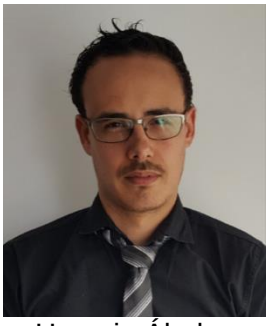
Universidad ORT

Inteligencia Artificial

Entrega de Obligatorio


Federico Alonso


Horacio Ábalos

12 de diciembre de 2022

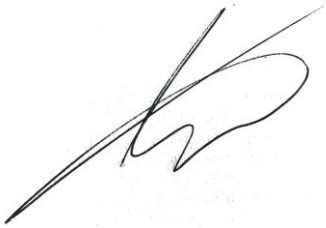
Link al Repositorio

Contenido

Declaración de autoría:

Nosotros, Horacio Ábalos y Federico Alonso, declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que:

a) La obra fue producida en su totalidad mientras realizábamos el obligatorio de Inteligencia Artificial;
b) Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
c) Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
d) En la obra, hemos acusado recibo de las ayudas recibidas;
e) Cuando la obra se basa en trabajo realizado juntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;
f) Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.



Alonso Fuentes, Federico Nicolás
12 de diciembre de 2022

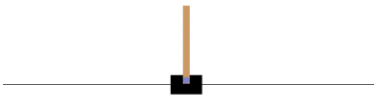
Abalos Cambre, Horacio Mathías
12 de diciembre de 2022

Introducción

El presente documento se brinda como especificación técnica y explicación del trabajo realizado por el equipo para la entrega del obligatorio de la materia Inteligencia Artificial. La propuesta recibida solicita resolver los problemas *2048* y *CartpoleV1*.

Cartpole

OpenAI nos propone el ejercicio de conservar verticalmente un objeto con forma de palo sobre un carro que solamente puede moverse en dos dimensiones, izquierda (0) y derecha (1). Dado que en todo momento existen dos posibles acciones (izquierda y derecha), la recompensa asociada a cada paso del entorno será +1 y, según la documentación del OpenAI, consideraremos resuelto el problema cuando se obtenga una recompensa acumulada de 500.



Para este ejercicio el espacio de observación es el siguiente

Action Space	Discrete(2)
Observation Shape	(4,)
Observation High	[4.8 inf 0.42 inf]
Observation Low	[-4.8 -inf -0.42 -inf]

Para información más detallada se sugiere visitar la documentación de OpenAI

2048

En este ejercicio, lo que se solicita es crear, mediante las técnicas aprendidas en clase, un agente que sea capaz de solucionar el juego 2048.

Para lo mismo contamos con las técnicas de suma 0 prendidas:

- MiniMax
- ExpectiMax

En ambos casos debemos contemplar el juego como una competencia entre 2 jugadores, uno será nuestro algoritmo y el otro el juego en sí.

Para resolver el mismo, se nos brinda el modelo del juego, y la solución de un agente randómico para que a partir de ahí podamos comenzar.

Adelante en el documento especificaremos todo el trabajo realizado por el equipo.

Ejecución de Agentes

CartPole

La solución de este ejercicio propone varios agentes que se utilizan para distintas finalidades:

- AgentCartAndPoleDimensions.py
  - Finalidad: Generar *Qtables* con las cuatro dimensiones de observación guardándolas en la carpeta V6.
  - python AgentCartAndPoleDimensions.py
  - Requerimientos:
    - Ejecutar el comando en la ubicación ../ Carpole_last_version
- AgenteSimCartAndPole.py
  - Finalidad: Similar juego de basado en *Qtables* generadas por el agente AgentCartAndPoleDimensions.py
  - python AgenteSimCartAndPole.py
  - Requerimientos:
    - Ejecutar el comando en la ubicación ../ Carpole_last_version
    - Especificar la cantidad de episodios a entrenar en el propio archivo modificando la variable episodios
    - Especificar la *Qtable* que se desea usar
    - El archivo de la *Qtable* debe estar situado en la misma ubicación que el agente
- AgentPoleDimensions.py
  - Finalidad: Generar *Qtables* solamente considerando las dimensiones de observación del pole guardándolas en la carpeta V5.
  - python AgentPoleDimensions.py
  - Requerimientos:
    - Ejecutar el comando en la ubicación ../ Carpole_last_version
- AgentSimulatorPole.py
  - Finalidad: Similar juego de basado en *Qtables* generadas por el agente AgentPoleDimensions.py mostrando la renderización del ambiente
  - python AgentSimulatorPole.py
  - Requerimientos:
    - Ejecutar el comando en la ubicación ../ Carpole_last_version
    - Especificar la cantidad de episodios a entrenar en el propio archivo modificando la variable episodios
    - Especificar la *Qtable* que se desea usar
    - El archivo de la *Qtable* debe estar situado en la misma ubicación que el agente
- FastAgentSimulator.py
  - Finalidad: Similar juego de basado en *Qtables* generadas por el agente AgentPoleDimensions.py sin visualizar la renderización del ambiente e imprimiendo estadísticas solamente al concluir la simulación
  - python FastAgentSimulator.py
  - Requerimientos:
    - Ejecutar el comando en la ubicación ../ Carpole_last_version
    - Especificar la cantidad de episodios a entrenar en el propio archivo modificando la variable episodios
    - Especificar la *Qtable* que se desea usar

- El archivo de la *Qtable* debe estar situado en la misma ubicación que el agente

Experimentamos ciertas dificultades en cuanto a la compatibilidad del ambiente causado por una actualización de gym cuya versión requrida es la 0.26.2.

En caso de suceder algún error se recomienda la desintalación (pip uninstall gym) y la instalación nuevamente (pip install gym) de gym.

2048

En el ejercicio 2048, dejamos preparado el agente ExpectiMax, con una profundidad de 5 niveles. En nuestra experiencia tiene una duración de entre 4 y 5 minutos. Para ejecutarlo basta con correr desde una consola el siguiente comando:

python Main.py

Lo que resultará en 3 ejecuciones del programa con el mejor agente que logramos programar de forma consecutiva. Una vez finalizado, nos generará un archivo de texto: EMAgent-d5.txt que contiene los detalles resumidos de cada ejecución y un resumen final.

```
INFO:root:
Total Moves: 1003
INFO:root:"WON THE GAME!!!!!!!!"
**********************************************************
INFO:root:

Round number "20"
INFO:root:
Total time: 0:03:56.031125
INFO:root:
Total Moves: 978
INFO:root:"WON THE GAME!!!!!!!!"
**********************************************************
INFO:root:"Results"**********************************************************
INFO:root:\Average Wons: 85.0%
INFO:root:\Average Moves: 1010.55
INFO:root:\Average Time: 255.88180115
```

CartPole

Análisis del tipo de solución y metodología de trabajo

Entendemos que en el problema propuesto no existen datos sobre los cuales un agente pueda aprender y, considerando que solamente disponemos de un entorno sobre el que podemos realizar acciones, el enfoque pertinente para esta solución es *Reinforcement Learning*.

En lo referente a la metodología de trabajo, utilizaremos una aproximación iterativa incremental, a través de la cual buscaremos dividir las tareas en subtareas más pequeñas sobre las cuales podamos tomar decisiones de mejora de manera específica. De esta forma, lograremos obtener una solución para posteriormente intentar mejorarla.[1]

Versión 1

Esta iteración simplemente ofrece las condiciones de trabajo para comenzar con una aproximación más refinada a partir de las siguientes.

Puntualmente se establece una meta muy corta, pero fundamental: el algoritmo funciona y guarda los valores en la tabla que almacena los valores de Q considerando todas las dimensiones de la observación (velocidad angular, posición del carro, velocidad del carro, ángulo del palo).

Se comienza a partir de la versión 2 con el análisis de entrenamiento.

Versión 2

El algoritmo utiliza ciertos valores (bien podrían ser parametrizables), los cuales comienzan arbitrariamente siendo:

a) *Learning rate* 0.1
b) *Discount* 0.85
c) Discretización de los espacios de observación (las 4 dimensiones) en 50 *buckets* (100 exceden la memoria utilizable por la ejecución)

A partir de los valores anteriores se verifica que no existen problemas de compilación ni de recursos de hardware para comenzar con un entrenamiento extenso. Por este motivo se decide implementar un entrenamiento de 10.000.000.000 (diez mil millones de episodios).

Cada cierta cantidad de iteraciones se guardará una *Qtable* para analizar un desempeño utilizando un agente que simule el juego.

Luego de aguardar la ejecución durante 2 días se decide suspender el entrenamiento (transcurridos 80 millones de episodios) y se procede a ejecutar la simulación.

No se dispone de una impresión de pantalla de esta *performance*, pero utilizando la última *Qtable* generada (a los 80 millones de episodios de entrenamiento) se obtiene un *reward* promedio superior a 9 en la simulación de 100 episodios de juego. Esto significa que el desempeño apenas fue superior que a las acciones absolutamente elegidas de manera aleatoria.

[1] Cada una de las versiones mencionadas en este documento se relaciona con una rama del repositorio dónde se pueden encontrar los recursos de cada etapa (Agente, Agente para simular el juego, *Qtables* generadas en el entrenamiento, graficas, etc.).

Analizando el desempeño obtenido, es claro que una discretización de semejante tamaño no es conveniente ya que, en una tabla de 4 dimensiones de estas características más las 2 propias de cada acción disponible, genera un universo de 12.500.000 posibles valores de Q. Inclusive considerando que se generaron 80 millones de episodios de entrenamiento, puede existir la posibilidad que alguna combinación de los estados discretos no haya sido observada.

Como consecuencia lógica, se intentará reducir el tamaño de la discretización de las observaciones.

Versión 3

Comienza el ciclo de entrenamientos con la discretización reducida a 20 *buckets* por dimensión de observación (se reduce el espacio de posibles combinaciones a 320.000).

Adicionalmente, se decide implementar sesiones de entrenamiento menos extensas que en la versión 2 para que las iteraciones sean mas cortas y significativas (de 1 a 10 millones).

Se realiza la simulación del juego y se obtiene exactamente el mismo resultado que en la versión 2.

Utilizando la Qtable *5600000-qtable.npy* se obtiene el siguiente desempeño en 10 episodios de simulación:

```
Reward Total en episodio  0 :  8.0
Reward Total en episodio  1 :  10.0
Reward Total en episodio  2 :  10.0
Reward Total en episodio  3 :  10.0
Reward Total en episodio  4 :  10.0
Reward Total en episodio  5 :  8.0
Reward Total en episodio  6 :  10.0
Reward Total en episodio  7 :  9.0
Reward Total en episodio  8 :  10.0
Reward Total en episodio  9 :  9.0
```

Imagen 1 - Desempeño de 10 episodios simulados con 5600000-qtable.npy

Lógicamente este no es el resultado esperado, por lo que se verifica que la *Qtable* no fue correctamente generada. A continuación, se expone uno de los episodios simulados imprimiento (a) la acción realizada y (b) la *Qtabe* en el estado discreto de cada paso, así como (c) el *reward* total obtenido en el episodio:

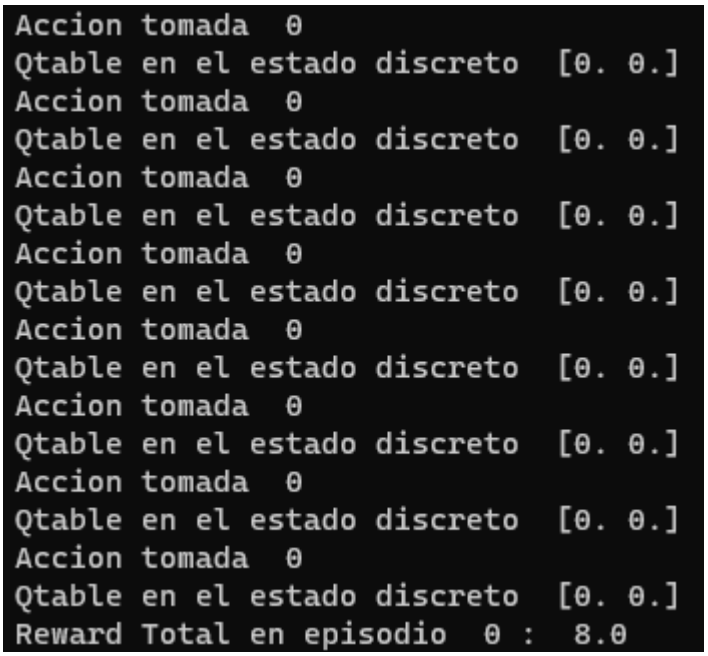


Imagen 2 - Detalle de la simulación con 5600000-qtable.npy

Resulta evidente que se debe modificar el código por lo que se concluye esta iteración dejando los siguientes puntos de mejora para la iteración 4:

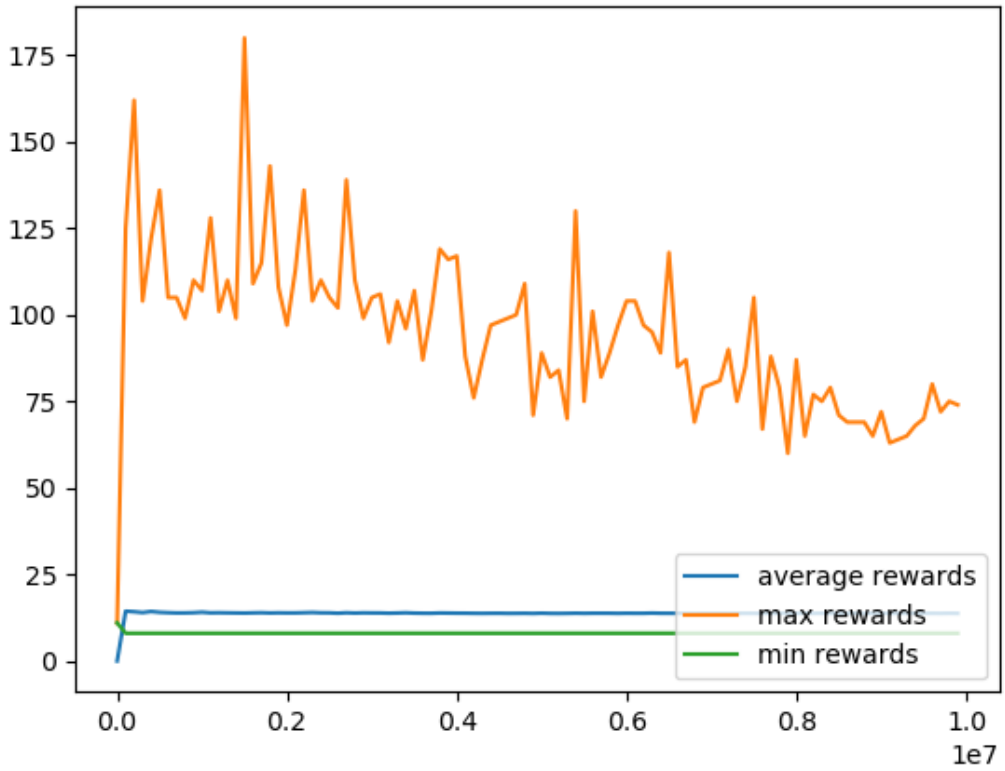
a) Verificar que la *Qtable* esté inicializándose y calculándose correctamente
b) Modificar el *Learning Rate* y *Discount* utilizados
c) Incluir algún tipo de métrica para analizar el entrenamiento realizado

Versión 4

Comienza la revisión del código y se confirma que no estaba calculándose, inicializándose y guardándose correctamente la *Qtable*. Una vez corregido el error se procede a modificar los parámetros de *Learning Rate* y *Discount* (representando el valor asignado a la importancia de los episodios futuros frente a los actuales), ambos para varias combinaciones pertenecientes al intervalo (0;1].

Se incluye código para generar métricas de *rewards* máximas, mínimas y promedio obtenidas en base a los episodios de entrenamiento, se guardan las *Qtables* sincronizadamente de forma que observando el grafico pueda ser posible utilizar la *Qtable* asociada al mejor desempeño.

Una vez introducidos los cambios se ejecuta un entrenamiento de 10 millones de episodios, pero no se observa una mejoría en el desempeño del modelo ya que en promedio los *rewards* continúan siendo bajos.



Gráfica 1 - Rewards máximo, promedio y mínimo para la versión 2 con 2 millones de episodios de entrenamiento

La Gráfica 1 nos muestra que el mejor *reward* máximo obtenido se produce aproximadamente a los 2 millones de episodios pero que posteriormente el modelo empeora su desempeño.

Se corre una iteración de 100 episodios con la *Qtable* guardada a los 100 mil y a los 2 millones de episodios, así como con la última generada a los 10 millones. En todos los casos se observa que el *reward* promedio es 10. Se adjunta solamente una de las impresiones (todas las obtenidas eran exactamente iguales) de la simulación con 100 episodios. La impresión corresponde al último episodio:

```
[6.66666667 6.66666667]
[6.66666667 6.66666667]
[6.66666667 6.66666667]
[6.66666667 6.66666667]
[6.66666667 6.66666667]
[6.66666667 6.66666667]
[6.66666667 6.66666667]
[6.66666667 6.66666667]
[6.66666667 6.66666667]
Reward Total en episodio  99 :  9.0
```

Imagen 3 - Impresion de Qtable[discrete_state] para la tabla generada a los 2 millones de episodios

Se observa que ambas acciones tienen la misma “consecuencia” para nuestro modelo dado que tienen el mismo valor asociado, esto genera algo de desconcierto dado que es evidente que no puede resultar el mismo valor para una determinada observación, es decir: suponiendo que el carro se este desplazado a la derecha (sin importar su ubicación), con el palo en ángulo mayor a 0 y velocidad angular positiva, no puede ser lógicamente admisible que el modelo entienda indistinta cualquier acción.

De lo anterior se desprende que, entre otras cosas que puedan alterar el funcionamiento, o bien:

a) el código no funciona como se espera en lo referente a la *Qtable* (sin embargo, parece estar calculando y guardando valores);
b) la cantidad de episodios de entrenamiento no fue suficiente (esto básicamente siempre es real);
c) se está introduciendo mucho ruido a través de dimensiones de observaciones que no son realmente necesarias

Claro está que para una primera versión de la solución se podría solamente considerar el ángulo del palo y su velocidad angular, ya que la velocidad del carro y su posición parecen no ser tan relevantes para que el juego acabe. Si bien esto es una simplificación, entendemos que es válida para acercarnos a una solución que funcione (o al menos mejore el resultado hasta ahora obtenido) y posteriormente incluir las restantes dimensiones de observación.

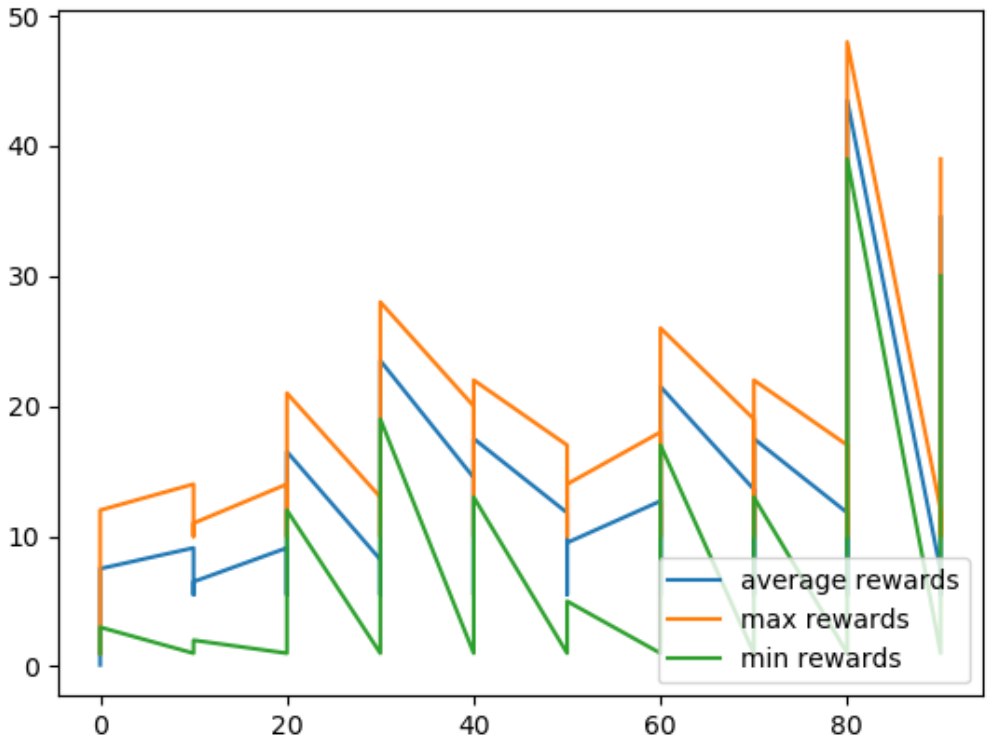
Versión 5 - AgentPoleDimensions

Considerando que la *Qtable* de la versión anterior estaba cargada con el mismo valor para ambas acciones, decidimos recomenzar el código. A partir de este momento, modificamos la implementación de la discretización para utilizar *KBinsDiscretizer* importándolo desde *sklearn.preprocessing* y el método de decrementar los valores de *Learning Rate* y *Eploration Rate* basados en una solución propuesta en el tutorial *Balancing CartPole with Machine Learning, Python* (Saito, 2018); a su vez las funciones de para decrementar ambos valores fueron basadas en el tutorial *OpenAI Gym: CartPole-v1 - Q-Learning* (Brooker, 2020).

Conforme mencionado anteriormente, debido a que se percibe que el modelo no está aprendiendo de las iteraciones realizadas, se decide comenzar nuevamente, cambiando el enfoque para una solución que solamente considere la velocidad angular y ángulo del *pole* para, posteriormente, incluir la velocidad y ubicación del *cart* en las dimensiones de la *Qtable*.

El código restante se reescribe con mayor atención a la reutilización y considerando criterios de mantenibilidad para su posterior modificación y evolución.

Se procede a graficar los *rewards* obtenidos y, tras correr una iteración de entrenamiento de apenas 100 episodios, se genera la Grafica XX que nos muestra claramente como los *rewards* promedio y mínimo modificaron el patrón hasta ahora percibido y pasan a acompañar al *reward* máximo.



Gráfica 2 - Rewards máximo, promedio y mínimo para la version 5 en 100 episodios de entrenamiento

A partir de la gráfica, se selecciona la *Qtable* guardada en la iteración 80. Al correr una simulación de 100 episodios se obtiene:

- Episodio: 0 Reward obtenido: 29.0
- Episodio: 1 Reward obtenido: 14.0
- Episodio: 2 Reward obtenido: 31.0
- Episodio: 3 Reward obtenido: 39.0
- Episodio: 4 Reward obtenido: 14.0
- Episodio: 5 Reward obtenido: 37.0
- Episodio: 6 Reward obtenido: 45.0
- Episodio: 7 Reward obtenido: 37.0
- Episodio: 8 Reward obtenido: 15.0
- Episodio: 9 Reward obtenido: 63.0
- Episodio: 10 Reward obtenido: 42.0
- Episodio: 11 Reward obtenido: 27.0
- Episodio: 12 Reward obtenido: 37.0
- Episodio: 13 Reward obtenido: 93.0
- Episodio: 14 Reward obtenido: 27.0
- Episodio: 15 Reward obtenido: 35.0
- Episodio: 16 Reward obtenido: 51.0
- Episodio: 17 Reward obtenido: 35.0
- Episodio: 18 Reward obtenido: 71.0
- Episodio: 19 Reward obtenido: 39.0
- Episodio: 20 Reward obtenido: 41.0

- Episodio: 21 Reward obtenido: 46.0
- Episodio: 22 Reward obtenido: 27.0
- Episodio: 23 Reward obtenido: 78.0
- Episodio: 24 Reward obtenido: 76.0
- Episodio: 25 Reward obtenido: 39.0
- Episodio: 26 Reward obtenido: 17.0
- Episodio: 27 Reward obtenido: 126.0
- Episodio: 28 Reward obtenido: 63.0
- Episodio: 29 Reward obtenido: 33.0
- Episodio: 30 Reward obtenido: 15.0
- Episodio: 31 Reward obtenido: 41.0
- Episodio: 32 Reward obtenido: 44.0
- Episodio: 33 Reward obtenido: 78.0
- Episodio: 34 Reward obtenido: 33.0
- Episodio: 35 Reward obtenido: 43.0
- Episodio: 36 Reward obtenido: 54.0
- Episodio: 37 Reward obtenido: 43.0
- Episodio: 38 Reward obtenido: 45.0
- Episodio: 39 Reward obtenido: 31.0
- Episodio: 40 Reward obtenido: 56.0
- Episodio: 41 Reward obtenido: 73.0
- Episodio: 42 Reward obtenido: 50.0
- Episodio: 43 Reward obtenido: 59.0
- Episodio: 44 Reward obtenido: 52.0
- Episodio: 45 Reward obtenido: 15.0
- Episodio: 46 Reward obtenido: 82.0
- Episodio: 47 Reward obtenido: 49.0
- Episodio: 48 Reward obtenido: 37.0
- Episodio: 49 Reward obtenido: 33.0
- Episodio: 50 Reward obtenido: 27.0
- Episodio: 51 Reward obtenido: 55.0
- Episodio: 52 Reward obtenido: 29.0
- Episodio: 53 Reward obtenido: 33.0
- Episodio: 54 Reward obtenido: 46.0
- Episodio: 55 Reward obtenido: 33.0
- Episodio: 56 Reward obtenido: 68.0
- Episodio: 57 Reward obtenido: 71.0
- Episodio: 58 Reward obtenido: 37.0
- Episodio: 59 Reward obtenido: 60.0
- Episodio: 60 Reward obtenido: 31.0
- Episodio: 61 Reward obtenido: 37.0
- Episodio: 62 Reward obtenido: 50.0
- Episodio: 63 Reward obtenido: 16.0
- Episodio: 64 Reward obtenido: 37.0
- Episodio: 65 Reward obtenido: 31.0
- Episodio: 66 Reward obtenido: 53.0
- Episodio: 67 Reward obtenido: 59.0
- Episodio: 68 Reward obtenido: 55.0

- Episodio: 69 Reward obtenido: 61.0
- Episodio: 70 Reward obtenido: 33.0
- Episodio: 71 Reward obtenido: 51.0
- Episodio: 72 Reward obtenido: 37.0
- Episodio: 73 Reward obtenido: 47.0
- Episodio: 74 Reward obtenido: 31.0
- Episodio: 75 Reward obtenido: 33.0
- Episodio: 76 Reward obtenido: 35.0
- Episodio: 77 Reward obtenido: 35.0
- Episodio: 78 Reward obtenido: 40.0
- Episodio: 79 Reward obtenido: 41.0
- Episodio: 80 Reward obtenido: 69.0
- Episodio: 81 Reward obtenido: 29.0
- Episodio: 82 Reward obtenido: 31.0
- Episodio: 83 Reward obtenido: 60.0
- Episodio: 84 Reward obtenido: 12.0
- Episodio: 85 Reward obtenido: 41.0
- Episodio: 86 Reward obtenido: 19.0
- Episodio: 87 Reward obtenido: 27.0
- Episodio: 88 Reward obtenido: 29.0
- Episodio: 89 Reward obtenido: 45.0
- Episodio: 90 Reward obtenido: 33.0
- Episodio: 91 Reward obtenido: 17.0
- Episodio: 92 Reward obtenido: 33.0
- Episodio: 93 Reward obtenido: 118.0
- Episodio: 94 Reward obtenido: 26.0
- Episodio: 95 Reward obtenido: 37.0
- Episodio: 96 Reward obtenido: 37.0
- Episodio: 97 Reward obtenido: 39.0
- Episodio: 98 Reward obtenido: 35.0
- Episodio: 99 Reward obtenido: 56.0

Los resultados de este enfoque son mucho más alentadores por lo que se procede a entrenar 2 mil episodios para posteriormente verificar su desempeño. Durante la ejecución, se observa que el entrenamiento arroja un episodio que es truncado al superar la cantidad de *reward* total obtenida superior a lo solicitado por el ambiente (500). Esto sucede a los 154 episodios de entrenamiento, por lo que se decide utilizar la *Qtable* generada (se había incluido en el código una condición para guardar la *Qtable* si el episodio de entrenamiento era truncado por el ambiente, es decir, si el agente había ganado el juego). Al correr 150 episodios de simulación con la *Qtable* anterior se obtuvo el siguiente resultado:

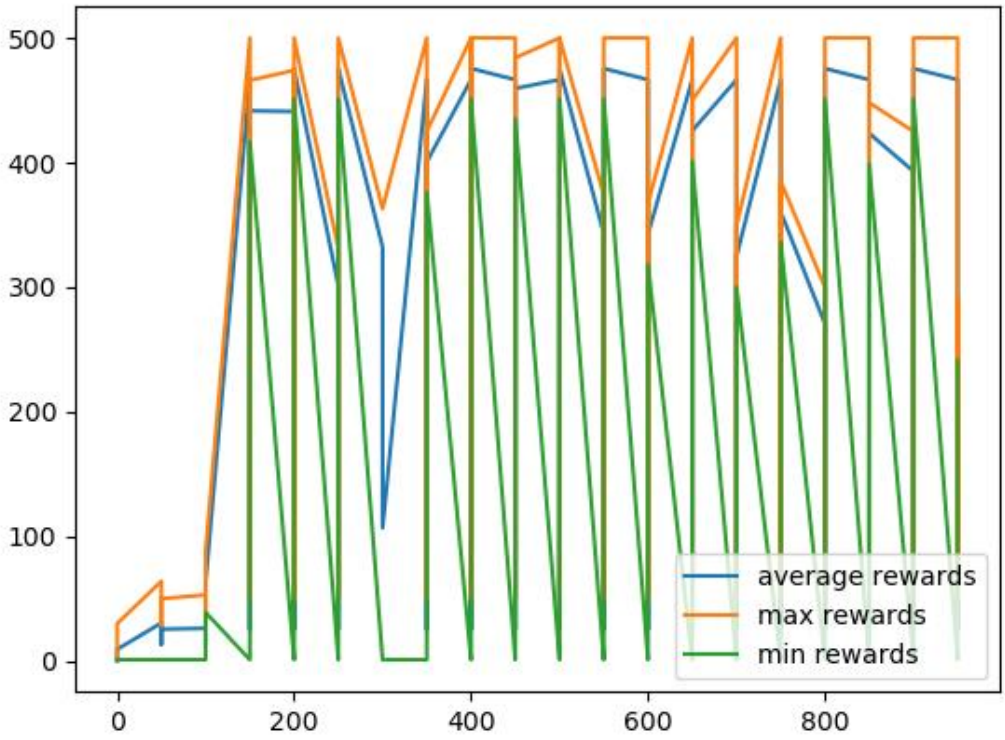
```
GANO!!!
Episodio:  140  Reward obtenido:  683.0
Episodio:  141  Reward obtenido:  466.0
GANO!!!
Episodio:  142  Reward obtenido:  745.0
GANO!!!
Episodio:  143  Reward obtenido:  644.0
GANO!!!
Episodio:  144  Reward obtenido:  885.0
GANO!!!
Episodio:  145  Reward obtenido:  674.0
GANO!!!
Episodio:  146  Reward obtenido:  788.0
GANO!!!
Episodio:  147  Reward obtenido:  661.0
GANO!!!
Episodio:  148  Reward obtenido:  778.0
GANO!!!
Episodio:  149  Reward obtenido:  833.0
Promedio de rewards:  749.9333333333333
Se ganaron:  142  veces!
Promedio de simulaciones ganadas  749.9333333333333
```

Imagen 4 - Resultado de simulación con Qtable generada a los 154 episodios de entrenamiento de la versión 5

Como podemos apreciar, se perdieron solamente 3 de los 150 episodios generados, esto induce a pensar que el error cometido es del 2% (lógicamente solo considerando los 150 episodios simulados).

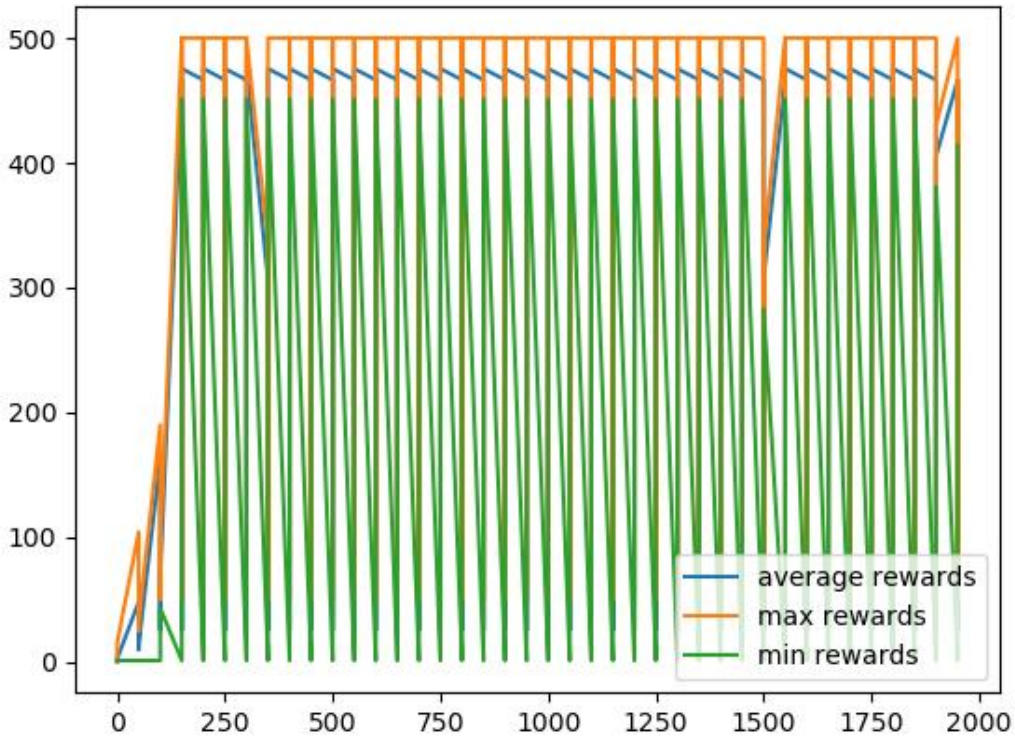
Se observa un comportamiento particular: No se toma ninguna acción para que el carro no se exceda de los límites permitidos. Esto se deriva de omitir la posición del carro en las dimensiones de observación utilizada, tarea reservada para la versión 6.

A continuación, se adjunta la gráfica del entrenamiento realizado con 1000 iteraciones:



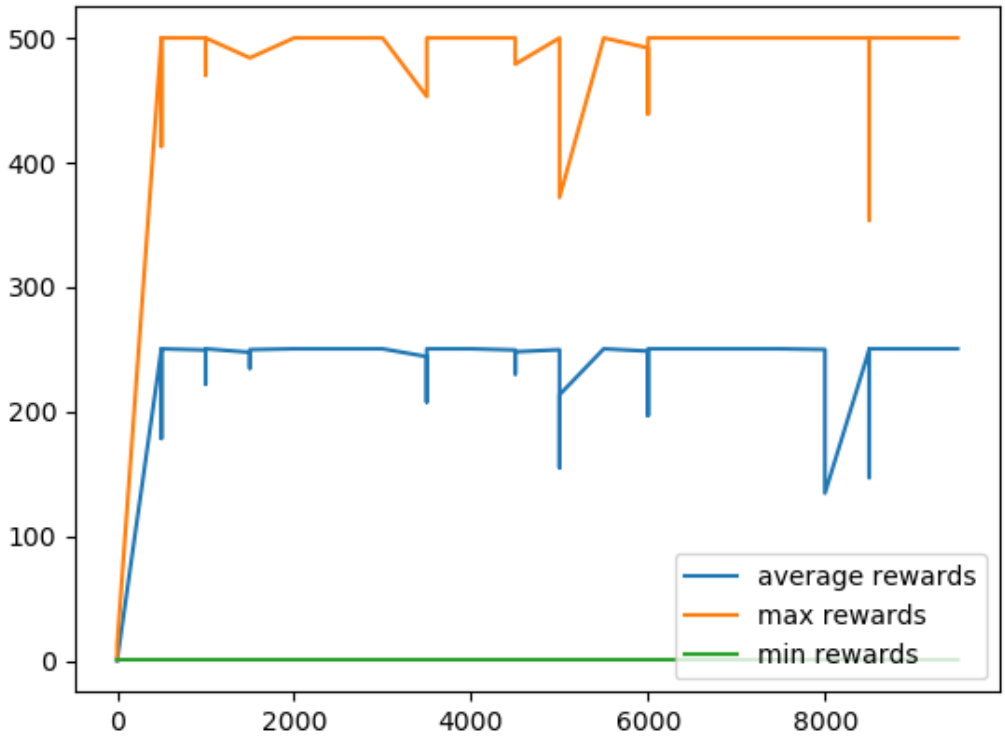
Gráfica 3 - Rewards máximo, promedio y mínimo en 2000 episodios de entrenamiento de la versión 5

A continuación, se adjunta la gráfica del entrenamiento realizado con 2000 iteraciones:



Gráfica 4 - Rewards máximo, promedio y mínimo en 2000 episodios de entrenamiento de la versión 5

Se realiza también un entrenamiento de 10000 iteraciones para verificar si el desempeño mejora con más entrenamiento, grafica adjunta:



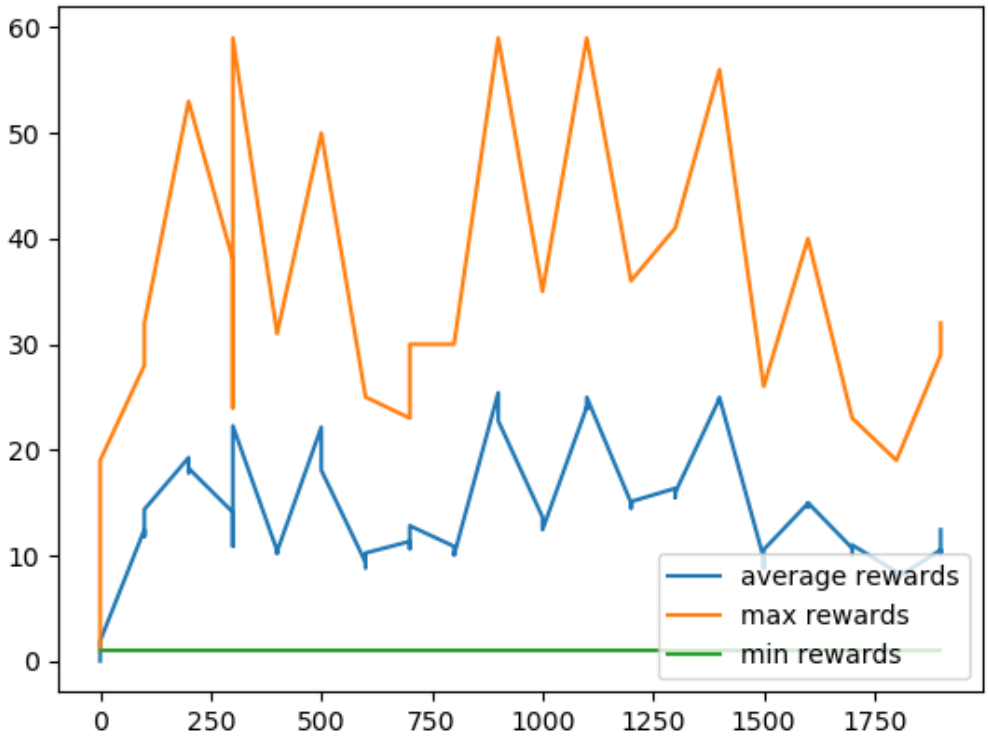
Gráfica 5 - Reward máximo, promedio y mínimo para 10000 iteraciones de entrenamiento en la versión 5

Si bien existe una mejora y conocemos la existencia de máximos y mínimos locales, no se puede observar en las gráficas 3, 4 y 5 un cambio tan significativo entre el resultado de los 1000, 2000 o 10000 episodios entrenados. Se entiende que sería conveniente entrenar el modelo con un numero de iteraciones muy superior (en el orden de los 20 millones) para verificar la existencia de posibles valores locales en los que el algoritmo se pueda haber estancado, y para maximizar la ocurrencia de las observaciones de los episodios en el espacio muestral.

La versión 5 se considera una versión entregable de este desafío, pero se intentará evolucionar la entrega a una versión 6 que considere la velocidad del carro y su posición con el objetivo de no permitir que se trunque el episodio por la posición. Adicionalmente, es un desafío interesante conseguir simular solamente un episodio que sea truncado únicamente por un tiempo máximo a ser determinado.

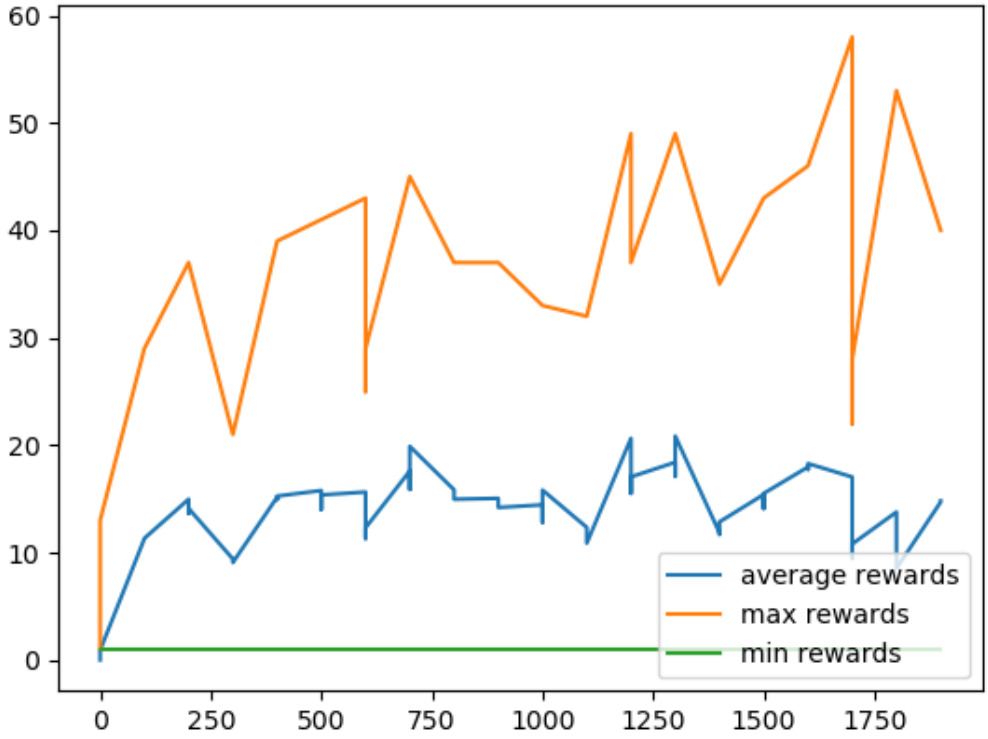
Versión 5.1 - Modificación de la discretización

Se intenta ajustar la discretización para un refinamiento más fino con 24 *buckets* en cada dimensión observada (velocidad angular y ángulo del pole).



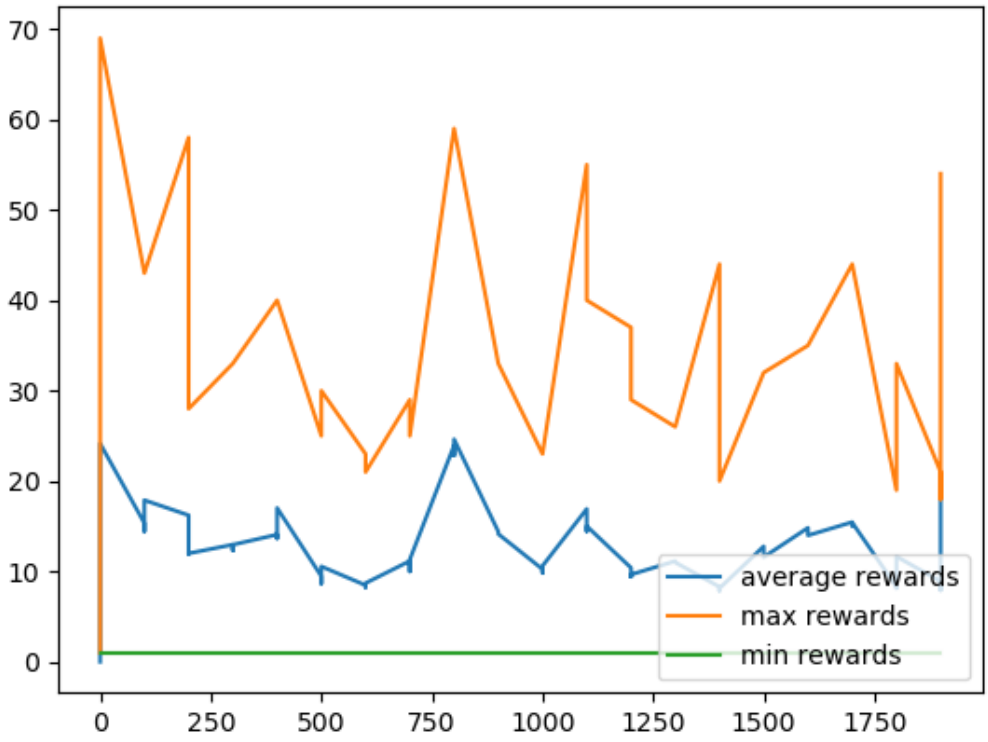
Gráfica 6 - Rewards máximo, promedio y mínimo para discretización de 24 buckets en dimensiones del pole

Con un refinamiento de 10 *buckets* por dimensión los resultados parecen ser un poco menos volátiles, sin embargo, no es un cambio relevante en el rendimiento.



Gráfica 7 - Rewards máximo, promedio y mínimo para discretización de 10 buckets en dimensiones del pole

Con un refinamiento de 2 *buckets* por dimensión



Gráfica 8 - Rewards máximo, promedio y mínimo para discretización de 2 buckets en dimensiones del pole

De acuerdo con lo observado en las gráficas 6, 7 y 8 varios intentos fueron realizados con menor y mayor cantidad de *buckets*, pero no se observa una mejora significativa del desempeño en cuanto al refinamiento, se opta por mantener los valores inicialmente utilizados y se procede a integrar las dimensiones de observación relacionadas al *cart*.

Versión 6 – Posición del cart y velocidad AgentCartAndPoleDimensions

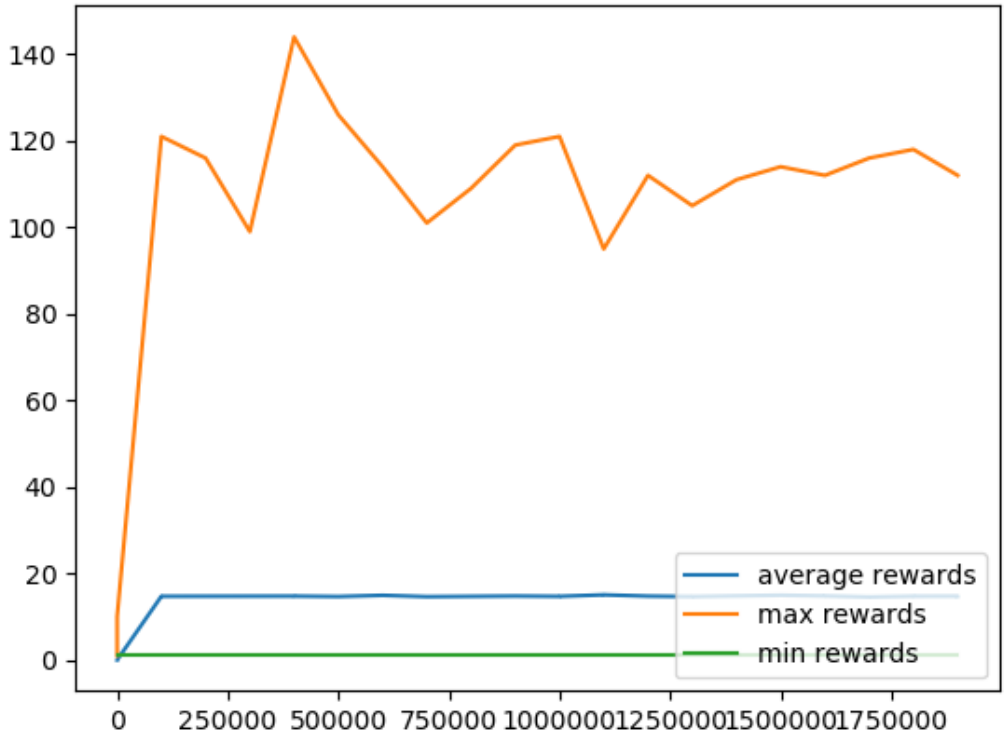
Se integran las dimensiones del *cart* y con apenas 2 mil episodios de entrenamiento no se puede distinguir una mejora en el desempeño comparándolo contra la versión 5.

Gráfica 9 - Reward máximo, promedio y mínimo para 2000 episodios entrenados considerando las dimensiones del cart, discretización de 12 buckets por dimensión

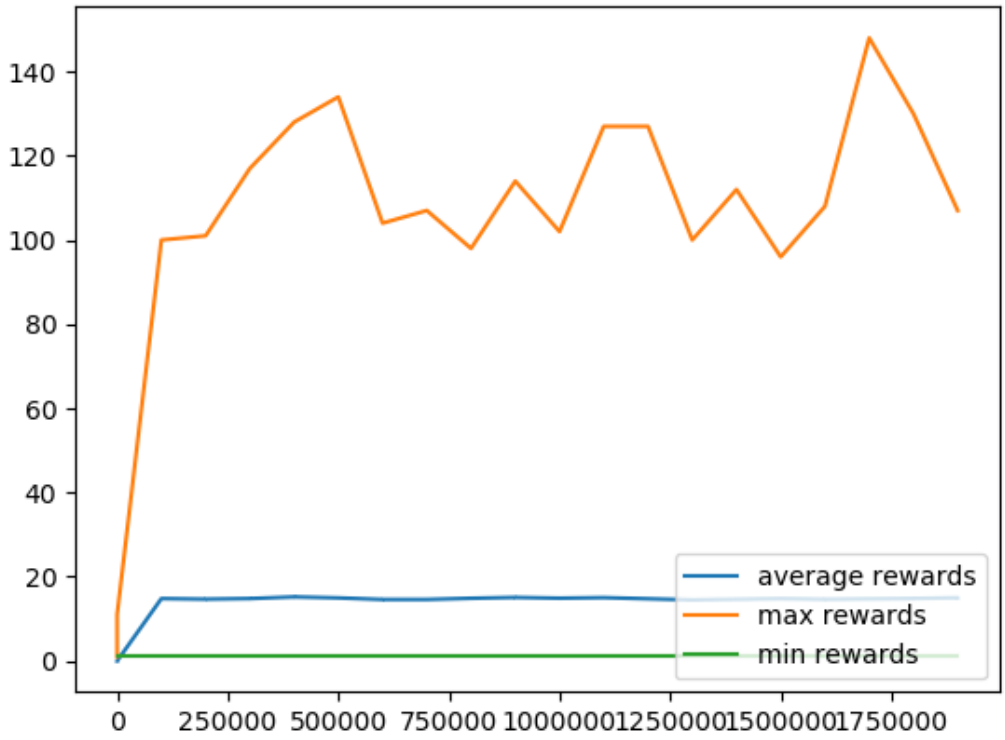
Se intenta con un refinamiento más grueso, con 6 *buckets* por dimensión del *cart*. En esta gráfica se observa un intento de continuar aprendiendo en un desempeño cuya grafica del *reward* parece tender a subir.

Gráfica 10 - Reward máximo, promedio y mínimo para 2000 episodios entrenados considerando las dimensiones del cart, discretización de 6 buckets por dimensión

Para verificar el comportamiento del desempeño se correrán 2 millones de episodios para comparar la gráfica correspondiente a la versión 5.



Gráfica 11 - Reward máximo, promedio y mínimo para 2 millones de episodios entrenados considerando las dimensiones del cart, discretización de 6 buckets por dimensión



Gráfica 12 - Reward máximo, promedio y mínimo para 2 millones de episodios entrenados considerando las dimensiones del cart, discretización de 10 buckets por dimensión

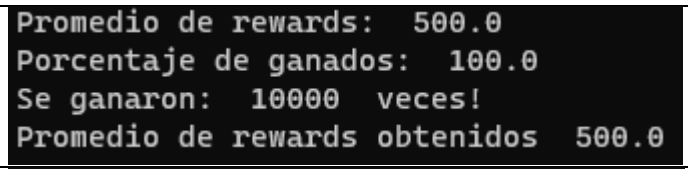
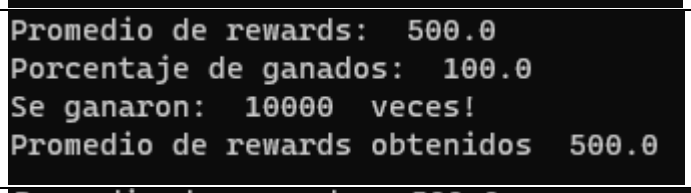
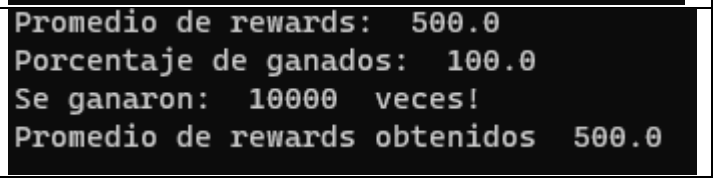
De la observación de las gráficas 9, 10, 11 y 12, surge que la integración de más dimensiones de observación introduce ruido en el modelo. Entendemos que, dado que no concluye ningún episodio de entrenamiento por ser truncado por el ambiente al alcanzar el máximo reward establecido no es, ni siquiera, necesario poner a prueba simulaciones de la versión 6 para compararla con la versión 5. Por lo tanto, se decide analizar los resultados de la versión 5.

Resultados de la versión 5

A lo largo de la solución de esta tarea realizamos varios entrenamientos que arrojaron un episodio siempre inferior al 200 que ya conseguía truncar el juego, es decir, ganar. Consideramos interesante ver como escala esa Qtable en particular con 150 episodios de juego. Poniendo a prueba los mejores entrenamientos obtenidos

101-qtable.npy	Promedio de rewards: 500.0 Porcentaje de ganados: 100.0 Se ganaron: 150 veces! Promedio de rewards obtenidos 500.0
128-qtable.npy	Promedio de rewards: 500.0 Porcentaje de ganados: 100.0 Se ganaron: 150 veces! Promedio de rewards obtenidos 500.0
137-qtable.npy	Promedio de rewards: 500.0 Porcentaje de ganados: 100.0 Se ganaron: 150 veces! Promedio de rewards obtenidos 500.0
154-qtable.npy	Promedio de rewards: 496.88666666666666 Porcentaje de ganados: 93.33333333333333 Se ganaron: 140 veces! Promedio de rewards obtenidos 496.88666666666666
157-qtable.npy	Promedio de rewards: 457.9266666666667 Porcentaje de ganados: 78.0 Se ganaron: 117 veces! Promedio de rewards obtenidos 457.9266666666667
113-qtable.py	Promedio de rewards: 500.0 Porcentaje de ganados: 100.0 Se ganaron: 150 veces! Promedio de rewards obtenidos 500.0

Para verificar el desempeño en un escenario de mayor rigurosidad (a pesar de no representar una muestra significativa del espacio de observaciones posible), simulamos 10000 episodios con las *Qtables* que mejor se desempeñaron:

101-qtable.npy	
113-qtable.npy	
9999-qtable.npy	

Simulación absurda de 1000000 episodios jugados:[2]

101-qtable.npy	
113-qtable.npy	
9999-qtable.npy	

Observando el aprendizaje

Basado en el trabajo *Q-Learning Analysis - Reinforcement Learning* (sentdex, 2019), decidimos generar un video con las graficas de las acciones asociadas a las *Qtable* generadas durante el aprendizaje de la versión 5. Por este motivo se incluye un archivo Video.py que genera a partir de las Qtable guardadas el archivo de video. Tanto el video (Aprendizaje 10000 episodios entrenamiento V5.avi) como el archivo que lo generase encuentran disponibles en la carpeta *Resources.* Lamentablemente la solucion converge en la primer *Qtable* generada por lo que no es posible apreciar una evolución del aprendizaje.

Opciones de mejora

- Sería interesante considerar el paso del tiempo en el cartpole
- Entendemos que los entrenamientos podrían haber sido de una cantidad mayor de episodios
- El seteo de los parámetros podría haber sido investigado en profundidad
- El código en líneas generales puede ser mejorado
- Se presentan una cantidad de Agentes que comparten mucha codificación
- No fue explorada la inclusión de las dimensiones de observación asociadas al *cart* de manera individual. Surge una opción de mejora a partir de la inclusión de cada una de ellas individualmente analizando el desempeño obtenido.

[2] Al momento de entregar este trabajo se continúa aguardando por la ejecución del millón de simulaciones jugadas utilizando las 3 *Qtable* mencionadas. Hasta el presente momento la ejecución lleva 48 horas y aún no ha concluído.

Lecciones Aprendidas

Claramente en el ejercicio de Cartpole se disponía de información que no era relevante para la solución. Alternativamente, no supimos como trabajar con ella y no tuvimos tiempo de investigar una solución que considere la posición y velocidad del cart.

Entendemos que un problema relativamente sencillo para un ser humano promedio como equilibrar un palo es algo desafiante de programar abstraídamente del ambiente sin caer en una solución que se acople demasiado en él.

Un problema de pocas dimensiones de observación puede escalar muchísimo en tiempo y en memoria cuando existen representaciones de continuidad asociadas a alguna de ellas. Esto requiere de un análisis que no siempre es trivial y se basa en la experiencia para saber como discretizar correctamente sin precipitarse en una omisión de información o en una discretización demasiado gruesa que no permita tomar decisiones adecuadas

2048

Abordaje del problema

Este problema decidimos abordarlo en primera instancia mediante la técnica MiniMax, para asegurarnos garantizar el mínimo resultado posible y porque pensamos que iba a dar los mejores resultados, y posteriormente implementamos ExpectiMax, ya que contábamos con la implementación en detalle de la técnica de juego del tablero.

Para poder abordar el problema con un enfoque MiniMax, debimos analizar el problema como un problema de competencia entre 2 contrincantes, uno somos nosotros, que deseamos maximizar nuestro puntaje, el otro es el juego mismo, que buscará minimizar nuestro puntaje, hacer la jugada que menos nos favorezca, o sea, colocar un dos o un cuatro en el lugar que más nos incomode.

Por lo anteriormente dicho, nosotros seremos Max, mientras que el juego será Min.

Para poder desarrollar el algoritmo, debemos además verificar un par de cosas extra, entre ellas:

a) Saber cuándo finalizar la búsqueda.
b) Saber cómo evaluar si un estado es mejor que el otro.

Para responder la primera pregunta debemos fijarnos en 3 situaciones.

La primera es si llegamos a la profundidad deseada, a medida que buscamos en profundidad los posibles estados se van ramificando. En el caso del jugador tendremos una rama hijo por cada movimiento posible, lo que nos da como mucho 4, no está aquí el problema, el problema está en las posibilidades del juego, ya que para cada lugar disponible tiene la opción de colocar un 2 o un 4, lo que ramifica ampliamente el problema. Para evitar tanto cómputo utilizaremos además la técnica Alpha Beta Prunning, la cual explicaremos luego.

Con respecto a saber evaluar cada estado, lo haremos con la función heurística, la misma fue evolucionando con el problema para buscar la óptima, dejaremos un log de las mejores encontradas y un resumen de los resultados de estas. El algoritmo irá buscando en profundidad las posibles opciones y se quedará con la que la función heurística maximice la utilidad.

Posteriormente, al implementar el agente ExpectiMax debimos tener en cuenta la distribución de las posibilidades por parte del tablero de colocar un 2 o un 4 en cada uno de los lugares libres.

Encontramos dificultades con la duración de la ejecución de este, por lo que tuvimos que crear una especie de poda que se explicará más adelante.

Una vez logrados ambas interpretaciones, las comparamos y nos dedicamos a mejorar la función heurística. Adelante en el documento le dedicamos una sección especial al análisis de ésta.

A continuación, explicaremos cada una de las interpretaciones y cómo fueron llevadas a cabo.

MiniMax

El funcionamiento de la técnica minimax es sencillo de explicar, se basa en que existen dos contrincantes, uno intenta maximizar las utilidades, mientras que el otro busca minimizarlas, lo que lleva a que tengamos que utilizar dos funciones, una para cada uno de ellos.

De estas funciones se derivará el movimiento que realizará cada uno de ellos.

En pseudocódigo, la función maxi y mini se desarrolla de la siguiente manera:

```
def function maxi(board):

    if algorithmHasEnded(board):
        return (None, heuristic_utility(board)) # Si el algoritmo finalizó devolvemos la utility que posee el board en el momento.

    (max_board, max_utility) = (None, -INF) # Asignamos un board nulo y una utility de menos infinito

    for child in board.children(): # Para cada uno de los posibles movimientos del jugador se abrirá una rama de estudio
        (_, utility) = mini(children) # Le damos el turno al jugador mini para que use la jugada que más nos perjudique y se siga estudiando en profundidad
        if utility > max_utility: # si esa utilidad encontrada es mejor que la que teníamos, la actualizamos y seguimos.
            (max_board, max_utility) = (children, utility)
    return (max_board, max_utility) # retornamos el board que beneficia al jugador

def function mini(board):
    (min_board, min_utility) = (None, +INF)

    if algorithmHasEnded(board):
      return (None, heuristic_utility(board)) # Si el algoritmo finalizó devolvemos la utility que posee el board en el momento.

    for child in board.children(): # Para cada uno de los posibles lugares donde se puede colocar un 2 o un 4 camos a abrir una rama y seguir estudiando en profundidad
      (_, utility) = maxi(board) # Le damos el turno al jugador maxi para que use la jugada que más le beneficie y se siga estudiando en profundidad
      if utility < min_utility:
        (min_board, min_utility) = (child, utility)

    return (min_board, min_utility) # retornamos el board que perjudica al jugador
```

Como podemos ver, son dos funciones que se llaman entre ellas de manera recursiva hasta que llegue a un fin, si no especificamos el nivel de profundidad deseado, continuarán hasta terminar, abriendo incontables cantidades de ramas, por eso se utiliza un parámetro para establecer la profundidad, dicho parámetro se irá reduciendo a medida que nos penetramos en el estudio, frenando el algoritmo al llegar a la profundidad deseada.

Para mejorar el poder de cómputo, utilizaremos además Alpha Beta Prunning, es una poda utilizada en este tipo de algoritmos para no seguir estudiando casos que no modificarán nuestro resultado, permitiéndonos mejorar la profundidad de estudio del algoritmo. Se llama de esta forma porque agrega dos parámetros extra al algoritmo, Alpha y Beta.

Alpha es el mejor valor que la función maxi puede garantizar.

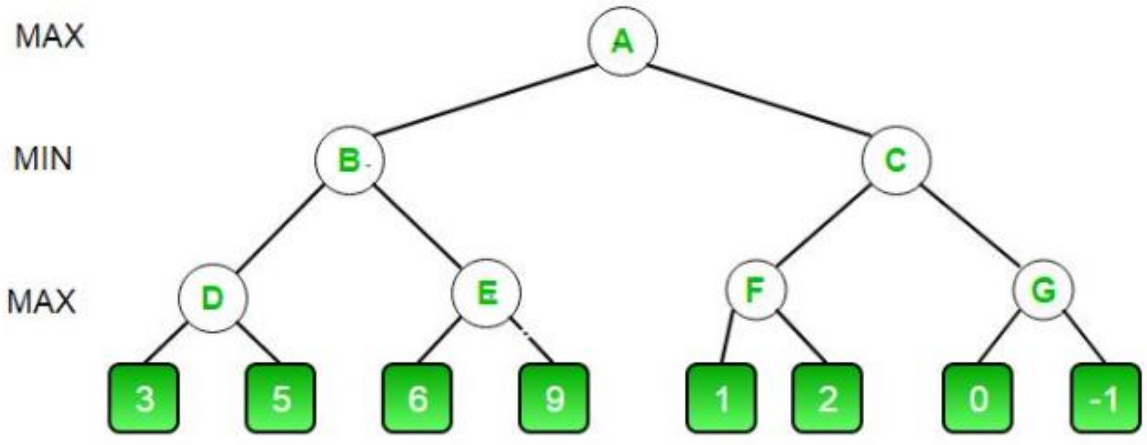
Beta es el mejor valor que la función mini puede garantizar.

El pseudocódigo anterior nos queda de la siguiente forma:

```
def function maxi(board, alpha, beta):

    if algorithmHasEnded(board):
        return (None, heuristic_utility(board))

    (max_board, max_utility) = (None, -INF)

    for child in board.children():
        (_, utility) = mini(children, alpha, beta)
        if utility > max_utility:
            (max_board, max_utility) = (children, utility)
        if max_utility >= beta:
            break
        if max_utility > alpha:
            alpha = max_score
    return (max_board, max_utility)

def function mini(board, alpha, beta):
    (min_board, min_utility) = (None, +INF)

    if algorithmHasEnded(board):
      return (None, heuristic_utility(board))

    for child in board.children():
      (_, utility) = maxi(board)
      if utility < min_utility:
        (min_board, min_utility) = (child, utility)
      if min_utility <= alpha:
        break
      if min_utility < beta:
        beta = min_utility

    return (min_board, min_utility)
```

El algoritmo se llama por primera vez con Alpha -INF y Beta + INF.

En la siguiente figura se muestra el ejemplo que vamos a desarrollar:



En un principio se llama a A con Alpha y Beta con sus valores por defecto, la función maxi seleccionará al mayor entre B y C, comenzaremos con B.

A B se le aplica la función mini, que seleccionará el mínimo entre D y E, comenzamos con D.

En D utilizamos la función maxi, y seleccionaremos el máximo entre 3 y 5, comenzará con 3 y se quedará con el máximo entre 3 y -INF, que es 3, como Alpha era -INF, actualizará la misma a 3, para analizar la rama derecha implementará Alpha Beta Prunning.

D se pregunta si Alpha (3) es mayor a Beta (+INF), lo cual no es verdadero, por lo que sigue analizando la rama derecha.

D recibe el valor 5, por lo que actualiza Alpha a 5 y retorna 5 a B.

B estaba utilizando la función mini, por lo que al recibir 5 actualiza a Beta, dejándolo con el mínimo entre 5 y +INF. mini ahora garantiza un valor de 5 o menos. Va a analizar ahora a E para averiguar si da un valor menor a 5.

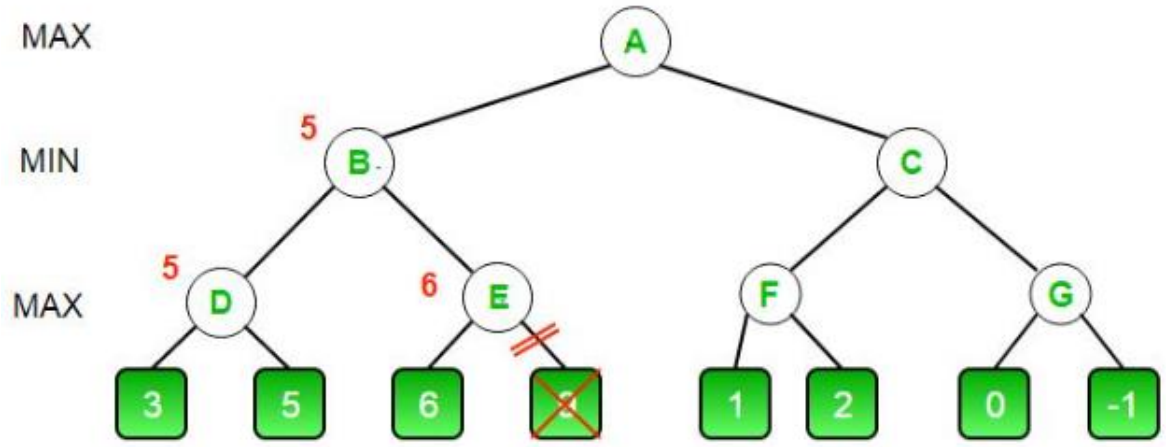
E utiliza la función maxi, y hereda los valores de Alpha (-INF) y Beta(5), E seleccionará el mayor resultado posible entre sus hijos.

E se fija primero en su hijo izquierdo que es 6, y actualiza a Alpha desde -INF a dicho valor. Ahora se fija si le vale la pena buscar en la rama izquierda.

Como Alpha (6) es mayor que Beta (5), E no precisa fijarse del lado derecho, ya que la función mini garantizó un valor de 5 o menos, E al encontrar un 6 sabe que mini de B no va a seguir por dicha rama, no siendo necesario seguir estudiando.

B recibe un 6 de D y un 5 de E, por lo que envía un 5 hacia A.

Por lo que la búsqueda nunca llegó al 9, no siendo computada dicha rama.



ExpectiMax

Como en este ejercicio conocemos el comportamiento del contrincante, y sabemos con precisión cuál es la distribución de sus jugadas, es que podemos implementar el agente ExpectiMax.

Sabemos que el juego, en un lugar vacío al azar, nos pondrá un 2 o un 4 con una distribución del 90 y 10 por ciento respectivamente, por lo que también implementamos dicho agente, pudiendo de esta forma compararlo con MiniMax y seleccionar el que nos es más conveniente.

La diferencia principal de este agente con el anterior es que no podemos realizar poda en el mismo, lo que conlleva a que los tiempos se incrementen notoriamente. Como consecuencia de lo anterior, los tiempos se vieron aumentados un 500%, y para poder trabajar con ellos decidimos hacer una especie de poda parcial. Si el algoritmo llegaba a una profundidad en la que sólo le falten 2 niveles, y además, si en ese momento tenemos más de 6 lugares libres en los que se puedan poner un número, entonces cortaremos el análisis y retornaremos el valor actual del tablero.

```
if len(emptyCells) >= 6 and d <= 2:
    return (None, self.heuristic_utility(board))
```

De esta forma logramos reducir a 1/5 los tiempos.

Hiperparámetros

En el proyecto tenemos dos hiperparámetros base para comenzar a analizar, la profundidad y la función heurística.

La profundidad

Una vez conseguido un algoritmo que funcionaba (habíamos implementado MiniMax con una profundidad de 5), decidimos con la misma función heurística cambiar la profundidad y registrar los valores obtenidos en cuanto a tiempo y cantidad de ganadas.

Obtuvimos la siguiente tabla:

Profundidad	Ganadas de 10	Tiempo Prom.	Tiempo Min.	Tiempo Max.
3	4	0:24	0:16	0:29
4	5	1:41	1:15	2:22
5	8	7:22	5:46	11:08

Como podemos ver, a medida que aumentamos la profundidad, el tiempo de procesamiento aumenta de forma exponencial y disminuye la efectividad del algoritmo.

Nos quedaremos con una profundidad de 5 que nos brinda una buena efectividad a costo de un tiempo de procesamiento acorde.

La función heurística

La función heurística es la que nos establece qué estado es mejor que otro. En un principio comenzamos con una función simple, basada en la suma del puntaje de los lugares, dividido entre la cantidad de espacios ocupados, de esta forma beneficiábamos que hubiera un gran puntaje y la mayor cantidad de espacios libres.

Obtuvimos un resultado razonable, probamos agregar el smoothness

, y se nos complicó en cuanto a la implementación del algoritmo, nos fuimos dando cuenta de errores que teníamos en el código. Una vez solucionados pudimos implementarla de forma correcta, y obtuvimos un peor resultado que la anterior.

Buscamos entonces mezclarlas y variando sus parámetros, entonces conseguimos una función mixta, que fue la que nos brindó la mayor efectividad, y fue con la que hicimos las pruebas de profundidad documentadas anteriormente.

Luego de poseer la función fuimos probando distintas funciones con una profundidad de 4, que nos permitía correrlas en un tiempo razonable como para ir probando.

Obtuvimos los siguientes resultados:

Variante	Ganadas de 10
Mixta	7
Favorecer smooth	0
Suma**2 / lugares vacíos	0
Mixta + Valor por posición	1
Sólo favorecer posición	0

Como resultado obtuvimos que tanto el smoothnes como la suma de valores en conjunto nos brindan los mejores parámetros de rendimiento.

En ese momento pensamos que habíamos logrado una función estable y consistente, pero al momento de calcular un valor ganador y otro perdedor, nos llevamos una sorpresa. Si bien mantenía una relación de que un valor ganador era mejor que uno perdedor, todos los valores se mantenían entre 0 y 1, y terminaba siendo 1 siempre que hubiera un sólo número en el tablero, lo que era un error. Mantenía relación con el smoothness, pero no era una buena función.

0	0	0	0		
0	0	0	0		
0	0	0	0		
2048	0	0	0		
				Pre-Smoothness	2048
				Smoothness	4194304
				Not-Zeros	1
				Sum2	4194304
				Snoes	4194304
				Weight	2
				Utility	**1**

0	0	0	0		
0	0	0	0		
0	0	0	0		
1024	0	0	0		
				Pre-Smoothness	1024
				Smoothness	1048576
				Not-Zeros	1
				Sum2	1048576
				Snoes	1048576
				Weight	2
				Utility	**1**

Verificamos de esta forma que podíamos mejorarla si agregábamos el máximo valor a la función. Para favorecer que siempre se intente tener un nuevo máximo sobre otras cosas, utilizamos el máximo valor al cuadrado, y para que el número se mantuviera entre 0 y 1 (para que sea compatible con la función que veníamos implementando), lo dividimos entre el cuadrado de 2048 que es el máximo que se podía alcanzar.

Luego vimos que la suma de todos los números era un valor que debía ser tomado con mayor relevancia, por lo que agregamos el factor de la suma, pero la dividimos entre 4096 (fuimos llegando a dicho valor con múltiples pruebas), de esta forma el factor nos quedó con un número que muy pocas veces superaba el 1.

Caso Ganador:

0	0	0	0		
0	0	0	0		
0	0	0	0		
2048	0	0	0		
				Pre-Smoothness	2048
				Smoothness	4194304
				Not-Zeros	1
				Sum^2	4194304
				Snoes	4194304
				Weight	2
				Mixed	1
				Max	2048
				Max_K	1
				Sum	2048
				Sum_K	0,5
				Utility	**2,5**

Caso Intermedio

0	0	0	0		
0	0	0	0		
0	0	0	0		
1024	0	0	0		
				Pre-Smoothness	1024
				Smoothness	1048576
				Not-Zeros	1
				Sum^2	1048576
				Snoes	1048576
				Weight	2
				Mixed	1
				Max	1024
				Max_K	0,25
				Sum	1024
				Sum_K	0,25
				Utility	**1,5**

Caso Perdedor

4	8	16	4		
2	4	512	1024		
128	512	1024	2		
2	1024	512	4		
				Pre-Smoothness	5006
				Smoothness	25060036
				Not-Zeros	16
				Sum^2	3948940
				Snoes	246808,75
				Weight	2
				Mixed	0,00984869894
				Max	1024
				Max_K	0,25
				Sum	4782
				Sum_K	1,167480469
				Utility	**1,427329168**

Finalmente, probamos dos funciones más que no tuvieron buenos resultados, una era contemplar que se tuviera un orden creciente en algún sentido, y la otra que se valoraran los números según su posición, dejando los de más valor abajo a la derecha e ir subiendo de valor de forma de serpiente (extrajimos dicha función de jugar el juego). Probamos varias veces ambos algoritmos con dichas funciones, pero no lograron resultados positivos notables, ni siquiera agregándolos como factor de la función que venía funcionando.

Tanto para analizar los casos que funcionaron como los que no, tuvimos que probar con valores, verificar nuestras intenciones en el Excel, debuguear el código y renderizarlo para ver cómo se comportaba la máquina. Vimos, por ejemplo, que algunos casos variaban dependiendo de si estábamos probando con ExpectiMax o con MiniMax, muchas veces no iba hacia cierto lugar porque seguramente dentro de las peores posibilidades no se obtenían los resultados. O vimos también que si bien programamos las funciones pensando en lo que realizamos cuando jugamos en persona, al llevarlo a código, y la máquina no salirse del guion (como sí lo hacemos nosotros), la técnica no era favorable.

Al final la función heurística nos quedó toda junta en una sóla función a efecto de ahorrar cómputo, quedó de la siguiente manera:

```
def max_sq_coef_sum(self, board: GameBoard, weight: int):
  count = 0 # cantidad de lugares ocupados
  sum_sq = 0 # la suma de los números al cuadrado
  sum = 0 # la suma de los números
  max = 0 # el número más elevado
  smoothness = 0
  for i in range(4):
    for j in range(4):
      if i < 3:
        smoothness += abs(board.grid[i][j] - board.grid[i + 1][j])
      if j < 3:
        smoothness += abs(board.grid[i][j] - board.grid[i][j + 1])
      sum_sq += board.grid[i][j] ** 2
      sum += board.grid[i][j]
      if board.grid[i][j] ** 2 > max:
        max = board.grid[i][j] ** 2
      if board.grid[i][j] != 0:
        count += 1
  snoes = int(sum_sq/count) # relación entre la suma de los números y los
lugares ocupados
  # favorece que sean números grandes y muchos lugares libres, cuanto más
grande mejor
  smoothness = smoothness ** weight # el smoothnes elevado al cuadrado,
usamos un weight = 2
  # favorece que los números similares permanezcan juntos, cuanto más grande
es peor
  return (snoes / smoothness) + (max / (2048**2)) + (sum / 4096)
  # nos sirvió relacionar en una división el snoes y smoothness
  # se suma además el número más elevado, relacionado con 2048 al cuadrado
  # también se contempla la suma de los números dividido entre 4096
```

Análisis de los Agentes

Durante el transcurso del obligatorio pudimos darnos cuenta de que no teníamos un agente estable, ante las mismas circunstancias, se corrían 10 veces para probar su funcionamiento y podía resultar en que ganaba 8, como podía resultar que ganaba 3, por lo que nos decidimos por realizar varios niveles para ponerlos a prueba.

En un primer nivel, corridas de 50 repeticiones con una profundidad de 4, que son corridas rápidas, y documentar el log en un archivo. Con este nivel probamos las funciones heurísticas.

Una vez tenida la función heurística, pasamos a un segundo nivel, corriendo los dos agentes con una profundidad de 4, pero 50 veces. En este caso decidimos si utilizar MiniMax o ExpectiMax.

Finalmente pasamos a un tercer nivel con 20 corridas y una profundidad de 5 para ExpectiMax o 6 para MiniMax, donde verificamos el rendimiento con corridas más prolongadas.

Los resultados son los siguientes:

Agent = MiniMax			
Deepth = 4			
	Win	Time	Moves
OldHeuristic	28%	9.7s	902
OldMaxSumSq	42%	10.6s	919
OldMaxSumSqOrd	0%	7.2s	634

Agent = MiniMax			
Deepth = 6			
	Win	Time	Moves
OldMaxSumSq	60%	1m 48s	845

Agent = ExpectiMax			
Deepth = 4			
	Win	Time	Moves
OldMaxSumSq	62%	34.1s	957

Agent = ExpectiMax			
Deepth = 5			
	Win	Time	Moves
OldMaxSumSq	80%	3m 56s	993

En la imagen, se puede ver en el cuadrante superior el comportamiento de las heurísticas, la nomenclatura es la siguiente:

OldHeuristic es la primera función heurística a la que llegamos, que relaciona el snoes / smoothness

OldMaxSumSq es la última función que empleamos, la que se explicó en la sección anterior.

OldMaxSumSqOrd es la que implementamos agregando peso al orden de las filas y columnas, esta no tuvo resultado y no se ve en el código.

Del lado derecho vemos como mejora la función al aumentar la profundidad de estudio. El objetivo que tuvimos era mantenerlo por debajo de los 5 minutos, por lo que llegamos a una profundidad de 5 para ExpectiMax y 6 para MiniMax.

Finalmente, podemos ver que, para el ejercicio planteado, ExpectiMax es mejor agente que MiniMax. Esto se debe a que muchas veces por no caer en el peor caso no toma opciones válidas para ir hacia la victoria, y además que al conocer específicamente cómo funciona el algoritmo del juego, podemos utilizarlo a nuestro favor, mejorando de esta forma nuestras ganancias.

El ejercicio lo dejamos pronto para correr con el agente ExpectiMax y una profundidad de 5, que fue la que dio mejores resultados y se mantuvo dentro de un tiempo razonable.

Referencias

El código implementado y el análisis teórico fue desarrollado en base a lo dado en clase y a varios artículos encontrados en internet, los más relevantes son los siguientes:

https://towardsdatascience.com/playing-2048-with-minimax-algorithm-1-d214b136bffb
https://towardsdatascience.com/how-to-represent-the-game-state-of-2048-a1518c9775eb
https://towardsdatascience.com/how-to-control-the-game-board-of-2048-ec2793db3fa9

https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/