

Universidad ORT Uruguay  
Facultad de Ingeniería

# Inteligencia Artificial Obligatorio

**Link al repositorio:**

[https://github.com/jstf13/Artificial\\_Intelligence](https://github.com/jstf13/Artificial_Intelligence)

Daniel Paggiola - 274741  
Juan Toledo - 222371

Profesor:  
Federico Armando

Entregado como requisito de la materia Inteligencia artificial  
10 de Julio de 2023

## **Índice**

<b>Introducción:</b>	<b>3</b>
<b>Mountain Car:</b>	<b>3</b>
<b>Q-learning</b>	<b>5</b>
<b>Datos Iniciales</b>	<b>6</b>
<b>Interacción con el simulador</b>	<b>7</b>
<b>Mejoras realizadas</b>	<b>7</b>
<b>Generación de modelo computado</b>	<b>11</b>
<b>Connect-Four</b>	<b>12</b>
<b>Ejemplo de Juego con el modelo</b>	<b>14</b>

# Introducción:

El presente informe se centra en el proyecto encargado por la empresa SpaceGPT, que consiste en la implementación de un agente inteligente para controlar un Rover en Marte. Sin embargo, se ha planteado un desafío adicional debido a la extraordinaria inteligencia del agente: el aburrimiento durante el viaje. Con el objetivo de garantizar el entretenimiento de la tripulación durante la duración estimada del viaje de 300 a 400 días terrestres, se ha desarrollado un sistema anti-aburrimiento basado en el juego Connect-Four con modificaciones específicas.

El proyecto se divide en dos tareas principales: el aprendizaje del manejo del vehículo y el desarrollo del sistema anti-aburrimiento. Para abordar estas tareas, se han proporcionado simuladores específicos que permitirán al equipo de trabajo cumplir con los objetivos establecidos.

La primera tarea, denominada "Mountain Car", se basa en la técnica de Q-Learning. El simulador proporcionado, permitirá al equipo desarrollar un agente inteligente capaz de aprender y controlar eficientemente el Rover en entornos desafiantes.

Connect-Four implica el desarrollo del sistema anti-aburrimiento utilizando una técnica Minimax. El objetivo es generar una conexión de 4 fichas en el tablero. En esta versión mejorada, se ha añadido una mayor complejidad al sistema anti-aburrimiento. Ahora, además de buscar la conexión de fichas, se permite "comer" las fichas del rival bajo ciertas condiciones. Para que una ficha pueda ser comida, debe insertarse de tal manera que encierre horizontal o diagonalmente una ficha rival. No se permite comer fichas de forma vertical, ya que esto haría el juego trivial.

# Mountain Car:

Para abordar la tarea de Mountain Car, nos enfocamos en la técnica de aprendizaje por refuerzo conocida como Q-Learning. Utilizamos el simulador proporcionado, [Aulas], para desarrollar un agente inteligente capaz de aprender y controlar eficientemente el Rover en entornos desafiantes.

Nuestro objetivo principal fue entrenar al agente para que aprendiera a manejar el vehículo en un escenario de montaña, donde debía alcanzar la cima de la colina. El desafío radica en el hecho de que el vehículo tenía una potencia limitada y debía aprender a aplicar la aceleración adecuada en el momento oportuno para superar las pendientes y evitar quedar atrapado en los valles.

Para abordar esta tarea utilizando Q-Learning, diseñamos un modelo de aprendizaje por refuerzo que constaba de un conjunto de estados, acciones y recompensas. En cada paso, nuestro agente tomaba una acción basada en el estado actual y actualizaba su función Q, que estimaba el valor de cada acción en un estado dado, utilizando la ecuación de actualización de Q-Learning.

Durante el entrenamiento del agente, implementamos un enfoque de exploración y explotación para equilibrar la exploración de nuevas acciones y la explotación de acciones ya conocidas. Esto permitió que nuestro agente aprendiera gradualmente la mejor política para alcanzar el objetivo de manera óptima.

Utilizamos el simulador mencionado anteriormente para interactuar con el agente, permitiendo su entrenamiento y evaluación. Experimentamos con diferentes configuraciones de parámetros, como la tasa de aprendizaje, el factor de descuento y la estrategia de exploración, con el fin de obtener los mejores resultados posibles en términos de tiempo de aprendizaje y rendimiento del agente.

En nuestro informe, incluimos detalles sobre la implementación del agente Q-Learning, así como los resultados obtenidos durante el proceso de entrenamiento y las pruebas posteriores. Utilizamos gráficos y comentarios para visualizar y analizar el desempeño del agente en diferentes episodios de entrenamiento y validación.

Además, proporcionamos un análisis exhaustivo y conclusiones sobre la efectividad de la técnica Q-Learning en la tarea de Mountain Car, destacando los desafíos que enfrentamos y las lecciones que aprendimos durante el desarrollo del agente inteligente. Estos hallazgos nos permitieron evaluar el cumplimiento de nuestros objetivos y proporcionar información valiosa para futuras mejoras en este campo de aplicación de la Inteligencia Artificial.

# Q-learning

```
alpha = 0.9
gamma = 0.99
num_episodes = 10000

best_reward = float('-inf')
best_episode = None
best_position = -1.2

for i in range(num_episodes):
    obs = env.reset()
    total_reward = 0
    alpha = max(0.01, min(0.5, 1.0 - math.log10((i+1)/25)))
    epsilon = max(0.01, min(1, 1.0 - math.log10((i+1)/25)))

    done = False

    while not done:
        state = get_state(obs)
        action = epsilon_greedy_policy(state, Q, epsilon)
        obs, reward, done, info = env.step(action)
        total_reward += reward

        next_state = get_state(obs)
        Q[state][action] = Q[state][action] + alpha * (reward + gamma * np.max(Q[next_state]) - Q[state][action])

    if total_reward > best_reward:
        best_reward = total_reward
        best_episode = i

    if best_position < next_state[0]:
        best_position = next_state[0]

    if i % 1 == 0:
        print("Episode #: Reward = {}, Best reward = {}, Position = {}, Epsilon = {}, Alpha = {}".format(i, total_reward, best_reward, best_position, epsilon, alpha))
        print("Position without discretization: {}".format(obs[0]))

print("\nBest episode:")
print("Episode #(), Best reward = {}, position = {}, Epsilon = {}, Alpha = {}".format(best_episode, best_reward, best_position, epsilon, alpha))
print("Final position without discretization: {}".format(obs[0]))
```

Este código es el utilizado para el entrenamiento de nuestro agente de aprendizaje por refuerzo utilizando el algoritmo de Q-learning. Durante el entrenamiento, el agente interactúa con un entorno (env), actualiza los valores de Q y busca maximizar la recompensa acumulada.

El algoritmo utiliza una política epsilon-greedy para seleccionar las acciones. Al principio del entrenamiento, el agente explora el entorno eligiendo acciones de forma aleatoria con una probabilidad epsilon, y con una probabilidad de (1 - epsilon) elige la mejor acción conocida según los valores de Q. A medida que avanza el entrenamiento, la exploración disminuye y el agente se basa cada vez más en la explotación de los conocimientos aprendidos. Los valores de alfa y epsilon se actualizan en cada episodio utilizando funciones logarítmicas que disminuyen a medida que avanza el entrenamiento. Esto permite un equilibrio entre la exploración y la explotación, favoreciendo la exploración al principio y la explotación posteriormente.

Al final del entrenamiento, se muestra la mejor ejecución encontrada, que corresponde al episodio con la mayor recompensa obtenida.

# Datos Iniciales

En el contexto del problema del Mountain Car, se realizaron modificaciones en los datos iniciales con el objetivo de mejorar la representación del entorno y reducir el tiempo de ejecución en las pruebas. Estas modificaciones se describen a continuación:

- **Recompensa:** El valor de la recompensa, que originalmente se establecía en -1, fue ajustado para proporcionar una penalización más gradual en función de los pasos dados por el agente. En lugar de utilizar -1 como recompensa constante, se adoptó la fórmula  $-0.1 * \text{step}$ , donde cada paso (step) se penaliza multiplicando por 0.1. Esto permite desalentar acciones que tomen más tiempo para resolver el problema.
- **Rango del espacio de observación:** El rango original de observación, que abarcaba desde -5 hasta 5, fue modificado para reflejar mejor las limitaciones reales del problema del Mountain Car. Ahora, el rango de observación se estableció utilizando la función `np.linspace(-1.2, 0.6, 10)`, dividiendo el espacio entre -1.2 y 0.6 en 10 puntos equidistantes. Esta adaptación proporciona un rango más realista para el agente y sus observaciones.
- **Rango de acciones:** El rango original de acciones, que iba desde -3 hasta 3, fue ajustado para reflejar las limitaciones del problema del Mountain Car. Se utilizó la función `np.linspace(-0.07, 0.07, 10)` para definir un nuevo rango de acciones. Este rango más limitado, dividido en 10 valores equidistantes, se ajusta mejor a las acciones factibles en el contexto del problema.
- **Eliminación de la visualización gráfica:** Con el fin de reducir el tiempo de ejecución en las pruebas, se decidió eliminar la visualización gráfica del entorno y las acciones del agente. Si bien la visualización gráfica proporciona información adicional, su ejecución implica un consumo de tiempo considerable. Esta modificación no afecta la funcionalidad del algoritmo de entrenamiento, centrándose en la lógica y los cálculos necesarios para resolver el problema.

En conclusión, los datos iniciales del problema del Mountain Car fueron modificados para ofrecer un rango de observación más realista, un rango de acciones más adecuado y una recompensa que penaliza progresivamente los pasos dados por el agente. Además, se eliminó la visualización gráfica para optimizar el tiempo de ejecución en las pruebas. Estas modificaciones buscan mejorar la representación del problema y permitir una mayor eficiencia en el proceso de entrenamiento.

# Interacción con el simulador

Durante la interacción con el simulador, llevamos a cabo varias iteraciones para entrenar a nuestro agente utilizando la técnica de Q-Learning en la tarea de Mountain Car. Cada iteración consistió en ajustar y probar diferentes configuraciones de parámetros con el objetivo de mejorar el rendimiento del agente.

En nuestra primera iteración, decidimos utilizar una tasa de aprendizaje (learning rate) de 0.1 y un factor de descuento (discount factor) de 0.95. Estos valores nos permiten controlar la importancia de las recompensas futuras y ajustar la velocidad de aprendizaje del agente. Además, establecimos el número de episodios de entrenamiento en 1000, lo que nos brindó suficiente tiempo para que el agente aprendiera y mejorará su desempeño a lo largo del tiempo.

En términos de la estrategia de exploración y explotación, utilizamos una política epsilon-greedy, donde epsilon ( $\epsilon$ ) representa la probabilidad de exploración y la restante  $1-\epsilon$  representa la probabilidad de explotación. En esta iteración, establecimos  $\epsilon$  en 0.5, lo que significaba que el agente tenía un 50% de probabilidad de elegir una acción aleatoria (exploración) y un 50% de probabilidad de elegir la mejor acción según su función Q actual (explotación). Esta estrategia nos permitió equilibrar la exploración de nuevas acciones y la explotación de acciones ya conocidas.

Además, inicializamos la función Q con valores de 0 para todas las combinaciones de estados y acciones posibles. Esto proporcionó un punto de partida neutral para el agente, permitiéndole aprender y ajustar los valores de Q a medida que interactuaba con el simulador y recibía recompensas.

## Mejoras realizadas

Con los valores anteriores no logramos que el vehículo llegase a su destino, por lo que no obtuvimos ninguna mejora. Entonces decidimos hacer variables los valores de alfa y epsilon en cada iteración o episodio, al reducir sus valores logarítmicamente obtuvimos una mejora significativa en la solución, permitiendo así al agente terminar su tarea y respondiendo satisfactoriamente a los cambios de los demás valores como número de episodios o cantidad de pasos permitidos por episodio.

Los valores quedan de la siguiente manera.

```
alpha = max(0.01, min(0.5, 1.0 - math.log10((i + 1) / 25)))  
epsilon = max(0.01, min(1, 1.0 - math.log10((i + 1) / 25)))
```

## 100 Episodios

Ahora, podemos ver los resultados obtenidos con un numero de 100 episodios y manteniendo el resto de los valores idénticos.

```
...
Episode 99 Reward -500.0 Alpha 0.3979400086720376 Epsilon 0.3979400086720376
Mejor ejecución:
Número de ejecución: 0
Recompensa: -500.0
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

Aquí apreciamos perfectamente que no logró llegar a la solución, y vemos que Alfa y Epsilon se detuvieron tempranamente, por lo que podemos decir que no tuvo el suficiente tiempo para poder explorar.

Procederemos a darle unos 200 episodios, aquí ya deberíamos de ver resultados favorables.

## 200 Episodios

```
Episode 192 Reward -197.0 Alpha 0.11230205300420304 Epsilon 0.11230205300420304
Episode 193 Reward -314.0 Alpha 0.11013827874181159 Epsilon 0.11013827874181159
Episode 194 Reward -217.0 Alpha 0.10790539730951965 Epsilon 0.10790539730951965
Episode 195 Reward -233.0 Alpha 0.10568393731556158 Epsilon 0.10568393731556158
Episode 196 Reward -142.0 Alpha 0.1034737825104447 Epsilon 0.1034737825104447
Episode 197 Reward -226.0 Alpha 0.10127481841050645 Epsilon 0.10127481841050645
Episode 198 Reward -239.0 Alpha 0.099086932262331 Epsilon 0.099086932262331
Episode 199 Reward -302.0 Alpha 0.09691001300805646 Epsilon 0.09691001300805646
Mejor ejecución:
Número de ejecución: 196
Recompensa: -142.0
```

+ Code + Markdown

Ahora podemos apreciar que el agente logró cumplir con la tarea asignada, vemos que los valores de Alfa y Epsilon disminuyeron considerablemente con respecto al anterior caso, y también vemos que el mejor caso fue el 196. Aquí podemos interpretarlo como que el agente estaba entrando al rango de mejores soluciones, por lo que podemos asumir que si le damos más episodios podrá resolverlo de una mejor forma.

Entendemos que el número de episodios debe de aumentar bruscamente si queremos ver resultados, por lo que probaremos con 500.



## 500 Episodios

```
Episode 492 Reward -117.0 Alpha 0.01 Epsilon 0.01
Episode 493 Reward -472.0 Alpha 0.01 Epsilon 0.01
Episode 494 Reward -113.0 Alpha 0.01 Epsilon 0.01
Episode 495 Reward -113.0 Alpha 0.01 Epsilon 0.01
Episode 496 Reward -112.0 Alpha 0.01 Epsilon 0.01
Episode 497 Reward -500.0 Alpha 0.01 Epsilon 0.01
Episode 498 Reward -118.0 Alpha 0.01 Epsilon 0.01
Episode 499 Reward -165.0 Alpha 0.01 Epsilon 0.01
Mejor ejecución:
Número de ejecución: 496
Recompensa: -112.0
```

Vemos una mejora, y es como lo esperábamos anteriormente, por lo que le daremos unos 5000 episodios para intentar mejorar este valor, aunque asumimos que desde este punto las mejoras serán cada vez menores.

## 5000 Episodios

```
Episode 4992 Reward -156.0 Alpha 0.01 Epsilon 0.01
Episode 4993 Reward -152.0 Alpha 0.01 Epsilon 0.01
Episode 4994 Reward -144.0 Alpha 0.01 Epsilon 0.01
Episode 4995 Reward -500.0 Alpha 0.01 Epsilon 0.01
Episode 4996 Reward -148.0 Alpha 0.01 Epsilon 0.01
Episode 4997 Reward -148.0 Alpha 0.01 Epsilon 0.01
Episode 4998 Reward -148.0 Alpha 0.01 Epsilon 0.01
Episode 4999 Reward -153.0 Alpha 0.01 Epsilon 0.01
Mejor ejecución:
Número de ejecución: 2757
Recompensa: -89.0
```

Logramos mejorar la solución, aunque fue una mejora considerable, no es una tan notoria como antes. Entonces si queremos mejorar el agente deberíamos de hacer cambios más inteligentes que simplemente aumentar los episodios. Aunque para probar nuestro punto, haremos una corrida ridículamente grande para demostrarlo y probaremos 100.000 episodios. Esperamos una mejora, pero no considerable, por lo que intentaremos variar otros valores para mejorar nuestra solución.

## 100.000 Episodios

```
Episode 99990 Reward -500.0 Alpha 0.01 Epsilon 0.01
Episode 99991 Reward -175.0 Alpha 0.01 Epsilon 0.01
Episode 99992 Reward -164.0 Alpha 0.01 Epsilon 0.01
Episode 99993 Reward -500.0 Alpha 0.01 Epsilon 0.01
Episode 99994 Reward -170.0 Alpha 0.01 Epsilon 0.01
Episode 99995 Reward -157.0 Alpha 0.01 Epsilon 0.01
Episode 99996 Reward -360.0 Alpha 0.01 Epsilon 0.01
Episode 99997 Reward -160.0 Alpha 0.01 Epsilon 0.01
Episode 99998 Reward -500.0 Alpha 0.01 Epsilon 0.01
Episode 99999 Reward -167.0 Alpha 0.01 Epsilon 0.01
Mejor ejecución:
Número de ejecución: 24032
Recompensa: -83.0
```

Aquí podemos observar perfectamente como después del episodio 24032 no hubo mejora alguna, podría producirse pero es probable que tarde mucho en mejorar un solo paso.

Con los datos obtenidos en las ejecuciones anteriores entendemos que tenemos un tiempo de ejecución con este modelo, sin entorno gráfico y con las siguientes especificaciones de computadora, un tiempo de 4 minutos por 10.000 episodios de 5000 pasos cada uno.

```
Operating System: Windows 10 Pro 64-bit (10.0, Build 19045)
Language: English (Regional Setting: English)
System Manufacturer: Acer
System Model: AN515-51
BIOS: V1.22
Processor: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz (8 CPUs), ~2.8GHz
Memory: 16384MB RAM
Page file: 12600MB used, 6099MB available

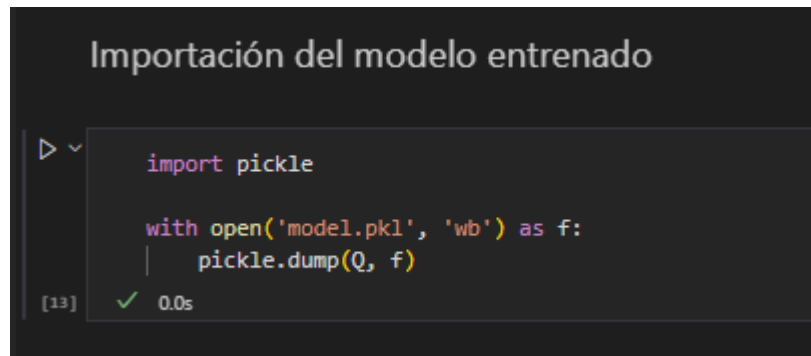
Name: NVIDIA GeForce GTX 1050 Ti
Manufacturer: NVIDIA
Chip Type: NVIDIA GeForce GTX 1050 Ti
DAC Type: Integrated RAMDAC
Device Type: Full Display Device
Approx. Total Memory: 12150 MB
Display Memory (VRAM): 4017 MB
Shared Memory: 8133 MB
```

Tenemos que tener en cuenta que el viaje que se plantea dura al menos 300 días, en este caso vemos que aunque sea poco, el agente podrá mejorar los resultados si utiliza la totalidad del tiempo, esto es un detalle importante a destacar.

# Generación de modelo computado

Se realizó un entrenamiento del modelo el cual cuenta con diez mil episodios y con los parámetros mencionados anteriormente.

El modelo entrenado se generó con el código mostrado a continuación podremos generar un archivo .pkl el cual nos permitirá usar un modelo previamente entrenado.



```
Importación del modelo entrenado

import pickle

with open('model.pkl', 'wb') as f:
    pickle.dump(Q, f)

[13] ✓ 0.0s
```

# Connect-Four

Para encarar este nuevo desafío se tomó la decisión de crear un agente **Minimax**, el cual hereda de la clase **Agent**, en este agente implementamos los métodos `next_action` y `heuristic_utility`.

El método `next_action` se encarga de leer la entrada del jugador, validar este movimiento y realizar la jugada.

El método `heuristic_utility` recibe el tablero, cuenta en primer lugar con el caso base que se encarga de verificar que exista un jugador ganador y además de verificar que el tablero esté completo, este último caso únicamente se toma en cuenta si no existe un ganador al completarse el mismo.

Si el tablero está completo significa que hubo un empate.

Luego se obtienen las posibles acciones a realizar, es decir los lugares libres de cualquier ficha y que se encuentren dentro del tablero. Y luego para cada movimiento válido posible se clona el tablero, se agrega la ficha correspondiente al jugador que posee el turno y se calcula de forma recursiva la utilidad (score) con el tablero clonado.

Por último se valida si el jugador que está jugando es el agente, en caso de ser verdadero se actualiza el score, al cual se le da el máximo valor calculado entre el obtenido por la heurística y el valor previo.

```
best_score = max(score, best_score)
```

En el caso de que el jugador actual sea el oponente, se actualizará el score al cual se le da el mínimo valor calculado entre lo obtenido por la heurística y el valor del score anterior como se muestra a continuación.

```
best_score = min(score, best_score)
```

Finalmente el método **`heuristic_utility`** retornara el **`best_score`**.

El algoritmo genera el árbol del juego completo y calcula la utilidad para los estados terminales aplicando la función heurística.

Se modificó la notebook Connect\_four.ipynb cambiando el agente utilizado. En la versión brindada el agente que jugaba era el InputAgent y ahora se utilizará el AgentMinimax. Los dos agentes que jugarán serán el implementado por nosotros contra el agente spaceGPT\_agent\_obf que fue brindado.

Para utilizar el nuestro agente implementado, se asigna como primer agente el implementado por nosotros:

```
input_agent = AgentMinimax()
```

Y finalmente se comienza a jugar:

```
from play import play_vs_other_agent, play_vs_loaded_agent

play_vs_loaded_agent(env, agent=input_agent)
```

El método play\_vs\_loaded\_agent asigna como oponente el agente spaceGPT\_agent\_obf que fue brindado:

```
def play_vs_loaded_agent(env, agent):
    enemy_agent = load_enemy_agent()
    play_vs_other_agent(env, agent, enemy_agent)

def load_enemy_agent():
    return SpaceGPTAgent(2, 4)
```

# Ejemplo de Juego con el modelo

