

Obligatorio

Inteligencia Artificial

Ingeniería en sistemas

M7A



Agustín Varela (303129)

Joaquín Mesa (295871)

[Link a repo](#)

[Wandb](#)

Flight-Level Adjustment Network

Desarrollo

Para abordar el problema, inicialmente implementamos el agente utilizando Q-Learning con espacio continuo. Una vez logrado esto, nos enfocamos en la discretización del espacio de estados. Posteriormente, implementamos la variante estocástica según lo indicado en el *paper*, que consiste en trabajar con un subconjunto de tamaño logarítmico del conjunto de datos. Esto permite reducir significativamente el tiempo de ejecución, sin comprometer la capacidad del agente para estimar de manera efectiva los valores óptimos, comparable al rendimiento obtenido utilizando el conjunto completo de datos.

Completadas estas etapas, procedimos a experimentar con diferentes combinaciones de hiperparámetros para determinar cuál ofrecía el mejor desempeño.

Definiciones

Gamma (Factor de descuento)

- Determina la importancia de las recompensas futuras. Un valor de gamma cercano a 1 hace que el agente valore más las recompensas futuras, mientras que un valor cercano a 0 hace que el agente se enfoque más en las recompensas inmediatas.

Alpha (Tasa de aprendizaje)

- Controla cuánto de la nueva información sobrescribe la antigua. Un valor alto de alpha hace que el agente aprenda rápidamente, pero puede causar inestabilidad. Un valor bajo hace que el aprendizaje sea más estable pero más lento.

Epsilon (Parámetro de exploración.)

- Controla la probabilidad de que el agente elija una acción aleatoria en lugar de la acción que cree que es la mejor. Esto ayuda a balancear la exploración de nuevas acciones y la explotación de acciones conocidas.

Pruebas de hiper parámetros

Inicialmente veníamos trabajando los hiperparametros en función de los datos de train lo cual estaba mal.. Por lo cual decidimos dejar lo que vinimos haciendo hasta ahora en el [\[Anexo\]](#). A partir de ahora haremos análisis del conjunto de test.

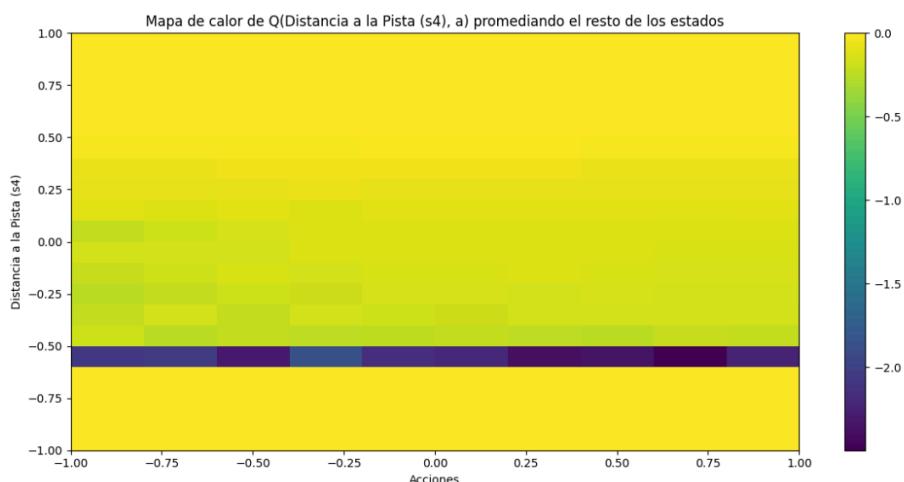
A pesar de esto, investigando y hablando con Fede y Santi notamos que teníamos cosas para mejorar: discretizar mejor los estados, implementar stochastic q learning (que hasta ahora no lo teníamos implementado) y graficar de distintas formas para entender mejor lo que sucede.

La discretización de los datos, la veníamos haciendo en partes iguales, ahora buscaremos a través de mapas de calor (graficando estados en función de acciones y promediando el resto de estados).

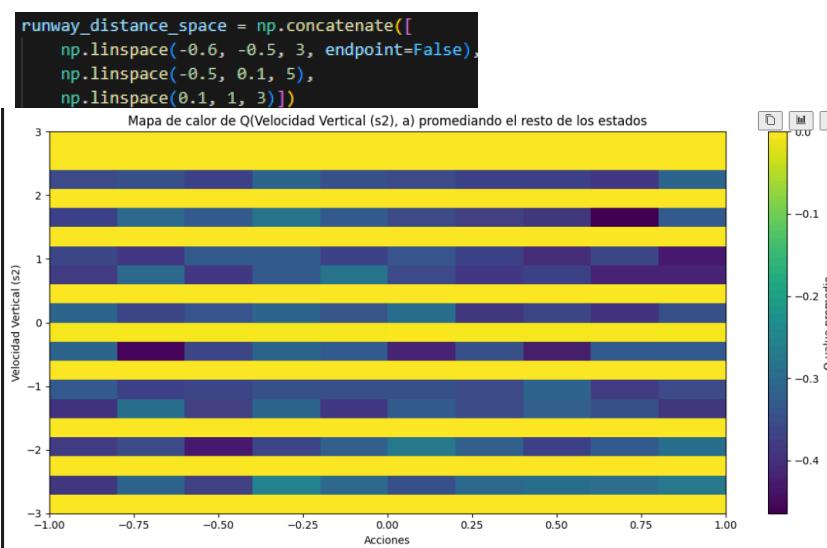
Inicialmente se quiso hacer una gráfica de calor donde se grafique la tabla Q pasada por UMAP para reducir las dimensiones y debido a dependencias de python y de librerías como numpy no pude aplicarlo para la gráfica. Mi idea era ver en la gráfica si ya tenía información para todos los estados para tratar de maximizar las acciones.

Entonces, se decidió hacer 4 gráficas, una por estado en función de acciones y promediando el resto de estados, de los cuales puedo deducir cómo mejorar la discretización.

Luego, la forma correcta para graficar los resultados de recompensa en función de episodios es a través de boxplots

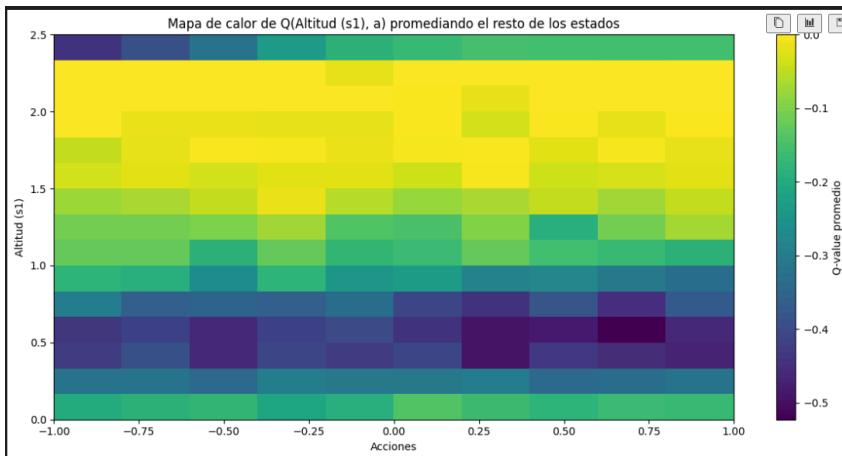


Dada la franja violeta que es donde se encuentran la mayoría de los datos, decidimos ese fragmento violeta partirla en 3 y luego la parte verde irla segmentando con intervalos más grandes. Descartamos el rango amarillo.



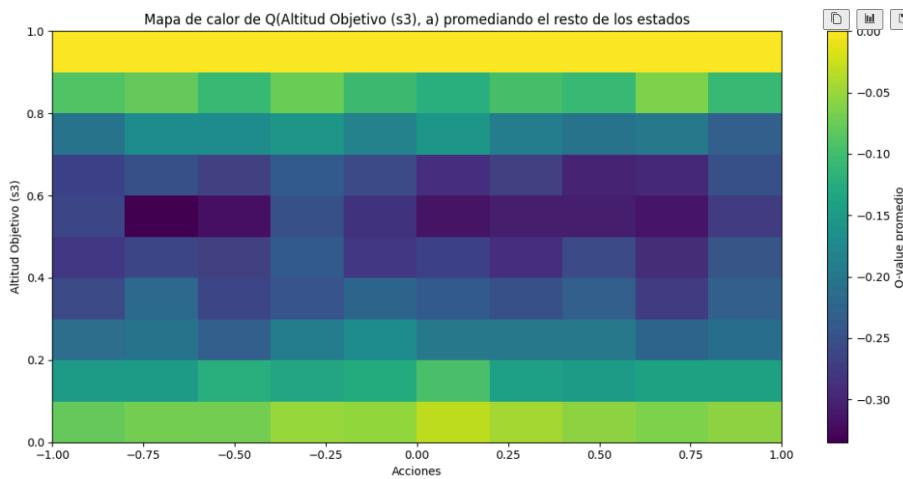
Para la velocidad vertical decidimos dejarlo con intervalos con la misma division, sacando los límites amarillos, es decir, limitamos en -2.7 y 2.5. Ya que la distribución de los datos es muy variable decidimos dejarlo así.

```
vertical_velocity_space = np.linspace(-2.7, 2.5, 15)
```



Para la altitud decidimos particionar la parte violeta en muchos más intervalos ya que vemos que se concentran muchos datos ahí, sin embargo el resto lo particionamos con intervalos más grandes.

```
altitude_space = np.concatenate([
    np.linspace(0, 0.2, 2, endpoint=False),
    np.linspace(0.2, 0.8, 6),
    np.linspace(0.8, 1.3, 3),
    np.linspace(1.3, 2.3, 3),
    np.linspace(2.3, 2.5, 3, endpoint=False)])
```



Para la altitud objetivo mantuvimos lo mismo que los anteriores.

```
target_altitude_space = np.concatenate([
    np.linspace(0, 0.2, 2, endpoint=False),
    np.linspace(0.2, 0.8, 12),
    np.linspace(0.8, 0.9, 2, )])
```

Somos conscientes que la discretización permite transformar un espacio continuo (teóricamente infinito) en una representación manejable y computacionalmente eficiente, agrupando rangos de valores en "buckets" discretos. Esto habilita al agente a aprender políticas útiles a partir de experiencias similares, sin necesidad de explorar cada punto exacto del espacio.

Sin una buena discretización:

- El agente puede generalizar mal o no aprender patrones relevantes.
- Se puede requerir una cantidad de memoria y de episodios de entrenamiento inabordable.
- Puede haber zonas importantes del estado (como cercanía a la pista) que no estén suficientemente representadas, lo que dificulta la toma de decisiones óptima.

Por lo tanto, la calidad del aprendizaje del agente depende directamente de como se discretiza cada variable: una mala discretización puede hacer que el agente nunca aprenda a aterrizar correctamente, aunque tenga la capacidad para hacerlo.

Justificación del uso de Stochastic Q learning

Stochastic q learning es una mejora del q learning tradicional diseñada para entornos con espacios de acción muy grandes. En estos escenarios, q learning se vuelve ineficiente porque en cada paso, necesita evaluar todas las acciones posibles para aplicar la operación de maximización, lo que implica un alto costo computacional.

En contraste, stochastic q learning realiza esta operación solo sobre un subconjunto aleatorio de acciones, cuyo tamaño puede ser logarítmico o sublineal respecto al total. Esto permite reducir drásticamente la complejidad computacional sin comprometer de forma significativa la calidad de la política aprendida.

El paper demuestra teóricamente que stochastic q learning converge al mismo valor óptimo que q learning tradicional en su versión sin memoria (memoryless), y que su variante con memoria mejora el desempeño en tiempo finito.

Entonces, stochastic q learning muestra una performance comparable e incluso superior a la de q learning, con una notable reducción en el tiempo de cómputo.

Entonces las ventajas que nos da son:

- Reduce la complejidad computacional por iteración
- Mantiene garantías teóricas de convergencia
- Escala mejor en entornos con muchos posibles movimientos
- Logra resultados competitivos incluso usando subconjuntos pequeños

Evidencia de ejemplo de uso de q learning vs stochastic q learning

Para ello medimos en un caso básico de ejecución para corroborar que hayamos implementado de forma correcta la parte stochastic, verificando que demore un poco menos que los resultados converjan al mismo valor.

```

episodes = 1000
epsilon = 0.99
gamma = 0.5
alpha = 0.5
rewards = agent.train_agent(env=env)
✓ 11m 43.4s

Episode: 0, Reward: -105.87, Epsilon: 0
Episode: 1, Reward: -105.41, Epsilon: 0

```

Sin stochastic

```

episodes = 1000
epsilon = 0.99
gamma = 0.5
alpha = 0.5
rewards = agent.train_agent(env=env)
✓ 9m 27.9s

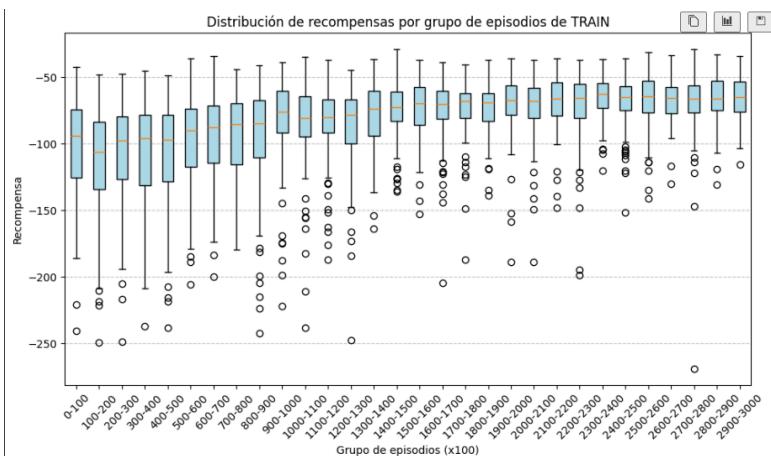
Episode: 0, Reward: -124.06, Epsilon: 0
Episode: 1, Reward: -121.23, Epsilon: 0

```

Con stochastic

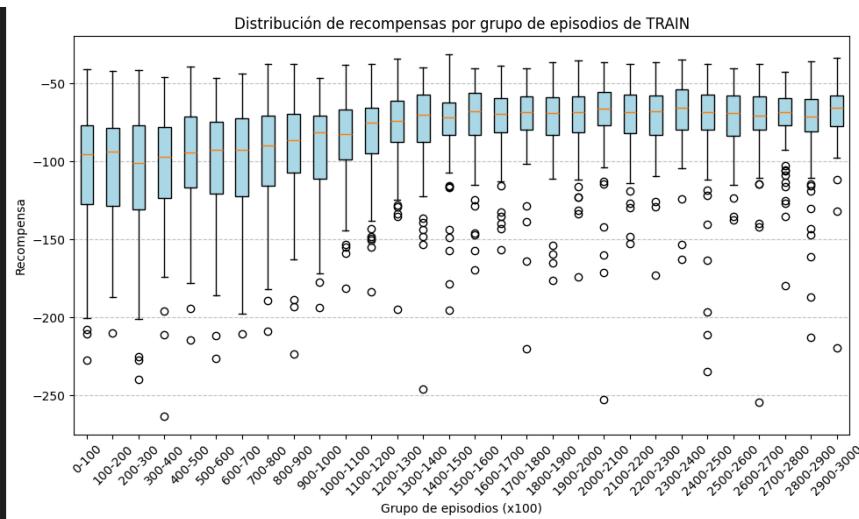
Prueba 10

- **Promedio de recompensas de test:** -60
- **Hiperparametros:** episodes =3000, epsilon=0.99, gamma=0.5, alpha=0.5
- **Conclusión:** Ahora probaremos cambiando los hiperparametros y veremos cómo fluctúa la gráfica. Esperábamos un crecimiento con una curva más pronunciada. Se apreció leve aprendizaje. Otra observación que notamos es que corriendo en human, la aeronave se mantiene muy bien en la altura target pero no aterriza.



Prueba 11

- **Promedio de recompensas de test:** -63
- **Hiperparametros:** episodes =3000, epsilon=0.99, gamma=0.9, alpha=0.5
- **Conclusión:** Probamos aumentando el gamma a 0.9 y no hubo cambios prácticamente, empeoro la recompensa un poquito. Ahora probaremos con el alpha a ver si cambia algo.

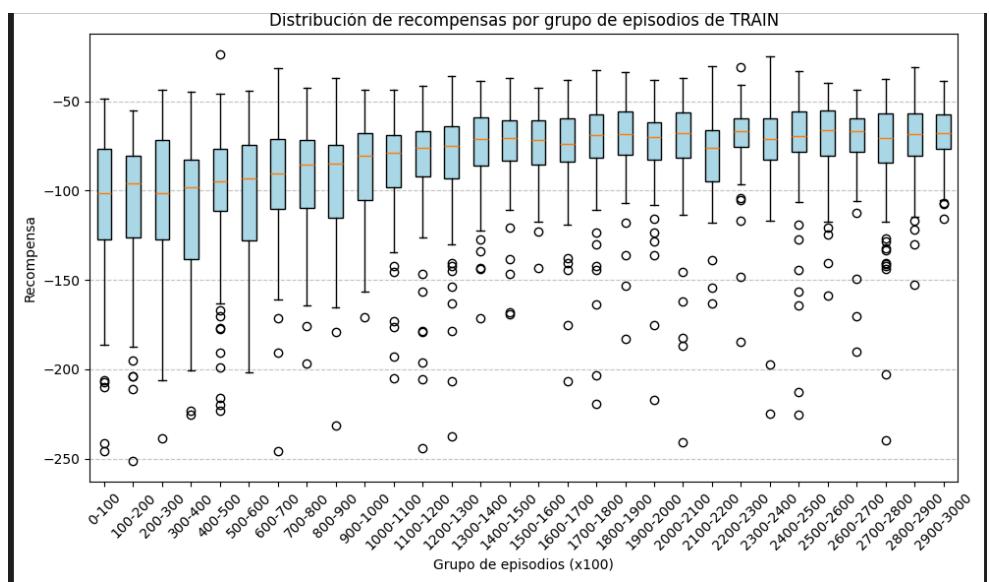


Prueba 12

- **Promedio de recompensas de test:** -69
- **Hiperparametros:** episodes =3000, epsilon=0.99, gamma=0.9, alpha=0.9
- **Conclusión:** Probamos aumentando el alpha a 0.9 y empeoro la recompensa un poquito.

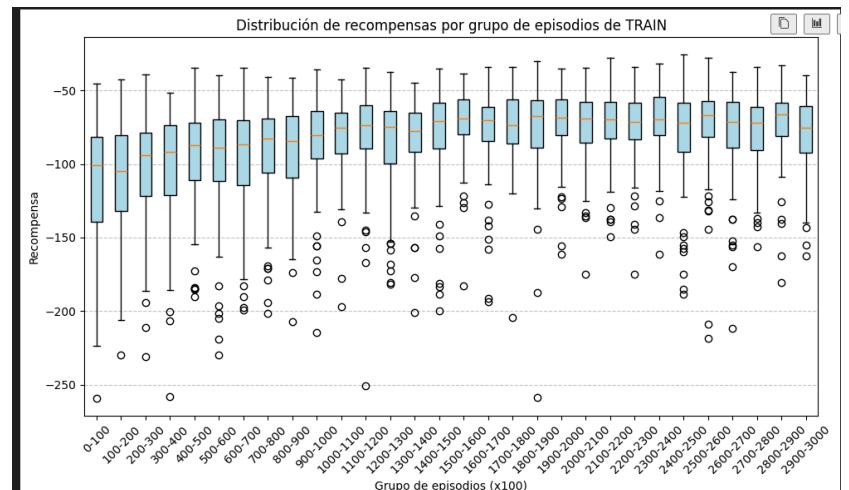
Ahora vamos a probar reducir las particiones de la velocidad vertical con el objetivo de ver si aprende mas rapido

y ve que cuando aterriza tiene más recompensa. Dado que en human sigue sin aterrizar para ningún episodio



Prueba 13

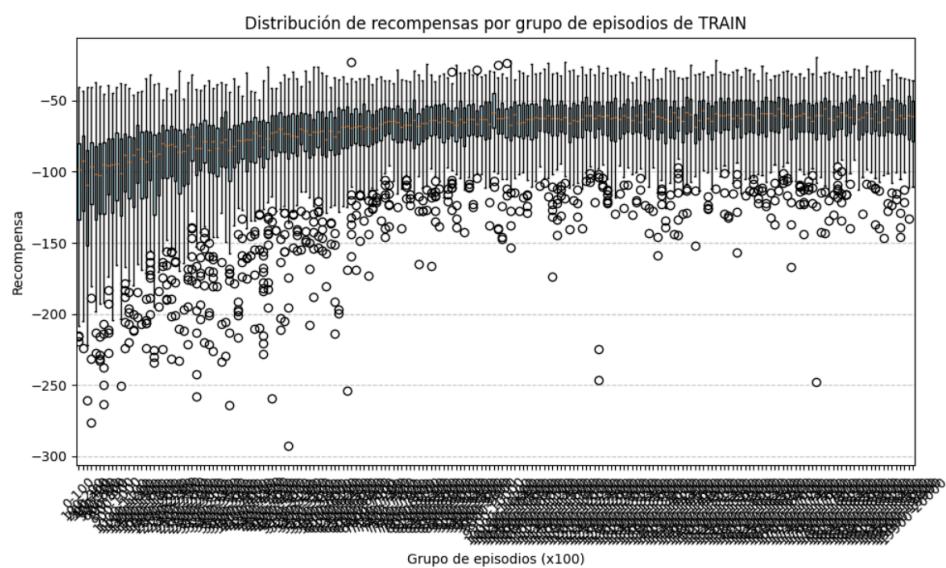
- **Promedio de recompensas de test:** -67
- **Hiperparametros:** episodes =3000, epsilon=0.99, gamma=0.9, alpha=0.9
- **Conclusión:** Probando con menos particiones en la velocidad vertical aceleramos un poco el proceso de entrenamiento y apenas mejoró el promedio de recompensas



```
vertical_velocity_space = np.linspace(-2.7,2.5, 3)
```

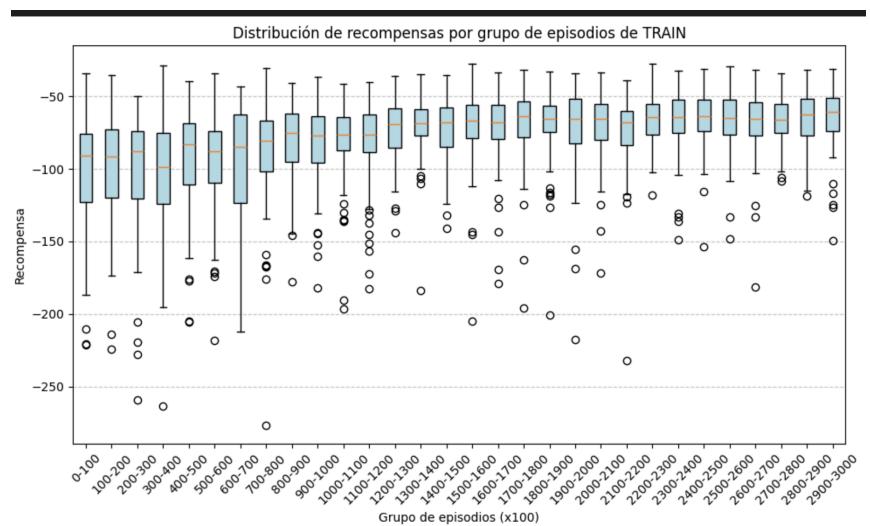
Prueba 14

- **Promedio de recompensas de test:** -55
- **Hiperparametros:** episodes =20000 , epsilon=0.99, gamma=0.5, alpha=0.5
- **Conclusión:** A partir del los 10000 episodios (aproximadamente la mitad), quedaron mal tipeados los valores del eje inferior, no tiene sentido seguir agregando episodios ya que prácticamente no hay una tendencia de mejora en las recompensas. Ahora voy a probar que pasa si aumento la cantidad de acciones , pienso que más resolución puede permitir descensos más suaves y decisiones más finas para llegar a pista.



Prueba 15

- **Promedio de recompensas de test:** -64
- **Hiperparametros:** episodes =3000 , epsilon=0.99, gamma=0.5, alpha=0.5
- **Conclusión:** No mejoró con agregarle más acciones, ahora vamos a probar cosas con el epsilon para aumentar la etapa de exploración, ya que nos damos cuenta viendo los valores de la gráfica de calor que siempre opta por mantener la altura target y nunca llega a aprender que aterrizando hacia la pista aumentaría la reward



Realizamos el siguiente cambio en el epsilon dado que nunca lograba aterrizar en la plataforma:

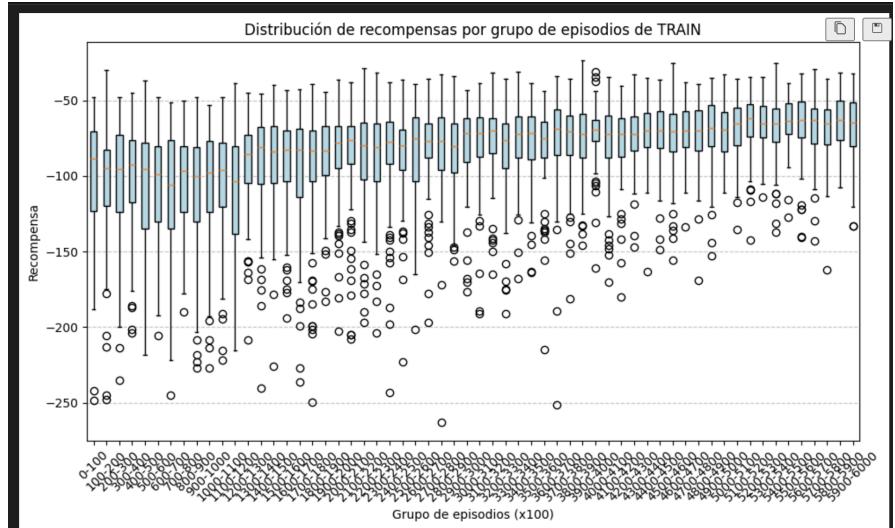
```

print(f'Episode: {episode}, reward: {total_reward:.2f}, epsilon: {epsilon:.2f}')
#epsilon = max(final_epsilon, initial_epsilon - epsilon_decay * episode)
if episode < episodes * 0.2:
    epsilon = initial_epsilon # Mantener el valor inicial durante el primer 20% de los episodios
elif episode >= episodes * 0.8:
    epsilon = final_epsilon # Fijar el valor final durante el último 20% de los episodios
else:
    epsilon = max(final_epsilon, initial_epsilon * np.exp(-episode / (episodes * 0.5)))

```

Prueba 16

- **Promedio de recompensas de test:** -58
- **Hiperparametros:** episodes =6000 , epsilon=0.99, gamma=0.5, alpha=0.5
- **Conclusión:** Vemos una notable mejora incluso en la recompensa, donde el rango de los datos de train va subiendo



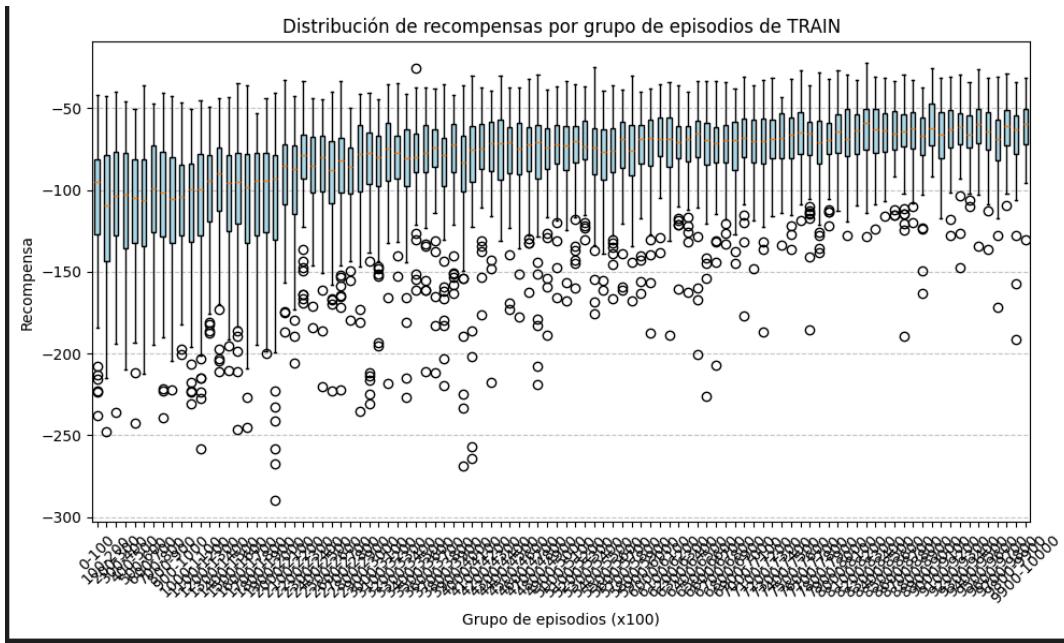
Prueba 17

- **Promedio de recompensas de test:** -59
- **Hiperparametros:** episodes =10000 , epsilon=0.99, gamma=0.5, alpha=0.5
- **Conclusión:** Dada la conclusión anterior que creímos que a mayores episodios mejoraba, nos damos cuenta que eso no sucede.

Seguimos teniendo el problema de la que la nave no aterrizó en la plataforma, para ello vamos a cambiar los hiperparámetros luego de probar aumentar las particiones del espacio de acciones, de agregar más particiones en algunos estados, y de dejar el epsilon mas grande para aumentar la exploración durante el entrenamiento.

Ahora vamos a probar con un gamma alto, con el objetivo de que priorice recompensas a largo plazo, es decir, que pruebe descender aunque pierda algo al inicio pero apueste a caer en

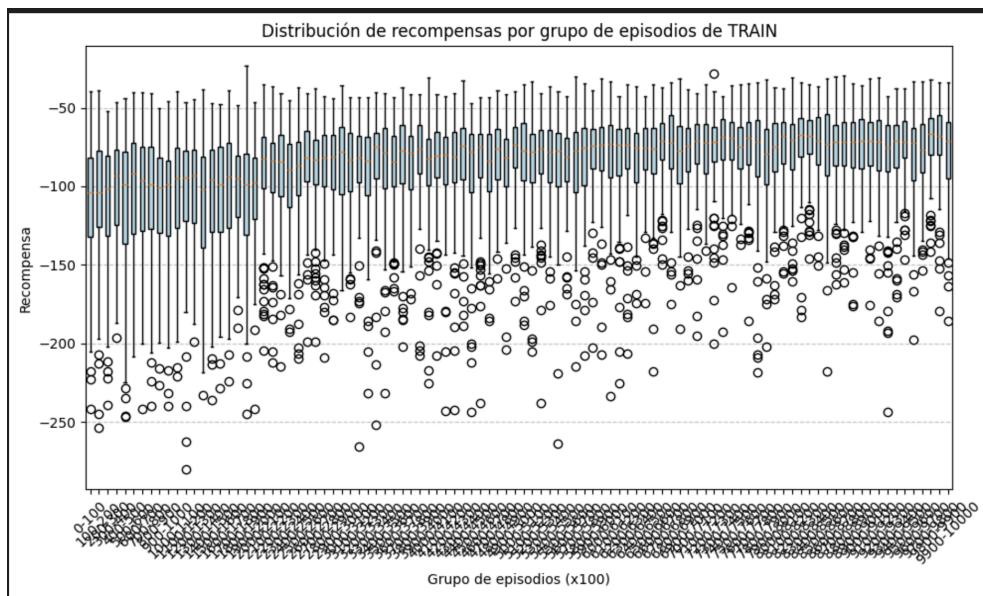
la plataforma y que sea mucho mejor la recompensa. Y un alpha bajo con el objetivo para que aprenda más lento pero “más seguro”.



Prueba 18

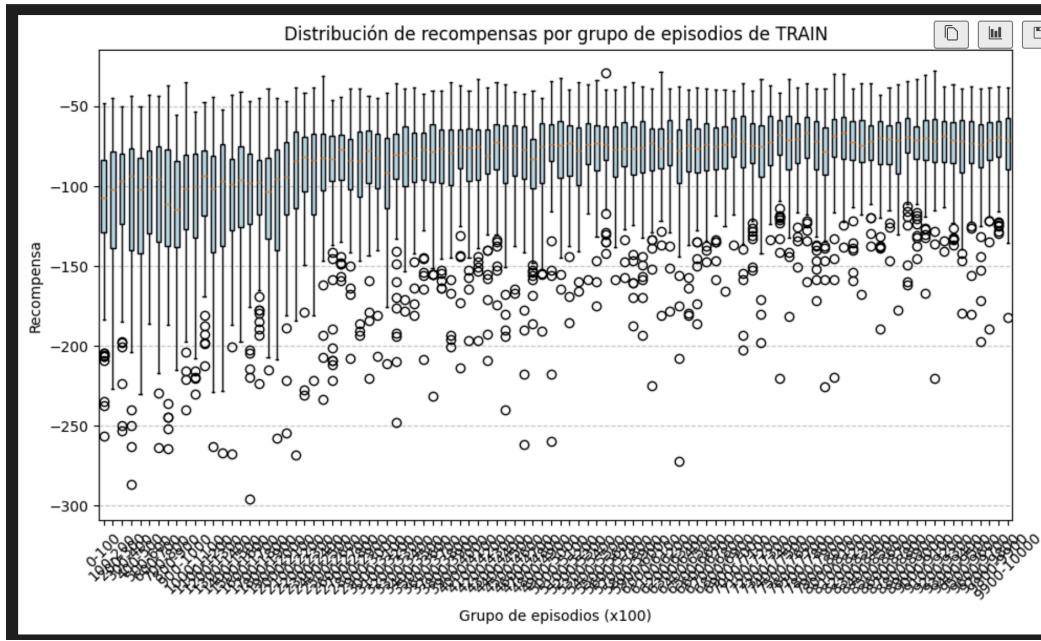
- **Promedio de recompensas de test:** -67
- **Hiperparametros:** episodes =10000 , epsilon=0.99, gamma=0.9, alpha=0.5
- **Conclusión:** La hipótesis de exploración que teníamos en base al teórico no funcionó, por lo tanto, aumentar el gamma no nos sirvió.

Pensamos que subir el gamma le daría más valor a las acciones que lo llevan a aterrizar, pero no lo logramos. Ahora probaremos descartar bajar el gamma en este caso, antes de deshacer el cambio del gamma.



Prueba 19

- **Promedio de recompensas de test:** -69
- **Hiperparametros:** episodes =10000 , epsilon=0.99, gamma=0.9, alpha=0.1
- **Conclusión:** Ahora concluimos que subir el gamma, con el alpha bajo y como estaba antes en 0.5 no fue una buena idea. Así que volveremos atrás a 0.5 y 0.5.



Cambio de discretización

Ahora vamos a volver a los hiperparametros de gamma y alpha con 0.5 pero probaremos una discretización un poco distinta para ver si mejora.

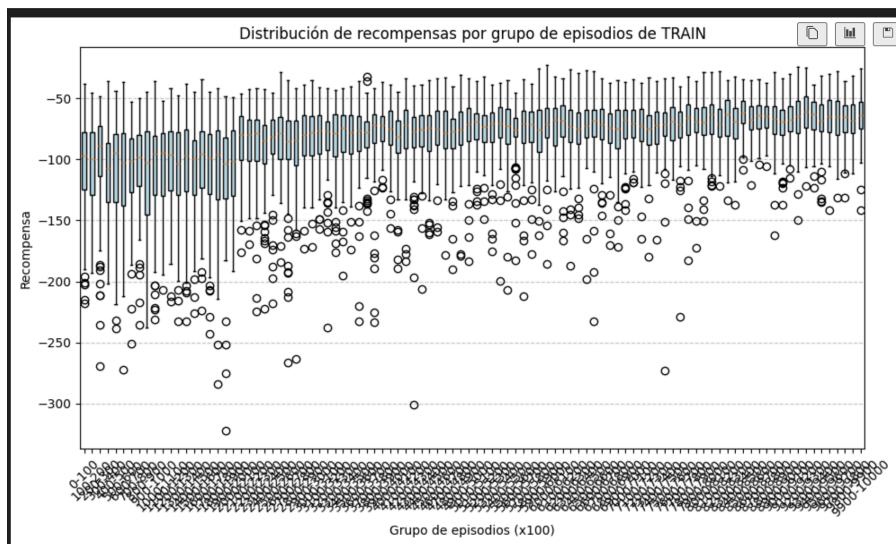
```
altitude_space = np.concatenate([
    np.linspace(0, 0.2, 2, endpoint=False),
    np.linspace(0.2, 0.8, 6),
    np.linspace(0.8, 1.3, 6),
    np.linspace(1.3, 2.3, 6),
    np.linspace(2.3, 2.5, 3)])
vertical_velocity_space = np.linspace(-2.7, 2.5, 3)

target_altitude_space = np.concatenate([
    np.linspace(0, 0.2, 2, endpoint=False),
    np.linspace(0.2, 0.8, 4),
    np.linspace(0.8, 0.9, 2)])
```

Le dimos más particiones a la altitud y le sacamos un poco de particiones a target, para ver si durante el entrenamiento logra valorizar más en algún momento descender.

Prueba 20

- **Promedio de recompensas de test:** -64
- **Hiperparametros:** episodes =10000 , epsilon=0.99, gamma=0.5, alpha=0.5
- **Conclusión:** Vemos que empeoró un poquito el resultado, la vez pasada que corrimos con estos hiperparametros, dio -60, por lo tanto deshacemos la discretización antes planteada. De la gráfica podemos analizar que no hay mucha mejora, queda bastante constante por lo tanto no promete que sea falta de episodios.



Dado que vemos que viendo el video generado por el Wrapper las decisiones son muy bruscas, es decir, sube muy rápido o intenta bajar muy de golpe, vamos a cambiar la discretización de las acciones y ver cómo actúa.

Cambio en la discretización de las acciones:

Antes habíamos probado discretizar en 20 de forma lineal y no mejoró.

Nueva discretización a probar:

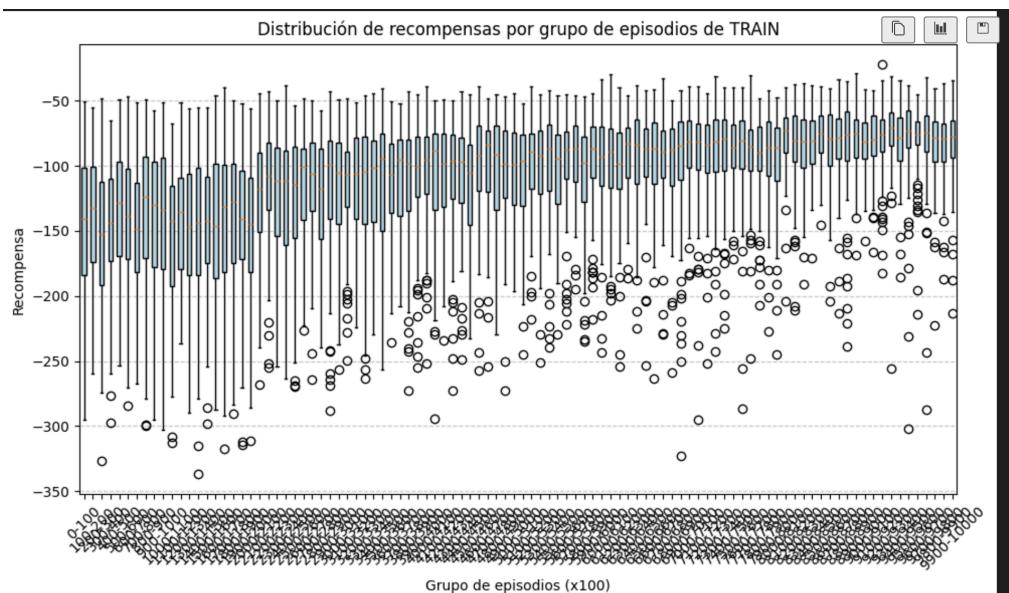
```
actions = np.concatenate([
    np.linspace(-1, -0.5, 3, endpoint=False),
    np.linspace(-0.5, 0.5, 8),
    np.linspace(0.5, 1, 6)
])
```

La idea de esto es poder suavizar los movimientos finos y los descendentes.

Prueba 21

Promedio de recompensas de test:

- **Hiperparametros:** episodes =10000 , epsilon=0.99, gamma=0.5, alpha=0.5
- **Conclusión:** La recompensa no mejoró con esta discretización, pero sin embargo, vemos un caso aislado que se ve, que estimo que logró aterrizar en la pista ya que vemos que la recompensa dió en el entorno de -20.



Prueba 21 (final)

Promedio de recompensas de test:

- **Hiperparametros:** episodes =20000 , epsilon=0.99, gamma=0.5, alpha=0.5
- **Recompensa:** -53

Dado que nos quedamos sin tiempo de seguir probando cosas, dejamos los hiperparametros y la discretización con lo mejor que teníamos hasta ahora (prueba 17).

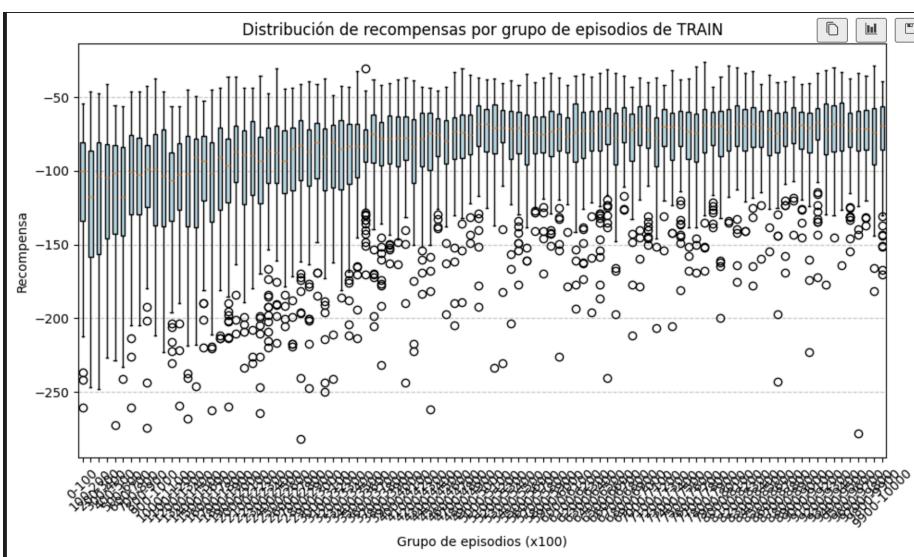
Ahora le agregamos más episodios logrando la mejor recompensa.

Este modelo quedó adjuntado en formato .pkl en el [\[Repo\]](#)

Experimentación con Double Q learning

A modo exploratorio hicimos una prueba entrenando un agente de double Q Learning, y lo que obtuvimos consideramos que es bastante bueno, sin ajustar el epsilon ni probar hiperparametros nuevos logramos una recompensa de -62. Apenas peor que la mejor lograda con el agente estocástico. Esto era esperable ya que se sabe que en matrices tan grandes de Q, donde tiene muchas dimensiones, es sabido que Q learning funciona mejor como explica en el paper. Por ello, no queríamos dejar de probarlo.

De todas formas, quedó la implementación en el código.



Una observación no menor es que vemos que en episodios tempranos ya aparecen buenas recompensas, es decir con valores menores a -50. Esto quiere decir que logró mejor exploración y estuvo cerca de aterrizar el agente o aterrizó. No seguimos profundizando ya que no es el objetivo del obligatorio hacerlo con Double Q Learning y por falta de tiempo.

Conclusión

Si bien consideramos que el trabajo fue sólido y logramos explorar una variedad de enfoques durante el entrenamiento, algunos resultaron más efectivos que otros. Probamos muchas de las alternativas disponibles y, aunque los resultados finales no fueron los ideales, creemos que esto se debió más a una cuestión de tiempo y necesidad de ajustes finos que a una falta de dedicación.

Particularmente, sentimos que hubiera sido beneficioso contar con más tiempo para realizar pruebas más exhaustivas sobre la discretización del entorno, explorar en mayor profundidad el espacio de acciones, y afinar con mayor precisión los hiperparámetros clave, como α (alpha) y γ (gamma).

Además, valoramos especialmente el intercambio con ambos docentes, quienes nos brindaron orientación y abrieron las puertas a nuevas ideas y pruebas que nos ayudaron a comprender mejor el problema y las posibles estrategias de solución.

Board-Oriented Reasoning for Emergent Domination

Recorrido del desarrollo

Se implementó un agente utilizando el algoritmo Minimax para tomar decisiones óptimas contra el jugador trainer. Se usó poda alfa-beta para reducir la cantidad de estados explorados. También creamos un agente ExpectiMax con el objetivo de encontrar el mejor modelo entre ambos.

Implementación de Alpha-Beta Pruning sobre Minimax

Aplicar Alpha-Beta Pruning a la solución basada en Minimax permitió mejorar significativamente el rendimiento sin alterar la calidad de las decisiones. La principal consecuencia fue una reducción del número de nodos evaluados, lo que llevó a menor uso de recursos computacionales (CPU y memoria) y mayor velocidad de ejecución.

Gracias a esta optimización, el agente pudo explorar mayor profundidad en el mismo tiempo, tomando decisiones más informadas. Además, el resultado final se mantuvo igual al de Minimax puro, ya que Alpha-Beta solo optimiza la búsqueda, sin modificar la lógica de decisión.

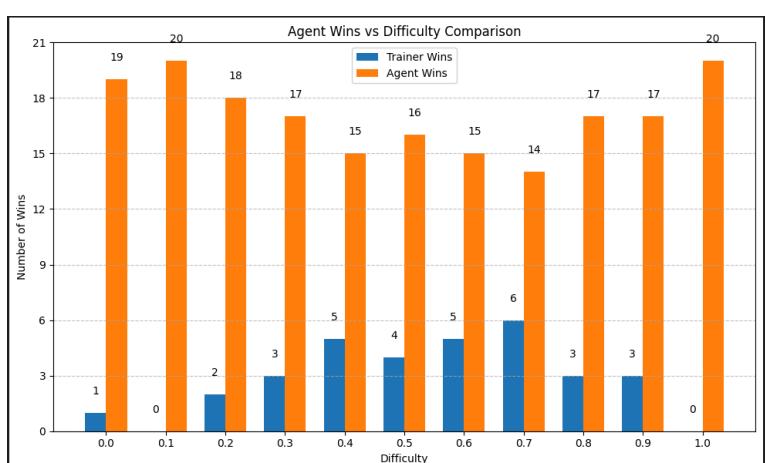
Función de utilidad inicial

La primera heurística que decidimos probar evalúa el estado actual del juego para determinar qué tan favorable es para el agente, mide la ventaja estratégica del agente en términos de opciones de movimiento, favoreciendo estados donde el agente tiene más posibilidades de acción que su oponente.

Primero implementamos el agente MiniMax con alpha-beta pruning evitando explorar ramas del árbol de búsqueda que no pueden influir en la decisión final. Luego implementamos el agente ExpectiMax con la misma heurística.

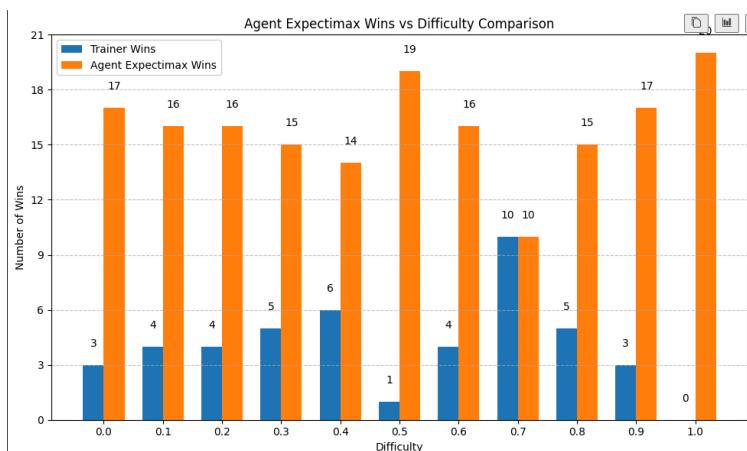
Prueba 1

Agente Minimax



Como vemos, la heurística fue muy buena, en la mayor dificultad le ganamos en todo al trainer. Sin embargo, cuando arrancan los movimientos aleatorios, decae un poco el éxito.

Agente Expectimax



Como se puede observar en el gráfico, se comporta peor el ExpectiMax.

Ahora vamos a agregar otra heurística y probar si mejora alguno de los modelos.

Segunda heurística agregada

La segunda heurística que decidimos agregar evalúa el estado del tablero contando la cantidad de segmentos largos, definidos como secuencias de tres o más fichas consecutivas en filas y columnas. Para cada fila y columna, recorre las celdas del tablero y suma al contador de segmentos largos cada vez que encuentra una secuencia válida. Al final, devuelve el total de segmentos largos encontrados.

Es clave para TacTix porque los segmentos largos ofrecen más opciones estratégicas al jugador, permitiendo seleccionar subsegmentos dentro de ellos y limitando las opciones del oponente. Además, priorizar estados con más segmentos largos ayuda a mantener el control del tablero y a manipular el flujo del juego, acercándose a un estado terminal favorable.

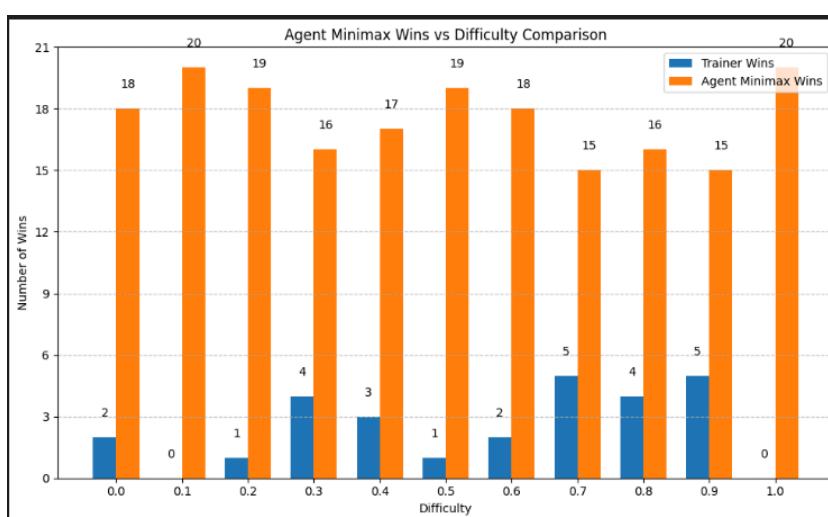
Prueba 2

Combinamos las heurísticas con la siguiente ponderación:

```
weight_1 = 0.7
weight_2 = 0.3

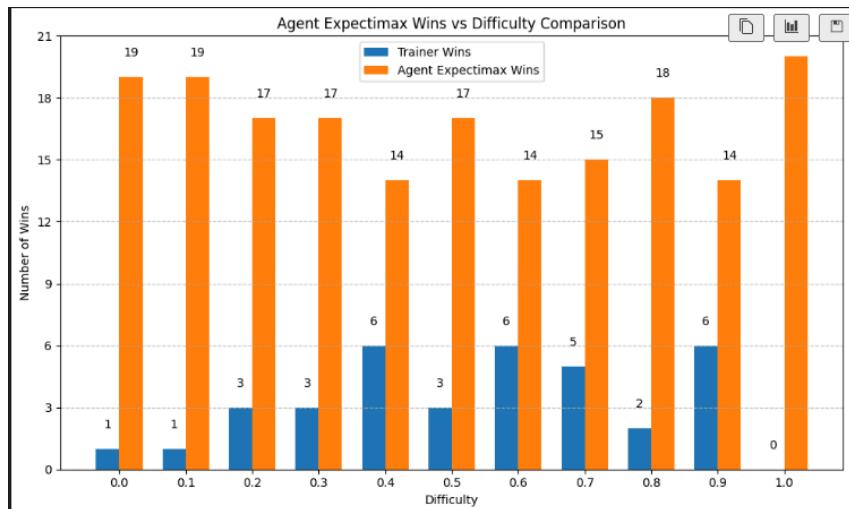
return weight_1 * heuristic_action_advantage + weight_2 * heuristic_segments
```

Agente Minimax



Comparado con el resultado anterior de minimax, vemos que tenemos menos partidos perdidos. Se ve una mejora en las dificultades medias, aunque, en 0.9 de dificultad pierde bastante.

Agente Expectimax



Como vemos, la segunda heurística pareció mejorar algún caso como con la dificultad 0.7 que antes solo el 50% de los juegos los ganaba. Sin embargo en otras dificultades ahora perdemos un poco más.

Conclusión: Ahora vamos a probar darle menos ponderación a la segunda heurística para que en dificultades altas, aun siga ganando más, y conservar un poco de lo bueno de la segunda heurística que mejora en las dificultades medias.

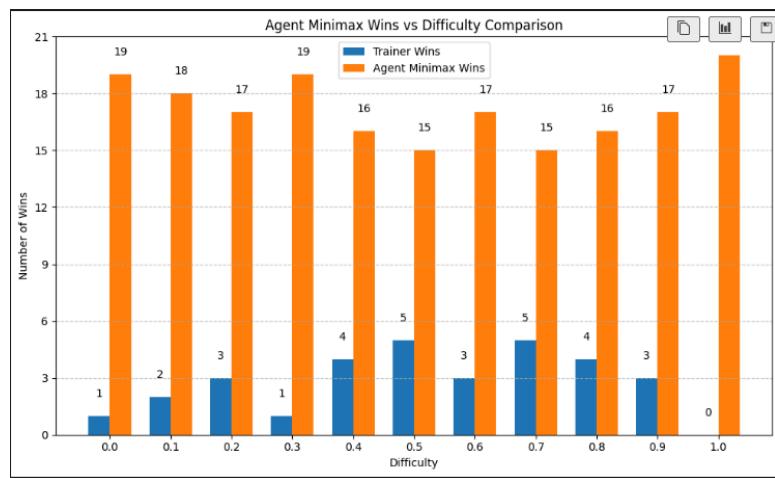
Prueba 3

Combinamos las heurísticas con la siguiente ponderación:

```
weight_1 = 0.85
weight_2 = 0.15

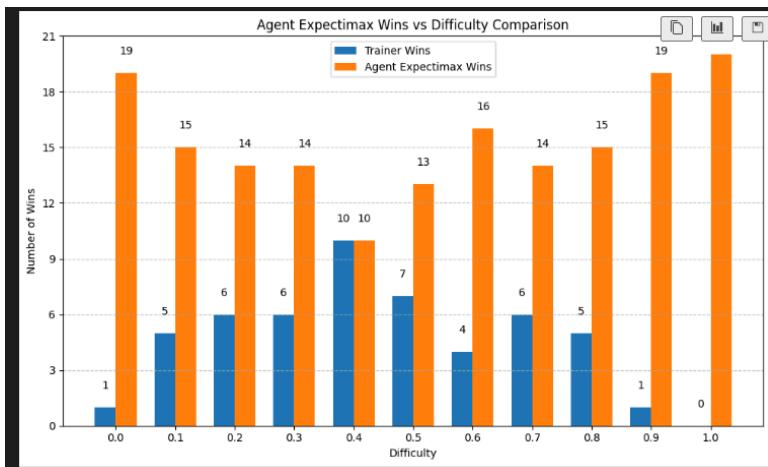
return weight_1 * heuristic_action_advantage + weight_2 * heuristic_segments
```

Agente Minimax



Vemos que se comporta como esperábamos, con los nuevos valores de las ponderaciones se logra en dificultades altas ganarle con mucha diferencia, y en el medio donde aparecen acciones randómicas, es normal que minimax pierda, pero de forma controlada, lejos de ganarle en la mayoría de los juegos.

Agente Expectimax



Para expectimax vemos que empeoro un poco esta ponderación de heurísticas. No pensamos que fuera a impactar negativamente sobre expectimax.

Verificamos que hayamos hecho correctamente el cambio en el código, y así lo fue. Así que simplemente para el algoritmo expectimax no sirvió.

Tercera heurística agregada (unique segments)

La heurística evalúa el estado del tablero contando la cantidad de segmentos únicos disponibles para el oponente. Un segmento único se define como una secuencia de exactamente una ficha consecutiva en filas o columnas. Su objetivo es identificar y reducir estas opciones predecibles del oponente, ya que son fáciles de eliminar y limitan su capacidad estratégica.

Esta heurística fue elegida porque complementa las existentes al enfocarse en minimizar las opciones más simples y directas del oponente, lo que permite al agente mantener un mayor control del tablero. Funciona recorriendo filas y columnas del tablero, contando los segmentos de una sola ficha, y devuelve el total de estos segmentos únicos.

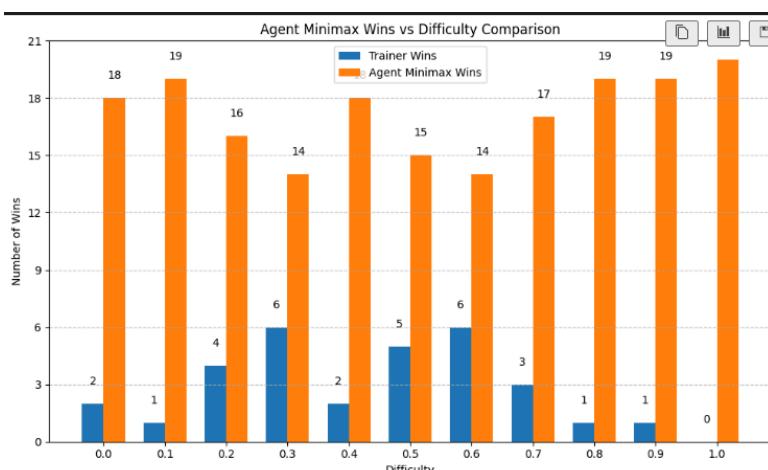
Prueba 4

Combinamos las heurísticas con la siguiente ponderación:

```
weight_1 = 0.6
weight_2 = 0.25
weight_3 = 0.15

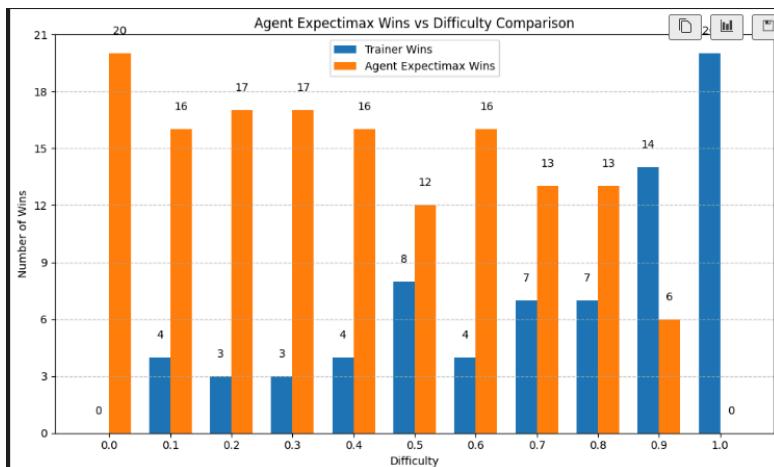
return (weight_1 * heuristic_action_advantage + weight_2 * heuristic_segments -
       weight_3 * heuristic_unique_segments) # Resta porque queremos reducir los segmentos únicos del oponente
```

Agente Minimax



Como vemos, el porcentaje de victoria mejora un poco para los casos de mayor dificultad, pero lo único que cambia es la distribución. Gana más en más dificultad pero pierde a menos dificultad que la prueba anterior.

Agente Expectimax



Como vemos, empeora totalmente el agente expectimax, pierde todas las partidas con la dificultad alta.

Descartamos la elección de la última heurística utilizada (unique segments) ya que simplemente cambia la distribución de como gana minimax respecto a las dificultades, y a expectimax lo empeora totalmente.

Cuarta heurística agregada (critical segments)

La heurística evalúa el estado del tablero identificando segmentos pequeños, definidos como secuencias de 1 o 2 fichas consecutivas en filas y columnas. Su objetivo es medir cuántos de estos segmentos vulnerables están disponibles para el oponente, ya que son más fáciles de eliminar y limitan sus opciones estratégicas. Esta heurística es útil porque permite al agente mantener el control del tablero al reducir las opciones simples del oponente, forzándolo a realizar movimientos desfavorables. Es especialmente relevante en las etapas finales del juego, donde los segmentos pequeños son más comunes y pueden influir directamente en el resultado.

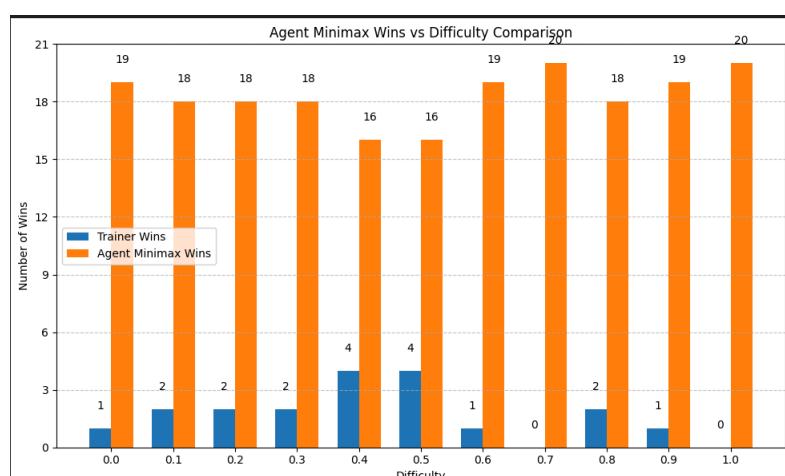
Prueba 5

Combinamos las heurísticas con la siguiente ponderación:

```
weight_1 = 0.6
weight_2 = 0.25
weight_3 = 0.15

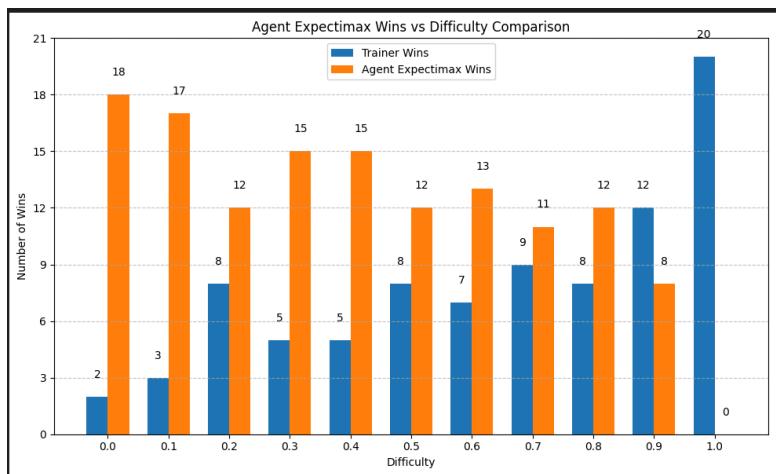
return (weight_1 * heuristic_action_advantage + weight_2 * heuristic_segments -
       weight_3 * heuristic_critical_segments) # Resta porque queremos reducir los segmentos criticos
```

Agente Minimax



Como vemos, mejoró abundante el resultado de minimax. Gana al menos un 80% en todas las dificultades. Perdió 19 partidas y ganó 201. En total gana el 91% de las partidas.

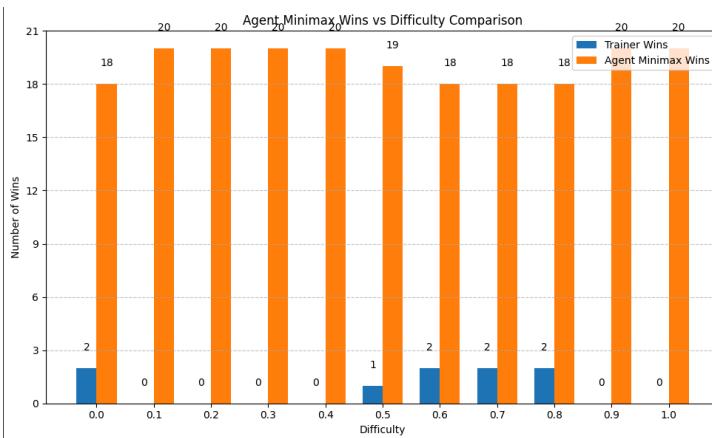
Agente Expectimax



Para el caso expectimax no funcionó la heurística, por lo tanto nos quedamos con el agente minimax.

Prueba 6

Como última prueba, vamos a probar aumentar la profundidad a 4 del agente Minimax para ver cómo se comporta. Debería mejorar un poco ya que evalúa de forma más precisa. Por otro lado, tiene más costo computacional pero esto lo contemplamos un poco con alpha-beta pruning.



Como vemos, mejoró bastante, ahora pierde solamente 9 partidas en 220 juegos.

Este modelo quedó adjuntado en formato .pkl en el [\[Repo\]](#)

Conclusión

El mejor agente logrado fue el MiniMax de la última prueba donde en promedio, sin importar la dificultad gana el 96% aproximadamente de las partidas, consideramos que es un muy buen resultado. Por falta de tiempo no seguimos probando combinaciones de ponderaciones que mejoren el resultado ni heurísticas.

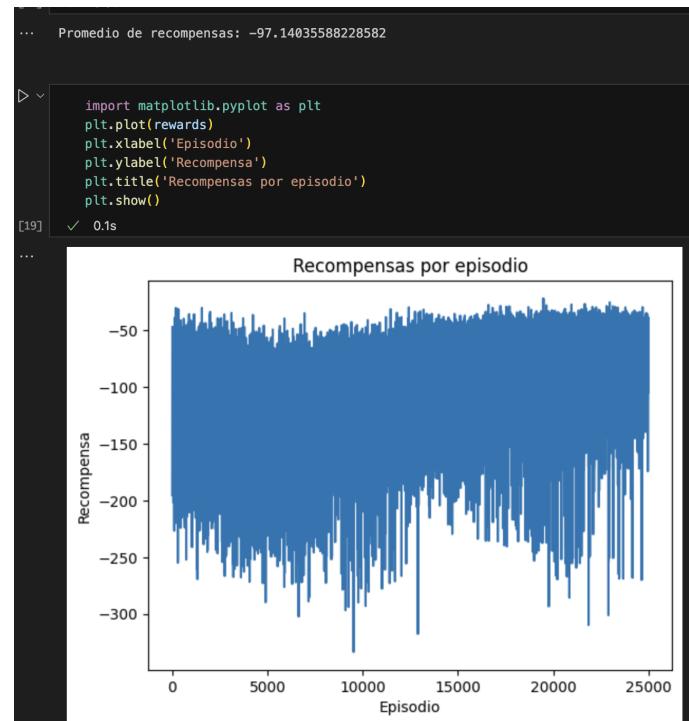
También se podría considerar aplicar memoización para ir guardando movimientos para situaciones dadas.

Pero consideramos que durante el proceso aprendimos bastante y probamos cosas que favorecieron los agentes y otras que no, llevándonos algo de experiencia en el manejo con agentes minimax y expectimax.

ANEXO

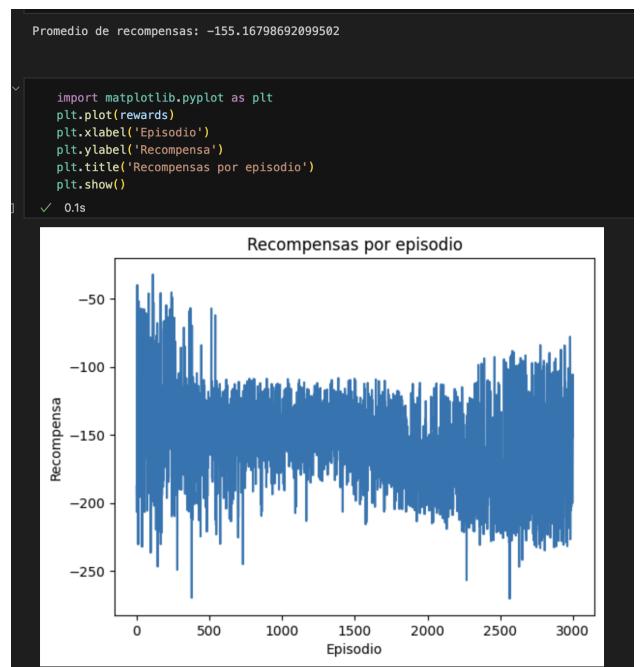
Prueba 1

- **Promedio de recompensas de prueba:** -97
- **Hiperparametros:** episodes=25000, epsilon=0.99, gamma=0.99, alpha=0.1
- **Conclusión:** Podemos observar muy poca mejora a medida que crecen los episodios, luego de los 15 arranca a bajar el aprendizaje, y luego a los 22 aproximadamente vuelve a mejorar el aprendizaje.



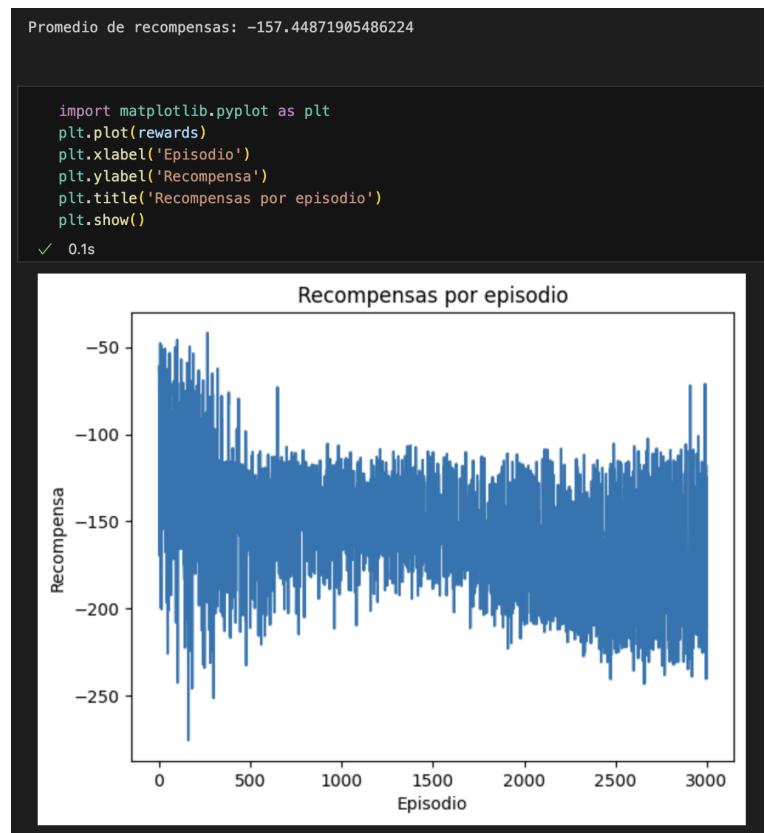
Prueba 2

- **Promedio de recompensas de prueba:** -155
- **Hiperparametros:** episodes=3000, epsilon=0.9, gamma=0.9, alpha=0.99
- **Conclusión:** Respecto a la prueba anterior aumentamos el valor de alpha casi a 1. Empero totalmente la gráfica así que descartamos la posibilidad de que aumentando el alpha mejore.



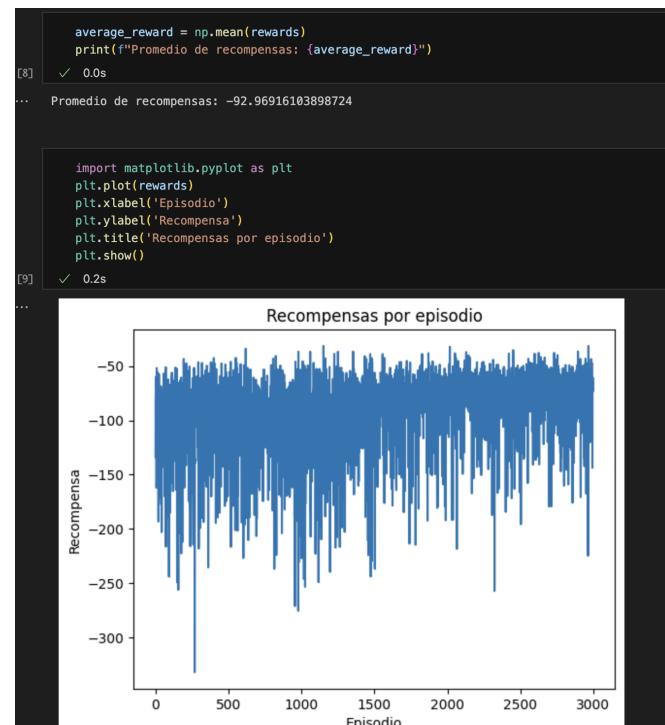
Prueba 3

- **Promedio de recompensas de prueba:** -157
- **Hiperparametros:** episodes=3000, epsilon=0.9, gamma=0.9, alpha=0.1
- **Conclusión:** Disminuimos un poco el epsilon y el gamma, ambos los bajamos de 0.99 a 0.9, y el alpha a 0.1 donde vimos una mejora notable. Sobre todo a la hora de que en los primeros episodios vemos que tienen valores prometedores de recompensa pero luego empeora. Sospechamos que sea por el alpha y que no este teniendo en cuenta las corridas anteriores, y hace que sea mas lento el aprendizaje..



Prueba 4

- **Promedio de recompensas de prueba:** -92
- **Hiperparametros:** episodes=3000, epsilon=0.99, gamma=0.99, alpha=0.7
- **Conclusión:** Decidimos probar volver al gamma que habiamos probado inicialmente, debido a que veiamos en la prueba anterior que obtenia recompensas mejores al inicio que al final, entonces deducimos que quizas faltaba exploracion.

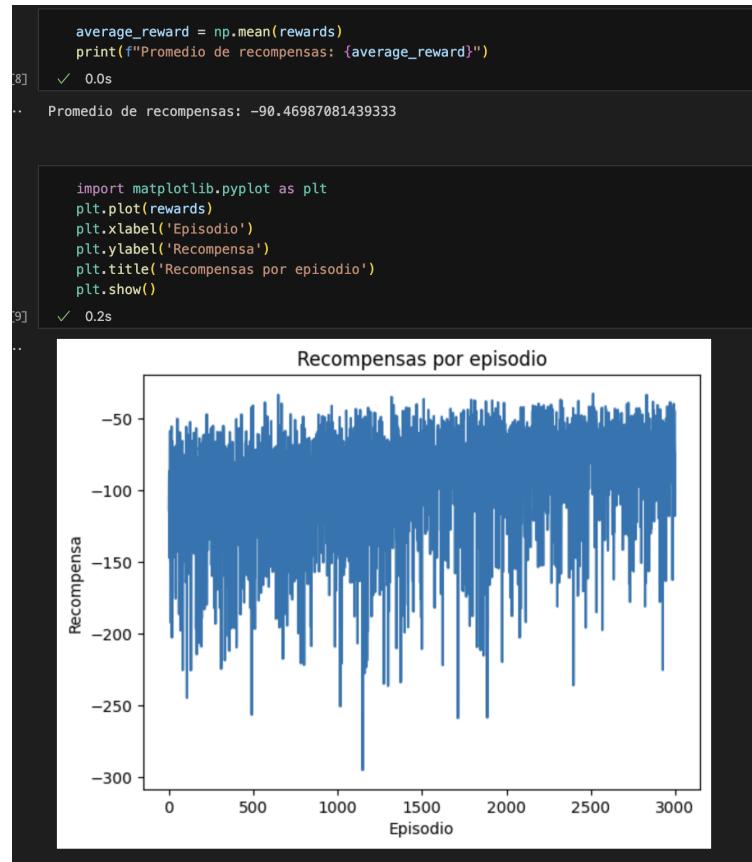


Prueba 5

- **Promedio de recompensas de prueba:** -90
- **Hiperparametros:** episodes=3000, epsilon=0.99, gamma=0.99, alpha=0.1
- **Conclusión:** Vemos una mejora notable donde la cota inferior de las recompensas por episodios tiende al valor más alto lentamente.

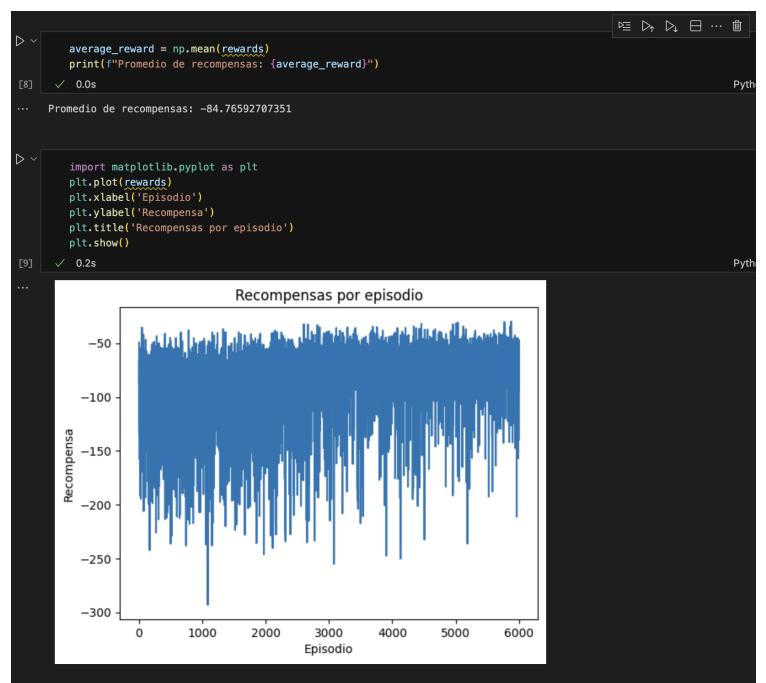
Se nos ocurre con mas episodios a ver que pasaría, e intentar corregir un poco en los episodios altos, las recompensas tan bajas. Intuimos que eso es debido a que quizás el alpha es muy grande y causa inestabilidad, debido al aprendizaje rápido.

Obs: En train dio resultado de 70



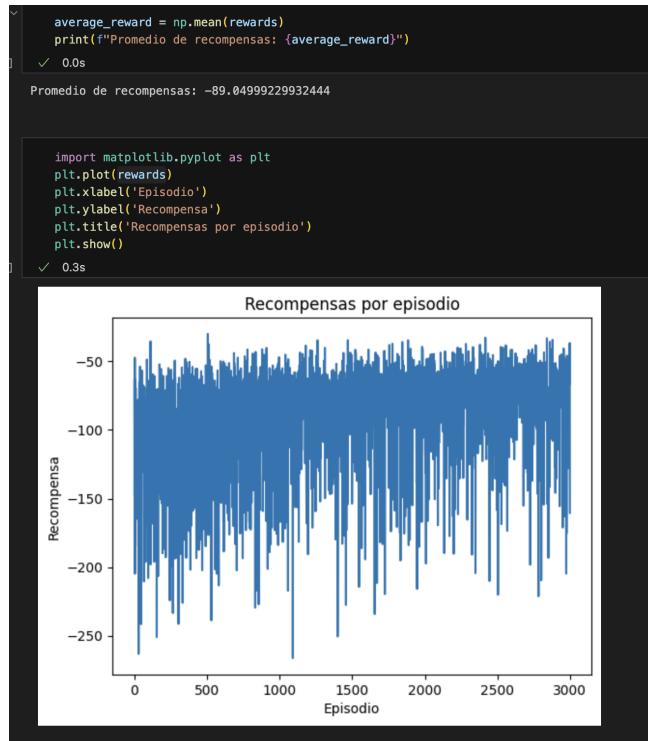
Prueba 6

- **Promedio de recompensas de prueba:** -84
- **Hiperparametros:** episodes=6000, epsilon=0.99, gamma=0.99, alpha=0.1
- **Conclusión:** Vemos una mejora que era como esperábamos y lo nombramos en la prueba 5, donde en el incremento de episodios el la diferencia de recompensa en altos episodios arranca a disminuir. Sin embargo, queremos corregir un poco las bajas recompensas en episodios altos.
- **Obs:** En test dio -59 la recompensa.



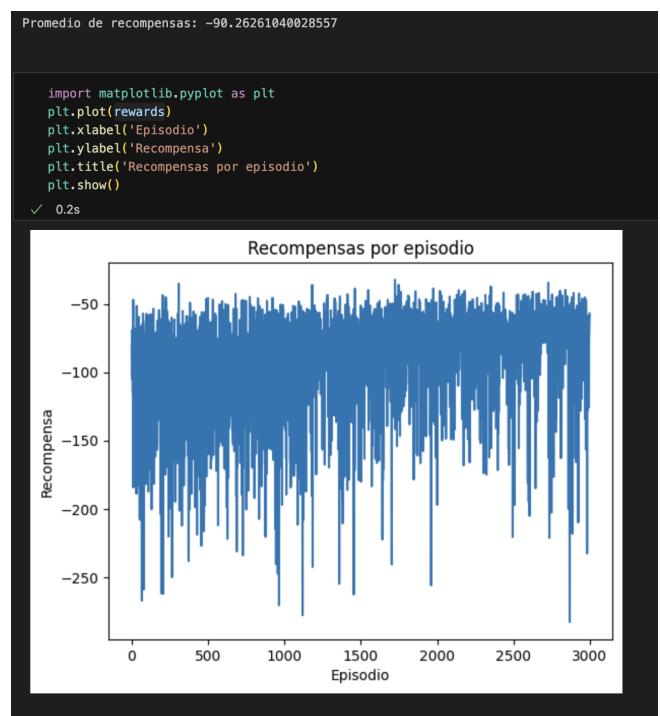
Prueba 7

- **Promedio de recompensas de prueba:** -89
- **Hiperparametros:** episodes=3000, epsilon=0.99, gamma=0.99, alpha=0.3
- **Conclusión:** Como dijimos en la prueba 5, probamos aumentar el alpha y vemos que durante los episodios aparecen más picos con recompensas mejores. De todas formas, somos conscientes que tener picos tan bajos de recompensa, nos tira abajo el promedio.



Prueba 8

- **Promedio de recompensas de prueba:** -90
- **Hiperparametros:** episodes = episodes=3000, epsilon=0.99, gamma=0.8, alpha=0.3
- **Conclusión:** Ahora probamos disminuir el gamma para ver que pasaba, y no hubo mucha mejora, quedo casi igual incluso empeorando las



Prueba 9

- **Promedio de recompensas de prueba:** -79
- **Hiperparametros:** episodes=6000, epsilon=0.99, gamma=0.8, alpha=0.3
- **Conclusión:** Dado que seguimos con el mismo problema de que aparecen recompensas muy bajas incluso en episodios altos, vamos a verificar que para todos los estados de la matriz Q tengamos suficiente información como para maximizar las acciones cuando baja epsilon a 0.01. Para esto planeamos hacer un mapa de calor de la tabla de q en función de los episodios para poder analizar si la taba está cargada cuando uso epsilon 0.1.
- **Obs:** En test -58

