



# UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO  
INGENIERÍA INFORMÁTICA

## Sistema Inteligente de detección de errores tácticos en ajedrez

---

### Autor

Miguel Moles Mestre

### Tutores

Francisco Javier Melero Rus  
Francisco Herrera Triguero



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE  
TELECOMUNICACIÓN

---

Granada, Julio de 2023



# Sistema Inteligente de detección de errores tácticos en Ajedrez

Miguel Moles Mestre (alumno)

**Palabras clave:** ajedrez, aprendizaje automático, clustering, DeepChess, motores

## Resumen

El ajedrez es un deporte que ha inspirado y ha ocupado la mente de millones de personas a lo largo de varios siglos, y lo sigue haciendo hoy en día. Durante su larga historia, personas brillantes han dejado su marca en el juego, cambiando la forma en la que se ve. Pero nada ha empujado más la evolución del ajedrez como la inteligencia artificial.

Este trabajo de fin de grado trata de usar técnicas de aprendizaje automático, una rama de la inteligencia artificial, para clasificar movimientos que empeoran la posición de ajedrez para el bando que las realiza, con el objetivo de que se use como herramienta para encontrar errores comunes en el juego de un jugador y así mejorar su nivel.

El proyecto tiene tres partes principales. La primera es buscar fallos en las partidas. Para ello, nos apoyaremos en los motores de ajedrez, que son programas que analizan posiciones del tablero, les ceden una puntuación dependiendo de qué color tiene mejor posición y sugiere uno o más movimientos que considera los mejores. Nos comunicaremos con ellos haciendo uso del protocolo UCI, que es el protocolo de comunicación que usan la mayoría de ellos.

La segunda parte extraerá características importantes de los errores. Para ello probaremos varias metodologías, que hacen uso de una red neuronal ya entrenada para derivar ciertos aspectos de sus cálculos en vectores de características útiles.

Finalmente, clasificaremos los errores con técnicas de clustering, haciendo uso de las características extraídas.



# Intelligent detection system of tactical mistakes in chess

Miguel Moles Mestre (student)

Que el presente trabajo, titulado *Sistema Inteligente de detección de errores tácticos en ajedrez*, Keywords: Chess, Machine Learning, Clustering, DeepChess, Chess Engines

## Abstract

Chess is a sport that has inspired and engaged the minds of millions of people throughout several centuries, and it continues to do so today. Throughout its long history, brilliant individuals have left their mark on the game, changing the way it is perceived. However, nothing has pushed the evolution of chess more than artificial intelligence.

This thesis aims to use machine learning techniques, a branch of artificial intelligence, to classify chess moves that worsen the position for the side making them, with the goal of using it as a tool to identify common mistakes in a player's game to improve their level.

The project consists of three main parts. The first part involves identifying flaws in chess games. To do so, we will rely on chess engines, which are programs that analyze positions, assign a score based on which side has a better position, and suggest one or more moves they consider to be the best. We will communicate with them using the Universal Chess Interface (UCI) protocol, which is the communication protocol used by most chess engines.

The second part will extract important features from the mistakes. We will use different methodologies that utilize a pre-trained neural network to derive certain aspects of its calculations into useful feature vectors.

Finally, we will classify the mistakes using clustering techniques, utilizing the extracted features.



---

Yo, **Miguel Moles Mestre**, alumno de la titulación **TITULACIÓN de la Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 47433562H, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Miguel Moles Mestre

Granada a 13 de Julio de 2023.



---

**D. Francisco Javier Melero Rus (tutor1)**, Profesor del Área de Lenguajes y Sistemas Informáticos del Departamento Lenguajes y Sistemas Informáticos de la Universidad de Granada.

**D. Francisco Herrera Triguero (tutor2)**, Profesor del Área de Ciencias de la Computación e Inteligencia Artificial del Departamento Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada.

**Informan:**

Que el presente trabajo, titulado *Sistema Inteligente de detección de errores tácticos en ajedrez*, ha sido realizado bajo su supervisión por **Miguel Moles Mestre (alumno)**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 13 de Julio de 2023.

**Los directores:**

**Francisco Javier Melero Rus (tutor1)**  
**Francisco Herrera Triguero (tutor2)**

**Francisco Herrera Tri-**



# Agradecimientos

Gracias a mi familia y amigos, a la universidad de Granada y a la UNED.  
Nunca olvidaré estos años.



# Índice general

<b>I</b>	<b>Introducción</b>	<b>1</b>
<b>1.</b>	<b>Introducción</b>	<b>3</b>
1.1.	Contexto . . . . .	3
1.2.	Descripción del problema . . . . .	4
1.3.	Estructura del proyecto . . . . .	5
<b>2.</b>	<b>Objetivos</b>	<b>7</b>
<b>II</b>	<b>Contexto</b>	<b>9</b>
<b>3.</b>	<b>Ajedrez</b>	<b>11</b>
3.1.	¿Qué es el ajedrez? . . . . .	12
3.1.1.	Piezas . . . . .	14
3.1.2.	Movimientos circunstanciales . . . . .	18
3.2.	Notación ajedrecística . . . . .	19
3.2.1.	Notación SAN . . . . .	20
3.2.2.	Notación FEN . . . . .	21
<b>4.</b>	<b>Motores de Ajedrez</b>	<b>25</b>
4.1.	Historia de los motores . . . . .	25
4.1.1.	Inicio de la disciplina . . . . .	25
4.1.2.	Humanos vs. Máquinas . . . . .	27
4.1.3.	Los motores ganan . . . . .	29
4.2.	Partes de un motor . . . . .	30
4.2.1.	Representación de un tablero . . . . .	30
4.2.2.	Generación de movimientos . . . . .	32
4.2.3.	Función de evaluación . . . . .	32
4.2.4.	Exploración del árbol de movimientos . . . . .	33
<b>5.</b>	<b>Aprendizaje Automático</b>	<b>37</b>
5.1.	Partes de un problema de aprendizaje automático . . . . .	37
5.1.1.	La Tarea T . . . . .	37
5.1.2.	La experiencia E . . . . .	38

5.1.3.	La medida de efectividad P . . . . .	39
5.1.4.	Modelos de aprendizaje . . . . .	40
5.2.	Generalización . . . . .	40
5.3.	Redes neuronales . . . . .	41
5.3.1.	Neurona o perceptrón . . . . .	41
5.3.2.	Estructura . . . . .	44
5.3.3.	Backpropagation . . . . .	45
5.3.4.	Gradiente descendiente: tipos e hiperparámetros . . . . .	47
5.4.	Clustering . . . . .	49
5.4.1.	K-medias . . . . .	50
<b>III</b>	<b>Propuesta</b>	<b>53</b>
<b>6.</b>	<b>Estado del arte</b>	<b>55</b>
6.1.	Aprendizaje automático y ajedrez . . . . .	55
6.1.1.	DeepChess . . . . .	55
6.1.2.	Leela Chess Zero . . . . .	57
6.1.3.	Stockfish y las redes NNUE . . . . .	57
6.2.	Herramientas para el entusiasta . . . . .	59
6.2.1.	Aimchess . . . . .	59
6.2.2.	Chess.com . . . . .	60
<b>7.</b>	<b>Propuesta</b>	<b>61</b>
7.1.	Análisis del problema . . . . .	61
7.2.	Base de datos inicial . . . . .	62
7.3.	Estructura del trabajo . . . . .	63
7.4.	Herramientas utilizadas . . . . .	63
<b>IV</b>	<b>Implementación</b>	<b>65</b>
<b>8.</b>	<b>Extracción de errores</b>	<b>67</b>
8.1.	Definición de error . . . . .	67
8.1.1.	Puntuación de un movimiento . . . . .	67
8.1.2.	Condiciones para el error . . . . .	68
8.2.	Objetivos . . . . .	69
8.3.	Diseño del programa . . . . .	69
8.3.1.	Estructura del programa . . . . .	71
8.4.	Resultados . . . . .	72
<b>9.</b>	<b>Extracción del vector de características</b>	<b>73</b>
9.1.	De dónde se extraerán las características . . . . .	73
9.1.1.	Características a partir de la penúltima capa . . . . .	73
9.1.2.	Extracción de Pos2Vec . . . . .	75

9.2. Selección del modelo entrenado . . . . .	75
9.3. Implementación y resultados . . . . .	78
9.3.1. Creación de las redes derivadas de DeepChess . . . . .	78
9.3.2. Transformación de la base de datos de errores . . . . .	79
<b>10. Clustering de datos</b>	<b>81</b>
10.1. Valores 0 . . . . .	81
10.2. Elección del número de clusters . . . . .	84
10.3. Ejecución de k-medias . . . . .	86
10.4. Validación del clustering . . . . .	86
<b>11. Conclusiones y trabajo futuro</b>	<b>99</b>
11.1. Ejemplos del clustering . . . . .	99
11.2. Caso práctico . . . . .	99
11.3. Conclusiones . . . . .	106
11.4. Trabajo futuro . . . . .	107



# Índice de figuras

3.1.	Posición inicial del Chaturanga para 4 personas[1]. . . . .	12
3.2.	Posición inicial del ajedrez [6] . . . . .	13
3.3.	Posibles movimientos del rey[6] . . . . .	14
3.4.	Posibles movimientos de la dama[6] . . . . .	15
3.5.	Posibles movimientos del alfil[6] . . . . .	15
3.6.	Posibles movimientos del caballo[6] . . . . .	16
3.7.	Posibles movimientos de la torre[6] . . . . .	16
3.8.	Movimientos del peón en una casilla inicial [6] . . . . .	17
3.9.	Casillas que amenaza el peón blanco [6] . . . . .	17
3.10.	Ambos reyes se han enroscado, uno en un lado diferente. [7] .	18
3.11.	Las blancas pueden capturar al paso el peón que se encuentra delante de la dama, haciendo jaque mate en el proceso [10] .	19
3.12.	Codificación de las casillas [11] . . . . .	20
3.13.	Otro ejemplo de la notación FEN [8] . . . . .	22
4.1.	El Ajedrecista [12] . . . . .	26
4.2.	Preparación de la simultánea de 1985 [18] . . . . .	28
4.3.	Enumeración de las casillas del método 0x88 . . . . .	31
4.4.	Representación del árbol de movimientos, parcialmente ex- plorado [4] . . . . .	34
4.5.	Orden de exploración en profundidad [13] . . . . .	35
5.1.	Diagrama de una neurona artificial[28] . . . . .	41
5.2.	Función de activación del perceptrón . . . . .	42
5.3.	Función de activación sigmoidal . . . . .	43
5.4.	Función de activación tangente hiperbólica . . . . .	43
5.5.	Función de activación ReLU . . . . .	43
5.6.	Función de activación Leaky ReLU . . . . .	44
5.7.	Conexiones entre las capas de neuronas . . . . .	45
5.8.	Conexiones entre las capas de neuronas . . . . .	48
6.1.	Topología de la Red DeepChess [14] . . . . .	56
6.2.	Estructura de la primera NNUE de Stockfish, HalfKP [38] .	58

8.1. Gráfico de cajas de la tabla anterior . . . . .	70
8.2. Bucle de eventos de AsyncIO [17] . . . . .	71
10.1. Método del codo para mov2vec . . . . .	84
10.2. Método del codo para dif2vec . . . . .	85
10.3. Método del codo para pos2vec1 . . . . .	85
10.4. Método del codo para pos2vec2 . . . . .	86
10.5. Evolución del error en la muestra de mov2vec . . . . .	87
10.6. Evolución del error en la muestra de dif2vec . . . . .	87
10.7. Evolución del error en la muestra de pos2vec1 . . . . .	88
10.8. Evolución del error en la muestra de pos2vec2 . . . . .	88
10.9. Mapa de calor de las proporción de la etiqueta más común junto con su valor (mov2vec) . . . . .	90
10.10Mapa de calor de las proporción de la etiqueta más común junto con su valor (dif2vec) . . . . .	91
10.11Mapa de calor de las proporción de la etiqueta más común junto con su valor (pos2vec1) . . . . .	92
10.12Mapa de calor de las proporción de la etiqueta más común junto con su valor (pos2vec2) . . . . .	93
10.13Mapa de calor de la diferenciación de etiquetas en mov2vec .	94
10.14Mapa de calor de la diferenciación de etiquetas en dif2vec .	95
10.15Mapa de calor de la diferenciación de etiquetas en pos2vec1 .	96
10.16Mapa de calor de la diferenciación de etiquetas en pos2vec2 .	97
11.1. Posiciones del cluster 9 de dif2vec . . . . .	100
11.2. Posiciones del cluster 16 de dif2vec . . . . .	101
11.3. Posiciones del cluster 17 de dif2vec . . . . .	102
11.4. Proporción de pertenencia a los clusters de mov2vec de los errores en las partidas de William Steinitz . . . . .	103
11.5. Proporción de pertenencia a los clusters de dif2vec de los erro- res en las partidas de William Steinitz . . . . .	104
11.6. Proporción de pertenencia a los clusters de pos2vec1 de los errores en las partidas de William Steinitz . . . . .	104
11.7. Proporción de pertenencia a los clusters de pos2vec2 de los errores en las partidas de William Steinitz . . . . .	105

# Índice de cuadros

3.1. Valores de las distintas piezas . . . . .	17
4.1. Algunas partidas entre motores y grandes maestros de 2014 a 2017[9] . . . . .	29
8.1. Visualización del tiempo para encontrar 10.000 errores . . . . .	69
8.2. Tiempos que tarda Stockfish en analizar una posición a cierta profundidad. Se han usado 200 posiciones aleatorias de la base de datos, y por cada nivel siempre se han utilizado las mismas.	70
8.3. Partidas, posiciones y errores analizados . . . . .	72
9.1. Matrices de confusión de las parejas de cualquier posición . . . . .	77
9.2. Matrices de confusión de las parejas de posiciones fáciles . . . . .	77
9.3. Matrices de confusión de las parejas de posiciones medio . . . . .	78
9.4. Matrices de confusión de las parejas de posiciones difíciles . . . . .	78
10.1. Valores 0s presentes en los datos . . . . .	81
10.2. Análisis de valores 0 después de eliminar características nulas . . . . .	82
10.3. . . . .	84
10.4. Etiquetas de validación, extraídas de [26]. Cada etiqueta es aplicada a los dos bandos . . . . .	89



# **Parte I**

# **Introducción**



# Capítulo 1

## Introducción

### 1.1. Contexto

El ajedrez es un juego de mesa que ha inspirado y ha ocupado la mente de millones de personas a lo largo de varios siglos, y lo sigue haciendo hoy en día. Durante su larga historia, personas brillantes han dejado su marca en el juego, cambiando la forma en la que se ve el juego. Pero nada ha empujado más la evolución del ajedrez como la inteligencia artificial.

La inteligencia artificial (IA) es uno de los campos de conocimiento que más ha crecido en estos últimos tiempos. Hoy en día, se emplea en una gran cantidad de campos, desde la medicina para el diagnóstico de enfermedades, análisis de proteínas o genómica; hasta en el marketing, con análisis de emociones o análisis de perfiles de clientes.

Este dominio es capaz de automatizar aquellas tareas que se consideraban inabarcables para la computación, pero que un humano puede realizar. Esto es porque es capaz de replicar la inteligencia humana, y como ChatGPT[5] ha demostrado, también puede superarla.

Entre la inteligencia artificial y el ajedrez apareció una sinergia que ha empujado a ámbas disciplinas a mejorar rápidamente durante décadas. El nivel de ajedrez humano ha mejorado de forma notable desde la introducción de programas capaces de jugar a un nivel alto al ajedrez, mientras que estos programas llegaron a una velocidad inimaginable.

La IA ha avanzado tanto en el mundo del ajedrez que cualquier teléfono *smartphone* es capaz de ganar a cualquier humano. Por ello, los esfuerzos de esta rama se centran sobre todo en mejorar el nivel de los jugadores. El objetivo de nuestro trabajo se centra también en ayudar al aficionado, y lo haremos clasificando errores de ajedrez de forma que se pueda sacar información de los errores de ajedrez de una persona.

Para resolver este problema, nos apoyaremos en dos ramas de la inteligencia artificial, el *deep learning* y el análisis de *clusters*

## 1.2. Descripción del problema

El problema que tenemos entre manos es la clasificación de errores de ajedrez, es decir, categorizar movimientos que empeoran de forma objetiva la posición para el bando que lo hizo; y hacer esto con técnicas de aprendizaje no supervisado.

Las técnicas de aprendizaje no supervisado para clasificar entran en el campo de análisis de clusters o *clustering*. Los algoritmos de clustering tienen en común un requisito: necesitan un criterio para saber si un elemento y otro son similares entre sí, si no se tiene es imposible agruparlos en categorías.

Este requisito es difícil de cumplir para movimientos o posiciones de ajedrez. No hay ningún algoritmo capaz de hacer eso pero, como se describirá en el estado del arte, los humanos expertos son capaces de distinguir posiciones similares, y coinciden en gran medida entre sí. Por lo tanto, tenemos la siguiente dicotomía: existe una forma de comparar posiciones de ajedrez, pero no tenemos ningún algoritmo para hacerlo. Es el escenario perfecto para usar aprendizaje automático.

Para ello, en este proyecto usaremos un método novel para extraer características que consideraremos importantes sobre un movimiento de ajedrez. Este método se basa en un modelo de aprendizaje descrito en la publicación de DeepChess[14], una red neuronal siamesa capaz de comparar dos posiciones de ajedrez e indicar en cuál es más probable que gane un bando u otro. Por la estructura de esta red, justificaremos que se pueden usar varios métodos para extraer características importantes del input usando capas intermedias de esa red.

Finalmente, se deberá definir lo que se considera un error de ajedrez, y construir un programa que sea capaz de encontrarlos. Para ello, se hará uso de motores de ajedrez, que es software capaz jugar a un nivel muy superior al de cualquier humano. Estos programas pueden dar una puntuación a una posición de ajedrez que indica qué bando está mejor, y cómo de mejor. Usando esta herramienta, se usará la diferencia entre las evaluaciones de las posiciones como indicador de la calidad de un movimiento, y se diseñará un software que analice posiciones a la máxima velocidad posible, haciendo uso de varias instancias de motores a la vez.

En resumen, el problema que se plantea tiene tres partes a tratar:

- La extracción de errores de ajedrez de una base de datos de partidas.

- La extracción de características importantes de un error.
- La clasificación de estos errores según dichas características.

### **1.3. Estructura del proyecto**

Este proyecto se estructura en cuatro partes, las cuales se subdividen en capítulos.

La primera parte está formada por la introducción, los objetivos a cumplir y el presupuesto.

La segunda parte consta del contexto necesario para este proyecto. Esto incluye una introducción del ajedrez y sus normas, de la computación en el ajedrez y de las ramas del aprendizaje automático que se usan.

La tercera parte describe la situación del estado del arte en el problema a tratar y la propuesta inicial de la solución.

Finalmente, la cuarta parte explicará el desarrollo de la solución, los resultados, las conclusiones y el trabajo futuro.



## Capítulo 2

# Objetivos

El objetivo principal de este proyecto es crear un proceso por el cual, a través de técnicas de aprendizaje automático no supervisado, **se puedan descubrir motivos tácticos ajedrecísticos comunes en los errores de un jugador de ajedrez.**

Este objetivo se puede dividir en varias partes:

- Desarrollar un programa capaz de obtener errores de ajedrez de un conjunto de partidas.
- Plantear varios métodos para extraer información importante de los errores de ajedrez.
- Usar técnicas de clustering para definir categorías de movimientos que nos aporten información.



## **Parte II**

# **Contexto**



## Capítulo 3

### Ajedrez

El **ajedrez** es uno de los juegos de mesa más antiguos y populares que existen. Su origen se encuentra lleno de misterios, y los historiadores no están de acuerdo sobre qué leyendas e historias son reales o no. Aun así, la mayoría están de acuerdo de que surgió en la India.

Uno de los juegos que considera un predecesor del ajedrez es el Chaturanga. Se jugaba en un tablero de  $8 \times 8$  casillas, y su nombre se traduce a cuadripartito, haciendo referencia a los 4 tipos de piezas que tenía (infantería, caballería, elefantes de guerra y carros de combate). La primera mención de este juego en la literatura india es en un poema épico del siglo VI a.C., y existía una versión para dos jugadores y otra para cuatro [3].

El ajedrez no es el único juego moderno que ha surgido del Chaturanga. Forma parte de una familia de juegos de mesa, en los cuales tenemos una temática similar de un ejército batallando contra otro, se juegan en tableros con casillas y tienen piezas que realizan funciones diferentes. En esta categoría, entran juegos como el shogi, el xiangqi y el janggi, actualmente populares en la parte oriental del globo.



Figura 3.1: Posición inicial del Chaturanga para 4 personas[1].

### 3.1. ¿Qué es el ajedrez?

Es un juego de estrategia entre dos oponentes, que toman el **bando blanco y negro** respectivamente. Cada uno de ellos cuenta con 16 piezas al inicio del juego, situadas en un tablero de  $8 \times 8$  casillas. Estos espacios tienen color blanco o negro, que se alterna en casillas contínuas, y pueden estar vacías o contener una única pieza. Por turnos, cada jugador mueve una de sus 16 piezas de una casilla a otra, comenzando por el bando blanco y respetando las normas de este juego.

Hay seis tipos diferentes de piezas en el ajedrez, y comienzan en diferentes números al comienzo de la partida: ocho **peones**, dos **caballos**, dos **alfiles**, dos **torres**, una **dama** y un **rey**. Las diferencias entre ellas se encuentran en cómo pueden desplazarse y atacar piezas enemigas, siendo algunas de ellas más capaces que otras, y por tanto dándoles más valor.

Entre ellas, el rey es la más importante, ya que el objetivo y la forma de ganar es capturando al del oponente. A esto se le llama hacer **jaque mate**, y se consigue amenazando la casilla donde éste se encuentra de forma que no pueda moverse a una casilla segura, protegerse con otra pieza o capturar la pieza amenazante. Si se ataca al rey enemigo, pero el oponente tiene la posibilidad de hacer una de las tres acciones anteriores, se denomina hacer un **jaque**, y quien lo recibe tiene la obligación de evitar la captura de su rey en su siguiente movimiento. Aparte de haciendo jaque mate al oponente, también se puede ganar en caso de abandono o si el contrario se queda sin tiempo en el reloj si la partida se juega con tiempo límite.



Figura 3.2: Posición inicial del ajedrez [6]

Otro resultado posible en el ajedrez es empatar, llamado **hacer tablas**. Esta situación es común en las partidas de alto nivel, y se puede llegar a ella de formas diferentes:

- Si un jugador ofrece tablas al otro y son aceptadas,
- Si se deja al rival sin movimientos legales sin hacer jaque mate (llamado rey ahogado),
- Si se repite la misma posición en el tablero tres veces,
- Si ningún jugador tiene piezas suficientes para hacer jaque mate al oponente, y
- Si se realizan 50 movimientos sin ninguna captura o ningún movimiento de peón.

Hoy en día, el ajedrez es considerado por el Comité Olímpico Internacional un deporte y las competiciones internacionales son reguladas por la **FIDE**<sup>1</sup>. En la mayoría de torneos los jugadores compiten a nivel individual aunque existen algunas competiciones por equipos siendo una de las más importantes las Olimpiadas de Ajedrez.

---

<sup>1</sup>Acrónimo de la Federación Internacional de Ajedrez, una organización internacional que conecta las diversas federaciones nacionales de ajedrez.

### 3.1.1. Piezas

Un turno en el ajedrez consiste en desplazar una única pieza de una casilla a otra, con la posibilidad de capturar a una del oponente. Esto se realiza ocupando la casilla en la que se encontraba la otra pieza, y eliminándola del tablero (siempre que el movimiento sea legal). Las piezas capturadas no vuelven al tablero hasta que acaba la partida, como en otros juegos como el shogi.

Como se ha dicho anteriormente, hay seis tipos de piezas en el ajedrez, cada una con patrones de movimientos diferentes. Éstas son:

- **El rey:** La pieza más importante de cada bando, sólo hay uno por cada color. Su casilla inicial es en el centro de la fila de atrás, en la casilla cuyo color es diferente al del bando. Puede moverse y capturar en las cuatro direcciones ortogonales y las cuatro diagonales, pero sólo una casilla de distancia.

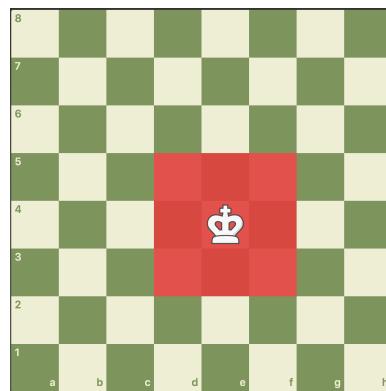


Figura 3.3: Posibles movimientos del rey[6]

- **La dama:** La pieza más poderosa; igual que el rey, sólo hay una por bando. Comienza en la otra casilla central de la fila trasera, y puede moverse en vertical, horizontal y diagonal todas las casillas que pueda, es decir, hasta que una pieza aliada o enemiga la bloquee (no puede saltar piezas).

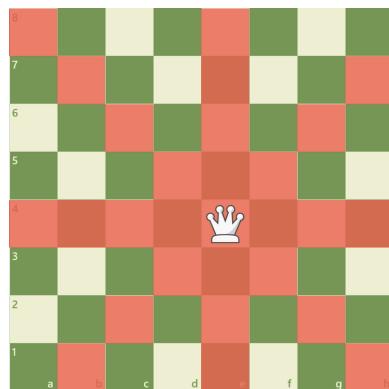


Figura 3.4: Posibles movimientos de la dama[6]

- **El alfil:** Hay dos por bando, a los lados del rey y la dama. Son capaces de moverse en las dos diagonales la distancia que puedan, igual que la dama.

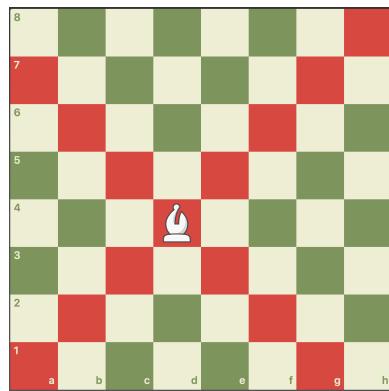


Figura 3.5: Posibles movimientos del alfil[6]

- **El caballo:** Con dos en cada color, los caballos son piezas con una característica especial: pueden saltar piezas. Se mueven y capturan realizando un movimiento en forma de ele: dos casillas en una dirección ortogonal, y después una hacia el lado. Comienzan a los lados de los alfiles, en la misma fila.

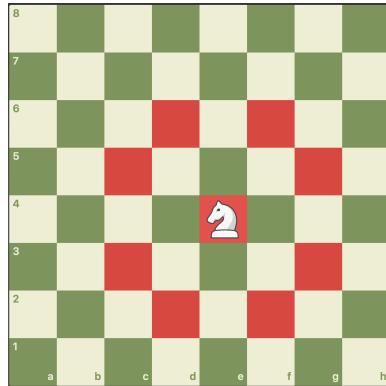


Figura 3.6: Posibles movimientos del caballo[6]

- **La torre:** Similar a la dama, puede moverse en todas las direcciones ortogonales las casillas que quieran. Hay dos por color, situadas en los bordes de la primera fila, al lado de los caballos.

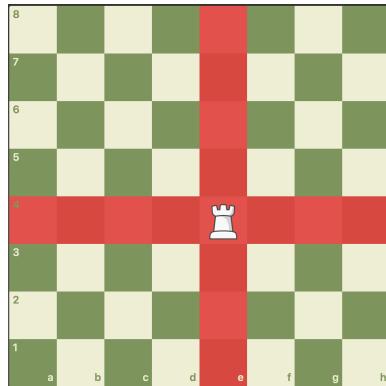


Figura 3.7: Posibles movimientos de la torre[6]

- **El peón:** La pieza más numerosa en el tablero y quizás la más compleja, los peones ocupan la segunda fila entera de cada bando, por lo que hay ocho por color. Los peones son la única pieza que tiene patrones diferentes de movimiento y captura: sólo se pueden desplazar una casilla hacia delante, pero capturan en las dos casillas diagonales que se sitúan delante de él. Además, si es la primera vez que se va a mover en la partida, puede moverse dos casillas en vez de una.



Figura 3.8: Movimientos del peón en una casilla inicial [6]



Figura 3.9: Casillas que amenaza el peón blanco [6]

De estas piezas, hay algunas que se consideran más valiosas que otras. Por este motivo, se ha asociado una puntuación relativa a cada tipo de pieza.

Cuadro 3.1: Valores de las distintas piezas

Pieza	Valor
Peón	1 punto
Caballo	3 puntos
Alfil	3 puntos
Torre	5 puntos
Dama	9 puntos

Estas son las puntuaciones comúnmente aceptadas y que se suelen enseñar a los principiantes, aunque hay mucha discusión entorno a ello [19]. De hecho, uno de los aspectos estratégicos más importantes en el ajedrez es el valor de la pieza que no aparece en la lista: el rey. Su captura gana la

partida, lo que implica que vale más que el resto de las piezas juntas, lo cual crea situaciones donde es óptimo ceder piezas a cambio de un ataque al rey enemigo.

### 3.1.2. Movimientos circunstanciales

Hay tres tipos de movimientos que sólo son realizables si se cumplen ciertas circunstancias. Éstos son:

- **El enroque:** Es la única excepción en donde se mueven dos piezas en el mismo turno. El movimiento consiste en desplazar el rey dos casillas hacia una de las torres aliadas, y colocar a ésta adyacente al rey, en el lado contrario de donde se situaba. Enrocarse con la torre más cercana al rey se denomina **enroque corto**, mientras que hacerlo en la otra dirección es un **enroque largo**.

Para enrocarse, deben cumplirse algunas condiciones: ni la torre ni el rey se han movido aún, no hay piezas entre el rey y la torre, el rey no se encuentra en jaque y, finalmente, ninguna de las casillas en las que se desplazará el rey puede estar amenazada.



Figura 3.10: Ambos reyes se han enrocado, uno en un lado diferente. [7]

- **Coronar:** Los peones no se pueden mover hacia atrás. Entonces, ¿qué sucede cuando llegan a la fila del fondo del tablero? El peón se dice que corona: se convierte en otra pieza de mejor calidad (caballo, alfil, caballo o dama). Esta posibilidad hace que un peón tenga mucho valor si avanza mucho en el tablero y en el caso que no haya muchas piezas en el tablero, ya que es más difícil capturarlo. Coronar el peón cuando llega a la última fila es obligatorio.
- **Comer al paso:** Este movimiento se puede hacer cuando un peón rival avanza dos casillas y queda al lado de otro peón aliado. En este

caso, se puede comer el peón rival de la misma forma que si se hubiese movido sólo una casilla. De hecho, la situación final es exactamente la misma que si hubiese pasado eso: el peón aliado se mueve una casilla en diagonal hacia adelante. Es el único caso en el que se captura una pieza y no se ocupa su casilla. Capturar al paso sólo se puede hacer el turno siguiente al que el oponente realiza el movimiento, en los siguientes el movimiento se consideraría ilegal.



Figura 3.11: Las blancas pueden capturar al paso el peón que se encuentra delante de la dama, haciendo jaque mate en el proceso [10]

### 3.2. Notación ajedrecística

La notación ajedrecística se usa para guardar los movimientos de una partida, para realizar análisis de posibles movimientos o para anotar posiciones específicas. Es usada en la literatura ajedrecista y por jugadores que quieren mantener un registro de sus partidas, o que quizás quieren reanudar un juego que se quedó sin completar.

Las primeras partidas de ajedrez anotadas usaban frases enteras para describir movimientos, como este ejemplo del jugador del siglo slowroman-capxviii@ Philidor: „El alfil de rey, a la cuarta casilla de su alfil de dama” [29]. La notación fue evolucionando con el tiempo, reduciéndose poco a poco, en lo que se llama **notación descriptiva**, que se caracteriza por nombrar a las columnas según la pieza que ocupa la última fila en la posición inicial, y en usar una numeración diferente para las filas dependiendo del bando del

movimiento. Esto último complicaba visualizar el tablero, ya que la sexta fila de las blancas se anotaba como la segunda de las negras, y el jugador debía rotar su imagen del tablero a cada movimiento descrito.

### 3.2.1. Notación SAN

Hoy en día, se usa la llamada **notación algebraica**, que resuelve este problema además de ser más compacta. Se basa en un sistema de coordenadas para identificar de manera unequivoca cada casilla del tablero: nombra a las filas con los números del 1 al 8 comenzando por la fila del bando blanco, y las columnas con letras de la *a* a la *h*, comenzando por la columna de la torre de dama.

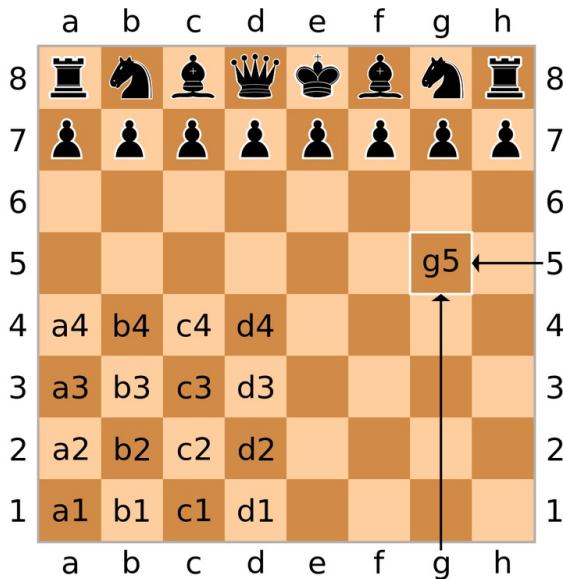


Figura 3.12: Codificación de las casillas [11]

Para anotar un movimiento, se escribe la inicial de la pieza que se mueve en mayúsculas, y las coordenadas de la casilla destino, excepto si la pieza es un peón, en cuyo caso la P se obvia y no se escribe. Por ejemplo, *Ag7* indica que el alfil se ha desplazado a la casilla *g7*. En el caso de que haya dos piezas del mismo tipo candidatas para hacer el movimiento, se añade después de la inicial información que elimine la ambigüedad, ya sea la fila o la columna de origen de la pieza. Así, el movimiento *Tad1* referencia que la torre que se encuentra en la columna *a* se ha desplazado a la casilla *d1*.

En el caso de que ocurra una captura, se añade una *x* entre la inicial de la pieza y las coordenadas, como puede ser el movimiento *Cxb3* o *axb5* (en el caso de los peones, siempre se escribe la columna inicial en caso de

captura).

Si ocurre un jaque, se anota con el signo + al final del movimiento, y si hay jaque mate, con ++ o #. El enroque corto se anota como 0-0, mientras que el largo como 0-0-0. Finalmente, se puede destacar un movimiento si es especialmente fuerte con signos de exclamación, o con signos de interrogación en el caso de que sea muy flojo, e incluso combinarlos si la jugada es interesante.

Desde 1980, la FIDE admite únicamente la notación algebraica en sus torneos. Hay tres variantes comúnmente aceptadas:

- Variante larga, en la que siempre se indica la casilla de inicio del movimiento.
- Variante corta, la que se ha explicado.
- Variante mínima, donde se omiten los indicadores de jaque mate, de jaque y de captura.

La **notación SAN** hace referencia a la notación algebraica corta (*Short Algebraic Notation*), y es la que se usará en este documento.

### 3.2.2. Notación FEN

La notación *Forsyth–Edwards* o **FEN** es la notación más usada para registrar posiciones de ajedrez concretas. La codificación proporciona toda la información necesaria para reanudar una partida desde el punto indicado sin tener que proveer todo el historial de movimientos, lo cual compacta mucho la información. Es muy usada para indicar el punto de inicio de un análisis o un problema de ajedrez, para describir la posición inicial de variantes de ajedrez como Chess960 (donde las piezas comienzan en posiciones diferentes cada partida) y para usarla en programas de ajedrez, ya que la mayoría de ellos la aceptan. La notación ocupa poco espacio pero a la vez es fácilmente legible, convirtiéndola en un punto intermedio entre una notación humana y una de ordenadores.

La codificación tiene dos partes. La primera parte y la principal es la posición de las piezas, que se hace de la siguiente manera:

- El tablero se lee por filas, desde la 8 hasta la 1, que se escriben separadas por barras (/).
- Por cada fila, se anotan las piezas de izquierda a derecha. Si nos encontramos con huecos vacíos, se escribe un número indicando cuántos hay seguidos.

- Las piezas se codifican en una letra, su inicial en inglés (K→rey, Q→dama, R→torre, B→alfil, N→caballo, P→peón), y se escribe en minúscula en el caso de que la pieza sea negra, y mayúscula en caso contrario.

Por ejemplo, la posición inicial de ajedrez se codificaría de la siguiente manera: rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR. Otro ejemplo se puede ver en la posición de la figura 3.13.



Figura 3.13: Otro ejemplo de la notación FEN [8]

La segunda parte de la codificación es la información complementaria a la posición. Se anota, por orden y separado por espacios, los siguientes datos:

- El color del bando al que le toca mover: *w* para las blancas, *b* para las negras.
- Los enroques legales en cada bando. Se escribe una K para el enroque corto y Q para el largo, en mayúscula para el blanco, minúscula para el negro. Así, KQq indica que todos los enroques están permitidos excepto el corto del bando negro. Para indicar que no hay enroques legales, se escribe un guión (-).
- Captura al paso: en el caso de que un peón se haya movido dos casillas, se escribe las coordenadas de la casilla que se encuentra detrás de él (aunque no sea posible capturarlo). Si no hay captura al paso legal, se indica con un guión (-).

- Número de medias jugadas realizadas desde la última captura o movimiento de peón, dato importante para la regla de los cuarenta movimientos.
- Número de turno entero actual. Un turno entero consiste en un movimiento de cada bando. Por lo tanto, este valor es 1 en la posición inicial, y se incrementa después de un movimiento de las negras.

Un FEN de ejemplo es el siguiente:  $\ddot{8}/5k2/3p4/1p1Pp2p/pP2Pp1P/P4P1K/8/8$   
b - - 99 50: Aquí nos indican que le toca mover a las negras, no hay enroques legales ni capturas al paso, y que habrá tablas en el siguiente movimiento si no se hace una captura o movimiento de peón.



## Capítulo 4

# Motores de Ajedrez

Since the mid 1960s, researchers in computer science have famously referred to chess as the ‘drosophila’ of artificial intelligence (AI). What they seem to mean by this is that chess, like the common fruit fly, is an accessible, familiar, and relatively simple experimental technology that nonetheless can be used productively to produce valid knowledge about other, more complex systems.[15]

Un motor de ajedrez es un programa que analiza posiciones de ajedrez y genera uno o varios movimientos que considera los más fuertes.

El ajedrez tiene una historia compleja, llena de etapas evolutivas importantes. Aun así, la revolución más grande que ha habido en su historia ha sido los motores. Hoy en día están en todas partes: en los teléfonos móviles, ordenadores y en las páginas web para jugar online. Todo el mundo tiene acceso a ellos, y son más fuertes que cualquier gran maestro, por lo que es natural que ahora se use menos de oponente, y más como una herramienta de aprendizaje y mejora del juego.

### 4.1. Historia de los motores

#### 4.1.1. Inicio de la disciplina

Los primeros intentos de crear máquinas de ajedrez fueron dispositivos mecánicos. El primero de ellos, *el Autómata Turco*, consistía en una figura humana a tamaño real sentada en una mesa llena de complicados mecanismos que circuló de gira por europa del 1770 al 1854. Tenía la capacidad de ganar a oponentes humanos, e incluso detectar movimientos ilegales, pero su inteligencia venía de un humano escondido dentro del aparato. La primera



Figura 4.1: El Ajedrecista [12]

instancia real de un ordenador de ajedrez aparecería en 1912, en forma de un autómata llamado *El Ajedrecista*, creado por el inventor español Leonardo Torres y Quevedo. Esta máquina era capaz de jugar un final con las piezas blancas, en el cual él tenía una torre y su rey mientras que el negro sólo el rey. *El Ajedrecista* era capaz de realizar jaque mate en todas las posibles posiciones, y era incluso capaz de detectar movimientos ilegales.

En las décadas de los 40 y 50, dos nombres conocidos establecieron las bases de los programas de ajedrez: Alan Turing y Claude Shannon. Shannon publicó en 1950 los detalles de un programa que podría potencialmente jugar a ajedrez contra un humano. Un año después, Turing creó el primer algoritmo para jugar a ajedrez, pero el hardware de la época no tenía suficiente potencia. En 1951, Dietrich Prinz implementó un programa capaz de resolver un mate en dos para el primer ordenador comercial, el *Ferranti Mark I*, pero no fue hasta 1957 que hubo un motor de ajedrez completamente automático. El programa tardaba 8 minutos en hacer un movimiento, pero era capaz de jugar una partida de principio a final, y marcó el comienzo real de la historia del ajedrez computacional.

En los años 60 y 70, los algoritmos de los motores mejoraron signifi-

cativamente. El algoritmo **MiniMax** fue inventado por el genio John Von Neumann en los años 20, y encaja a la perfección para el ajedrez. El algoritmo explora un árbol de posibles acciones en un juego con contrincantes, intentando maximizar la puntuación para un jugador mientras minimiza para el otro. En los años posteriores, MiniMax fue ajustado específicamente para el ajedrez, priorizando mejorar la profundidad de búsqueda. Se incluyeron técnicas heurísticas avanzadas, bases de datos de aperturas y finales, y una forma alternativa de recorrer el árbol de movimientos llamada profundización iterativa. Pero la optimización más importante fue la llamada **poda alfa beta**, que reduce el espacio de búsqueda hasta un 90 %.

Durante este tiempo algunos grandes maestros del ajedrez ayudaron a la mejora de los motores de ajedrez, con su profundo conocimiento del juego. Destaca principalmente **Mikhail Botvinnik**, anterior campeón del mundo de ajedrez, que escribió algunas publicaciones sobre técnicas de selección de movimientos.

El hardware, que fue el factor limitante para Turing y Shannon, mejoró de forma exponencial durante esos años, siguiendo la ley de Moore. También apareció hardware específico para motores de ajedrez en este tiempo. Estas mejoras consiguieron elevar el nivel de los ordenadores a una velocidad sorprendente. En 1967, MacHack VI se convirtió en el primer ordenador en ganar a un oponente en un torneo, con un elo aproximado de 1300; pero ya en el 76 tenemos a otro, llamado Chess 4.5, que ganó el torneo *Minnesota Open* jugando a un nivel de 2271 de elo.

Por último, se creó una competición alrededor de los motores. El primer torneo de programas de ajedrez se organizó en 1970, en Nueva York, y cuatro años después se convirtió en un evento anual, denominado World Chess Computer Championship (WCCC). La aparición de competición en el campo impulsó la creación de nuevas empresas, centradas en el desarrollo de software y hardware de ajedrez.

#### 4.1.2. Humanos vs. Máquinas

Para el comienzo de los 80, los motores de ajedrez se habían convertido en un negocio lucrativo por la expansión de los ordenadores personales. Crear algoritmos mejores significaba diferenciarse de la competencia, y por ello el nivel de juego de las máquinas siguió aumentando.

El primer motor en llegar al nivel de un gran maestro fue **Deep Thought** que, en 1988, empató en primer puesto con el GM Tony Miles en el US Open. Cuando esta barrera fue superada, sólo quedaba el último paso: llegar al nivel del campeón del mundo de ajedrez, **Garry Kasparov**, uno de los jugadores más brillantes de la historia. Pero ésto resultó ser más difícil de lo esperado. En 1985, previo a la victoria de Deep Thought, se celebró en Hamburgo



Figura 4.2: Preparación de la simultánea de 1985 [18]

una simultánea inusual: Kasparov contra 32 de los motores más fuertes de la época, y las máquinas perdieron todas las partidas. En 1989, hubo un encuentro entre Kasparov y Deep Thought a dos partidas, y Kasparov salió victorioso.

Los años 90 fueron testigos de una guerra humanos vs máquinas, en la que los ordenadores siguieron cogiendo fuerza. Aun así, no consiguieron ningún éxito hasta **Deep Blue**. Deep Blue fue el resultado del esfuerzo de los creadores del motor Deep Thought; y IBM, que los tomó bajo su ala. En el año 1996, después de 7 años de mejora continuos, consiguieron celebrar otro encuentro contra Kasparov, donde consiguieron hacer historia: ganaron la primera partida, convirtiéndose en los primeros en vencer al campeón del mundo en una partida clásica. El encuentro acabó 4-2 en favor de Garry, pero esa victoria les indicó que no estaban muy lejos de ser los vencedores del encuentro. Por ello, durante un año, trabajaron en mejorar el motor, incluso contratando al gran maestro Joel Benjamin como consultor, que fue el responsable de crear un libro de aperturas. Después de una preparación concienzuda, desafiaron de nuevo a Kasparov y consiguieron ganar el icónico encuentro.

La década después de la victoria de Deep Blue fue testigo de un cambio progresivo de papeles entre las máquinas y los humanos. Mientras que eran los motores quienes acababan siendo superiores, los encuentros contra jugadores humanos seguían siendo competitivos. En el 2000, el programa Deep Junior participó en el Dortmund Sparkassen Chess Meeting, un torneo de élite en el participaron 9 jugadores de élite y él, y en el que quedó 6º. En 2002, el nuevo campeón del mundo Vladimir Kramnik empató su encuentro contra Deep Fritz; y el año siguiente Kasparov empató contra dos motores: Deep Junior 7 y X3d Fritz. La transición terminó en el encuentro de 2006 entre Kramnik y el motor Deep Fritz, que acabó con un resultado decisivo 4-2.

Cuadro 4.1: Algunas partidas entre motores y grandes maestros de 2014 a 2017[9]

GM	Motor	Resultado	Handicap	Año
Daniel Naroditsky	Stockfish 5	0.5-3.5	Asistencia del motor Rybka 3	2014
Hikaru Nakamura	Stockfish 5	0.5-1.5 0.5-1.5	Asistencia de Rybka 3 Peón y piezas blancas	2014
Sergie Movsesian	Komodo 9.2	1-3 0.5-1.5	Peón y piezas blancas Intercambio	2015
Victor Mikhalevski	Komodo 1538.00	0.5-1.5 0-2 0.5-1.5	Peón y piezas blancas Intercambio y piezas blancas Dos peones	2016

#### 4.1.3. Los motores ganan

El nivel de los motores fue subiendo lento, pero de forma continuada. Una nueva competición surgió en 2010, el Top Chess Engines Competition (TCEC) que, en comparación al WCCC, se centraba en juegos largos con hardware de alto nivel, y pronto se convirtió en una referencia para ver el nivel más alto que se ha alcanzado en el ajedrez. Y el nivel era tan alto que los humanos no podían competir.

Las partidas humano vs. máquina se realizaban ahora con hándicaps, es decir, el motor comenzaba con alguna desventaja (normalmente material) o el jugador tenía alguna ventaja. Aun así, usualmente los vencedores acababan siendo los motores.

Los motores se volvieron muy populares. El motivo era claro: ahora todo el mundo podía tener a un gran maestro en casa, en su ordenador, lo cual es muy atractivo. Se convirtieron en una herramienta indispensable para cualquier aficionado y, con la llegada de los smartphones, se podían usar en cualquier momento y lugar. De 2006 a 2017, los motores se infiltraron en todas las áreas del ajedrez, desde las retransmisiones de partidas hasta el análisis, y parecía que ningún cambio drástico iba a ocurrir.

Pero entonces llegó AlphaZero, con un enfoque completamente diferente. Un grupo de investigadores de la compañía Google AI Deep Mind descartó todo el conocimiento previo sobre ajedrez y creó una red neuronal profunda capaz de aprender jugando contra sí misma. Para ello, hicieron uso de técnicas de aprendizaje por refuerzo, y de un método de exploración de árboles nunca usado anteriormente en juegos de mesa, el método de Monte-Carlo, un método probabilístico que se basaba en la confianza que tenía la red neuronal en sus sugerencias de movimientos[35].

Lo más sorprendente de AlphaZero no fue su revolucionaria estructura, pero la fuerza de su juego. El equipo creó un encuentro de 100 partidas entre su nuevo motor y Stockfish 8, que era el motor más fuerte en el momento. AlphaZero ganó 28 partidas, mientras que no sufrió una sola derrota.

Hoy en día, el enfoque basado en redes neuronales es algo inusual. Leela Chess Zero, un motor de código abierto que usa este enfoque, es el mejor representante y se puede considerar el sucesor de AlphaZero. En 2019, era capaz de vencer a cualquier motor tradicional, y pasó a un segundo plano cuando Stockfish 12 sustituyó su función de evaluación por una red neuronal al año siguiente. Esta red neuronal es una poco profunda, más centrada en ser eficiente en computación que en la complejidad. La combinación de la fuerza bruta de Stockfish junto con la intuición que provee la red neuronal ha llevado a la dominancia de Stockfish en las competiciones posteriores, hasta el día de hoy.

## 4.2. Partes de un motor

Los motores de ajedrez son complejos y muy variados. Sin embargo, si reducimos sus acciones en sus términos más simples, ellos hacen dos cosas:

- **Evaluuar posiciones, y**
- **Explorar movimientos**

Lo que diferencia cada uno de los motores son los algoritmos y las representaciones de datos que usan para realizar estas tareas.

### 4.2.1. Representación de un tablero

Un motor necesita establecer cómo va a representar una posición de ajedrez en memoria. El resto de partes del motor se diseña a su alrededor y a la vez se encuentra afectado por ella. Cada una tiene sus ventajas y desventajas: unas son intuitivas, otras aportan información extra para facilitar cálculos. De hecho, no es extraño que se usen varias de ellas para diferentes partes del programa.

La representación más simple en la que se puede pensar es de un array 8x8 de enteros que almacene, por ejemplo, el tipo de pieza de cada casilla con un número y el color usando el signo. Este método es llamado **lista cuadrada**, y tiene un problema por el que no se usa: a la hora de obtener los movimientos legales, se debe comprobar constantemente si cada patrón acaba dentro del tablero. Una solución es usar un array más grande, y llenar los espacios que se encuentren fuera del tablero con un valor especial. Los

112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Figura 4.3: Enumeración de las casillas del método 0x88

tamaños más usuales son de 12x12 o de 12x10. Otra forma de solucionar esto es el llamado **método 0x88**, que usa un array unidimensional de  $8 \times 16 = 128$ . El método se basa en usar dos tableros, el de la izquierda siendo el de la partida y el de la derecha el de los movimientos ilegales. Esta representación tiene dos ventajas fundamentales comparado con los tamaños 12x12 o 12x10:

- Al tener potencias de 2 como dimensiones, se pueden usar máscaras de bits para saber si dos coordenadas se encuentran en la misma fila o columna.
- Comprobar si un número se encuentra en el tablero de movimientos legales es tan sencillo como realizar una operación AND de la coordenada de la casilla con el número 0x88 y, si el resultado es 0, se encuentra dentro.

Otra forma completamente diferente de almacenar una posición es usando **bitboards**. En el caso de un tablero de ajedrez, un bitboard de la posición ocupa 64 bits, es decir, una sola palabra de una CPU. Esta representación añade complejidad al necesitar más de un bitboard, pero a cambio se pueden hacer operaciones de bits entre tableros de forma ágil. Dos formas posibles de usarlos es usar un bitboard para cada combinación de pieza y color (12 en total), o usar un bitboard para cada tipo de pieza y dos más para los colores (8 en total). Aunque esa cantidad de bitboards es suficiente, se suelen añadir más para información, como uno que enseña las casillas atacadas por un bando o qué piezas atacan una casilla en concreto. Las posibilidades son enormes.

Pero estas representaciones también tienen sus problemas. Uno de los más importantes es generar los vectores de ataque de las piezas deslizantes: alfíes, torres y damas. Los movimientos posibles de estas piezas dependen de los espacios libres de la posición, lo cual requiere una larga lista de operaciones de bits. Para solucionar esto, se han probado con éxito los **magic bitboards**, que usan una compleja función hash para obtener estos vectores rápidamente.

### 4.2.2. Generación de movimientos

Para buscar el mejor movimiento de una posición, primero debemos generar los movimientos legales. Ésta es la función del generador de movimientos.

Es usual que los generadores hagan su labor en dos partes:

- Primero, crean una lista de todos los posibles movimientos que no se salgan fuera del tablero o pisen a piezas aliadas. Estos movimientos se denominan **pseudolegales**. En el caso de que el rey se encuentre en jaque, sólo cuentan aquellos movimientos que despejen la amenaza.
- Despues, es necesario invertir un tiempo extra asegurando de que, después del movimiento, el rey del bando que mueve no queda en jaque. Esto es importante para eliminar los movimientos pseudolegales de las **piezas clavadas**, o los movimientos ilegales del rey.

La implementación de estos dos pasos depende en gran medida de la representación del tablero que se use, y la eficiencia con la que se ejecutan es clave para aumentar la cantidad de posiciones que el motor puede analizar en un determinado periodo de tiempo.

### 4.2.3. Función de evaluación

La función de evaluación es una función heurística que intenta aproximar las posibilidades de victoria de una posición. Usualmente, su función es ser una guía para los métodos de búsqueda, para encontrar posiciones prometedoras y descartar otras sin esperanza. La función ideal sería aquella capaz de decir si una posición es una victoria asegurada, derrota asegurada o empate.

$$F(pos) = r \in \{Win, Loss, Draw\}$$

En la práctica, tener esta función es imposible, por lo que se debe realizar una aproximación.

Anteriormente, las funciones de evaluación eran hechas a mano, es decir, se basaban en factores que se creían importantes. Normalmente, estas funciones tenían la forma de una suma ponderada de los valores de estos factores junto con la importancia que se creía que tenían:

$$F(pos) = \sum_{i=1}^N f_i(pos) * w_i$$

Los factores que se tomaban en cuenta eran numerosos y el valor de sus pesos cambiaban de motor a motor. Algunos de estos factores son los siguientes:

- Diferencia de material entre los bandos
- Estructura de peones
- Seguridad del rey
- Motivos estratégicos como control del centro del tablero, movilidad, espacio y tiempo
- Calidad de las piezas (piezas atrapadas o que valen más de lo usual por la posición concreta)

Además, el valor de estos factores se cambiaba también dependiendo de la fase del juego (apertura, medio juego o final), y cómo se determinaba la fase también cambiaba de motor a motor.

Hoy en día, la mayoría de motores de ajedrez han sustituido este complicado diseño por una red neuronal multicapa, que es capaz de aprender todos estos factores por sí mismo, sin ningún tipo de intervención humana.

#### 4.2.4. Exploración del árbol de movimientos

Para evaluar y devolver el mejor movimiento en ajedrez, es necesario observar más allá y explorar las posibilidades que se le da al oponente. Los posibles movimientos de una posición de ajedrez se pueden representar en forma de árbol, el cual se denomina **árbol de movimientos**. En él, cada nodo es una posible posición, las transiciones entre nodos son movimientos de los jugadores, y el nodo raíz es la posición actual. El bando que le toca mover en cada nodo depende de la profundidad, y se van alternando cada vez que se profundiza. Los nodos hoja son aquellas posiciones con un resultado, ya sean tablas o un jaque mate.

Como es natural, los motores quieren explorar este árbol lo mejor posible para encontrar las líneas de movimientos más beneficiosas. La situación ideal sería explorar todo el árbol de decisiones, pero con un factor de ramificación medio de 35 (que significa que de media hay 35 posibles movimientos en una posición [34]), el número de nodos se vuelve inalcanzable con su crecimiento exponencial. Lo mejor que se puede hacer es explorar una parte de él, y esperar que esa información sea suficiente para elegir el mejor movimiento.

Muchos esfuerzos han sido focalizados en encontrar formas de hacer más eficiente este proceso de exploración. Una de esas formas es explorar únicamente los nodos más útiles, es decir, hacer una **búsqueda selectiva**. Para

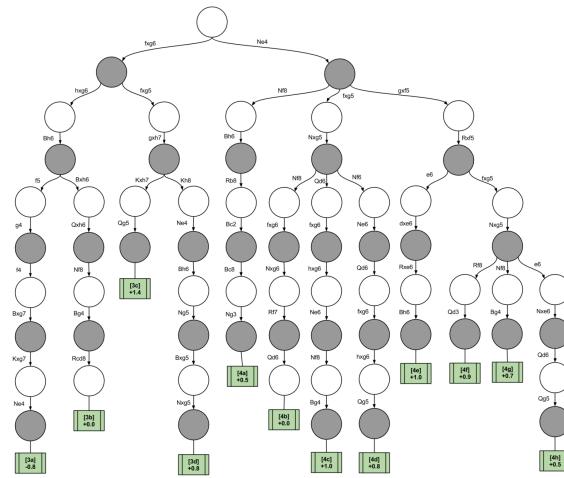


Figura 4.4: Representación del árbol de movimientos, parcialmente explorado [4]

ampliar o recortar la búsqueda, se usan tres tipos de métodos principalmente:

- **Podas** del árbol, que eliminan directamente una ramificación que no consideran útil,
  - **Reducciones**, que limitan la exploración de una rama ya sea por profundidad o número de nodos, y finalmente
  - **Extensiones**, que aumentan la búsqueda en ramas especialmente interesantes.

## Búsqueda en profundidad

El orden en el que explorar el árbol de movimientos es bastante importante. La búsqueda en profundidad prioriza profundizar en una rama antes que saltar a otra, si es posible alcanzando un nodo hoja. Cuando se llega a la profundidad deseada, la ramificación que se explora es la primera que se encuentra al ir hacia atrás en el camino tomado.

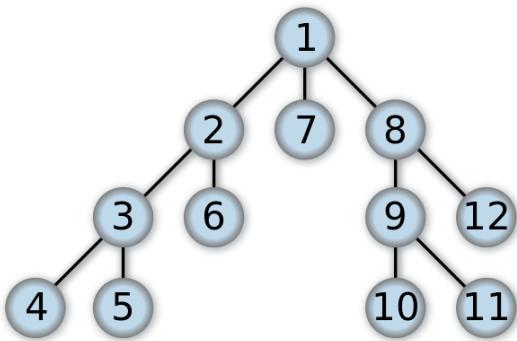


Figura 4.5: Orden de exploración en profundidad [13]



## Capítulo 5

# Aprendizaje Automático

El aprendizaje automático es un subcampo de las ciencias de la computación cuyo objetivo es desarrollar técnicas y algoritmos para que las máquinas aprendan a solucionar un problema a partir de datos del mismo.

Una definición más correcta es la que da Thomas Mitchell en uno de sus libros, *Machine Learning*: un programa informático se dice que aprende de la experiencia (E) con respecto a una clase de tarea (T), con una medida de efectividad (P), si su efectividad en la tarea T, medida por P, mejora con la experiencia E [25].

En este capítulo expondremos los fundamentos del aprendizaje automático, poniendo especial énfasis en los modelos de redes neuronales y las técnicas de clústering, usadas en este proyecto.

### 5.1. Partes de un problema de aprendizaje automático

Siguiendo la definición de Mitchell, podemos encontrar ciertas partes comunes a todos los problemas de Machine Learning.

#### 5.1.1. La Tarea T

La tarea define el objetivo que queremos alcanzar, y la elección de las demás partes del aprendizaje se hacen acorde a ella. Por eso, se puede decir que es la parte más importante a definir.

El aprendizaje automático está mejor situado para tareas que sabemos que se pueden resolver, pero que no podemos crear una lista de pasos de cómo hacerlo. Un claro ejemplo es la tarea de clasificar imágenes según aparezca

un perro o un gato. Sabemos identificar gatos y perros, pero no sabemos identificar qué pasos seguimos para ello y, por tanto, no podemos hacer un programa clásico que lo resuelva.

La lista de tareas que encajan en la descripción anterior y en las que el machine learning ha obtenido buenos resultados es muy larga. Algunos ejemplos comunes son:

- **Regresión:** El objetivo es predecir un valor numérico según una serie de entradas. La función que se le pide al algoritmo de aprendizaje es de la forma:

$$f : \mathbb{R}^d \longrightarrow \mathbb{R}$$

- **Clasificación:** En estas tareas se quieren clasificar las entradas en diferentes categorías establecidas de antemano. El número de categorías cambia según el problema.

Siendo  $n$  el número de categorías, la función output del aprendizaje será de la forma:

$$f : \mathbb{R}^d \longrightarrow \{1, 2, \dots, n\}$$

- **Agrupamiento o Clustering:** Una tarea muy similar a las de clasificación, con la diferencia de que las categorías no se saben de antemano. El algoritmo de aprendizaje agrupa los ejemplos que tiene en el entrenamiento, y clasifica el input en el grupo que más se parezca.

En la mayoría de algoritmos, el número de categorías a deducir se decide antes del entrenamiento, con excepción de algunas formas de clustering.

### 5.1.2. La experiencia E

La experiencia sobre nuestra tarea es lo que hace posible el aprendizaje. La cantidad y calidad de ésta suele ser el factor limitante en cuanto a la calidad de la solución que podamos obtener.

Solemos referirnos a esta experiencia como los datos que tenemos del problema y, a cada ejemplo individual de los datos, **instancias**. Cada instancia está formada por uno o más valores atómicos que representan diferentes medidas, los cuales se llaman **características**.

Los tipos de datos que tenemos modelan el tipo de aprendizaje que se realizará. Los principales tipos de enfoques son:

- **Aprendizaje supervisado:** En este enfoque, cada instancia tiene asociada un dato extra llamado **objetivo o etiqueta**. El algoritmo

de aprendizaje deberá encontrar las relaciones que hay entre el conjunto de datos y las etiquetas para predecir las mismas según las otras características.

Algunos ejemplos de aprendizaje supervisado son la regresión y la clasificación, discutidos en el apartado anterior.

- **Aprendizaje no supervisado:** Ahora sólo tenemos las instancias de datos, pero ninguna etiqueta. En estos casos, los algoritmos estudian las propiedades de los datos, como su estructura interna o las dependencias que tienen entre sí.

Algunos ejemplos son el clustering, la reducción de dimensionalidad de datos o la detección de anomalías.

- **Aprendizaje por refuerzo:** La experiencia del problema se obtiene durante el aprendizaje, es decir, se aprende del *feedback* que devuelve el entorno donde se trabaja. Este aprendizaje de tipo ensayo-error se utiliza en campos como en la robótica, donde sensores recogen información del mundo exterior y se cambia el comportamiento según ello; y también en los juegos como el ajedrez, donde el resultado de la partida se usa para recompensar o castigar la forma de jugar.

Hay más tipos de aprendizajes, y además no son excluyentes entre sí: hay soluciones que combinan algunos de esos tipos.

### 5.1.3. La medida de efectividad P

Los algoritmos de aprendizaje automático necesitan una medida cuantitativa para saber la calidad de las hipótesis que genera y poder comparar unas con otras. Si no la tuviésemos, no sabría qué tipo de función es buena para resolver la tarea que nos planteamos resolver. De aquí viene la importancia de la medida de efectividad: es la manera que tenemos de decirle al algoritmo de aprendizaje qué soluciones consideramos como buenas. Hay muchos problemas donde saber qué soluciones queremos es trivial: en clasificación queremos acertar en la clase del input el mayor número de veces, y en regresión queremos acercarnos lo máximo posible al valor etiquetado en los datos. En estos casos la medida viene ya con el problema, para clasificación normalmente usamos *accuracy* (excepto cuando se quiere evitar desbalanceo de clase, que podemos recurrir a medidas como la *f1*) y en el caso de regresión, *MSE*.

Pero hay otros tantos problemas donde es muy difícil saber exactamente qué nos mide la calidad de una solución. Un ejemplo con el que vamos a tratar es el problema del *clustering*, donde podemos llegar a agrupar los datos según una lista interminable de medidas cuantitativas de similitud.

Saber cuál será la que nos aporte los resultados que queremos será a veces resultado de prueba y error.

#### 5.1.4. Modelos de aprendizaje

Un **modelo de aprendizaje** es un candidato a solución de la tarea a resolver, es decir, es el resultado del aprendizaje. Para producir un modelo de calidad, se necesitan escoger dos elementos: una **clase de funciones** que contenga todos los modelos que consideremos posibles soluciones, y un **algoritmo de aprendizaje** que sea capaz de buscar en la clase de funciones un candidato al problema usando los datos y la medida escogida.

El conjunto de datos que caracterizan cada función dentro de su clase se llaman **parámetros** o **pesos**, y es sobre esos valores en los que el algoritmo explora la clase de funciones. Un ejemplo es la clase de las parábolas, que tienen una representación matemática de  $ax^2 + bx + c$ . Para describir una parábola en concreto, sólo debemos dar los valores de las variables  $a$ ,  $b$  y  $c$ , por lo tanto dicha clase de funciones tiene 3 pesos. Todo algoritmo de aprendizaje automático se dedica, de una forma u otra, a optimizar esos pesos a la tarea a solucionar, y la cantidad de parámetros es un buen indicador de la **complejidad** de la clase de funciones, que afecta de muchas maneras al aprendizaje.

A parte de los parámetros, hay una serie de valores que también deben ajustarse correctamente para conseguir un buen modelo. Éstos son los **hiperparámetros**, que ajustan las clases de funciones y los algoritmos para que funcionen mejor para las tareas concretas. La diferencia fundamental entre los hiperparámetros y los parámetros son que los primeros **no se modifican al entrenar**.

## 5.2. Generalización

Lo que diferencia un problema de aprendizaje automático de la regresión matemática es la necesidad de ajustar correctamente nuevos valores nunca vistos, no sólo con los datos con los que fue entrenado el modelo. A esta capacidad de predecir cualquier tipo de entrada se le denomina **capacidad de generalización**.

La medida de generalización ideal sería el **error fuera de la muestra**, que mediría el rendimiento del modelo según todas las posibles entradas del problema, lo cual suele ser imposible de saber.

### 5.3. Redes neuronales

Las redes neuronales o *perceptrones multicapa* son modelos de aprendizaje que han mostrado tener una cantidad de aplicaciones muy grande. Son la base de la rama del *deep learning*, y las redes neuronales convolucionales, un tipo específico de red neuronal, han sobrepasado a todos los otros modelos en cuanto a reconocimiento de patrones y otras aplicaciones importantes.

El nombre de redes viene dado por la estructura interna que tienen, ya que son una composición de funciones estructuradas en capas de *neuronas*, en las cuales se usa el output de una como input de la otra. El número de capas del modelo se llama **profundidad**, y es el origen del *deep learning*. En esta modalidad, se hacen uso de estructuras muy profundas, las cuales obtienen una complejidad enorme y son capaces de aproximar casi cualquier tipo de función.

La parte de *neuronal* viene dado por la unidad más básica del modelo: el perceptrón o neurona, cuyo nombre tiene origen en que se inspiraron en el cerebro humano a la hora de crearlo.

#### 5.3.1. Neurona o perceptrón

Cada neurona de la red tiene la misma estructura que una neurona animal: coge los distintos estímulos o inputs que recibe y los procesa, generando un único output. La inspiración biológica era más aparente en su creación, actualmente tiene poca similitud.

La estructura de una neurona puede resumirse en este esquema:

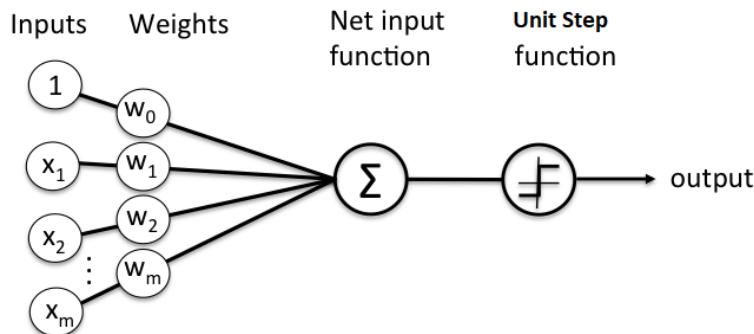


Figura 5.1: Diagrama de una neurona artificial[28]

Primero se multiplican las entradas o **inputs** con el vector de **pesos**, que son valores propios de la neurona que se actualizan durante el entrenamiento. Despues, se suman todos los valores y se envían a la **función de activación**, y el resultado se toma como la salida de la neurona.

Se han probado muchas funciones de activación durante el tiempo. La primera de ellas fue la que dió origen al *perceptrón*:

$$s(x) = \begin{cases} 1 & \text{si } x > 0 \\ -1 & \text{si } x < 0 \end{cases}$$

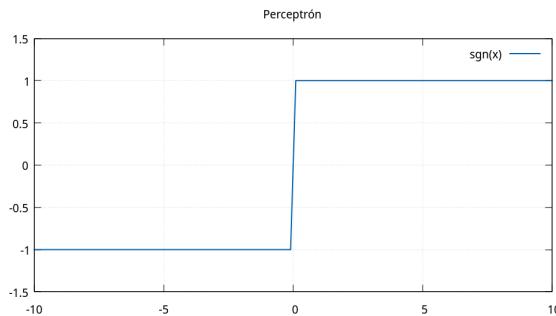


Figura 5.2: Función de activación del perceptrón

El problema que tiene y el motivo por el que ya no se use es el mal comportamiento que tiene su derivada. La función no es derivable en el 0, y para el resto de valores de  $x$ , su derivada es 0. Esto hace que no sea posible usar el mejor algoritmo que conocemos para entrenar redes neuronales, el gradiente descendiente.

Por ello se han encontrado mejores funciones de activación, aunque más complejas. La primera de ellas es la función sigmoide:

$$s(x) = \frac{1}{1 + e^{-x}}$$

Otra muy similar es la de la tangente hiperbólica:

$$s(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Estas funciones tienen a su favor, aparte de ser parecidas a la activación del perceptrón, que son infinitamente derivables en los reales, es decir, pertenece a la clase  $C^\infty(\mathbb{R})$ . La diferencia más significativa entre las dos es su imagen: la sigmoidal obtiene valores en el intervalo  $(0, 1)$ , mientras que la tangente hiperbólica, en  $(-1, 1)$ .

Otra familia de funciones de activación que se están usando recientemente son las ReLU, que consisten en eliminar (o reducir en gran medida) los valores negativos.

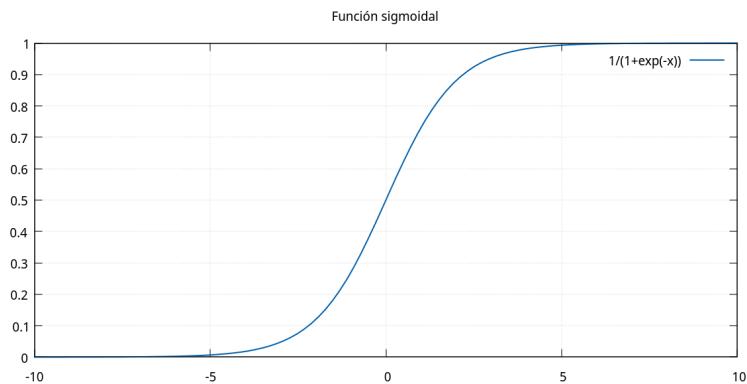


Figura 5.3: Función de activación sigmoidal

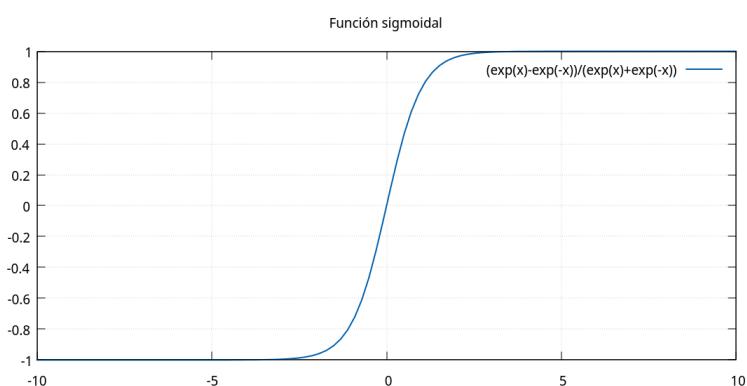


Figura 5.4: Función de activación tangente hiperbólica

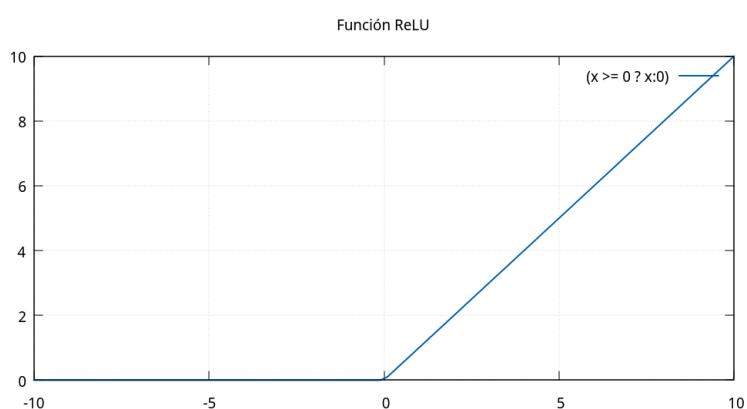


Figura 5.5: Función de activación ReLU

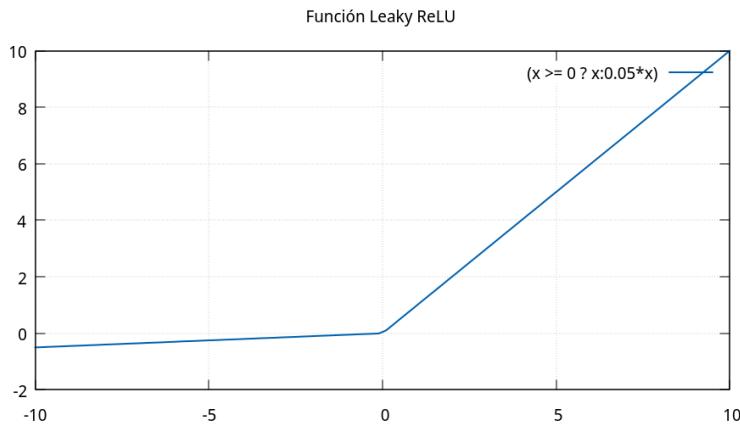


Figura 5.6: Función de activación Leaky ReLU

Estas funciones no son infinitamente derivables. Aun así, por la simplicidad que tienen y por el hecho que es derivable a trozos, se pueden usar con el algoritmo de aprendizaje después de algunas optimizaciones.

Algo que tienen en común todas estas funciones es que ninguna de ellas es lineal. Por ello es tan importante esta función en las redes neuronales, porque es la única parte que introduce no-linealidades y introduce información. Además, se ha demostrado que una red neuronal de  $n$  capas con una función de activación lineal es completamente equivalente a otra con una sola capa. A cambio, en el caso no lineal, se ha demostrado que cuando  $n$  tiende a infinito, la red neuronal es un mecanismo de aproximación universal de funciones.

### 5.3.2. Estructura

Las redes están formadas por capas de neuronas de tamaño variable. En los casos que usaremos, estas capas se conectan de forma secuencial, de forma que el output de las neuronas de una pasa a ser el input de las neuronas de la siguiente. La primera capa del modelo es la capa input (input layer), que contiene los datos de entrada aun sin transformar, mientras que la última es la capa output (output layer), la salida de nuestra red neuronal. Al resto de capas intermedias se les llama capas ocultas (hidden layers).

Este tipo de estructuras son las más comunes y sencillas. Hay muchos más tipos de estructuras de conexión, y se pueden complicar todo lo que nosotros queramos: redes con conexiones recursivas, conexiones que se saltan capas, conexiones capa-capá restringidas... El caso es que no haremos uso de ellas.

En la sección anterior dijimos que cada input de cada neurona tiene un peso asociado. Entonces supongamos que tenemos la primera capa con 200

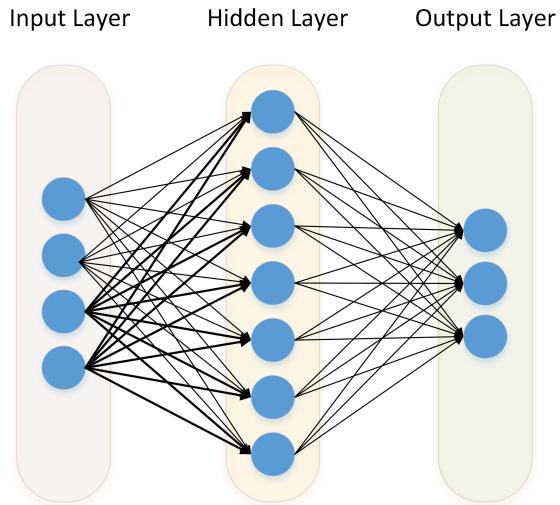


Figura 5.7: Conexiones entre las capas de neuronas

neuronas, y la segunda con 400. Al conectar las dos capas, obtendremos un total de  $200 \times 400 = 80.000$  pesos a aprender. Esto hace que sea peligroso aumentar la cantidad de neuronas por capa, ya que aumenta de forma multiplicativa los valores a actualizar en el aprendizaje. En cambio, el número de capas aumenta los pesos de forma lineal, por lo que parecería una buena idea expandir usando esta dimensión. El problema de esto es que aumentas en gran medida la complejidad de la clase de funciones que representa la red, lo cual tiene consecuencias en cuanto a la facilidad de sobreajuste y el margen de diferencia teórico entre el error en muestra y el error fuera de la muestra.

### 5.3.3. Backpropagation

El gradiente descendiente es un algoritmo iterativo de minimización de funciones multivariable derivables. La idea básica es la siguiente: calcular el vector gradiente de la función objetivo, que es la dirección de máximo crecimiento, y usar la dirección contraria a él para cambiar los parámetros/pesos/argumentos de la función hacia dicha dirección.

Para las redes neuronales, el cálculo del gradiente del error según los pesos se deja a otro algoritmo, llamado backpropagation, que usa principalmente la regla de la cadena para calcular de forma eficiente las derivadas parciales de la NN según los pesos de cada neurona.

Para explicar su funcionamiento, llamaremos  $o_j$  al output de una neurona cualquiera  $j$ ,  $w_{kj}$  a los pesos de dicha neurona y  $x_i$  a todos los outputs de las neuronas de su capa anterior. Si la neurona  $j$  se encuentra justamente en

la primera capa oculta,  $x_i$  serían los inputs de la red neuronal. Finalmente,  $\varphi$  será la función de activación de la red y  $E(y, t)$  la función de error, donde  $y$  es el output final de la red y  $t$  el valor esperado. Introducimos la variable  $net_j$  para facilitar la notación:

$$o_j = \varphi\left(\sum_{k=1}^n w_{kj}x_k\right) = \varphi(net_j)$$

Nuestro objetivo es obtener una expresión lo más sencilla posible para  $\frac{\partial E}{\partial w_{ij}}$ . Usando dos veces la regla de la cadena, obtenemos:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

Ahora vamos a desarrollar las tres derivadas parciales de la última igualdad. Empezamos con la última, que es muy sencilla:

$$\frac{\partial net_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\sum_{k=1}^n w_{kj}x_k\right) = \frac{\partial(w_{ij}x_i)}{\partial w_{ij}} = x_i$$

El segundo factor de la igualdad es, simplemente, la derivada de la función de activación evaluada en el valor  $net_j$ ,

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \varphi(net_j)}{\partial net_j} = \varphi'(net_j)$$

Y finalmente, el primer factor, que es el complicado. Si la neurona está en la capa output, entonces es bastante sencillo ya que  $E$  (el error) dependerá directamente de  $o_j$

$$\frac{\partial E}{\partial o_j} = \frac{\partial E(y, t)}{\partial y_j}$$

lo cual es fácil de obtener si se usa una de las funciones de error usuales en las NN (error cuadrático medio, error cross-entropy, etc.).

En cambio, si la neurona se encuentra en una capa oculta, la función de error sigue dependiendo de  $o_j$ , sólo que de forma menos directa. Para verlo, tenemos que usar la regla de la cadena como al principio:

$$\frac{\partial E}{\partial o_j} = \sum_{k \in M} \left( \frac{\partial E}{\partial net_k} \frac{\partial net_k}{\partial o_j} \right) = \sum_{k \in M} \left( \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial net_k} \frac{\partial net_k}{\partial o_j} \right) = \sum_{k \in M} \left( \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial net_k} w_{jk} \right)$$

donde  $M$  son las neuronas de la capa siguiente a la neurona  $j$ . De aquí podemos sacar la fórmula recursiva final del gradiente del error:

$$\frac{\partial E}{\partial w_{ij}} = o_j \delta_j$$

donde

$$\delta_j = \begin{cases} \frac{\partial E}{\partial o_j} \varphi'(net_j) & \text{si } j \text{ es una neurona output} \\ \varphi'(net_j) \sum_{k \in M} \delta_k w_{jk} & \text{si } j \text{ es una neurona oculta} \end{cases}$$

El nombre de este algoritmo hace referencia a cómo se usa esta fórmula matemática: se obtiene el gradiente según los pesos de las neuronas de la capa final para después "propagarse" hacia atrás con esa recursión. Esto es fácilmente implementable aunque no lo parezca con su complejidad matemática, usando programación dinámica.

#### 5.3.4. Gradiente descendiente: tipos e hiperparámetros

El gradiente descendiente es un algoritmo muy sencillo de base. Para las redes neuronales, su esquema básico es el siguiente:

```

if  $i \geq maxval$  then
     $i \leftarrow 0$ 
else
    if  $i + k \leq maxval$  then
         $i \leftarrow i + k$ 
    end if
end if
```

---

#### Algorithm 5.1 Gradiente descendiente

---

```

for  $i$  in  $1, 2, \dots, n\_epochs$  do
    for all  $x, t$  in data do
         $y \leftarrow \text{NN}(x)$ 
        gradiente  $\leftarrow \text{backpropagation}(y, t)$ 
        for all  $n$  in neuronas do
             $n.\text{pesos} = n.\text{pesos} - \theta \text{gradiente}[n.\text{pesos}]$ 
        end for
        actualizar  $\theta$ 
    end for
end for
```

---

Cada vez que se itera sobre todos los datos de aprendizaje se le llama *epoch*, y la longitud del aprendizaje de la red neuronal se controla con el número de epochs.

La parte más importante es el parámetro  $\theta$ : se le suele llamar *learning rate*, y es un escalar que determina cuánto se debe seguir la dirección del

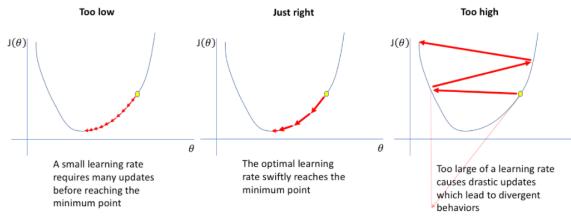


Figura 5.8: Conexiones entre las capas de neuronas

gradiente descendiente. La parte difícil de este algoritmo es usar correctamente este parámetro, y hay muchas formas para ello que funcionarán mejor o peor según el caso específico con el que se trabaje. Si el learning rate es demasiado grande, el algoritmo no convergerá a ninguna solución, mientras que si es demasiado pequeño se quedará atascado en un mínimo local que quizás no sea el más óptimo, además de necesitar más iteraciones de las óptimas.

Antes de darle un valor al parámetro, debemos decidir si queremos que éste siga constante en todas las iteraciones (poco común), o si mejor que vaya cambiando. Esta es la denominada política del learning rate, y no hay ninguna que funcione mejor que otra, aunque las más populares tienen ciertas similitudes entre sí. Generalmente, es mejor empezar con un valor grande para que pueda saltar mínimos locales poco óptimos, e ir reduciéndolo a medida que el aprendizaje avanza. Algunas de las políticas más usadas son:

- **Basados en iteraciones:** El learning rate se divide por una constante entre iteraciones o epochs. Añade otro hiperparámetro, el *decay*, que indica la cantidad por la que se divide.  $\theta_{n+1} = \frac{\theta_n}{1+d}$ .
- **Caída exponencial:** Similar al anterior, la velocidad del aprendizaje decae suavemente según una función exponencial.  $\theta = \theta_0 \exp -kt$ , donde  $\theta_0$  y  $k$  son hiperparámetros, y  $t$  es el número de epochs o iteraciones.
- **Basados en escalones:** En este caso, se divide el learning rate por una cantidad importante cada cierto número de epochs. Los cambios son más bruscos y menos frecuentes que la política anterior, y en una gráfica toma forma de escalones, de ahí su nombre.
- **Basados en ciclos:** Un método novel para este hiperparámetro es el basado en ciclos [37, pág. 1]. Aquí, se aumenta y reduce el valor del learning rate una y otra vez, siguiendo un patrón de onda triangular. Este método ha mostrado mejores resultados que el resto a lo largo del aprendizaje, aunque degrada temporalmente la calidad del modelo en

los picos de los valores. Añade dos hiperparámetros extra: la longitud de los ciclos y el rango de valores a tomar.

- **Basados en inercia:** Análogo a una pelota bajando una colina, el learning rate aumenta cuando el gradiente apunta a la misma dirección durante algunas iteraciones o epochs. Esta política es combinable con muchas otras, y añade un sólo hiperparámetro: la masa de la "pelota", que indica cuánto debe aumentar el aprendizaje al seguir la misma dirección.
- **Métodos adaptativos:** Aquí, la velocidad de aprendizaje se adapta durante el aprendizaje siguiendo ciertos valores heurísticos. Algunos ejemplos que han tenido bastante éxito son Adagrad, Adadelta, RMSprop y Adam.

## 5.4. Clustering

El **clustering** es la tarea de agrupar una serie de objetos de forma que objetos similares se encuentren en el mismo grupo, mientras que objetos diferentes estén en grupos diferentes [33, cap. 22]. Es un problema muy importante debido a su uso en muchos y diversos campos, como minería de datos, reconocimiento de patrones o bioinformática [23].

Al trabajar con clustering, se nos presentan varios problemas. El primero de ellos es que puede haber contradicciones entre los dos objetivos de la definición. La similitud entre objetos no suele ser transitiva, es decir, si  $A$  es similar a  $B$  y  $B$  similar a  $C$ , eso no quiere decir que  $A$  se asimile a  $C$ . Como el clustering intenta crear grupos cerrados, ¿cómo podríamos agrupar a  $A, B, C$  en el caso de que  $A$  y  $C$  no se parezcan?

- La partición  $A, B, C$  cumple que los objetos no similares estén en diferentes particiones.
- La partición  $A, B, C$  cumple que los objetos similares estén en la misma partición.

Por este motivo, los algoritmos de clustering intentan cumplir un objetivo u otro. Lo más común es resolver el de los objetos similares, aunque existen algunos del otro tipo.

Otro de los problemas es la falta de un resultado correcto. Hoy en día se usan varios algoritmos de clustering, y cada uno de ellos se basa en una definición diferente de cluster que es válida. Por ello, los resultados pueden ser muy diferentes, y sin embargo estar correctos.

### 5.4.1. K-medias

K-medias es un algoritmo de clustering ampliamente utilizado, que separa los datos en k grupos, donde k es un número predefinido por el usuario. El algoritmo asigna a cada cluster un **centroide**, que se usa para identificar a qué agrupación pertenece cada instancia: pertenece a aquella cuyo centroide asociado se encuentra más cerca.

La medida que error que usa el algoritmo es la **suma de las distancias entre cada dato y su centroide asociado**, y por lo tanto el objetivo del algoritmo es encontrar unos centroides  $\mu_1, \mu_2, \dots, \mu_k$  que generen unas particiones  $S_1, S_2, \dots, S_k$  que minimicen la siguiente formula:

$$\sum_{i=1}^k \sum_{x_j \in S_i} \|x_j - \mu_i\|^2$$

El algoritmo funciona de la siguiente manera:

1. Inicialización: Se seleccionan k puntos aleatorios del espacio de forma aleatoria como los centroides iniciales de los clusters.
2. Asignación de puntos de datos: Cada instancia de los datos se asigna a su cluster, aquel cuyo centroide se encuentre más cerca.
3. Actualización de centroides: Ahora que tenemos los clusters definidos, se recalculan los centroides usando la media de todos los puntos asociados a cada agrupación.
4. Repetición: los pasos 2 y 3 se repiten hasta que se cumpla uno de los dos criterios de parada: o los centroides cambian poco (ajustable con un hiperparámetro), o se llega a un máximo de iteraciones.
5. Resultado: el algoritmo devuelve las particiones finales, y los centroides por si se quieren usar para clasificar nuevos datos.

Este algoritmo, aunque sencillo, es bastante potente; pero tiene ciertas características importantes que se han de tener en cuenta:

- Para obtener un buen resultado, la elección de un buen valor para el hiperparámetro k es necesaria. Mientras que no hay ninguna regla estricta para elegir su valor, se pueden usar algunas heurísticas para ello; pero encontrar el valor óptimo requiere conocimiento del dominio y experimentación.
- El algoritmo no es capaz de trabajar con atributos categóricos, sólo numéricos

- El resultado de k-medias puede variar dependiendo de la inicialización de los centroides. Por ello, se suele ejecutar varias veces con inicializaciones diferentes para obtener una mejor solución.
- K-medias es sensible a datos atípicos (outliers).



# **Parte III**

# **Propuesta**



# Capítulo 6

## Estado del arte

### 6.1. Aprendizaje automático y ajedrez

En esta sección se analizarán algunos de los sistemas de aprendizaje automático que más impacto han tenido en el ajedrez, para así tener una amplia perspectiva de lo que se ha conseguido con estas técnicas. Nos centraremos principalmente en redes neuronales, dada su popularidad en uso en el ajedrez.

#### 6.1.1. DeepChess

DeepChess [14] es un motor de ajedrez publicado en 2016, que usa redes neuronales profundas para aprender a jugar desde zero, sin ningún conocimiento a priori del juego. La red neuronal toma el papel de función de evaluación, con una diferencia importante: la red no otorga una puntuación a una posición, sino que **compara posiciones de ajedrez entre sí**.

La idea es que los motores tradicionales usan la evaluación de una posición como un **representante**, para comparar las posiciones entre sí y saber cuál beneficia más a un bando. DeepChess compara las posiciones directamente entre sí usando su red neuronal, y usan una versión modificada de la búsqueda alfa-beta que almacena **posiciones en vez de evaluaciones** como límites de poda.

La topología de la red neuronal se divide en dos partes:

- **Las torres de autoencoders**, también llamada **Pos2vec**, se encarga de ser un extractor de características no lineal de las posiciones de ajedrez que compara. La red toma como input una representación en 12 bitboards de la posición (uno por pieza y color), y ciertas flags para representar a quién le toca y los enroques legales, haciendo un total

de 773 inputs binarios. Hay 5 capas en total, de tamaños 773 (input), 600, 400, 200 y 100, que usan ReLU como función de activación.

- **Las capas de comparación.** La red final usa dos copias de Pos2Vec, una para cada posición, e introduce 4 capas más para comparar las características de las dos posiciones. Las capas tienen tamaños 400, 200, 100 y 2, usando ReLU como función de activación excepto la última capa, que usa SoftMax. La capa output (la de tamaño 2) muestra la posibilidad de que ganen las blancas en la posición asociada.

Primero se entrena la parte Pos2vec de la red capa a capa, usando aprendizaje no supervisado. Se construye un autoencoder con la primera capa (773-600-773), se entrena con posiciones de ajedrez, se fijan los pesos para que no puedan cambiar más y se añade la siguiente capa (773-600-400-600-773), así hasta la última. Por cada capa, se ejecutan 200 epochs con 2 millones de posiciones.

La segunda parte del entrenamiento es para la red entera, y se hace con aprendizaje supervisado. Los pesos de Pos2Vec se liberan para que pueda seguir aprendiendo, pero las dos copias usan pesos compartidos (es decir, usan los mismos pesos). La red se entrena con 1000 epochs, con 1 millón de parejas de posiciones por cada una. Estas parejas están formadas por una posición de una partida en la que el blanco gana, y otra posición en la que gana el negro. El etiquetado es (1,0) o (0,1) en términos de probabilidad, y la medida de error que se usa es *cross-entropy* (la más usada para clasificación probabilística).

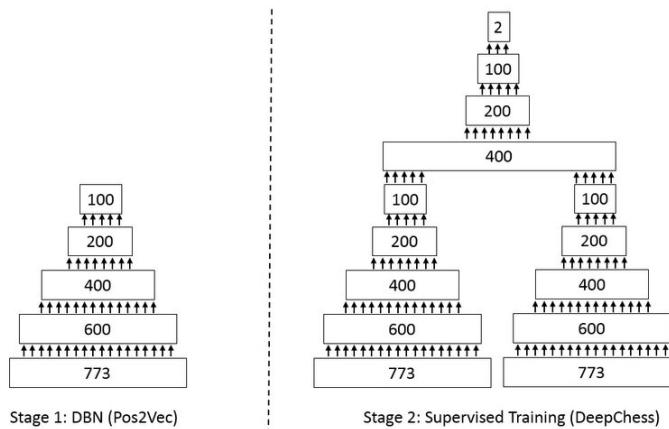


Figura 6.1: Topología de la Red DeepChess [14]

### 6.1.2. Leela Chess Zero

*Leela Chess Zero* es una adaptación al ajedrez del motor *Leela Zero Go* [21] (del juego de mesa Go), que a su vez está basado en las publicaciones de DeepMind de Google sobre el proyecto AlphaZero [36]. El proyecto ha estado liderado por *Gari Linscott*, que también es un desarrollador del motor Stockfish.

Leela Chess Zero, al igual que aquellos proyectos en los que se basa, no contiene conocimiento específico al comienzo (excepto por las normas básicas), pero aprende a jugar bien al ajedrez usando **aprendizaje por refuerzo** jugando contra sí mismo.

La parte de Leela que aprende es su **red neuronal convolucional profunda**, que hace a su vez de función de evaluación, y generador y seleccionador de movimientos. Hacer tantas funciones es posible gracias a su estructura, que es de un cuerpo principal que se encarga de extraer características complejas, y varias cabezas que interpretan esas características de diferentes maneras y hacen la función de output. Las cabezas más importantes son la de la función de evaluación y la del orden de movimientos, que asigna un índice de interés.<sup>a</sup> cada movimiento posible.

El cuerpo es una torre de bloques convolucionales residuales con capas de *Squeeze and Excitation*. Las capas convolucionales son de tamaño 8x8 con número de filtros variable, que usan un kernel 3x3 con stride 1. El número de bloques también es variable. Las variables se introducen para que se puedan entrenar tamaños de redes distintas, dependiendo de las necesidades del usuario (una red grande consume más tiempo y recursos que una pequeña, mientras que aporta un mejor análisis) [16].

Para usar plenamente las sugerencias de movimientos de la red neuronal, Leela Chess Zero usa un método de búsqueda en el árbol de movimientos poco habitual, que es el método de *Monte Carlo* (MCTS) [24]. Este método de búsqueda es **estocástico**, es decir, no realiza las mismas acciones en cada ejecución. La búsqueda de Monte Carlo, para decidir qué nodo explorar, escoge un número aleatorio según **una distribución de probabilidad**, que en este caso se deriva del índice de interés. Cuanto más puntuación tenga un movimiento, más probable es que se explore. Esto da resultado a exploraciones muy desequilibradas, profundizando mucho en ciertas líneas de movimientos.

### 6.1.3. Stockfish y las redes NNUE

Stockfish[39] es un motor gratuito de código abierto con años de historia. Su primera versión fue lanzada en noviembre de 2008, como un derivado de otro motor *open-source*, Glaurung versión 2.1. El motor fue desarrollado

por Tord Romstad, Marco Costalba, and Joona Kiiski, y actualmente lo desarrolla y mantiene la comunidad de Stockfish.

En 2020, se discutió la introducción de una NNUE en el motor, para sustituir la función de evaluación clásica. Ello vino por una modificación de Stockfish creada por programadores de motores shogi en 2019, como una prueba de concepto de que era viable usar las NNUE en el ajedrez.

NNUE significa *Red Neuronal Eficientemente Actualizable*, y fueron inventadas e introducidas en el Shogi por Yu Nasu en 2018 [27]. Estas redes están pensadas para tomar el papel de heurística en la búsqueda alfa-beta, y su principal atractivo es la eficiencia para **recalcular evaluaciones al hacer o deshacer un movimiento**, gracias a que sólo una fracción de las neuronas deben ser recalculadas. Además, como la red requiere de poca computación, sus cálculos son mejor ejecutados **usando instrucciones SIMD, propias de CPUs y no de GPUs**, lo cual hace que funcionen eficientemente en la mayoría de dispositivos.

Las características principales de las arquitecturas NNUE son las siguientes:

- El input está **sobreparametrizado** (introduce más información de la necesaria), para que haya sobreespecialización en los pesos de la primera capa oculta.
- No tienen muchas capas, y esas capas no son muy grandes.

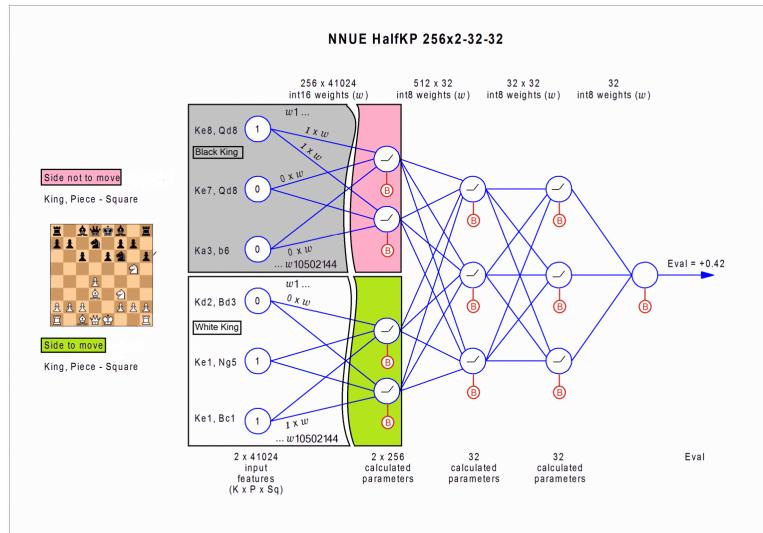


Figura 6.2: Estructura de la primera NNUE de Stockfish, HalfKP [38]

En Julio del mismo año, un mes después de la discusión, se decidió crear una versión NNUE de Stockfish con el código actual. En agosto, Stockfish

NNUE era más fuerte que el Stockfish clásico por, al menos, 80 puntos de ELO. Finalmente, en Septiembre se publicó Stockfish 12, que usaba por defecto una NNUE para evaluar y que dio un salto de nivel enorme en comparación con su versión anterior.

Actualmente, Stockfish 15.1 usa una NNUE llamada HalfKAv2, que es entrenada de forma distribuida por voluntarios.

## 6.2. Herramientas para el entusiasta

El ajedrez es un juego muy popular, que se enseña en países de todo el mundo, lo cual le convierte en un mercado muy grande. Muchos jugadores quieren mejorar su nivel y llegar más alto, ya sea por competitividad o por pura diversión. Ahora exploraremos aquellas herramientas y servicios que se encuentran disponibles para el entusiasta de ajedrez.

### 6.2.1. Aimchess

Aimchess [31] es una plataforma digital que usa inteligencia artificial para analizar los juegos online del aficionado y crear un programa de entrenamiento que se adapta a sus debilidades y fuerzas.

Para los juegos online, Aimchess se puede conectar a cuentas de Chess.com, Lichess o Chess24, y obtenerlos directamente de ahí. Después de analizarlos, la plataforma es capaz de crear un informe detallado sobre la fuerza del jugador en distintos aspectos del juego, como la apertura, el manejo del tiempo, tácticas o finales.

Las lecciones que proporciona son muy variadas, desde prevención de errores grandes o mejora de la intuición, hasta tácticas y finales. Dichas lecciones se personalizan con las partidas del jugador, para que aprenda de sus errores.

Aimchess proporciona dos tipos de planes: uno completamente gratuito con ciertas limitaciones, y uno de pago. El programa de pago incluye:

- Informes ilimitados de partidas,
- Lecciones de entrenamiento ilimitadas,
- Una página de estadísticas sobre el propio nivel del jugador,
- Compartir información y lecciones con un entrenador de ajedrez si se tiene uno,
- Conectarse con varias cuentas de plataformas de juego online.

### **6.2.2. Chess.com**

Chess.com [32] es una página de juego online con muchas herramientas para mejorar al ajedrez. Es una plataforma todo en uno, donde se pueden entrenar puzzles de ajedrez, revisar partidas anteriores y jugar partidas a diferentes tiempos, todo sin salir de la página web.

Aunque la plataforma es gratuita, algunas herramientas están limitadas o bloqueadas a no ser que se tenga una cuenta premium. Aun así, todos tienen acceso a tres rompecabezas diarios (donde se tiene que encontrar la variación ganadora), y pueden usar su intuitivo analizador de partidas una vez al dia, entre otras muchas cosas.

El punto fuerte de esta plataforma es que te permite estudiar una partida justo después de jugarla, de forma que puedes ver con ayuda de un motor y casi al momento tus fallos, antes de que se olviden.

# Capítulo 7

## Propuesta

### 7.1. Análisis del problema

El objetivo de este TFG es clasificar errores de ajedrez usando técnicas de aprendizaje no supervisado, es decir, realizar técnicas de clustering sobre movimientos concretos de ajedrez.

La mejor parte del clustering es los pocos requisitos que pide: un criterio para diferenciar los elementos individuales, para saber si se deben considerar similares o no. Aun así, esto no es nada sencillo de realizar para movimientos concretos en cierta posición, ni siquiera para posiciones de ajedrez.

La forma más común para dar un criterio entre elementos para clustering es definiendo **una distancia** entre ellos, y ésto se suele hacer eligiendo una serie de características (numéricas o categóricas) que se creen útiles para diferenciar los elementos que queremos separar. El conjunto de estos valores es llamado **vector de características**, y cuando se tiene elegido sólo queda escoger una distancia entre vectores (Euclídea, Manhattan, etc.) y escoger la importancia que tiene cada una para darles cierto peso. Por ejemplo, para clasificar relojes se puede medir el tamaño, el peso, el precio y si es de pulsera o de pared; y después normalizar los valores y usar la distancia euclídea entre ellos.

En cuanto al problema específico de los errores de ajedrez, hay algunas obvias: las piezas que quedan en el tablero, el valor medio de las mismas, la cantidad de columnas abiertas, la seguridad del rey o la diferencia de evaluación entre la posición antes del movimiento y después. Pero como sucedió con las funciones de evaluación de los motores clásicos, es complicado darle una importancia a cada uno de los factores, o incluso comparar posiciones de medio juego y final.

Esta situación es similar a las funciones de evaluación clásicas de los

motores de ajedrez: hay muchos factores complejos, cuya importancia relativa no está clara. Así que, siguiendo la analogía, queremos usar técnicas de aprendizaje automático para solucionar este problema y elegir las características y su importancia, como hicieron las NNUE.

Después de decidir las características que se usarán, queda realizar el análisis de clustering, es decir, observar y analizar qué estructura terminan teniendo estos datos y ver qué información podemos extraer. Los datos no serán el factor limitante en este caso, ya que las bases de datos de partidas de ajedrez son enormes. Los límites de la calidad del clustering serán la capacidad de cómputo y memoria y, por supuesto, cómo se adaptan dichos datos al algoritmo.

## 7.2. Base de datos inicial

La base de datos que se ha elegido es una derivada de la base de partidas de Lichess. De todas las partidas jugadas en su plataforma online desde enero de 2013 hasta febrero de 2022, se han extraído las partidas que cumplen los siguientes requisitos: cada jugador debe tener una calificación mínima de 2200 en la plataforma, y la media de calificación de los dos jugadores debe sobrepasar los 2400.

El nivel de los jugadores se ha limitado para reducir la cantidad de tipos de errores con los que tratar, y se ha reducido hacia valores superiores para eliminar aquellos errores obvios como dejarse una pieza o no defender un jaque mate en un movimiento. Aun así, en el desarrollo del proyecto se ha observado que quedan errores de este tipo. Una teoría de ello es que, en un número importante de partidas, uno o ambos jugadores tenían poco tiempo restante para realizar los movimientos.

Hay que tener en cuenta que la calificación de Lichess no refleja el elo real de los jugadores ya que la plataforma usa un sistema de puntuación GLICKO 2 en vez del sistema ELO usual, lo cual complica comparar las puntuaciones. Aun así, los ratings de Lichess están más inflados que el resto de valores, por lo que es muy probable que haya partidas de jugadores con menos de 2000 de ELO en la base de datos [22].

El formato inicial de la base de datos extraída fue CBV, un formato creado por ChessBase para sus programas. Este formato no tiene buen soporte si no se usa software de ChessBase, por lo que se transformó a formato PGN, un formato de almacenamiento de partidas en texto plano.

El tamaño final de la base de datos es de 12,184,568 partidas (un 0.4 % de la cantidad inicial).

### 7.3. Estructura del trabajo

El workflow del proyecto se ha separado en tres partes:

- **Análisis de posiciones para obtener errores:** Usando varias instancias de un motor de ajedrez en paralelo, podemos extraer los movimientos de una partida de ajedrez que consideramos errores aprovechando todos los recursos disponibles. Primero, explicaremos nuestra definición de error, la estructura del programa y las herramientas usadas. Después, estudiaremos el rendimiento usando diferentes configuraciones de análisis.
- **Extracción de características usando un modelo ya entrenado de DeepChess:** Extraeremos un vector de características de una capa de neuronas intermedia en un modelo preentrenado de DeepChess. Primero, haremos una lista de los modelos candidatos para ser escogidos y veremos con experimentos cuál es el más indicado para ello.
- **Análisis de clusters con el algoritmo k-medias:** Haremos particiones de datos usando el algoritmo de clustering k-medias. Investigaremos cuál es el número de clusters indicado para los datos que tenemos, y extreremos la máxima información posible.

Cada una de estas partes tiene su propio capítulo donde se entra en detalle. El motivo de esta separación es la total independencia que tiene cada parte en el workflow del proyecto y lo diferente que son entre sí. El desarrollo de cada parte se ha hecho también de forma independiente, la única comunicación que hay entre ellas son los archivos input/output que se generan de una parte y son usados por la otra.

### 7.4. Herramientas utilizadas

El proyecto se ha realizado en **python**, en la versión 3.11, principalmente por la cantidad de librerías disponibles para él, por ser el lenguaje más usado para aprendizaje automático y porque es lo que se usa en las implementaciones de DeepChess. Las librerías que se han usado para python son:

- NumPy y SciPy, para el manejo de arrays multidimensionales
- Pandas, para usar arrays multi-tipo y trabajar con ficheros CSV.
- Python-Chess para todo tipo de herramientas de ajedrez, desde codificaciones hasta comunicación con los motores de ajedrez,

- AsyncIO para comunicación asíncrona con los motores y paralelización,
- Tensorflow en varias versiones, para tratar con las redes neuronales
- TfDeploy, para cargar redes neuronales de TensorFlow antiguas,
- Scikit-Learn para los algoritmos de clustering

También se hizo uso de Anaconda, para crear ecosistemas virtuales de Python para probar algunas implementaciones de DeepChess antiguas.

Durante un tiempo, también se usó C++ con la familia de librerías Boost para la extracción de errores, pero fue abandonado al ver el amplio soporte de herramientas de ajedrez en Python.

El motor para evaluar las posiciones será Stockfish, versión 15.1. La decisión viene determinada por una serie de factores:

- Stockfish 15.1 es uno de los motores más fuertes disponible actualmente.
- Stockfish es completamente open-source (licencia GPL v3)
- Stockfish usa el protocolo UCI para comunicación, y por tanto tiene buen soporte del software de ajedrez.
- La única red neuronal que usa es una NNUE, la cual está pensada para ser eficiente al ser usadas con CPUs, lo cual es casi un requisito al no disponer de ninguna GPU.

## **Parte IV**

# **Implementación**



# Capítulo 8

## Extracción de errores

### 8.1. Definición de error

Para extraer los errores de las partidas necesitamos definir qué se considera un error. De forma intuitiva, se puede decir que un error de ajedrez es un movimiento que empeora objetivamente la posición del bando que lo realiza. Para ello, vamos a definir una puntuación para los movimientos usando motores de ajedrez.

#### 8.1.1. Puntuación de un movimiento

La **puntuación de un movimiento** será la valoración de la posición anterior al movimiento menos la valoración de la posición después del movimiento, desde el punto de vista del bando que lo realiza. Para la **valoración de la posición** usaremos la evaluación de un motor de ajedrez, en este caso usaremos Stockfish 15, con ciertas limitaciones a la búsqueda para acotar el tiempo.

$$Val(pos, mov) = \begin{cases} Val(pos) - Val(pos + mov) & \text{si mueven las blancas} \\ (Val(pos) - Val(pos + mov)) * -1 & \text{si mueven las negras} \end{cases}$$

$$Val(pos) = Stockfish(pos, lims)$$

La valoración de la posición, por tanto, se interpreta de forma diferente de la valoración de una posición: un valor de 0 nos indica que el movimiento fue perfecto, mientras que un valor alto indica que ha empeorado la valoración de la posición por esa cantidad para ese bando. Hay que tener en cuenta que la valoración no será perfecta ya que los motores de ajedrez no lo son (más aun con las limitaciones que introducimos), por lo que las puntuaciones no

reflejan el caso ideal (hay evaluaciones de movimientos con un valor negativo en algunos casos).

### 8.1.2. Condiciones para el error

La valoración de un movimiento nos permite entonces definir un error: aquel movimiento que empeore la posición más que cierto umbral. El umbral por defecto que se ha establecido es de 50cp, o 0.5 puntos. Aun así, por la naturaleza del juego y de la puntuación de las posiciones, se han encontrado algunos errores que queremos filtrar. Éstos son:

- Una posición completamente perdida que va a peor.
- Una posición completamente ganada que empeora un poco, pero sigue ganada.

Para quitarlas, se han categorizado las posiciones evaluadas según su puntuación relativa a su bando. Una posición está:

- **Completamente ganada** si puntúa más de 500 cp.
- **Ganada** si puntúa entre de 500 y 250 cp.
- **Igualada** si puntúa entre 250 y -250 puntos.
- **Perdida** si puntúa entre de -250 y -500 puntos.
- **Completamente Perdida** si puntúa menos de -500 puntos.

Aprovechando que la evaluación de una posición empeora o se queda igual después de hacer un movimiento, se han creado los siguientes filtros:

- Si la posición después del movimiento está completamente ganada, se descarta el error.
- Si la posición antes del movimiento está completamente perdida, se descarta el error.
- Si la posición después del movimiento se encuentra ganada, se duplica el umbral para aceptar el movimiento.
- Si la posición antes del movimiento se encuentra ganada, también se duplica el umbral para aceptar el movimiento.
- En el resto de casos (la posición inicial y/o la posición final están iguales), el umbral se deja sin modificar.

También eliminaremos los 5 primeros turnos de la partida, ya que los motores son bastante torpes con ellas.

Cuadro 8.1: Visualización del tiempo para encontrar 10.000 errores

movimientos por error	tiempo de evaluación	posiciones evaluadas	tiempo
1 de cada 5	0.2 segundos	50.000 posiciones	2,77 horas
1 de cada 10	0.2 segundos	100.000 posiciones	5,55 horas
1 de cada 5	0.5 segundos	50.000 posiciones	7,94 horas
1 de cada 10	0.5 segundos	100.000 posiciones	13,89 horas

## 8.2. Objetivos

El programa tiene como objetivo extraer los errores humanos de partidas lo más rápido posible y, como hemos explicado anteriormente, usaremos a Stockfish para evaluar los movimientos. Pero la calidad de la valoración de los motores dependen en gran medida del tiempo y los recursos que tienen para analizar. Por ello, el diseño debe encontrar un compromiso entre velocidad y calidad de evaluación, lo cual es difícil teniendo en cuenta lo rápido que puede escalar el tiempo final.

Las evaluaciones que se hagan de las posiciones tampoco pueden ser *demasiado buenas*. El motivo es que los errores se obtienen para que sean posteriormente analizadas por una red neuronal que no explora el árbol de movimientos, sólo extrae información (de manera muy compleja) de la posición que tome como input. Por ello, no sería muy útil alimentarle un error cuyo refutamiento se encuentra en 20 movimientos. La red no es capaz de extraer nada de ahí. Esto nos deja con evaluaciones de profundidad "poco profundas", de 5 a 8 movimientos de profundidad. Ahí nos encontramos con un nuevo problema: las evaluaciones de una posición pueden cambiar **sustancialmente** de una profundidad a otra.

Otra característica deseable del programa sería que **fuese independiente del motor**. Para ello, nos comunicaremos con Stockfish usando el protocolo UCI, de forma que cambiar el motor será trivial.

## 8.3. Diseño del programa

Para controlar el tiempo de evaluación, se usarán los **límites de evaluación** que el protocolo UCI nos permite poner al motor. Éstos son **tiempo, profundidad y nodos (posiciones) a explorar**, y se pueden imponer en la combinación que se desee. El límite de tiempo es, obviamente, muy útil y lo usaremos siempre como límite máximo de recursos. Limitar la profundidad de búsqueda también es muy útil, ya que puede hacer que la evaluación

Cuadro 8.2: Tiempos que tarda Stockfish en analizar una posición a cierta profundidad. Se han usado 200 posiciones aleatorias de la base de datos, y por cada nivel siempre se han utilizado las mismas.

Profundidad	Mejor tiempo (sin mate)	Tiempo Medio	Peor Tiempo
5	0.002	0.0108	0.038
6	0.004	0.0268	0.089
7	0.009	0.0713	0.225
8	0.011	0.1696	0.739
9	0.021	0.3614	1.685
10	0.051	0.7749	4.429

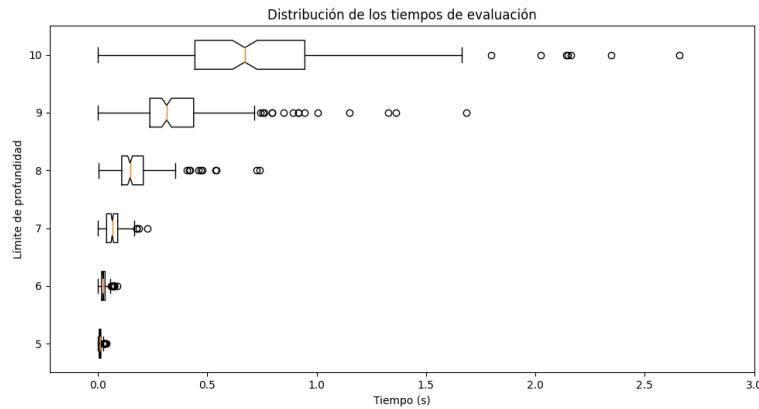


Figura 8.1: Gráfico de cajas de la tabla anterior

acabe antes si la posición de ajedrez es sencilla, y así optimizar un poco el uso de recursos. Finalmente, el límite de posiciones, que desgraciadamente no es de mucha ayuda y no se usará, ya que establecer el límite concreto dependería de los nodos por segundo, del tipo de búsqueda que realiza el motor, de la complejidad de la posición, etc.

Como vemos en la tabla de tiempos, las evaluaciones fijas en una profundidad tienen una variabilidad bastante grande. Teniendo en cuenta que queremos tener cerca de 500.000 errores para el clustering y cómo escalan los tiempos, hemos tomado de límite 8 movimientos de profundidad, con 0.25 segundos de máximo. Así, interrumpiremos la evaluación de las posiciones más lentas, según la gráfica de cajas, el 20 % más lento.

Para acelerar el proceso, vamos a usar las otras CPUs que disponemos. Para ello, sería un desperdicio otorgárselas a Stockfish, dado que las eva-

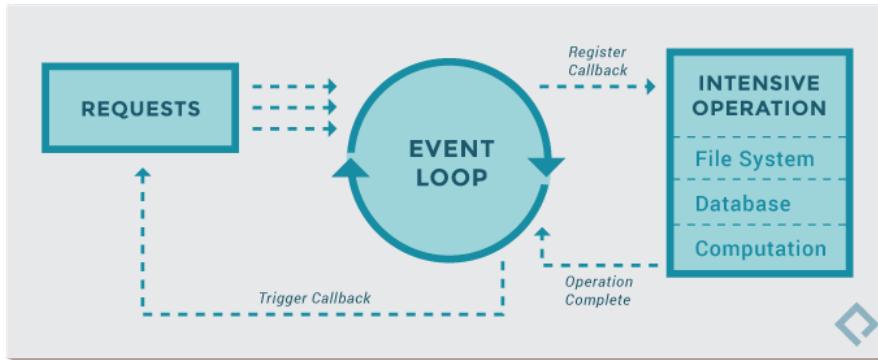


Figura 8.2: Bucle de eventos de AsyncIO [17]

luaciones son bastante cortas por lo que no tiene demasiado tiempo de usar el paralelismo. Entonces nos dirigiremos en otra dirección, se hará uso de varias instancias del motor a la vez. Esto es complicado de hacer, porque el protocolo UCI sólo permite enviar **una posición para analizar** cada vez, es decir, debemos esperar que acabe el análisis antes de enviar otra posición. Si trabajamos con una instancia, no hay problema: podemos bloquear el programa esperando al motor. Con varias instancias, y teniendo en cuenta la variabilidad de los tiempos de análisis, perderíamos mucho tiempo valioso de CPU.

La comunicación asíncrona parece ser la solución. La forma de hacer eso en Python es con el módulo AsyncIO, con una estructura de bucle de eventos. Esta estructura funciona de la siguiente manera: se crean una o varias tareas con esperas asíncronas, se añaden al bucle de eventos y después desviamos la ejecución a ese bucle. A partir de ahí, tenemos una cola de tareas que ejecutar. Cogemos la primera y, cuando llegamos a la espera asíncrona, dejamos la tarea y cogemos la siguiente. Cuando la espera asíncrona ha acabado, la tarea pasa automáticamente a la cola de tareas, lista para ser ejecutada cuando se pueda. En este caso, las esperas asíncronas son las evaluaciones del motor de ajedrez.

### 8.3.1. Estructura del programa

El programa se apoya fundamentalmente en una función asíncrona que se añade una vez al bucle de eventos de AsyncIO por cada instancia de motor que se quiere ejecutar. El pseudocódigo de la tarea es el siguiente:

```
motor = run Stockfish 15.1                                ▷ espera asíncrona
for partida en archivo_partidas do
    Ignorar los primeros 5 movimientos de partida
```

Cuadro 8.3: Partidas, posiciones y errores analizados

Partidas	Posiciones	Errores
37.839	2.903.607	501.858
0.38/seg	29.16/seg	5.04/seg

```

eval-previo = motor.evalua( partida.posicion )      ▷ espera asíncrona
eval-actual = 0
for movimiento en partida do
    partida.realiza-movimiento( movimiento )
    eval-actual = motor.evalua( partida.posicion ) ▷ espera asíncrona
    if movimiento es un error then
        Anotar movimiento en archivo
    end if
end for
end for

```

Las tareas se ejecutan todas en la misma hebra en el bucle de eventos. En cambio, todas las instancias de Stockfish con las que se comunica se ejecutan en su propia CPU.

Los errores encontrados se anotan en un fichero CSV, con la siguiente información:

- La posición antes del movimiento, en formato FEN
- El movimiento en sí, en formato SAN (notación algebráica)
- La evaluación del movimiento
- La evaluación de las posiciones de antes y después del movimiento

## 8.4. Resultados

Ejecutando 6 instancias a la vez en el sistema (Intel(R) Core(TM) i5-8265U CPU, 8 threads), se han extraído 501858 errores.

# Capítulo 9

## Extracción del vector de características

### 9.1. De dónde se extraerán las características

El objetivo de este trabajo es ver si se pueden clasificar los errores que se han extraido, usando la red neuronal **DeepChess**.

Como se ha dicho anteriormente, la red neuronal de DeepChess es una red compleja, con varias partes, que es capaz de comparar dos posiciones de ajedrez e indicar en cuál es más probable que gane un bando, sin buscar en el árbol de movimientos, y con una precisión muy alta (98 %).

Como la red compara dos posiciones, la idea introducir la posición antes del error y la posición de después, e interpretar de varias formas ciertas capas de neuronas de su compleja estructura para obtener un vector de características. En el proceso de clustering, se comprobará si los errores se organizan con alguna estructura usando estos vectores.

#### 9.1.1. Características a partir de la penúltima capa

Este método surge de analizar el proceso matemático que sucede desde el output de la penúltima capa hasta el output de la red. DeepChess devuelve dos valores: cada uno asociado a la probabilidad de que gane el blanco en una de las posiciones. La última capa usa de función de activación a *softmax*, que en el caso de tener dos inputs funciona de la siguiente manera:

$$\text{softmax}(a, b) = \left( \frac{\exp a}{\exp a + \exp b}, \frac{\exp b}{\exp a + \exp b} \right)$$

Los valores que se alimentan a esta función son una suma ponderada en-

tre los pesos de las dos neuronas, y los 100 valores que devuelve la penúltima capa.

$$\text{Output} = \text{softmax}(n_0, n_1) = \text{softmax}\left(\sum_{i=1}^{100} w_{0,i} * \text{out}_i, \sum_{i=1}^{100} w_{1,i} * \text{out}_i\right)$$

La cosa es que softmax es una función monótona creciente, es decir, si  $a > b$  y  $\text{softmax}(a, b) = (a', b')$ , entonces  $a' > b'$ . Esto quiere decir que es indiferente para la predicción, ya que sólo se comprueba si un valor es mayor que el otro. Si eliminamos softmax de la ecuación y añadimos un signo de comprobación, llegamos a esto.

$$\sum_{i=1}^{100} (w_{0,i} * \text{out}_i) (>, <) \sum_{i=1}^{100} (w_{1,i} * \text{out}_i)$$

Nuestra suposición es que se puede extraer información sobre la comparación que realiza DeepChess a través de **alguna combinación de los pesos y los outputs de la penúltima capa**. Intentaremos realizar dos combinaciones de ellos, que se han llamado **mov2vec** y **dif2vec**.

En mov2vec, añadiremos toda la información posible al vector de características. Para ello, uniremos todos los factores que forman parte de la comprobación, por lo que tendrá 200 elementos.

$$\text{mov2vec}(\text{error}) = (w_{0,i} * \text{out}_i | i = 1 \dots 100) \cup (w_{1,i} * \text{out}_i | i = 1 \dots 100)$$

En cambio, para dif2vec nos centramos en las diferencias entre los dos lados de la comprobación, ya que al final nos interesa qué es aquello que crea un desequilibrio entre los dos lados de la comprobación. Para ello, usaremos la diferencia entre los factores de las dos neuronas. Por lo tanto, el vector tendrá 100 elementos.

$$\text{dif2vec}(\text{error}) = ((w_{0,i} - w_{1,i}) * \text{out}_i | i = 1 \dots 100)$$

En estos dos métodos, las características deberían estar en la misma medida, es decir, deberían ser comparables entre sí, algo poco común en datos de clustering donde cada característica sólo se puede comparar con sí misma.

### 9.1.2. Extracción de Pos2Vec

La red de DeepChess contiene una parte que se dedica a extraer características no lineales de una posición: la torre de autoencoders **Pos2Vec**. Un autoencoder es una red neuronal a la que se fuerza a codificar el input en un tamaño de memoria menor que el inicial y después reconstruirlo lo mejor posible durante el entrenamiento. Posteriormente, se corta la parte de decodificación.

La idea que queremos probar es si se puede usar la codificación de Pos2Vec para clasificar las posiciones (en vez de los movimientos). Para ello, introduciremos las posiciones donde ha ocurrido el error y extraeremos su codificación, que será un vector de 100 valores.

Pos2Vec pasa por dos períodos de entrenamiento: la parte de aprendizaje no supervisado, donde se entrena los autoencoders, y la de aprendizaje supervisado como una parte de la red final. También nos interesa ver qué diferencias hay entre esas dos, así que usaremos dos modelos distintos: **Pos2Vec1** se referirá a la red antes del entrenamiento, mientras que **Pos2Vec2** a la del final.

Realizar clustering de estos datos presenta una serie de retos bastante únicos. En primer lugar, las características no son comparables entre sí. Entre que no son lineales, y no podemos interpretar qué significa, no se puede hacer la suposición de que tienen algo en común. Pero la característica más complicada es que **tampoco se pueden comparar con sí mismas**. Al no ser lineales, es posible (y probable) que los incrementos no signifiquen lo mismo en distintos valores.

## 9.2. Selección del modelo entrenado

El entrenamiento de la red descrito en la publicación es muy intenso en recursos, y los autores no publicaron ni el modelo ni el código. Por ello, buscaremos una implementación de terceras personas, que ya haya entrenado al modelo.

Para que una implementación sea apta para ser seleccionada, le pedimos una serie de requisitos: deben de almacenar el modelo final que han entrenado, debe estar el código de creación y entrenamiento del modelo y, finalmente, la estructura del modelo y el proceso de entrenamiento no puede haberse desviado mucho del descrito en la publicación. Hemos encontrado tres implementaciones que cumplen con los criterios:

- <https://github.com/Bot-Benedict/DeepChess>, que dice tener un accuracy del 97 %. No se han encontrado diferencias entre el proceso que

sigue y el de la publicación.

- [\*https://github.com/oripress/DeepChess\*](https://github.com/oripress/DeepChess), usa leaky ReLU en vez de una ReLU normal como función de activación, pero por lo general sigue la publicación.
- [\*https://github.com/vajjhala/deepchess\*](https://github.com/vajjhala/deepchess), también usa leaky ReLU en vez de ReLU, y para el learning rate usa la política ADAM en vez de usar decay.

Estos son repositorios que no han sido considerados por un motivo u otro:

- [\*https://github.com/lysnikolaou/python-deepchess\*](https://github.com/lysnikolaou/python-deepchess), el entrenamiento es demasiado diferente al del paper (pesos del DBN constantes en el entrenamiento del modelo final, learning rate constante y con valor diferente y, finalmente, usa la estructura minimizada del modelo para el primer entrenamiento)
- [\*https://github.com/dangeng/DeepChess\*](https://github.com/dangeng/DeepChess), usa como función de activación una sigmoidal en vez de SoftMax, lo cual seguro que ha afectado al entrenamiento. Además, introduce una capa de *batch normalization* entre cada capa de la red siamesa.
- [\*https://github.com/fauzanardh/DeepChessAI\*](https://github.com/fauzanardh/DeepChessAI), aunque tiene el mismo nombre, no es la implementación del modelo que buscamos.
- [\*https://github.com/NguyenHieu201/DeepChessProject\*](https://github.com/NguyenHieu201/DeepChessProject), le falta parte del proyecto por implementar.
- [\*https://github.com/PaulConerardy/DeepChess\*](https://github.com/PaulConerardy/DeepChess), no tiene los archivos del modelo entrenado.

Cabe destacar lo complicado que es hacer funcionar los modelos de los repositorios elegidos. El modelo de *oripress* sólo funciona con Python 2.7, que fue abandonado en 2020 [30]; tensorflow 0.10.0 (completamente incompatible con la actual 2.12) y TfDeploy 0.3.3, una versión antigua de una librería discontinuada desde 2016. Los otros modelos no son menos sencillos.

La implementación que usaremos finalmente será aquella que tenga mejor precisión con sus predicciones. Para probarlas, hemos escogido 20.000 posiciones aleatorias de la base de datos de partidas, y las hemos analizado con Stockfish. La idea es crear parejas entre posiciones igual que en la publicación: una donde eventualmente ganan las blancas, y otra donde ganan las negras. El modelo acierta si predice bien cuál es cual. La evaluación de Stockfish se usará para crear tres distinciones entre las posiciones, y crear parejas entre ellas. Estas distinciones son:

Cuadro 9.1: Matrices de confusión de las parejas de cualquier posición

	Predicciones de DeepChess					
	Bot-Benedict		oripress		vajjhala	
	Blancas	Negras	Blancas	Negras	Blancas	Negras
Victoria Blanca	6519	3450	6372	3270	6172	3795
Victoria Negra	3481	6550	3628	6730	3828	6205
Precisión	<b>65.345 %</b>		<b>65.510 %</b>		<b>61.885 %</b>	

Cuadro 9.2: Matrices de confusión de las parejas de posiciones fáciles

	Predicciones de DeepChess					
	Bot-Benedict		oripress		vajjhala	
	Blancas	Negras	Blancas	Negras	Blancas	Negras
Victoria Blanca	9073	927	9233	767	7556	2444
Victoria Negra	991	9009	643	9357	2553	7447
Precisión	<b>90.41 %</b>		<b>92.95 %</b>		<b>75.015 %</b>	

- Posiciones fáciles, que tienen una evaluación de 3 puntos para un bando u otro (acorde con el ganador). Las posiciones de las parejas tendrán al menos 6 puntos de diferencia entre sí.
- Posiciones medio, que tienen una evaluación entre 1 y 3 puntos para un bando u otro (acorde con el ganador). Las posiciones de las parejas tendrán al menos 2 puntos de diferencia entre sí.
- Posiciones difíciles, que tienen una evaluación de menos de 1 punto para un bando u otro.
- Cualquier posición, no hay filtro para ellas. Hay algunas posiciones cuya evaluación es muy grande para un bando, pero finalmente vence el otro. Esas posiciones sólo están aquí.

Se han creado 20.000 parejas de cada categoría, y se han analizado con los tres candidatos. Las matrices de confusión resultantes se muestran en las siguientes tablas:

Viendo la precisión en las parejas que toman todas las posiciones, tenemos dos candidatos: Bot-Benedict y oripress. Tomaremos el primero de ellos por tener una precisión algo mayor en las partidas difíciles.

Cuadro 9.3: Matrices de confusión de las parejas de posiciones medio

	Predicciones de DeepChess					
	Bot-Benedict		oripress		vajjhala	
	Blancas	Negras	Blancas	Negras	Blancas	Negras
Victoria Blanca	7716	2284	7651	2349	8234	1766
Victoria Negra	2240	7760	2065	7935	1772	8228
Precisión	<b>77.38 %</b>		<b>77.93 %</b>		<b>82.31 %</b>	

Cuadro 9.4: Matrices de confusión de las parejas de posiciones difíciles

	Predicciones de DeepChess					
	Bot-Benedict		oripress		vajjhala	
	Blancas	Negras	Blancas	Negras	Blancas	Negras
Victoria Blanca	4956	5044	5200	4800	5152	4848
Victoria Negra	4411	5589	4834	5166	4745	5255
Precisión	<b>52.725 %</b>		<b>51.83 %</b>		<b>52.035 %</b>	

Algo más que se puede observar es que la precisión no se acerca al 98 % prometido en la publicación. Esto creemos que se debe, principalmente, porque las posiciones de nuestra base de datos son diferentes a las que se han usado para entrenar: nuestra base de datos contiene partidas de todo tipo de límites de tiempo, y jugadas por humanos. En cambio, el paper (y por ello estas implementaciones) usa las partidas de CCLR, jugadas por motores de ajedrez con 40 minutos de tiempo. Las posiciones que tenemos son más caóticas por ello, y es normal que falle más.

### 9.3. Implementación y resultados

En esta sección se explicará el desensamblado de la red neuronal, la velocidad de transformación de movimientos en vectores, y la naturaleza de los datos resultantes.

#### 9.3.1. Creación de las redes derivadas de DeepChess

Para proceder con los 4 métodos de extracción de características, usaremos redes neuronales derivadas de DeepChess que hagan los procedimientos

de manera automática. La red neuronal de Bot-Benedict se creó con Tensorflow [40], una librería open-source de aprendizaje automático centrada en redes neuronales profundas. Gracias a la enorme cantidad de herramientas que proporciona, somos capaces de guardar y recuperar redes neuronales de archivos, crear cualquier tipo de red usual de forma sencilla y flexible, e incluso crear capas personalizadas para las redes, ajustadas a nuestras necesidades.

El proceso de creación de las cuatro redes fue el siguiente:

1. Se cargaron los modelos DeepChess y Pos2Vec (antes de ser entrenado como parte de toda la red). Bot-Benedict usó la versión 1.x de TensorFlow (última vez actualizada en 2019).
2. Los pesos de las redes fueron extraídos y almacenados con NumPy.
3. Con la versión actual de Tensorflow, 2.x, se crearon cáscaras vacías de las cuatro redes, dos de ellas tienen la misma topología que Pos2Vec mientras que dif2vec y mov2vec es similar a la red de DeepChess, excepto que se ha sustituido su capa output por una capa personalizada que hace lo que hemos descrito en secciones anteriores.
4. Se cargaron los pesos de las redes entrenadas, y se cargaron estos modelos vacíos con ellos.
5. Se guardaron los modelos en el ordenador, para no tener que repetir estos pasos.

Ahora, si se quiere realizar alguna de las transformaciones, se puede cargar rápidamente cualquiera de las 4 redes y usarla.

### 9.3.2. Transformación de la base de datos de errores

Los 501.858 errores se transformaron con las 4 redes en 12 minutos 33 segundos, a 2665.9 transformaciones por segundo. Este ritmo tan rápido sólo es posible al usar muchas posiciones al mismo tiempo, ya que entonces la librería TensorFlow es capaz de optimizar al máximo las simples operaciones de la red (que se reducen a sumar, multiplicar y usar la función ReLU).

Los datos finales son cuatro matrices de números en coma flotante, tres de tamaño 501.858 x 100 y una de 501.858 x 200.

Las matrices se almacenaron en 4 archivos npy diferentes, un formato de NumPy que es más eficiente en memoria que el formato CSV, mientras que es igual de fácil de manejar.



# Capítulo 10

## Clustering de datos

La primera parte de cualquier proceso de clustering es observar la naturaleza de los datos, para poder tomar decisiones más informadas sobre los procesos a utilizar.

En este caso, disponemos de cuatro sets de datos de muchas dimensiones, extraídos de capas de una red neuronal. Las neuronas usan la función de activación ReLU, por lo que se espera tener muchos valores 0.

### 10.1. Valores 0

La tabla nos muestra algunos datos sorprendentes. En primer lugar, la mayoría de las características en mov2vec y dif2vec son nulas, ya que 85 de las 100 neuronas de la penúltima capa nunca se han activado. Las similitudes en todos los análisis de este apartado son entendibles, ya que las transformaciones que les diferencian no modifican los 0s de los outputs de las neuronas.

Después, en los dos pos2vec, el porcentaje de características no nulas se aleja bastante del porcentaje de valores no nulos. Esto indica que las carac-

Cuadro 10.1: Valores 0s presentes en los datos

	Porcentaje no 0	Caract. no 0	Inst. no 0
mov2vec	10.1 %	30	Todas
dif2vec	10.1 %	15	Todas
pos2vec1	31.9 %	85	Todas
pos2vec2	6.21 %	48	Todas

Cuadro 10.2: Análisis de valores 0 después de eliminar características nulas

	Porcentaje de valores no 0	Caract. con ¡1% de valores
mov2vec	67.16 %	6 características de 30
dif2vec	67.16 %	3 características de 15
pos2vec1	37.52 %	27 características de 85
pos2vec2	12.95 %	32 características de 48

terísticas se encontrarán bastante vacías, y que si comparamos instancias de datos, seguramente no coincidan en qué datos no 0 tienen.

Por supuesto, las características 0 son eliminadas de los datos, ya que no aportan ningún tipo de información, pero los datos siguen bastante vacíos, y eliminar características beneficiará en gran medida al clustering.

Debemos encontrar un equilibrio entre eliminar características e información. Imaginémonos que tenemos una característica con un 1 % de valores no 0. Si comparamos dos instancias aleatorias,

- El 98,01 % de las veces, ambos valores serán 0,
- El 1.98 % de las veces, un valor será 0
- El 0.01 % de las veces, ningún valor será 0

De estos casos, sólo los dos primeros nos aportan información, es decir, sólo el 2 % de las veces es útil. Una idea posible es **unir características poco pobladas en una sola**. Veamos qué casos pueden sucederse si unimos dos características con un 1 % de valores no 0 (supondremos que ningún valor no nulo se superpone).

- El 96,03 % de las veces, ambos valores serán 0,
- El 3.84 % de las veces, un valor será 0,
- El 0.02 % de las veces, ningún valor será 0 cuando uno de ellos debería serlo (el 0 de una característica se sobreescribió por un valor)
- El 0.02 % de las veces, ningún valor será 0 correctamente.

Como se puede ver, se reduce una dimensión, a cambio de un 0.02 % de que la comparación sea errónea. En el caso de juntar 10 características de esta manera, cada una con un 1 %, suponiendo que no se superponen valores no nulos,

- El 81 % de las veces, ambos valores serán 0,
- El 19 % de las veces, un valor será 0,
- El 0.9 % de las veces, ningún valor será 0 cuando uno de ellos debería serlo (el 0 de una característica se sobreescribió por un valor)
- El 0.1 % de las veces, ningún valor será 0 correctamente.

El intercambio es eliminar 9 dimensiones a cambio de tener un 1 % de realizar una comparación errónea, un intercambio muy provechoso.

Hemos usado esta técnica para acumular todas las características que no llegasen al 5 % de valores no 0, hasta que ese porcentaje llegase al 10 %. Cuando hemos llegado a ese punto, los hemos acumulado en otra características. Otro problema que no hemos tocado con este método es la suposición de que los valores no 0 no se superponen. Para ello, se ha creado un algoritmo basado en órdenes aleatorios que minimice la superposición. Funciona de la siguiente manera:

1. Las características a fusionar se ordenan de forma aleatoria.
2. Una por una, siguiendo el orden, se construye la solución:
  - a) La característica se acumula junto con las otras,
  - b) Se suman la cantidad de datos superpuestos,
  - c) Si los datos no nulos de las características sobrepasa el 10 %, se acumulan en otra característica nueva.
3. La cantidad de datos superpuestos se compara con el mejor orden que hemos obtenido. Si es mejor, la guardamos.
4. Se vuelve a la instrucción 1. si no se ha llegado al máximo de iteraciones.

El algoritmo es rápido y sólo es necesario ejecutarlo una vez para los datos de pos2vec2 (el único que ha necesitado más de una agrupación), así que se han realizado unas 1000 iteraciones. Aquí están los resultados:

Las dimensiones de los datos de pos2vec1 y pos2vec2 han sido reducidas considerablemente, ya que tenían características muy vacías. Después de este proceso, estamos preparados para preparar los datos.

Cuadro 10.3:

	Características transformadas	Características finales
mov2vec	$6 \rightarrow 2$	26
dif2vec	$3 \rightarrow 1$	13
pos2vec1	$35 \rightarrow 2$	52
pos2vec2	$33 \rightarrow 1$	16

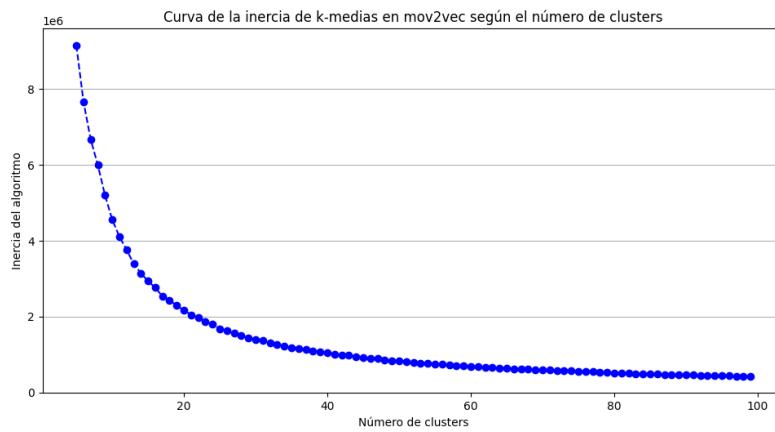


Figura 10.1: Método del codo para mov2vec

## 10.2. Elección del número de clusters

Para saber en qué rango se debe buscar el número de clusters, usaremos la heurística llamada **el método del codo** [20].

Este método consiste en ejecutar k-medias varias veces en los datos, cada vez incrementando el número de clusters. Después de cada ejecución, se anota la medida de error, que recordemos es la media de las distancias entre los datos del mismo cluster. Es natural que el error se reduzca al aumentar el número de clusters, pero llega un punto en el que seguir aumentando el hiperparámetro no se traduce en un descenso significativo, es decir, el nuevo cluster no aporta mucho. Si dibujamos la gráfica del error, se verá un descenso significativo que después se reduce de forma significativa. El nombre del método viene porque el rango de valores óptimo del hiperparámetro está situado en el “codo” de la gráfica.

Parece que el rango de valores del hiperparámetro es similar para mov2vec, dif2vec y pos2vec2, entre 18 y 22. Mientras tanto, para pos2vec1 el codo de

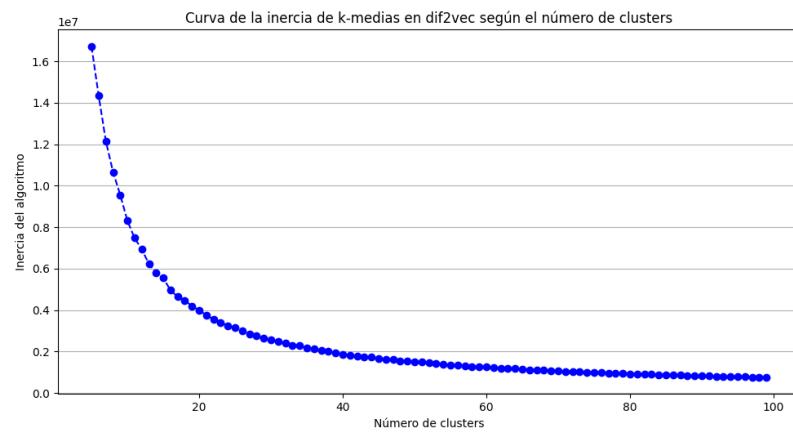


Figura 10.2: Método del codo para dif2vec

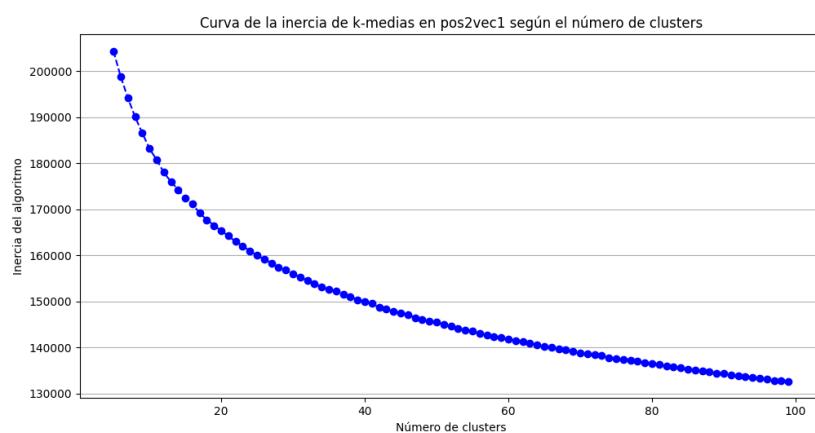


Figura 10.3: Método del codo para pos2vec1

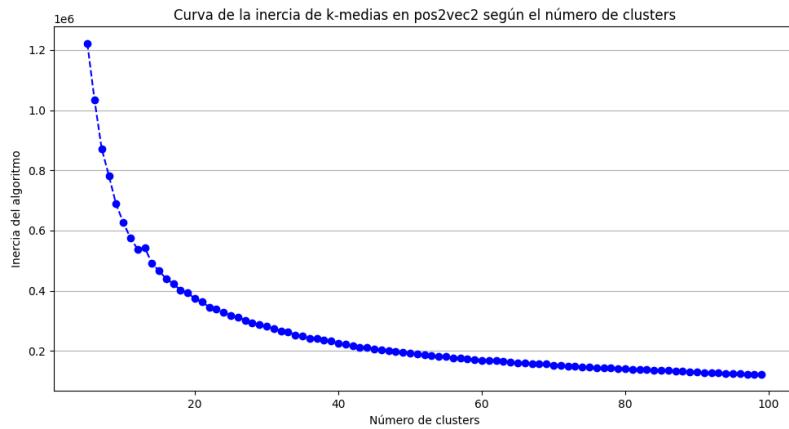


Figura 10.4: Método del codo para pos2vec2

la gráfica no está claro por donde se encuentra, pero en torno al 30.

Eligiremos usar 20 clusters para todos los datasets excepto para pos2vec1, que usaremos 30.

### 10.3. Ejecución de k-medias

La implementación del algoritmo k-medias que se ha usado viene de la librería Scikit-Learn. Se ha usado el número de clusters especificado anteriormente, y las condiciones de parada son 200 iteraciones o convergencia absoluta (los clusters no cambian en absoluto en una iteración). El algoritmo se ha ejecutado 10 veces por la inicialización aleatoria de los centroides.

### 10.4. Validación del clustering

Comprobar la calidad de los clusters no es una tarea nada fácil. Tenemos la situación de que el problema entre manos no se ha resuelto antes, y por lo tanto no podemos comparar con nada.

El método que se usará consiste en buscar patrones ajedrecísticos en las posiciones, y comparar la incidencia que tienen entre diferentes clusters. Usaremos los mismos patrones que el trabajo de final de grado de Adrián Rodríguez Montero[26], en el cual los usa para etiquetar posiciones. Todos los valores de las etiquetas son números enteros, y la mayoría de ellas se usan para ambos bandos. Hay un total de 38 etiquetas diferentes.

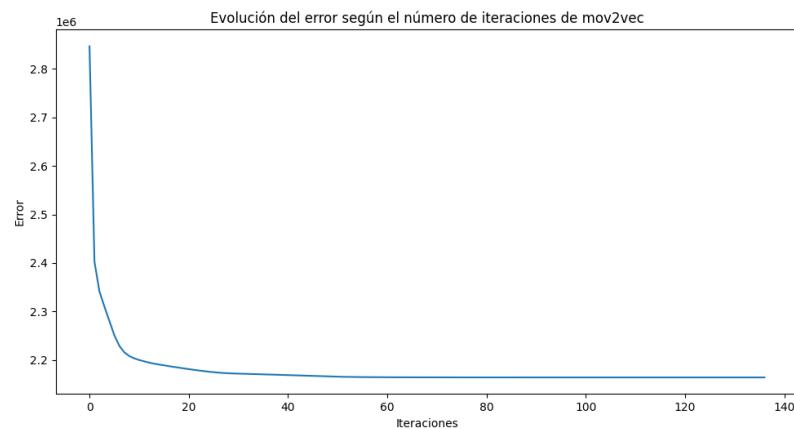


Figura 10.5: Evolución del error en la muestra de mov2vec

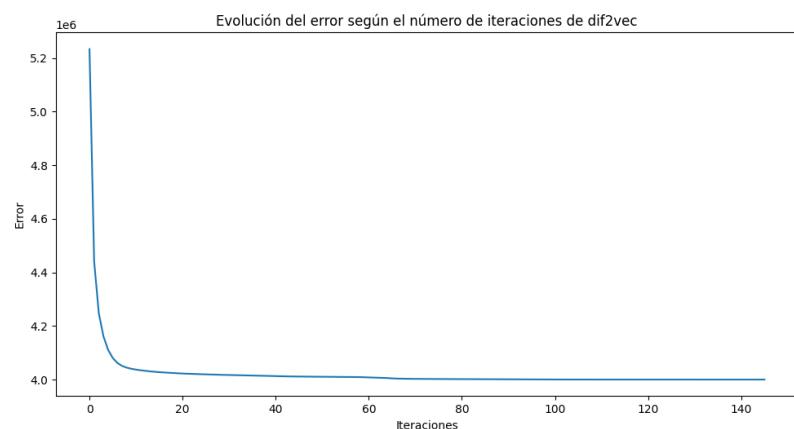


Figura 10.6: Evolución del error en la muestra de dif2vec

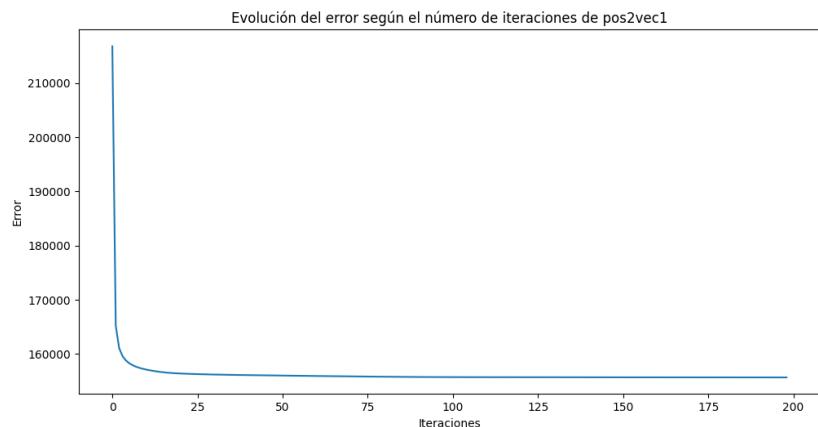


Figura 10.7: Evolución del error en la muestra de pos2vec1

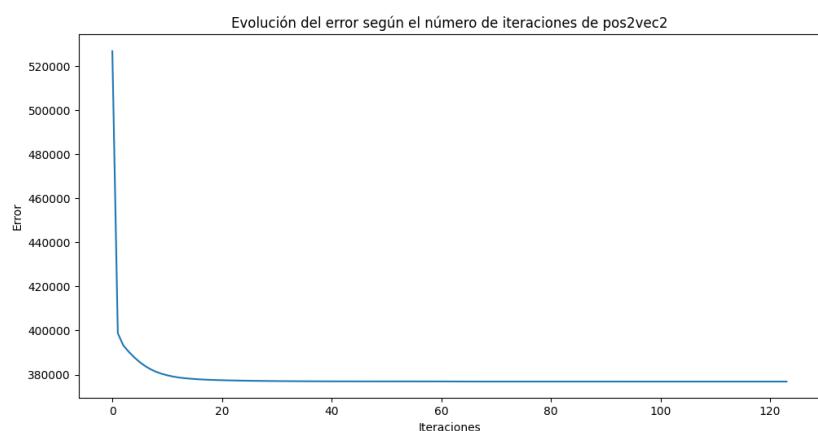


Figura 10.8: Evolución del error en la muestra de pos2vec2

Cuadro 10.4: Etiquetas de validación, extraídas de [26]. Cada etiqueta es aplicada a los dos bandos

Tipo	Etiqueta	Clase
Etiquetas comunes	Ventaja de espacio	-1,0,1
	Columnas abiertas	0-8
	Alfiles del mismo color	0,1
	Alfiles de distinto color	0,1
Una etiqueta por bando	Columnas semiabiertas	0-6
	Peones	0-8
	Caballos	0-2
	Alfiles	0-2
	Torres	0-2
	Damas	0-2
	Torre en séptima	0, 1
	Torres dobladas	0, 1
	Torres ligadas	0, 1
	Pistola de Alekhine	0, 1
	Peones doblados	0-3
	Peones aislados	0-6
	Peones retrasados	0-4
	Peones pasados	0-6
	Islas de peones	0-4
	Falanges de peones	0-3
	Peones conectados	0-3

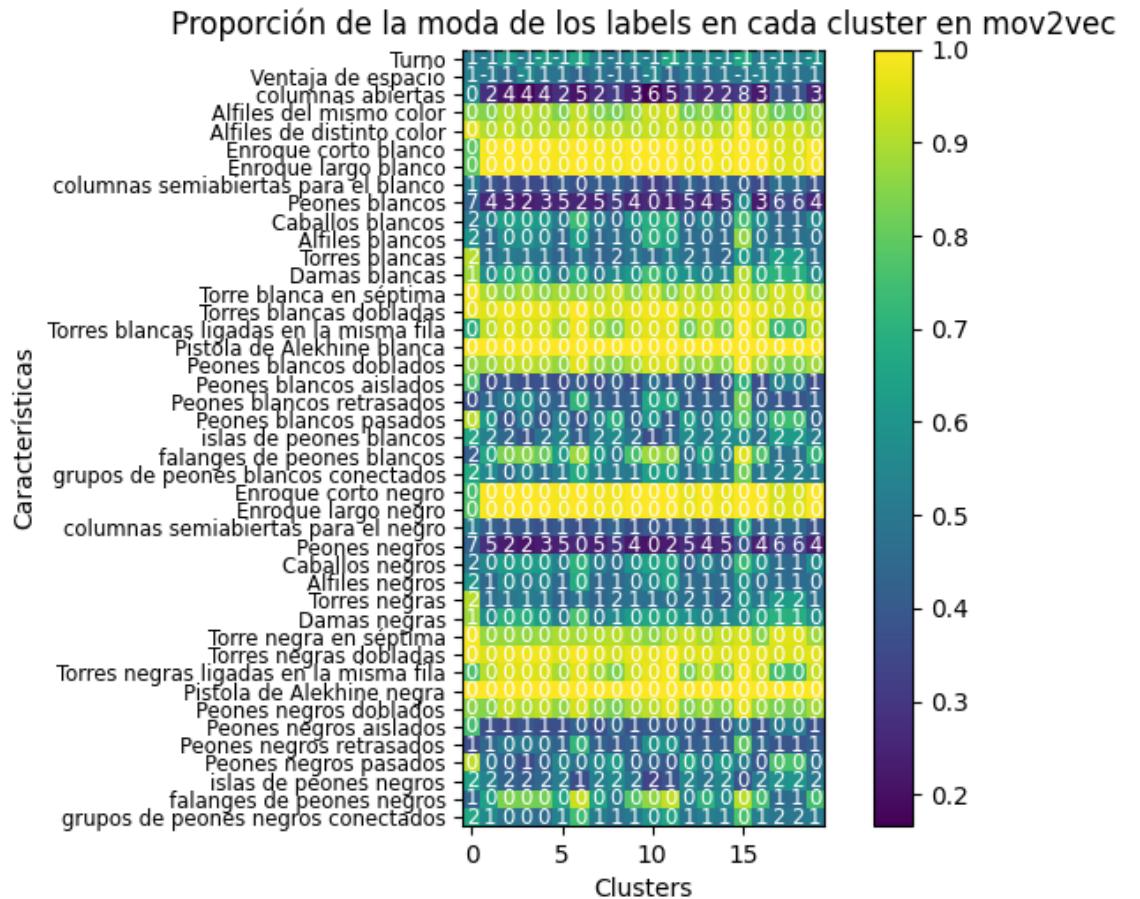


Figura 10.9: Mapa de calor de las proporción de la etiqueta más común junto con su valor (mov2vec)

Así, podemos observar cuál es el valor más común en cada combinación de etiqueta y cluster (su moda), así como la proporción total en la que se encuentran presentes.

Es obvio que hay desequilibrios en las representaciones de algunos valores en las etiquetas. Además, la proporción de la moda hunde a aquellas etiquetas con más de un posible valor. Por eso, haremos una visualización con una puntuación distinta, que otorga un valor alto a una combinación de etiqueta-cluster en el caso que su proporción de valores sea diferente que la habitual.

$$Punt(etiq, cluster) = \max_{x \in etiq} \left( \frac{pIntracluster_x - pTotal_x}{1 - pTotal_x} \right)$$

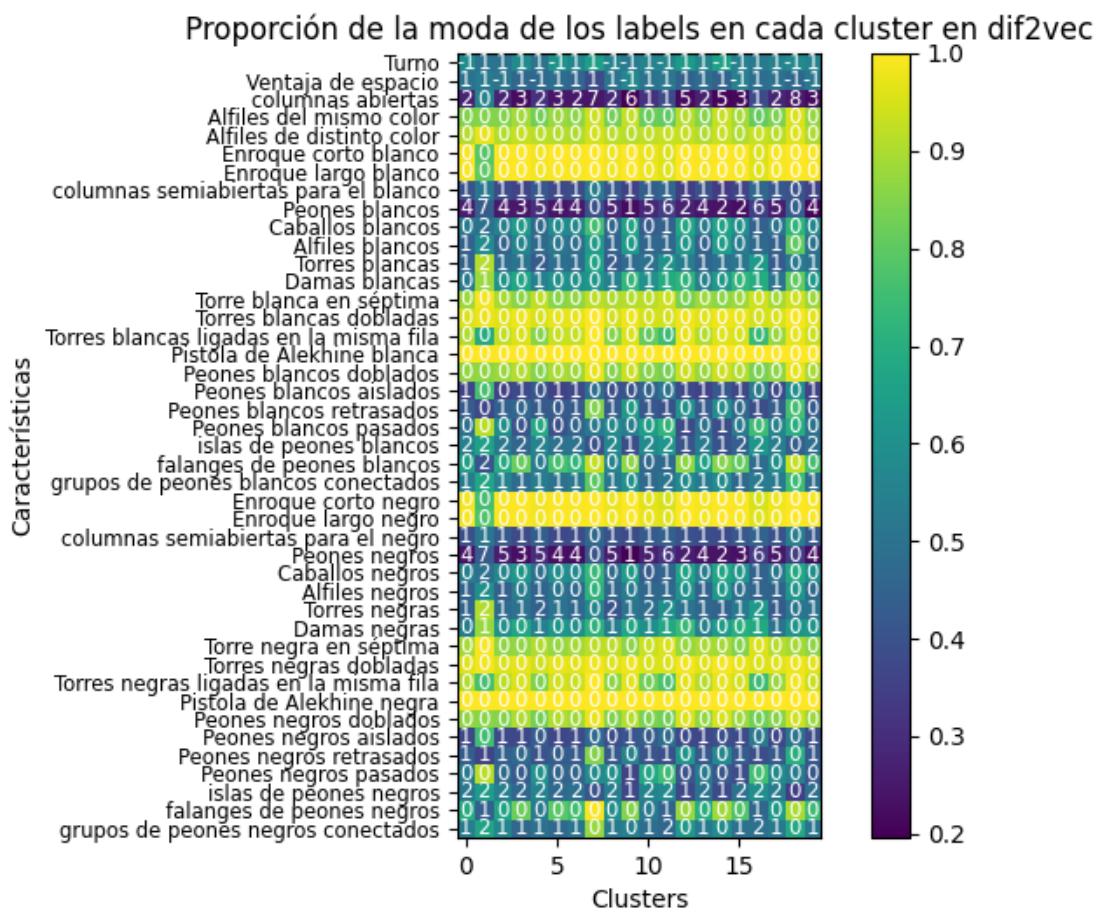


Figura 10.10: Mapa de calor de las proporción de la etiqueta más común junto con su valor (dif2vec)

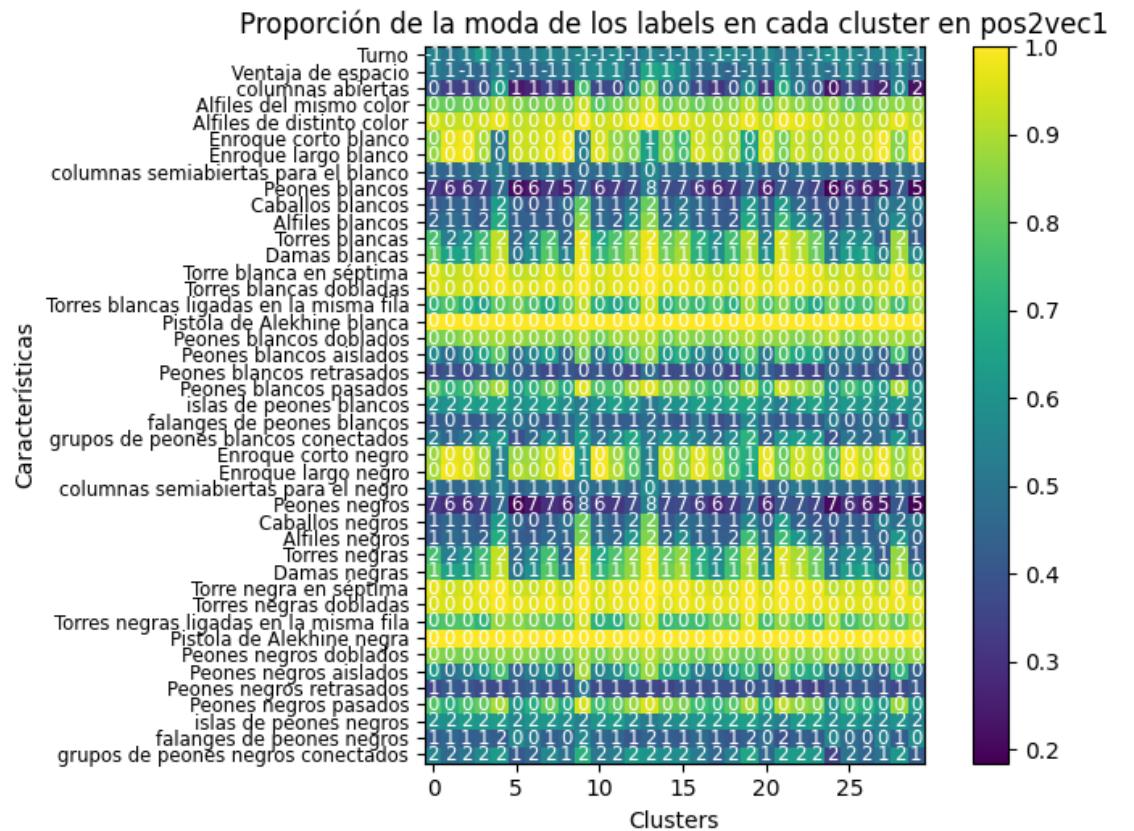


Figura 10.11: Mapa de calor de las proporción de la etiqueta más común junto con su valor (pos2vec1)

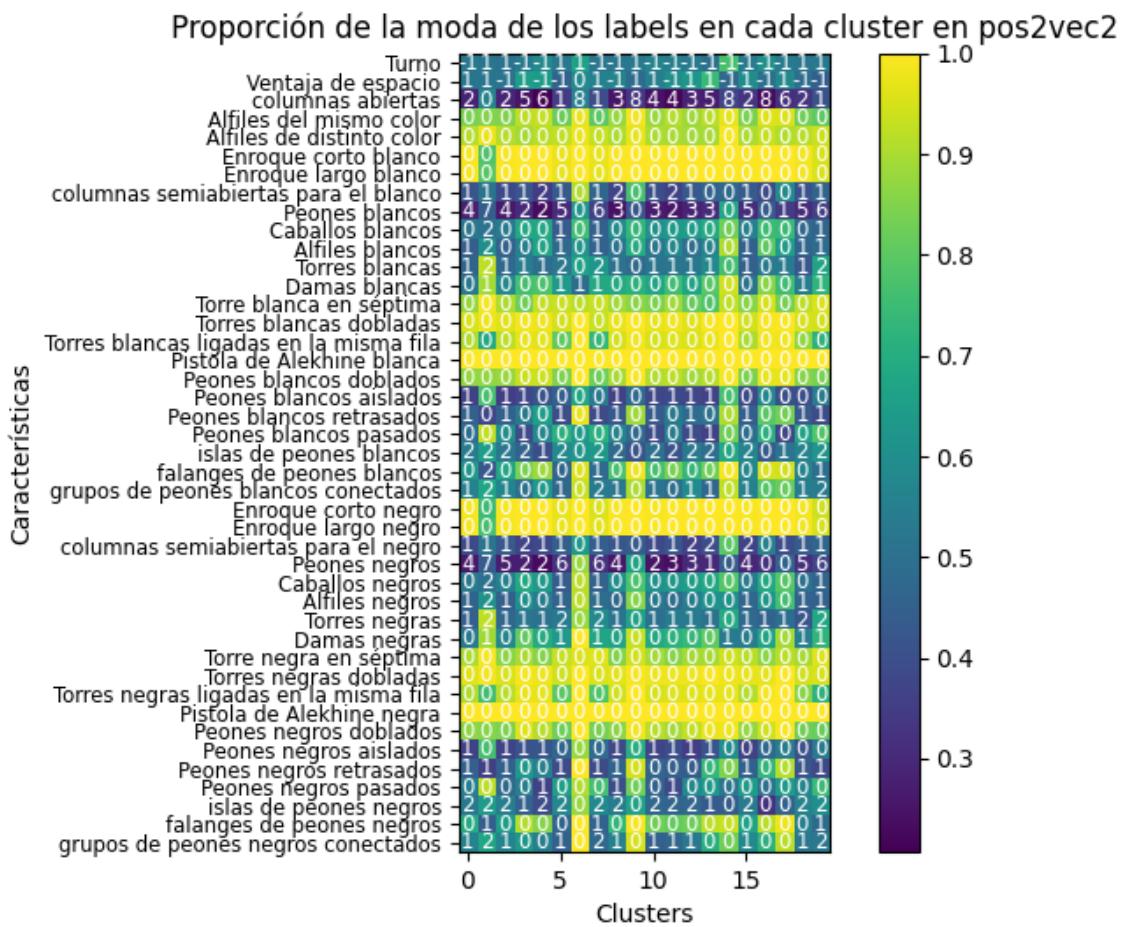


Figura 10.12: Mapa de calor de las proporción de la etiqueta más común junto con su valor (pos2vec2)

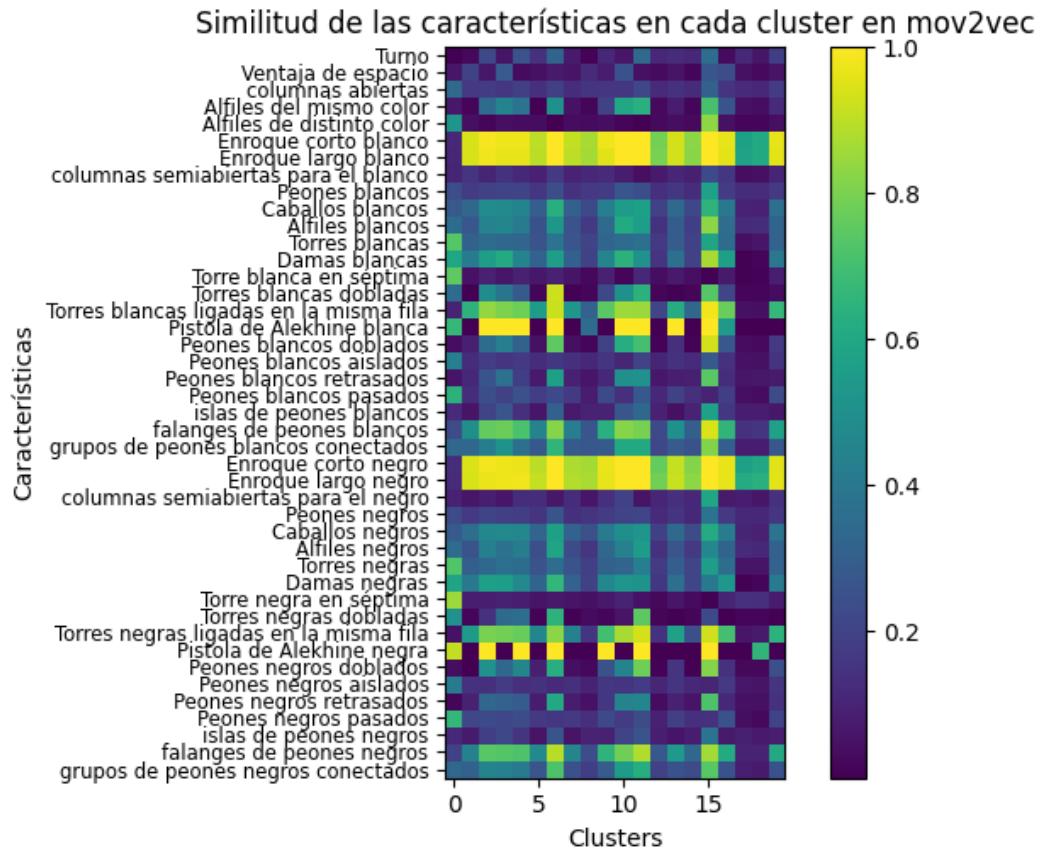


Figura 10.13: Mapa de calor de la diferenciación de etiquetas en mov2vec

El valor máximo de esta puntuación es 1 (en el caso de que sólo haya un valor de etiqueta en el cluster), mientras que la mínima es 0 (si la proporción es exactamente igual que en todos los datos).

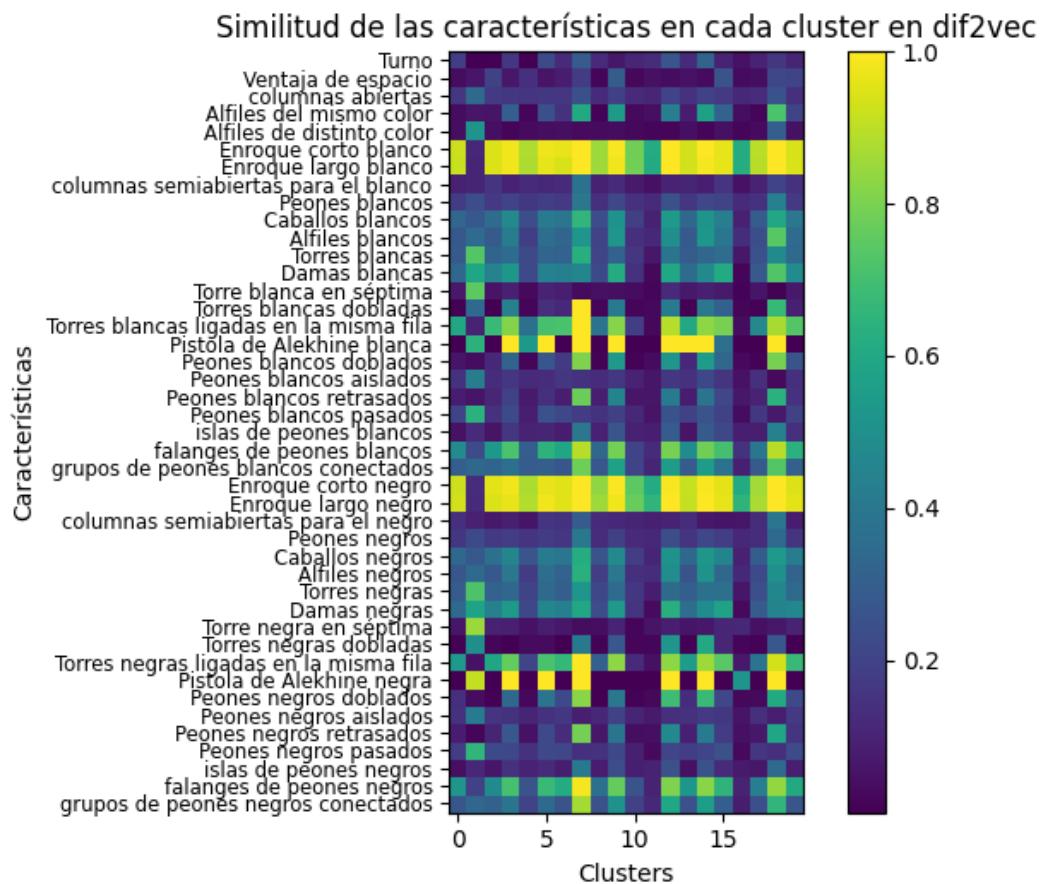


Figura 10.14: Mapa de calor de la diferenciación de etiquetas en dif2vec

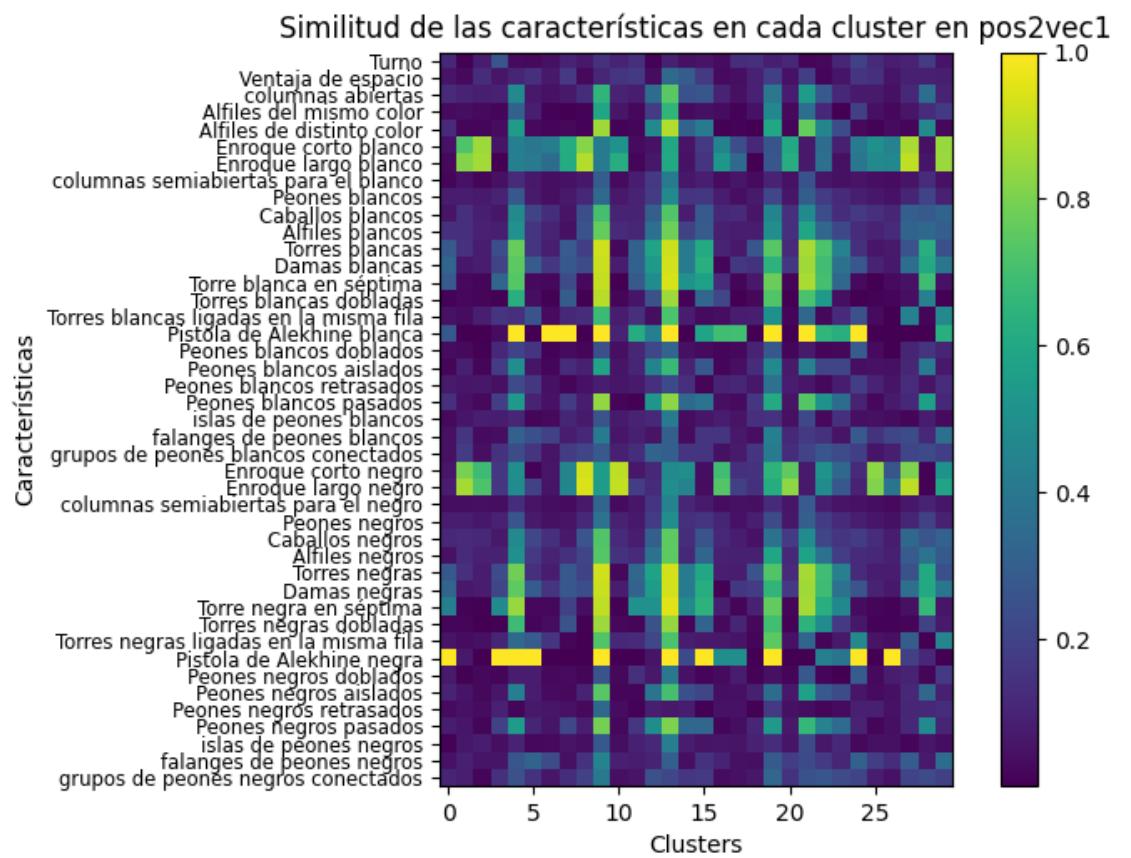


Figura 10.15: Mapa de calor de la diferenciación de etiquetas en pos2vec1

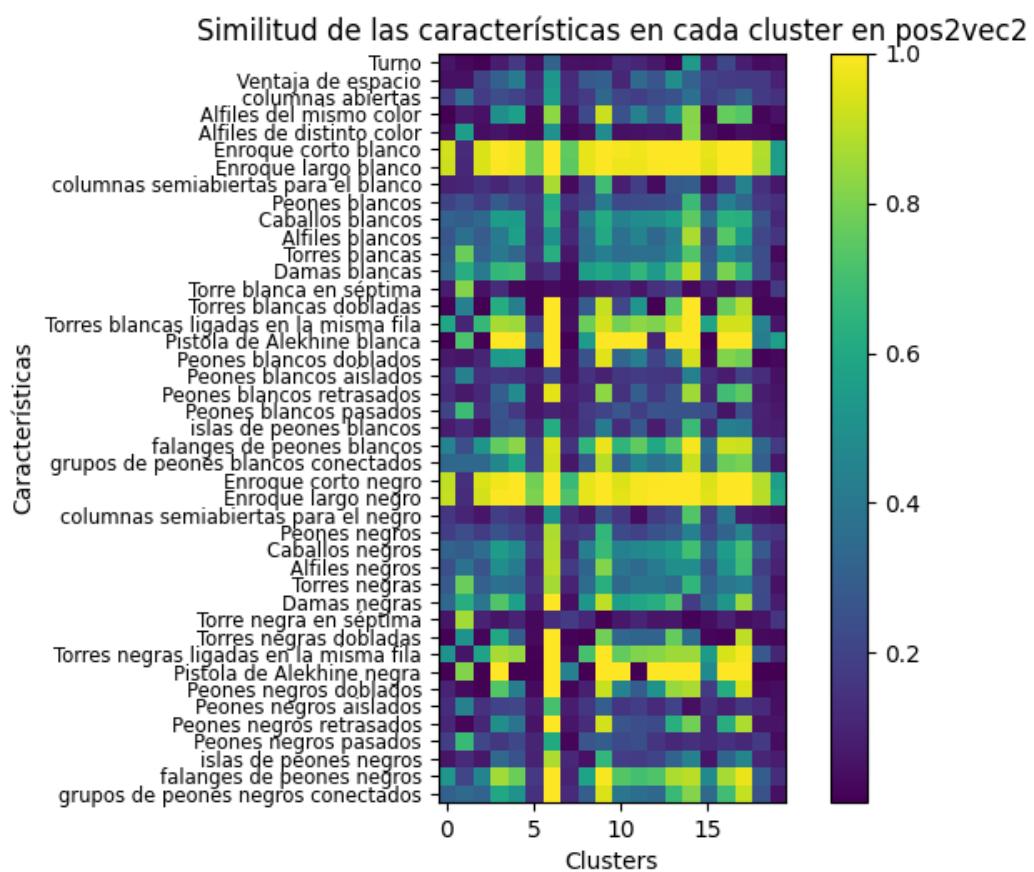


Figura 10.16: Mapa de calor de la diferenciación de etiquetas en pos2vec2



# Capítulo 11

## Conclusiones y trabajo futuro

### 11.1. Ejemplos del clustering

Veamos algunas de las posiciones de los clusters, para visualizar las similitudes que presentan.

Cuatro posiciones y movimientos del noveno cluster de dif2vec se pueden ver en ???. Las posiciones son del medio juego, y todas tienen damas y al menos una torre por bando. Se han intercambiado algunos caballos y alfiles, pero muchas de las piezas siguen en juego. Las posiciones pueden considerarse más abiertas que cerradas, pero no se sitúan en ningún extremo.

El cluster 16, en cambio, tienen pocas piezas, como se puede apreciar en la figura ???. La mayoría de sus posiciones consisten en finales, donde el riesgo de que hagan jaque mate es muy bajo y la victoria suele conseguirse coronando un peón.

Por último, en el cluster 17 ?? destaca especialmente que las posiciones son muy cerradas. Muchas piezas se mantienen en el tablero, pero ya no estamos en la apertura. En estas partidas se suele maniobrar mucho, mejorar las posiciones de las piezas antes de hacer un movimiento drástico.

### 11.2. Caso práctico

El objetivo principal de este proyecto era encontrar motivos tácticos ajedrecísticos comunes en los errores de un jugador. Para probar si esto es posible, usaremos las partidas del primer jugador considerado campeón del mundo, **William Steinitz** (1836-1900). Hemos obtenido 590 partidas suyas,

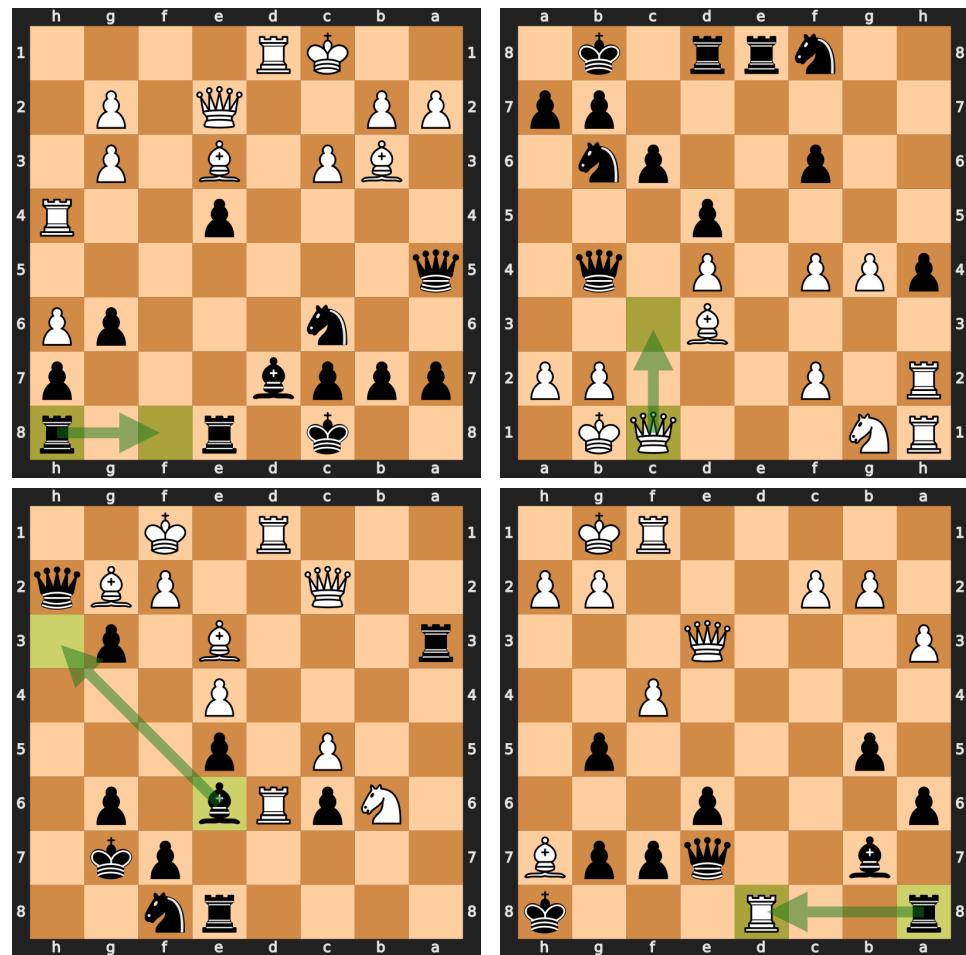


Figura 11.1: Posiciones del cluster 9 de dif2vec

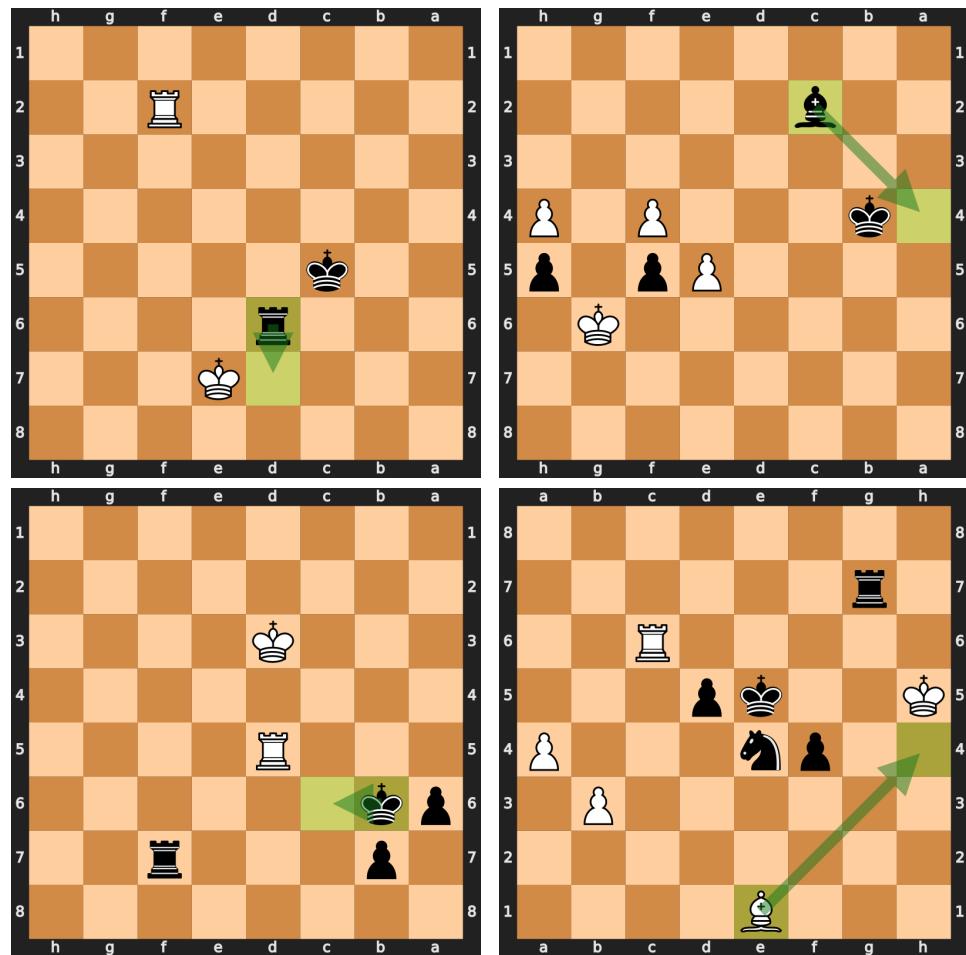


Figura 11.2: Posiciones del cluster 16 de dif2vec

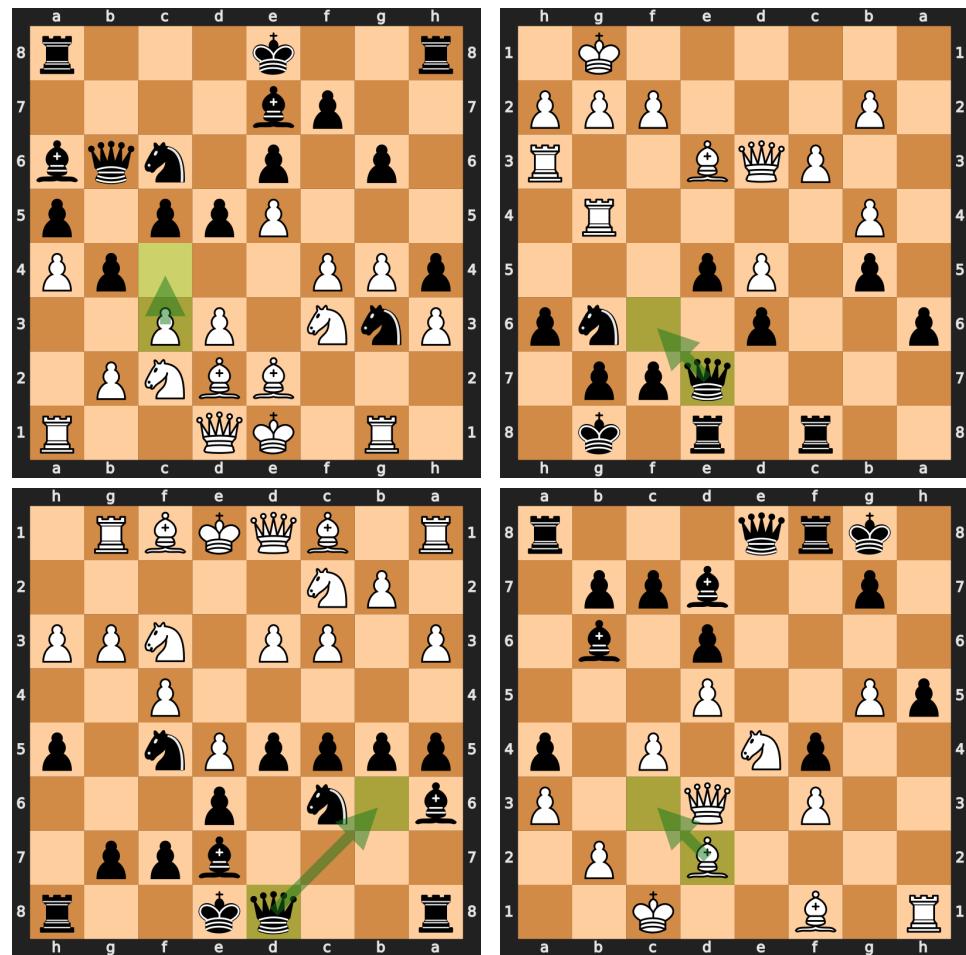


Figura 11.3: Posiciones del cluster 17 de dif2vec

Porcentaje de pertenencia de los datos a los clusters de mov2vec

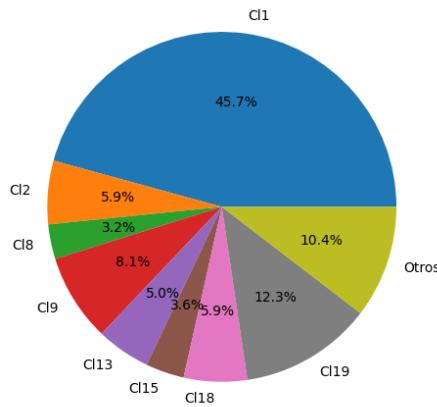


Figura 11.4: Proporción de pertenencia a los clusters de mov2vec de los errores en las partidas de William Steinitz

de todos los puntos de su carrera ajedrecística [2].

En esas partidas, se han analizado 20749 movimientos de Steinitz, y se han detectado 3020 que corresponden con nuestra definición de error. Se transforman con los cuatro tipos de modelos y se modifican sus características de la misma forma que los datos de entrenamiento. Después, se etiquetan usando los mismos centroides que en la clasificación, y obtenemos la siguiente proporción de pertenencia a los clusters:

De los datos de mov2vec 11.4, vemos que hay muchas posiciones en el cluster número 1, y una cantidad notable en el número 19. Si vemos el mapa de calor de la medida de similitudes 10.13, podemos ver que el primer cluster (cluster 0 en el mapa de calor) separa bien la cantidad de torres en el tablero, y si hay torres en séptima. En el mapa de calor de la mod 10.9, observamos que en dicho cluster es bastante probable que haya dos torres en ambos bandos, y la proporción de posiciones con torres en séptima es mayor que en otros clusters (aun así no llega a ser la moda, seguramente por la poca representación del valor). El cluster 19 (18 en los mapas de calor), al parecer no consigue diferenciar bien ninguna de las etiquetas que se han escogido. El cluster 2 y el 18, que tienen un 5% de representación, tampoco consiguen separar ninguna de las etiquetas correctamente según la medida de diferenciación.

Observemos ahora la gráfica de dif2vec 11.5. El cluster 2 (1 en los mapas

Porcentaje de pertenencia de los datos a los clusters de dif2vec

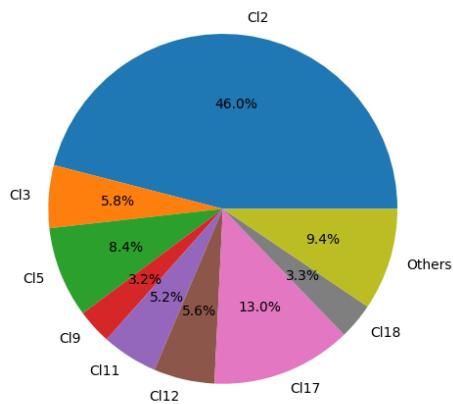


Figura 11.5: Proporción de pertenencia a los clusters de dif2vec de los errores en las partidas de William Steinitz

Porcentaje de pertenencia de los datos a los clusters de pos2vec1

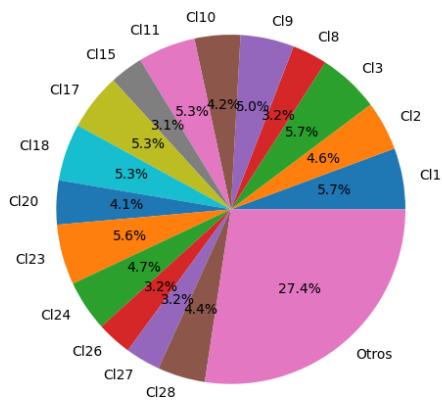


Figura 11.6: Proporción de pertenencia a los clusters de pos2vec1 de los errores en las partidas de William Steinitz

Porcentaje de pertenencia de los datos a los clusters de pos2vec2

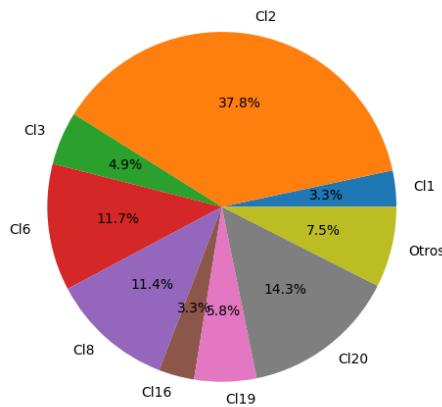


Figura 11.7: Proporción de pertenencia a los clusters de pos2vec2 de los errores en las partidas de William Steinitz

de calor) parece separar mejor que el resto de clusters el número de torres y las torres en séptima, y si hay alfiles de distinto color. También es el cluster que mejor separa si hay peones pasados en las posiciones. Más sorprendentemente, es el único que "separa mal" la posibilidad de enrocarse (seguramente derivado a que hay un desequilibrio de representación de valores). En cuanto vamos a las modas 10.9, vemos que la proporción de posiciones con dos torres blancas y negras es muy alta, incluso comparada con otros clusters. Además, la cantidad de peones más común para cada bando es 7 (lo que sucede únicamente en dicho cluster). La cantidad usual de caballos y alfiles es 2, la cantidad de damas es 1, y la probabilidad de que no haya peones pasados es extremadamente alta. Esto indica que las posiciones están en la fase de apertura, o a posiciones muy cerradas y bloqueadas donde no se capturan muchas piezas, aunque la proporción de posiciones donde el enroque aun es posible nos dirige a la primera opción. El segundo cluster con más representación, el 17, no ha conseguido separar bien ninguna de las etiquetas, así que no podemos sacar mucha información.

El clustering de pos2vec1 no nos aporta nada de información, ya que las proporciones de pertenencia a las agrupaciones son demasiado similares.

Finalmente, en pos2vec2, hay una gran representación de los clusters 2, 20, 8 y 6 (en los mapas de calor, su número es uno menos). El cluster 2 separa muy bien las torres en séptima, y también el número de peones pasados ???. En las modas 10.12, vemos que este cluster representa las posiciones de

apertura por los mismos motivos que en dif2vec. El cluster 20 no separa muy bien las etiquetas, y las modas parecen similares que el cluster anterior, sólo que las proporciones son mucho menos dominantes. Lo mismo sucede con los clusters 8 y 6.

En resumen, con un análisis rápido, podemos saber que el 50 % de los errores de Steinitz sucedieron en posiciones tempranas, donde la apertura justo había comenzado. Si uno quería prepararse para jugar contra él, una buena idea hubiese sido forzarle a jugar una apertura difícil. Si uno quería prepararse para jugar contra él, una buena idea hubiese sido forzarle a jugar una apertura difícil.

### 11.3. Conclusiones

Todos los objetivos de este proyecto se han ido cumpliendo uno a uno. Para obtener movimientos considerados errores, primero se ha explicado el funcionamiento principal de un motor de ajedrez, para después diseñar e implementar un programa que haga uso de estas potentes herramientas. Dicho programa hace uso de todo el cómputo disponible, analizando posiciones de ajedrez el tiempo necesario según los datos recogidos. Este programa ha actuado como filtro de la base de datos para obtener 500.000 errores para posteriores procesos, y ha analizado las partidas de William Steinitz para su posterior análisis.

También se han expuesto los cuatro métodos escogidos para obtener características de la red neuronal DeepChess. Después de estudiar las implementaciones disponibles en internet, se ha escogido la que funcionaba mejor, y se han derivado cuatro redes neuronales de ella que extraen las características de forma rápida y eficaz.

Finalmente, se han analizado las cuatro bases de datos que se han obtenido al transformar los errores con las cuatro redes. Después de reducir su dimensionalidad hasta un mínimo, se ha escogido un número de clusters que se ajusta a la naturaleza de los datos. Finalmente, se ha hecho uso del algoritmo k-medias para obtener las agrupaciones finales, y los centroides usados para clasificar futuros errores.

Para validar los datos, se ha hecho uso de etiquetas generadas por algoritmos creados por Adrián Rodríguez Montero. Usando algunas medidas estadísticas, se ha conseguido ver qué representa cada cluster, y la calidad de dicha representación.

Los resultados son prometedores. Se ha conseguido analizar las partidas de un gran maestro con las técnicas tan experimentales que se han usado. Varios de los métodos de extracción de características han coincidido en un buen porcentaje, pero hay una gran proporción de errores de los que no se

ha extraído información del clustering. El peor método ha sido, sin duda, pos2vec1. Sospechamos que esto es porque la red que hace la extracción nunca fue entrenada con el objetivo de diferenciar posiciones buenas y malas, por lo que no ha aprendido cuáles son las características que nos indican eso. Sus malos resultados se pueden ver especialmente en el mapa de calor de similitudes 10.15, y también la curva del error de k-medias según el número de clusters 10.3 indica que los datos eran reticentes a agruparse.

#### 11.4. Trabajo futuro

Todos y cada uno de los procesos de este trabajo son mejorables, por lo que los resultados tienen mucho espacio para cambiar. Muchos de los problemas que se han encontrado o se han solucionado de una forma poco óptima o siguen ahí, en algunos casos por ignorancia o en otros por falta de tiempo.

En primer lugar, la definición de error se puede cambiar para que la extracción de características funcione mejor. Experimentar con definir el error según la diferencia entre los dos outputs de la red neuronal DeepChess podría otorgar mejores resultados usar un motor externo, o quizás usar una combinación de ambos.

También, como se puede apreciar en el índice de acierto de las implementaciones de DeepChess con los datos que tenemos, se puede escoger un modelo que consiga un mejor análisis, como Leela Chess 0, que extrae tanto de una posición sin analizar posibles líneas que puede sugerir el siguiente movimiento con mucho acierto (aunque será complicado idear cómo extraer un vector de características).

El preprocesamiento de los datos antes del clustering se puede mejorar. No se han analizado formas de normalizar o escalar las características, que aunque sería perjudicial para mov2vec y pos2vec (consideramos que las proporciones de las características son importantes en esos datos), hubiese ayudado a los dos métodos de pos2vec al seguro tener distribuciones sesgadas. Además, los datos de estas bases de datos son absolutamente positivos, lo cual reduce el espacio métrico donde están definidos enormemente.

Sería interesante experimentar con diferentes distancias o medidas de normalización para el error de k-medias, que afecten a los valores 0 de un modo u otro. Un ejemplo posible es penalizar a las agrupaciones con características no 0 distintas, o usar una medida logarítmica, en la que la distancia entre el 0 y el 0.01 sea la misma que entre 0.01 y 1.

En el clustering en sí, está claro que hay ciertas agrupaciones que es necesario subdividirlas ya que no tienen ningún criterio según las etiquetas. Quizás sería beneficioso usar un número muy alto de clusters para que se

especialicen mucho, y después unir esos pequeños clusters de una manera diferente según cada etiqueta. Así, el cluster 2, 6 y 33 significan lo mismo para el número de torres, pero para el número de peones significan cosas diferentes.

Finalmente, se puede profundizar más en cómo extraer información de los agrupamientos. La medida de similitud que se ha creado para los mapas de calor tiene varios problemas, como la mala representación cuando hay desequilibrio de la proporción de valores en la etiqueta. También se pueden usar más etiquetas, como la mitado del tablero en la que está situado cada rey para detectar el lado del enroque, los peones situados en frente del rey o alguna medida para ver si una posición es abierta o cerrada.

Todas estas mejoras y propuestas indican vías de mejora para seguir esta línea en este problema en concreto, pero es posible que sea mejor probar estas técnicas con otro objetivo. Clasificar posiciones o movimientos de ajedrez no se ha conseguido hacer de forma satisfactoria, así que probar el método novel que hemos usado para extraer características tiene un problema: no sabemos si la calidad conseguida es el límite posible de este problema, o el límite del método. Además, este problema tiene una naturaleza un tanto extraña: queremos dividir los datos de entrenamiento lo máximo posible, pero el mejor caso posible cuando se haga una prueba con las partidas de un jugador es que haya un desequilibrio claro de las clases de sus errores.

Pero, como hemos dicho, los resultados son bastante prometedores. Los métodos de extracciones de características que se han usado son implementables en una gran cantidad de redes neuronales, y pueden ayudar a distinguir los factores que llevan a la red a hacer predicciones. Hoy en día, aun se considera que las redes neuronales son cajas negras, por lo que toda información que nos ayude a entender cómo funcionan es beneficiosa.

# Bibliografía

- [1] URL: <https://auction.catawiki.com>.
- [2] URL: <https://www.pgnmentor.com/files.html#players>.
- [3] Yuri Averbakh. *A history of chess: from Chaturanga to the present day*. SCB Distributors, 2012.
- [4] Kevin Binz. *Decision Trees In Chess*. 2015. URL: <https://kevinbinz.com/2015/02/26/decision-trees-in-chess/>.
- [5] *ChatGPT, tecnología de OpenAI*. URL: <https://openai.com/chatgpt>.
- [6] chess.com. *Chess Pieces Names, Moves & Values*. URL: <https://www.chess.com/terms/chess-pieces>.
- [7] chess.com. *Chess terms: Castling*. URL: <https://www.chess.com/terms/castling-chess>.
- [8] chess.com. *Chess Terms: Forsyth-Edwards Notation (FEN)*. URL: <https://www.chess.com/terms/fen-chess>.
- [9] chess.com. *Partidas con hándicap entre humanos y máquinas*. 2016. URL: <https://www.chess.com/article/view/man-versus-machine-historical-archive>.
- [10] chess.com. *The elusive En Passant CheckMate*. URL: [https://www.chess.com/blog/rat\\_4/the-elusive-en-passant-checkmate](https://www.chess.com/blog/rat_4/the-elusive-en-passant-checkmate).
- [11] Wikimedia Commons. URL: [https://commons.wikimedia.org/wiki/File:SCD\\_algebraic\\_notation.svg](https://commons.wikimedia.org/wiki/File:SCD_algebraic_notation.svg).
- [12] Wikimedia Commons. URL: [https://commons.wikimedia.org/wiki/File:Ajedrecista\\_primer01.JPG](https://commons.wikimedia.org/wiki/File:Ajedrecista_primer01.JPG).
- [13] Wikimedia Commons. URL: <https://commons.wikimedia.org/wiki/File:Depth-first-tree.svg>.
- [14] Omid E. David, Nathan S. Netanyahu y Lior Wolf. «DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess». En: *Artificial Neural Networks and Machine Learning – ICANN 2016*. Springer International Publishing, 2016, págs. 88-96. DOI: 10.1007/978-3-319-44781-0\_11. URL: [https://doi.org/10.1007\%2F978-3-319-44781-0\\_11](https://doi.org/10.1007\%2F978-3-319-44781-0_11).

- [15] Nathan Ensmenger. «Is chess the drosophila of artificial intelligence? A social history of an algorithm». En: *Social studies of science* 42.1 (2012), págs. 5-30.
- [16] *Estructura de la NN de Leela Chess Zero*. URL: <https://lczero.org/dev/backend/nn>.
- [17] Michael Flaxman. *Python 3's Killer Feature: asyncio*. URL: <https://eng.paxos.com/python-3s-killer-feature-asyncio>.
- [18] Frederic Friedel. *Kasparov and thirty years of computer chess*. 2015. URL: <https://en.chessbase.com/post/kasparov-and-thirty-years-of-computer-chess>.
- [19] Aditya Gupta, Christopher Grattoni y Arnav Gupta. «Determining Chess Piece Values Using Machine Learning». En: *Journal of Student Research* 12.1 (2023).
- [20] P.S. Nalwade Kalpana D. Joshi1. «Modified K-Means for Better Initial Cluster Centres». En: *International Journal of Computer Science and Mobile Computing* (), págs. 219-223.
- [21] *Leela Zero, motor de Go*. URL: <https://github.com/leela-zero/leela-zero>.
- [22] lichess.com. *Chess rating systems*. 1999. URL: <https://lichess.org/page/rating-systems>.
- [23] T Soni Madhulatha. «An overview on clustering methods». En: *arXiv preprint arXiv:1205.1117* (2012).
- [24] Nicholas Metropolis y S. Ulam. «The Monte Carlo Method». En: *Journal of the American Statistical Association* 44.247 (1949). PMID: 18139350, págs. 335-341. DOI: 10.1080/01621459.1949.10483310. URL: <https://www.tandfonline.com/doi/abs/10.1080/01621459.1949.10483310>.
- [25] T.M. Mitchell. *Machine Learning*. McGraw-Hill International Editions. McGraw-Hill, 1997. ISBN: 9780071154673. URL: <https://books.google.es/books?id=EoYBngEACAAJ>.
- [26] Adrián Rodríguez Montero. «Fundamentos del Deep Learning y desarrollo de un modelo de análisis de posiciones de ajedrez». En: (2023). URL: <https://github.com/adrianrm6/TFG>.
- [27] Yu Nasu. «Efficiently updatable neural-network-based evaluation functions for computer shogi». En: *The 28th World Computer Shogi Championship Appeal Document* 185 (2018).
- [28] *Perceptron Algorithm - A Hands On Introduction*. 2020. URL: <https://www.section.io/engineering-education/perceptron-algorithm/>.
- [29] François-André Danican Philidor. *Analyse du jeu des Échecs*. 1777.

- [30] the Python Software Foundation. *Sunsetting Python 2*. 2020. URL: <https://www.python.org/doc/sunset-python-2/>.
- [31] *Página principal de Aimchess*. URL: <https://aimchess.com/>.
- [32] *Página principal de Chess.com*. URL: <https://www.chess.com/>.
- [33] Shai Shalev-Shwartz y Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.
- [34] Claude E Shannon. «Programming a computer for playing chess». En: *first presented at the National IRE Convention, March 9, 1949, and also in Claude Elwood Shannon Collected Papers*. IEEE Press. 1993, págs. 637-656.
- [35] David Silver et al. «Mastering chess and shogi by self-play with a general reinforcement learning algorithm». En: *arXiv preprint arXiv:1712.01815* (2017).
- [36] David Silver et al. «Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm». En: *CoRR* abs/1712.01815 (2017). arXiv: 1712 . 01815. URL: <http://arxiv.org/abs/1712.01815>.
- [37] Leslie N. Smith. «Cyclical Learning Rates for Training Neural Networks». En: *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*. 2017, págs. 464-472. DOI: 10 . 1109/WACV . 2017 . 58.
- [38] *Stockfish NNUE, Chess Programming Wiki*. URL: [https://www.chessprogramming.org/Stockfish\\_NNUE](https://www.chessprogramming.org/Stockfish_NNUE).
- [39] *Stockfish, página oficial*. URL: <https://stockfishchess.org/>.
- [40] *Tensorflow, una librería open-source centrada en redes neuronales profundas*. URL: <https://www.tensorflow.org/>.



