Universidad ORT Uruguay

Facultad de Ingenería Ing.Bernard Wand-Polak

Grupo M7B - Inteligencia Artificial

Obligatorio

Andrés Naistat-Hauser - 273470

Nicolás Cababie - 203656

Año 2024

Repositorio OBL

Código

Respecto al código, hemos realizado modificaciones sobre la notebook original. En primer lugar modificamos el ejemplo de episodio que se nos brindó para hacer una función de entrenamiento acorde al modelo. En dicha función agregamos una línea de código para actualizar el Q en cada ejecución y otra para registrar valores en Wandb. Luego agregamos el código necesario para poder realizar una ejecución sola en Wandb y para poder hacer un sweep. Por último, agregamos el código necesario para guardar y cargar el modelo en formato .pkl y también para poder grabar un vídeo y subirlo a Wandb.

Taxi

Uno de los primeros desafíos que tuvimos que enfrentar a la hora de implementar este modelo fue entender cómo funciona el ambiente. Para esto, empezamos a imprimir un renderizado de que es lo que estaba sucediendo cada vez que se procesaba una acción.

Luego empezamos a realizar pruebas cambiando los hiper parámetros de alpha, gamma y epsilon junto con la cantidad de episodios. Tomamos como premisa que un valor alto de alpha significa que el agente da más peso a la información reciente mientras que un valor bajo significa que el agente considera más las experiencias pasadas. Luego con el hiper parámetro gamma, entendemos que un valor cercano a 1 hace que el agente valore más las recompensas a largo plazo y que un valor más bajo hace que se enfoque más en las recompensas inmediatas. Por último entendemos que un valor más alto de epsilon fomenta la exploración mientras que un valor más bajo la explotación.

Una vez encontrado un conjunto de hiper parámetros que consideramos buenos, empezamos a registrar nuestras ejecuciones en Wandb. Algunas de las métricas que guardamos fue la recompensa total del episodio para ver cómo mejora el rendimiento del agente con el tiempo y el valor promedio de Q también para ver como evoluciona.

Empezamos a variar la cantidad de episodios en cada entrenamiento, primero empezamos con cantidades chicas para entrenar el modelo de forma rápida y luego empezamos a subir la cantidad con el objetivo de que el agente pueda explorar diferentes estados y acciones, y explotar las mejores acciones aprendidas.

Para encontrar la mejor combinación de hiper parámetros decidimos introducir sweeps en nuestro entrenamiento. En un momento empezamos a tener problemas al ejecutar los entrenamientos, cuando transcurría más de una hora del entrenamiento VS Code empezó a colapsar. Como solución decidimos ajustar mejor nuestros hiper parámetros y definimos los siguientes posibles valores para cada hiper parámetro:

- alpha: 0.1, 0.3, 0.5
- gamma: 0.8, 0.9, 1
- epsilon: 0.1, 0.5
- épocas: 5000, 10000

Una vez completado el sweep, procedimos a analizar los resultados generados para seleccionar el mejor conjunto de hiper parámetros para nuestro agente. Tomamos como

referencia aquel conjunto que maximice la recompensa obtenida. Finalmente los hiper parámetros elegidos fueron:

- alpha: 0.5
- gamma: 0.9
- epsilon: 0.1
- épocas: 5000

Una vez finalizada la etapa de investigaciones y pruebas, decidimos computar el modelo final. Para llevar a cabo esto utilizamos la librería “pickle” para crear nuestros modelos en formato .pkl y poder manejarlos y utilizarlos fácilmente.

Finalmente creamos un video de 10 ejecuciones del agente para visualizar su comportamiento y lo cargamos en Wandb. Utilizamos la librería de “imageio”.

El reporte con análisis, vídeo y las gráficas elegidas se encuentran en el siguiente enlace:
Taxi Report

Péndulo

El segundo ejercicio que tuvimos que resolver fue el del péndulo, al principio nos encontramos con problemas parecidos al taxi: entender bien el contexto, como funciona la recompensa y cuál era el objetivo. Una vez pasado eso fuimos adaptando el código para que funcione y jugamos un poco con los hiperparametros pero no veíamos progreso alguno, el péndulo daba vueltas simplemente.

Procedimos a investigar y nos encontramos con que una buena forma de entrenar el modelo era el ir bajando el epsilon a medida que avanza el entrenamiento, probamos con esto y vimos una gran mejora en los resultados. Esto se implementó de forma que se pasan 3 valores de epsilon, un mínimo, el “decay”, y epsilon inicial. El mínimo indica hasta dónde puede bajar el valor de epsilon a lo largo de nuestra ejecución, el valor inicial es con el que se arranca la ejecución y por último el decay es un valor por el cual se multiplica al valor actual de esa corrida para bajar un porcentaje del valor.

Una vez que pasamos esta etapa de ir probando un poco, procedimos a registrar las corridas en Wandb al igual que con el taxi.

Los primero registros en Wandb fueron corridas solas, luego cuando aprendimos a utilizar los sweeps armamos uno con los siguientes valores:

- alpha: 0.1, 0.2, 0.3
- gamma: 0.8, 0.9 1.0
- epsilon: 0.1, 0.3, 0.5
- epochs: 500, 1000
- epsilon_min: 0.01
- epsilon_decay: 0.995

El sweep estaba configurado en modo grid por lo tanto se probaron todas las combinaciones posibles de estos valores. Tras analizar los resultados ordenamos la tabla

en orden según el “total_reward” y en base a este decidimos tomar los valores de la segunda corrida ya que la primera tenía buenos resultados pero con muy poca diferencia salvo el tiempo de ejecución, este era de 43 min en el primero y de 4 min en el segundo. Por la poca diferencia que había en el “total_reward” decidimos ir por los valores del segundo ya que el costo de tiempo no vale esa diferencia en el resultado. Luego de esto creamos el modelo en formato .pkl con los hiperparametros seleccionados.

Los valores seleccionados fueron:

- alpha: 0.3
- gamma: 0.9
- epsilon: 0.3
- epochs: 1000
- epsilon_min: 0.01
- epsilon_decay: 0.995

Como último paso al igual que con el taxi nos tomamos el tiempo de registrar en Wandb un video del péndulo en funcionamiento utilizando el modelo de cargado en el .pkl anteriormente.

Algo que faltó mencionar es que al principio teníamos un problema al ejecutar el péndulo, en torno a los 10 minutos de la ejecución se crasheaba VS Code y teníamos que volver a comenzar. La solución a esto fue achicar el espacio de discretización de “vel_space”, una vez que lo achicamos dejó de ocurrir este error, parecía estar relacionado con el uso de la RAM.

El reporte con análisis, vídeo y las gráficas elegidas se encuentran en el siguiente enlace:
Pendulum Report

En la entrega, dejamos adjunto ambos reportes de Wandb en formato PDF junto con las notebooks utilizadas. Para visualizar mejor los reportes, recomendamos acceder mediante los enlaces.

Coin Game

Antes de inclinarnos sobre uno de los dos agentes vamos a definir cada uno.

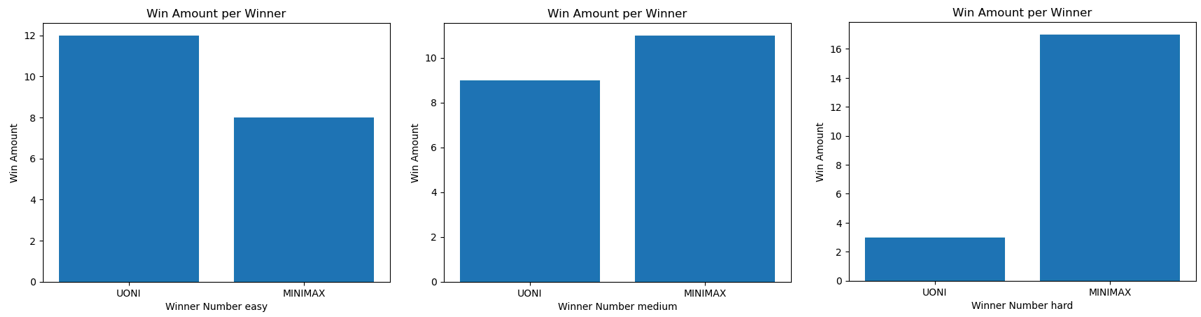
El algoritmo Minimax busca que las ganancias de un jugador sean las pérdidas del otro. En este algoritmo se asume que el oponente jugará de manera óptima, intentando maximizar su ganancia y minimizar la del otro jugador.

Expectimax es una variación de Minimax y es usado principalmente en escenarios donde las acciones del oponente son aleatorias y no deterministas. En este algoritmo se calcula el valor esperado de las acciones del oponente en lugar de asumir que el oponente va a jugar de manera óptima.

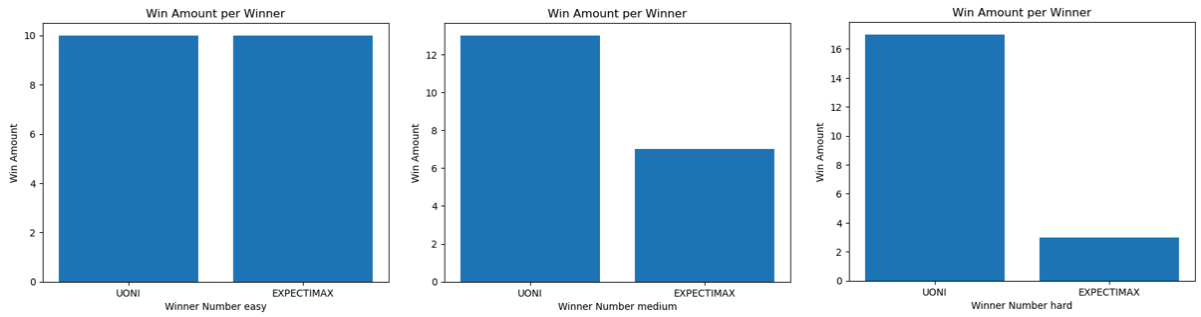
Los motivos que nos llevaron a elegir a Minimax fueron los siguientes: el juego es completamente determinístico y de suma cero donde cada acción tiene un resultado que se puede predecir. Consideramos oportuno que cada jugador intente maximizar su propia ganancia y minimizar la del oponente. Minimax evalúa las jugadas basándose en la peor respuesta posible del oponente a diferencia de Expectimax que toma decisiones basándose en promedios esperados. En términos de costo computacional, la ejecución de Minimax es más rápida que la de Expectimax, esto se debe a que Expectimax calcula el valor esperado en cada nodo del árbol de decisión.

De todos modos, hemos implementado y probado ambos agentes para corroborar nuestras suposiciones y en base a 20 partidas en cada una de las dificultades con cada uno de los agentes llegamos a los siguientes resultados:

UONI vs Minimax:



UONI vs Expectimax:



Frente a escenarios más complicados, Minimax es más robusto y consistente con sus decisiones gracias a que busca minimizar la peor pérdida posible en juegos de este tipo.

Heurísticas consideradas para Minimax:

1. Cantidad de monedas restantes
2. Cuántas filas aún tienen monedas (ya que cuantas más filas haya, más movimientos posibles habrá)
3. Ponderar las monedas (cuánto más avanzado sea el juego, más valor tendrán las monedas)
4. Combinación de la 2 y 3

Nos decidimos por utilizar la **heurística número 4** debido a que consideramos que es una buena forma de evaluar el estado del juego ya que se fija en la disponibilidad de movimientos y la importancia de las monedas según su posición.

También el tener más filas con monedas implica más opciones de movimiento y cuanto más adelante en el juego se esté, más valor tendrá cada acción de seleccionar monedas.

Respecto a las demás heurísticas consideradas, la número 1 es demasiado simple y no aporta suficiente información acerca del estado de juego. Luego las heurísticas 2 y 3 consideramos que por separado no terminan de aportar todo su potencial y decidimos que al juntarlas se podrán obtener mejores estimaciones.

Un valor alto de esta heurística indica un estado favorable, donde hay muchas opciones de movimiento y posiciones estratégicamente importantes. Mientras que un valor bajo indica lo opuesto. Para considerar un valor alto o bajo se compara con los valores de otros estados.

Para evaluar la función en un estado favorable y desfavorable y corroborar que la función en un estado ganador (Eval(win)) sea mayor que la función en un estado perdedor (Eval(loss)) podemos plantear el siguiente ejemplo:

Estado Favorable (win):

- Número de filas con monedas: 3
- Fila 1: 1 moneda → 1 x 1 = 1 (ponderación: nro de fila x cantidad de monedas)
- Fila 3: 2 monedas → 3 x 2 = 6
- Fila 4: 1 moneda → 4 x 1 = 4

Heurística total: 3 + 1 + 6 + 4 = 14

Estado Desfavorable (loss):

- Número de filas con monedas: 2
- Fila 1: 0 monedas
- Fila 2: 1 moneda → 2 x 1 = 2
- Fila 4: 1 moneda → 4 x 1 = 4

Heurística total: 2 + 2 + 4 = 8

En conclusión, la heurística para un estado favorable es mayor que para uno desfavorable:

- Eval(win) = 14
- Eval(loss) = 8

Finalmente podemos afirmar que **Eval(win) > Eval(loss)**

Alpha-Beta Pruning

Decidimos implementar el algoritmo alpha-beta pruning.

Con esta estrategia lo que buscamos es reducir la cantidad de nodos que se evalúan en el árbol de búsqueda, cumpliendo con uno de nuestros objetivos de buscar la mejor jugada posible.

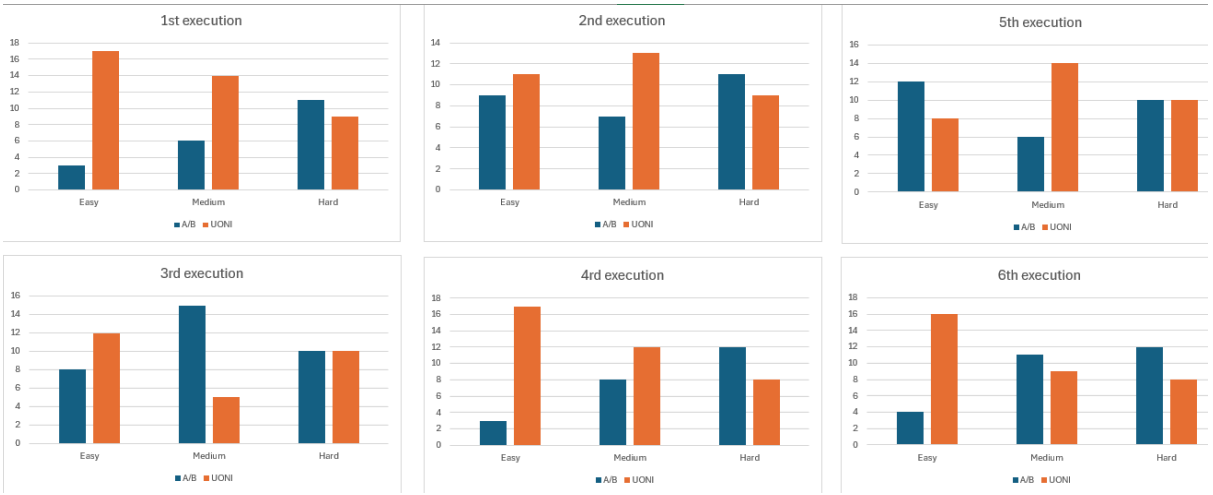
Se tomó como referencia el código creado para el agente Minimax y se le realizaron algunas modificaciones:

- Se agregaron los parámetros ‘alpha’ y ‘beta’ a la función minimax
- La primera vez que se llama a la función se inicializan los valores de alpha en $-\infty$ y beta en ∞.
- Para el jugador que maximiza, se recorre cada posible estado y se actualiza alpha con el valor máximo encontrado.
- Para el jugador que minimiza, se recorre cada posible estado y se actualiza beta con el valor mínimo alcanzado.
- El break ocurre cuando se determina que una rama no puede proporcionar un mejor resultado que lo que ya se haya encontrado.
- La heurística implementada consiste en realizar la ‘nim-sum’. Esta es la operación XOR aplicada a todas las filas. Si el nim-sum es 0, la posición es perdedora para el jugador que está a punto de mover. Si el nim-sum no es 0, entonces hay una estrategia ganadora para el jugador que mueve.

Al momento de implementar la heurística en un principio dejamos implementada la heurística que usamos en el agente minimax. Luego pensamos que deberíamos tener una heurística precisa que nos garantice una jugada óptima y nos decidimos por usar ‘nim-sum’ que se basa en una estrategia matemática que garantiza una jugada óptima. Si el resultado es 0, el estado del tablero es una posición perdedora entonces la función retorna -1, si el resultado es otro, retornamos 1 indicando que el estado del tablero es una posición ganadora.

Realizamos 6 ejecuciones donde en cada ejecución se juega 20 veces en cada una de las tres dificultades. A continuación mostraremos los resultados en gráficos de barras donde se podrá apreciar el número de victorias de cada agente en cada dificultad.

Nomenclatura: **A/B** refiere al agente que usa alpha-beta pruning y **UONI** es el agente oponente.



Con estos resultados, podemos intuir que el agente A/B podría haber encontrado una mejor forma de manejar el juego bajo la dificultad más difícil y que el agente UONI se desempeña mejor en las otras dos dificultades. En juegos de estrategia y matemática es común que los resultados varíen según el entorno de juego.

Referencias bibliográficas

Coin game y alpha-beta pruning:

- Minimax in Python: Learn How to Lose the Game of Nim: https://realpython.com/python-minimax-nim/#implement-a-nim-specific-minimax-algorithm
- Nim (juego): https://es.wikipedia.org/wiki/Nim_(juego)