Practica 2: Aprendizaje en entornos complejos

Alejandro López Cuéllar, Miguel Ángel Vera Frutos, Alejandro Belda Fernández

9 de marzo de 2025

1. Introducción

En la práctica anterior realizamos un experimento relacionado con el problema del **bandido de k-brazos**. Este experimento es útil para conocer un problema real y que inclusive se trabaja a día de hoy en su implementación, mejoría y soluciones. Con ello también conocimos conceptos básicos del aprendizaje por refuerzo como pueden ser el concepto de recompensa, exploración o explotación. Una vez aprendido esto podemos pasar a problemas en los que haya un cierto contexto que no se nos da en el problema del **bandido de k-brazos**. En este tipo de problemas los entornos no suelen ser fijos, por ello hay múltiples opciones para tratar estos problemas; sin embargo, una de las más efectivas es mediante la resolución de las ecuaciones de *Bellman* que pueden ser resueltas mediante el cálculo matricial o el cálculo iterativo, que serán implementados por los algoritmos que realizaremos.
Para poder realizar estas pruebas, trabajaremos con la librería *Gymnasium* la cual nos otorga múltiples entornos de diferente complejidad para poder experimentar y probar con ellos; así como realizar los algoritmos pertinentes que resuelvan dichos entornos.
Entonces los objetivos de la práctica son los siguientes:

- Familiarizarnos y trabajar con los entornos de *Gymnasium*
- Estudiar algoritmos básicos que resuelvan estos problemas
- Realizar un estudio acerca de estos experimentos y pruebas

Finalizado este apartado comentar que abarcaremos los apartados de desarrollo en el que se abarcará una definición formal del problema, un apartado de algoritmos en los que explicaremos los algoritmos usados, un siguiente apartado en que estudiaremos y analizaremos los resultados obtenidos del experimento y finalmente un último apartado incluyendo las conclusiones.

2. Desarrollo

Los entornos realizados para nuestra práctica han sido: *Taxi-v3, MountainCar-v0, LunarLander-v3*. Una definición formal para cada uno de estos problemas sería la siguiente:

2.1. Taxi

2.1.1. Descripción

Este entorno se basa en una cuadrícula 5x5 con 4 ubicaciones designadas para la recogida y entrega de un cliente. El taxi comenzará en alguna casilla aleatoria y el cliente en una casilla aleatoria entre las 4 ubicaciones designadas. El objetivo del entorno es que el taxi recoja al cliente y lo deje en la casilla designada como su destino.

2.1.2. Espacio de Acciones

- 0: Mover hacia el sur (abajo)
- 1: Mover hacia el norte (arriba)
- 2: Mover hacia el este (derecha)
- 3: Mover hacia el oeste (izquierda)
- 4: Recoger al pasajero
- 5: Dejar al pasajero

2.1.3. Espacio de Observación

Existen 500 estados discretos, obtenidos como resultado de 25 posibles posiciones del taxi, 5 del pasajero(si esta en el taxi también cuenta) y 4 posibles localizaciones de recogida y destino. Es importante notar que hay 400 estados que realmente se pueden alcanzar durante un episodio. Los estados faltantes corresponden a situaciones en las que el pasajero está en la misma ubicación que su destino, lo que típicamente señala el final del episodio. Cuatro estados adicionales pueden observarse justo después de un episodio exitoso, cuando tanto el pasajero como el taxi están en el destino. Esto da un total de 404 estados discretos alcanzables.

$$\text{estado} = ((\text{taxi\_row} \times 5 + \text{taxi\_col}) \times 5 + \text{passenger\_location}) \times 4 +$$

2.1.4. Estado Inicial

El estado inicial se selecciona de los estados posibles donde el pasajero no está ni en su destino ni dentro del taxi. Hay 300 estados iniciales posibles: 25 posiciones del taxi, 4 ubicaciones del pasajero (excluyendo dentro del taxi) y 3 destinos (excluyendo la ubicación actual del pasajero).

2.1.5. Recompensas

- -1 por cada paso, a menos que se active otra recompensa.
- +20 al dejar al pasajero.
- -10 al ejecutar ilegalmente las acciones de recoger y dejar.

2.1.6. Fin del Episodio

- **Terminación**: El taxi deja al pasajero en su destino.
- **Truncamiento**: La longitud del episodio es 200.

2.1.7. Métodos utilizados

Como el espacio es discreto no es necesario utilizar ningún tipo de discretización. Los algoritmos utilizados han sido:

- Monte Carlo On Policy

- Monte Carlo Off Policy
- Sarsa
- Sarsa Semi Gradiente
- Q-Learning

Añadir que no hemos realizado una implementación de métodos de aproximación por redes neuronales porque creemos que el problema a nivel computacional no es demasiado complejo, lo que creemos que hace innecesario la utilización de ese tipo de algoritmos aquí.

2.2. Mountain Car

2.2.1. Descripción

El Mountain Car es un entorno en el que consiste en un coche colocado de forma estocástica en el fondo de un valle, con las únicas acciones posibles siendo las aceleraciones que se pueden aplicar al coche en cualquiera de las dos direcciones. El objetivo del agente es acelerar estratégicamente el coche para alcanzar el estado objetivo en la cima de la colina derecha. De las dos versiones del Gymnasiun de este problema usaremos la que tiene el espacio de acciones discreto.

2.2.2. Espacio de Acciones

- 0: Acelerar a la izquierda
- 1: No hacer nada
- 2: Acelerar a la derecha

2.2.3. Espacio de Observaciones

Índice	Descripción
0	Posición del coche (valor flotante entre -1.2 y 0.6)
1	Velocidad del coche (valor flotante entre -0.07 y 0.07)

En nuestra representación por tiling hemos usado 20 intervalos en cada dimensión con un número de mosaicos de 10 para realizar la discretización. Esto nos otorga un espacio de observación de 4000.

Tabla 1: Espacio de observación del problema de Mountain Car.

2.2.4. Dinámicas de transición

$$\begin{aligned}\text{velocidad}_{t+1} &= \text{velocidad}_t + (\text{acción} - 1) \cdot \text{fuerza} \\ &\quad - \cos(3 \cdot \text{posición}_t) \cdot \text{gravedad} \\ \text{posición}_{t+1} &= \text{posición}_t + \text{velocidad}_{t+1}\end{aligned}$$

2.2.5. Estado Inicial

La posición del automóvil se asigna con un valor aleatorio uniforme en el rango $[-0{,}6, -0{,}4]$. La velocidad inicial del automóvil siempre se asigna como 0.

2.2.6. Recompensas

- -1 hasta el final del episodio.

2.2.7. Fin del episodio

- **Terminación**: El coche alcanza la meta.
- **Truncamiento**: El número de pasos del episodio alcanza los 200.

2.2.8. Métodos utilizados

- Monte Carlo Off Policy
- Monte Carlo On Policy
- Sarsa
- Sarsa Semi Gradiente
- Q-Learning
- Deep Q Learning

Comentar que se usó el algoritmo *Tiling* para realizar la discretización. También decir que no implementamos Deep Sarsa por la gran cantidad de algoritmos implementados en el ejercicio. Pensamos que se podría hacer demasiado pesado añadir otro algoritmo de tanta complejidad aquí también.

2.3. Lunar Lander

2.3.1. Descripción

Este entorno es un problema clásico de optimización de trayectoria de cohetes. Según el principio del máximo de Pontryagin, es óptimo encender el motor a máxima potencia o apagarlo. Esta es la razón por la que este entorno tiene acciones discretas: motor encendido o apagado.

La plataforma de aterrizaje siempre está en las coordenadas $(0, 0)$. Las coordenadas son los dos primeros números en el vector de estado. Es posible aterrizar fuera de la plataforma de aterrizaje. El combustible es infinito, por lo que un agente puede aprender a volar y luego aterrizar en su primer intento.

2.3.2. Espacio de Acciones

- 0 No hacer nada
- 1 Activa el motor izquierdo.
- 2 Activa el motor principal.
- 3 Activa el motor derecho.

2.3.3. Espacio de Observación

El espacio de observación es un vector de 8 dimensiones que incluye las coordenadas del módulo lunar en x e y, sus velocidades lineales en x e y, su ángulo, su velocidad angular y dos valores booleanos que indican si cada pata está en contacto con el suelo o no.

2.3.4. Recompensas

Se otorga una recompensa después de cada paso. La recompensa total de un episodio es la suma de todas las recompensas obtenidas en cada paso dentro de ese episodio.

- Aumenta o disminuye dependiendo de qué tan cerca o lejos esté el módulo lunar de la plataforma de aterrizaje.
- Aumenta o disminuye dependiendo de la velocidad del módulo lunar (más lenta es mejor).
- Disminuye si el módulo lunar está inclinado (el ángulo no es horizontal).
- Aumenta en 10 puntos por cada pata que esté en contacto con el suelo.
- Disminuye en 0.03 puntos por cada fotograma en que un motor lateral esté encendido.
- Disminuye en 0.3 puntos por cada fotograma en que el

motor principal esté encendido.
- El episodio recibe una recompensa adicional de -100 o +100 puntos en caso de que el módulo lunar se estrelle o aterrice con éxito, respectivamente.

2.3.5. Estado Inicial

El módulo lunar comienza en la parte superior central de la pantalla con una fuerza inicial aleatoria aplicada a su centro de masa.

2.3.6. Fin del episodio

- **Terminación**: Se aterriza correctamente
- **Impacto**: La nave sufre una colisión fuerte contra el suelo
- **Salir del mapa**: La nave abandona los límites del mapa
- **Nave muerta**: La nave no se mueve y tampoco está en contacto con nada.

2.3.7. Métodos utilizados

- Deep Q Learning
- Deep Sarsa

Comentar aquí algunas cosas. No se realiza discretizacion en este problema y en su lugar se trabaja con tensores para representar los estados. Esto porque como sabemos, el problema tiene un espacio continuo de 8 variables, por lo que realizar una discretizacion por *tilings* tendria el problema de que el espacio de observación sería enorme. Por ejemplo, supongamos un número de intervalos de 10 para cada dimensión y además un número de mosaicos de 10. Esto nos daría un espacio de observación discretizado de

$$10^9$$

, por lo que sería muy ineficiente para gestionar la Q. Esto también descarta usar Salsa Semi Gradiente con tiling, pero aún podríamos usarlo mediante una aproximación lineal usando los pesos y calculando la Q como el producto punto de los pesos por el vector de estados. Esto nos dejaría un weights de tamaño

$$8x4$$

. Sobre el papel esto es posible, pero aun con esas el problema es demasiado complejo y hay que tener en cuenta demasiados factores, los cuales son demasiados para este algoritmo. Esto trae como resultado que pese a que en algunos episodios sí que llega a completar el problema de promedio no aprende nada y la política final no es óptima ni efectiva, esto se debe a que la forma de actualizarse y de aprender de este algoritmo es bastante menos compleja que la que usan los otros métodos, lo que hace que se adapten mejor a entornos de mayor complejidad. Por ende la conclusión que hemos podido extraer es que al menos en nuestras pruebas no es una alternativa usarlo, aunque creemos que con una optimización de parámetros y un número de episodios bastante alto(posiblemente entrando en los millones), sí que quizás lograra una buena política; pero en ese punto la implementación por redes neuronales es simplemente más efectiva y más visualizando que la red usada es relativamente sencilla.

3. Algoritmos

Ahora explicaremos los algoritmos utilizados para el aprendizajes de los agentes a lo largo de los entornos utilizados en Gymnasium:

3.1. Monte Carlo Off Policy

En este algoritmo, conseguimos estimar la función de valor de una política objetivo π utilizando experiencias generadas a partir de una política de comportamiento diferente b. Además, se basa en la reponderación de las muestras utilizando importancia ponderada (*importante sampling*).

- Primero, realizaremos la inicialización del algoritmo, para ello, definiremos la politica objetivo π, la politica de comportamiento b. Además, también generaremos una tabla de valores de acción $Q(s, a)$ que se inicializa arbitrariamente y un contador de pesos $C(s, a)$ inicializado a 0.
- Generaremos un episodio siguiendo la política de comportamiento b y registraremos la secuiencia de estados, acciones y recompensas.
- Ahora se procesa el episodio al revés (*backward update*). Para ello, inicializamos un retorno acumulado G y un factor de ponderación W. Luego, para cada paso de t del episodio, desde el último hasta el primero actualizamos el retorno, el contador de pesos y el valor de acción, seguidamente, si la acción tomada no pertenece a la política objetivo, salimos del bucle, en otro caso, actualizamos el factor de ponderación.
- Repetir hasta cierto número de episodios.

Además añadir que este algoritmo es una buena opción dada su política off-policy. Ya que diferencia de los métodos on-policy; Monte Carlo Off-Policy permite aprender de experiencias generadas por políticas diferentes a la política que está siendo aprendida. Esto lo hace especialmente útil cuando se tiene una política exploratoria fija, mientras se busca una política más óptima. Además, este enfoque no requiere actualizaciones a través de cada paso del episodio, sino que espera hasta que el episodio termine, lo cuál puede ser una opción interesante de evaluar, al igual que la versión on policy.

3.2. Monte Carlo On Policy

En este caso, este algoritmo utiliza la misma política tanto para tomar decisiones como para realizar las actualizaciones, es decir, la política de comportamiento y política objetivo son la misma.

En este caso, se ha modificado el algoritmo ya que el entorno del Taxi tiene un espacio de observación y de acción pequeño, y el control de visitas hacia que aprendiese más lento el agente, entorpeciendo el aprendizaje.

- Primero de todo, primero inicializamos la tabla de valores $Q(s, a)$ de forma arbitraria.
- Gerenamos un episodio, donde obtenemos el conjuntos de estados, acciones y recompensas a partir de la política π que es ejecutada usando la estrategia ϵ-greedy.
- En cada paso hacia atrás del episodio, se calcula el re-

Algorithm 1 Monte Carlo Off-Policy

Inicializar: Tabla de valores $Q(s, a)$ arbitrariamente
Contador de pesos $C(s, a) \leftarrow 0$ para todo s, a
Definir política objetivo π y política de comportamiento b
for cada episodio **do**
  Generar un episodio $(s_0, a_0, r_1, s_1, ..., s_T)$ siguiendo b
  $G \leftarrow 0$ (retorno acumulado)
  $W \leftarrow 1$ (factor de ponderación)
  for $t = T$ hasta 0 (de atrás hacia adelante) **do**
    $G \leftarrow r_{t+1} + \gamma G$
    $C(s_t, a_t) \leftarrow C(s_t, a_t) + W$
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \frac{W}{C(s_t, a_t)} \cdot (G - Q(s_t, a_t))$
    if $\pi(a_t|s_t) = 0$ **then**
      break
    end if
    $W \leftarrow W \times \frac{\pi(a_t|s_t)}{b(a_t|s_t)}$
  end for
end for

torno acumulado G.
- Se actualizan los valores de $Q(s, a)$ de manera incremental, sin controlar el número de veces que se ha visitado cada par estado-acción.
- Repetir hasta cierto número de episodios.

Este algoritmo es útil porque a diferencia de otros métodos, en los que la actualización de la política ocurre durante el proceso de toma de decisiones, Monte Carlo On-Policy espera a que el episodio termine para actualizar la política, lo que le permite evaluar el retorno completo de las acciones realizadas. Además, una ventaja clave de este algoritmo es la posibilidad de volver a visitar episodios previos para refinar las estimaciones de recompensa y ajustar las políticas. Esto le permite al agente mejorar continuamente su estrategia, maximizando las recompensas a largo plazo.

Algorithm 2 Monte Carlo On-Policy (sin control de visitas)

Inicializar: Tabla de valores $Q(s, a)$ arbitrariamente
Definir política inicial π con estrategia ϵ-greedy
for cada episodio **do**
  Generar un episodio $(s_0, a_0, r_1, s_1, ..., s_T)$ siguiendo π
  $G \leftarrow 0$ (retorno acumulado)
  for $t = T$ hasta 0 (de atrás hacia adelante) **do**
    Obtener (s_t, a_t, r_{t+1}) del episodio
    $G \leftarrow r_{t+1} + \gamma G$
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \frac{1}{N}(G - Q(s_t, a_t))$
  end for
end for

3.3. Q-Learning

Este tipo de algoritmo basado en valor se utiliza para encontrar la mejor acción a tomar en cada estado de un entorno. La idea es aprender una función de valor de acción, denominada Q, que nos dice qué tan buena es una acción en un estado determinado.

- Primero, inicializamos la tabla de valores Q para todos los estados s y acciones a posibles. También, será necesario establecer una tase de aprendizaje α, un factor de descuento γ y la política de exploración (como en nuestro caso ϵ-greedy.
- Para cada episodio, el agente empieza en un estado inicial s, entonces, elige una accion a con nuestra política. Ejecutamos la acción a y observamos la recompensa r y el siguiente estado s'. Actualizamos el valor de la tabla Q de acuerdo a la fórmula de actualización de Q-Learning (mostrada en el pseudocodigo)
- Repetir hasta cierto número de episodios.

Añadir que *Q-Learning* es un algoritmo excelente dada su capacidad para aprender una política óptima de acción en un entorno desconocido, sin necesidad de un modelo explícito del sistema. Ademas se adapta a entornos dinámicos donde no se dispone de información completa sobre el sistema, y su convergencia a una política óptima en entornos discretos o discretizados lo hace robusto y efectivo.

Algorithm 3 Q-Learning

Inicializar: Tabla de valores $Q(s, a) = 0$ para todos los s y a
Definir tasa de aprendizaje α, factor de descuento γ, y probabilidad de exploración ϵ
while no ha convergido **do**
  Inicializar el estado s
  for cada episodio **do**
    Obtener a mediante ϵ-Greedy
    Ejecutar acción a, obtener recompensa r y siguiente estado s'
    Actualizar $Q(s, a)$ según la fórmula:

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

    Establecer $s \leftarrow s'$
  end for
end while

3.4. SARSA

En este algoritmo parecido a Q-Learning, pero con una diferencia clave en la forma en que actualiza la tabla de valores de acción-estado. Mientras que Q-Learning utiliza la acción de mayor valor en el siguiente estado para actualizar los valores, SARSA utiliza la acción que realmente se toma en el siguiente estado (lo que hace que sea un algoritmo .on-policy", es decir, depende de la política seguida durante el aprendizaje).

- Primero, inicializamos la tabla de valores Q para todos los estados s y acciones a posibles. También, será necesario establecer una tase de aprendizaje α, un factor de descuento γ y la política de exploración (como en nuestro caso ϵ-greedy.
- Para cada episodio, el agente empieza en un estado inicial s, entonces, elige una accion a con nuestra política. Ejecutamos la acción a y observamos la recompensa r y el siguiente estado s'. Actualizamos el valor de la tabla Q de acuerdo a la fórmula de actualización de SARSA (mostrada en el pseudocodigo)
- Repetir hasta cierto número de episodios.

Añadir que SARSA es una buena opción cuando se quiere

aprender y mejorar una política en línea y de manera controlada, ya que se basa en las decisiones reales del agente durante su interacción con el entorno. Al actualizar la función de valor en función de las acciones que el agente efectivamente toma, el algoritmo se adapta de manera más segura a las políticas exploratorias. Además, SARSA es útil en situaciones donde el agente debe balancear la exploración y explotación sin depender de una política de comportamiento completamente aleatoria o determinística.

Algorithm 4 SARSA (State-Action-Reward-State-Action)

Inicializar: Tabla de valores $Q(s, a) = 0$ para todos los s y a
Definir tasa de aprendizaje α, factor de descuento γ, y probabilidad de exploración ϵ
for Cada episodio **do**
  Inicializar el estado s_0
  Elegir acción a_0 en s_0 usando política ϵ-greedy
  while el estado no es terminal **do**
    Ejecutar acción a_0, obtener recompensa r y siguiente estado s_1
    Elegir acción a_1 en s_1 usando política ϵ-greedy
    Actualizar $Q(s_0, a_0)$ según la fórmula:

$$Q(s_0, a_0) \leftarrow Q(s_0, a_0) + \alpha \left(r + \gamma Q(s_1, a_1) - Q(s_0, a_0)\right)$$

    Establecer $s_0 \leftarrow s_1$ y $a_0 \leftarrow a_1$
  end while
end for

3.5. Deep Q-Learning

Deep Q Learning es un método de aproximación no lineal, este parte de Q learning, pero realiza una aproximación de Q mediante la utilización de redes neuronales. A continuación explicamos el algoritmo paso a paso.

- **Inicialización**:
  - Inicializar la red $Q(s, a; \theta)$, la red objetivo $Q'(s, a; \theta')$, y la memoria M.
- **Interacción con el entorno**:
  - Seleccionar a_t con ϵ-greedy, ejecutar acción, observar r_{t+1}, s_{t+1}.
- **Almacenamiento**:
  - Almacenar $(s_t, a_t, r_{t+1}, s_{t+1})$ en M.
- **Entrenamiento**:
  - Si el batch es grande, tomar muestra y calcular:

$$y_i = r_i + \gamma \max_{a'} Q'(s_{i+1}, a'; \theta')$$

  - Calcular error y actualizar θ con gradiente descendente.
- **Actualización de la red objetivo**:
  - Cada τ pasos, actualizar $\theta' \leftarrow \theta$.

Antes de ver el pseudocódigo añadir que este tipo de algoritmo es una excelente alternativa cuando se trabaja con espacio continuo. Ya que permite trabajar sin el uso de la discretizacion, simplemente se trabaja con tensores. Esto facilita la implementación y sobre todo es más óptimo para problemas de espacios grandes, permitiendo al algoritmo alcanzar una política óptima. También decir que al trabajar con redes neuronales tienen un alto potencial para hacerse más complejos a medida que se necesiten para resolver problemas de mayor complejidad.

Algorithm 5 Deep Q-Learning (DQN)

Require: Red $Q(s, a; \theta)$, red objetivo $Q'(s, a; \theta')$, memoria M
Inicializar $Q(s, a; \theta)$, $Q'(s, a; \theta')$, M
for cada episodio **do**
  Inicializar s_0
  for cada paso **do**
    Seleccionar a_t con ϵ-greedy
    Ejecutar a_t, observar r_{t+1}, s_{t+1}
    Almacenar $(s_t, a_t, r_{t+1}, s_{t+1})$ en M
    if batch grande **then**
      for cada transición en el batch **do**
        Calcular $y_i = r_i + \gamma \max_{a'} Q'(s_{i+1}, a'; \theta')$
        Actualizar θ usando δ_i
      end for
    end if
    if pasos τ **then**
      Actualizar $\theta' \leftarrow \theta$
    end if
  end for
end for

3.6. Sarsa Semi Gradiente

Es una variante del algoritmo SARSA que utiliza aproximación de funciones para estimar la función de valor acción estado $Q(s, a)$. A diferencia de SARSA tradicional, que utiliza una tabla para almacenar los valores $Q(s, a)$, SARSA semi-gradiente aproxima $Q(s, a)$ usando una función parametrizada (por ejemplo, una función lineal o una red neuronal) y actualiza los parámetros de esta función utilizando el método del semi-gradiente.

- Se inicializan los pesos $w(s, a)$ a cero para todos los pares estado-acción. También se establecen la tase de aprendizaje α, un factor de descuento γ y la política de exploración (como en nuestro caso ϵ-greedy.
- Se elige una acción a en el estado s usando la política ϵ-greedy, lo que garantiza que el agente o explote según el valor de ϵ,
- Después de ejecutar la acción a y recibir la recompensa r, el agente elige una nueva acción a' en el siguiente estado s', se calcula el valor objetivo, que depende del valor esperado de la siguiente acción.
- Calculamos el error temporal, que es la diferencia entre el valor objetivo y el valor actual y actualizamos los pesos mediante el gradiente del error temporal, lo que ajusta los valores de Q para que se aproximen más a los valores óptimos.

Volver a comentar que SARSA Semi-Gradiente parte del algoritmo SARSA al usar aproximación de funciones para estimar el valor de acción Q(s,a), lo que permite manejar espacios de estado y acción más grandes. Utilizando gradientes, el algoritmo ajusta los parámetros del modelo para mejorar las predicciones de Q(s,a) de manera continua y eficiente, lo que lo hace adecuado para problemas con grandes espacios

de estado o entornos continuos. El uso de gradientes permite que SARSA semi-gradiente sea más flexible y escalable. Por último decir que este método también puede implementar como un método de aproximación no lineal mediante el uso de redes neuronales. Lo que lo vuelve aún más completo para problemas donde el espacio es continuo y de gran tamaño.

Algorithm 6 SARSA Semi-Gradiente

Inicializar: Pesos $w(s, a) = 0$ para todos los s y a
Definir tasa de aprendizaje α, factor de descuento γ, probabilidad de exploración ϵ, decaimiento de ϵ, y valor mínimo de ϵ
while no ha convergido **do**
    Inicializar el estado s
    Elegir acción a en s usando política ϵ-greedy
    while el estado no es terminal **do**
        Ejecutar acción a, obtener recompensa r y siguiente estado s'
        Elegir acción a' en s' usando política ϵ-greedy
        Calcular valor objetivo:

$$\text{target} = r + \gamma \cdot Q(s', a')$$

        Calcular el error temporal (TD error):

$$\delta = \text{target} - Q(s, a)$$

        Actualizar pesos:

$$w(s, a) \leftarrow w(s, a) + \alpha \cdot \delta$$

        Actualizar ϵ mediante decaimiento:

$$\epsilon \leftarrow \text{máx}(\epsilon \cdot \epsilon_{\text{decay}}, \text{min\_epsilon})$$

        Establecer $s \leftarrow s'$ y $a \leftarrow a'$
    end while
end while

3.7. Deep Sarsa

Continuando con los algoritmos implementados seguimos con Deep Sarsa. Deep Sarsa es una implementación que parte de SARSA a la cual en vez de utilizar Q usa un método de aproximación no lineal para estimar Q, este método es la utilización de redes neuronales. Ahora incluimos la explicación paso a paso del algoritmo

- **Inicialización**:
  - Inicializar la red $Q(s, a; \theta)$, la red objetivo $Q'(s, a; \theta')$, y la memoria M.
- **Interacción con el entorno**:
  - Seleccionar acción según π, ejecutar acción, observar recompensa y estado.
- **Almacenamiento**:
  - Almacenar la transición $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ en M.
- **Actualización**:
  - Si el batch es grande, calcular el valor objetivo $y_i = r_{i+1} + \gamma Q'(s_{i+1}, a_{i+1}; \theta')$.
  - Calcular el error de predicción δ_i y actualizar los pesos de la red θ.
- **Actualización de la red objetivo**:
  - Cada τ pasos, actualizar $\theta' \leftarrow \theta$.

Añadir que este método es extremo útil para problemas de espacios continuos, sobre todo cuando el espacio continuo se vuelve muy grande, ya que permite realizar una aproximación óptima de Q mediante una red neuronal, ademas tambien permite trabajar con tensores. Comentar que además tiene mucho potencial dado que con base en la complejidad del problema podemos realizar una red neuronal más o menos compleja. A continuación el pseudocodigo.

Algorithm 7 Deep SARSA Simplificado

Require: Red $Q(s, a; \theta)$, red objetivo $Q'(s, a; \theta')$, memoria M
Inicializar $Q(s, a; \theta)$, $Q'(s, a; \theta')$, M
for cada episodio **do**
    Inicializar s_0, seleccionar a_0
    for cada paso **do**
        Ejecutar a_t, observar r_{t+1}, s_{t+1}
        Almacenar $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ en M
        if batch grande **then**
            for cada transición en el batch **do**
                Calcular $y_i = r_{i+1} + \gamma Q'(s_{i+1}, a_{i+1}; \theta')$
                Actualizar θ usando δ_i
            end for
        end if
        if pasos τ **then**
            Actualizar $\theta' \leftarrow \theta$
        end if
    end for
end for

3.8. Epsilon Greedy

Para la toma de decisiones hemos utilizado la política de épsilon greedy con decaimiento del épsilon. En el algoritmo simplemente lo que se hará es seleccionar un número aleatorio y si ese se encuentra por debajo del valor de épsilon se tomará una acción aleatoria(Exploración), si el valor sale mayor se escogerá la acción óptima (Explotación). Posteriormente, se realiza el decaimiento de épsilon, este decaimiento según la implementación puede ocurrir tras cada paso, tras cada episodio o después de elegir acción. Añadir que épsilon greedy con decaimiento nos parece la política óptima de selección de acción, ya que combina de manera idónea la exploración con la explotación. A continuación mostramos el pseudocódigo:

3.9. Tiling Code

Para el entorno de Mountain Car hemos implementado la utilización de **Tiling Code** para discretizar el espacio continuo a uno discreto. Para discretizar un espacio continuo la técnica lo hace con una serie de rejillas (llamadas **tilings**) que se superponen, lo que permite representar el espacio de estados de manera eficiente en algoritmos de aprendizaje por refuerzo. Cada *tiling* consiste en una división del espacio en intervalos o celdas, y cada observación continua se mapea a un índice de celda dentro de cada *tiling*. Para evitar el sobreajuste y mejorar la generalización, se usan múltiples *tilings* desplazados unos de otros. Es decir, cada *tiling* se genera con un pequeño desplazamiento en relación a los otros, lo que garantiza

Algorithm 8 Selección de acción ϵ-Greedy con decaimiento de ϵ

Require: $Q(s, a)$, estado actual s, ϵ_0, ϵ_{min}, factor de decaimiento γ
Ensure: Acción a
Inicializar $\epsilon \leftarrow \epsilon_0$
if un número aleatorio $r \in [0, 1]$ es menor que ϵ **then**
  $a \leftarrow$ acción aleatoria
else
  $a \leftarrow \arg\max_{a'} Q(s, a')$ ▷ Elegir acción con mayor valor Q
end if
Reducir ϵ: $\epsilon \leftarrow \max(\epsilon_{\text{min}}, \epsilon \times \gamma)$ ▷ Decaimiento de ϵ
return a

que el aprendizaje no dependa únicamente de una partición particular del espacio, sino que se distribuya a través de varias representaciones. Cada índice en un *tiling* se mapea a una celda específica, y la combinación de los índices de todas las rejillas forma una representación discreta del estado. El pseudo código es el siguiente:

Algorithm 9 Discretización del espacio continuo usando Tiling Code

Require: $n_{\text{tilings}} > 0$, $n_{\text{bins}} > 0$
Ensure: Estado discretizado
Inicializar n_{tilings}, n_{bins}, $desplazamiento$
Inicializar $estado_{\text{continuo}}$, $estado_{\text{discreto}} \leftarrow []$
for cada componente c en $estado_{\text{continuo}}$ **do**
  $indices_{\text{tilings}} \leftarrow []$
  for $tiling_{\text{id}} = 1$ hasta n_{tilings} **do**
    $indice \leftarrow \text{floor}((c + desplazamiento \times tiling_{\text{id}}) \times n_{\text{bins}} / rango_{\text{maximo}})$
    $indices_{\text{tilings}}$.append($indice$)
  end for
  $estado_{\text{discreto}}$.append($indices_{\text{tilings}}$)
end for
$estado_{\text{discreto\_aplanado}} \leftarrow$ aplanar($estado_{\text{discreto}}$)
return $estado_{\text{discreto\_aplanado}}$

4. Evaluación/Experimentos

Comentar simplemente los experimentos han sido realizados en Google Colab. En los notebooks entregados se podrá apreciar las librerías usadas, parámetros utilizados, ejecutarlos en Colab, ver los resultados, contrastarlos, etc. Además se incluirán unos videos mostrando la resolución de los diferentes entornos, habrá una carpeta por problema y tantos videos como agentes. Entrando en el propio documento dividiremos la sección por cada uno de los estudios y en el estudio mostraremos las gráficas de evolución de las recompensas y del tamaño de los episodios. Recomendamos ampliamente contrastar esta sección con los notebooks asi como ver los videos incluidos.

4.1. Taxi

El agente ya empieza mejorando desde el principio aprendiendo a buena velocidad, convergiendo rápidamente en la proporción de recompensas y a un ritmo más lineal en el tamaño de los episodios, estabilizándose por el episodio 1100 más o

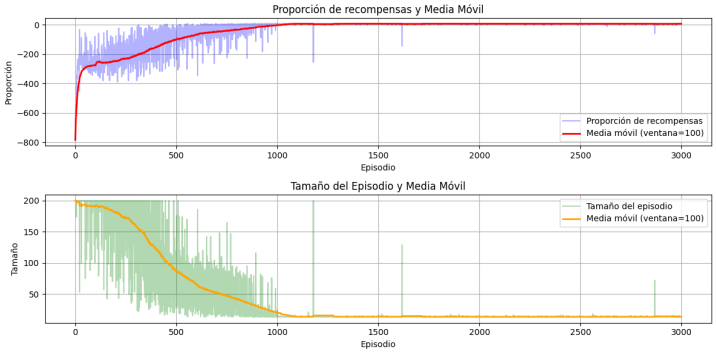

Figura 1: Resultados Taxi Q Learning

menos. Podemos ver que para haber utilizado 3000 episodios para el aprendizaje del agente, se ha podido adaptar perfectamente al entorno e incluso encontrar una solución óptima en este problema.

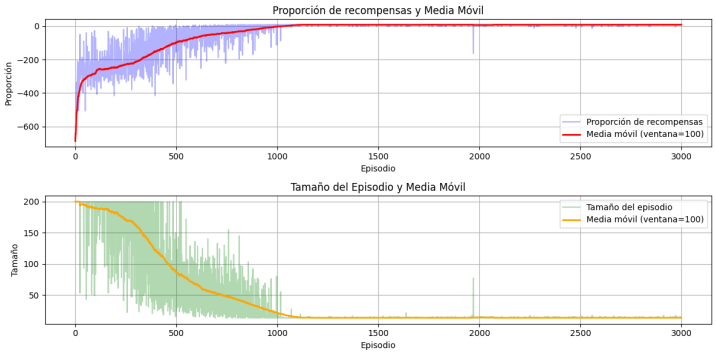

Figura 2: Resultados Taxi Sarsa

Respecto a Sarsa, tiene un comportamiento muy parecido a Q Learning, este tiene una convergencia muy rápida al principio en la proporción de resultados (pasando de -700 a -300 en los primeros 100 episodios) que luego se va estabilizando hasta en el episodio 1100 donde se queda en valores entre 5 y 7.5. Respecto al tamaño de episodios tiene un aprendizaje bastante estable a lo largo del tiempo, es este caso si se puede observar que una vez alcanza un número bajo de pasos dentro de un capítulo, no tiene subidas imprevistas como ocurre en algunos momentos en Q Learning, pero esto no afecta en absoluto al resultado final.

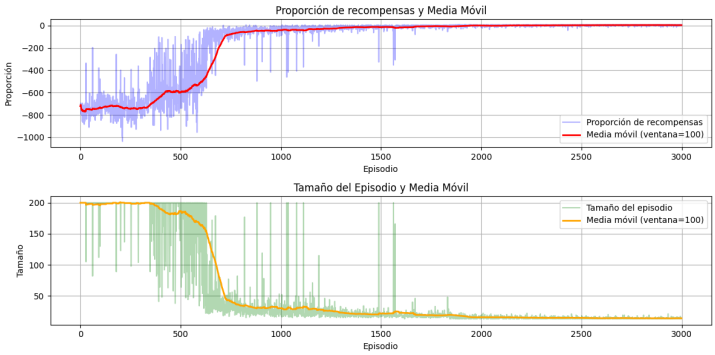

Figura 3: Resultados Taxi Monte Carlo Off Policy

Respecto al algoritmo de Monte Carlo Off Policy, podemos observar que comparado con los métodos tabulares, en este caso al principio mantiene tanto la proporción de recompensas como el tamaño del episodio, pero a partir del episodio 400 empieza a aprender aumentando las recompensas promedias y reduciendo el tamaño promedio. Se puede apreciar que entre los episodios 750 y 1750 aunque el tamaño de los episodios en la media movil se vaya reduciendo paulatinamente, el tamaño real del episodio es bastante inestable llegando a veces incluso a truncarse algún episodio.

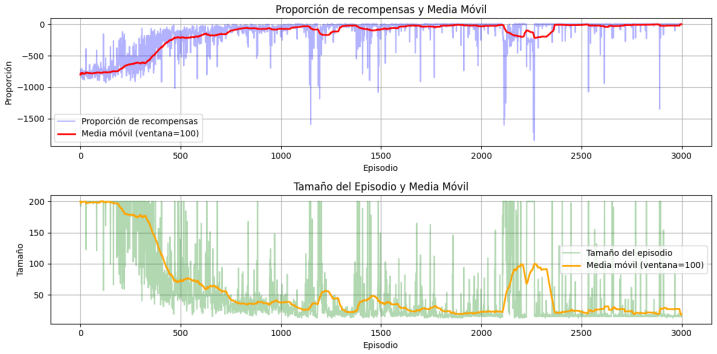

Figura 4: Resultados Taxi Monte Carlo On Policy

En On Policy podemos ver que es el algoritmo que peor rendimiento tiene en general para este entorno, aunque se vea que aprende a lo largo del tiempo, se puede apreciar que durante todo el aprendizaje no llega a converger del todo, incluso en sus últimos episodios se ve que la media móvil del tamaño aumenta un poco.

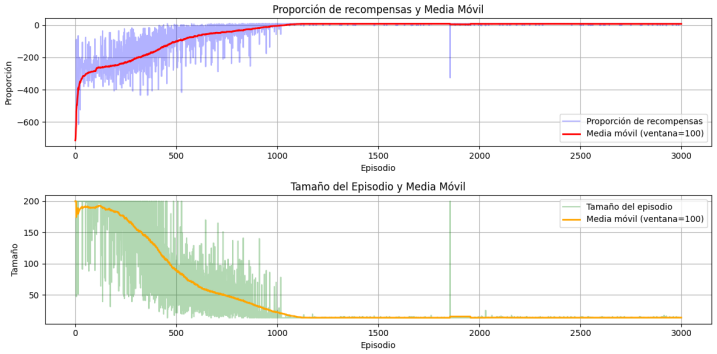

Figura 5: Resultados Taxi Sarsa Semi Gradiente

Respecto a Sarsa Semi Gradiente, tiene un comportamiento muy parecido a Sarsa Tradicional, se ve que aprende rápidamente y converge de una manera estable a lo largo del tiempo hasta el episodio 1100 más o menos. Una vez que converge, también se observa que no tiene problemas para mantener esa proporción y tamaño de los episodios a lo largo del tiempo, como estaba pasando en Monte Carlo.

4.2. Mountain Car

Como es obvio el agente comienza con episodios de 200 pasos los cuales terminan por truncamiento, pero como podemos apreciar lentamente comienza a mejorar esto reduciendo el tamaño de los episodios y por ende mejorando las recompensas. Esto hasta llegar al episodio 5000 donde se estabilizan las resoluciones sobre un valor de 120 pasos y -120 de recompensa. Comentar también que su recompensa media en los 20000 episodios ha sido de -136.4789. Con estos resultados podemos determinar que este agente se adapta al problema y su algoritmo resulta óptimo y efectivo para la resolución del problema así como encontrar una buena solución. En el video se podrá apreciar como quizás admite algo de mejora, ya que realiza algunos movimientos que no son necesarios para alcanzar la solución; sin embargo, su política final resulta suficientemente buena para lograr la resolución.

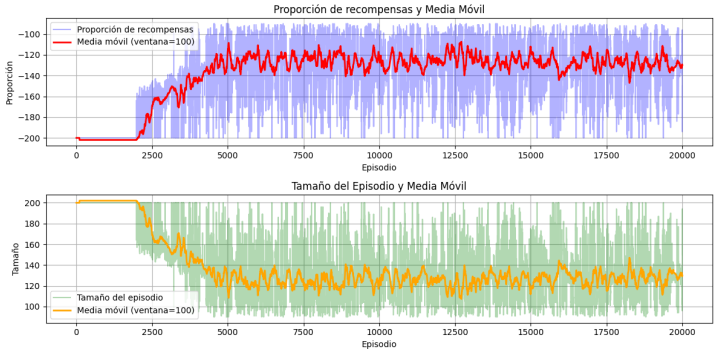

Figura 6: Resultados Mountain Car Sarsa Semi Gradinte

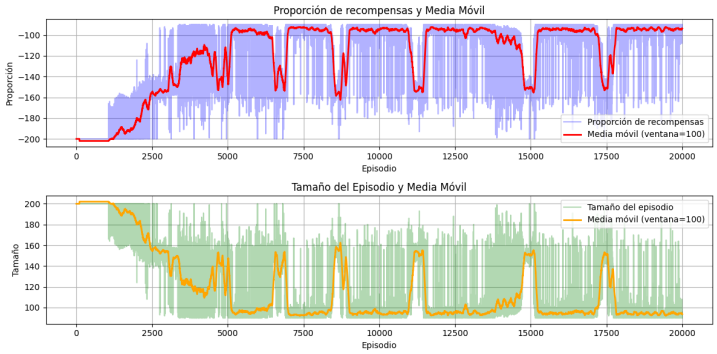

Figura 7: Resultados Mountain Car Q Learning

En Q Learning podemos el cómo inicialmente va mejorando poco a poco hasta lograr sus mejores recompensas que se encuentran entorno -120 y -100. Lo más destacable de este entrenamiento es que presenta cierta inestabilidad con picos -100, pero con algunas caídas que llegan hasta -140. Sigue siendo una solución muy óptima pese a esto y la política final parece adaptarse de manera perfecta, ya que la resolución del problema es perfecta. Por lo que quitando ese pequeño inconveniente el algoritmo de Q Learning parece tratar este problema de idónea, viéndose reflejado en su recompensa promedio de -117.25185

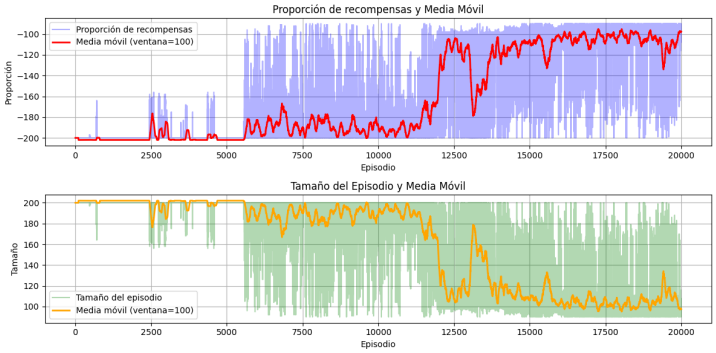

Figura 8: Resultados Mountain Car Deep Q Learning

El agente Deep Q Learning parece que se le dificulta el empezar a notas mejoras en el tamaño de sus episodios, así como sus recompensas, tardando hasta el episodio 12500 para poder lograr una optimización en torno a los -120 y -100 de recompensa. Esto son valores bastante buenos y como sus últimos episodios los mantiene con un promedio alto, consigue que la resolución del problema sea simplemente perfecta, ya que su política al final mejora mucho. Sin embargo, el tardar tanto le trae como consecuencia una recompensa media de -159.31655. Lo cual no está mal y admite mejora. Entonces su principal problema se debe a la tardanza en lograr mejorar, esto se puede deber a que quizás los métodos de aproximación no lineal no trabajen del todo bien con este problema. Aunque nos decantamos a que esto se debe al uso de redes neuronales con tiling que parece no ser una estrategia que mejore los resultados, por lo visto en el siguiente entorno quizás una aproximación mejor será en este método usar tensores en lugar las *active_features*. Pero en conclusión el algoritmo es efectivo para el problema.

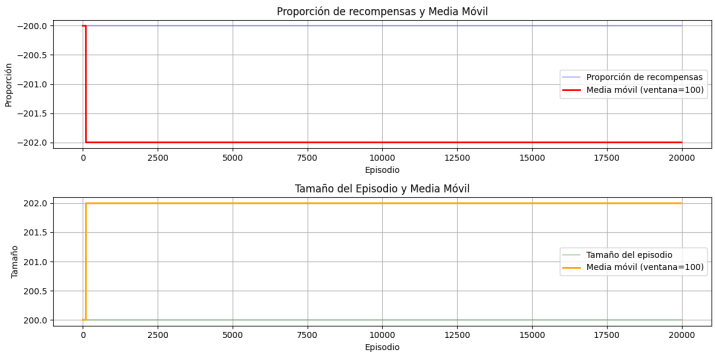

Figura 9: Resultados Mountain Car Monte Carlo

Unimos los dos Monte Carlo porque los resultados son iguales. Ninguno de los 2 logra la resolución del problema, la versión on policy no aparece aprender nada y la off policy sí que consigue la idea de que es necesario retroceder; pero no sabe cuanto ni que hacer con ese impulso. Por lo que ninguno de los 2 resuelve el problema y sus episodios están atascados en un tamaño de 200 pasos. Su recompensa promedio por ende es de -200. Creemos que estos resultados se deben a 2 motivos. El primero es que Monte Carlo vuelve a visitar los episodios recorridos y esto creemos que al combinarlo con un espacio de observación continuo que necesita discretización, parece que le trae problemas y que hace que los algoritmos no se adapten. El segundo problema pensamos que es a su actualización episódica, la actualización de Q se da tras cada episodio, no tras cada paso, lo cual quizás trae problemas para que converja. En conclusión para este problema Monte Carlo no se presenta como una opción.

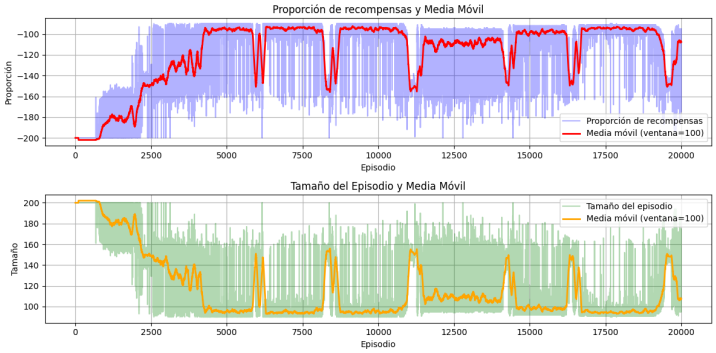

Figura 10: Resultados Mountain Car Sarsa

Sarsa obtiene resultados similares a Q Learning podemos apreciar también ese entrenamiento algo inestable, aunque su política final es óptima y realiza una solución perfecta. Aprende y mejora relativamente rápido y sus caídas en recompensas no son elevadas, su recompensa media es de -115.84365. Quizás esta inestabilidad se debe a la utilización de métodos tabulares con tiling que parece ser que para este problema hace que el entrenamiento sea algo inestable, aunque, por otro lado, se consiguen agentes excelentes que resuelven de manera eficaz el problema.

4.3. Lunar Lander

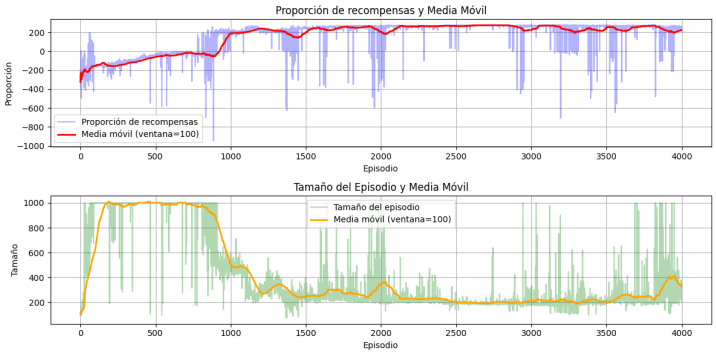

Figura 11: Resultados Lunar Lander Deep Q Learning

Este agente logra resultados impresionantes, sobre el episodio 1000 ya consigue alcanzar recompensas difícilmente superables para este problema con recompensas de 200 y algo más. Inclusive el tamaño de los episodios también decae rápidamente con un tamaño óptimo de 200 pasos. Comentar que la red neuronal usada es sencilla y que las recompensas medias del entrenamiento han sido de 165.76473469594697. Lo cual demuestra que el agente es efectivo y óptimo en la resolución del problema. También en el video se aprecia una resolución del ejercicio bastante buena, aunque el siguiente algoritmo es

verdad que la logra en un menor tiempo, pero eso no quita que esta sea un muy buena solución. Para este agente pode-

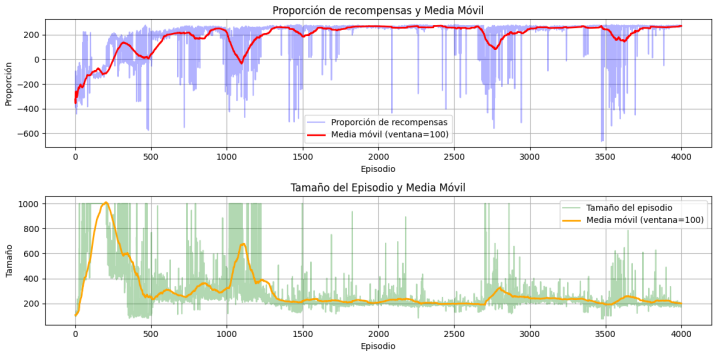

Figura 12: Resultados Lunar Lander Deep Sarsa

mos repetir los comentarios anteriormente dichos, inclusive con más ímpetu dado que este algoritmo parece alcanzar una solución óptima de 200 en un menor número de pasos, decayendo también rápidamente el tamaño de los episodios. El entrenamiento también luce bastante estable y con valores muy positivos. Su promedio de recompensas en el entrenamiento está en 196.1899928322434. Por lo que simplemente el algoritmo se adapta de manera increíblemente buena al problema, inclusive la solución que podemos ver en los videos es difícilmente mejorable.

- Configuración experimental: herramientas, entornos de prueba, datasets empleados, etc.
- Métricas utilizadas para evaluar el desempeño del método propuesto.
- Resultados obtenidos, presentados en tablas o gráficos según corresponda.
- Análisis crítico de los resultados y comparación con otros enfoques.

5. Conclusiones

Podemos extraer las siguientes conclusiones

- Los algoritmos de Monte Carlo parece que no son viables de cara a trabajar con entornos complejos, sobre todo entornos continuos que necesitan de discretización. En vista de los resultados obtenidos, se observa fácilmente como este algoritmo simplemente no es viable. Aunque a favor del algoritmo decir que para entornos más sencillos y con espacios de acción y observación discretos sí que parece una alternativa viable.
- Para entornos continuos es necesario aplicar discretización por ejemplo, mediante la técnica de Tiling. Además también se puede optar por métodos de aproximación que permitan aproximar Q de manera óptima, por ende resolver el problema.
- Podemos también apreciar que en entornos continuos demasiado complejos, aplicar Tiling no es una solución y algoritmos de aproximación lineal no son demasiado óptimos para resolver el problema.
- Algoritmos tabulares pueden resolver problemas de espacios continuos siempre cuando se pueda discretizar el espacio. Además añadir que para entornos discretos estos algoritmos obtienen excelentes resultados y en espacios discretizados también han obtenido un excelente rendimiento, quizás a mejorar algo la estabilidad de los entrenamientos, pero no afecta demasiado. Aunque obviamente ante entornos muy complejos no son una alternativa
- Comentada la limitación de Sarsa Semi Gradiente, añadir que pese a eso en entornos continuos no tan complejos o entornos discretos, el algoritmo consigue un entrenamiento eficiente y estable.
- Los algoritmos de aproximación no lineales son una solución muy óptima ante entornos continuos y entornos muy complejos. Comentar su escalabilidad que les permite realizar una red neuronal adaptada a la complejidad del problema. Comentar también sobre todo que mejora al usar los tensores en lugar del tiling y que quizás para entornos simples no es la mejor solución dada la complejidad intrínseca de las redes neuronales.
- Algunas ideas que se podrían mejorar sería la implementación de Sarsa Semi Gradiente para problemas más complejos, un aprendizaje más estable en algoritmos tabulares en Mountain Car, seguir explorando variaciones de Monte Carlo que permitan trabajar con entornos más complejos y el no usar tiling con ciertos métodos como Deep Q Learning.

6. Repositorio

El repositorio en el que se realiza el trabajo es el siguiente: https://github.com/DeMiKe16/RL_VLB.git