Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Predicción de movimientos humanos en ajedrez adaptable al ELO

Tesis de Licenciatura en Ciencias de la Computación

Gianfranco Bogetti

Director: Diego Fernandez Slezak

Codirector: Agustín Sansone

Buenos Aires, 2025

PREDICCIÓN DE MOVIMIENTOS HUMANOS EN AJEDREZ ADAPTABLE AL ELO

Con el rápido avance de la inteligencia artificial (IA), que alcanza en ocasiones niveles de rendimiento sobrehumano, surge una valiosa oportunidad para que los humanos aprendan de los algoritmos y colaboren con ellos. Sin embargo, las diferencias en los enfoques de resolución de problemas entre humanos y máquinas a menudo dificultan la comprensión e integración de los sistemas de IA en el aprendizaje humano.

Este trabajo presenta el desarrollo de un modelo de inteligencia artificial adaptable al nivel de habilidad humana, medido a través del ELO, que predice movimientos en ajedrez. Para lograrlo, se modificó la arquitectura de redes neuronales de Leela Chess Zero para integrar el ELO como un parámetro de entrada. El modelo fue entrenado y evaluado utilizando más de 10 millones de partidas de ajedrez de la base de datos de Lichess.

Los resultados muestran que el modelo es capaz de alcanzar niveles de exactitud incluso mayores a los de trabajos relacionados como Maia Chess, además de aportar la adaptabilidad del ELO. Este enfoque no solo amplía su aplicabilidad a otros dominios más allá del ajedrez, sino que también abre nuevas posibilidades para el desarrollo de tutores artificiales personalizados.

Palabras claves: Inteligencia artificial, Comportamiento humano, Nivel de habilidad, Aprendizaje, Tutores artificiales, Ajedrez.

PREDICCIÓN DE MOVIMIENTOS HUMANOS EN AJEDREZ ADAPTABLE AL ELO

With the rapid advancement of artificial intelligence (AI), which at times reaches superhuman performance levels, a valuable opportunity arises for humans to learn from algorithms and collaborate with them. However, differences in problem-solving approaches between humans and machines often hinder the understanding and integration of AI systems into human learning.

This work presents the development of an artificial intelligence model adaptable to human skill level, measured through ELO, that predicts chess moves. To achieve this, the neural network architecture of Leela Chess Zero was modified to integrate ELO as an input parameter. The model was trained and evaluated using more than 10 million chess games from the Lichess database.

The results show that the model is capable of reaching accuracy levels even higher than those of related works such as Maia Chess, while also incorporating ELO adaptability. This approach not only broadens its applicability to other domains beyond chess, but also opens up new possibilities for the development of personalized artificial tutors.

Keywords: Artificial intelligence, Human behavior, Skill level, Learning, Artificial tutors, Chess.

Índice general

1. INTRODUCCIÓN

1.1. Problema a estudiar

La inteligencia artificial (IA) ha experimentado un crecimiento exponencial en sus capacidades, redefiniendo constantemente los límites del rendimiento cognitivo. Este avance ha permitido a las máquinas no solo igualar, sino en muchos casos superar, el desempeño humano en tareas complejas, como el reconocimiento de patrones, la resolución de problemas y la toma de decisiones estratégicas [14][15][16]. Sin embargo, este progreso plantea nuevos desafíos, especialmente en la interacción entre humanos y sistemas de IA.

Uno de los principales problemas radica en que las soluciones desarrolladas por la IA suelen diferir significativamente de las estrategias utilizadas por los seres humanos. Estas diferencias, aunque pueden ofrecer resultados altamente efectivos, dificultan su comprensión y asimilación por parte de los usuarios humanos. Esto genera una desconexión entre los sistemas de IA y las personas, reduciendo su aplicabilidad en situaciones donde la colaboración es fundamental, como en el caso del aprendizaje.

Además, los sistemas de IA suelen optimizar su rendimiento para maximizar métricas globales, como la tasa de éxito o la precisión, sin tener en cuenta las características individuales de los usuarios humanos que interactúan con ellos. Esta falta de personalización y adaptabilidad representa una limitación clave para su integración efectiva en aplicaciones prácticas.

El juego del ajedrez ofrece un escenario ideal para abordar este problema. Históricamente, ha servido como un banco de pruebas tanto para el desarrollo de la IA como para el estudio de la psicología cognitiva. Alan Turing fue pionero en explorar las posibilidades de las máquinas para jugar ajedrez, desarrollando en 1951 un programa capaz de jugar una partida completa de ajedrez [1]. Claude Shannon, en 1950, publicó "Programming a Computer for Playing Chess", donde discutió estrategias para que una máquina pudiera jugar ajedrez de manera efectiva [2].

Herbert A. Simon y Allen Newell también realizaron contribuciones significativas al campo de la IA y el ajedrez. Juntos, sentaron las bases para la simulación del pensamiento humano y la resolución de problemas [3][4].

La victoria de Deep Blue contra Garry Kasparov en 1997 marcó un hito al demostrar que las máquinas podían superar a los mejores jugadores humanos [5]. Desde entonces, motores como AlphaZero han llevado esta capacidad aún más lejos, empleando redes neuronales y aprendizaje por refuerzo para alcanzar niveles de rendimiento sobrehumanos sin depender de datos de partidas humanas [6].

Sin embargo, estos avances también ponen de manifiesto las limitaciones de los motores de ajedrez actuales cuando se trata de replicar y comprender el comportamiento humano en el juego. Por ejemplo, motores como AlphaZero [7], Leela Chess Zero [20][21] y Stockfish [17] están diseñados para maximizar la probabilidad de victoria, sin considerar las decisiones que un jugador humano con un nivel específico de habilidad probablemente tomaría en una situación dada. Esto los hace inadecuados para escenarios donde la meta es entender, predecir o incluso enseñar comportamiento humano.

En este contexto, Maia Chess [8][18], desarrollado en 2020, introdujo un enfoque innovador al crear modelos diseñados para predecir movimientos humanos en función de rangos

específicos de ELO. Sin embargo, este enfoque segmentado presenta limitaciones, ya que no permite la adaptabilidad a valores de ELO intermedios o personalizados. Además, exige múltiples modelos para diferentes niveles, lo que complica su uso debido a la baja versatilidad de la solución.

Por lo tanto, existe una necesidad creciente de desarrollar modelos de IA que no solo sean altamente exactos, sino también capaces de adaptarse dinámicamente al nivel de habilidad humana, facilitando su empleo y así la comprensión, la enseñanza y la colaboración entre humanos y máquinas. Este desafío va más allá del ajedrez, ya que plantea preguntas fundamentales sobre cómo diseñar sistemas de IA que puedan integrarse de manera efectiva en entornos donde la interacción humano-computadora sea crucial.

1.2. Objetivos del trabajo

El objetivo principal de este trabajo es desarrollar un modelo de inteligencia artificial único y adaptable, capaz de predecir con exactitud los movimientos realizados por jugadores humanos en partidas de ajedrez, considerando su nivel de habilidad representado por el ELO. Este modelo busca abordar la limitación de enfoques previos, como Maia Chess, que dependen de modelos distintos para distintos rangos de ELO. Para superar estos problemas, el presente trabajo propone una modificación a la arquitectura de Leela Chess Zero que permita ajustar dinámicamente el valor de ELO al que se espera que el modelo juegue, logrando así una solución más adaptable. Este enfoque requiere entrenar redes neuronales profundas con una gran cantidad de datos de manera representativa de partidas humanas, obtenidos de bases de datos en línea, para permitir que el modelo aprenda a emular los movimientos que los jugadores reales tomarían en situaciones específicas.

2. DATOS

2.1. ELO

El sistema de puntuación ELO [22] es un método matemático basado en cálculos estadísticos para evaluar la habilidad relativa de los jugadores en deportes como el ajedrez. Este sistema utiliza los resultados de partidas entre jugadores para ajustar sus puntuaciones, reflejando cambios en su desempeño relativo.

La idea central es que, dadas las puntuaciones R_A y R_B de dos jugadores, es posible estimar la probabilidad esperada de que un jugador obtenga un resultado específico (ganar, empatar o perder) en una partida. La puntuación esperada para R_A se calcula con la fórmula:

$$E_A = \frac{1}{(1 + 10^{\frac{R_A - R_B}{400}})} \tag{2.1}$$

Tras una partida, el sistema ajusta las puntuaciones en función de la desviación entre los resultados esperados y los obtenidos, siguiendo la fórmula:

$$R'_A = R_A + K(S_A - E_A) \tag{2.2}$$

Donde K es el factor de ajuste, que depende del nivel de ELO del jugador:

$$K = \begin{cases} 32 & \text{si Elo} < 2100 \\ 24 & \text{si Elo} \in [2100; 2400] \\ 16 & \text{si Elo} > 2400 \end{cases} \tag{2.3}$$

y S_A es el resultado de la partida para el jugador A (1 para victoria, 0.5 para tablas, 0 para derrota).

En este trabajo, el sistema ELO sirve como parámetro principal para ajustar dinámicamente el modelo desarrollado. Al integrar el ELO como un input en la red neuronal, se busca no solo emular el comportamiento humano, sino también adaptarlo al nivel de habilidad del jugador.

2.2. Portable Game Notation

El formato Portable Game Notation (PGN) es un estándar ampliamente empleado en la comunidad ajedrecista para registrar, almacenar y compartir partidas de ajedrez. Su popularidad radica en su simplicidad, portabilidad y legibilidad para humanos, lo que lo convierte en una herramienta esencial para el análisis y el estudio del juego. Un archivo PGN consta de dos partes principales Metadatos (tags) y secuencia de movimientos.

En este trabajo, el formato PGN es la fuente principal de datos para el entrenamiento de la red neuronal. Su estructura estandarizada permite extraer tanto los movimientos como los metadatos asociados, que luego son procesados y transformados en representaciones numéricas (chunks) para ser usadas en el modelo de inteligencia artificial.

2.2.1. Metadatos

Incluyen información relevante sobre la partida, como los nombres de los jugadores, sus clasificaciones de ELO, el control de tiempo, el resultado final, entre otros detalles. Estos datos están diseñados para proporcionar un contexto completo de la partida, facilitando su análisis y categorización.

- Event: El nombre del evento o torneo en el que se jugó la partida.
- Site: La ubicación del evento, que en el caso de Lichess suele ser "Lichess.org".
- Date: La fecha en la que se jugó la partida.
- Round: El número de la ronda del torneo en la que se jugó la partida.
- White: El nombre del jugador con las piezas blancas.
- Black: El nombre del jugador con las piezas negras.
- Result: El resultado de la partida, que puede ser "1-0"(blanco gana), "0-1"(negro gana) o "1/2-1/2"(empate).
- BlackElo: El Elo del jugador con las piezas negras.
- BlackRatingDiff: Cambio en el rating del jugador con las piezas negras después de la partida.
- ECO: El código ECO (Encyclopaedia of Chess Openings) que representa la apertura utilizada.
- Opening: Nombre de la apertura utilizada en la partida.
- Termination: Razón de la terminación de la partida.
- TimeControl: Control de tiempo utilizado en la partida.
- UTCDate: Fecha en formato UTC en que se jugó la partida.
- UTCTime: Hora en formato UTC en que se jugó la partida.
- WhiteElo: El Elo del jugador con las piezas blancas.
- WhiteRatingDiff: Cambio en el rating del jugador con las piezas blancas después de la partida.

2.2.2. Notación algebraica estándar

La notación algebraica estándar es un método utilizado universalmente para registrar partidas de ajedrez de forma compacta.

En este sistema, cada casilla del tablero está identificada mediante una coordenada compuesta por una letra y un número. Las columnas se etiquetan de la a a la h, de izquierda a derecha desde la perspectiva de las blancas, y las filas se numeran del 1 al 8 de abajo a arriba también desde la perspectiva de las blancas.

Cada movimiento se describe por la pieza movida, indicada por una letra (N para el caballo, B para el alfil, R para la torre, Q para la dama y K para el rey, si la jugada involucra un peón, no se especifica la inicial) y la casilla de destino, y en caso de capturas, se añade una x antes de la casilla.

Las jugadas especiales, como enroques, se anotan como $O-O$ (enroque corto) y $O-O-O$ (enroque largo). Y las promociones se anotan con la posición del peón que promociono seguido del indicador de la pieza a la que se quiere promocionar.

Por último puede haber ambigüedades, estas se resuelven indicando luego del identificador de la pieza su posición y la posición de destino.

Ejemplos:

- $e4$: un peón avanza a la casilla $e4$.
- $Rf3$: una torre se mueve a $f3$.
- $Qxe5$: la dama captura en $e5$.
- $Nf4d5$: el caballo de $f4$ se mueve a $d5$.

2.2.3. Movimientos

Registra los movimientos realizados durante la partida en notación algebraica estándar, lo que permite reconstruir y analizar cada posición desde el inicio hasta el final de la partida.

```
1. e4 { [%eval 0.36] [%clk 0:03:00] } 1... c5 { [%eval 0.32] [%clk 0:03:00] }
B1 B2        B3             B4         N1  N2        N3            N4
```

- B1: Número del movimiento del jugador con las piezas blancas. (1.)
- N1: Número del movimiento del jugador con las piezas negras. (1...)
- B2: Movimiento del jugador con las piezas blancas. (e4)
- N2: Movimiento del jugador con las piezas negras. (c5)
- B3: Evaluación del movimiento efectuado por el jugador con las piezas blancas.
- N3: Evaluación del movimiento efectuado por el jugador con las piezas negras.
- B4: Tiempo restante del jugador con las piezas blancas.
- N4: Tiempo restante del jugador con las piezas negras.

La valuación es generada por un motor de ajedrez y se expresa en centipawns. Si el valor es positivo indica una ventaja para las blancas, mientras que un valor negativo señala una ventaja para las negras. En caso de existir una secuencia forzada de mate, se indicará en este campo precedido por el signo "#".

2.2.4. Ejemplo

```
[Event "Rated Blitz game"]
[Site "https://lichess.org/XXXXXX"]
[Date "2023.06.01"]
[Round "-"]
[White "user1"]
[Black "user2"]
[Result "1-0"]
[BlackElo "1474"]
[BlackRatingDiff "-6"]
[ECO "B32"]
[Opening "Sicilian Defense: Open"]
[Termination "Normal"]
[TimeControl "180+2"]
[UTCDate "2023.06.01"]
[UTCTime "00:00:07"]
[WhiteElo "1486"]
[WhiteRatingDiff "+9"]

1. e4 { [%eval 0.36] [%clk 0:03:00] } 1... c5 { [%eval 0.32] [%clk 0:03:00] }
2. Nf3 { [%eval 0.0] [%clk 0:03:01] } 2... Nc6 { [%eval 0.0] [%clk 0:03:01] }
3. d4 { [%eval 0.33] [%clk 0:03:02] } 3... cxd4 { [%eval 0.38] [%clk 0:03:03] }
4. Nxd4 { [%eval 0.29] [%clk 0:03:03] } 4... Nxd4 { [%eval 0.83] [%clk 0:03:05] }
5. Qxd4 { [%eval 0.82] [%clk 0:03:04] } 5... d6 { [%eval 0.92] [%clk 0:03:05] }
6. Nc3 { [%eval 0.87] [%clk 0:03:05] } 6... e5 { [%eval 2.58] [%clk 0:03:06] }
7. Bb5+ { [%eval 1.51] [%clk 0:02:59] } 7... Bd7 { [%eval 2.25] [%clk 0:03:06] }
8. Bxd7+ { [%eval 1.3] [%clk 0:02:58] } 8... Qxd7 { [%eval 1.39] [%clk 0:03:08] }
9. Qe3 { [%eval 1.0] [%clk 0:02:56] } 9... Nf6 { [%eval 0.93] [%clk 0:03:07] }
10. O-O { [%eval 0.7] [%clk 0:02:55] } 10... Be7 { [%eval 0.65] [%clk 0:03:08] }
11. b3 { [%eval 0.18] [%clk 0:02:38] } 11... O-O { [%eval 0.07] [%clk 0:03:07] }
12. Bb2 { [%eval 0.01] [%clk 0:02:39] } 12... Ng4 { [%eval 1.13] [%clk 0:03:08] }
13. Qd2 { [%eval 0.32] [%clk 0:02:35] } 13... Rfd8 { [%eval 1.05] [%clk 0:02:54] }
14. Nd5 { [%eval 0.46] [%clk 0:02:33] } 14... Nf6 { [%eval 0.42] [%clk 0:02:50] }
15. Nxf6+ { [%eval 0.61] [%clk 0:02:27] } 15... Bxf6 { [%eval 0.47] [%clk 0:02:51] }
16. Rad1 { [%eval 0.27] [%clk 0:02:26] } 16... Qc6 { [%eval 0.49] [%clk 0:02:46] }
17. c4 { [%eval -0.31] [%clk 0:02:24] } 17... Rac8 { [%eval 1.31] [%clk 0:02:41] }
18. Rfe1 { [%eval 0.24] [%clk 0:02:24] } 18... Qa6 { [%eval 1.54] [%clk 0:02:38] }
19. a3 { [%eval 0.06] [%clk 0:02:20] } 19... Rc6 { [%eval 1.47] [%clk 0:02:32] }
20. Qe3 { [%eval 0.24] [%clk 0:02:13] } 20... Rdc8 { [%eval 1.36] [%clk 0:02:22] }
21. Qh3 { [%eval 0.05] [%clk 0:02:07] } 21... b5 { [%eval 0.04] [%clk 0:02:16] }
22. cxb5 { [%eval 0.15] [%clk 0:01:57] } 22... Qxb5 { [%eval 0.05] [%clk 0:02:17] }
23. Rxd6 { [%eval 0.06] [%clk 0:01:47] } 23... Rc2 { [%eval 0.22] [%clk 0:02:00] }
24. g3 { [%eval -4.5] [%clk 0:01:11] } 24... Rxb2 { [%eval #3] [%clk 0:01:59] }
25. Qxc8# { [%clk 0:01:02] } 1-0
```

2.2.5. Lichess Database

La Lichess Database [19] es una de las colecciones más extensas y accesibles de partidas de ajedrez disponibles actualmente. Proveniente de la plataforma en línea Lichess.org, esta base de datos incluye millones de partidas jugadas por jugadores de todos los niveles, desde principiantes hasta grandes maestros, abarcando una amplia variedad de formatos de tiempo, desde partidas ultrarrápidas como bullet hasta partidas clásicas más lentas.

Esta base de datos utiliza el formato PGN para almacenar las partidas, además de contener evaluaciones automáticas generadas por motores de ajedrez de alto rendimiento como Stockfish. Estas evaluaciones permiten identificar errores, imprecisiones y las mejores jugadas posibles en cada posición. Esta accesibilidad convierte a la Lichess Database en una herramienta indispensable para investigaciones relacionadas con el ajedrez.

En este trabajo, la Lichess Database se emplea como la fuente principal de datos para el entrenamiento y validación del modelo de inteligencia artificial. Las partidas son filtradas y procesadas para seleccionar únicamente aquellas que cumplen criterios específicos, como

el control de tiempo y la disponibilidad de ELO, asegurando que los datos utilizados sean representativos y adecuados para los objetivos del estudio.

2.2.6. Análisis de los datos

Para este trabajo se analizó las partidas registradas en la base de datos durante un período de seis meses, desde enero hasta junio de 2023. Para evaluar la calidad y estructura del conjunto de datos utilizado en este trabajo, se realizaron análisis que muestran la distribución de dos variables clave:

1. **Distribución de los tiempos de partida**
   Nos permitirá entender cómo se distribuyen las partidas en función del control de tiempo utilizado. Nos interesa analizar esta distribución, ya que es esperable que la manera en la que juegan los jugadores puede variar significativamente dependiendo del tiempo disponible para cada partida. Un mismo jugador podría tomar decisiones muy diferentes en partidas rápidas en comparación con aquellas de mayor duración, lo que afectaría el entrenamiento del modelo.

   Por esta razón, es fundamental identificar un control de tiempo que esté lo suficientemente representado en el conjunto de datos y que sea lo suficientemente estable en cuanto a la variabilidad de las partidas, de modo que podamos seleccionar partidas dentro de ese rango y asegurar que el modelo se entrene en un contexto de juego más homogéneo.

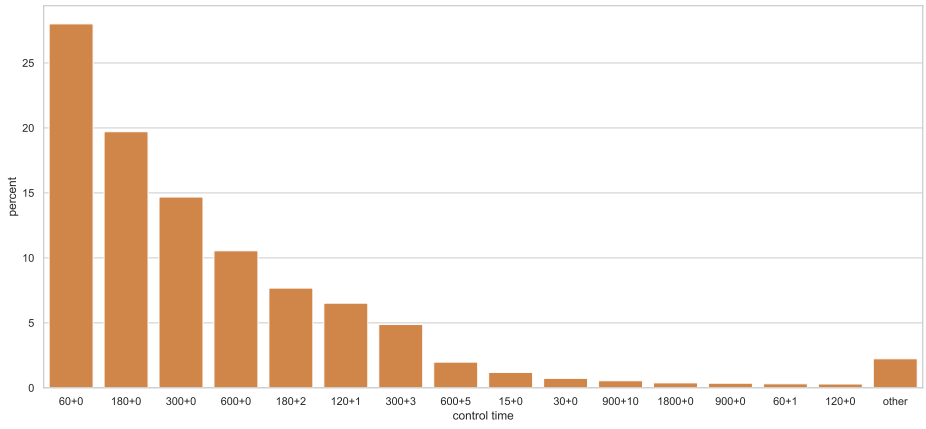

Fig. 2.1: Análisis distribución de tiempo de las partidas

El gráfico de distribución del tiempo de partida muestra que $60 + 0$ es el control de tiempo más frecuente en el dataset, pero otro lado en partidas muy rápidas (como $60 + 0$ o $180 + 0$), se sabe que las decisiones suelen ser impulsivas y con mayor cantidad de errores debido a la escasa cantidad de tiempo disponible, mientras que en partidas más largas (como $900 + 0$ o $1800 + 0$), los jugadores pueden pensar más profundamente, reflejando un estilo más posicional y estratégico.

El tiempo $600+0$ está lo suficientemente representado para llevar a cabo este trabajo y representa un punto intermedio en el que los jugadores pueden tomar decisiones

razonadas sin llegar a un juego extremadamente rápido o lento, lo que facilita la generación de un modelo equilibrado y nos permite capturar un estilo de juego estable. Por otro lado, en Maia Chess se tomó la decisión de solo descartar controles de tiempo iguales o inferiores a $60 + 0$ lo cual generara más variabilidad o ruido en los datos, que es algo que estamos tratando de reducir, por eso solo usaremos un control de tiempo.

2. **Distribución del ELO**
   Nos permite analizar la cantidad de partidas disponibles en cada rango de ELO dentro del control de tiempo elegido, lo que ayuda a identificar posibles sesgos en los datos y determinar qué niveles de habilidad están mejor representados. Si bien el objetivo es generar un modelo que generalice principalmente entre 1100 y 1900 de ELO porque es en este rango en el que predice Maia Chess, es importante analizar la distribución completa para conocer los porcentajes de partidas disponibles en cada nivel de habilidad y evaluar si la cantidad de datos es equilibrada en todo el rango de interés.

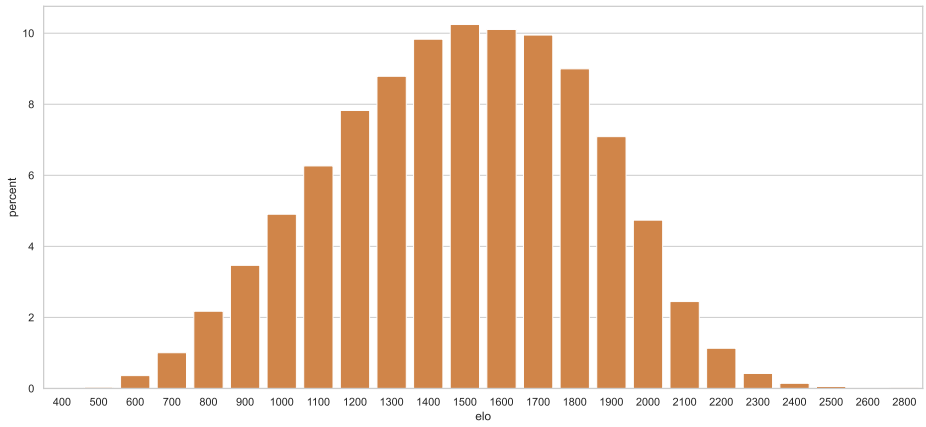

Fig. 2.2: Análisis distribución de ELO de partidas 600+0

En el gráfico nuestra una de distribución normal de los valores de ELO, se observa que la mayor concentración de partidas se encuentra en el rango de 1000 a 2000, lo cual es favorable dado que el objetivo del modelo es generalizar dentro de este intervalo. Sin embargo, también hay una cantidad significativa de partidas por fuera de los 1000 y 2000 con una disminución progresiva en la cantidad de datos a medida que el ELO aumenta o decrece.

3. DESARROLLO

En esta sección se detalla el proceso de desarrollo llevado a cabo durante este estudio, cuyo objetivo principal fue crear un modelo de inteligencia artificial capaz de predecir movimientos humanos en ajedrez, adaptándose dinámicamente al nivel de habilidad del jugador representado por su ELO.

El desarrollo comenzó con la recopilación y preparación de datos, fundamentales para garantizar la calidad y representatividad del entrenamiento del modelo. Posteriormente, se implementaron las modificaciones necesarias en la solución existente de Leela Chess Zero, adaptándola para cumplir con los requerimientos específicos de este trabajo. Finalmente, se llevaron a cabo una serie de experimentos diseñados para ajustar y validar el modelo, abordando los desafíos técnicos y metodológicos que surgieron durante el proceso.

A continuación, se describen en detalle las etapas del desarrollo, los problemas encontrados y las soluciones implementadas, proporcionando una visión completa del camino recorrido para alcanzar los objetivos planteados.

3.1. Lela Chess Zero

3.1.1. Leela Chess Zero Engine

Leela Chess Zero (Lc0) es un motor de ajedrez de código abierto hecho en $C++$ basado en redes neuronales, inspirado en AlphaZero de Google DeepMind. A diferencia de los motores de ajedrez tradicionales como Stockfish, que solo utilizan tablas de evaluación y árboles de búsqueda, Leela Chess Zero se basa en árboles de búsqueda y en el aprendizaje profundo (deep learning) con aprendizaje por refuerzo (reinforcement learning) para jugar al ajedrez.

Leela Chess Zero busca en un árbol de movimientos y estados de juego. Cada estado de juego es un nodo en el árbol, con un $VALUE$ que es la probabilidad de victoria desde la perspectiva del jugador actual y una lista priorizada de las probabilidades de un movimiento sobre todos los movimientos legales disponibles, llamadas $POLICY$. A diferencia de los motores tradicionales, Leela Chess Zero utiliza su red neuronal para la generación del $VALUE$ como de las $POLICIES$. Luego, Leela Chess Zero expande el árbol para obtener una mejor comprensión del nodo raíz, que es la posición actual.

Leela Chess Zero usa PUCT (Predictor + Upper Confidence Bound tree search) [12]. Evalúa nuevos nodos realizando un recorrido, comienza desde el nodo raíz (la posición actual), se elige un movimiento para explorar y repetimos por los distintos nodos del árbol hasta llegar a una posición de juego que aún no se ha examinado (o una posición que termina el juego, llamada nodo terminal).

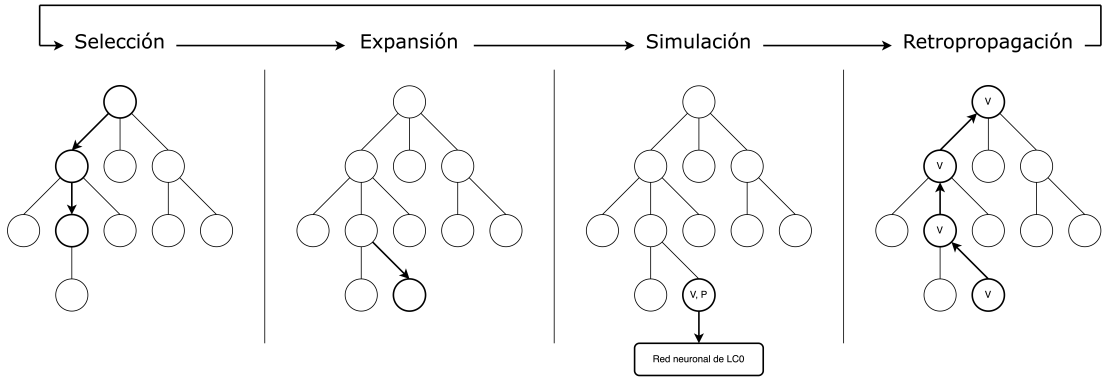

Fig. 3.1: MCTS + PUCT

Se expande el árbol con esa nueva posición (suponiendo que no sea un nodo terminal) y se usa la red neuronal para crear una primera estimación del $VALUE$ para la posición y la $POLICY$ de movimientos continuos. En Leela Chess Zero, una $POLICY$ para un nodo es una lista de movimientos y una probabilidad para cada movimiento. Después de agregar este nodo al árbol, se respalda ese nuevo $VALUE$ a todos los nodos visitados durante este recorrido. Esto mejora lentamente la estimación del $VALUE$ de diferentes rutas a través del árbol de juego.

Cuando se juega un movimiento en el tablero, el movimiento elegido se convierte en la nueva raíz del árbol. La antigua raíz y los otros hijos de ese nodo raíz se eliminan.

3.1.2. Topología de la red neuronal

La entrada a la red neuronal son 112 planos de (8×8) cada uno, correspondiente a las 64 casillas del tablero de ajedrez. Cada plano codifica una característica específica del estado actual del juego, permitiendo a la red capturar información detallada y estructurada sobre la posición, siendo esta una representación multidimensional del estado del tablero.

La red consiste en un "body" (torre residual) y varias "heads" de salida adjuntas a él.

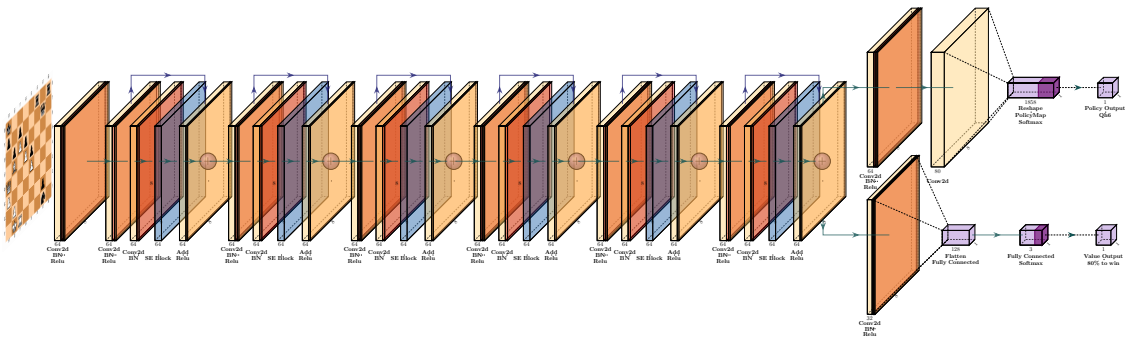

Fig. 3.2: Arquitectura de la solución

Inputs

Los 112 planos de entrada se dividen en dos grupos:

- Historial de tableros:
  Estado de las 8 posiciones más recientes (la actual + las 7 jugadas anteriores), para cada una de las 8 posiciones tengo 13 planos con la siguiente información:
  - 6 planos con las piezas del bando al que le toca mover (P, N, B, R, Q, K).
  - 6 planos con las piezas del rival en esa misma posición (P, N, B, R, Q, K).
  - 1 plano de repetición (el valor será 1 si la posición ya apareció antes, 0 en otro caso).
- Metadata de la partida:
  Son 8 planos con información que no depende del tablero en sí, pero es necesaria para valorar la posición. Y estos son:
  - Un plano con el Side-to-move (1=blancas, 0=negras).
  - Cuatro planos con los derechos de enroque (blancas O-O, blancas O-O-O, negras O-O, negras O-O-O).
  - Un plano con el contador de la regla de los 50 movimientos.(valor normalizado entre 0 y 1).
  - Un plano con el número de jugadas transcurridas (full-move count, normalizado)
  - Un plano de reserva.

Body

El body es una torre residual con Squeeze and Excitation (SE) [13] layers. El número de *blocks* residuales y $filters$ (channels) por bloque varía entre las redes. Los valores típicos para ($blocks \times filters$) son (10×128), (20×256), (24×320). Las SE layers tienen $se\_channels$ canales (típicamente alrededor de 32).

La red comienza con una convolución al input que transforma los datos de ($112 \times 8 \times 8$) a ($filters \times 8 \times 8$). A continuación, se pasa a una torre residual compuesta por múltiples bloques, donde cada bloque incluye dos convolution layers que mantienen la dimensionalidad en ($filters \times 8 \times 8$). También se incluye una SE layer después de estas convoluciones. Esta SE layer está compuesta por una global average pooling layer que reduce los datos de ($filters \times 8 \times 8$) a $filters$, seguido de una fully connected layer que transforma $filters$ a $se\_channels$, una activación ReLU. Luego, otra Fully connected layer lleva los datos de $se\_channels$ a ($2 \times filters$), que se divide en dos vectores de tamaño $filters$, W y B. Se aplica la función Sigmoid a W para obtener Z, y la salida de la SE layer se calcula como $(Z \times input) + B$.

Finalmente, se añade la conexión residual de la torre y se aplica la función de activación ReLU. Todas las convoluciones utilizan un tamaño de kernel de (3×3) y un stride de 1. Cabe destacar que la normalización por batch (batch normalization) ya está incorporada en los pesos, por lo que no es necesario realizar ninguna normalización adicional durante la inferencia.

Policy head

- $POLICY\_CONVOLUTION$
  La red ejecuta una convolución que transforma los datos de $(filters \times 8 \times 8)$ a $(filters \times 8 \times 8)$, seguida de otra convolución que los convierte en una matriz de $(80 \times 8 \times 8)$. A partir de esta matriz, se extrae un vector de longitud 1858 y como loss function utiliza *Cross Entropy*. Cabe destacar que no se aplica ninguna función de activación en la salida. Esta fue la policy head utilizada en la investigación.

- $POLICY\_CLASSICAL$
  Se ejecuta una convolución que transforma los datos de $(filters \times 8 \times 8)$ a $(policy\_conv\_size \times 8 \times 8)$, donde *policy_conv_size* es un parámetro. Luego, se pasa a una fully connected layer que convierte $(policy\_conv\_size \times 8 \times 8)$ en un vector de longitud 1858 y como loss function también utiliza *Cross Entropy*. Es importante señalar que tampoco se aplica ninguna función de activación en la salida.

Value head

Está compuesta por una parte común en la que se realiza una convolución que transforma los datos de $(filters \times 8 \times 8)$ a $(32 \times 8 \times 8)$, seguida de otra convolución que convierte los datos de $(32 \times 8 \times 8)$ en un vector de longitud 128. Luego, se aplica la función de activación ReLU.

- $VALUE\_WDL$
  Se realiza una fully connected layer que transforma un vector de longitud 128 en un vector de longitud 3. A continuación, se aplica la función de activación Softmax y como loss function utiliza *Cross Entropy*. Esta fue la value head utilizada en la investigación.

- $VALUE\_CLASSICAL$
  Se realiza una fully connected layer que transforma un vector de longitud 128 en un escalar. A continuación, se aplica la función de activación Tanh y como loss function utiliza MSE.

Moves left head

Se realiza una convolución que transforma los datos de $(filters \times 8 \times 8)$ a $(mlh\_channels \times 8 \times 8)$, donde $mlh\_channels$ es un parámetro. Luego, se pasa a una fully connected layer que convierte $(mlh\_channels \times 8 \times 8)$ en un vector de tamaño fc_size, que es otro parámetro, seguido de la aplicación de la función de activación ReLU. A continuación, otra fully connected layer transforma el vector de tamaño fc_size en un escalar, y se aplica nuevamente la función de activación ReLU, y como loss function utiliza MSE.

Total Loss Function

la función de pérdida total se define como la suma ponderada de las tres pérdidas parciales de cada head:

$$L_{total} = w_{policy}L_{policy} + w_{value}L_{value} + w_{moves\text{-}left}L_{moves\text{-}left} + \lambda \tag{3.1}$$

Donde λ es el peso del término de regularización $L2$ (reg_term_weight), que es utilizado para evitar que los pesos crezcan en exceso y reducir el overfitting.

Outputs

- Policy head: un vector de longitud 1858. Que representa la probabilidad estimada de cada movimiento legal.
- Value head: un escalar o un vector de longitud 3 (W-D-L). Que representa la probabilidad de victoria/empate/derrota desde la perspectiva del jugador activo.
- Moves left head: un escalar. Que representa la estimación del número de movimientos que faltan hasta el final de la partida

3.1.3. Lela Chess Zero Training

lczero-training es la parte del proyecto Leela Chess Zero que se enfoca exclusivamente en el entrenamiento de la red neuronal del motor de ajedrez. Este subproyecto maneja los aspectos técnicos y logísticos del proceso de entrenamiento. Es un conjunto de scripts, herramientas y un flujo de trabajo automatizado diseñado para gestionar la formación de nuevas redes neuronales que dirigen el motor Leela Chess Zero, este proyecto está hecho en $Python$ basados principalmente en las bibliotecas $Numpy$ y $Tensorflow$.

Los datos de entrada de la red son archivos chunks, que son fragmentos de datos estructurados de manera que la red neuronal pueda usarlos eficazmente, estos contienen representaciones de las posiciones en el tablero, los movimientos realizados, y otros aspectos importantes que la red necesita para aprender a jugar.

Cada chunk es un pequeño conjunto de datos que representa un momento o una secuencia de la partida que la red neuronal utiliza para aprender. Al usar estos chunks, la red neuronal puede procesar la información en lotes, lo que optimiza el entrenamiento. Un ejemplo de la estructura que pueden tener los chunks es la siguiente:

```
struct V6TrainingData {
  uint32_t version;
  uint32_t input_format;
  float probabilities[1858];
  uint64_t planes[104];
  uint8_t castling_us_ooo;
  uint8_t castling_us_oo;
  uint8_t castling_them_ooo;
  uint8_t castling_them_oo;
  uint8_t side_to_move_or_enpassant;
  uint8_t rule50_count;
  uint8_t invariance_info;
  uint8_t dummy;
  float root_q;
  float best_q;
  float root_d;
  float best_d;
  float root_m;
  float best_m;
  float plies_left;
  float result_q;
  float result_d;
  float played_q;
  float played_d;
  float played_m;
  float orig_q;
```

```
  float orig_d;
  float orig_m;
  uint32_t visits;
  uint16_t played_idx;
  uint16_t best_idx;
  float policy_kld;
  uint32_t reserved;
} PACKED_STRUCT;
```

3.1.4. Training data tool

El proyecto Trainingdata-Tool para Lela Chess Zero realiza una tarea esencial en el proceso de entrenamiento de la red neuronal. Toma datos en formato PGN, luego procesa estos archivos PGN y extrae la información relevante de cada partida. Luego, convierte esta información en chunks, que son los datos esperados por Lela Chess Zero Training para que la red neuronal puede utilizar para aprender.

Estructura de los chunks generados:

```
struct V4TrainingData {
  uint32_t version;
  float probabilities[1858];
  uint64_t planes[104];
  uint8_t castling_us_ooo;
  uint8_t castling_us_oo;
  uint8_t castling_them_ooo;
  uint8_t castling_them_oo;
  uint8_t side_to_move;
  uint8_t rule50_count;
  uint8_t move_count;
  int8_t result;
  float root_q;
  float best_q;
  float root_d;
  float best_d;
} PACKED_STRUCT;
```

3.2. Modificaciones realizadas

Con el fin de introducir el concepto del ELO se efectuaron modificaciones en la topología de la red neuronal, en Leela Chess Zero Engine, Leela Chess Zero Training y Training data tool.

3.2.1. Training data tool

Para integrar el concepto de ELO en el modelo, fue necesario modificar el funcionamiento de la herramienta encargada de procesar los datos crudos (raw data). Estas modificaciones permitieron extraer el ELO de las partidas y almacenarlo en la estructura de datos utilizada para el entrenamiento de la red neuronal. Se actualizó V4TrainingData para incluir un campo adicional destinado al ELO, quedando definida de la siguiente manera:

```
struct V4TrainingData {
  uint32_t version;
  float probabilities[1858];
  uint64_t planes[104];
  uint8_t castling_us_ooo;
  uint8_t castling_us_oo;
  uint8_t castling_them_ooo;
```

```
  uint8_t castling_them_oo;
  uint8_t side_to_move;
  uint8_t rule50_count;
  uint8_t move_count;
  int8_t result;
  float root_q;
  float best_q;
  float root_d;
  float best_d;
  float elo;
} PACKED_STRUCT;
```

La incorporación del campo *float elo* permite que cada chunk de datos incluya directamente el nivel de habilidad del jugador en el momento de la partida. Este cambio es esencial para habilitar la capacidad del modelo de ajustar su predicción según el nivel de ELO.

3.2.2. Topología de la red neuronal

El objetivo principal de esta modificación fue introducir el concepto de ELO en la red neuronal, permitiendo que el modelo adapte sus predicciones en función del nivel de habilidad del jugador. Tras analizar la estructura existente, se decidió incorporar un nuevo plano en los parámetros de entrada.

La red original estaba configurada para recibir 112 planos de dimensiones (8×8), que representan distintas características del estado del tablero. Con esta modificación, se añadió un plano adicional, incrementando el total a 113 planos de dimensiones (8×8). Este nuevo plano contiene el valor del ELO, repetido uniformemente en todas sus posiciones, asegurando que esta información esté disponible en cada punto del tablero, se eligió este diseño por las siguientes ventajas:

- Integración directa del ELO en la red:
  Al incluir el ELO como un plano adicional, se preserva la estructura de entrada esperada por la red, minimizando cambios en las etapas posteriores de procesamiento.
- Compatibilidad con las características espaciales:
  Al mantener las dimensiones 8×8, el plano del ELO se alinea con los demás planos de entrada, permitiendo que la red procese esta información en conjunto con las representaciones del tablero.
- Compatibilidad con la integración de futuras características:
  Este enfoque es fácilmente adaptable para incluir otras características adicionales en el futuro, manteniendo la consistencia estructural de la red.

3.2.3. Leela Chess Zero Training

Para integrar el ELO en el modelo, se realizaron varios ajustes en el código de entrenamiento. Estas modificaciones abarcaron tres aspectos principales:

- Adaptación de la arquitectura de la red:
  Se actualizaron los procesos de generación de la red neuronal para incorporar los cambios previamente realizados en la topología. Esto incluyó la configuración de la capa adicional que introduce el valor del ELO como un plano (8×8), permitiendo que la red procese esta información junto con las características del estado del tablero.

- Lectura de la nueva estructura de datos:
  Se implementó código adicional para leer y procesar la nueva estructura de datos generada, que ahora incluye el campo elo. Esta actualización asegura que los chunks utilizados en el entrenamiento sean compatibles con la arquitectura modificada, permitiendo que el ELO sea considerado como una entrada.

- Preprocesamiento del ELO:
  Tras varios experimentos, se determinó que el valor raw del ELO tenía un impacto desproporcionado en el modelo, dificultando su capacidad para aprender patrones de juego correctamente. Para mitigar este problema, se implementó una normalización del ELO dividiendo su valor entre 4000. Esta transformación ajusta el rango del ELO a una escala más manejable, reduciendo su influencia excesiva y facilitando un entrenamiento más estable y eficiente.

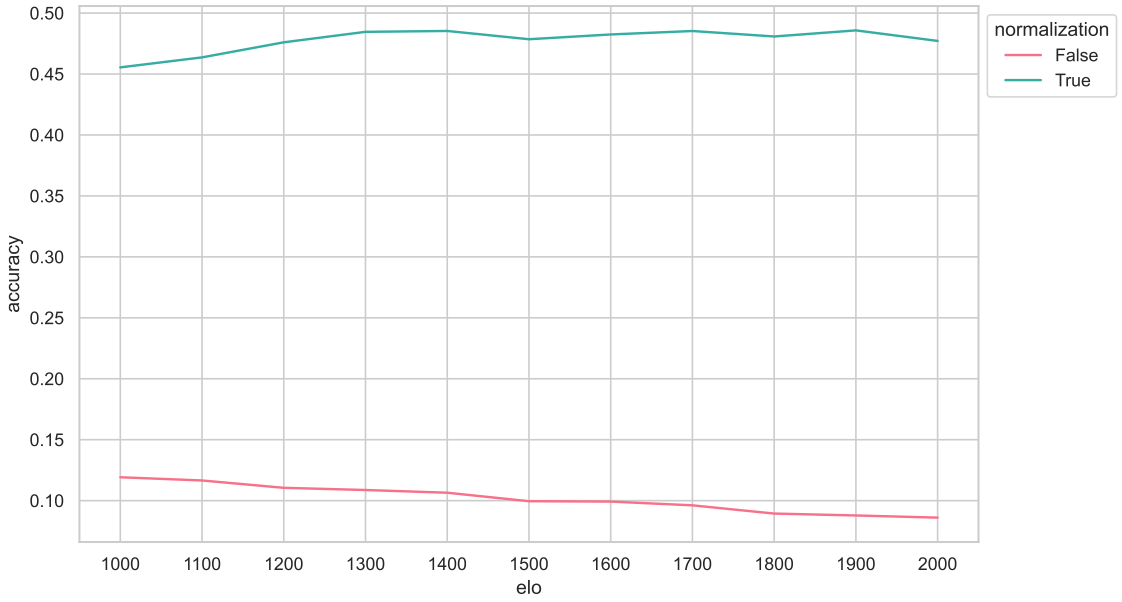

Fig. 3.3: Resultados de un modelo con normalización contra uno sin normalización

Estas modificaciones no solo garantizaron la compatibilidad del modelo con los datos actualizados, sino que también mejoraron su capacidad de aprendizaje, al permitir una integración balanceada del ELO como característica clave.

3.2.4. Leela Chess Zero Engine

Se realizaron modificaciones en el engine para adaptarlo a las nuevas necesidades del modelo, integrando el ELO como un parámetro que permite ajustar las predicciones de la red neuronal en función del nivel de habilidad del jugador. Los cambios implementados incluyen:

- Recepción del ELO como parámetro de programa:
  El engine fue actualizado para aceptar el valor de ELO como un parámetro proporcionado al momento de su ejecución. Este parámetro indica el nivel de habilidad esperado y permite personalizar las predicciones efectuadas por el modelo.

- Almacenamiento interno del ELO:
  Una vez recibido, el valor del ELO es almacenado internamente en el engine. Este almacenamiento asegura que el parámetro esté disponible para cada llamada posterior realizada al modelo, permitiendo que las predicciones sean coherentes y específicas al nivel de habilidad definido.

- Provisión del ELO al modelo:
  Durante cada consulta al modelo, el engine utiliza el valor almacenado de ELO para incluirlo como parte de las entradas suministradas a la red neuronal en el momento del encoding de la posición. Esto garantiza que el modelo reciba consistentemente el ELO como un plano adicional en la estructura de entrada, tal como lo requiere su nueva topología.

Estas modificaciones fueron diseñadas para mantener la eficiencia del flujo de datos dentro del engine y asegurar que el ELO sea incorporado de manera coherente en cada predicción. Este enfoque no solo adaptan el engine a la nueva topología de los modelos, sino que también simplifica la configuración del engine al centralizar el manejo del ELO como un parámetro de programa.

3.3. Arquitectura de la solución

La solución se estructura en tres etapas principales: preprocesamiento, entrenamiento y validación, que trabajan de manera integrada para desarrollar y evaluar el modelo de inteligencia artificial.

1. **Preprocesamiento**
   El proceso comienza con la descarga de los archivos Raw PGN desde la base de datos de Lichess. Estos archivos son procesados mediante el script PGN Ingestion, que se encarga de filtrar los datos y almacenarlos en una base de datos MongoDB para facilitar su manejo.
   A partir de esta base de datos, el script PGN Generator permite generar archivos PGN de entrenamiento, aplicando filtros específicos según las necesidades del modelo, como el control de tiempo o el rango de ELO.
   Posteriormente, los Training PGN son procesados con el script Chunk Generator, que utiliza una versión modificada de la herramienta trainingdata-tool para incluir el valor de ELO en los datos generados. Esto da lugar a los Training Chunks, estructuras optimizadas que se utilizan directamente en la etapa de entrenamiento.

2. **Entrenamiento**
   En esta etapa, se emplea el módulo LC0 Training, modificado para adaptarse a la nueva topología de la red neuronal. Este componente procesa los Training Chunks y entrena el modelo de inteligencia artificial, ajustando sus pesos para predecir movimientos humanos de manera precisa y adaptativa al nivel de habilidad representado por el ELO.

3. **Validación**
   La validación del modelo se realiza utilizando el script Validation CSV Generator, que extrae partidas directamente desde la base de datos MongoDB y las procesa con el LC0 Engine, también modificado para trabajar con la nueva arquitectura. Este

proceso genera un archivo CSV que contiene los resultados obtenidos por el modelo. Finalmente, el script Metrics Generator toma este CSV y calcula métricas clave, como la exactitud y F1-score, para evaluar el desempeño del modelo y su alineación con los movimientos humanos.

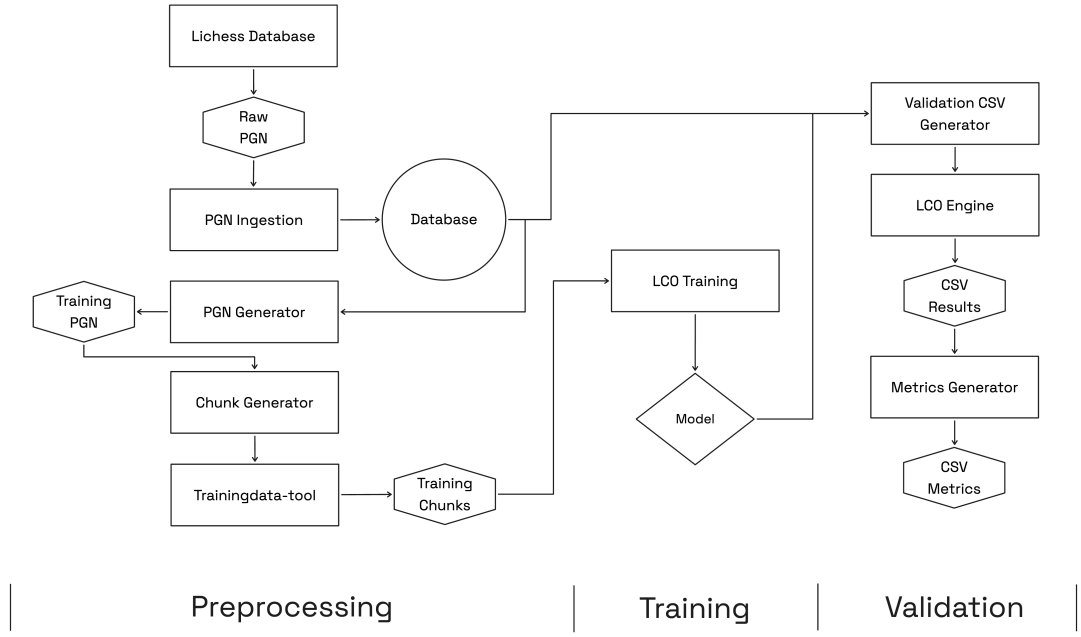

Fig. 3.4: Arquitectura de la solución

3.4. Manipulación y almacenado de los datos

Dada la magnitud de los datos involucrados en esta investigación, fue necesario implementar herramientas específicas que permitieran automatizar y estandarizar su procesamiento. Para ello, se desarrollaron una serie de scripts diseñados para manejar grandes volúmenes de datos, automatizando tareas como la ingestión, filtrado y preparación de los mismos. Estas herramientas no solo facilitaron la manipulación de los datos, sino que también garantizaron la reproducibilidad de los experimentos realizados.

Para el almacenamiento, se optó por utilizar una base de datos MongoDB, que ofreció la flexibilidad necesaria para manejar estructuras de datos complejas y permitió un acceso rápido y eficiente a la información durante todas las etapas del desarrollo y entrenamiento del modelo. Este enfoque permitió trabajar con los datos de manera escalable y organizada, asegurando la integridad y consistencia de los mismos a lo largo del estudio.

3.4.1. PGN Ingestion

Este script se encarga de la ingestión de las partidas de ajedrez a la base de datos, el input son lotes de archivos PGN raw en los cuales puede haber cualquier cantidad de partidas de ajedrez.

Input:

- connection_string: Connection string de la base de datos MongoDB en la cual se quieren almacenar las partidas.
- database: Nombre de la base de datos en la cual se quieren almacenar las partidas.
- collection: Nombre de la collection en la base de datos en la cual se quieren almacenar las partidas.
- path: Dirección del directorio donde se encuentran los archivos pgn que se desean almacenar.

Este script también se encargaba de filtrar las partidas que no eran relevantes para la investigación. Los criterios de filtrado fueron los siguientes:

- Partidas sin ELO: Se filtraron todas las partidas en la que no estuviera el ELO de alguno de los jugadores.
- Partidas con Bots: Se filtraron todas las partidas en la que el título de cualquiera de los jugadores fuera "BOT" que corresponde con los jugadores no humanos.
- Partidas sin tiempo: Se filtraron todas las partidas en la que no estuviera el tiempo en las jugadas.

3.4.2. PGN Generator

Este script se encarga de la generación de archivos PGN a partir de partidas en la base de datos, de esta manera se puede generar subconjuntos de partidas basándose en las propiedades de estas, que posteriormente se utilizaran para generar los archivos finales con los cuales se entrenara la red neuronal.

Input:

- connection_string: Connection string de la base de datos MongoDB de la cual se quieren extraer las partidas.
- database: Nombre de la base de datos de la cual se quieren extraer las partidas.
- collection: Nombre de la collection en la base de datos de la cual se quieren extraer las partidas.
- path: Dirección del directorio donde se va a generar el archivo de salida.
- filter: Condición que deben cumplir las partidas para estar en el archivo generado.
- skip: Cantidad de partidas que cumplan con la condición que se quieren saltear.
- limit: Cantidad de partidas que cumplan con la condición que se guardaran el archivo de salida.

3.4.3. Chunk Generator

Este script se encarga de la generación de los archivos chunks, que son la entrada para el entrenamiento de la red neuronal, a partir de un archivo PGN, normalmente la salida del PGN Generator.

Input:

- pgn_path: Dirección del PGN que se utilizara para generar los chunks.
- chunk_path: Dirección del directorio donde se van a generar los archivo de salida.
- test_size: Porcentaje de los chunks que se utilizaran para training y test.

El resultado de este script es la generación de 2 directorios (train y test) dentro de la dirección elegida como output con los chunks generados en la proporción seleccionada. Para la generación de chunks se utilizó el proyecto trainingdata-tool al cual se le realizaron las modificaciones necesarias para adaptarlo a este trabajo.

3.4.4. Validation CSV Generator

Este script se encarga de la validación de los modelos generados en los entrenamientos. Para hacer esto, se conecta con la base de datos para obtener las partidas que serán utilizadas. Luego utilizando la librería *chess* de Python sé instancia engine usando el engine Lela Chess Zero al cual se le realizaron modificaciones y a este engine se le pasa como parámetro los pesos de la red neuronal obtenidos en el entrenamiento. Por último se procede a hacer jugar al engine para obtener los movimientos, esto se almacena con más información de las partidas para posteriormente poder hacer análisis sobre estos.

Input:

- connection_string: Connection string de la base de datos MongoDB de la cual se quieren extraer las partidas.
- database: Nombre de la base de datos de la cual se quieren extraer las partidas.
- collection: Nombre de la collection en la base de datos de la cual se quieren extraer las partidas.
- path: Dirección del directorio donde se va a generar el archivo de salida.
- engine: Dirección del binario que se utilizara como engine.
- weights: Dirección del archivo en el que se encuentran los pesos de la red neuronal.
- no_elo: Flag utilizado para no utilizar los cambios realizados en esta investigación y funcionar como lo hacía originalmente.
- static_elo: Flag utilizado para utilizar un ELO fijo durante toda la evaluación independientemente del ELO de los jugadores de la partida, si este flag está activado el parámetro ELO es requerido, si este flag no está activo entonces el parámetro ELO será ignorado y se utilizara el ELO del jugador que va a realizar el movimiento.

- elo: ELO utilizado en el caso de que el flag static elo estuviera activo.
- filter: Condición que deben cumplir las partidas para utilizarse en la validación.
- skip: Cantidad de partidas que cumplan con la condición que se quieren saltear.
- limit: Cantidad de partidas que cumplan con la condición que se utilizaran.

Output:

- fen: La posición del tablero a predecir en formato FEN.
- white_elo: ELO del jugador con las piezas blancas.
- black_elo: ELO del jugador con las piezas negras.
- real: Movimiento realizado por el jugador activo en la partida real.
- predicted: Movimiento predicho por el modelo sobre esa partida.
- turn: Flag utilizado para indicar quien es el jugador activo.
- clock: Tiempo restante del jugador activo.
- control_time: Control de tiempo utilizado en la partida.
- eval: Evaluación de la partida en ese momento, si la partida contaba con una.
- is_end: Flag utilizado para indicar si la partida finalizo.

3.4.5. Metrics Generator

Este script se encarga de la generación de métricas a través del procesamiento de los resultados de la validación realizada con el script Validation CSV Generator, para poder extraer información y posteriormente generar los gráficos y reportes para este trabajo.

Input:

- validation_path: Dirección del directorio en el que se encuentran almacenado los archivos CSV que se utilizaran para generar las métricas.
- output_path: Dirección del directorio que se utilizara para almacenar los resultados.
- file_names: Nombre de los archivos que se tomaran en cuenta para la generación de las métricas.

Output:

- accuracy: Accuracy obtenida para esa partida.
- f1_weighted: F1 weighted obtenida para esa partida.
- elo: ELO promedio de los jugadores de la partida.
- model: Nombre del file_name.

4. EXPERIMENTOS Y RESULTADOS

Esta sección presenta los experimentos realizados para evaluar el desempeño del modelo desarrollado y los resultados obtenidos en cada uno de ellos. El objetivo principal del trabajo es desarrollar un modelo de inteligencia artificial para ajedrez capaz de predecir movimientos humanos adaptándose al nivel de ELO, y compararlo con los modelos de Maia Chess como referencia. Para ello, se utilizaron partidas con un control de tiempo de 10 minutos, seleccionadas por su equilibrio entre calidad y volumen de datos disponibles.

El proceso de entrenamiento se dividió en una serie de experimentos diseñados para evaluar el impacto de diferentes configuraciones del modelo y parámetros de entrenamiento. Cada experimento buscó ajustar y optimizar aspectos clave del modelo, como la incorporación del ELO, el tamaño de los datos de entrenamiento y los hiperparámetros empleados, asegurando un enfoque sistemático y riguroso en la validación de resultados.

A continuación, se detallan las configuraciones utilizadas y los resultados obtenidos en cada experimento.

4.1. Move Matching Accuracy

Exactitud de coincidencia de movimientos (Move Matching Accuracy) es la métrica de evaluación que usaremos para medir que tan humanos son los movimientos generados por el modelo. Es una métrica muy empleada en trabajos existentes y es la que se utilizara a lo largo de todo el trabajo.

Esta métrica mide la frecuencia con la que el movimiento predicho por el modelo coincide exactamente con el movimiento que realizó un jugador humano en una posición dada. Para calcularla, se toman posiciones reales jugadas por humanos, se predice el movimiento con el modelo, y se verifica si la predicción coincide con el movimiento real. La exactitud se expresa como el porcentaje de aciertos sobre el total de intentos.

Este enfoque permite evaluar qué tan bien el modelo replica el comportamiento humano desde la imitación del estilo de juego, incluso si este incluye errores o jugadas subóptimas.

4.2. Hiperparametros

Los siguientes hiperparámetros fueron configurados para definir la arquitectura y el entrenamiento del modelo en este trabajo. Estos valores reflejan decisiones importantes sobre el diseño de la red neuronal, la preparación de los datos y las estrategias de entrenamiento.

1. Configuración del Entrenamiento

    - **batch_size:** Especifica el número de ejemplos que se procesan en cada iteración de entrenamiento.
    - **num_batch_splits:** Divide el batch en múltiples partes para procesar de manera paralela.
    - **test_steps:** Indica la frecuencia con la que se evalúa el conjunto de prueba durante el entrenamiento.

- **train_avg_report_steps:** Especifica cada cuántos pasos se generan informes del promedio de los valores del entrenamiento.
- **total_steps:** Define el número total de pasos de entrenamiento, lo que marca la duración del proceso.
- **checkpoint_steps:** Permite guardar puntos de control del modelo, facilitando la recuperación y análisis.
- **shuffle_size:** Define el tamaño del búfer de mezcla, asegurando que los datos se utilicen en un orden aleatorio para mejorar el entrenamiento.

2. Estrategia de Aprendizaje

- **lr_values:** Lista de tasas de aprendizaje utilizadas durante el entrenamiento, permitiendo ajustes más precisos en etapas avanzadas.
- **lr_boundaries:** Define los pasos en los que se reduce la tasa de aprendizaje, ajustando dinámicamente el ritmo de aprendizaje.
- **policy_loss_weight:** Pesos asignados a la función de pérdida de policy del modelo.
- **value_loss_weight:** Pesos asignados a la función de pérdida de value del modelo.
- **moves_left_loss_weight:** Pesos asignados a la función de pérdida de moves left del modelo.
- **reg_term_weight:** Peso que se asigna al término de regularización en la función de pérdida total del modelo.

3. Arquitectura del Modelo

- **filters:** Establece el número de filtros, determinando la capacidad del modelo para capturar características.
- **residual_blocks:** Indica el número de bloques, que profundizan la red y mejoran su capacidad de representación.
- **se_ratio:** Es la relación de compresión en la arquitectura de Squeeze-Excite, que permite que la red ajuste dinámicamente la importancia de los canales.
- **policy:** Define el enfoque de la head policy, que predice los movimientos más probables.
- **pol_embedding_size, pol_encoder_layers, pol_encoder_heads, pol_encoder_d_model, pol_encoder_dff:** Configuran la estructura interna de la head policy, definiendo el tamaño de los embeddings, el número de capas, las attention head y los parámetros de los vectores en la atención.
- **policy_d_model:** Peso que se asigna al término de regularización en la función de pérdida total del modelo.
- **value:** Configura la salida de value para predecir el resultado del juego.
- **moves_left:** Ajusta la salida del modelo para estimar los movimientos restantes.

4.3. Experimento 0

El objetivo de este primer experimento fue obtener un modelo base funcional que cumpliera con los objetivos propuestos en este trabajo. Para ello, se procesaron y filtraron partidas de ajedrez, preparándolas como datos de entrada para el entrenamiento del modelo.

4.3.1. Preprocesamiento

- **Carga inicial**:
  Se analizaron un total de $611,005,007$ partidas de ajedrez, correspondientes a los meses de enero a junio de 2023. Estas partidas fueron cargadas en una base de datos MongoDB, aplicando un primer filtro para seleccionar únicamente aquellas que incluían información completa sobre el tiempo de juego, el ELO de los jugadores y las evaluaciones generadas por un motor. Tras este filtro, se obtuvieron $59,285,255$ partidas.

- **Filtrado por control de tiempo**:
  Las partidas fueron refinadas aún más para incluir únicamente aquellas con un control de tiempo de 10 minutos (600 segundos). De estas, se separaron $120,000$ partidas para el conjunto de validación y $10,706,873$ partidas para el conjunto de entrenamiento.

- **Generación de chunks**:
  Utilizando las partidas seleccionadas para entrenamiento, se generaron los chunks necesarios para el entrenamiento del modelo. Este proceso produjo un total de $124,102$ chunks de entrenamiento y $31,025$ chunks de prueba.

4.3.2. Hiperparametros

Para este experimento, la configuración de hiperparámetros fue seleccionada cuidadosamente con el objetivo de evitar una red neuronal demasiado grande, asegurando así que el entrenamiento pudiera completarse de manera factible en una primera etapa. Esta decisión permitió optimizar el uso de recursos computacionales y facilitar el ajuste inicial del modelo. Estos fueron los siguientes:

```
name: 'model0-64x6'
gpu: 0
dataset:
    ...
training:
    batch_size: 128
    num_batch_splits: 1
    test_steps: 2000
    train_avg_report_steps: 50
    total_steps: 400000
    checkpoint_steps: 10000
    shuffle_size: 250000
    lr_values:
        - 0.1
        - 0.01
        - 0.001
        - 0.0001
    lr_boundaries:
        - 80000
```

```
        - 200000
        - 360000
    policy_loss_weight: 1.0
    value_loss_weight: 1.0
    moves_left_loss_weight: 1.0
model:
  filters: 64
  residual_blocks: 6
  se_ratio: 8
  policy: 'convolution'
  pol_embedding_size: 64
  pol_encoder_layers: 1
  pol_encoder_heads: 4
  pol_encoder_d_model: 64
  pol_encoder_dff: 128
  policy_d_model: 64
  value: 'wdl'
  moves_left: 'v1'
```

4.3.3. Resultados

Los resultados obtenidos fueron los siguientes:

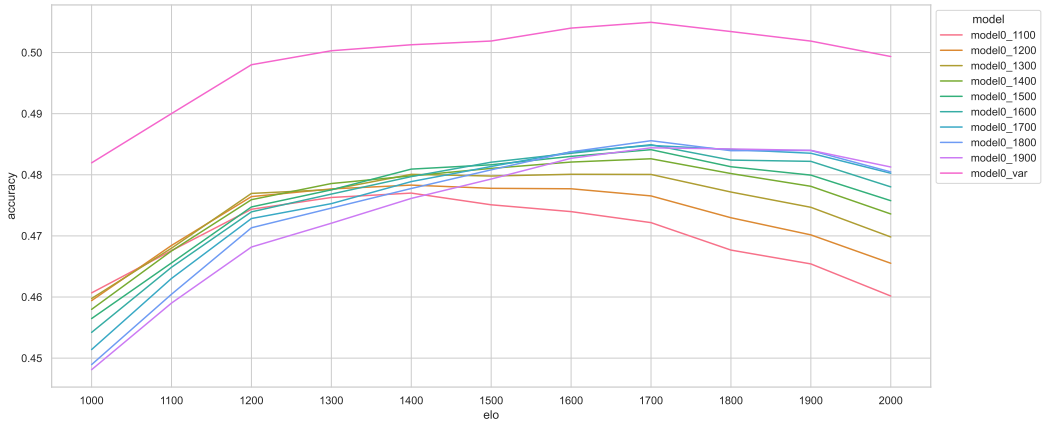

Fig. 4.1: Resultados obtenidos del experimento 0

El gráfico muestra la move-matching accuracy del modelo utilizando diferentes valores fijos de ELO, desde 1100 hasta 1900 y usando elo variable (el del jugador a realizar el movimiento). Los resultados evidencian una tendencia ascendente, donde el modelo configurado para niveles de ELO más altos, como 1700, 1800 y 1900, alcanzan una move-matching accuracy alrededor de 48.5 %, mientras que los para niveles más bajos, como 1100 y 1200, tienen una move-matching accuracy menor, cercana del 47.0 %, además el modelo con la configuración variable obtuvo resultados superiores a 49.0 %, obteniendo su mejor rendimiento entre 1600 y 1800. Esto sugiere que el modelo es más efectivo al predecir movimientos de jugadores con mayor habilidad, probablemente debido a que estos tienden a seguir patrones más consistentes. Por el contrario, en niveles de ELO más bajos, la variabilidad y la imprevisibilidad de los movimientos dificultan la identificación de patrones claros, lo que impacta negativamente en el modelo.

A pesar de estas diferencias, la variación en la move-matching accuracy entre los niveles más bajos y más altos de ELO no es extremadamente grande (aproximadamente 1.5

puntos), lo que indica que el modelo mantiene un desempeño razonablemente consistente en todo el rango evaluado. Esto confirma que la arquitectura desarrollada cumple con el objetivo principal del trabajo: adaptarse a diferentes niveles de habilidad para predecir movimientos humanos.

4.4. Experimento 1

Tras analizar los resultados del experimento anterior, se identificó que el filtro de partidas validadas (aquellas con evaluación incluida) limitaba significativamente el tamaño del conjunto de datos disponible para el entrenamiento, sin ser una condición estrictamente necesaria para alcanzar los objetivos del modelo. Por ello, este experimento se diseñó con el objetivo de aumentar el volumen de datos de entrenamiento, eliminando dicho filtro y considerando únicamente las partidas que contaran con información de tiempo y ELO.

4.4.1. Preprocesamiento

- **Carga inicial**:
  Se volvieron a analizar las mismas $611,005,007$ partidas de ajedrez, correspondientes a los meses de enero a junio de 2023. Estas partidas también fueron cargadas en una base de datos MongoDB, aplicando un primer filtro para seleccionar únicamente aquellas que incluían información completa sobre el tiempo de juego y el ELO de los jugadores, esta vez sin filtrar las que tuvieran evaluaciones generadas por un motor. Tras este filtro, se obtuvieron $610,818,695$ partidas.

- **Filtrado por control de tiempo**:
  Se volvieron a incluir únicamente aquellas con un control de tiempo de 10 minutos (600 segundos). De estas, se separaron nuevamente $120,000$ partidas para el conjunto de validación y 63,992,072 partidas para el conjunto de entrenamiento.

- **Generación de chunks**:
  Utilizando las partidas seleccionadas para entrenamiento, se generaron los chunks necesarios para el entrenamiento del modelo. Este proceso produjo un total de $801,369$ chunks de entrenamiento y 200,342 chunks de prueba.

4.4.2. Hiperparametros

Para garantizar una comparación justa y evaluar el impacto directo de esta modificación en los datos, se utilizó exactamente la misma configuración de hiperparámetros que en el experimento anterior. Esto permitió atribuir cualquier diferencia en los resultados exclusivamente al cambio en la preparación de los datos, asegurando un análisis más controlado y objetivo.

4.4.3. Resultados

Los resultados obtenidos fueron los siguientes:

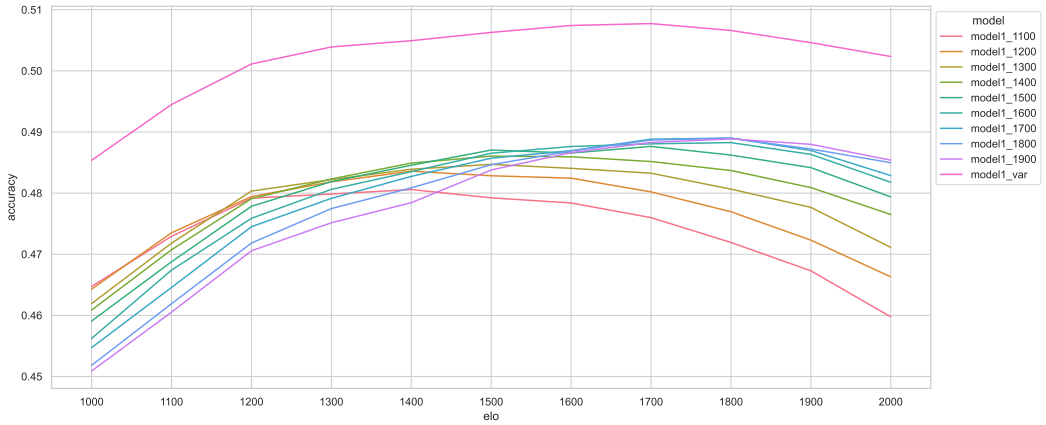

Fig. 4.2: Resultados obtenidos del experimento 1

Al comparar los resultados del primer y segundo experimento, ambos experimentos muestran una tendencia ascendente en la move-matching accuracy conforme aumenta el ELO. Pero se observa una mejora general en los resultados del modelo en todos los niveles de ELO.

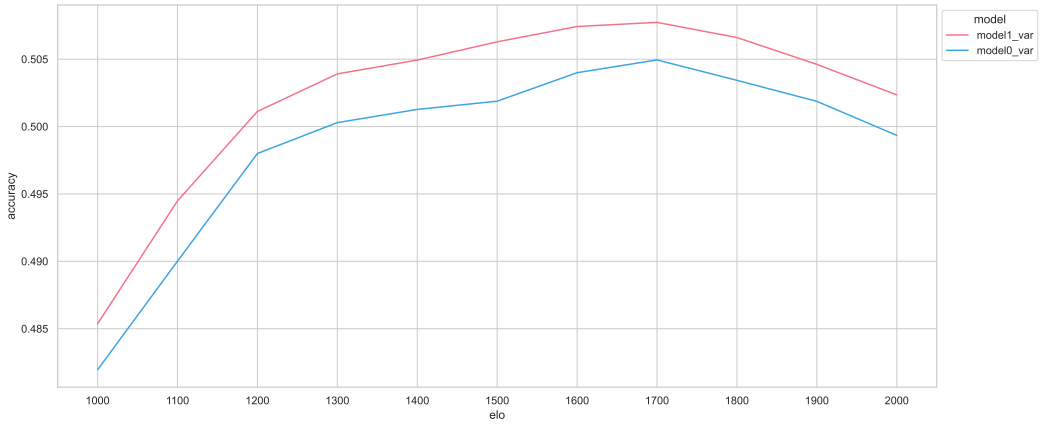

Fig. 4.3: Comparación entre el experimento 1 y el experimento 0

En conclusión, el segundo experimento logró una mejora respecto al primero, confirmando que un mayor volumen de datos puede potenciar el desempeño del modelo.

4.5. Experimento 2

Tras analizar los resultados del segundo experimento, el enfoque en este experimento fue continuar mejorando el rendimiento del modelo mediante un refinamiento adicional del conjunto de datos. En particular, se decidió eliminar las partidas con valores de ELO inferiores a 1000 y superiores a 2000, ya que estos rangos contienen significativamente

menos partidas en comparación con los niveles intermedios, lo que podría introducir un sesgo en el entrenamiento y afectar la capacidad del modelo para generalizar.

4.5.1. Preprocesamiento

- **Carga inicial**:
  Se utilizaron las mismas $611,005,007$ partidas de ajedrez, y los mismos filtros, dando por resultado las mismas $610,818,695$ partidas.

- **Filtrado por control de tiempo**:
  Se volvieron a incluir únicamente aquellas con un control de tiempo de 10 minutos (600 segundos) y las partidas con valores de ELO superiores a 1000 e inferiores a 2000. De estas, se separaron nuevamente $120,000$ partidas para el conjunto de validación y $51,269,362$ partidas para el conjunto de entrenamiento.

- **Generación de chunks**:
  Utilizando las partidas seleccionadas para entrenamiento, se generaron los chunks necesarios para el entrenamiento del modelo. Este proceso produjo un total de $644,268$ chunks de entrenamiento y $161,066$ chunks de prueba.

4.5.2. Hiperparametros

Al igual que el experimento anterior para garantizar una comparación justa y evaluar el impacto directo de esta modificación en los datos, se usó exactamente la misma configuración de hiperparámetros que en el primer experimento.

4.5.3. Resultados

Los resultados obtenidos fueron los siguientes:

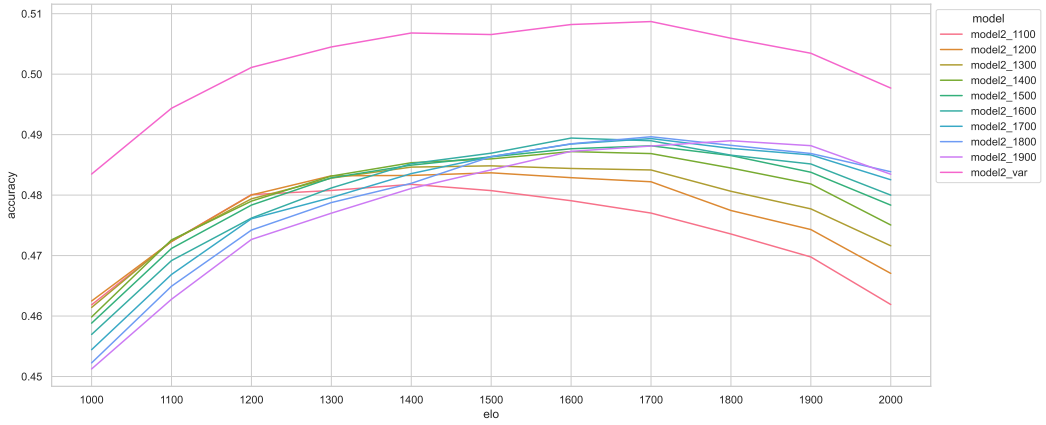

Fig. 4.4: Resultados obtenidos del experimento 2

Al analizar los resultados del experimento 2, se vuelve a observar una tendencia ascendente conforme aumenta el ELO. Sin embargo, la move-matching accuracy del modelo

mejora sutilmente en comparación con el segundo experimento, pero estas diferencias no son estadísticamente significativas ni consistentes a lo largo de todos los niveles de ELO.

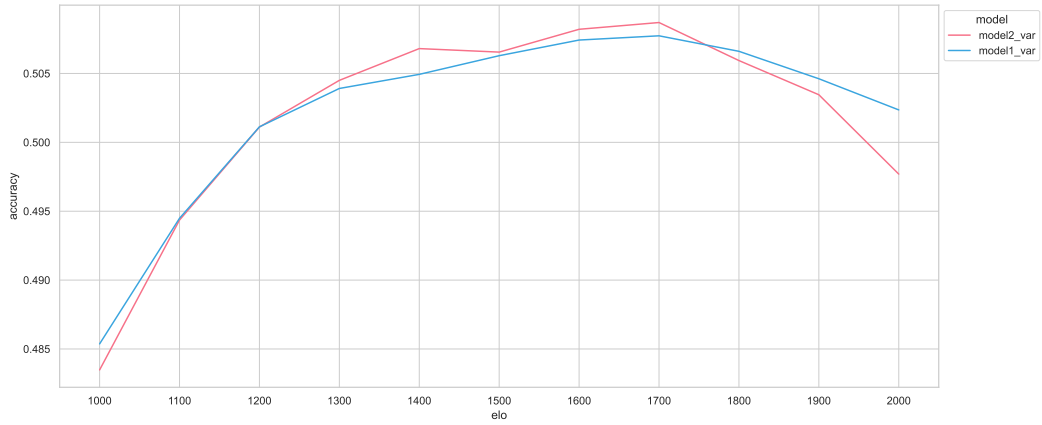

Fig. 4.5: Comparación entre el experimento 2 y el experimento 1

En términos generales, este experimento confirma que concentrar los datos en un rango más representativo no perjudicó el rendimiento del modelo, no obstante tampoco resultó en mejoras sustanciales. Esto podría deberse a que el modelo ya había alcanzado un rendimiento cercano a su límite en el experimento 1, donde el aumento del volumen de datos ya había proporcionado un beneficio significativo.

4.6. Resultados de Maia

Para evaluar el desempeño del modelo desarrollado en este trabajo, es necesario contar con un punto de referencia que permita realizar una comparación significativa. En este caso, se utilizarán los resultados de los modelos de Maia.

Los resultados de Maia ofrecen una base de comparación ideal debido a su similitud en objetivos y enfoque con el modelo propuesto, además de su capacidad para representar un estándar en el campo. A continuación, se presentan los resultados obtenidos por los modelos Maia, que servirán como referencia para analizar y contrastar los logros alcanzados en este trabajo.

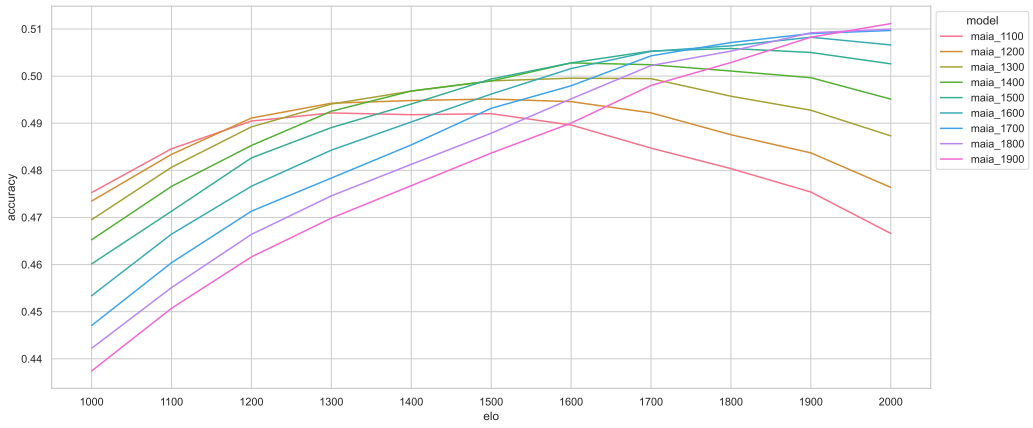

Fig. 4.6: Resultados obtenidos de los modelos de Maia

Al comparar los resultados del experimento 2 con los obtenidos por los modelos Maia, se observan diferencias en el desempeño a lo largo de los niveles de ELO evaluados. El modelo del experimento 2 muestran una move-matching accuracy máxima entre 48.5 % y 51.0 %, mientras que los modelos Maia mantienen una move-matching accuracy entre 47.5 % y 51.5 % para los diferentes niveles de ELO.

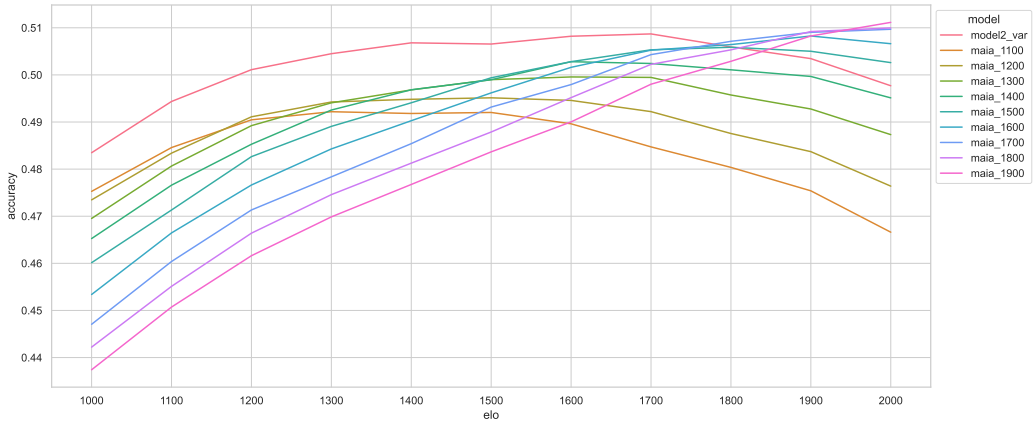

Fig. 4.7: Comparación entre el experimento 2 y los modelos de Maia

En comparación con Maia, el modelo del experimento 2 logra mejores resultados a rangos de ELO más bajos, pero se queda por debajo en los rangos de ELO más altos. Esto sugiere que, para mejorar los resultados de Maia, sería necesario explorar ajustes adicionales en la arquitectura, el diseño de los datos de entrenamiento o los hiperparámetros.

4.7. Optimización de hiperparametros

Tras comparar los resultados obtenidos con los modelos Maia, se pueden identificar varias áreas clave en las que el modelo desarrollado en este trabajo puede mejorar para

cerrar la brecha en move-matching accuracy. Una de las estrategias fundamentales para alcanzar este objetivo es la optimización de los hiperparámetros, que desempeñan un papel crucial en el rendimiento del modelo durante el entrenamiento.

En esta sección, se explorarán diferentes configuraciones de hiperparámetros con el objetivo de maximizar los resultados del modelo en todos los niveles de ELO. Para ello se realizará un análisis iterativo basado en los resultados obtenidos en cada etapa.

El propósito de esta optimización no es solo mejorar la move-matching accuracy global del modelo, sino también asegurar que pueda competir directamente con los resultados de los modelos Maia en términos de predicción de movimientos humanos adaptados al ELO. A lo largo de esta sección, se describirá el enfoque adoptado, los experimentos realizados y los resultados obtenidos tras la implementación de las configuraciones optimizadas.

La optimización se llevó a cabo tomando como base los datos del Experimento 2 y utilizando la configuración de hiperparámetros previamente establecida como punto de partida. Este enfoque permitió mantener la consistencia con los experimentos anteriores, asegurando que las mejoras observadas fueran atribuibles directamente a los ajustes efectuados durante el proceso de optimización, y no a cambios en otros aspectos del modelo o los datos.

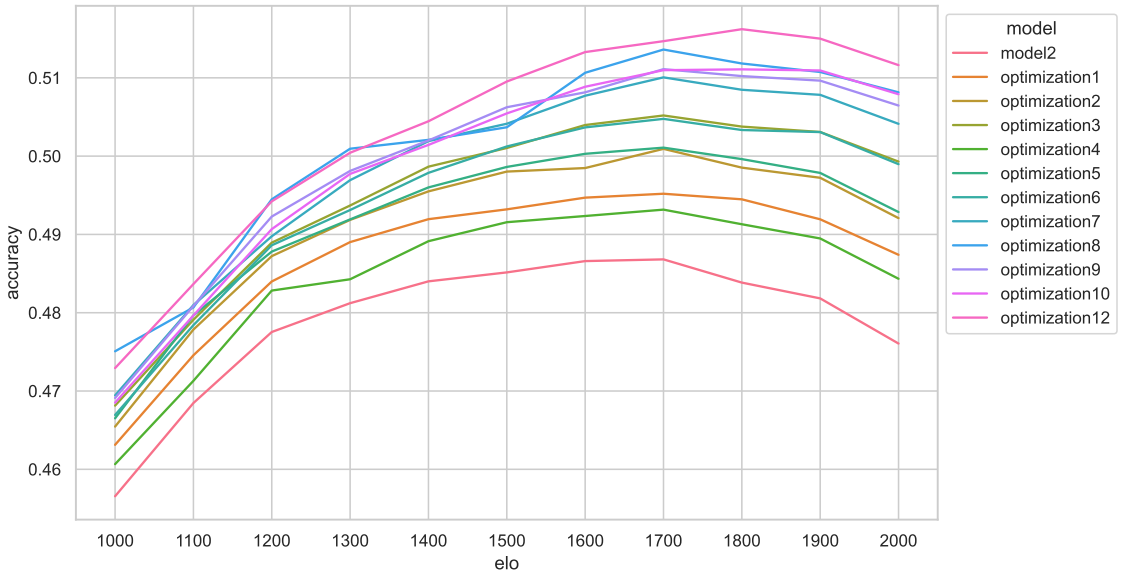

Fig. 4.8: Resultados obtenidos de la optimización de hiperparametros

Luego de múltiples experimentaciones y ajustes de hiperparámetros, se logró identificar una configuración que produjo una mejora significativa en el rendimiento del modelo. En particular, se obtuvo un aumento superior al 3 % en la move-matching accuracy (más de 6 % de mejora de los valores obtenido con el modelo antes de la optimización), consolidando así un avance notable en la capacidad predictiva del modelo.

4.8. Experimento 3

En este último experimento, tras un exhaustivo proceso de optimización de hiperparámetros, se construyó un modelo final con el propósito de obtener la mejor versión

posible. Esta configuración final representa la culminación del proceso de mejora continua, integrando los ajustes más efectivos identificados a lo largo de las experimentaciones previas para alcanzar un desempeño óptimo en distintos niveles de ELO.

4.8.1. Preprocesamiento

Para este experimento, se utilizó el mismo conjunto de datos de entrenamiento y validación generados en el Experimento 2

4.8.2. Hiperparametros

Para este experimento, la configuración de hiperparámetros se basó en los ajustes obtenidos a partir del proceso de optimización realizado previamente. Se seleccionaron las configuraciones que demostraron el mejor desempeño en los experimentos anteriores, asegurando un equilibrio entre move-matching accuracy, capacidad de generalización y eficiencia computacional. De esta manera, el modelo final integra los hiperparámetros más efectivos identificados a lo largo de las iteraciones de optimización.

```
name: 'model3-64x6'
gpu: 0
dataset:
    ...
training:
    batch_size: 192
    num_batch_splits: 1
    test_steps: 2000
    train_avg_report_steps: 50
    total_steps: 400000
    checkpoint_steps: 10000
    shuffle_size: 250000
    lr_values:
        - 0.1
        - 0.01
        - 0.001
        - 0.0001
    lr_boundaries:
        - 80000
        - 200000
        - 360000
    policy_loss_weight: 1.0
    value_loss_weight: 1.0
    moves_left_loss_weight: 1.0
model:
  filters: 128
  residual_blocks: 32
  se_ratio: 32
  policy: 'convolution'
  pol_embedding_size: 64
  pol_encoder_layers: 1
  pol_encoder_heads: 4
  pol_encoder_d_model: 64
  pol_encoder_dff: 128
  policy_d_model: 64
  value: 'wdl'
  moves_left: 'v1'
```

4.8.3. Resultados

Los resultados obtenidos fueron los siguientes:

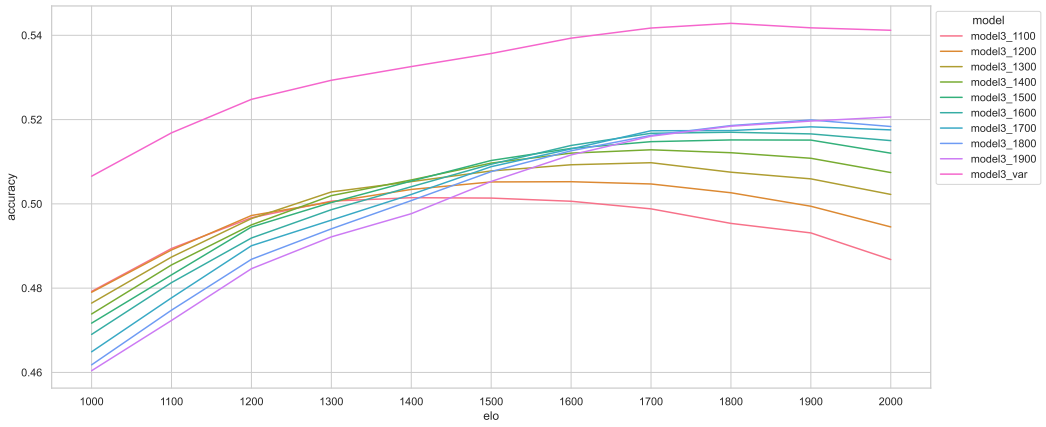

Fig. 4.9: Resultados obtenidos del experimento 3

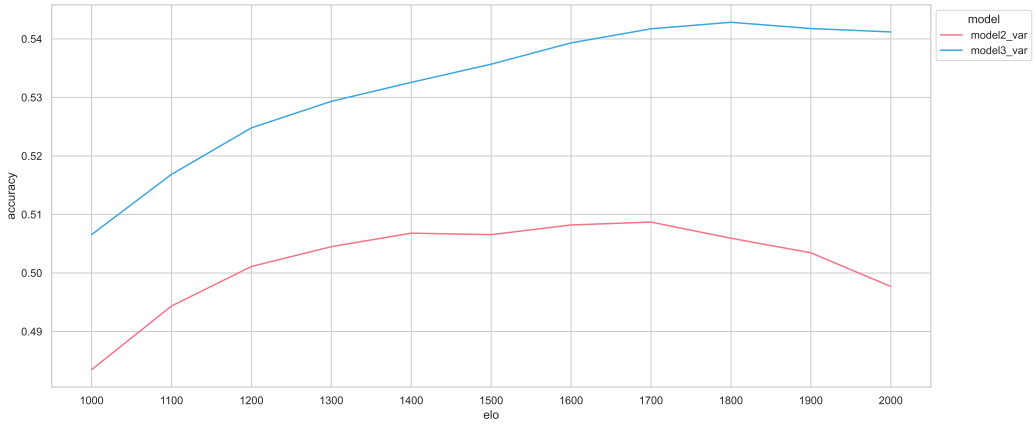

Fig. 4.10: Comparación entre el experimento 3 y el experimento 2

Al analizar los resultados del último experimento, se observa una mejora significativa en la move-matching accuracy del modelo en comparación con el experimento anterior a lo largo de todo el rango del ELO.

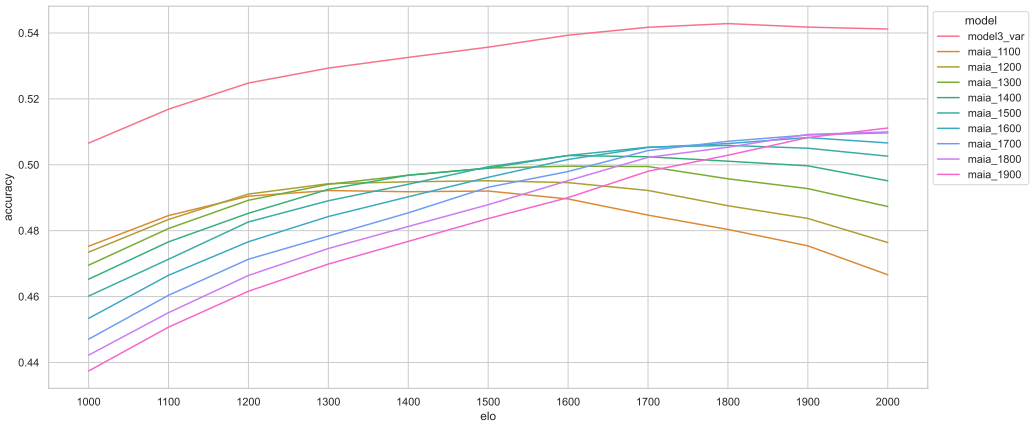

Fig. 4.11: Comparación entre el experimento 3 y los modelos de Maia

Este experimento representa la culminación del proceso de optimización de hiperparámetros investigado previamente, logrando refinar aún más el rendimiento del modelo. Como resultado, se han obtenido métricas que superan incluso a los modelos Maia, consolidando este enfoque como una solución más precisa y versátil para la predicción de movimientos humanos en ajedrez.

5. CONCLUSIONES

Este trabajo abordó la problemática de la dificultad de integración y comprensión de los modelos de inteligencia artificial en el aprendizaje humano debido a las diferencias en la forma en que los humanos y las máquinas resuelven problema a través del desarrollo de un modelo de inteligencia artificial adaptable al nivel de habilidad humana, utilizando el ELO como parámetro de referencia, replicando el comportamiento de jugadores en distintos niveles de habilidad.

Para entrenar el modelo, se trabajó con una gran cantidad de datos provenientes de la base de datos de Lichess, se optó por utilizar partidas con un control de tiempo de 600+0, ya que este formato ofrecía un equilibrio entre rapidez y profundidad en la toma de decisiones, asegurando que los movimientos reflejaran un estilo de juego analítico y consistente.

En la fase de preprocesamiento, se implementaron herramientas para transformar los PGN en un formato estructurado que permitiera su uso en el entrenamiento. Se adaptó la herramienta de trainingdata-tool para extraer y almacenar correctamente el ELO dentro de la nueva estructura de datos, lo que facilitó la integración de este valor como una entrada clave en el modelo.

El modelo propuesto se basa en Leela Chess Zero, un motor de ajedrez basado en aprendizaje profundo y aprendizaje por refuerzo. Sin embargo, para adaptar su arquitectura a la predicción de movimientos humanos, se realizaron modificaciones en su estructura, introduciendo el ELO como un parámetro de entrada. Esto implicó la adición de una nueva capa en la red neuronal, aumentando el número de canales de entrada, donde la capa adicional contenía el valor del ELO replicado en cada celda de la representación del tablero.

El entrenamiento del modelo se realizó utilizando más de 10 millones de partidas organizadas en chunks de datos. Se llevaron a cabo múltiples experimentos para evaluar distintas configuraciones de hiperparámetros y se encontró la combinación más efectiva.

Al comparar el rendimiento con los modelos Maia Chess, que fueron tomados como referencia debido a su enfoque en la predicción de movimientos humanos, se observó que el modelo desarrollado logró superar a Maia en términos de adaptabilidad y consistencia. La optimización de los hiperparámetros permitió alcanzar niveles de precisión más altos en los rangos de ELO más representativos, consolidando este modelo como una solución más efectiva para replicar el comportamiento humano en ajedrez.

Este estudio no solo aporta avances en la comprensión del comportamiento humano en ajedrez, sino que también representa un paso adelante en la interacción entre la inteligencia artificial y el pensamiento estratégico humano. El modelo desarrollado podría aplicarse en distintos ámbitos, como la creación de tutores personalizados de ajedrez, análisis estadístico del juego humano y desarrollo de herramientas de entrenamiento basadas en IA.

6. TRABAJO FUTURO

Si bien los resultados obtenidos en este trabajo han demostrado mejoras en la predicción de movimientos humanos en ajedrez, aún existen múltiples líneas de investigación y optimización que podrían explorarse para continuar avanzando en esta dirección.

1. Actualización de la Herramienta de Procesamiento de Datos
   Una de las mejoras más inmediatas consiste en realizar las modificaciones necesarias para que trainingdata-tool sea compatible con la última versión de LC0. Esto permitiría aprovechar las optimizaciones recientes del motor de ajedrez, asegurando que el procesamiento y la generación de datos de entrenamiento se beneficien de los avances en eficiencia y representación del juego.

2. Análisis del Comportamiento del Modelo
   Para comprender mejor las fortalezas y debilidades del modelo desarrollado, sería valioso efectuar un análisis más detallado de su comportamiento en distintos escenarios.

   - Evaluación estadística: Medir el grado de alineación del modelo con las decisiones humanas en diversas posiciones de juego.
   - Análisis ajedrecista: Examinar cómo el modelo maneja posiciones tácticas, estratégicas y de finales, comparándolo con el comportamiento esperado de jugadores humanos de diferentes niveles de ELO.
   - Identificación de escenarios clave: Determinar en qué tipos de posiciones el modelo logra predecir con mayor precisión y en cuáles presenta dificultades, con el fin de ejecutar ajustes en la arquitectura o el entrenamiento.

   Este análisis no solo ayudaría a validar los resultados obtenidos, sino que también permitiría detectar posibles mejoras y aplicaciones prácticas.

3. Parametrización del Tiempo de Partida
   Hasta ahora, el modelo ha sido entrenado utilizando un único control de tiempo (600+0), lo que ha garantizado consistencia en el aprendizaje. Sin embargo, una mejora importante sería parametrizar el control de tiempo dentro del modelo, permitiendo que la red neuronal pueda ajustar su predicción en función del tiempo de la partida. Esta parametrización ampliaría las capacidades del modelo, permitiendo su uso en un rango más amplio de escenarios de juego y brindando predicciones más precisas en diferentes modalidades de ajedrez.

Estas mejoras permitirían modernizar la infraestructura del modelo, analizar su comportamiento en profundidad y ampliar su aplicabilidad en el mundo real. La combinación de estos avances consolidaría aún más la capacidad del modelo para predecir movimientos humanos en ajedrez y ofrecer nuevas herramientas tanto para la investigación como para el aprendizaje y análisis del juego.

Bibliografía

[1] Turing A. M. (1953). Digital computers applied to games. *En: B. V. Bowden (Ed.), Faster than thought: A symposium on digital computing machines (pp. 286-295). Pitman Publishing Corporation.*

[2] Shannon C. E. (1949). Programming a computer for playing chess. *En: Philosophical Magazine, 41(314), 256-275.* doi: 10.1080/14786445008521796

[3] Newell A., Shaw J. C., & Simon H. A. (1958). Chess-playing programs and the problem of complexity. *En: IBM Journal of Research and Development, 2(4), 320-335.* doi: 10.1147/rd.24.0320

[4] Simon H. A., & Newell A. (1971). Human problem solving: The state of the theory in 1970. *En: American Psychologist, 26(2), 145-159.* doi: 10.1037/h0030806

[5] Campbell M., Hoane A. J., & Hsu F. H. (2002). Deep Blue. *En: Artificial Intelligence, 134(1-2), 57-83.* doi: 10.1016/S0004-3702(01)00129-1

[6] Silver D., Schrittwieser J., Simonyan K., Antonoglou I., Huang A., Guez A., Hubert T., Baker L., Lai M., Bolton A., Chen Y., Lillicrap T., Hui F., Sifre L., Van den Driessche G., Graepel T. & Hassabis D. (2017). Mastering the game of Go without human knowledge. *En: Nature, 550, 354-359.* doi: 10.1038/nature24270

[7] Silver D., Hubert T., Schrittwieser J., Antonoglou I., Lai M., Guez A., Lanctot M., Sifre L., Kumaran D., Graepel T., Lillicrap T., Simonyan K., & Hassabis D. (2017) Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *En: https://arxiv.org/abs/1712.01815* doi: 10.48550/arXiv.1712.01815.

[8] McIlroy-Young R., Sen S., Kleinberg J., & Anderson A. (2020). Aligning superhuman AI with human behavior: Chess as a model system. *En: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 1677-1687.* doi: 10.1145/3394486.3403219

[9] McIlroy-Young R., Wang R., Sen S., Kleinberg J., & Anderson A. (2021). Detecting Individual Decision-Making Style: Exploring Behavioral Stylometry in Chess. *En: Advances in Neural Information Processing Systems (NeurIPS 2021).* doi: 10.48550/arXiv.2208.01366

[10] McIlroy-Young R., Wang R., Sen S., Kleinberg J., & Anderson A. (2022). Learning Models of Individual Behavior in Chess. *En: Proceedings of the 28th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 1253-1263.* doi: 10.1145/3534678.3539367

[11] Rosemarin H. &, Rosenfeld A. (2019). Playing Chess at a Human Desired Level and Style. *En: Proceedings of the 7th International Conference on Human-Agent Interaction, 76-80.* doi: 10.1145/3349537.3351904.

[12] Kocsis L. & Szepesvári C. (2006). Bandit Based Monte-Carlo Planning. *En: J. Fürnkranz, T. Scheffer, & M. Spiliopoulou (Eds.), Machine Learning: ECML 2006. Lecture Notes in Computer Science (Vol. 4212, pp. 282–293). Springer.* doi: 10.1007/11871842_29.

[13] Hu J., Shen L., Albanie S., Sun G. & Wu E. (2018). Squeeze-and-Excitation Networks. *En: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 7132–7141).* doi: 10.48550/arXiv.1709.01507.

[14] He K., Zhang X., Ren S. & Sun J. (2016). Deep residual learning for image recognition. *En: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 770-778.* doi: 10.1109/CVPR.2016.90.

[15] Esteva A., Kuprel B., Novoa R. A., Ko J., Swetter S. M., Blau H.M., & Thrun S. (2017). Dermatologist-level classification of skin cancer with deep neural networks. *En: Nature, 542(7639), 115-118.* doi: 10.1038/nature21056.

[16] Brown, N. & Sandholm, T. (2019). Superhuman AI for multiplayer poker. *En: Science, 365(6456), 885-890.* doi: 10.1126/science.aay2400.

[17] Stockfish. *URL: https://stockfishchess.org/*

[18] Maia Chess. *URL: https://maiachess.com/*

[19] Lichess Database. *URL: https://database.lichess.org/*

[20] Leela Chess Zero. *URL: https://lczero.org/*

[21] Leela Chess Zero Repository. *URL: https://github.com/LeelaChessZero/*

[22] Elo A. E. (1978). The rating of chessplayers, past and present. *En: Arco Publishing.* ISBN: 9780668047218.