Jugando con algoritmos

Julio Ribas de Novales
dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
julribde@alum.us.es

Felipe Peña Núñez
dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
felpennun@alum.us.es

El objetivo principal de este trabajo es estudiar el comportamiento de distintos algoritmos de aprendizaje por refuerzo. Se busca analizar y comparar el rendimiento de estos algoritmos en diversos entornos, con el fin de entender cómo se comportan y qué factores influyen en su desempeño. A través de este estudio, se pretende obtener conocimientos más profundos sobre los algoritmos de aprendizaje por refuerzo y su aplicabilidad en diferentes contextos.

Tras realizar el análisis estadístico comparativo del rendimiento de los distintos algoritmos de aprendizaje por refuerzo sobre los entornos estudiados, se han obtenido varias conclusiones importantes. Se observó que algunos algoritmos presentaban un rendimiento superior en determinados tipos de entornos, mientras que otros se destacaban en diferentes escenarios. Esto sugiere que no existe un algoritmo universalmente óptimo, sino que la elección del algoritmo adecuado depende del entorno específico y las características del problema. Además, se pudo comprobar que la incorporación de un nuevo entorno y algoritmo al análisis comparativo enriqueció la comprensión de los resultados y proporcionó perspectivas adicionales.

Palabras Clave—Inteligencia Artificial, otras palabras clave…

I. INTRODUCCIÓN

La inteligencia artificial ha experimentado avances significativos en los últimos años, y el aprendizaje por refuerzo se ha convertido en una técnica fundamental para permitir a los agentes autónomos aprender a tomar decisiones en entornos complejos y dinámicos. A diferencia de otros enfoques de aprendizaje automático, donde los agentes se basan en datos previamente etiquetados, el aprendizaje por refuerzo se centra en la interacción directa con el entorno, donde los agentes aprenden a través de la prueba y error, y a partir de las señales de recompensa que reciben.

El aprendizaje por refuerzo se inspira en cómo los seres vivos aprenden a través de la retroalimentación y el refuerzo positivo. Un agente de aprendizaje por refuerzo aprende a través de la exploración activa del entorno, tomando acciones y observando las consecuencias de esas acciones. El objetivo es maximizar una señal de recompensa numérica, que indica qué tan bien está realizando sus acciones en función de sus objetivos. A medida que el agente interactúa repetidamente con el entorno, va ajustando su comportamiento para maximizar las recompensas obtenidas.

En este contexto, surge la necesidad de comprender en profundidad el comportamiento de los algoritmos de aprendizaje por refuerzo. Si bien se han desarrollado diversos algoritmos en este campo, su desempeño puede variar significativamente según las características del entorno en el que se aplican. Por lo tanto, es esencial llevar a cabo un análisis comparativo del rendimiento de estos algoritmos en diferentes escenarios para identificar sus fortalezas y limitaciones.

Este trabajo se enmarca en ese contexto, con el objetivo principal de estudiar el comportamiento de distintos algoritmos de aprendizaje por refuerzo. Se busca analizar su rendimiento en una variedad de entornos y comprender cómo influyen diversos factores en su desempeño. Además, se propone implementar un nuevo entorno y algoritmo de aprendizaje por refuerzo para enriquecer aún más el análisis comparativo.

El conocimiento obtenido a través de este estudio puede tener aplicaciones significativas en diversas áreas, como la robótica, la optimización de recursos, el control de procesos, entre otros. Al comprender mejor los algoritmos de aprendizaje por refuerzo y su adaptabilidad a diferentes entornos, podremos desarrollar sistemas más inteligentes y eficientes capaces de aprender y tomar decisiones autónomas.

El objetivo principal de este trabajo es estudiar y comparar el rendimiento de los algoritmos Q-learning y Montecarlo en entornos de aprendizaje por refuerzo ayudándonos con la librería gymnaium, esta aporta entornos de ya creados que usaremos además de guías para crear el tuyo propio. Se busca analizar el tiempo de ejecución, la recompensa y la longitud medias de los episodios generados por cada algoritmo, así como evaluar las políticas resultantes. Además, se propone implementar y evaluar un nuevo entorno llamado Golf. También se considera la implementación del algoritmo Sarsa y su comparación con los algoritmos existentes en los tres entornos.

El presente documento se descompone en diversos puntos los cuales facilitan la organización y entendimiento del mismo.

Entre ellos encontramos:

- Introducción. Se comenta contexto en el que se desarrolla el trabajo presentado, del mismo modo indica la problemática o el objetivo que se marca, y cómo se ha enfocado la solución propuesta.

- Preliminares. Se hace una breve introducción de las técnicas empleadas y también trabajos relacionados, si los hay.
- Metodología. Trata acerca de la descripción del método implementado en el trabajo. Esta parte es la correspondiente a lo realmente desarrollado en el trabajo.
- Resultados. En esta sección se detallará tanto los experimentos realizados como los resultados conseguidos.
- Conclusiones. Se comparten las reflexiones alcanzadas una vez terminado el trabajo.

II. Preliminares

A. *Métodos empleados*

En este trabajo se han empleado diferentes métodos y técnicas para abordar el estudio y comparación de los algoritmos de aprendizaje por refuerzo.

- Q-learning: es una técnica de aprendizaje por refuerzo utilizada en aprendizaje automático. El objetivo de este es aprender una serie de normas que le diga a un agente qué acción tomar bajo qué circunstancias. No requiere un modelo del entorno y puede manejar problemas con transiciones estocásticas y recompensas sin requerir adaptaciones. Para cualquier proceso de decisión de Markov finito (PDMF) (finite Markov decision process en inglés), Q-learning encuentra una política óptima en el sentido de que maximiza el valor esperado de la recompensa total sobre todos los pasos sucesivos, empezando desde el estado actual.1 Q-learning puede identificar una norma de acción-selección óptima para cualquier PDMF, dado un tiempo de exploración infinito y una norma parcialmente aleatoria1 "Q" nombra la función que devuelve la recompensa que proporciona el refuerzo y representa la "calidad" de una acción tomada en un estado dado.[3]
- Montecarlo: el algoritmo de aprendizaje de Monte-Carlo nos sirve para estimar la función de valor, toma muestras directamente de episodios realizados por un agente dada la política que elegimos, funciona mejor para ambientes que tienen las características de juegos, es decir, que siempre tienen un final. [4]
- Sarsa: es una ligera variación del popular algoritmo Q-learning, mencionado anteriormente. Al contrario que este, la técnica SARSA, es una política On y utiliza la acción realizada por la política actual para conocer el valor Q. [5]

B. *Trabajo Relacionado*

Para entender algo más acerca del aprendizaje por refuerzo aplicado a videojuegos encontramos un proyecto bastante interesante el cual comparte cierto parecido al nuestro, además de consultar exhaustivamente la documentación de la librería gymnasium:

- Comparación de técnicas de aprendizaje por refuerzo jugando a un videojuego [1]
- Documentación librería gymnasium [6]

III. Metodología

Para llevar a cabo nuestro trabajo hemos realizado una serie de métodos agrupados en distintas clases como comentaremos a continuación. Obviaremos los métodos ya proporcionados por los profesores en la clase “aprendizaje_por_refuerzo.py”, ya que nos ayudamos de estos para crear los mencionados anteriormente.

Grosso modo, podemos dividir las clases del proyecto en tres grupos: algoritmos de entrenamiento, aplicación de los algoritmos sobre los distintos entornos, herramientas de representación gráfica y por último creación de nuevos entornos (golf). De esta manera nos aseguramos una cierta modularidad en todo el conglomerado de clases, pudiendo, en caso de errores acotar fácilmente su localización, además de poder entrenar y observar los resultados de los distintos entornos por separado.

Así mismo, se presentan a continuación los distintos métodos agrupados según correspondencia:

- Algoritmos de entrenamiento: con esto nos referimos a funciones que engloben tanto al método de aprendizaje por refuerzo (proporcionado a partir de la clase “aprendizaje-por-refuerzo.py”) como a todo lo que necesiten para que sus resultados puedan ser representados e interpretados gráficamente.

  En el proyecto encontramos la clase “algoritmos.py” referente a este grupo.

  Qlearning/SARSA/Montecarlo (env,p,n_ep)

  Entrada: un entorno env, política p (solo para Qlearning y SARSA) y número de episodios n_ep

  Salida: tiempo de ejecución del entrenamiento, gráficos relacionados con las recompensas los pasos por episodio obtenidos.

  1. factor_de_descuento ← 0.95
  2. tasa_de_aprendizaje ← 0.1
  3. env.reset()
  4. algoritmo← Q_Learning/Montecarlo/SARSA(env, factor_de_descuento, tasa_de_aprendizaje, politica)

ENTRENAMIENTO ---

5. inicio ← tiempo_actual()

6. algoritmo.entrena(número_episodios)

7. fin ← tiempo_actual()

ENTRENAMIENTO ---

8. imprimir("Tiempo de ejecución: ", fin - inicio)

9. recompensas ← convertir_a_tupla(env.return_queue)

10. episodios ← convertir_a_tupla(env.length_queue)

11. tabla_recompensas ← []

12. tabla_episodios ← []

13. longitud_acumulada ← 0

16. para e en rango(longitud(recompensas)) hacer

16.1. tabla_recompensas.agregar(recompensas[e][0])

17. para e en rango(longitud(episodios)) hacer

17.1. tabla_episodios.agregar(episodios[e][0])

17.2. longitud_acumulada ← longitud_acumulada + episodios[e][0]

18. gráficos.plot_recompensa_acumulada(tabla_recompensas, número_episodios)

19. gráficos.plot_longitud_episodios(tabla_episodios, longitud_acumulada)

- Aplicación de los algoritmos sobre los distintos entornos: consisten en clases en la cuales se prepara el entorno para ser entrenado y posteriormente llamar a los métodos de la clase explicada anteriormente para obtener resultados. Estas clases no tienen métodos propios como tal sino que simplemente se encargan de crear el entorno mediante el función gym.make() que proporciona la librería gymnasium y así posteriormente usar la envoltura RecordEpisodeStatistics comentada en el documento enunciado del trabajo para obtener datos como la recompensa y longitud media de los episodios. Al ejecutar estas clases se devuelven los resultados obtenidos de someter al entorno en cuestión a los distintos algoritmos de entrenamiento con las distintas políticas cuando corresponda.

  En el proyecto encontramos referente a este grupo las clases “frozen_lake.py”, “taxi.py” y “golf.py”.

- Herramientas de representación gráfica: en este grupo encontramos una única clase con dos métodos para la representación de gráficos gracias a la librería matplotlib. La clase en cuestión es “gráficos.py”.

  plot_recompensa_acumulada (rec)

  Entrada: objetos tipo array con las recompensas obtenidas en cada episodio (rec).

  Salida: gráfico que representa los episodios en el eje x y las recompensas acumuladas en el eje y.

  1. plt.plot(np.cumsum(rec))

  2. media_valores ← np.mean(rec)

  3. titulo ← f'Recompensa acumulada (Media por episodio: {media_valores:.2f})'4. plt.title(titulo)

  5. plt.xlabel("Episodios")

  6. plt.ylabel("Recompensas")

  7. plt.show()

  plot_longitud_episodios (pasos)

  Entrada: objeto tipo array con los pasos realizados en cada episodio.

  Salida: gráfico que representa los episodios en el eje x y los pasos tomados en el eje y.

  1. plt.plot(pasos)

  2. media_valores ← np.mean(pasos)

  3. titulo ← f'Longitud de los episodios (Media por episodio: {media_valores:.2f})'

  4. plt.title(titulo)

  5. plt.xlabel("Episodios")

  6. plt.ylabel("Pasos")

  7. plt.show()

- Creación de nuevos entornos: al igual que los entornos frozen lake y taxi ya estaban implementados por la propia librería, había un tercero el cual teníamos que hacer de cero, para ello nos ayudamos de la documentación de gymnasium. [6]

  En este caso se trata de la clase "golf_env.py", la cual aviso de antemano que tras intentarlo con ansia no conseguimos dar con la tecla para hacer que al ejecutar la clase "golf.py" nos diera resultados. Por lo que no veo conveniente compartir aquí el pseudocódigo correspondiente a los métodos "init(self)", "reset(self, seed, options)" y "step(self, action)" que componen la clase. Del mismo modo se pueden ver en el proyecto adjunto en la entrega.

IV. Resultados

Para tratar el tema de los resultados vamos a dividir el contenido en tres apartados relacionados con los tres entornos sobre los cuales hemos tenido que evaluar los distintos algoritmos. Recordar que sobre los algoritmos a evaluar son Qlearning y SARSA con políticas voraz y épsilon voraz y Montecarlo únicamente con política voraz. Comenzamos por Frozen Lake.

- **Frozen Lake:** al ejecutar la clase "frozen_lake.py" descrita anteriormente se nos representa para cada algoritmo evaluado el tiempo que ha tardado en ejecutarse y dos gráficas que representan la recompensa acumulada y los pasos realizados a lo largo de los episodios.

  Si comparamos primero los tiempos de ejecución, estos se muestran bastante parecidos en todas las variantes de los algoritmos Qlearning y SARSA lo cual tiene sentido ya que ambos utilizan una tabla para almacenar y actualizar los valores de acción (Q-values) que representan la utilidad esperada de tomar una acción en un estado determinado. Ambos algoritmos también siguen un proceso iterativo de exploración y explotación, donde se toman acciones de acuerdo con una política específica para equilibrar la exploración del entorno y la explotación de las acciones conocidas.

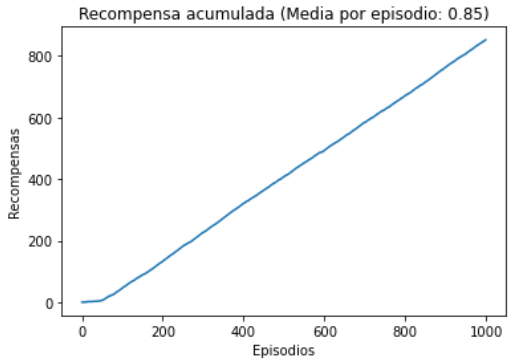
  Estas similitudes en la estructura y enfoque de los algoritmos pueden llevar a resultados parecidos en términos de la calidad de la política aprendida y las recompensas acumuladas en el entorno.

  Sin embargo, es importante destacar que en entornos más complejos o con configuraciones específicas, las diferencias entre los algoritmos pueden ser más evidentes.

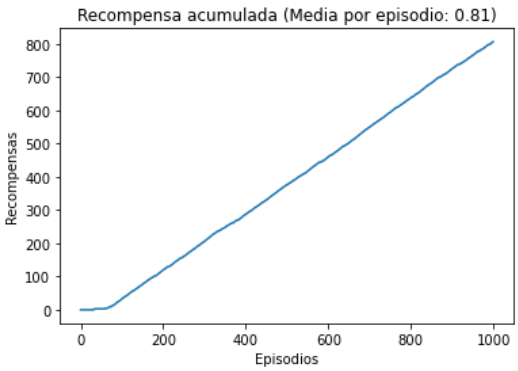
  Montecarlo en cambio sí que tarda bastante más. Generalmente, este algoritmo tiende a requerir más tiempo de ejecución que Qlearning debido a su naturaleza basada en simulaciones completas de episodios. En cada episodio, Montecarlo necesita realizar un número fijo de iteraciones completas antes de actualizar los valores de la función de valor. Esto puede ser costoso computacionalmente, especialmente cuando se trata de entornos con un gran número de estados o cuando los episodios son largos.

  Por otro lado, Qlearning es un algoritmo de aprendizaje por refuerzo basado en valores de acción, y generalmente tiene un tiempo de ejecución más rápido que Montecarlo. Esto se debe a que Qlearning actualiza los valores de acción de forma incremental a medida que interactúa con el entorno, sin necesidad de esperar hasta el final de un episodio completo.

  Fijándonos en los gráficos obtenidos al igual que ocurre con los tiempos de ejecución los resultados por parte de Qlearning y SARSA son bastante similares, tanto en la recompensa acumulada como en la longitud de los episodios. Véase la similitud en las figuras 1 y 2.



1. Figura 1: Qlearning política épsilon voraz



2. Figura 2: SARSA política épsilon voraz

En cambio el algoritmo de Montecarlo no refleja ningún resultado interesante, es más, las recompensas obtenidas a partir de este son siempre nulas.

Pasando al tema de la longitud de episodios hay un factor muy interesante y consiste en la diferencia que marca la elección de una política u otra en el resultado obtenido.

Para ponernos en contexto, La diferencia entre una política voraz (greedy) y una política épsilon voraz (epsilon-greedy) radica en cómo eligen las acciones en un algoritmo de aprendizaje por refuerzo.

Política Voraz (Greedy):

Una política voraz selecciona siempre la acción con el valor de acción más alto según los valores de acción o estado-valor estimados. Es decir, elige la acción que se considera óptima en función de la información disponible en ese momento. En otras palabras, la política voraz busca la recompensa inmediata más alta sin preocuparse por la exploración adicional.

La política voraz es simple y eficiente, pero puede ser muy explotadora. Esto significa que puede quedar atrapada en una acción subóptima si se encuentra en un máximo local y no explora suficientemente otras acciones para descubrir acciones de mayor recompensa a largo plazo.

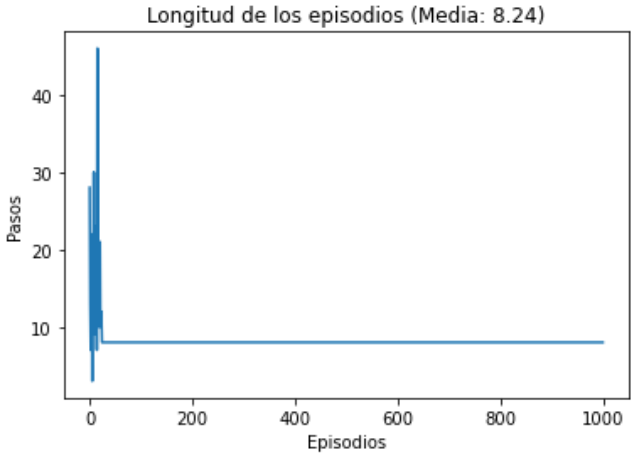
Política Épsilon Voraz (Epsilon-Greedy):

La política épsilon voraz combina la exploración y la explotación al permitir una cierta cantidad de exploración aleatoria incluso cuando se considera la acción óptima. En lugar de elegir siempre la mejor acción, la política épsilon voraz selecciona una acción aleatoria con probabilidad épsilon (epsilon) y la mejor acción con probabilidad 1 - épsilon.

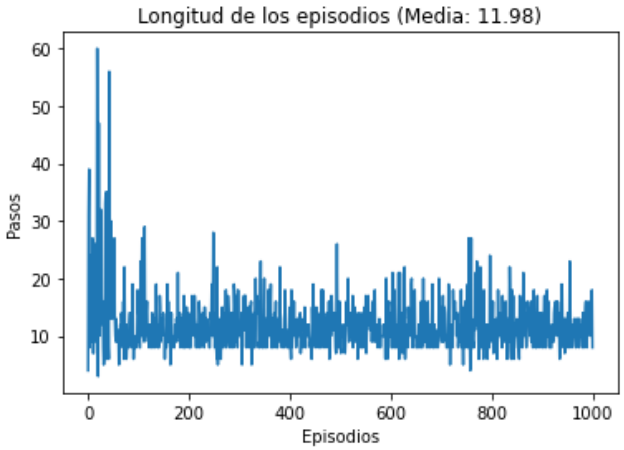
La probabilidad épsilon determina el nivel de exploración: cuanto mayor sea el valor de épsilon, mayor será la probabilidad de realizar una acción aleatoria. Esta exploración aleatoria permite a la política épsilon voraz explorar nuevas acciones y evitar quedar atrapada en máximos locales subóptimos.
La política épsilon voraz es más flexible y equilibra la explotación de acciones conocidas con la exploración de nuevas acciones. Permite descubrir acciones potencialmente mejores y, a largo plazo, puede conducir a mejores resultados.

En resumen, la principal diferencia entre una política voraz y una política épsilon voraz radica en la cantidad de exploración que permiten. La política voraz elige siempre la mejor acción conocida, mientras que la política épsilon voraz incluye una cantidad de exploración aleatoria para descubrir acciones potencialmente mejores y evitar quedar atrapada en máximos locales subóptimos.

¿A qué queremos llegar con esto?, como se puede observar en las figuras 3 y 4, usando en este caso el Método de Qlearning como ejemplo (el comportamiento en los otros dos es similar) vemos como en el entrenamiento con política voraz el número de pasos se mantiene al llegar a un cierto punto ya que esta política como hemos mencionado antes busca la recompensa más inmediata por lo que deja de buscar alternativas mientras que en la épsilon voraz siempre se guarda un pequeño porcentaje de aleatoriedad para seguir explorando en lugar de explotar un único camino.



3. Figura 3: política voraz



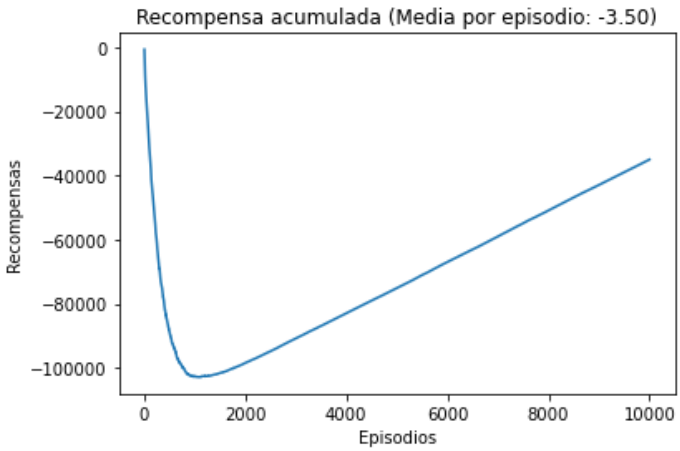
4. Figura 4: política épsilon voraz

- **Taxi:** en este caso vamos a ir más al grano ya que se han dejado claro bastantes conceptos generales

en el apartado anterior (Frozen Lake). Al igual que antes, tras ejecutar el módulo “taxi.py” se obtienen los resultados de entrenar el entorno con los distintos algoritmos.

Montecarlo sigue igual que antes aunque en este caso los métodos Qlearning y SARSA presentan resultados bastante interesantes. Vemos en la figura 5, representativa de la aplicación de SARSA al entorno ‘taxi’ como al inicio decrece la función hasta llegar a un punto a partir del cual asciende lentamente. Esto se debe a que en este entorno existe la recompensa negativa, por lo que hasta que el agente no ha explorado lo suficiente y da con la tecla de lo que ocurre no empieza a obtener recompensas positivas, mientras tanto el taxi sigue en movimiento lo cual hace que la recompensa acumulada decrezca.

Cabe recalcar que no adjunto imágenes con todas las gráficas resultantes por la absurda cantidad de tamaño que ocuparía en el documento y del mismo modo porque se pueden extraer las mismas conclusiones mostrando solo los resultados adecuados.



5. Figura 5: SARSA política voraz

El resto de métricas son bastante similares conceptualmente hablando a lo presentado en ‘Frozen Lake’, tanto en tiempo de ejecución como en longitud de los episodios.

- **Golf:** como comentamos antes, al ejecutar la clase “golf.py” no devuelve resultados, sabemos que el fallo se encuentra en el módulo de implementación del entorno “golf_env.py” pero desconocemos de qué trata ya que intentamos seguir las instrucciones de la documentación de gymnasium.

V. Conclusiones

A la hora de afrontar el trabajo, al principio nos dio mucho respeto debido a la falta de entendimiento que teníamos acerca de la aplicación de métodos de entrenamiento a entornos proporcionados. A base de darnos de golpes contra la documentación de gymnasium logramos empezar a entender conceptos y maneras de aplicarlos. Todo nuestro conocimiento del tema está reflejado en este documento, y aunque hay cosas que no están completadas como por ejemplo los resultados del entorno ‘golf’ no creo que nos haga justicia ya que hemos entendido como crear un entorno personalizado desde cero y así queda demostrado en el código del proyecto, aunque habrá algún fallo que no detectamos.

Aún así, decir que ha sido una experiencia totalmente nueva para nosotros hacer un trabajo de investigación de este tipo y aunque no queramos volver a hacer otro en un periodo razonable de tiempo debido al agobio que hemos sufrido estos últimos días, en un futuro no nos importaría.

Referencias

[1] Comparación de técnicas de aprendizaje por refuerzo jugando a un videojuego de tenis. https://oa.upm.es/55888/1/TFM_PABLO_SAN_JOSE_BARRIOS.pdf

[2] Página web del curso IA de Ingeniería del Software. https://www.cs.us.es/cursos/iais.

[3] Página de wikipedia acerca del algoritmo de q-learning en el aprendizaje por refuerzo. https://es.wikipedia.org/wiki/Q-learning

[4] Aprendizaje por Refuerzo: Predicción Libre de Modelo. https://medium.com/aprendizaje-por-refuerzo-introducci%C3%B3n-al-mundo-del/aprendizaje-por-refuerzo-predicciones-sin-modelo-45e66528aa98

[5] Aprendizaje por refuerzo SARSA. Aprendizaje por refuerzo SARSA – Barcelona Geeks

[6] Documentación gymnasium. Basic Usage - Gymnasium Documentation - The Farama Foundation