

Universidad ORT Uruguay

Facultad de Ingeniería

## Obligatorio Inteligencia Artificial

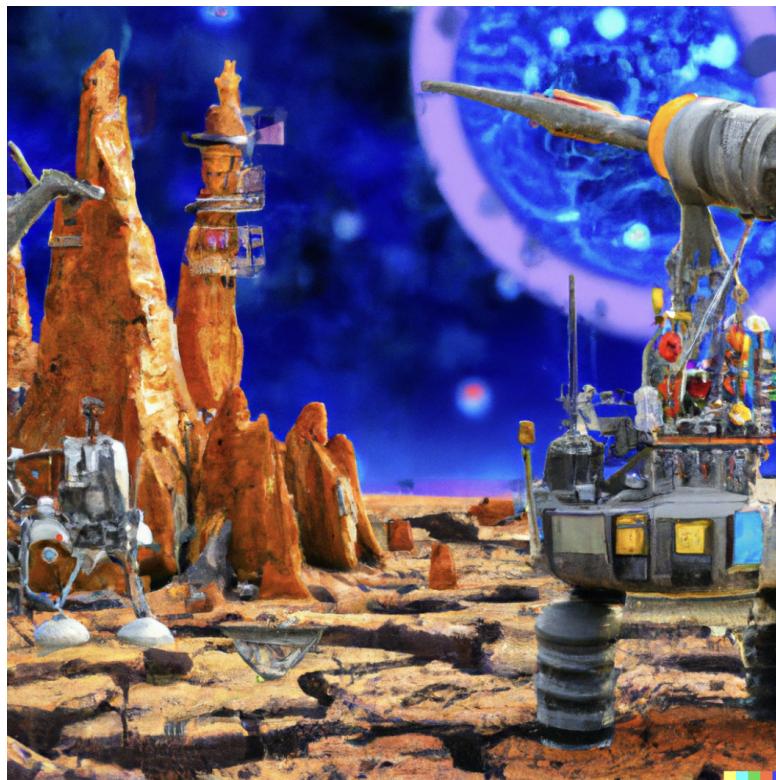


Figura 1: “Artificial Intelligence conducting a fleet to Mars, and providing entertainment to the passengers”

@DALL-E 2

**Federico Iglesias - 244831**

**Marcelo Pérez - 227362**

**Francisco Rossi - 219401**

**2022**

# Índice

<b>1. Resumen del abordaje de la tarea</b>	<b>3</b>
1.1. Cartpole . . . . .	3
1.1.1. Proceso . . . . .	3
1.1.2. Parámetros óptimos encontrados . . . . .	4
1.2. 2048 . . . . .	6
1.2.1. Proceso . . . . .	7
<b>2. Desempeño de las soluciones</b>	<b>9</b>
2.1. CartPole . . . . .	9
2.2. 2048 . . . . .	9
<b>3. Notas de advertencia</b>	<b>10</b>
<b>4. Anexo CartPole utilizando enfoque de “exploración adaptativa”</b>	<b>11</b>
<b>5. Bibliografía</b>	<b>13</b>

# 1. Resumen del abordaje de la tarea

## 1.1. Cartpole

El objetivo fue trabajar en base al algoritmo **Q-learning** para poder crear un modelo capaz de soportar un aterrizaje en Marte de forma completamente automática. Para esto nos basamos en el ambiente de simulación creado por **gym Cartpole** para crear nuestro propio simulador.

Para adentrarnos en la implementación del modelo, es importante definir bien el concepto de **Q-learning**. Se trata de un algoritmo de aprendizaje por refuerzo, comúnmente utilizado en problemas **MDP**, que permite el aprendizaje en base a la interacción del agente (con el ambiente). El objetivo principal del **Q-learning** es obtener una política óptima y así, maximizar el valor esperado de la recompensa. Por lo tanto, en primera instancia se planeó que el agente interactúe con un simulador y aprenda, antes de que haga la interacción con el sistema de aterrizaje real. Lo que trata de hacer es estimar una política óptima, que le indique al agente cómo actuar frente a las distintas situaciones que se puede llegar a encontrar interactuando con el sistema, de manera de soportar el aterrizaje.

Un enfoque que toma este algoritmo es el de no siempre recurrir al mismo método para obtener una acción, sino que considera la exploración de forma estocástica, con el fin de descubrir alguna nueva acción para determinado estado y así intentar mejorar su aprendizaje.

### 1.1.1. Proceso

Nuestra primera implementación del modelo utilizando la técnica **Q-learning** interactuaba con el simulador obteniendo malos resultados. Pudimos encontrar varios problemas relacionados a esto. Lo primero que decidimos cambiar fue la tabla del algoritmo, encargada de ir guardando los valores para cada estado, ya que este contenía valores discretos y el ambiente proporciona elementos continuos. Se trabajó con una herramienta de **numpy (linspace)** para adaptar el espacio de valores continuos a un grupo discreto. También sirvió para ajustar los límites de valores posibles ya que según la documentación, cuando se sobrepasa algunos valores específicos en su posición, velocidad, ángulo o velocidad angular el sistema falla en mantener el equilibrio.

Luego de algunos otros cambios en nuestro código, como por ejemplo calcular la acción de forma aleatoria según un **épsilon**, el modelo mejoró. Según la documentación de **gym**, el objetivo de las recompensas para la versión 1 del simulador es de 475 pasos sin dejar caer el *pole*. Nuestro agente podía alcanzar rápidamente los 500 pasos, siendo este el máximo, pero en promedio nunca superaba los 400 pasos sin antes dejar caer el *pole*.

Se observó que esto fue debido a los parámetros que habíamos seleccionado. Nuestro **épsilon**, era demasiado grande por lo que al explorar demasiado nunca llegaba en promedio a 500 pasos sin dejarlo caer como se puede observar en la siguiente imagen (aproximadamente 1 hora corriendo):

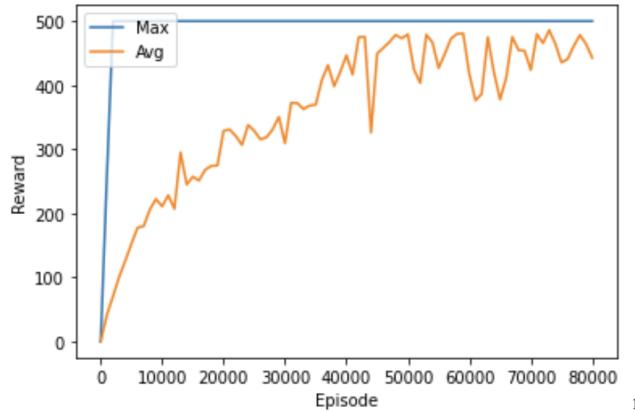


Figura 2: Resultados primer versión.  
Gráfica encontrada en Documentation/CartPole/80kRun.png

Ajustamos el **épsilon** y también se incluyó en la gráfica el Mínimo de cada intervalo de episodios para poder visualizar la varianza:

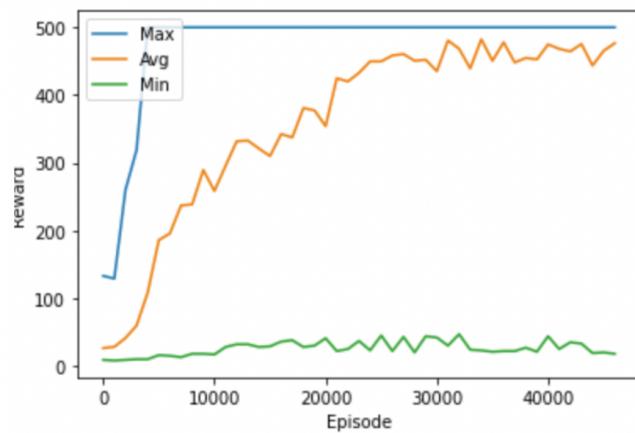


Figura 3: Resultados ajustando épsilon  
Gráfica encontrada en Documentation/CartPole/40kRun.png

Luego de ganar 5 veces, se corta la ejecución. Se puede observar que se obtuvieron muy buenos resultados en un número significativamente menor de episodios, con el promedio alcanzando valores cercanos a 500.

### 1.1.2. Parámetros óptimos encontrados

- **Epsilon:** 0.06.
- **Episodios:** Menos de 50000
- **Gamma:** 0.995
- **Learning Rate (lr):** 0.14



Figura 4: “Artificial Intelligence playing CartPole”

@DALL·E 2

## 1.2. 2048

Todos los integrantes del equipo teníamos experiencia previa con el juego 2048 y conocíamos algunas estrategias para jugar prósperamente, como intentar acumular fichas de valor alto en una esquina (intentamos integrar esto a nuestra solución, pero finalmente lo descartamos).

La primera interacción con esta tarea fue de prueba con el simulador `GameBoard`. El Random Agent provisto permitió correr simulaciones con pasos aleatorios y así comenzar a conocer la naturaleza del problema, auxiliados por la función de `board.render()` para la visualización del tablero. Los primeros resultados fueron de un alcance muy limitado, con un tiempo total de 0:00:00.656262 y 60 movidas hasta perder.

```
Output exceeds the size limit. Open the full output data in a text editor
      0      0      0      0
      0      0      0      0
      0      0      0      4
      0      0      0      2

Next Action: "UP" , Move: 0
      0      0      0      4
      0      0      0      2
      0      0      0      0
      4      0      0      0

Next Action: "DOWN" , Move: 1
      0      0      0      0
      0      0      0      0
      0      2      0      4
      4      0      0      2

Next Action: "RIGHT" , Move: 2
      0      0      2      0
      0      0      0      0
      0      0      2      4
      0      0      4      2

Next Action: "UP" , Move: 3
      0      0      4      4
...
Total time: 0:00:00.656262

Total Moves: 60
B000000000!!!!!!
```

Figura 5: Resultado Jugar 2048 con agente Random

Se evaluó la posibilidad de usar las técnicas **Minimax** y **Expectimax**, y se decidió por la última dada la mejor adecuación al ambiente. El `GameBoard` agrega fichas de forma aleatoria, como se puede ver en la implementación de su función: `__add_random_tile`. Tanto **Minimax** como **Expectimax** buscan la acción que maximice los resultados del agente, pero **Minimax** espera del oponente la acción que minimice su retorno, mientras que **Expectimax** espera una acción estocástica de su oponente. Por tanto, **Expectimax** es la técnica adecuada para el problema.



Figura 6: “Passengers of a spaceship playing board games”  
@DALL-E 2

### 1.2.1. Proceso

Se desarrolló la clase `ExpectimaxAgent`, que implementa la interfaz `Agent` y por tanto define las funciones de `play` y `heuristic_utility`.

La función de heurística tiene cuatro opciones, una de ellas por cada una de las posibilidades sugeridas (smoothness, valor, cantidad de espacios vacíos) y otra que corresponde a la suma de todas las anteriores (sum). Todas estas heurísticas están fuertemente relacionadas con posibilidades de ganar el juego y también son relativamente fáciles de calcular. Se puede indicar la heurística a usar cuando se instancia la clase.

Del mismo modo, se hizo parametrizable la `depth` que utiliza el algoritmo para evaluar la heurística. Esta `depth` corresponde a la cantidad de movimientos que simula, tanto del agente como del “oponente” (lease, las fichas 2 o 4 agregadas aleatoriamente entre las jugadas del agente).

Asimismo, se agregó una condición que corta las ramificaciones de simulaciones en los casos en que hay más de 5 casilleros vacíos y `depth` mayor o igual a tres, dado que estos casos corresponden generalmente a las primeras jugadas y no es necesaria tanta rigurosidad en el análisis, prefiriendo decidir estos movimientos más rápidamente.

A partir de este modelo, se simularon 7 jugadas para cada tupla (heurística, `depth`), variando el `depth` entre 1 y 5, ya que para valores mayores se notó que el tiempo de juego era excesivo (colisionando con la segunda propiedad de una buena heurística, según la cual debería ser fácil y rápida de computar). A continuación una tabla con los datos obtenidos, que utilizamos para configurar la versión final del agente:

heurística/depth	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>
“empty”	Avg. Time (s): 0.06  Avg. Moves: 257  W: 0 - L: 7	Avg. Time (s): 0.37  Avg. Moves: 303  W: 0 - L: 7	Avg. Time (s): 3.95  Avg. Moves:505  W: 0 - L: 7	Avg. Time (s): 8.88  Avg. Moves: 548  W: 0 - L: 7	Avg. Time (s): 60.47  Avg. Moves: 766  W: 0 - L: 7
“value”	Avg. Time (s): 0.06  Avg. Moves: 253  W: 0 - L: 7	Avg. Time (s): 0.28  Avg. Moves: 238  W: 0 - L: 7	Avg. Time (s): 5.15  Avg. Moves: 666  W: 0 - L: 7	Avg. Time (s): 8.66  Avg. Moves: 548  W: 0 - L: 7	Avg. Time (s): 48.27  Avg. Moves: 628  W: 0 - L: 7
“smoothness”	Avg. Time (s): 0.07  Avg. Moves: 242  W: 0 - L: 7	Avg. Time (s): 0.40  Avg. Moves: 246  W: 0 - L:7	Avg. Time (s): 5.53  Avg. Moves: 905  W: 3 - L: 4	Avg. Time (s): 9.51  Avg. Moves: 551  W: 0 - L: 7	Avg. Time (s): 70.54  Avg. Moves: 895  W: 3 - L: 4
“sum”	Avg. Time (s): 0.16  Avg. Moves: 543  W: 0 - L: 7	Avg. Time (s): 0.96  Avg. Moves: 464  W: 0 - L: 7	Avg. Time (s): 10.28  Avg. Moves: 940  W: 6 - L: 1	Avg. Time (s): 25.07  Avg. Moves: 926  W: 4 - L: 3	Avg. Time (s): 98.90  Avg. Moves: 943  W: 6 - L: 1

Se aclara que se corrieron aún más iteraciones y en los resultados se identificó que todas las heurísticas, para algún `depth`, lograron ganar partidas. La mayor cantidad de partidas ganadas está dada por el modelo que utiliza la heurística “sum”, con `depth` igual a 4. Por esta razón, se optó por dejar la opción “sum” como default en la clase **ExpectimaxAgent** implementada, con el valor `depth` en 3. Así, el agente gana el juego en la mayoría de los casos, y en un tiempo aproximado de 10 segundos. Notamos también que la heurística *smoothness* es la que mayor precisión aporta al total de la suma.

La implementación final permite modificar los parámetros antedichos para utilizar las variaciones que se prefieran. Los parámetros opcionales son:

- `depth=n`: toma el valor natural n como `depth` a evaluar. Por defecto n=3.
- `heuristic={heuristic_option}`; puede tomar los valores “empty”, ”value”, ”smoothness”, en otro caso o por defecto se utiliza la heurística correspondiente a la suma de los tres valores antedichos.

Los valores de los pesos utilizados en las heurísticas son: `Smoothness_weight = 3`. `Empty_weight = 50000`.

## 2. Desempeño de las soluciones

### 2.1. CartPole

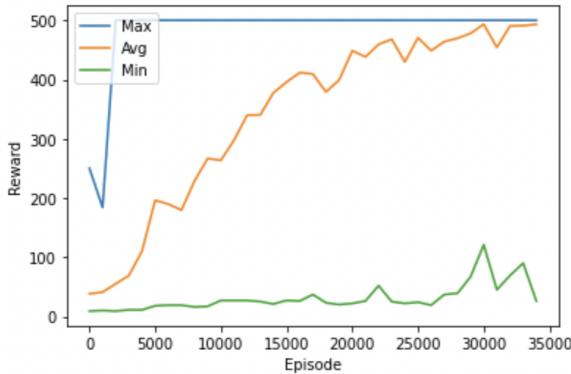


Figura 7: Resultados Cartpole con parametros óptimos

Se cortó la ejecución del simulador luego de alcanzar el objetivo 5 veces en promedio. Esta corrida se puede encontrar en la carpeta `Documentation/CartPole/DesempeñoEpsilon001`, donde se puede observar que se limita la exploración luego de superar el umbral (475 pasos) por primera vez, y de esta forma poder alcanzar el objetivo más regularmente.

Aunque se puede observar que sigue perdiendo al menos algunas partidas cada mil episodios, se ve una tendencia de mejora con respecto a la varianza entre el promedio y el mínimo. Si se intentara correr por mas tiempo y dejar que siga aprendiendo, como hipótesis creemos que la brecha disminuiría significativamente. A pesar de esto, creemos haber obtenido buenos resultado, dado que en promedio cada 1000 episodios, el simulador logra alcanzar el objetivo seguidamente luego de alcanzarlo por primera vez.

### 2.2. 2048

Los resultados se evaluaron corriendo 100 juegos con la versión final del agente **Expectimax** sin instanciarlo con parámetros, es decir, con las opciones por defecto. El mismo logró ganar el juego en 58 de las pruebas realizadas.

El tiempo promedio de ejecución fue de 9.72 segundos y el promedio de movimientos: 923. Esta corrida se puede encontrar en `Documentation/2048.html`

Se agrega que para `Expectimax(depth=5)` se puede obtener un promedio de partidas ganadas aún mayor, pero con un tiempo de juego cercano a los dos minutos por partida.

### 3. Notas de advertencia

Queremos mencionar algunas advertencias antes de probar nuestro sistemas.

Para el CartPole existen algunas dependencias que se necesitan para poder ejecutar nuestro simulador. Entre ellas están gym, pygame, matplotlib y numba que sirven para distintas funcionalidades requeridas para correrlo.

Nuestro sistema de interacción 2048, viene configurado por defecto con valores iniciales que el equipo considera óptimos. Pero si el usuario lo desea se pueden alterar, por ejemplo la profundidad del árbol o que heurísticas utilizar.

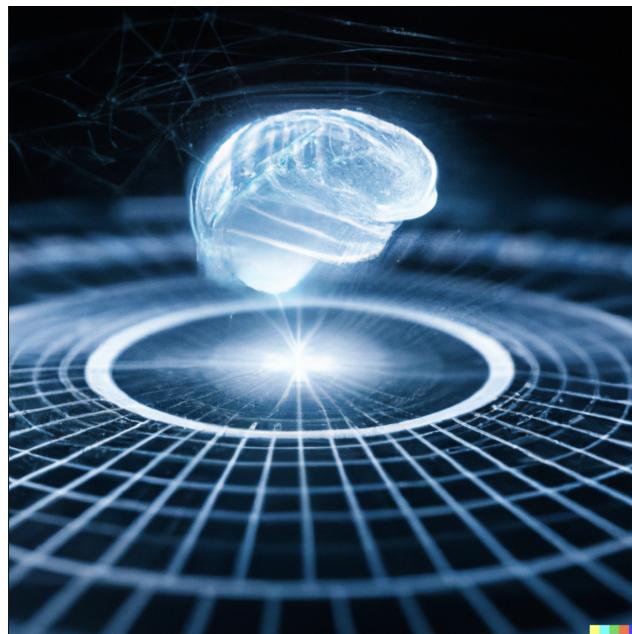


Figura 8: “Realistic Artificial Intelligence using advanced algorithms to win game”

@DALL-E 2

#### 4. Anexo CartPole utilizando enfoque de “exploración adaptativa”

A pesar de contar con tiempos bastante buenos para resolver el problema, decidimos alcanzar niveles de optimización superiores. Para esto se realizó una investigación sobre el código y pudimos observar que al menos una vez cada 1000 episodios se alcanzaban los 500 pasos cumpliendo el objetivo final. Esto quiere decir que hubo aprendizaje quedando registro de esto en la tabla para ciertos estados. Este simple hecho nos hizo darnos cuenta que si limitamos el explorar aún más, podríamos obtener mejores resultados en cuanto a performance. Luego de varias corridas probando distintas formas de limitar el epsilon, llegamos a la conclusión de que si en alguna corrida alcanzó los 500 pasos, el epsilon debía ser reducido cercano a 0 y si en ninguna corrida se lograba el objetivo se volvía a setear el epsilon para que pudiera seguir explorando en algunos casos.

Los resultados son los siguientes:

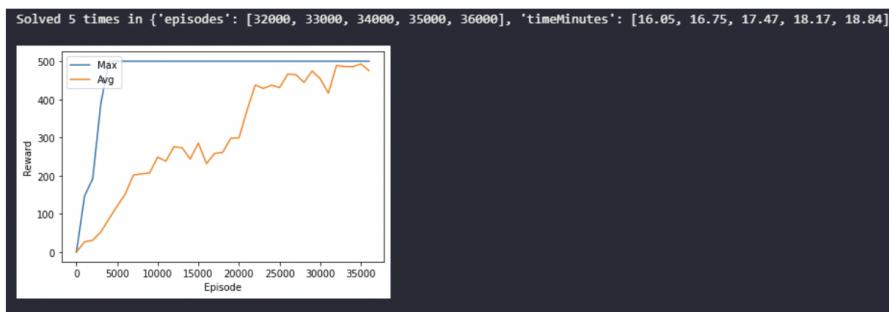


Figura 9: Resultados con exploración adaptativa 1

El algoritmo resolvió el problema superando los 475 pasos que indica la documentación, 5 veces seguidas en la mitad del tiempo, el más rápido en 16 minutos. Pero al limitar tanto la exploración, se puede observar en la gráfica que nunca llega a 500 pasos exactamente pero sí muy cercano a este valor (492).

Hicimos una última corrida para comprobar que no haya sido una corrida especial la anterior y decidimos remover el epsilon si en alguna corrida se obtiene los 500 pasos en vez de solamente acercarlo a 0. Los resultados fueron aún mejores y fue la mejor corrida que pudimos obtener, resolviendo el problema completamente en 4 minutos.

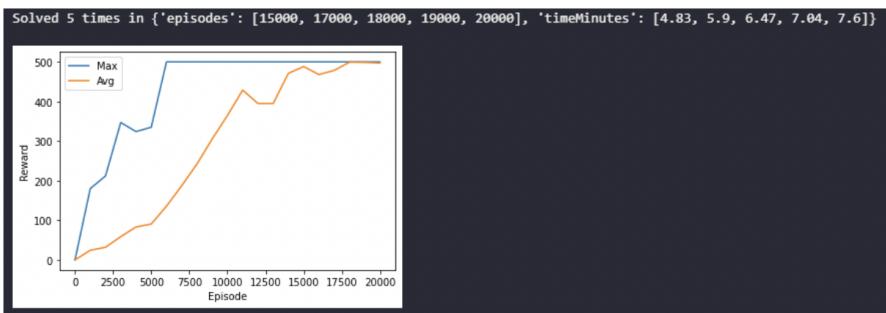


Figura 10: Resultados con exploración adaptativa 2

En este caso acercándose aún más a los 500 pasos (499) en promedio pero no obteniendo los 500 máximos. Esta corrida se puede encontrar en la carpeta Documentation/CartPole/AdaptiveEpsilon.html

Consideramos que aunque le falte un único paso para llegar a los 500 en un promedio de 1000 corridas por vez, es un excelente resultado pero creemos que no es una forma muy ortodoxa de resolver el problema ya que modificamos el épsilon a conveniencia en cada corrida del simulador. Por esto, quisimos documentarlo, pero no es nuestra solución final del problema.

## 5. Bibliografía

- Michel Tokic. (n.d.). Adaptive epsilon-greedy Exploration in Reinforcement Learning Based on Value

**Retrieved from:**

<http://tokic.com/www/tokicm/publikationen/papers/AdaptiveEpsilonGreedyExploration.pdf>

- Maciej Balawejder. (2021). Solving Open AI's CartPole Using Reinforcement Learning Part-1

**Retrieved from:**

<https://medium.com/analytics-vidhya/q-learning-is-the-most-basic-form-of-reinforcement-learning-which-doesnt-take-advantage-of-any-8944e02570c5>

- @ovolve. (2014). What is the optimal algorithm for the game 2048?

**Retrieved from:**

<https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048>

- Dorian Lazar. (2020). How to apply Minimax to 2048

**Retrieved from:**

<https://towardsdatascience.com/playing-2048-with-minimax-algorithm-1-d214b136bffb>

- PPTS del curso

- <https://artint.info/2e/html/ArtInt2e.Ch12.S4.html>
- <https://labs.openai.com/s/5HhVF36FNnZZjfBqR2p297sy>
- <https://labs.openai.com/s/7wyNcJYBSdgydZeJL5rvnd5q>
- <https://labs.openai.com/s/ny6Nnyi2Ud1ezSWo3V945t5L>
- <https://labs.openai.com/s/yAU7AhTDMFu8NSZFf8GExQ7z>



Figura 11: "Research in ancient library, drawing by Giovanni piranesi"

@DALL-E 2