Universidad Ort Uruguay
Facultad De Ingeniería

Obligatorio de Inteligencia Artificial

Sofia Barreto 258216
Martin Gulla 254564
Joaquin Sommer 184441

Indice

1. Mountain Car

1.1. Técnica de Q-Learning

Para el sistema de Mountain Car, utilizamos la técnica de aprendizaje por refuerzos Q-learning. Esta técnica de aprendizaje por refuerzos es modelo free y se basa en aprender una serie de reglas que le permitan al agente tomar la acción óptima para un estado dado. En el caso del Mountain Car, se busca aprender a manejar el vehículo y lograr que llegue a la posición objetivo.
A continuación, detallaremos la interacción con el simulador, los parámetros utilizados, el tiempo de ejecución y los resultados obtenidos.

1.2. Interacción con el simulador

La interacción con el simulador se basa en un ciclo en el cual, para cada episodio, el agente recibe un estado inicial del entorno que se convierte en un estado discreto, esto lo hacemos ya que el estado es continuo (el par posición, velocidad), precisamos discretizar para hacer que los datos entren en “buckets” para que el auto sepa qué acción tomar, si no discretización, es muy probable que el valor no exista en la tabla Q ya que es un valor muy específico. Durante cada episodio, el agente selecciona y realiza acciones, recibiendo recompensas y actualizando la Tabla Q en consecuencia. Cuando el agente alcanza el estado objetivo o termina el episodio sin lograr el objetivo, se inicia un nuevo episodio.

1.3. Parámetros utilizados

Tasa de Aprendizaje**(lr)**: Este valor, fijado en 0.1, determina cuánto aprende el agente de cada nueva recompensa. Un valor más alto haría que el agente diera más importancia a las recompensas más recientes, mientras que un valor más bajo haría que el aprendizaje fuera más lento y estable.

Factor de Descuento**(gamma):** Este valor, fijado en 0.95, determina cuánto valora el agente las recompensas futuras en comparación con las inmediatas. Un valor más alto haría que el agente valorará más las recompensas futuras.

Número de Episodios**(num_episodes):** Este es el número total de veces que el agente intentará alcanzar el objetivo desde el inicio. En este proyecto, se realizaron 5000 intentos.

Intervalo de Visualización**(display_interval):** Este parámetro establece cada cuántos episodios se mostrará información sobre el progreso del aprendizaje. En este proyecto, la información se muestra cada 1000 episodios.

Tamaño del Espacio Discreto**(discrete_size):** Este valor establece cuántos estados posibles considerará el agente. Dividimos el espacio de estados se dividió en una cuadrícula de 20x20.

1.3.1 Parámetros de Exploración

La capacidad del agente para explorar el entorno se rige por una serie de parámetros de exploración.
Al principio, cuando el agente no sabe mucho sobre el entorno, es beneficioso explorar: probar acciones al azar para descubrir cuáles pueden llevar a recompensas. Pero a medida que el agente aprende, es mejor explotar ese conocimiento, eligiendo las acciones que se sabe que conducen a recompensas.

Para gestionar este equilibrio entre exploración y explotación, se utiliza un enfoque llamado política epsilon-greedy. El parámetro "epsilon" controla la proporción de exploración frente a explotación.

Epsilon**(eps)**: Inicialmente, se establece un valor de 1 para epsilon, lo que significa que el agente comienza explorando completamente al azar.

Inicio y Fin del Decaimiento de Epsilon**(start/end_eps_decay)**: A lo largo de los episodios de entrenamiento, queremos que el valor de epsilon disminuya, de modo que el agente pase de la exploración a la explotación. Se establece un rango de episodios durante el cual epsilon disminuirá gradualmente. En este proyecto, el decaimiento comienza desde el primer episodio y finaliza en la mitad de los episodios totales.

Valor de Decaimiento de Epsilon**(eps_decay_val)**: Este es el valor en el que epsilon disminuirá en cada episodio durante el rango de decaimiento.

1.3.2 Tabla Q

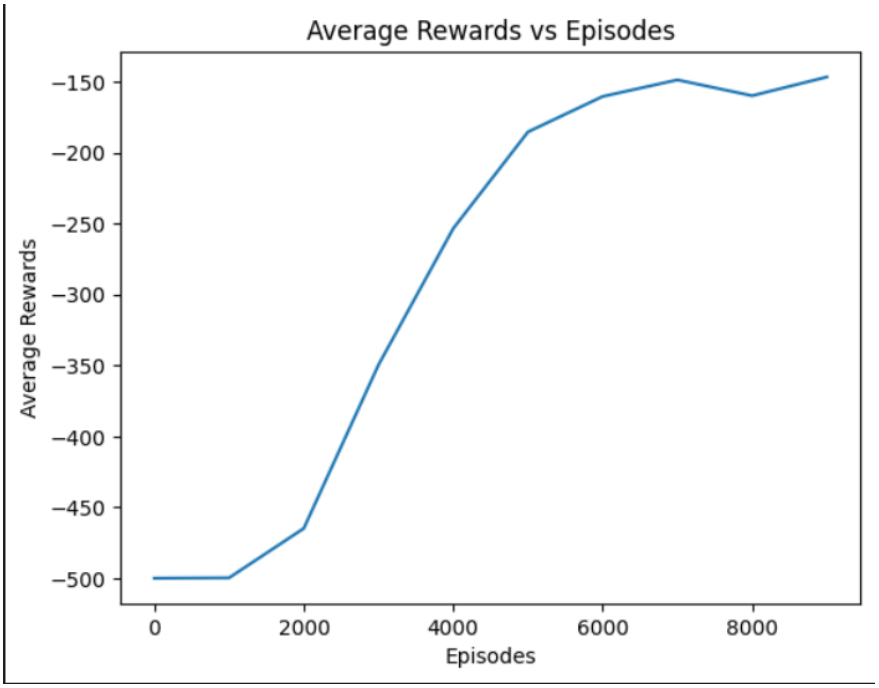
La Tabla Q es una estructura de datos esencial para el aprendizaje del agente. Cada entrada en la tabla corresponde a un par estado-acción y contiene un valor que representa la recompensa esperada para esa acción en ese estado.

Inicialmente, se establecen todos los valores en la tabla Q de manera aleatoria entre -2 y 0. La tabla Q tiene una dimensión que corresponde al tamaño del espacio de estados (en este caso, una matriz de 20x20 que representa posiciones y velocidades discretas) y una tercera dimensión correspondiente a las posibles acciones que el agente puede tomar.

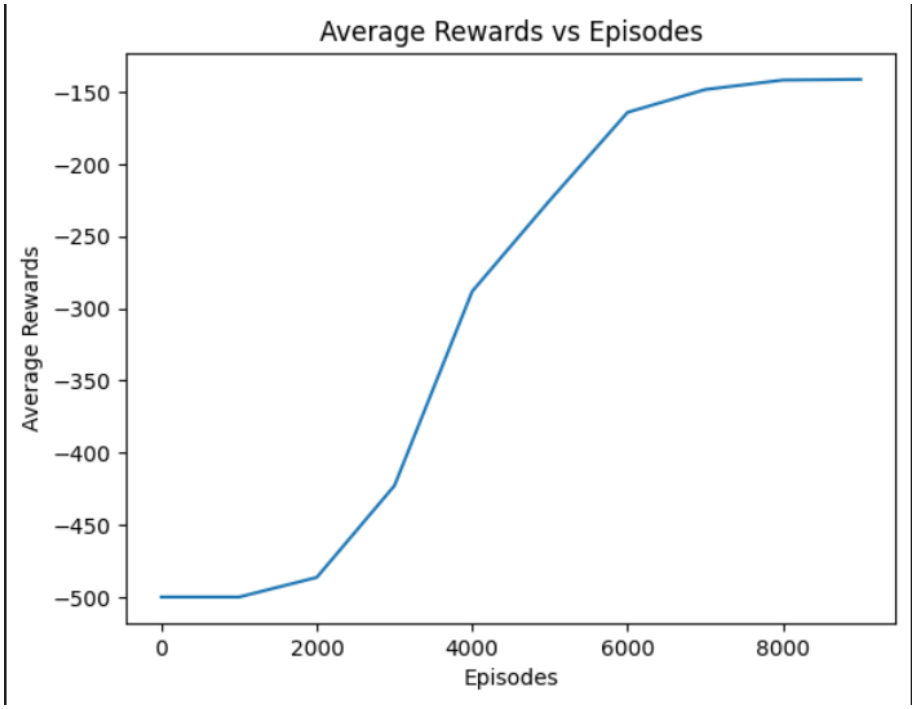
1.3.3 Parámetros utilizados en las iteraciones

Iteration	Learing Rate	Discount	Number of episodes	Discrete Size	Epsilon	Average reward
1	0.1	0.95	10000	[20,20]	1	149.8
2	0.2	0.95	10000	[20,20]	1	157.4
3	0.1	0.9	10000	[20,20]	1	134.4
4	0.1	0.95	10000	[20,20]	0.75	144.4
5	0.1	0.95	40000	[30,30]	1	112.4
6	0.2	0.97	40000	[30,30]	0.8	120.8

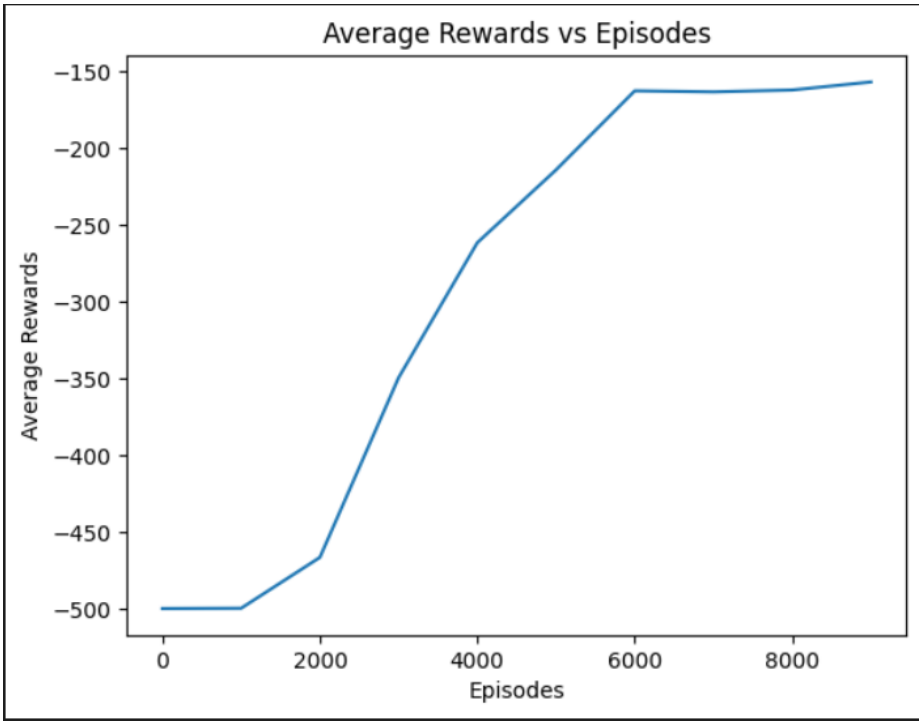
Iteration 1.



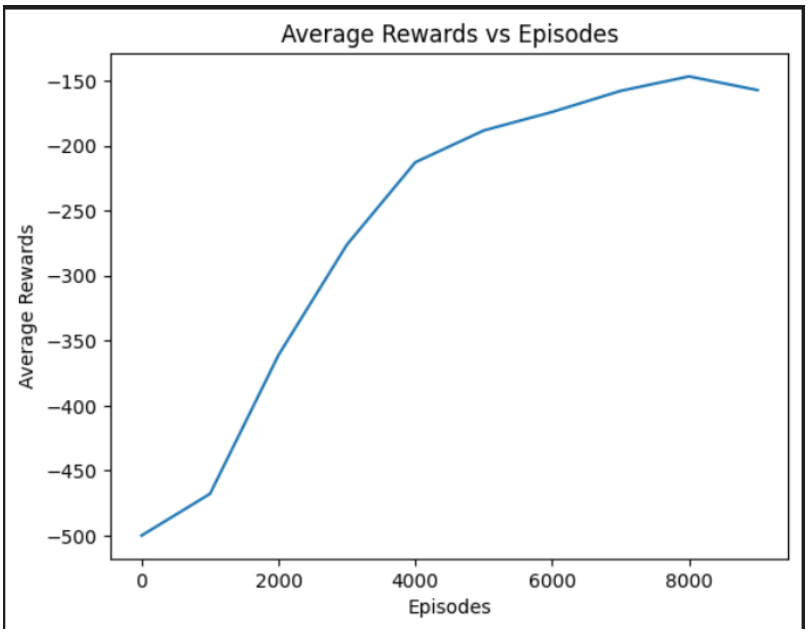
Iteration 2.



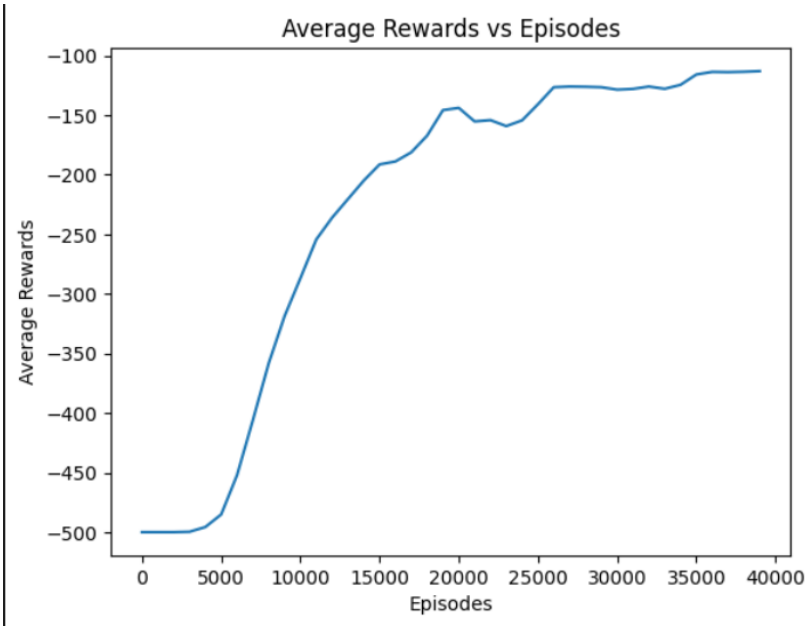
Iteration 3.



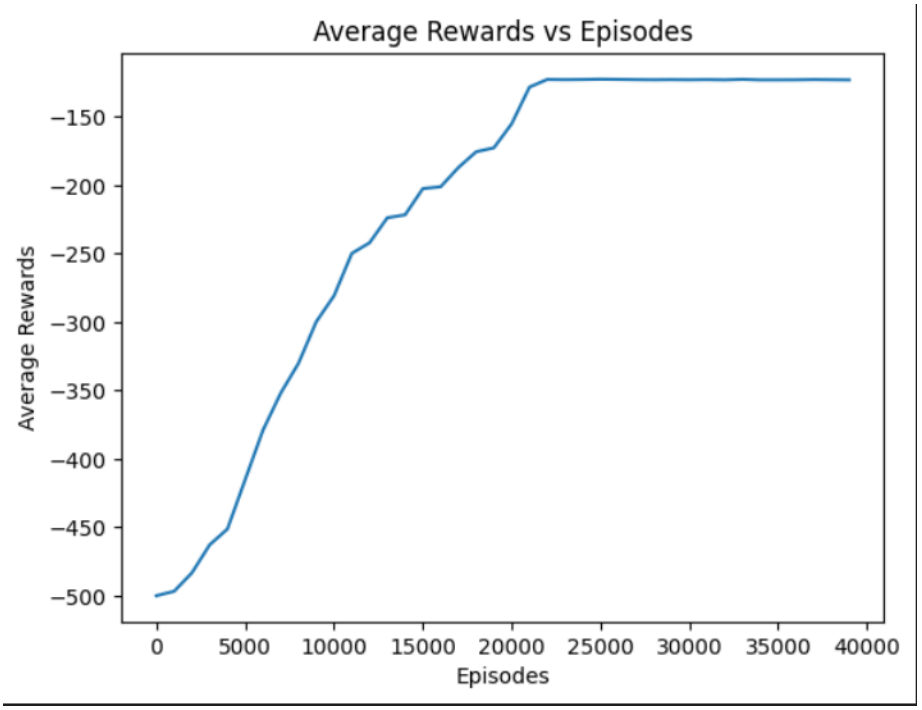
Iteration 4.



Iteration 5.



Iteration 6.



1.4. Resultados obtenidos

Tomaremos en cuenta la iteración 5, cuya política obtuvo la mejor performance de todas las que ejecutamos. Podemos ver que en 40000 episodios, la evaluación de la política obtuvo, en promedio, una recompensa de 112.4. Consideramos que el resultado obtenido fue altamente satisfactorio.

Train 3m 9.5s

```
# Iteration 5

env = MountainCarEnv(render_mode="rgb_array")
lr = 0.1
gamma = 0.95
num_episodes = 40000
display_interval = 1000
discrete_size = [30, 30]

# Exploration settings
eps = 1
start_eps_decay = 1
end_eps_decay = num_episodes // 2
eps_decay_val = eps / (end_eps_decay - start_eps_decay)

q_table = np.random.uniform(low=-2, high=0, size=(discrete_size + [env.action_space.n]))    Cannot access member "n" for type "Space[Unknown]"⏎  Member "n" is unknown

rewards, new_q_table = train_Q_car(env=env, lr=lr, gamma=gamma, num_episodes=num_episodes, display_interval=display_interval, discrete_size=discrete_size, eps=eps, q_table=q_table, iteration=5)

plt.plot(np.arange(0, num_episodes, display_interval), rewards)    "arange": Unknown word.
plt.xlabel("Episodes")    "xlabel": Unknown word.
plt.ylabel("Average Rewards")    "ylabel": Unknown word.
plt.title("Average Rewards vs Episodes")
plt.show()
✓ 3m 9.5s
```

Test 24.1s

```
env = MountainCarEnv(render_mode="human")
test_episodes = 5
rewards = test_q_learning_agent(env, q_table, test_episodes)
print(f"Average steps per episode: {np.mean(rewards)}")
✓ 24.1s
```

2. Sistema Anti-Aburrimiento Connect-Four

2.1. Técnica Expectimax

La técnica Expectimax está diseñada para manejar incertidumbre y aleatoriedad en la toma de decisiones de los jugadores o en el entorno. Utilizamos la técnica Expectimax para el juego Connect Four, donde el jugador 1 es el jugador que maximiza y el jugador 2 es el "jugador de azar" (Chance player).

El "jugador de azar" es un jugador que toma decisiones de manera probabilística en lugar de tratar de minimizar la utilidad del oponente. En el algoritmo, el "jugador de azar" calcula la utilidad esperada promediando las utilidades de los nodos hijos, ponderadas por la probabilidad de cada acción.

La función `expectimax` alterna entre capas de maximización y capas de azar en el árbol de búsqueda. En las capas de maximización, el agente intenta maximizar su utilidad, mientras que en las capas de azar, el agente calcula la utilidad esperada promediando las utilidades de los nodos hijos.

2.2. Técnica Minmax

El algoritmo Minimax está diseñado para juegos de dos jugadores con información perfecta, donde ambos jugadores toman decisiones racionales y tratan de maximizar su propia utilidad. Utilizamos la técnica Minimax para el juego Connect Four, donde el jugador 1 es el jugador que maximiza y el jugador 2 es el jugador que minimiza.

En el algoritmo Minimax, ambos jugadores asumen que su oponente juega de manera óptima y toma decisiones racionales. El jugador que maximiza intenta maximizar su utilidad, mientras que el jugador que minimiza intenta minimizar la utilidad del jugador que maximiza.

La función `minimax` alterna entre capas de maximización y capas de minimización en el árbol de búsqueda. En las capas de maximización, el agente intenta maximizar su utilidad seleccionando la acción con la mayor utilidad de los nodos hijos. En las capas de minimización, el agente asume que su oponente seleccionará la acción con la menor utilidad (la que minimiza la utilidad del agente) de los nodos hijos.

El algoritmo Minimax explora exhaustivamente el árbol de búsqueda hasta una profundidad específica, evaluando los estados finales del juego utilizando una función heurística de utilidad. Luego, propaga estos valores de utilidad hacia arriba en el árbol de búsqueda, alternando entre maximizar y minimizar la utilidad en cada nivel. Al final, el algoritmo elige la acción que conduce al mayor valor de utilidad para el jugador que maximiza, asumiendo que el oponente juega de manera óptima.

2.3. Parámetros utilizados

Nuestros algoritmos de poseen, en total, cuatro hiperparametros:

max_depth: Límite de profundidad con la cual se evalúa el valor de los estados sucesores.

line_weight: Peso otorgado a la heurıstica de count_lines, al cual se la multiplica.

potential_win_weight: Peso otorgado a la heurıstica de count_potential_wins, al cual se la multiplica.

sandwich_weight: Peso otorgado a la heurıstica de count_sandwiches, al cual se la multiplica.

Los valores finales para ellos fueron los siguientes:

- max_depth (expectimax) = 3
- max_depth (minimax) = 7
- line_weight = 10
- potential_win_weight = 100
- sandwich_weight = 5

Estos valores, óptimos para nuestros agentes Expectimax y Minimax, pues son los que mejores resultados presentaron de todos los que probamos, fueron resultado de buscar la mejor combinación de hiperparametros.
Algunas de las técnicas que usamos para optimizar estos hiperparametros fueron:

- Observar los estados perdedores del agente y determinar las causas de estos. Por ejemplo, vimos que muchas veces el agente perdía por no hacer un “sandwitch” o no intentaba hacerlo, como solución agregamos la heurística “count_sandwiches”.
- También, intentamos quitar y agregar nuevas heurísticas como: cantidad de movimientos disponibles, cantidad de fichas para cada jugador y el grupo más grande de fichas. En todos estos casos, el resultado empeoró, por lo que decidimos eliminarlas.
- Observar la cantidad total de movimientos con la que el agente perdía/ganaba el juego. Si veíamos que el agente perdía y además lo hacía con una cantidad de total movimientos muy baja, entonces esto significaba que la combinación de hiperparametros era mala dado que el agente ni había sido capaz de acercarse a un potencial estado ganador.

2.2. Heurísticas utilizadas

Para evaluar el valor del tablero en un estado dado, y así preferir ciertos estados frente a otros, combinamos tres heurísticas:

count_lines: Esta heurística cuenta el número de líneas que un jugador tiene en el tablero. Para hacerlo, recorre cada celda del tablero y, si encuentra una ficha del jugador, verifica si forma una línea de cuatro casillas en cualquier dirección (horizontal, vertical, diagonal).

count_potential_wins: Esta heurística cuenta el número de posibles victorias que un jugador tiene en el tablero. Al igual que con el conteo de líneas, esta función recorre cada celda del tablero y, si encuentra una ficha del jugador, verifica si forma una posible victoria en cualquier dirección.

count_sandwiches: Un "sándwich" se define como una ficha del jugador que está entre dos fichas del oponente, en dirección horizontal o diagonal (no vertical). Esta función cuenta el número de "sándwiches" que un jugador tiene en el tablero. Para hacerlo, recorre cada celda del tablero y, si encuentra una ficha del jugador, verifica si es posible formar un "sándwich" en cualquier dirección.

2.3. Se les pide hacer jugar a ambos agentes analizando los resultados

Vamos a realizar varias pruebas, primero cada modelo contra el proporcionado con el resto del código (SpaceGPT), luego entre ellos. Estas pruebas van a ser 10 partidas de “ida” y 10 de “vuelta” contra SpaceGPT, es decir, vamos a ir alternando quien empieza el juego, ya que esto influye en gran medida quien es el ganador, dado a que este no tiene que defenderse tanto como el otro de posibles victorias. Entre ellos, seran 20 partidas de ida y vuelta.

Jugador 1 (primero)	Jugador 2 (segundo)	Victoria
Minimax	SpaceGPT	10-0
SpaceGPT	Minimax	10-0
Expectimax	SpaceGPT	10-0
SpaceGPT	Expectimax	10-0
Minimax	Expectimax	10-0
Expectimax	Minimax	10-0

Siempre el que gana es el que va primero.