='valid')




class RewardWrapperFinal100(gym.RewardWrapper):
    def __init__(self, env):
        super().__init__(env)

    def step(self, action):
        state, reward, done, truncated, info = self.env.step(action)
        reward = reward if state != 47 else 100


        if state in [37, 38, 39, 40, 41, 42, 43, 44, 45, 46]:
```

```
            print(f'DONE: {done}')
            done = True
            print(f'DONE: {done}')

        return state, reward, done, truncated, info

class CustomWrapper(gym.Wrapper):
    def __init__(self, env):
        env.unwrapped.P[47] = {0: [(0.3333333333333333, np.int64(36), -100, False),
    (0.3333333333333333, np.int64(35), -1, False), (0.3333333333333333, np.int64(47),
    0, True)], 1: [(0.3333333333333333, np.int64(35), -1, False), (0.3333333333333333,
    np.int64(47), 0, True), (0.3333333333333333, np.int64(47), 0, True)], 2:
    [(0.3333333333333333, np.int64(47), 0, True), (0.3333333333333333, np.int64(47), 0,
     True), (0.3333333333333333, np.int64(36), -100, False)], 3: [(0.3333333333333333,
    np.int64(47), 0, True), (0.3333333333333333, np.int64(36), -100, False),
    (0.3333333333333333, np.int64(35), -1, False)]}
        super().__init__(env)

    def step(self, action):
        state, reward, is_done, truncated, info = self.env.step(action)

        if state == 47:
            reward = 0
            is_done = True

        return state, reward, is_done, truncated, info
```

Listing 6: Funciones auxiliares utilizadas por los algoritmos