Universidad ORT Uruguay

Facultad de Ingeniería

Inteligencia Artificial

Obligatorio

Clavijo, Tomás (235426)

Oller, Germán (242312)

Docente: Federico Armando

2023

Mountain Car

Explicación del problema

El MDP del Coche en la Montaña es un MDP determinista que consiste en un coche ubicado de manera estocástica en el fondo de un valle sinusoidal, donde las únicas acciones posibles son las aceleraciones que se pueden aplicar al coche en cualquier dirección. El objetivo del MDP es acelerar estratégicamente el coche para llegar al estado objetivo en la cima de la colina derecha. Hay dos versiones del dominio del coche en la montaña en Gym: una con acciones discretas y otra con acciones continuas. En este caso, nos referimos a la versión con acciones discretas.

Q - Learning

El Q-learning es un algoritmo de aprendizaje por refuerzo que se utiliza para encontrar una política óptima en un entorno de toma de decisiones basado en un MDP. En el contexto del Mountain Car, el Q-learning sería una opción para resolver el problema de encontrar la mejor estrategia de aceleración para llegar a la cima de la colina.

El objetivo del Q-learning es aprender una función de valor, llamada Q-función, que asigna un valor a cada par estado-acción. Esta función representa la utilidad esperada de tomar una acción específica en un estado dado. El algoritmo utiliza un proceso iterativo de actualización de la Q-función para ir mejorando las estimaciones de los valores de Q a medida que se exploran y se obtienen recompensas del entorno.

En el caso del Mountain Car, utilizamos Q-learning porque es un algoritmo capaz de encontrar soluciones óptimas en entornos complejos y con acciones discretas. El algoritmo podría explorar y aprender la mejor estrategia para acelerar el coche de manera eficiente y alcanzar la cima de la colina, maximizando la recompensa obtenida en el proceso. A través de iteraciones, el Q-learning puede mejorar gradualmente su política hasta encontrar la estrategia óptima para resolver el problema del Mountain Car.

Exploit - Explore

En el contexto del aprendizaje por refuerzo, las etapas de "exploit" (explotar) y "explore" (explorar) son dos componentes importantes para encontrar un equilibrio entre la explotación de acciones conocidas y la exploración de acciones desconocidas.

Etapa de Exploit (Explotar): Durante esta etapa, el agente utiliza el conocimiento adquirido hasta el momento para seleccionar las acciones que se consideran óptimas según la información actual. El objetivo principal en esta etapa es maximizar la recompensa obtenida a corto plazo. El agente elige las acciones que se espera que proporcionen los mayores beneficios basados en su conocimiento previo, lo que le permite aprovechar al máximo las acciones que ha aprendido que son efectivas.

Etapa de Explore (Explorar): En contraste con la etapa de exploit, durante la etapa de explore, el agente se centra en descubrir nuevas acciones y explorar el entorno para obtener más información. Durante esta fase, el agente toma acciones no necesariamente óptimas de acuerdo con su conocimiento actual, con el objetivo de recolectar datos y mejorar su modelo del entorno. Esto implica tomar acciones aleatorias o menos frecuentes para descubrir posibles acciones que puedan llevar a mejores recompensas a largo plazo.

La combinación de ambas etapas es crucial para lograr un aprendizaje efectivo. La etapa de exploit permite al agente aprovechar su conocimiento actual y maximizar las recompensas a corto plazo. Por otro lado, la etapa de explore permite al agente descubrir nuevas acciones y recopilar información adicional para mejorar su conocimiento y encontrar soluciones óptimas a largo plazo.

En el contexto específico del Mountain Car, el agente utilizará la etapa de exploit para aplicar las acciones que ha aprendido que son más efectivas para alcanzar la cima de la colina. Al mismo tiempo, durante la etapa de explore, el agente probaría

acciones no tan óptimas o incluso aleatorias para explorar diferentes estrategias y descubrir si existen acciones más eficientes o caminos alternativos hacia el objetivo.

```python
def epsilon_greedy_policy(state, Q, epsilon=0.1):
    explore = np.random.binomial(1, epsilon)
    if explore:
        action = env.action_space.sample()
        print('explore')
    # exploit
    else:
        action = np.argmax(Q[state])
        print('exploit')
    return action
```

Figura 1: Epsilon policy

Como se observa en la imagen, el parámetro épsilon en la función *epsilon_greedy_policy* indica la probabilidad de exploración en el algoritmo de Q-Learning. Determina la proporción de veces en las que el agente elige una acción al azar (exploración) en lugar de elegir la acción con el valor máximo de Q (explotación).

Cuando epsilon es igual a 0, la política es completamente greedy, lo que significa que el agente siempre elige la acción con el valor máximo de Q. Por otro lado, cuando epsilon es igual a 1, la política es completamente aleatoria y el agente siempre elige acciones al azar.

En general, se utiliza un valor de epsilon entre 0 y 1, que permite un equilibrio entre la exploración y la explotación. Un valor comúnmente utilizado es 0.1, lo que significa que el agente elige una acción al azar el 10% del tiempo y elige la acción óptima el 90% del tiempo.

En el caso de nuestro trabajo, hemos optado por aplicar la lógica de la política epsilon - greedy utilizando la función *max_action*. En nuestro enfoque, utilizamos la

expresión *action = np.random.choice([0,1,2]) if np.random.random() < eps else max_action(Q, state)* para implementar esta política.

Esta expresión nos permite tomar decisiones de exploración y explotación de manera equilibrada. En cada iteración, evaluamos si un número aleatorio generado por *np.random.random()* es menor que el valor de *eps*, que representa nuestra probabilidad de exploración. Si este resultado es verdadero, seleccionamos una acción al azar de las opciones disponibles *[0,1,2]* mediante np.random.choice. Esto nos brinda la oportunidad de explorar el entorno de manera aleatoria.

Por otro lado, si el resultado de la evaluación es falso, aplicamos la función *max_action(Q, state)* para seleccionar la acción óptima basada en los valores almacenados en la matriz Q. Esta estrategia de explotación nos permite aprovechar el conocimiento acumulado y elegir la acción que se considera más beneficiosa en función de la información disponible.

Al combinar la selección aleatoria de acciones con la elección óptima basada en la función *max_action*, estamos equilibrando la exploración y la explotación en nuestro enfoque, permitiendo así un aprendizaje más eficaz en entornos de toma de decisiones basados en recompensas.

```python
def max_action(Q, state, actions=[0, 1, 2]):
    values = np.array([Q[state,a] for a in actions])
    action = np.argmax(values)

    return action
```

Figura 2: Max_action

```python
while not done:
    action = np.random.choice([0,1,2]) if np.random.random() < eps \
        else max_action(Q, state)
```

Figura 3: Condición

Discretización de estados

Como se ha podido observar en nuestra implementación, hemos parametrizado las acciones entre 0, 1 y 2, debido a cómo están definidas en el entorno Mountain Car de Gym. El espacio de acciones consta de 3 acciones discretas determinísticas, que se describen de la siguiente manera:

Num	Observation	Value	Unit
0	Accelerate to the left	Inf	position (m)
1	Don't accelerate	Inf	position (m)
2	Accelerate to the right	Inf	position (m)

Figura 4: Discretización de estados

Al parametrizar las acciones en nuestro código como [0, 1, 2], nos aseguramos de que las acciones seleccionadas se correspondan correctamente con las acciones definidas en el entorno. Esto nos permite interactuar adecuadamente con el mismo y tomar decisiones basadas en las acciones permitidas por el espacio de acciones del problema. Estamos logrando que nuestras acciones se ajusten a las opciones válidas y específicas del entorno, lo que permite un control adecuado del comportamiento del agente durante el entrenamiento y la toma de decisiones.

Epsilon variable

En el código que realizamos, decidimos incorporar una estrategia de épsilon variable para entrenar nuestro agente en el entorno Mountain Car. En cada iteración del bucle de entrenamiento, actualizamos el valor de épsilon utilizando la expresión eps = eps - 2/n_games if eps > 0.01 else 0.01.

```
eps = eps - 2/n_games if eps > 0.01 else 0.01
```

Figura 5: Epsilon variable

La elección de esta estrategia se basa en el equilibrio entre la exploración y la explotación. Al principio del entrenamiento, cuando nuestro agente no tiene conocimiento previo del entorno, es crucial que explore diferentes acciones para aprender de los resultados. Por lo tanto, utilizamos un valor inicial alto de épsilon (1), lo que nos garantiza una alta probabilidad de exploración y nos permite descubrir nuevas estrategias.

Conforme nuestro agente adquiere más conocimiento y experiencia, reducimos gradualmente el valor de épsilon. Esto aumenta la probabilidad de que nuestro agente elija acciones óptimas basadas en los valores de la matriz Q que ha aprendido hasta ese momento. A medida que disminuye el valor de épsilon, el agente tiende a explotar más su conocimiento acumulado y toma decisiones más informadas.

Esta estrategia de épsilon variable nos ayuda a lograr un equilibrio entre la exploración inicial y la explotación posterior, lo que resulta en un agente más eficiente y con un mejor rendimiento a medida que avanza el entrenamiento. Nuestro objetivo es lograr que nuestro modelo de agente pueda obtener buenos resultados de manera consistente en diferentes ejecuciones.

En resumen, en nuestro código, implementamos la estrategia de épsilon variable para adaptar la tasa de exploración de nuestro agente en el entorno Mountain Car.

Esto nos permite ajustar el equilibrio entre la exploración y la explotación durante el proceso de entrenamiento, lo que mejora el rendimiento general del agente y su capacidad para aprender y tomar decisiones óptimas en el entorno.

Linspace

Como se ha mencionado anteriormente, en este juego, el objetivo es controlar un automóvil para que suba una colina empinada. Para esto, debemos definir los espacios de observación del juego.

```
pos_space = np.linspace(-1.2, 0.6, 12)
vel_space = np.linspace(-0.07, 0.07, 20)
```

Figura 6: Definición del Linspace

En el código, se utiliza la biblioteca de Python numpy

La primera línea de código que se observa define un espacio de observación para la posición del automóvil. El juego tiene una observación baja y una observación alta para la posición, siendo -1,2 y 0,6 respectivamente, ambos inclusive.
La función np.linspace se utiliza para crear un arreglo de valores igualmente espaciados dentro de un rango especificado. En este caso, se genera un arreglo pos_space con 12 valores que representan las posibles posiciones del automóvil en el juego.

La segunda línea de código define un espacio de observación para la velocidad del automóvil. Al igual que con la posición, se utiliza la función np.linspace para generar un arreglo vel_space con 20 valores que van desde -0,07 hasta 0,07, ambos inclusive. Estos valores representan las posibles velocidades del automóvil en el juego.

En resumen, estos espacios de observación se utilizan para representar las posibles combinaciones de posición y velocidad del automóvil en el juego.

Parámetros

En el algoritmo de aprendizaje por refuerzo llamado Q-Learning para entrenar un agente en el entorno del juego se utilizan diversos parámetros. Uno de ellos, el epsilon, ya fue mencionado anteriormente, sin embargo, también se utilizaron otros como alpha y gamma.

Tasa de aprendizaje (alpha): El parámetro controla en qué medida el agente debe actualizar sus estimaciones de Q - valor en función de las recompensas y las estimaciones actuales. Un valor de alpha bajo significa que el agente aprende lentamente y ajusta sus estimaciones gradualmente, mientras que un valor alto permite actualizaciones más rápidas y ajustes más significativos. En nuestro caso, decidimos utilizar un valor de alpha = 0,1, esto indica que el agente aprende de manera moderada y realiza ajustes sustanciales en sus estimaciones. Consideramos que es una elección razonable si se desea un equilibrio entre la estabilidad del aprendizaje y la capacidad de adaptación a nuevas situaciones.

Factor de descuento (gamma): El parámetro gamma determina la importancia relativa de las recompensas futuras en relación con las recompensas inmediatas. Un valor de gamma cercano a 1 implica un enfoque de largo plazo, donde el agente considera las recompensas futuras de manera más significativa, mientras que un valor cercano a 0 da más peso a las recompensas inmediatas y promueve un enfoque más oportunista. Un valor de gamma = 0.99 indica que el agente tiene una perspectiva de largo plazo y valora las recompensas futuras de manera importante, pero no de manera excesiva. Esto puede ser beneficioso para el juego Mountain Car, donde el agente necesita tener en cuenta las consecuencias a largo plazo de sus acciones, como generar suficiente impulso para superar la colina.

Cantidad de episodios: La cantidad de episodios puede variar dependiendo de varios factores, como la complejidad del entorno y la dificultad del problema en sí. Sin embargo, en general, se sugiere ejecutar múltiples episodios hasta que el agente aprenda una política lo suficientemente buena.
En el algoritmo Q-learning, el agente actualiza iterativamente los valores Q en función de las recompensas recibidas y las acciones tomadas en cada episodio. A

medida que el agente explora y aprende más sobre el entorno, sus estimaciones de los valores Q se vuelven más precisas, lo que le permite tomar mejores decisiones.
En el caso del Mountain Car, que es un problema de control de movimiento donde un automóvil debe aprender a subir una colina empinada, se puede comenzar con un número bajo de episodios, 100 o 200, y observar cómo el agente progresa. Luego, se pueden ajustar los parámetros y ejecutar más episodios si es necesario.
En nuestro caso fuimos de menos a más, finalizando con un total de 10.000 episodios; logrando así, experimentar con diferentes cantidades de episodios y monitorear el progreso del agente.

Resultados

Durante la ejecución del algoritmo Q-Learning, se tomó la decisión de imprimir los valores del puntaje obtenido cada 1000 episodios. Este enfoque se adoptó con el propósito de visualizar el progreso del aprendizaje. La siguiente figura muestra los resultados obtenidos:

```
episode 1000 score -217.0 epsilon %.3f 0.01
episode 2000 score -160.0 epsilon %.3f 0.01
episode 3000 score -154.0 epsilon %.3f 0.01
episode 4000 score -194.0 epsilon %.3f 0.01
episode 5000 score -184.0 epsilon %.3f 0.01
episode 6000 score -121.0 epsilon %.3f 0.01
episode 7000 score -157.0 epsilon %.3f 0.01
episode 8000 score -152.0 epsilon %.3f 0.01
episode 9000 score -212.0 epsilon %.3f 0.01
```

Figura 7: Impresión de ejecución Q-learning

Además, al finalizar la ejecución, se solicitó generar un gráfico que representara el proceso de aprendizaje, con el fin de verificar su correcto funcionamiento. Encontrándose la policy óptima en torno al valor -150.

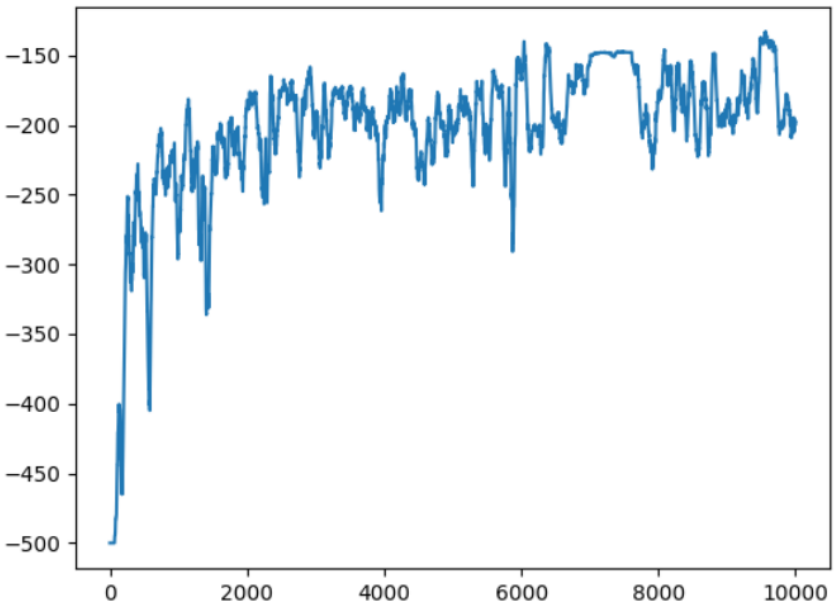


Figura 8: Gráfica (x = episodios, y= score)

Asimismo, para obtener una comprensión más práctica y visual del desempeño del agente, se utilizó el visualizador proporcionado por la biblioteca "gym". Este visualizador permitió observar de manera interactiva si el automóvil lograba alcanzar la cima y cómo se desarrollaba su proceso de aprendizaje. Se puede observar a continuación:

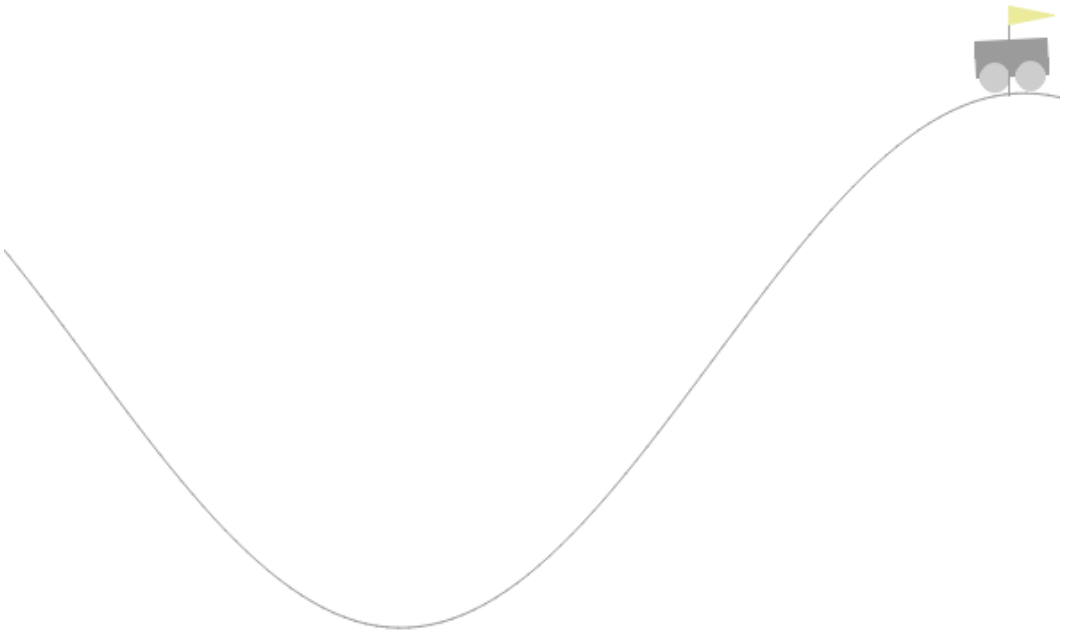


Figura 9: Representación del desafío

Estas medidas fueron implementadas con el objetivo de monitorear y evaluar el rendimiento del agente en el entorno del Mountain Car, facilitando la comprensión de su progreso y asegurando el funcionamiento adecuado del algoritmo Q-Learning.

Conclusión

Observando la gráfica obtenida al final de la ejecución concluimos que el objetivo se cumplió, ya que como se puede observar el aprendizaje fue evolucionando y comportándose de manera esperada, como se muestra a continuación:

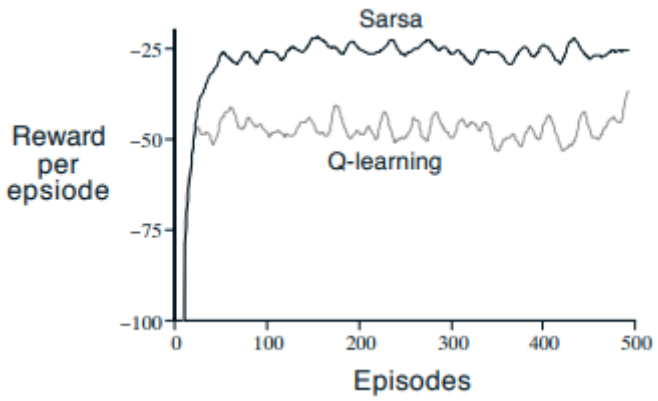


Figura 10: Gráfico Q-learning

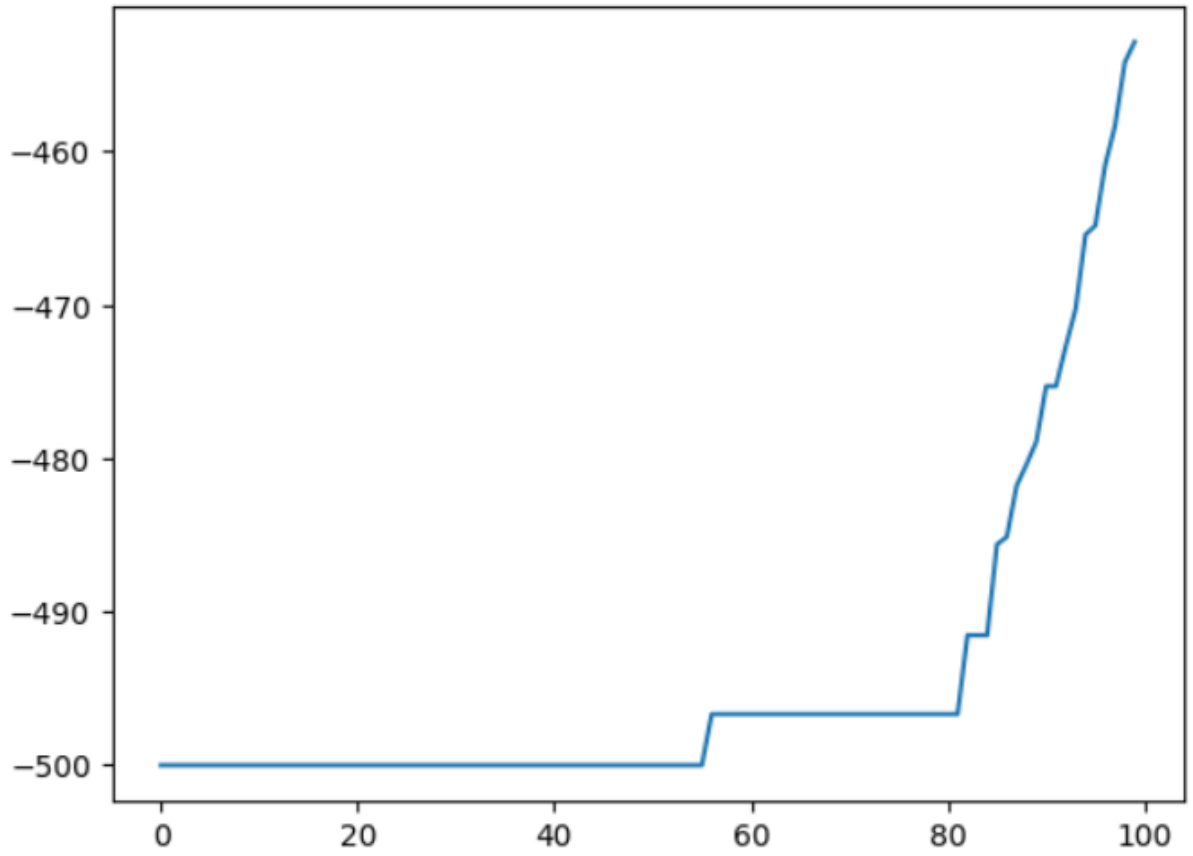
En general, se espera que el puntaje aumente a medida que el agente aprende y mejora su política. Sin embargo, es común observar una fase inicial en la que el puntaje es bajo o fluctúa ampliamente debido a la exploración y el proceso de aprendizaje. Conforme el agente adquiere más conocimiento sobre el entorno, el puntaje tiende a estabilizarse y mostrar una tendencia ascendente.

Como se observa en la gráfica esperada y en la resultante, se cumple lo mencionado y también la característica de convergencia, donde en ideal, el puntaje debería converger a un valor máximo, indicando que el agente ha alcanzado una política óptima y ha aprendido a resolver el problema de manera efectiva.

Otros experimentos

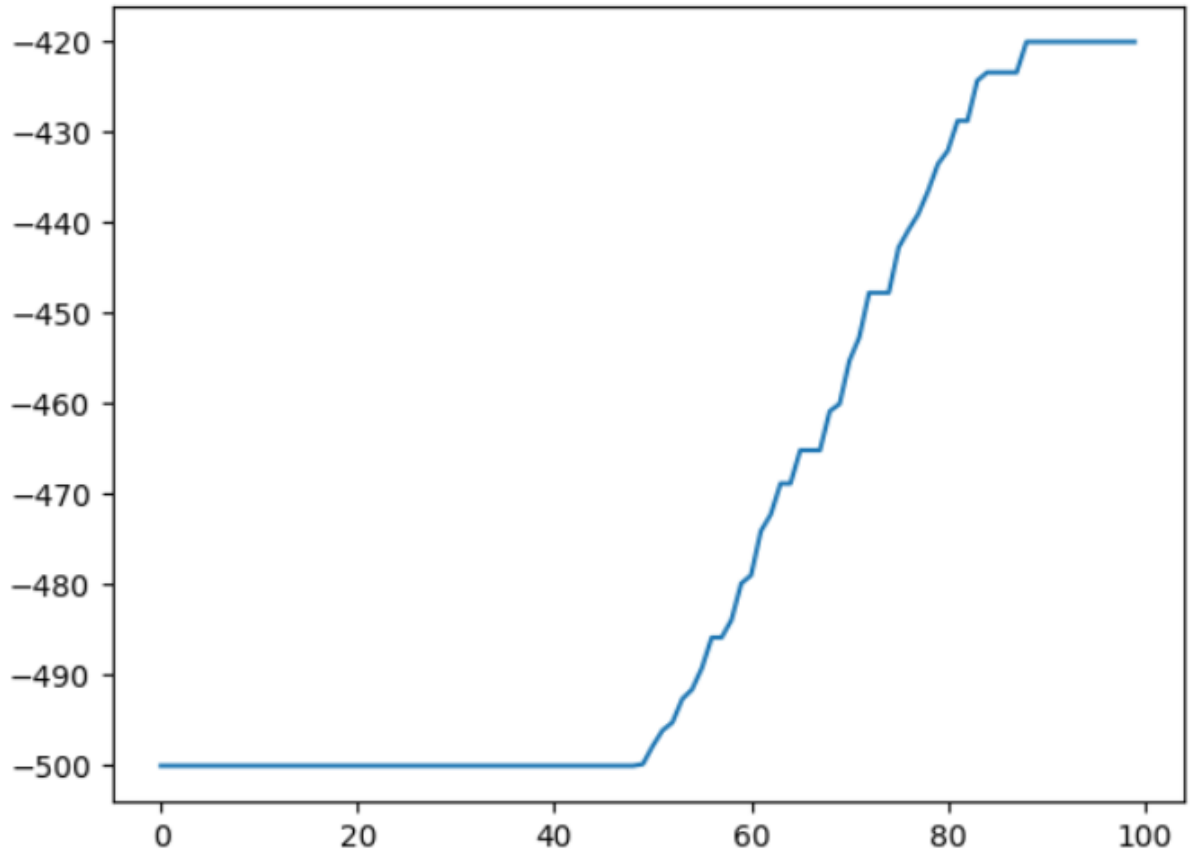
A continuación se muestran dos de los experimentos que se fueron realizando:

1) Número de episodios = 100; Alpha = 0,1; Gamma = 0,9



Se observa un aprendizaje continuo, donde el valor de la policy va mejorando, sin llegar a estabilizarse en torno a un valor promedio.

2) Número de episodios = 100; Alpha = 0,2; Gamma = 0,8



Se observa un aprendizaje en principio continuo, donde sin llegar a la policy adecuada, se comienza a estabilizar a partir del episodio número 90.

Sistema anti-aburrimiento

Descripción del juego

Aborda el popular juego conocido como "Connect 4", el cual presenta una variante única que permite la captura de fichas al encerrarlas tanto en diagonal como horizontalmente. El objetivo del juego es lograr conectar cuatro fichas del mismo color de manera consecutiva en el tablero. En caso de que el tablero se llene sin que ningún jugador logre conectar cuatro fichas, se declara un empate.

Reglas del juego

- **Objetivo:** El objetivo del juego es ser el primer jugador en conectar cuatro fichas del mismo color en una línea horizontal, vertical o diagonal en el tablero.

- **Tablero:** El juego se juega en un tablero vertical compuesto por 7 columnas y 6 filas.

- **Jugadores:** El juego involucra a dos jugadores, uno que utiliza fichas de color rojo y otro que utiliza fichas de color amarillo.

- **Turnos:** Los jugadores se turnan para colocar una ficha en el tablero. En cada turno, un jugador puede colocar una ficha en la columna de su elección.

- **Captura de fichas:** Una vez que una ficha es colocada en el tablero, si se logra encerrar una o más fichas del oponente en una línea diagonal u horizontal, estas fichas capturadas son removidas del tablero y se consideran puntos para el jugador que realizó la captura.

- **Ganador:** El primer jugador en conectar cuatro fichas del mismo color en una línea horizontal, vertical o diagonal, gana la partida. En caso de que el tablero se llene sin que ningún jugador logre conectar cuatro fichas, se declara un empate.

Árbol

Como ya se mencionó, el “Connect Four” es un juego de estrategia en el que dos jugadores se turnan para colocar fichas de su color en un tablero vertical. El objetivo es ser el primero en conectar cuatro fichas del mismo color de forma vertical o diagonal.

Un árbol de casos del “Connect Four” representa las diferentes configuraciones del tablero a medida que el juego progresa. Cada nodo del árbol representa un estado del tablero en un momento dado, y las ramas del nodo representan las diferentes acciones que se pueden tomar en ese estado.

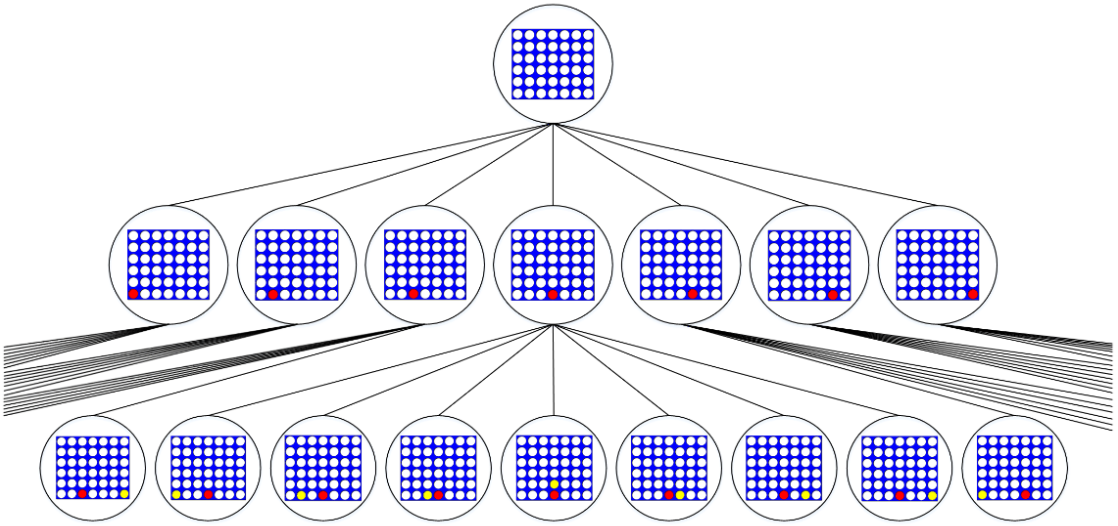


Figura 10: Árbol de casos

En cada nivel del árbol, los nodos representan los posibles estados del tablero después que un jugador haya realizado una acción (es decir, colocar una ficha en una columna). Las ramas que parten de cada nodo representan las diferentes columnas en las que se puede colocar una ficha.

El árbol de casos se expande a medida que el juego progresa, generando nuevos nodos y ramas a medida que los jugadores realizan movimientos

Cabe destacar que el tamaño del árbol de casos del “Connect Four” puede crecer rápidamente a medida que el juego avanza, ya que el número de posibles configuraciones del tablero aumenta exponencialmente.

Elección del algoritmo

Con el objetivo de programar el sistema “anti-aburrimiento”, debimos analizar cuál era el algoritmo más adecuado para resolverlo. La elección entre el algoritmo Minimax y el algoritmo Expectimax dependió de las características y los objetivos específicos del *four connect* (escenario). A continuación, se presentan algunas razones por las que consideramos a Minimax como la mejor opción:

Naturaleza de juego de suma cero: El juego Four Connect es un juego de suma cero, lo que significa que cualquier ganancia obtenida por un jugador se corresponde con una pérdida equivalente para el oponente. En este tipo de juegos competitivos, el algoritmo Minimax es particularmente adecuado, ya que busca maximizar la utilidad propia mientras minimiza la utilidad del oponente.

Optimización de la estrategia defensiva: En Four Connect, además de buscar la victoria, también es importante evitar que el oponente forme una línea de cuatro fichas. El algoritmo Minimax tiene en cuenta la posibilidad de que el oponente realice movimientos óptimos y busca minimizar la utilidad del oponente. Por lo tanto, es más adecuado para generar estrategias defensivas y considerar los posibles contraataques del oponente.

Exploración exhaustiva del espacio de búsqueda: El algoritmo Minimax permite explorar de manera exhaustiva el espacio de búsqueda del juego Four Connect mediante la generación de un árbol de juego. Esto garantiza que se analicen todas las posibles secuencias de movimientos y se elija la mejor jugada posible. Aunque la complejidad del juego puede limitar la exploración en profundidad, Minimax sigue siendo una opción viable para generar estrategias basadas en un análisis completo.

Enfoque más conservador: El algoritmo Expectimax, a diferencia del Minimax, considera las acciones del oponente de manera probabilística y promedia los resultados de los posibles movimientos. En Four Connect, donde la interacción directa con el oponente puede afectar significativamente el resultado, el enfoque

más conservador de Minimax puede ser más apropiado para tomar decisiones estratégicas basadas en suposiciones más seguras y consistentes.

Minimax

El algoritmo Minimax es utilizado para la toma de decisiones en juegos de adversarios, como el sistema "anti-aburrimiento". Funciona generando un árbol de juego que representa todas las posibles secuencias de movimientos y resultados del juego. En cada nivel del árbol, se alternan los jugadores maximizando y minimizando la utilidad, asumiendo que el oponente también juega de manera óptima. Su objetivo es encontrar la mejor jugada para un jugador, considerando que el oponente está tomando las decisiones que minimizan la utilidad para el jugador. Esto implica calcular la utilidad esperada en cada estado del juego y retroceder a través del árbol de juego para determinar la secuencia de movimientos que conduce a la mejor utilidad posible.

En el contexto de nuestro sistema, el algoritmo Minimax se utiliza para tomar decisiones estratégicas y seleccionar la mejor acción a realizar. Al generar el árbol de juego y evaluar las utilidades, el algoritmo busca maximizar la utilidad para el jugador mientras minimiza la utilidad para el oponente.

Dado que el juego en cuestión puede tener una gran cantidad de opciones y posibilidades, hemos incorporado un parámetro "d" (depth) para limitar la profundidad del árbol generado. Esto nos permite controlar el nivel de exploración y análisis del algoritmo, brindando una solución más eficiente y adecuada para el contexto del sistema.

Heurística de utilidad

Cuando la profundidad del árbol de juego alcanza su límite máximo, hemos incorporado una heurística de utilidad en nuestra implementación del algoritmo Minimax. Esta heurística nos permite estimar la utilidad de un estado en particular sin explorar todas las ramificaciones del árbol de juego. La inclusión de esta heurística es necesaria debido a restricciones de tiempo o capacidad computacional que pueden dificultar la exploración exhaustiva del espacio de juego.

A través de un análisis manual del juego, hemos identificado que una estrategia efectiva para derrotar al oponente en el sistema implica capturar las fichas del oponente y construir pilas de fichas, especialmente formando combinaciones de tres. Basándonos en este hallazgo estratégico, hemos diseñado una heurística que considera dos aspectos clave: la cantidad de fichas alineadas y la cantidad total de fichas en el tablero.

La evaluación de la cantidad de fichas alineadas nos permite determinar qué jugador tiene una ventaja en términos de formación de líneas y posibles victorias. Además, al tener en cuenta la cantidad total de fichas en el tablero, podemos evaluar la posición y la capacidad de cada jugador para formar combinaciones futuras. Estos aspectos son tenidos en cuenta en la heurística para estimar la utilidad de un estado y guiar la toma de decisiones del algoritmo Minimax.

```
#Contamos cantidad de O´s en el tabler y le restamos la cantidad de
X´s
        o_count = 0;
        x_count = 0;
        for column in range(board.heigth):
            for row in range(board.length):
                if board.grid[column][row] == "X":
                    x_count = x_count + 1;
                elif board.grid[column][row] == "O":
                    o_count = o_count + 1;
```

Figura 11: Cantidad de X y O en el tablero

```
 #Contamos la cantidad de alineaciones de 3 O s que hay, restamos las
alineaciones de 3 X s que hay.
        o_aligned = 0;
        x_aligned = 0;
        # Nos fijamos en alineaciones horizontales
        for i in range(board.heigth):
            for j in range(board.length - 3):
                if board.grid[i][j] == "X" and board.grid[i][j+1] ==
"X" and board.grid[i][j+2] == "X" and board.grid[i][j+3] == "X":
                    x_aligned += 1
                if board.grid[i][j] == "O" and board.grid[i][j+1] ==
"O" and board.grid[i][j+2] == "O" and board.grid[i][j+3] == "O":
                    o_aligned += 1

        # Nos fijamos en alineaciones verticales
        for i in range(board.heigth - 3):
            for j in range(board.length):
                if board.grid[i][j] == "X" and board.grid[i+1][j] ==
"X" and board.grid[i+2][j] == "X" and board.grid[i+3][j] == "X":
                    x_aligned += 1
                if board.grid[i][j] == "O" and board.grid[i+1][j] ==
"O" and board.grid[i+2][j] == "O" and board.grid[i+3][j] == "O":
                    o_aligned += 1

        # Nos fijamos en alineaciones diagonales
        for i in range(board.heigth - 3):
            for j in range(board.length - 3):
                if board.grid[i][j] == "X" and board.grid[i+1][j+1] ==
"X" and board.grid[i+2][j+2] == "X" and board.grid[i+3][j+3] == "X":
                    x_aligned += 1
                if board.grid[i][j] == "O" and board.grid[i+1][j+1] ==
"O" and board.grid[i+2][j+2] == "O" and board.grid[i+3][j+3] == "O":
                    o_aligned += 1
```

Figura 12: Cantidad de alineaciones de fichas (parte 1)

```
        # Nos fijamos en alineaciones diagonales (inclinacion negativa)
        for i in range(3, board.heigth):
            for j in range(board.length - 3):
                if board.grid[i][j] == "X" and board.grid[i-1][j+1] ==
"X" and board.grid[i-2][j+2] == "X" and board.grid[i-3][j+3] == "X":
                    x_aligned += 1
                if board.grid[i][j] == "O" and board.grid[i-1][j+1] ==
"O" and board.grid[i-2][j+2] == "O" and board.grid[i-3][j+3] == "O":
                    o_aligned += 1
        return 300*(o_count - x_count) + 300*(o_aligned - x_aligned);
```

Figura 13: Cantidad de alineaciones de fichas (parte 2)

Para evaluar el estado del juego, se implementó una heurística que considera varios factores. En primer lugar, se contabiliza la cantidad de fichas "X" y "O" presentes en el tablero. Este conteo se realiza mediante un bucle que recorre todas las filas y columnas del tablero y suma la cantidad de fichas encontradas para cada jugador.

A continuación, se procede a contar las alineaciones de tres fichas "X" y "O" en el tablero. Se consideran las alineaciones en sentido horizontal, vertical y diagonal (tanto ascendente como descendente). Para cada tipo de alineación, se utilizan bucles anidados para recorrer las posiciones del tablero y verificar si se forma una alineación de tres fichas del mismo tipo. Cada vez que se encuentra una alineación, se incrementa el contador correspondiente.

Finalmente, se calcula la utilidad del estado del juego utilizando la fórmula: 100 * (cantidad de fichas "O" - cantidad de fichas "X") + 50 * (alineaciones de fichas "O" - alineaciones de fichas "X"). Se multiplican los valores por 300 y 300 para optimizar el rendimiento de la inteligencia artificial y maximizar sus posibilidades de victoria.

Durante las pruebas realizadas, se evaluaron diferentes situaciones de juego y se asignaron valores a través de la función de evaluación. Se determinó que, en todos los casos, el valor asignado a una derrota (eval(loose)) es menor que el valor asignado a un empate (eval(draw)), y que este último es menor que el valor asignado a una victoria (eval(win)).

Esto indica que, según la evaluación realizada, obtener una victoria en el juego tiene el mayor valor, seguido de un empate y, finalmente, una derrota. Esta jerarquía de valores refleja la importancia de alcanzar la victoria y la preferencia de evitar una derrota en el juego.

Aclaraciones: Para implementar la función heurística se añadió un getter a la clase Board que permite al agente obtener el tablero (grid).

Resultados

Pruebas contra agente precargado: El algoritmo fue sometido a 20 enfrentamientos contra otro agente, previamente cargado, utilizando el mismo juego. En cada enfrentamiento, el algoritmo implementado utilizando el algoritmo Minimax logró obtener la victoria en todas las ocasiones. Estos resultados muestran una eficiencia notable del algoritmo, demostrando su capacidad para tomar decisiones estratégicas y maximizar su utilidad en cada jugada.

Pruebas contra humano: En los enfrentamientos contra un humano, se observó que el humano tiene la posibilidad de ganarle al algoritmo, pero requiere de varias jugadas y estrategias bien planificadas para lograrlo. Por otro lado, es común que si el programador se distrae o comete errores, el algoritmo Minimax logre obtener la victoria. Estos resultados indican que el nivel de inteligencia del agente desarrollado es bastante aceptable y presenta un desafío para los jugadores humanos.

Conclusión

Las pruebas realizadas para evaluar la eficiencia del algoritmo Minimax desarrollado han arrojado resultados satisfactorios. En los enfrentamientos contra un agente precargado, el algoritmo obtuvo una tasa de victorias del 100% en las 20 ocasiones probadas. Además, en los enfrentamientos contra un humano, el algoritmo demostró ser un oponente desafiante, requiriendo estrategias bien planificadas y múltiples jugadas para que el humano pueda vencerlo. Estos resultados respaldan la efectividad y el nivel de inteligencia del agente desarrollado, lo que sugiere un buen desempeño en la toma de decisiones estratégicas en el juego "Connect 4" con la variante implementada.

En una partida de Four Connect entre un jugador humano y el algoritmo Minimax, se observa una victoria del algoritmo Minimax sobre el jugador humano:

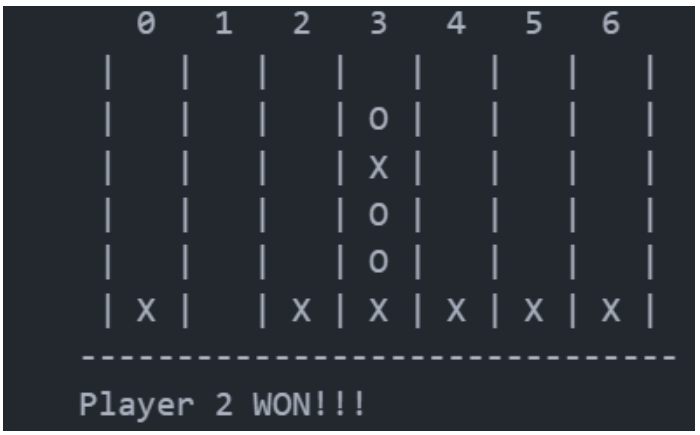


Figura 14: Minimax vs Humano

En un enfrentamiento entre el algoritmo Minimax y SpaceGPT, se desarrolla una partida de Four Connect donde el algoritmo Minimax logra obtener la victoria:

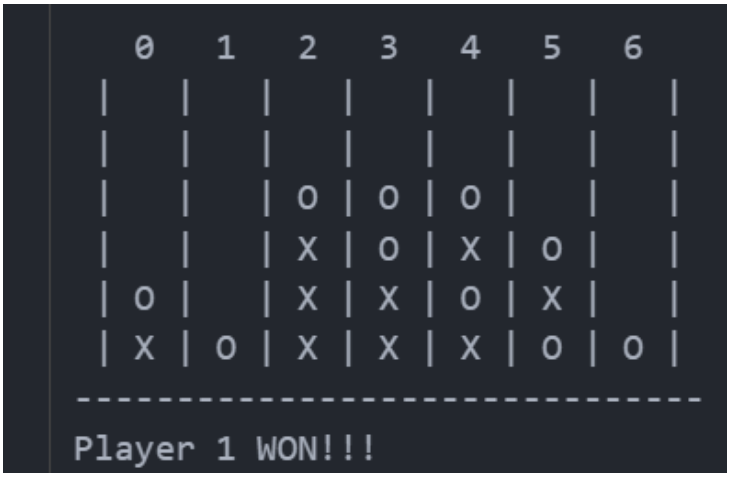


Figura 15: Minimax vs SpaceGPT

Bibliografía

- (2023). Gymnasium: Classic Control - Mountain Car. Recuperado de https://gymnasium.farama.org/environments/classic_control/mountain_car/#mountain-car

- Sutton, R. S., & Barto, A. G. (2014, 2015). Temporal-Difference Learning. In Reinforcement Learning: An Introduction (2nd ed., pp. 143-161). Cambridge, Massachusetts: The MIT Press.

- Yovine, S. (2022). Inteligencia Artificial Q-learning. Universidad ORT Uruguay.

- Yovine, S. (2023). Inteligencia Artificial: Introducción a Juegos Alternados de Suma Cero de Dos Jugadores. Universidad ORT Uruguay.