Dominando Otelo: Estrategias de búsqueda adversaria en juegos de turno

Miguel Galán Lerate
Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
miggaller@alum.us.es

Darío Rodríguez Saster
Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
darrodsas@alum.us.es

***Resumen*—El objetivo principal de este proyecto era crear una herramienta basada en inteligencia artificial que sea capaz de jugar al clásico juego de otelo, también conocido como reversi, othello o yang. Para solucionar este problema hemos conseguido, mediante la combinación de algoritmos de búsqueda y redes neuronales, construir jugadores capaces de tomar decisiones estratégicas que actúen a su favor en la partida. Se ha desarrollado un agente que utiliza el algoritmo minimax con poda alfa-beta a la hora de explorar el espacio de decisiones. Junto al agente se ha implementado una red neuronal que ayudará a proporcionar una heurística de evaluación de las posiciones intermedias, ahorrando el tener que hacerlas a mano.**

I. Introducción

Otelo, también conocido como Reversi, es un juego de estrategia originado en el Londres del siglo XIX. Se caracteriza por ser un juego **determinista** (no interviene el azar y los resultados de las acciones son siempre predecibles), de **suma cero** (las ganancias de un jugador implican necesariamente las pérdidas del otro), y de **información perfecta** (todos los jugadores tienen acceso completo al estado del tablero en todo momento). Estas propiedades lo convierten en un entorno ideal para el desarrollo y evaluación de agentes inteligentes en el ámbito de la inteligencia artificial (IA).

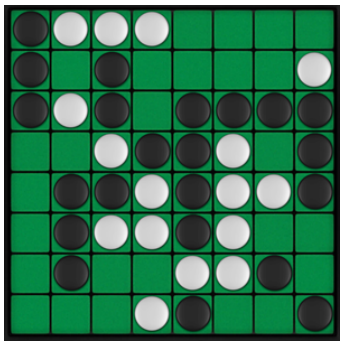

Fig. 1. Tablero reversi

El diseño de agentes capaces de tomar decisiones racionales en contextos competitivos ha sido durante décadas una de las áreas más activas en investigación en IA. Los juegos de tablero como Otelo proporcionan un marco controlado y perfectamente definido para experimentar con algoritmos de toma de decisiones, al tiempo que permiten evaluar el rendimiento y la capacidad de planificación de dichos agentes.

En este proyecto, se ha abordado la implementación de un agente capaz de jugar a Otelo de forma eficiente mediante el uso de técnicas de búsqueda adversaria. En concreto, se emplea el algoritmo **minimax** con **poda alfa-beta**, una mejora clásica que permite reducir significativamente el número de estados evaluados sin comprometer la calidad de las decisiones. Sin embargo, el principal reto radica en gestionar la complejidad computacional que implica explorar el espacio de estados posibles, especialmente en las fases medias y finales del juego.

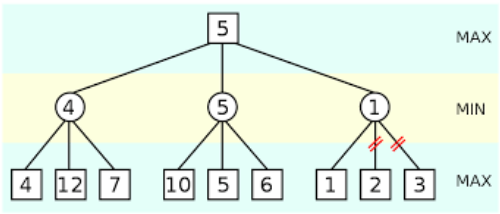

Fig. 2. Algoritmo minmax con poda alfa-beta

Este documento se estructura de la siguiente manera: en la **Sección II. Preliminares** se describen los fundamentos teóricos y las herramientas utilizadas, incluyendo el algoritmo minimax con poda alfa-beta y la estructura básica del agente. A continuación, la **Sección III. Metodología** detalla el proceso de desarrollo del proyecto, acompañado de pseudocódigo explicativo y decisiones de implementación. La **Sección IV. Resultados** presenta los experimentos realizados y los resultados obtenidos, y finalmente, en la **Sección V. Conclusiones**, se expone una reflexión general sobre el trabajo, incluyendo tanto sus aportaciones como posibles líneas de mejora futura. Finalmente, se incluye una **Sección VI. Glosario**, donde se incluyen la definición de ciertas palabras claves citadas en el documento.

II. Preliminares

En esta sección se introducen brevemente las técnicas y métodos empleados en el desarrollo del proyecto.

A. *Métodos empleados*

El agente desarrollado para jugar a Otelo se apoya principalmente en algunos pilares fundamentales de la inteligencia artificial relacionados con la toma de decisiones en juegos:

- **Búsqueda adversaria:** Esta técnica es utilizada habitualmente en escenarios donde intervienen más de un jugador. Esto genera una incertidumbre debido a que no es posible controlar los cambios de estado que realiza el jugador contrincante. En primer lugar, vamos a citar ciertas características del juego Reversi que hace posible la implementación de un agente que tome decisiones acertadas basándose en una búsqueda adversaria:
  - Es **determinista** por lo que todas las acciones a realizar pueden llegar a ser predecibles.
  - Participan dos jugadores
  - Está basado en turnos
  - Se trata de un juego de **suma nula** por tanto si un jugador gana el otro pierde.
  - **Información perfecta:** Cada jugador que participe en el juego tiene conocimiento total del estado de la partida en todo momento.

  Estas características pueden llegar a ser modeladas de la siguiente manera:
  - Conjunto de **estados** S, teniendo por situación inicial s_0.
  - Conjunto de **jugadores** $P = \{1, \ldots, n\}$, en nuestro caso para el Reversi $n = 2$. Conjunto de **acciones** A, dependendiendo del estado y del jugador.
  - **Función de transición** $T : S \times A \to S$, de manera que se determine la manera en la que cambia el estado del juego al aplicar una acción.
  - **Función de finalización** $g : S \to \{\text{true}, \text{false}\}$, indicando si el estado actual se trata de un estado terminal.
  - **Función de utilidad** $U : S \times P \to R$, asignando un valor numérico a los estados terminales para cada jugador, reflejando qué tan favorables son esos estados.

  La idea principal de la búsqueda adversaria es determinar una **política** $\pi : S \to A$ que devuelva la acción más óptima para cada estado, indicando la jugada a realizar.

- **Algoritmo minmax:** Se utiliza el algoritmo *minimax* para explorar los posibles movimientos del jugador y del oponente, modelando el juego como un árbol de decisiones. Este enfoque considera que ambos jugadores juegan de forma óptima y selecciona las jugadas que maximizan la ganancia del agente de manera que se minimice la pérdida máxima dada por los movimientos del adversario.

  La idea principal del algoritmo **minmax** consiste en, dado un estado, generar todas las posibles acciones sucesivas hasta un cierto límite de **niveles**. Se aplica una función que analice los estados obtenidos evaluando las ventajas y desventajas de cada posición y se elige aquella acción que lleve al jugador a la mejor posición. Dicha función se denomina **función de evaluación** que devuelve valores positivos para indicar las buenas situaciones para el jugador y valores negativos para indicar las buenas situaciones del adversario. El objetivo principal del algoritmo es maximizar el valor de dicha función sobre la posición actual del juego.

  La aplicación del algoritmo *Minimax* sigue de la siguiente manera:

  1) **Generación del árbol de juego**: A partir del nodo que representa el estado actual del juego, se generan todos los posibles estados sucesores, expandiendo el árbol hasta alcanzar nodos terminales.
  2) **Evaluación de nodos terminales**: Se calcula el valor de la función de evaluación $U(s)$ para cada nodo terminal s del árbol generado.
  3) **Propagación de valores**: Se evalúan los nodos intermedios del árbol a partir de los valores de sus hijos. Si el nodo pertenece al jugador MAX, se selecciona el valor máximo; si pertenece al jugador MIN, se selecciona el valor mínimo. Esto simula el comportamiento racional de ambos jugadores.
  4) **Retropropagación**: Se repite el paso anterior de forma recursiva hasta llegar al nodo raíz (estado actual del juego).
  5) **Selección de jugada**: Se elige la acción correspondiente al nodo hijo de la raíz que optimiza el valor de la evaluación. Esta será la jugada seleccionada por el agente.

  En caso de no ser posible la realización completa del árbol, se dispone de una función que evalúe un estado intermedio siguiendo una **heurística** dependiendo de esta la calidad de la estrategia dada por el algoritmo. Algunas condiciones de corte de generación del árbol son las siguientes:
  - Algún jugador ha ganado la partida (estado terminal alcanzado).
  - Se han explorado N niveles en profundidad, donde N es un parámetro establecido previamente.
  - Se ha agotado el tiempo máximo asignado para la búsqueda.
  - Se ha alcanzado una situación estática, es decir, los estados sucesivos no presentan cambios significativos o mejoras, lo que indica que continuar explorando no aportará beneficios relevantes.

- **Poda *alpha-beta*:** Dado el elevado número de estados posibles que pueden derivarse en cada turno, se implementa una optimización mediante *poda alpha-beta*, la cual permite reducir significativamente el número de nodos que deben evaluarse, descartando aquellas ramas que no afectarán al resultado final. Esto mejora el rendimiento sin comprometer la calidad de las decisiones tomadas.

La poda alfa-beta es una técnica de optimización aplicada al algoritmo Minimax que permite reducir el número de nodos evaluados en el árbol de juego, sin afectar al resultado final. Su objetivo es evitar la exploración de aquellas ramas que no pueden influir en la decisión óptima, mejorando así la eficiencia computacional del proceso de búsqueda.

La idea de esta técnica es que cada nodo del árbol se analiza teniendo en cuenta el valor que ha alcanzado hasta el momento y el valor de su nodo padre. Esto define, en cada instante, un intervalo (α, β) de posibles valores que puede tomar el nodo evaluado.

El significado intuitivo de estos parámetros depende del tipo de nodo:

- **En nodos MAX:**
  - α representa el valor actual del nodo.
  - β representa el valor del padre.
- **En nodos MIN:**
  - β representa el valor actual del nodo.
  - α representa el valor del padre.

La poda se produce cuando, en algún momento, se cumple que $\alpha \geq \beta$. En ese instante, no es necesario analizar los sucesores restantes del nodo, ya que no podrán influir en la decisión final.

- En nodos MIN, esta situación se denomina **poda-β**.
- En nodos MAX, se denomina **poda-α**.

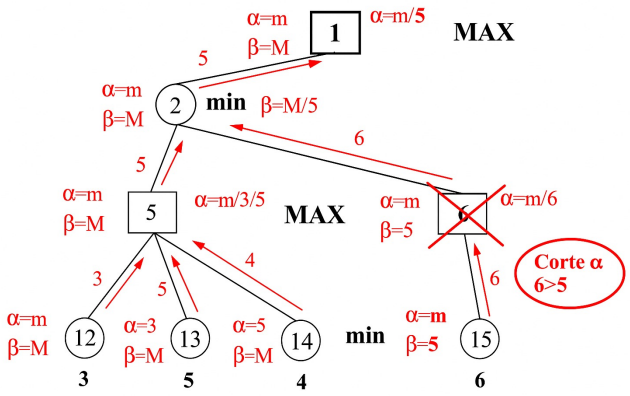

Fig. 3. Ejemplo de poda-α

Desde un punto de vista computacional, si r es el factor de ramificación (número de hijos por nodo) y m es la profundidad máxima del árbol, la complejidad en tiempo y espacio del algoritmo Minimax es del orden $O(r^m)$. La poda alfa-beta puede reducir este número significativamente en el mejor de los casos, aproximando la complejidad a $O(r^{m/2})$, lo cual permite explorar árboles más profundos en el mismo tiempo.

- **Entrenamiento supervisado a partir de autogeneración de datos:** Se ha desarrollado un generador automático de datos que simula partidas completas de Reversi entre jugadores que realizan movimientos válidos de forma aleatoria. A partir de estas simulaciones, se extraen todos los estados intermedios del tablero, y a cada uno se le asigna como etiqueta el resultado final de la partida (victoria o derrota) desde la perspectiva del jugador que realizó la jugada. Este enfoque permite crear un conjunto de datos masivo y equilibrado sin necesidad de recopilar partidas humanas, lo que facilita el entrenamiento de modelos de aprendizaje profundo de manera totalmente autónoma. Además, esta estrategia favorece una cobertura amplia de estados posibles del juego, promoviendo la generalización del modelo.
- **Redes neuronales convolucionales (CNN):** Para dotar al agente de una capacidad de evaluación heurística avanzada del tablero, se ha entrenado una red neuronal convolucional (CNN) usando los datos generados automáticamente. La arquitectura CNN permite capturar patrones espaciales del tablero de Otelo, analizando relaciones locales entre fichas y configuraciones estratégicas relevantes. La red toma como entrada una representación del tablero (matriz 8×8) codificada como un tensor tridimensional, y devuelve una probabilidad estimada de victoria desde esa posición concreta. Este valor se utiliza como función de evaluación en el algoritmo *minimax* con poda alfa-beta, lo cual permite acortar la profundidad de búsqueda manteniendo una toma de decisiones inteligente. El uso de la red como heurística resulta especialmente útil en etapas intermedias del juego, donde aún no se ha alcanzado un estado terminal y la evaluación basada únicamente en el conteo de fichas resulta poco informativa.

III. Metodología

En esta sección se describe la modelización del juego Othello (Reversi), la generación de partidas para el entrenamiento del agente, el entrenamiento de la red neuronal que evalúa posiciones y la implementación del algoritmo Minimax con poda Alpha-Beta utilizado para la toma de decisiones del agente inteligente.

En primer lugar, es necesario modelar el juego reversi, siendo este el entorno de trabajo donde se aplicará el algoritmo calculando la utilidad por una red neuronal que previamente ha sido entrenada con partidas de prueba generadas automáticamente. Por simplificación, se define que el jugador humano será siempre el jugador 1(negro) y la IA será el jugador 2(blanco).

A. Modelización del juego Reversi

A continuación, se describen los principales componentes de la lógica del juego Reversi. Cada bloque se acompaña de una breve explicación y su correspondiente pseudocódigo.

Inicialización del tablero

Al comenzar el juego, se debe inicializar un tablero de tamaño 8×8, donde todas las casillas están vacías, salvo

las cuatro centrales, que contienen dos fichas negras y dos blancas colocadas en forma diagonal, según las reglas estándar de Reversi.

```
Inicializar tablero:
  Crear tablero inicial 8 × 8
  Asignar fichas iniciales: 2 blancas, 2 negras
```

Detección de movimientos válidos

En cada turno, es necesario determinar qué movimientos puede realizar el jugador actual. Un movimiento es válido si, al colocar una ficha en una casilla vacía, se encierra al menos una ficha del oponente entre la ficha colocada y otra del jugador en alguna dirección.

```
Movimientos válidos(jugador, tablero):
  movimientos ← []
  Para cada celda vacía:
    Si movimiento válido para jugador:
      Agregar movimiento a movimientos
  Devolver movimientos
```

Realizar un movimiento

Una vez determinado un movimiento válido, el jugador coloca su ficha en la casilla correspondiente. Todas las fichas del oponente que queden encerradas entre esta ficha y otra del jugador se voltean. Después, el turno pasa al jugador contrario.

```
Hacer movimiento(movimiento, jugador):
  Si movimiento válido:
    Colocar ficha en tablero
    Voltear fichas del oponente
    Cambiar turno al otro jugador
```

Volteo de fichas

Este procedimiento se encarga de identificar y voltear las fichas del oponente que queden encerradas entre fichas del jugador, en cualquiera de las ocho direcciones posibles (horizontal, vertical o diagonal).

```
Voltear fichas(x, y, jugador):
  Para cada dirección:
    Recorrer fichas del oponente
    Si al final hay ficha del jugador:
      Voltear fichas del oponente
```

Condición de fin del juego

El juego termina cuando ninguno de los dos jugadores puede realizar un movimiento válido. Esto se verifica comprobando los movimientos posibles de ambos jugadores.

```
Fin del juego():
  Ambos jugadores no tienen movimientos válidos
```

Determinación del ganador

Una vez finalizado el juego, se cuentan las fichas de cada jugador en el tablero. El jugador con mayor cantidad de fichas es declarado ganador. En caso de empate en el número de fichas, el resultado también se declara como empate.

```
Obtener ganador():
  Contar fichas de cada jugador
  Si negro gana: devolver 1
  Si blanco gana: devolver 2
  Sino devolver 0 (empate)
```

Generación de datos para entrenamiento

Para entrenar modelos de inteligencia artificial en el contexto del juego Reversi, es necesario contar con un conjunto de datos que represente diferentes situaciones del tablero junto con la información del resultado final de la partida. Para ello, se simulan múltiples partidas completas de forma automática, seleccionando movimientos aleatorios en cada turno.

Durante la simulación, se registran los estados del tablero en cada jugada y el jugador que realizó dicha acción. Al finalizar la partida, se etiqueta cada estado con el resultado desde la perspectiva del jugador que hizo el movimiento en ese turno: victoria $(+1)$, derrota (-1) o empate (0). Estos datos son almacenados para su posterior uso en el entrenamiento del modelo.

```
Función generate_game_data(num_games):
  tableros ← []
  etiquetas ← []
  Para i=1 hasta num_games:
    Crear nueva instancia de juego
    estados ← []
    jugadores ← []
    Mientras game.is_game_over() es falso:
      Copiar estado actual del tablero
      Añadir copia a estados
      Añadir jugador actual a jugadores
      Obtener movimientos válidos para jugador actual
      Si no hay movimientos válidos:
        Cambiar jugador actual al otro jugador
        Continuar ciclo
      Elegir movimiento aleatorio de movimientos válidos
      Ejecutar movimiento en el juego
  ganador ← game.get_winner()
    Para cada estado y jugador:
      Si hay empate: etiqueta ← 0
      Sino si jugador es ganador: etiqueta ← 1
      Sino: etiqueta ← -1
      Añadir estado a tableros
      Añadir etiqueta a etiquetas
  Guardar datos en archivo
```

B. Algoritmo Alpha-Beta con evaluación por red neuronal

Para dotar al sistema de inteligencia artificial con capacidad de decisión avanzada, se implementa el algoritmo de búsqueda adversaria **Alpha-Beta**, una versión optimizada del algoritmo *Minimax*. Esta variante reduce el número de nodos evaluados en el árbol de decisiones al descartar ramas que no influyen en la decisión final, usando dos parámetros: α (mejor valor garantizado para el jugador maximizador) y β (mejor valor garantizado para el jugador minimizador).

En lugar de emplear una heurística clásica, la evaluación de los estados del juego se realiza utilizando una red neuronal entrenada previamente. Esta red toma como entrada el estado actual del tablero y devuelve una estimación del valor de utilidad desde la perspectiva del jugador maximizador.

```
Func alphabeta(s, depth,α, β, jugador, max, modelo):
  Si depth = 0 o juego terminado:
    Devolver (None, utility(s, modelo))
  movimientos ← movimientos válidos para jugador
  Si movimientos vacíos:
    Cambiar turno al otro jugador y cambiar max
    Devolver alphabeta cambiando de jugador
  Si maximizando:
    max_eval ← −∞
    mejor_movimiento ← None
    Para cada movimiento en movimientos:
      s_copia ← copia profunda de s
      aplicar movimiento para jugador en s_copia
      eval ← alphabeta con jugador no MAX
      Si eval > max_eval:
        max_eval ← eval
        mejor_movimiento ← movimiento
      α ← máximo entre α y eval
      Si β ≤ α:
        Romper ciclo
    Devolver (mejor_movimiento, max_eval)
  Si minimizando:
    min_eval ← +∞
    mejor_movimiento ← None
    Para cada movimiento en movimientos:
      s_copia ← copia profunda de s
      aplicar movimiento para jugador en s_copia
      eval ← alphabeta con jugador MAX
      Si eval < min_eval:
        min_eval ← eval
        mejor_movimiento ← movimiento
      β ← mínimo entre β y eval
      Si β ≤ α:
        Romper ciclo
    Devolver (mejor_movimiento, min_eval)
```

La función **utility** es la encargada de estimar el valor de un estado del juego usando una red neuronal previamente entrenada. Toma como entrada el estado del tablero y devuelve un valor numérico que representa la probabilidad de victoria desde la perspectiva del jugador maximizador.

```
Función utility(s, modelo):
  Convertir tablero de s a arreglo 8x8x1
  Evaluar con modelo para obtener valor de utilidad
  Devolver valor
```

C. Uso del agente IA

El agente de inteligencia artificial actúa cuando le corresponde el turno al jugador controlado por la máquina (jugador2). Para decidir su jugada, utiliza el algoritmo `alphabeta`, evaluando los posibles movimientos a una profundidad determinada mediante una red neuronal entrenada. Si no hay movimientos válidos, se salta el turno. En caso contrario, el agente elige la mejor jugada disponible según la evaluación de utilidad y la ejecuta en el tablero.

```
Si jugador actual = jugador2 (IA):
  movimientos ← movimientos válidos para jugador2
  Si movimientos vacíos:
    Cambiar jugador actual
    Continuar con siguiente iteración
  α← −∞
  β← +∞
  movimiento ← alphabeta en estado actual
  Mostrar movimiento elegido
  Si movimiento no es None:
    Aplicar movimiento para jugador2 en el juego
```

IV. Resultados

En esta sección se detallan los experimentos llevados a cabo, los parámetros utilizados y los resultados obtenidos tras la implementación del sistema de inteligencia artificial para el juego Othello. Además, se incluye un análisis razonado de dichos resultados.

A. Experimentos Realizados

Se han diseñado y ejecutado varios experimentos con el objetivo de evaluar el rendimiento del agente de IA en diferentes configuraciones. En concreto, se han llevado a cabo los siguientes:

- **Evaluación del agente frente a un jugador aleatorio:** Se enfrentó el agente inteligente (con Poda Alpha-Beta) contra un oponente que realiza movimientos aleatorios. El objetivo fue verificar que la IA es capaz de superar consistentemente a un jugador no estratégico.
- **Pruebas de rendimiento:** Se midió el tiempo de decisión del agente para distintas profundidades del árbol de búsqueda (desde profundidad 1 hasta 4), evaluando la escalabilidad y eficiencia del algoritmo.

B. Parámetros Utilizados en los Experimentos

Durante los experimentos, se utilizaron los siguientes parámetros de configuración:

- **Tamaño del tablero:** 8x8 (configuración estándar del juego Othello).
- **Profundidades de búsqueda:** Desde 1 hasta 6 niveles, para comparar el impacto en rendimiento y precisión de decisión.
- **Algoritmos evaluados:** Minimax con Poda Alpha-Beta.
- **Heurística utilizada:**
  - Diferencia de fichas entre jugadores.
- **Entrenamiento de la red neuronal:**
  - Número total de partidas utilizadas para el entrenamiento: **10.000**.
  - Se realizaron comparaciones de rendimiento del agente tras **1**, **50** y **100 épocas** de entrenamiento

para evaluar la mejora progresiva de la calidad de las decisiones.

- **Condiciones de las partidas:**
  - Agente vs agente y agente vs jugador aleatorio.
  - Se jugaron 3 partidas por cada configuración.
- **Hardware:** Las pruebas se ejecutaron en un equipo con procesador Ryzen 7 con 24 GB de RAM.

C. Resultados Obtenidos

A continuación se presentan los resultados más relevantes para cada experimento:

- **Agente vs Jugador Aleatorio:**
  - Se jugaron 3 partidas por cada configuración de entrenamiento (1, 50 y 100 épocas) enfrentando al agente contra un oponente que selecciona movimientos aleatorios.
  - **Con 1 época de entrenamiento:** El agente mostró un comportamiento poco consistente. Aunque conocía las reglas básicas del juego y era capaz de realizar jugadas válidas, se observaron decisiones subóptimas. El porcentaje de victorias fue del 33%(1 victoria, 1 empate y 1 derrota), y el número medio de fichas capturadas fue de 30.

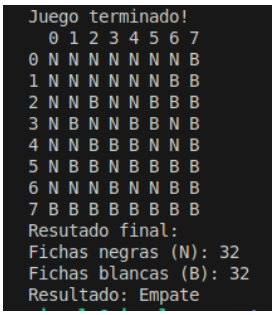

Fig. 4. Resultado de partida con 1 época de entrenamiento.

  - **Con 50 épocas de entrenamiento:** El comportamiento del agente mejoró significativamente. Comenzó a priorizar posiciones estratégicas (bordes y esquinas) y a reducir los errores no forzados. Ganó las 3 partidas jugadas, con una media de 38 fichas capturadas. Se observó una mayor capacidad para limitar la movilidad del oponente.

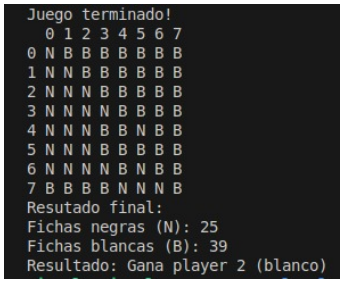

Fig. 5. Resultado de partida con 100 épocas de entrenamiento.

  - **Con 100 épocas de entrenamiento:** El agente alcanzó un nivel de juego notablemente más sólido y estable. No solo ganó las 3 partidas jugadas, sino que además lo hizo con márgenes amplios, capturando en promedio 48 fichas. Su toma de decisiones fue más refinada y anticipó eficazmente los movimientos del rival, mostrando una estrategia defensiva y ofensiva equilibrada.estrategia defensiva y ofensiva equilibrada.

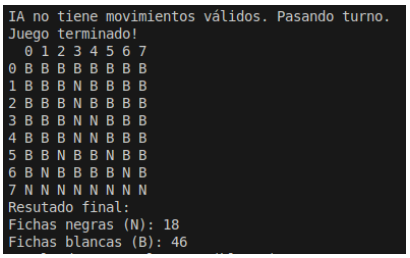

Fig. 6. Resultado de partida con 100 épocas de entrenamiento.

- **Rendimiento por profundidad:**

TABLA I
RENDIMIENTO DEL AGENTE CON PODA ALPHA-BETA SEGÚN LA PROFUNDIDAD DE BÚSQUEDA.

Depth	Tiempo medio por jugada (s)	Nodos explorados	Victoria
1	0.311	287	Sí
2	0.530	637	Sí
3	2.513	2084	No
4	4.618	5259	Sí
5	15.800	14776	Sí

D. Análisis de Resultados

La elevada tasa de victoria contra jugadores aleatorios demuestra que el agente no solo toma decisiones válidas, sino que además aplica una estrategia ganadora. También se ha comprobado que el número de nodos explorados crece exponencialmente con la profundidad, lo cual valida el uso de podas y heurísticas en problemas combinatorios como Othello.

Por otra parte, observamos que no es necesario profundizar demasiado en el árbol de decisiones para obtener buenos resultados, observamos que con niveles de profdundidad 1 y 2 obtenemos tiempos de respuesta bastante óptimos obteniendo una victoria holgada. Sin embargo conforme vamos aumentando el nivel de profundidad, los tiempos de ejecución aumentan considerablemente y a pesar de obtener la victoria, habría que valorar si es favorable sacrificar tiempo de ejecución por una posible mejora de resultados.

En conclusión, el agente implementado muestra un comportamiento eficaz y estratégico, siendo capaz de jugar partidas completas de Othello con tiempos de respuesta razonables y un alto rendimiento frente a oponentes no inteligentes. La implementación de Poda Alpha-Beta ha resultado ser crucial para obtener estos resultados.

V. Conclusiones

En este proyecto se ha conseguido construir un agente basado en inteligencia artificial que, utilizando algoritmos como el mencionado minimax con poda alfa-beta, sea capaz de interpretar el papel de un oponente en una partida al clásico juego de otelo. Tras barajar algunas opciones, se optó por modelar el tablero como una matriz 8x8 en la que los valores 1 y 2 nos informan de si la ficha es blanca o negra. Una vez el tablero estaba desarrollado, se empezó con el entrenamiento del agente usando el algoritmo minimax, con el cual se puede generar el árbol de juego que posteriormente es evaluado desde los nodos terminales pasando por los intermedios hasta el nodo raíz (estado del juego), todo esto apoyado por una heurística de decisión para casos en los que no se pueda recorrer el árbol al completo. Para poder entrenar a la red neuronal que nos ayuda con la heurística se produjeron datos de entrenamiento a base de partidas generadas aleatoriamente; el agente, posteriormente, se encarga de utilizar esta red neuronal ya entrenada para la toma de decisiones. Gracias al uso del algoritmo minimax con poda alfa-beta hemos podido solventar una tarea que, sin ayuda de este recurso o de uno similar, parecía prácticamente imposible a nivel de rendimiento computacional.

Con respecto a algunas mejoras a futuro, se podría plantear el uso de un algoritmo distinto como negamax, una variante del minimax que haría que el código fuera más sencillo y compacto; también se podría cambiar el entrenamiento de la red neuronal, utilizando, en lugar de partidas aleatorias, registros de partidas entre agentes fuertes o partidas humanas de gran nivel.

VI. Glosario

Agente: Entidad autónoma capaz de percibir su entorno y tomar decisiones racionales para alcanzar un objetivo. En este proyecto, el agente es el jugador controlado por IA.

Algoritmo minimax: Algoritmo clásico de decisión usado en juegos de dos jugadores. Evalúa posibles jugadas asumiendo que ambos jugadores juegan óptimamente, eligiendo movimientos que maximizan la ganancia mínima esperada.

CNN (Red Neuronal Convolucional): Tipo de red neuronal profunda especialmente útil para procesar datos con estructura espacial, como imágenes o tableros de juego. En este caso, se usa para evaluar posiciones del tablero de Otelo.

Heurística: Aproximación o regla empírica usada para evaluar rápidamente un estado intermedio. En este caso, una red neuronal actúa como heurística para predecir la probabilidad de victoria desde una posición dada.

Poda alfa-beta: Optimización del algoritmo minimax que reduce el número de nodos evaluados al eliminar ramas del árbol de decisión que no afectan el resultado final. Mejora la eficiencia sin perder calidad en la decisión.

Referencias

[1] https://www.cs.us.es/ fsancho/Blog/posts/Minimax.md
[2] https://es.wikipedia.org/wiki/Poda_alfa-beta
[3] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, 4th ed., Pearson, 2020, ch. 5, “Adversarial Search”.
[4] https://chatgpt.com/ como apoyo para redacción