MAESTRÍA EN INTELIGENCIA ARTIFICIAL

REPORTE DE TRABAJO PRÁCTICO

TP1: MONTE CARLO ES

Autor:
Maximiliano Torti

Docente:
Miguel Augusto Azar

Este trabajo fue realizado en abril de 2025.

Índice general

1 Desarrollo

1.1. Introducción

En el presente trabajo práctico, se desarrolla la implementación del algoritmo Monte Carlo ES en un modelo que, luego de ser entrenado, es capaz de jugar al tres en linea. El objetivo es poder evaluar la implementación del algoritmo, sus beneficios y defectos, y también su rendimiento en un juego simple y reducido.

Para lograrlo, se implementaron en lenguaje Python las clases *Board*, y *TicTacToe* que modelan el tablero y las reglas de juego respectivamente (entorno) ,y la clase abstracta *Player* que da soporte a las operaciones de los posibles jugadores (agentes). Heredando de *Player*, se crearon las clases *UserPlayer*, *BotPlayer* y *MonteCarloEsControl* que permiten jugadores humanos, bots aleatorios y bots guiados por el algoritmo de Monte Carlo ES respectivamente.

Un script principal facilita la ejecución de los casos de uso principales: comenzar un juego entre 2 jugadores o entrenar un agente de Monte Carlo.

1.2. Entorno

El juego comienza luego de definir 2 tipos de jugadores entre las 3 posibilidades, instanciar la clase *TicTacToe* y ejecutar el método *start*. En ese instante se imprime en pantalla un tablero de tres en línea como se muestra en la figura 1.1.

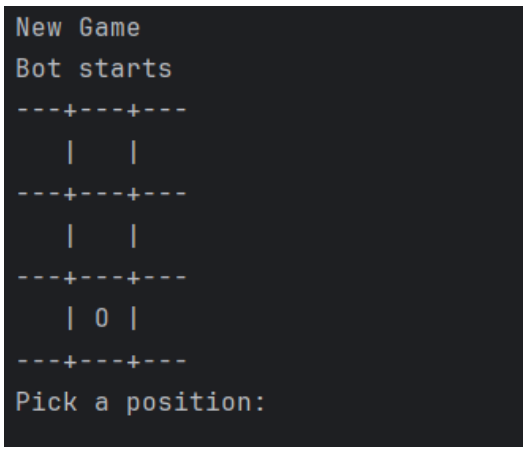

FIGURA 1.1. Tablero de tres en linea.

El comportamiento inicial dependerá de los jugadores seleccionados como jugador 1 y 2 y del azar, que le asignará a cualquiera de los anteriores la jugada inicial. Cada vez que un jugador realice una jugada valida, el tablero se refrescará y la partida continuará hasta que haya un ganador o se produzca un empate (tablero lleno). En este momento se detiene el juego y se informa el resultado, como se muestra en la figura 1.2.

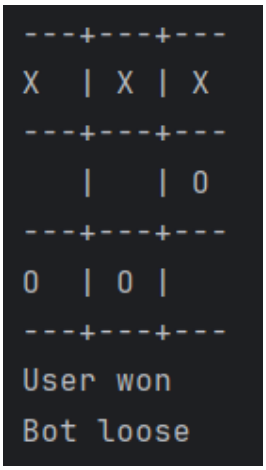

FIGURA 1.2. Juego finalizado.

1.3. Agentes

Cuando un jugador es humano y le llega su turno, este debe introducir por teclado un número del 0 al 8, correspondientes a las posiciones que se muestran en la figura 1.3. Si el jugador ingresa una posición inválida o una posición ya tomada, se deberá volver a introducir la jugada.

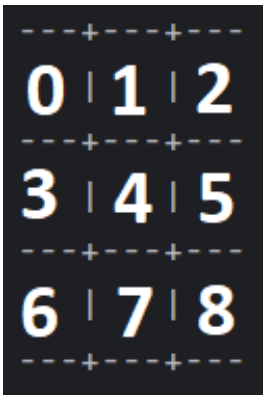

FIGURA 1.3. Posiciones en el tablero.

El bot aleatorio elige, en forma automática y aleatoria, cualquier posición disponible en el tablero. Este no sigue ninguna estrategia y resulta útil únicamente para probar el juego y para realizar el entrenamiento del agente de Monte Carlo.

El agente de Monte Carlo realiza movimientos en forma automática siguiendo una política aprendida. En cada turno observa el estado actual del tablero, y toma la acción que se evaluó como mejor opción para ese estado durante el entrenamiento.

1.4. Entrenamiento

Para entrenar el agente de Monte Carlo, se ejecutaron múltiples juegos consecutivos de tres en línea donde participaban el agente en entrenamiento y bots aleatorios. Dentro del agente de Monte Carlo se implementó el algoritmo de la figura 1.4, con los siguientes detalles:

- $A(s)$ contiene las acciones posibles que son jugar las posiciones del 0 al 8.
- S contiene los estados posibles del tablero. El tablero de tres en línea tiene en total $3^9 = 19{,}683$ estados diferentes, la mayoría de los cuales son ilegales.
- Dado que resulta costoso generar las políticas para todos los estados del tablero y también filtrar solo los estados posibles (alrededor de 5.478), se decidió comenzar con $\pi(s)$ y $Q(s, a)$ vacíos e inicializarlos en forma aleatoria a medida que el agente se encontraba con estados que nunca había visto antes.
- Para el comienzo exploratorio de Monte Carlo, se ejecutaron en el tablero un número aleatorio de jugadas aleatorias alternadas entre jugador 1 y 2 previo a comenzar el episodio. Cabe destacar que se realizó un chequeo del estado del tablero luego de estas inicializaciones para verificar la validez.
- Como recompensa se asignó -0.1 por cada jugada que no concluyera en el final del juego, -10 a una jugada perdedora, -2 a una jugada de empate y 10 a una jugada ganadora.
- Adicionalmente se limito el número de jugadas del agente en 20 y se agrego una penalización de -50 si se supera este valor. De esta forma se prioriza que aprenda a jugar posiciones válidas.
- Se fijo un factor de descuento de 0.9.
- El entrenamiento se repitió durante 500.000 episodios, demorando aproximadamente una hora.
- Los valores anteriores se obtuvieron en forma empírica, a partir de varios entrenamientos y analizando los resultados obtenidos.

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:
$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
$Q(s, a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
$Returns(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):
Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0
Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
$\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$

FIGURA 1.4. Algoritmo de Mote Carlo ES.

1.5. Resultados

Luego de entrenar el agente Monte Carlo ES durante 500.000 episodios se obtuvo un modelo capaz de identificar las estrategias principales del tres en línea. Por ejemplo, si el agente comienza el juego, este toma siempre el centro del tablero como se observa en la figura 1.5. También, si el agente tiene posibilidad de ganar,

toma la posición ganadora; o por el contrario, si el contrincante tiene posibilidad de ganar, el agente intenta bloquear esa posibilidad, como se observa en las figuras 1.6 y 1.7.

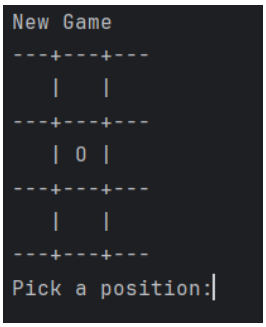

FIGURA 1.5. Agente tomando el centro.

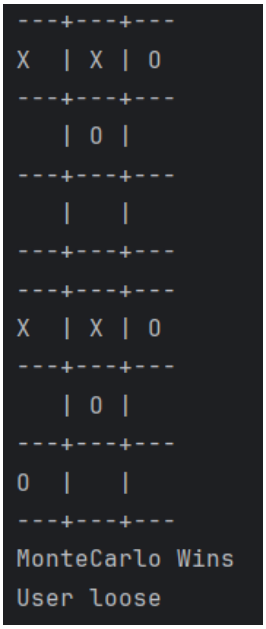

FIGURA 1.6. Agente tomando la posición ganadora.

Lo sorprendente es que el agente también aprende estrategias avanzadas, Por ejemplo, si teniendo tomado el centro el usuario elije posición 1, 3, 5 o 7, la victoria esta asegurada con el patrón de la figura 1.8. A la vez, si el jugador intenta ejecutar la misma estrategia sobre el agente, este la evita eligiendo las esquinas como su primer movimiento.

Como contrapartida el agente no es perfecto, cuando se intenta ejecutar sobre el mismo otra estrategia ganadora mostrada en la figura 1.9, este falla eligiendo repetidamente una posición ya tomada. Esto se debe a que en el entrenamiento no se exploró el espacio de estados y acciones lo suficiente como para encontrar una mejor política.

1.6. Futura mejoras

Como futura mejora se plantea crear una clase *Trainer* que agregue visualizaciones de métricas útiles durante el entrenamiento como el valor Q medio de todos

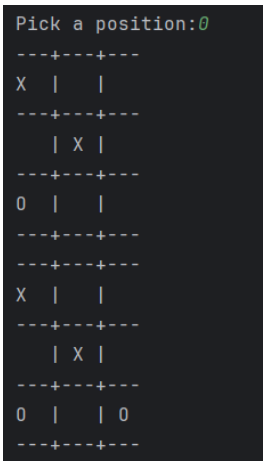

FIGURA 1.7. Agente bloqueando la posición ganadora del humano.

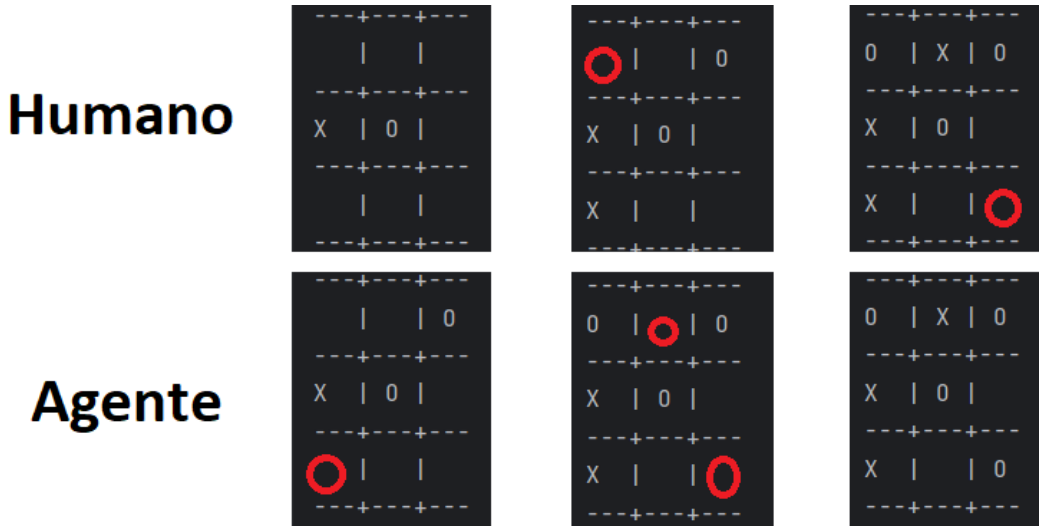

FIGURA 1.8. Agente ejecutando estrategia que asegura la victoria.
En rojo posiciones ganadoras.

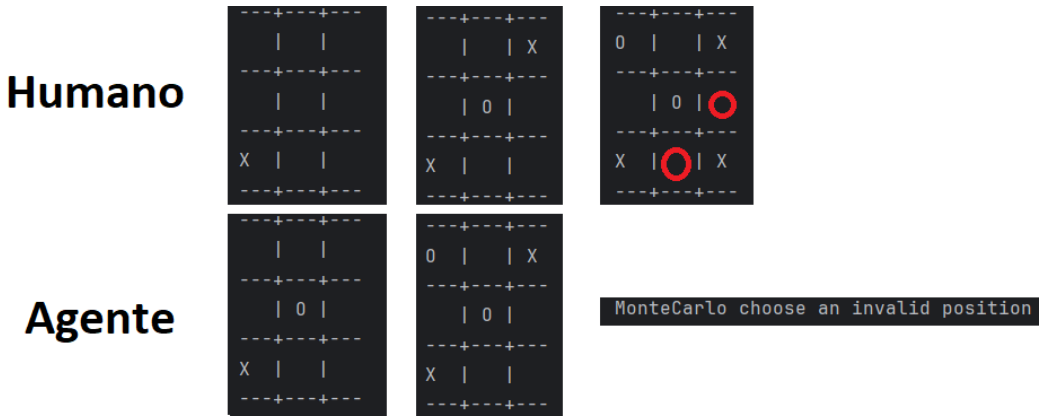

FIGURA 1.9. Agente fallando en tomar posiciones válidas.

los estados cada cierto número de episodios, o la cantidad de victorias, pérdidas y empates logrados en los últimos 100 episodios.

1.7. Conclusiones

El algoritmo mostró resultados muy buenos en el desafío planteado, no solamente aprendiendo los movimientos básicos, sino también aprendiendo estrategias avanzadas lo cual es inesperado dada la sencillez del algoritmo. Como puntos débiles, aún en un juego simple como el tres en línea, se deben mantener estructuras para todos los estados posibles del tablero (que en este caso es del orden de los miles) lo que requiere de un entrenamiento de muchos episodios (en el orden del millón), haciéndolo lento. Otro punto débil del algoritmo es que debido que el algoritmo no puede "extrapolar" conocimiento, es decir aprovechar lo aprendido para un estado para resolver estados similares.

A Script principal

El script principal del proyecto contiene un procesador de líneas de comando que facilita la ejecución de los diferentes casos de uso. Los usos básicos del procesador se detallan a continuación:

- Mostrar la ayuda.

```
$ python src/main.py --help
```

- Jugar un humano contra el bot aleatorio.

```
$ python src/main.py play --games=<number_of_games> --bot
```

- Jugar un humano contra el agente de Monte Carlo.

```
$ python src/main.py play --games=<number_of_games> \
    --monte_carlo=<model_file>
```

- Entrenar un nuevo agente de Monte Carlo.

```
$ python src/main.py train --episodes=<number_of_episodes>
```

```
> python src/main.py --help
This script allows you to play or train a Tic Tac Toe game using different players.
You can play against a bot, a Monte Carlo model, or another human player.
It also allows you to train a Monte Carlo model using reinforcement learning.

Usage:
    main.py play --games=<number_of_games> [--bot] [--monte_carlo=<model_file>] [--human]
    main.py train --episodes=<number_of_episodes>

Options:
    -h --help                       Show this screen.
    --bot                           Play against a bot.
    --episodes=<number_of_episodes> Number of episodes to train the Monte Carlo model.
    --games=<number_of_games>       Number of games to play.
    --human                         Play against another human player.
    --monte_carlo=<model_file>      Play against a trained Monte Carlo model.
```

FIGURA A.1. Uso del script principal por línea de comando.

B Cobertura y calidad de código

Dada la complejidad del entorno, los agentes y el algoritmo de Monte Carlo ES, se implementaron test unitarios de las diferentes clases a fin de verificar el comportamiento de los diferentes métodos. Estos test se pueden ejecutar mediante el comando:

```
$ pytest
```

También se puede analizar la cobertura de código lograda con los test unitarios desarrollados mediante el siguiente comando:

```
$ coverage run -m pytest
$ coverage report
```

Name	Stmts	Miss	Cover
src__init__.py	0	0	100%
src\game__init__.py	0	0	100%
src\game\board.py	38	0	100%
src\game\monte_carlo.py	85	0	100%
src\game\players.py	47	0	100%
src\game\tic_tac_toe.py	55	0	100%
tests__init__.py	0	0	100%
tests\game__init__.py	0	0	100%
tests\game\test_board.py	64	0	100%
tests\game\test_monte_carlo.py	105	0	100%
tests\game\test_players.py	59	0	100%
tests\game\test_tic_tac_toe.py	132	0	100%
TOTAL	585	0	100%

FIGURA B.1. Reporte de cobertura.

Para mantener una buena calidad de código se ejecutaron análisis estáticos mediante la herramienta SonarLint y se resolvieron las malas prácticas y los problemas de seguridad encontrados.

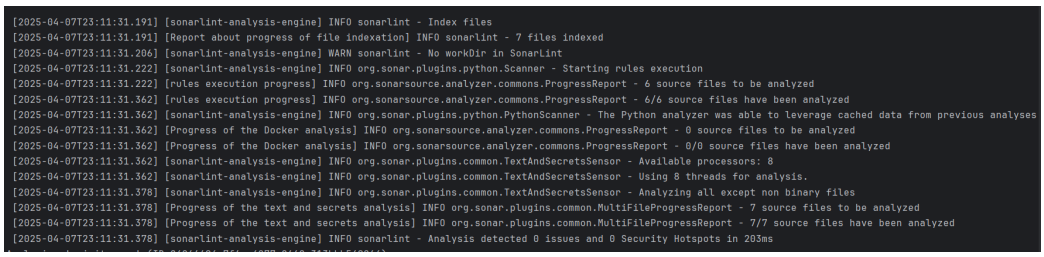

FIGURA B.2. Análisis estático de código.

C Repositorio

El repositorio con el código fuente, los test unitarios y archivos adicionales de utilidad se encuentra disponible en el siguiente link:

https://github.com/maxit1992/MIA_RL1/tree/main/tp1